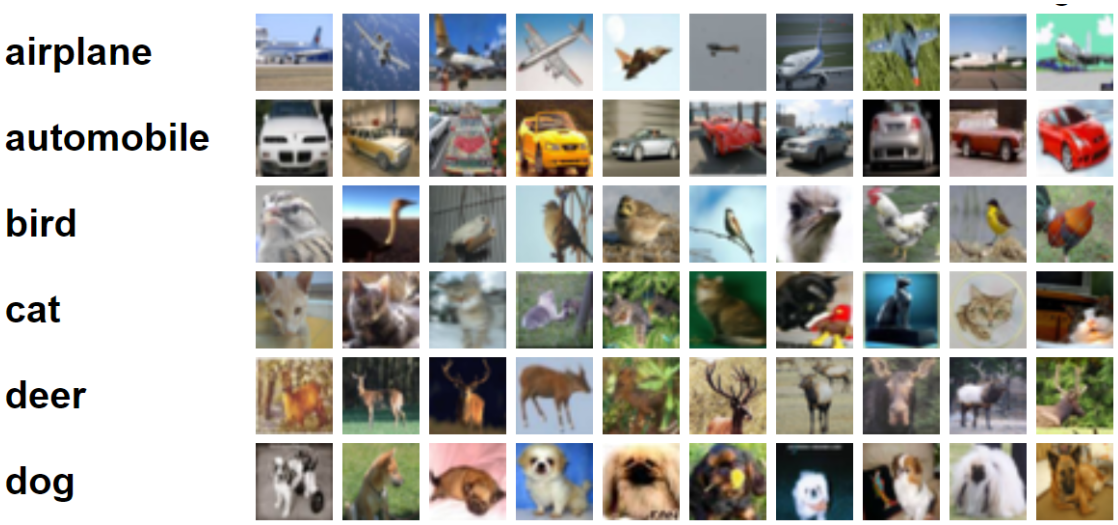
Implementation Documentation:

Dataset:
CIFAR-10 is a widely used dataset in machine learning and computer vision research. This dataset consists of 60,000 color images with dimensions of 32 by 32 pixels, divided into 10 categories with 6,000 images in each category. The dataset contains images of various objects from different categories such as airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks.

The dataset is divided into 50,000 training images and 10,000 testing images. The training set is used to train machine learning models, while the test set is used to evaluate the performance of the trained models.

CIFAR-10 is often used as a benchmark dataset for image classification tasks. Many advanced models have been trained on this dataset, and it has been used to evaluate the performance of various deep learning architectures.

Obtaining the CIFAR-10 dataset and performing the necessary preprocessing:



Section 1 - Importing Required Libraries:

```
#importig necessary libraries

import torch
from torch import nn
import torch.optim as optim
from torchvision import datasets, transforms
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler
from torch.utils.data import DataLoader, random_split, SubsetRandomSampler
import torch.nn.functional as F

import numpy as np
import matplotlib.pyplot as plt
from random import randrange
from skimage.util import random_noise
from tqdm.notebook import tqdm
```

We import the necessary libraries for building and training a neural network using PyTorch.

Brief description of each imported library:

- Torch: The main library used for building and training neural networks.

- nn: A sub-library that provides implementations of various neural network architectures and loss functions.

- Optim: A subset of PyTorch that provides various optimization algorithms for training neural networks.

- Datasets: A sub-library that provides pre-built datasets for training and testing neural networks. We obtain the CIFAR-10 dataset using this library.

- Transforms: A sub-library that provides various image transformations such as resizing and normalization.

- SubsetRandomSampler: A PyTorch class that randomly samples elements from a given dataset.

- DataLoader: A PyTorch class that loads data from a dataset and provides batches of data to a neural network.

- F: A sub-library of PyTorch that provides various activation functions and other useful functions.

- Numpy: A popular and useful library for performing fast and parallel numerical computations.

- Matplotlib: A library for plotting graphs.

- Random: A library for generating random numbers.

- Random_noise: A function from the Skimage.util module that adds random noise to an image.

-Tqdm: A library for creating progress bars in Python.

