In [1]:
```
import numpy as np
import torch
import cv2
```

In [2]:
```
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor
from torch.utils.data import DataLoader
from torch import nn
import matplotlib.pyplot as plt
```

In [3]:
```
# Get cpu, gpu or mps device for training.
device = (
    "cuda"
    if torch.cuda.is_available()
    else "mps"
    if torch.backends.mps.is_available()
    else "cpu"
)
print(f"Using {device} device")
```

```
Using cpu device
```

In [4]:
```
#You may need this to access the datasets.
import ssl
ssl._create_default_https_context = ssl._create_unverified_context
```

In [5]:
```
#This is the training data.
from torchvision.datasets import CIFAR10
train_data = datasets.CIFAR10(root="train/",
train=True,
download=True,
transform = ToTensor())
```

```
Files already downloaded and verified
```

In [6]:
```
#Test data.
test_data = datasets.CIFAR10(root="test/",
train=False,
download=True,
transform = ToTensor())
```

```
Files already downloaded and verified
```

In [7]:
```
print(train_data)
```

```
Dataset CIFAR10
    Number of datapoints: 50000
    Root location: train/
    Split: Train
    StandardTransform
Transform: ToTensor()
```

In [8]:
```
print(test_data)
```

```
Dataset CIFAR10
    Number of datapoints: 10000
    Root location: test/
    Split: Test
    StandardTransform
Transform: ToTensor()
```

In [9]:
```
dir(train_data)
```

Out[9]:
```
['__add__',
 '__annotations__',
 '__class__',
 '__class_getitem__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__len__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__orig_bases__',
 '__parameters__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__slots__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 '_check_integrity',
 '_format_transform_repr',
 '_is_protocol',
 '_load_meta',
 '_repr_indent',
 'base_folder',
 'class_to_idx',
 'classes',
 'data',
 'download',
 'extra_repr',
 'filename',
 'meta',
 'root',
 'target_transform',
 'targets',
 'test_list',
 'tgz_md5',
 'train',
 'train_list',
 'transform',
 'transforms',
 'url']
```

In [10]:
```
train_data.class_to_idx
```

Out[10]:
```
{'airplane': 0,
 'automobile': 1,
 'bird': 2,
 'cat': 3,
 'deer': 4,
 'dog': 5,
 'frog': 6,
 'horse': 7,
 'ship': 8,
 'truck': 9}
```

In [11]:
```
test_data.class_to_idx
```

Out[11]:
```
{'airplane': 0,
 'automobile': 1,
 'bird': 2,
 'cat': 3,
 'deer': 4,
 'dog': 5,
 'frog': 6,
 'horse': 7,
 'ship': 8,
 'truck': 9}
```

In [12]:
```
batch_size = 64

# Create data loaders.
train_dataloader = DataLoader(train_data, batch_size=batch_size)
test_dataloader = DataLoader(test_data, batch_size=batch_size)

for X, y in train_dataloader:
    print(f"Shape of X [N, C, H, W]: {X.shape}")
    print(f"Shape of y: {y.shape} {y.dtype}")
    break
```

```
Shape of X [N, C, H, W]: torch.Size([64, 3, 32, 32])
Shape of y: torch.Size([64]) torch.int64
```

In [13]:
```
def label_array(dataloader):
    # initialize the empty list
    cifar10_test_labels = []

    # read data from dataloader
    for inputs, labels in dataloader:
        # add labels to list cifar10_test_labels
        cifar10_test_labels.append(labels)

    # convert cifar10_test_labels to tensor
    cifar10_test_labels = torch.cat(cifar10_test_labels,dim=0)

    return cifar10_test_labels
```