Section 2 - Performing Necessary Preprocessing on Dataset Images:

```
# standard cast into Tensor and pixel values normalization in [-1, 1] range
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

train_data=datasets.CIFAR10('data',train=True,download=True,transform=transform)
test_data=datasets.CIFAR10('data',train=False,download=True,transform=transform)
```

Using this code snippet, we perform the following preprocessing steps on the input images:

First, to input the images into neural networks and provide parallel processing capability, we convert the input images into tensors.

Then, we normalize the input image data such that after normalization, the mean and variance of the pixels in each channel become 0.5. This is done to ensure that the pixel values are within the range [-1, 1].

Next, we use the Datasets library to obtain the CIFAR-10 dataset in the form of training and evaluation data.

Then, using the Datasets library, we obtain the CIFAR-10 dataset in the form of training and testing data.

Section 3 - Loading Images and Separating Training, Validation, and Testing Data:

```
num_workers=1 # number of subprocesses to use for data loading
batch_size=50
valid_size=0.2 # percentage of training set to use as validation

train_length = len(train_data)
indices=list(range(train_length)) # obtain training indices that will be used for validation
split = int(np.floor(valid_size * train_length))

np.random.shuffle(indices) # randomly shuffle data indeces

train_idx, valid_idx = indices[split:], indices[:split]

# define samplers for obtaining training and validation batches
train_sampler=SubsetRandomSampler(train_idx)
validation_sampler=SubsetRandomSampler(valid_idx)

# define data loaders
train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

train_loader=DataLoader(train_data,num_workers=num_workers,batch_size=batch_size,sampler=train_sampler)
validation_loader=DataLoader(train_data,num_workers=num_workers,batch_size=batch_size,sampler=validation_sa
mpler)
test_loader=DataLoader(test_data,shuffle=True,num_workers=num_workers,batch_size=batch_size)
```

Firstly, the data is read from the dataset using a loader or a process with a batch size of 50, with 20% of it being set aside as validation data.
The training and validation data are then separated after shuffling, which balances the training and validation data.

Using SubsetRandomSampler, we randomly select samples from the training and validation data and put them into the train_sampler and valid_sampler variables. Finally, using DataLoader, we prepare the training and validation data to be fed into the training process, and after shuffling, we prepare the test data to be fed into the evaluation process.

Section 4 - Displaying Images from the Dataset:

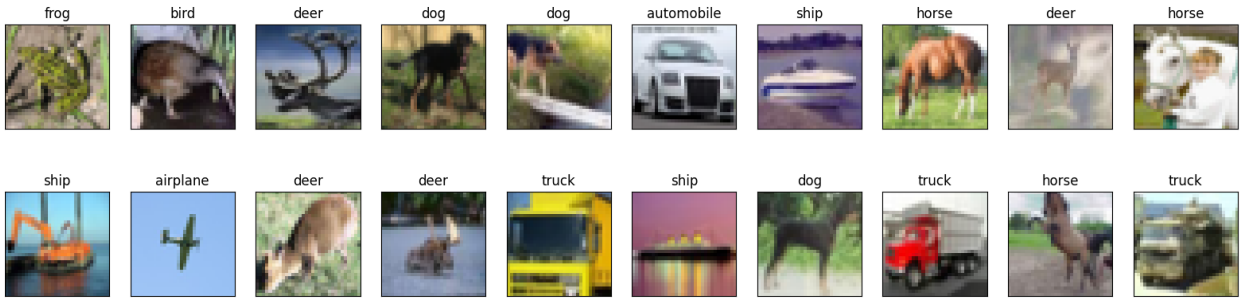
```
image_dataiter = iter(train_loader)
data = next(image_dataiter)
normal_imgs, normal_labels = data

classes=['airplane', 'automobile', 'bird', 'cat', 'deer',
          'dog', 'frog', 'horse', 'ship', 'truck']
def RGBshow(img):
    img=img*0.5+0.5
    plt.imshow(np.transpose(img,(1,2,0)))

fig=plt.figure(1,figsize=(20,5))
for idx in range(20):
    ax=fig.add_subplot(2,10,idx+1,xticks=[],yticks=[])
    RGBshow(normal_imgs[idx])
    ax.set_title(classes[normal_labels[idx]])
```

Using this code snippet and the Matplotlib library, we display images from the dataset along with their labels.