In [14]:
```
# check function label_array
label_array(test_dataloader)
```

Out[14]:
```
tensor([3, 8, 8,  ..., 5, 1, 7])
```

In [15]:
```
label_array(test_dataloader).shape
```

Out[15]:
```
torch.Size([10000])
```

In [16]:
```
class CIFARConvNet(nn.Module):
    def __init__(self):
        super(CIFARConvNet, self).__init__()
        #the convolution layers
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding='sa
            nn.ReLU()
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5, padding='s
            nn.ReLU()
        )
        #the MaxPool layers
        self.pool = nn.MaxPool2d(kernel_size=2)
        nn.Dropout(0.25)

        #the standard layers
        self.fc1 = nn.Sequential(
            nn.Flatten(),
            nn.Linear(4096, 256),
            nn.ReLU(),
            nn.Dropout(0.4)
        )
        self.fc2 = nn.Sequential(
            nn.Linear(256,10)
        )

    def forward(self, x):
        x = self.conv1(x)
        x = self.pool(x)
        x = self.conv2(x)
        x = self.pool(x)
        x = self.fc1(x)
        x = self.fc2(x)

        return x
```

In [148…
```
# declare variable to store training history
train_his = {
    "train_loss": [],
    "train_acc": [],
    "val_loss": [],
    "val_acc": []
}
# split train_data into train_subset and validation_subset
train_subset, val_subset = torch.utils.data.random_split(
        train_data, [40000, 10000], generator=torch.Generator().manual_seed(1))

# create dataloader for train_subset and validation_subset
train_dataloader = DataLoader(train_subset, batch_size=batch_size)
val_dataloader = DataLoader(val_subset, batch_size=batch_size)

# calculate steps per epoch for training set
```

```
train_steps = len(train_dataloader.dataset) // batch_size
val_steps = len(val_dataloader.dataset) // batch_size
```