Examples of images in the CIFAR-10 dataset:



1 - For this part, you should implement a Convolutional Neural Network on the CIFAR10 data set and report and visualize the Accuracy and Loss values by performing the Train, Validation, Test steps.

Section 1 - Implementing a Convolutional Neural Network:

```python
# define the CNN architecture
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        # convolutional layer
        self.conv1 = nn.Conv2d(3, 16, 5)
        self.conv2 = nn.Conv2d(16, 32, 5)
        self.conv3 = nn.Conv2d(32, 64, 5)
        self.conv4 = nn.Conv2d(64, 128, 5)

        # max pooling layer
        self.pool = nn.MaxPool2d(2, 2)

        # fully connected layers
        self.fc1 = nn.Linear(128 * 2 * 2, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        # add sequence of convolutional and max pooling layers
        x = self.pool(F.relu(self.conv1(x)))

        x = self.conv2(x)
        x = nn.functional.relu(x)

        x = self.conv3(x)
        x = nn.functional.relu(x)

        x = self.conv4(x)
        x = nn.functional.relu(x)

        # flattening
        x = x.view(-1, 128 * 2 * 2)

        # fully connected layers
        x = self.fc1(x)
        x = nn.functional.relu(x)

        x = self.fc2(x)
        x = nn.functional.relu(x)

        x = self.fc3(x)

        return x


model = CNN()
model.cuda()

print(model)
```

The architecture of the convolutional neural network is as follows:

A convolutional neural network (CNN) is defined for image classification problems. This CNN consists of four convolutional layers, each followed by a ReLU activation function and a max pooling layer. The convolutional layers have 16, 32, 64, and 128 filters with a kernel size of 5x5, respectively. The max pooling layer has a kernel size of 2x2 and a stride of 2.

After the convolutional and max pooling layers, the output is flattened and reshaped to be fed into the fully connected layers with ReLU activation functions. The first fully connected layer has 256 neurons, the second has 128 neurons, and the final layer has 10 neurons, which corresponds to the number of classes for image classification.

In the forward section, the input images pass through the convolutional and max pooling layers in order and then reach the fully connected layers. After applying all these layers on the image in the CNN, the output is determined in the final layer.

2- Implementation of Loss Function and Optimizer:

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

The loss function used for the training process is CrossEntropy, and the optimizer used is SGD. The learning rate used for the optimization process is 0.01

.

3- Implementation of Model Training and Validation:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# number of epochs to train the model
n_epochs = 30

#List to store loss to visualize
valid_loss_min = np.Inf # to save the best model with lower validation loss while
training process
train_losslist = []
train_acc_list = []
valid_acc_list = []
history = {'train_loss': [], 'valid_loss': [], 'train_acc': [], 'valid_acc': []}

for epoch in range(n_epochs):
    print(f'Epoch {epoch+1} ...')
    # keep track of training and validation loss
    train_loss = 0.0
    valid_loss = 0.0
    train_correct = 0
    valid_correct = 0
    train_total = 0
    valid_total = 0

    # Train
    model.train()
    for data, target in tqdm(train_loader):

        # move tensors to GPU
        data, target = data.to(device), target.to(device)

        optimizer.zero_grad()
        # forward pass
        output = model(data)
        loss = criterion(output, target)
        # backward pass
        loss.backward()
        optimizer.step()

        train_loss += loss.item()*data.size(0)

        # update training accuracy
        _, predicted = torch.max(output.data, 1)
        train_total += target.size(0)
        train_correct += (predicted == target).sum().item()
```

In this code snippet, the training process is done using the Pytorch library for 30 epochs. Initially, variables are defined to store the model's loss on the training and validation data, and after each epoch, they are updated with the corresponding values. During the training process, we feed the data along with their labels into the neural network, and after computing the derivatives using the Pytorch library, we adjust the weights.