In [159…

```
import time
lr = 1e-3
num_epochs = 60
# initialize model
model = CIFARConvNet().to(device)
# select loss function
loss_fn = nn.CrossEntropyLoss()
# select optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=lr)

# measure how long training is going to take
print("[INFO] training the network...")
startTime = time.time()

for epochs in range(num_epochs):
    # set the model in training mode
    model.train()
    # initialize the total training and validation loss
    total_train_loss = 0
    total_val_loss = 0
    # initialize the number of correct predictions in the training step
    train_accuracy = 0
    val_accuracy = 0
    # loop over the training set
    for (x, y) in train_dataloader:
        # send the input to the device
        (x, y) = (x.to(device), y.to(device))
        # perform a forward pass and calculate the training loss
        pred = model(x)
        loss = loss_fn(pred, y)
        # zero out the gradients, perform the backpropagation step, and update t
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        # add the loss to the total training loss so far and calculate the numbe
        total_train_loss += loss
        train_accuracy += (pred.argmax(1) == y).type(torch.float).sum().item()

    # switch off autograd for evaluation
    with torch.no_grad():
        # set the model in evaluation mode
        model.eval()
        # loop over the validation set
        for (x, y) in val_dataloader:
            # send the input to the device
            (x, y) = (x.to(device), y.to(device))
            # make the predictions and calculate the validation loss
            pred = model(x)
            total_val_loss += loss_fn(pred, y)
            # calculate the number of correct predictions
            val_accuracy += (pred.argmax(1) == y).type(torch.float).sum().item()

    # calculate the average training loss and validation loss
    avg_train_loss = total_train_loss / train_steps
    avg_val_loss = total_val_loss / val_steps

    # calculate the training accuracy
```

```
    train_accuracy = train_accuracy / len(train_dataloader.dataset)
    val_accuracy = val_accuracy / len(val_dataloader.dataset)

    # update our training history
    train_his["train_loss"].append(avg_train_loss.cpu().detach().numpy())
    train_his["train_acc"].append(train_accuracy)
    train_his["val_loss"].append(avg_val_loss.cpu().detach().numpy())
    train_his["val_acc"].append(val_accuracy)

    # print the model training information
    print("    EPOCH: {}/{}".format(epochs + 1, num_epochs))
    print("       Train loss: {:.6f}, Train accuracy: {:.4f}".format(avg_train_l
    print("       Validation loss: {:.6f}, Validation accuracy: {:.4f}".format(a


# finish measuring how long training took
endTime = time.time()
print("[INFO] total time taken to train the model: {:.2f}s".format(endTime - sta
```

```
[INFO] training the network...
    EPOCH: 1/60
       Train loss: 1.585631, Train accuracy: 0.4235
       Validation loss: 1.295445, Validation accuracy: 0.5347
    EPOCH: 2/60
       Train loss: 1.240099, Train accuracy: 0.5583
       Validation loss: 1.102593, Validation accuracy: 0.6119
    EPOCH: 3/60
       Train loss: 1.081914, Train accuracy: 0.6166
       Validation loss: 1.014575, Validation accuracy: 0.6363
    EPOCH: 4/60
       Train loss: 0.984198, Train accuracy: 0.6538
       Validation loss: 1.001759, Validation accuracy: 0.6458
    EPOCH: 5/60
       Train loss: 0.903931, Train accuracy: 0.6805
       Validation loss: 0.959133, Validation accuracy: 0.6648
    EPOCH: 6/60
       Train loss: 0.841330, Train accuracy: 0.7029
       Validation loss: 0.952182, Validation accuracy: 0.6652
    EPOCH: 7/60
       Train loss: 0.786465, Train accuracy: 0.7238
       Validation loss: 0.898338, Validation accuracy: 0.6879
    EPOCH: 8/60
       Train loss: 0.724433, Train accuracy: 0.7420
       Validation loss: 0.921266, Validation accuracy: 0.6811
    EPOCH: 9/60
       Train loss: 0.678609, Train accuracy: 0.7597
       Validation loss: 0.889362, Validation accuracy: 0.6898
    EPOCH: 10/60
       Train loss: 0.635303, Train accuracy: 0.7738
       Validation loss: 0.907758, Validation accuracy: 0.6935
    EPOCH: 11/60
       Train loss: 0.595771, Train accuracy: 0.7856
       Validation loss: 0.913831, Validation accuracy: 0.6873
    EPOCH: 12/60
       Train loss: 0.564334, Train accuracy: 0.7983
       Validation loss: 0.935990, Validation accuracy: 0.6939
    EPOCH: 13/60
       Train loss: 0.529436, Train accuracy: 0.8104
       Validation loss: 0.981988, Validation accuracy: 0.6916
    EPOCH: 14/60
       Train loss: 0.498014, Train accuracy: 0.8206
       Validation loss: 0.982573, Validation accuracy: 0.6915
    EPOCH: 15/60
       Train loss: 0.465997, Train accuracy: 0.8311
       Validation loss: 1.044847, Validation accuracy: 0.6920
    EPOCH: 16/60
       Train loss: 0.429090, Train accuracy: 0.8447
       Validation loss: 1.082340, Validation accuracy: 0.6952
    EPOCH: 17/60
       Train loss: 0.412586, Train accuracy: 0.8483
       Validation loss: 1.088463, Validation accuracy: 0.6861
    EPOCH: 18/60
       Train loss: 0.381381, Train accuracy: 0.8599
       Validation loss: 1.142104, Validation accuracy: 0.6900
    EPOCH: 19/60
       Train loss: 0.374458, Train accuracy: 0.8629
       Validation loss: 1.114530, Validation accuracy: 0.6952
    EPOCH: 20/60
       Train loss: 0.350023, Train accuracy: 0.8735
```

```
    Validation loss: 1.198752, Validation accuracy: 0.6846
EPOCH: 21/60
    Train loss: 0.340961, Train accuracy: 0.8742
    Validation loss: 1.254526, Validation accuracy: 0.6850
EPOCH: 22/60
    Train loss: 0.317285, Train accuracy: 0.8834
    Validation loss: 1.321228, Validation accuracy: 0.6758
EPOCH: 23/60
    Train loss: 0.305987, Train accuracy: 0.8886
    Validation loss: 1.305097, Validation accuracy: 0.6871
EPOCH: 24/60
    Train loss: 0.282666, Train accuracy: 0.8949
    Validation loss: 1.324823, Validation accuracy: 0.6830
EPOCH: 25/60
    Train loss: 0.285432, Train accuracy: 0.8963
    Validation loss: 1.413260, Validation accuracy: 0.6815
EPOCH: 26/60
    Train loss: 0.274008, Train accuracy: 0.8981
    Validation loss: 1.441490, Validation accuracy: 0.6810
EPOCH: 27/60
    Train loss: 0.262669, Train accuracy: 0.9039
    Validation loss: 1.424560, Validation accuracy: 0.6858
EPOCH: 28/60
    Train loss: 0.251488, Train accuracy: 0.9083
    Validation loss: 1.477663, Validation accuracy: 0.6867
EPOCH: 29/60
    Train loss: 0.243733, Train accuracy: 0.9092
    Validation loss: 1.501709, Validation accuracy: 0.6883
EPOCH: 30/60
    Train loss: 0.239148, Train accuracy: 0.9133
    Validation loss: 1.486093, Validation accuracy: 0.6926
EPOCH: 31/60
    Train loss: 0.219644, Train accuracy: 0.9192
    Validation loss: 1.579766, Validation accuracy: 0.6856
EPOCH: 32/60
    Train loss: 0.217019, Train accuracy: 0.9211
    Validation loss: 1.509071, Validation accuracy: 0.6959