```
    #validate
    model.eval()
    for data, target in tqdm(validation_loader):
        # move tensors to GPU
        data, target = data.cuda(), target.cuda()
        # forward pass
        output = model(data)

        loss = criterion(output, target)

        # update average validation loss
        valid_loss += loss.item()*data.size(0)
        # update validation accuracy
        _, predicted = torch.max(output.data, 1)
        valid_total += target.size(0)
        valid_correct += (predicted == target).sum().item()

    # calculate average losses and accuracy
    train_loss = train_loss/len(train_loader.dataset)
    valid_loss = valid_loss/len(validation_loader.dataset)
    train_acc = 100 * train_correct / train_total
    valid_acc = 100 * valid_correct / valid_total
    train_losslist.append(train_loss)

    # print training/validation statistics
    print('Epoch: {} \tTraining Loss: {:.6f} - Validation Loss: {:.6f} - Training Accuracy: {:.2f}% -
Validation Accuracy: {:.2f}%'.format(
        epoch+1, train_loss, valid_loss, train_acc, valid_acc))

    # save model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}). The new model saved.'.format(valid_loss_min,
valid_loss))
        torch.save(model.state_dict(), 'CNN-2.pt')
        valid_loss_min = valid_loss

    history['train_loss'].append(train_loss)
    history['valid_loss'].append(valid_loss)
    history['train_acc'].append(train_acc)
    history['valid_acc'].append(valid_acc)
```

During the validation stage, we evaluate the model using the images from the validation set and plot the corresponding error and accuracy graphs on the validation data. At the end of each epoch, we examine the changes in the error on the validation data and save the best model during the training process. This prevents overfitting of the model on the training data.

4- Displaying Error and Accuracy Plots for Training and Validation Data:

```
fig=plt.figure(1,figsize=(10,5))

plt.subplot(1, 2, 1)
plt.plot(history['train_loss'], 'bo-', label='Train')
plt.plot(history['valid_loss'], 'r--', label='Valid')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history['train_acc'], 'bo-', label='Train')
plt.plot(history['valid_acc'], 'r--', label='Valid')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
```

We can use the Matplotlib library to plot the error and accuracy graphs for the training and validation data.

5 -Implementation of Model Evaluation on Test Data:

This code snippet is implemented for the final evaluation of the model's accuracy and error on the test data. We put the model in evaluation mode, and then feed the test data into the model for classification. We compare the model's output with the actual labels of the test data, and then compute the accuracy of the model in predicting the correct or incorrect labels.

```
model.load_state_dict(torch.load('CNN-1.pt'))

# test the model
model.eval()

test_loss = 0.0
test_correct = 0
test_total = 0

with torch.no_grad():
    for data, target in tqdm(test_loader):
        # move tensors to GPU
        data, target = data.cuda(), target.cuda()
        # forward pass
        output = model(data)

        loss = criterion(output, target)

        # update test loss
        test_loss += loss.item()*data.size(0)
        # update test accuracy
        _, predicted = torch.max(output.data, 1)
        test_total += target.size(0)
        test_correct += (predicted == target).sum().item()

# calculate average test loss and accuracy
test_loss = test_loss/len(test_loader.dataset)
test_acc = 100 * test_correct / test_total

# print test statistics
print('Test Loss: {:.6f} - Test Accuracy: {:.2f}%'.format(test_loss, test_acc))

# add to history
history['test_loss'] = test_loss
history['test_acc'] = test_acc
```

You can view the results of this model in section 3 or the Results section.

In this section, we will use the methods mentioned in the last descriptive questions to improve the accuracy of the CNN network implemented in the first section and report and visualize the results.

As you know, in section 1, we became familiar with the concepts and advantages of CNN networks. We also mentioned several ways to increase the accuracy of CNN networks. Now, we intend to increase the results obtained in the previous section by using methods such as changing the architecture of the neural network, adjusting hyperparameters, and changing the optimization system.

1- Implementing a new CNN architecture:

```
class CNN(nn.Module):

    def __init__(self):

        super(CNN, self).__init__()

        self.conv_layer = nn.Sequential(

            # Conv Layer block 1
            nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Conv Layer block 2
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Dropout2d(p=0.05),