EPOCH: 33/60
    Train loss: 0.216126, Train accuracy: 0.9196
    Validation loss: 1.567184, Validation accuracy: 0.6887
EPOCH: 34/60
    Train loss: 0.208557, Train accuracy: 0.9246
    Validation loss: 1.655564, Validation accuracy: 0.6891
EPOCH: 35/60
    Train loss: 0.201002, Train accuracy: 0.9269
    Validation loss: 1.628972, Validation accuracy: 0.6922
EPOCH: 36/60
    Train loss: 0.201495, Train accuracy: 0.9270
    Validation loss: 1.661271, Validation accuracy: 0.6884
EPOCH: 37/60
    Train loss: 0.200366, Train accuracy: 0.9270
    Validation loss: 1.659268, Validation accuracy: 0.6832
EPOCH: 38/60
    Train loss: 0.191297, Train accuracy: 0.9311
    Validation loss: 1.627513, Validation accuracy: 0.6899
EPOCH: 39/60
    Train loss: 0.179723, Train accuracy: 0.9335
    Validation loss: 1.706143, Validation accuracy: 0.6863
EPOCH: 40/60
    Train loss: 0.177106, Train accuracy: 0.9358
```

```
    Validation loss: 1.770249, Validation accuracy: 0.6869
EPOCH: 41/60
    Train loss: 0.181447, Train accuracy: 0.9335
    Validation loss: 1.719392, Validation accuracy: 0.6875
EPOCH: 42/60
    Train loss: 0.177426, Train accuracy: 0.9358
    Validation loss: 1.767121, Validation accuracy: 0.6899
EPOCH: 43/60
    Train loss: 0.169567, Train accuracy: 0.9395
    Validation loss: 1.877388, Validation accuracy: 0.6889
EPOCH: 44/60
    Train loss: 0.163309, Train accuracy: 0.9399
    Validation loss: 1.823595, Validation accuracy: 0.6879
EPOCH: 45/60
    Train loss: 0.168265, Train accuracy: 0.9389
    Validation loss: 1.905448, Validation accuracy: 0.6838
EPOCH: 46/60
    Train loss: 0.166925, Train accuracy: 0.9393
    Validation loss: 1.871604, Validation accuracy: 0.6821
EPOCH: 47/60
    Train loss: 0.171206, Train accuracy: 0.9378
    Validation loss: 1.880969, Validation accuracy: 0.6878
EPOCH: 48/60
    Train loss: 0.158327, Train accuracy: 0.9422
    Validation loss: 1.915547, Validation accuracy: 0.6870
EPOCH: 49/60
    Train loss: 0.157527, Train accuracy: 0.9427
    Validation loss: 1.867690, Validation accuracy: 0.6864
EPOCH: 50/60
    Train loss: 0.157031, Train accuracy: 0.9434
    Validation loss: 1.969535, Validation accuracy: 0.6922
EPOCH: 51/60
    Train loss: 0.148118, Train accuracy: 0.9478
    Validation loss: 1.985334, Validation accuracy: 0.6918
EPOCH: 52/60
    Train loss: 0.149415, Train accuracy: 0.9441
    Validation loss: 1.946991, Validation accuracy: 0.6933
EPOCH: 53/60
    Train loss: 0.142246, Train accuracy: 0.9475
    Validation loss: 1.949271, Validation accuracy: 0.6886
EPOCH: 54/60
    Train loss: 0.144950, Train accuracy: 0.9482
    Validation loss: 1.992840, Validation accuracy: 0.6889
EPOCH: 55/60
    Train loss: 0.147576, Train accuracy: 0.9472
    Validation loss: 2.061358, Validation accuracy: 0.6958
EPOCH: 56/60
    Train loss: 0.145506, Train accuracy: 0.9487
    Validation loss: 1.994285, Validation accuracy: 0.6895
EPOCH: 57/60
    Train loss: 0.141268, Train accuracy: 0.9499
    Validation loss: 1.984730, Validation accuracy: 0.6892
EPOCH: 58/60
    Train loss: 0.139257, Train accuracy: 0.9507
    Validation loss: 2.011512, Validation accuracy: 0.6935
EPOCH: 59/60
    Train loss: 0.140522, Train accuracy: 0.9498
    Validation loss: 1.989849, Validation accuracy: 0.6929
EPOCH: 60/60
    Train loss: 0.134466, Train accuracy: 0.9516
```

```
      Validation loss: 2.142626, Validation accuracy: 0.6915
[INFO] total time taken to train the model: 2701.16s
```

In [158…

```
# this variable store the result of training with 400+ epochs
len(train_his_400plus["val_acc"])
```

Out[158…

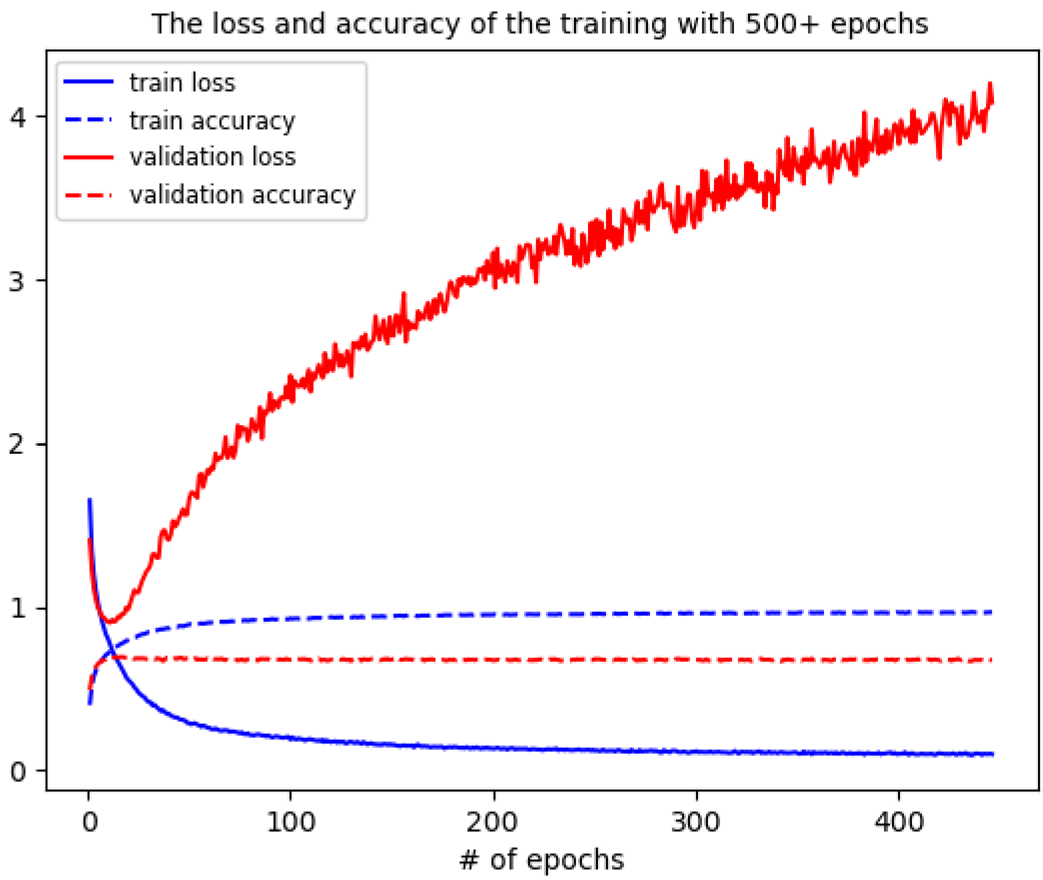
```
446
```

In [142…

```
# function for plotting the loss and accuracy
def plot_train_loss(loss_dict, txt):
    # create data for y-axis
    y1=np.array(loss_dict["train_loss"])
    y2=np.array(loss_dict["train_acc"])
    y3=np.array(loss_dict["val_loss"])
    y4=np.array(loss_dict["val_acc"])
    # create data for x-axis
    X = np.linspace(1, len(y1), num=len(y1)).reshape(-1,1)
    # plotting
    plt.xlabel('# of epochs')
    plt.plot(X, y1, 'b-')
    plt.plot(X, y2, 'b--')
    plt.plot(X, y3, 'r-')
    plt.plot(X, y4, 'r--')
    plt.legend(['train loss','train accuracy','validation loss','validation accu
    plt.title(txt, fontsize=10)
```

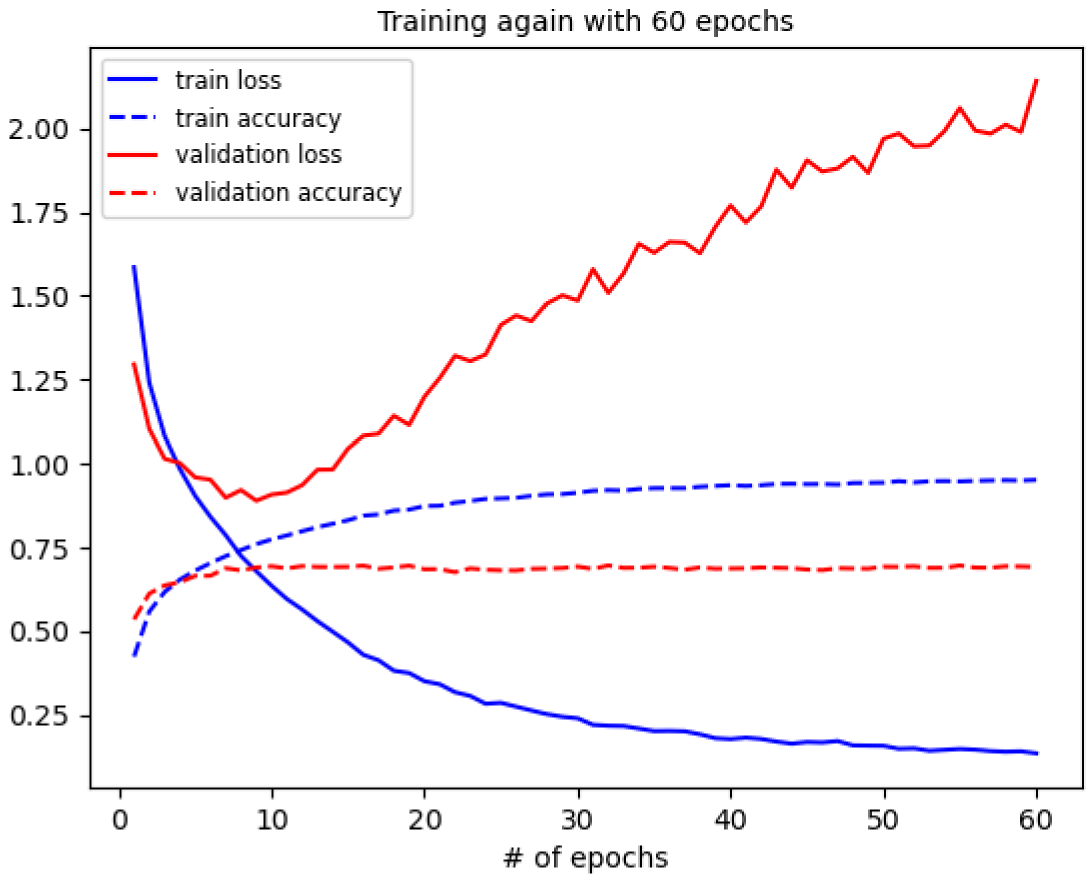
In [157…

```
plot_train_loss(train_his_400plus, "The loss and accuracy of the training with 5
```



In [160…

```
plot_train_loss(train_his, "Training again with 60 epochs")
```



In [161…

```
# save trained model to local directory
torch.save(model.state_dict(), "cifar_convnet1")
```

In [162…

```
# load the model
model1 = CIFARConvNet().to(device)
model1.load_state_dict(torch.load("cifar_convnet1", weights_only=True))
```

Out[162…

```
<All keys matched successfully>
```

In [163…

```
def get_prediction(dataloader, model):
    pred = []
    # set model to prediction state
    model.eval()
    # loop through the dataloader
    for inputs, labels in dataloader:
        with torch.no_grad():
            # send the input to the device
            inputs = inputs.to(device)
            # make the predictions
            outputs = model(inputs)
            # identify the predicted class
            predict = outputs.max(dim=1)
            # add them to the list
            pred.append(predict.indices)

    # convert list to tensor
    pred = torch.cat(pred,dim=0)
    return pred
```

In [164…

```
def calculate_accuracy(test_label, predicted):
    # calculate the accuracy
```

```
    acc = sum(test_label==predicted).item()/len(test_label)
    return acc
```

In [165…

```
def confusion_dict(test_label, predicted):
    conf_dict = {}
    # convert tensor to numpy
    test_np = test_label.numpy()
    predicted_np = predicted.numpy()

    for key in range(10):
        # filter the incorrect prediction in predicted list
        error_label = predicted_np[(test_np==key) & (predicted_np!=key)]
        # calculate the frequency of each incorrect label
        unique, counts = np.unique(error_label, return_counts=True)
        # add the top 2 incorrect predictions into conf_dict
        if len(counts) < 2:
            conf_dict[key] = tuple(unique)
        else:
            conf_dict[key] = tuple(unique[np.argsort(counts)[len(counts)-2:]])

    return conf_dict
```

In [166…

```
# predict
predicted = get_prediction(test_dataloader, model1)
```

In [167…

```
# get test label
test_label = label_array(test_dataloader)
```

In [168…

```
# accuracy
calculate_accuracy(test_label,predicted)
```

Out[168…

```
0.6897
```

In [169…

```
# confusion dictionary
conf_dict = confusion_dict(test_label,predicted)
conf_dict
```

Out[169…

```
{0: (2, 8),
 1: (8, 9),
 2: (4, 5),
 3: (6, 5),
 4: (2, 7),
 5: (2, 3),
 6: (2, 3),
 7: (4, 5),
 8: (1, 0),
 9: (0, 1)}
```

In [170…

```
# function for giving comments based on the confusion dictionary
def comment(conf_dict):
    # get class based on index
    def get_class(index):
        return list(test_data.class_to_idx.keys())[list(test_data.class_to_idx.v

    # plot first incorrect prediction
    def plot_top_diff(test_label, predicted, txt, key1, key2, key3=-1):
        test_np = test_label.numpy()
        predicted_np = predicted.numpy()
        # get index of images with label key1
```

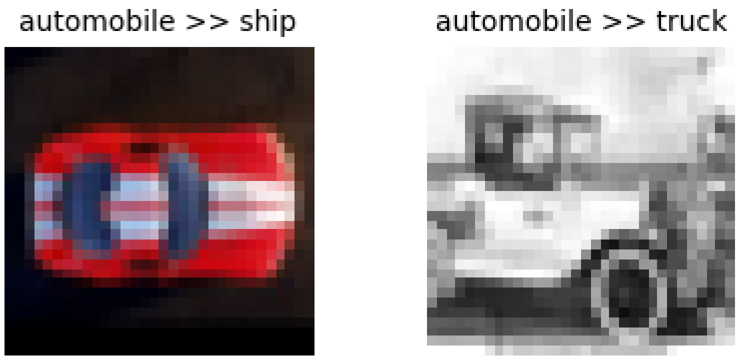
```
        key1_idx = np.argwhere(test_np==key1)
        # filter prediction of image with label key1
        key1_predict = predicted_np[test_np==key1]
        # get index of prediction: key1 to key2
        key2_idx = np.argwhere(key1_predict==key2)
        # plot first incorrect predicted image: key1 to key2
        plt.figure(figsize=(5, 2))
        plt.figtext(0.5, -0.1, txt, wrap=True, horizontalalignment='center', for
        plt.subplot(121); plt.imshow(test_data.data[key1_idx[key2_idx[0][0]][0]]
        plt.title("{} >> {}".format(get_class(key1),get_class(key2)), fontsize=1
        if key3 > -1:
            # get index of prediction: key1 to key3
            key3_idx = np.argwhere(key1_predict==key3)
            # plot first incorrect predicted image: key1 to key3
            plt.subplot(122); plt.imshow(test_data.data[key1_idx[key3_idx[0][0]]
            plt.title("{} >> {}".format(get_class(key1),get_class(key3)), fontsi

    # print comment
    for key in conf_dict.keys():
        no_of_err = 1 if type(conf_dict[key]) is int else len(conf_dict[key])
        if no_of_err==0:
            print("{} are classified properly.\n".format(get_class(key)))
        elif no_of_err==1:
            err = 0 if type(conf_dict[key]) is int else conf_dict[key][0]
            txt = "{} are confused with {}.\n".format(get_class(key),
                                                      get_class(err))
            plot_top_diff(test_label, predicted, txt, key, err)
        else:
            txt = "{} are confused with {} and {}.\n".format(get_class(key),
                                                             get_class(conf_dict
                                                             get_class(conf_dict
            plot_top_diff(test_label, predicted, txt, key, conf_dict[key][0], cc
```

In [171…

```
# comment on the errors
comment(conf_dict)
```

airplane are confused with bird and ship.



automobile are confused with ship and truck.

bird are confused with deer and dog.

cat are confused with frog and dog.

deer are confused with bird and horse.

dog >> bird

dog >> cat

dog are confused with bird and cat.

frog >> bird

frog >> cat

frog are confused with bird and cat.

horse >> deer

horse >> dog

horse are confused with deer and dog.

ship >> automobile

ship >> airplane

ship are confused with automobile and airplane.

truck are confused with airplane and automobile.