            # Conv Layer block 3
            nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )

        self.fc_layer = nn.Sequential(
            nn.Dropout(p=0.1),
            nn.Linear(4096, 1024),
            nn.ReLU(inplace=True),
            nn.Linear(1024, 512),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.1),
            nn.Linear(512, 10)
        )


    def forward(self, x):

        # conv layers
        x = self.conv_layer(x)

        # flatten
        x = x.view(x.size(0), -1)

        # fc layer
        x = self.fc_layer(x)

        return x


model = CNN()
model.cuda()

print(model)
```

This code snippet is similar to the previous question, for implementing a convolutional neural network (CNN). However, there are differences between this architecture and the previous one.

This CNN consists of three convolutional blocks, each containing two convolutional layers and one max pooling layer, followed by a ReLU activation function. The first convolutional block takes a 3-channel (RGB) input and outputs 32 feature maps. The next convolutional blocks increase the number of feature maps to 64 and 256, respectively. Batch normalization is applied after each convolutional layer to improve stability and training speed. In addition, a dropout layer is added after the second max pooling layer to prevent overfitting.

After the convolutional layers, the output is flattened to enter the fully connected neural network and passes through three fully connected layers with ReLU activation function. The first fully connected layer has 1024 neurons, the second layer has 512 neurons, and the final layer has 10 neurons, corresponding to the number of classes for classification.

In general, the main differences between this architecture and the previous one are the number of convolutional blocks, the use of batch normalization, and the addition of dropout layers. These changes can improve the performance and stability of the model during training.

2- Implementation of Loss Function and Optimizer:

```
# specify loss function

criterion = nn.CrossEntropyLoss()
```

Loss function for the training process: CrossEntropy

```
#Changing the optimizer setting while training
if epoch <= 10:
  optimizer = optim.Adam(model.parameters(), lr=learning_rate)
elif epoch > 10 and epoch <= 25:
  optimizer = optim.Adam(model.parameters(), lr=(learning_rate)/10)
else:
 optimizer = optim.Adam(model.parameters(), lr=(learning_rate)/50)
```

Optimizer: Adam

Learning rate: In this method, to optimize the training process and ultimately increase the accuracy of the neural network, we change the learning rate during the training. At the beginning of the training, the learning rate is set to 0.001, and after 10 iterations, it is divided by 10. After 25 iterations, it is divided by 50. This allows the neural network to explore at the beginning of the training and to extract at the end of the training.

3- Implementation of the training and validation part of the model:

```
    #validate
    model.eval()
    for data, target in tqdm(validation_loader):
        # move tensors to GPU
        data, target = data.cuda(), target.cuda()
        # forward pass
        output = model(data)

        loss = criterion(output, target)

        # update average validation loss
        valid_loss += loss.item()*data.size(0)
        # update validation accuracy
        _, predicted = torch.max(output.data, 1)
        valid_total += target.size(0)
        valid_correct += (predicted == target).sum().item()

    # calculate average losses and accuracy
    train_loss = train_loss/len(train_loader.dataset)
    valid_loss = valid_loss/len(validation_loader.dataset)
    train_acc = 100 * train_correct / train_total
    valid_acc = 100 * valid_correct / valid_total
    train_losslist.append(train_loss)

    # print training/validation statistics
    print('Epoch: {} \tTraining Loss: {:.6f} - Validation Loss: {:.6f} - Training Accuracy: {:.2f}% -
Validation Accuracy: {:.2f}%'.format(
        epoch+1, train_loss, valid_loss, train_acc, valid_acc))

    # save model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}). The new model saved.'.format(valid_loss_min,
valid_loss))
        torch.save(model.state_dict(), 'CNN-2.pt')
        valid_loss_min = valid_loss

    history['train_loss'].append(train_loss)
    history['valid_loss'].append(valid_loss)
    history['train_acc'].append(train_acc)
    history['valid_acc'].append(valid_acc)
```

In this code snippet, the training process is performed using the PyTorch library for 30 iterations. We first define variables to store the model's loss value on the training and validation data, and

then initialize them after each iteration. In the training process, we input the data along with their labels into the neural network, and after calculating the derivatives using the PyTorch library, we adjust the weights.

In the validation step, we evaluate the model using the images from the validation set and plot the corresponding error and accuracy graphs on the validation data. At the end of each iteration, we examine the changes in the error on the validation data and save the best model during the training process. This prevents the model from overfitting on the training data.

4- Displaying the error and accuracy graphs of the model on the training and validation data:

```
fig=plt.figure(1,figsize=(10,5))

plt.subplot(1, 2, 1)
plt.plot(history['train_loss'], 'bo-', label='Train')
plt.plot(history['valid_loss'], 'r--', label='Valid')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history['train_acc'], 'bo-', label='Train')
plt.plot(history['valid_acc'], 'r--', label='Valid')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
```

Using the matplotlib library, we plot the error and accuracy graphs on the training and validation data.

5- Implementation of the evaluation part of the model on the test data:

```
model.load_state_dict(torch.load('CNN-2.pt'))

# test the model
model.eval()

test_loss = 0.0
test_correct = 0
test_total = 0

with torch.no_grad():
    for data, target in tqdm(test_loader):
        # move tensors to GPU if CUDA is available
        data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # update test loss
        test_loss += loss.item()*data.size(0)
        # update test accuracy
        _, predicted = torch.max(output.data, 1)
        test_total += target.size(0)
        test_correct += (predicted == target).sum().item()

# calculate average test loss and accuracy
test_loss = test_loss/len(test_loader.dataset)
test_acc = 100 * test_correct / test_total

# print test statistics
print('Test Loss: {:.6f} - Test Accuracy: {:.2f}%'.format(test_loss, test_acc))

# add to history
history['test_loss'] = test_loss
history['test_acc'] = test_acc
```

This code snippet is for the final evaluation of the model's accuracy and error on the test data. We put the model in evaluation mode and then input the test data into the model for classification. We compare the model's output with the actual labels of the test data and then calculate the accuracy of the model based on the correct and incorrect predictions.

You can see the results obtained from this model in section 3 or in the result analysis.

Conclusion and Results analysis:

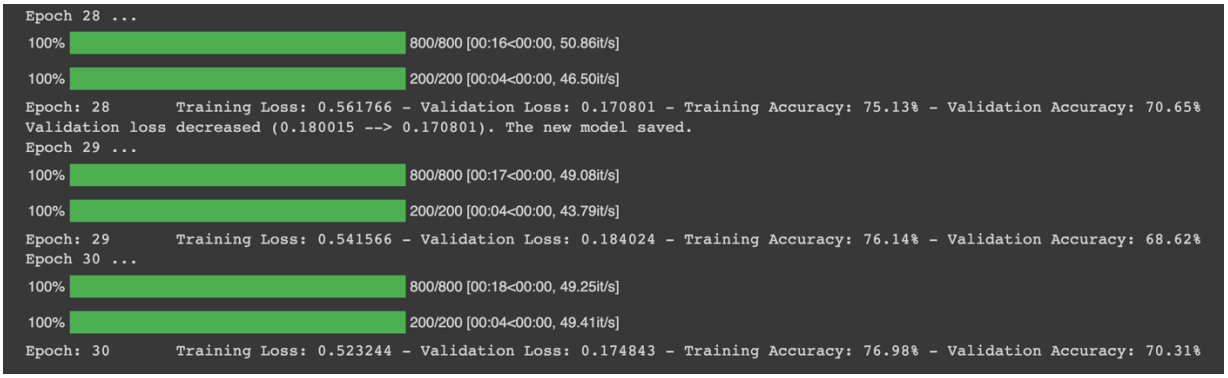
In this report, we first reviewed the concepts of Convolutional Neural Networks (CNNs) and then explained the implemented functions. Now we move on to the results and discussion section, where we present and analyze the results of the implemented models. In this section, we will recap the questions asked and examine the results of implementing each section.

1- For this section, we had to implement a Convolutional Neural Network on the CIFAR10 dataset and report and visualize the values of Accuracy and Loss by running the Train, Validation, Test stages.
The required functions to solve this question are explained in section 2 or the explanation section of the implementation methods. Now, we present the results.

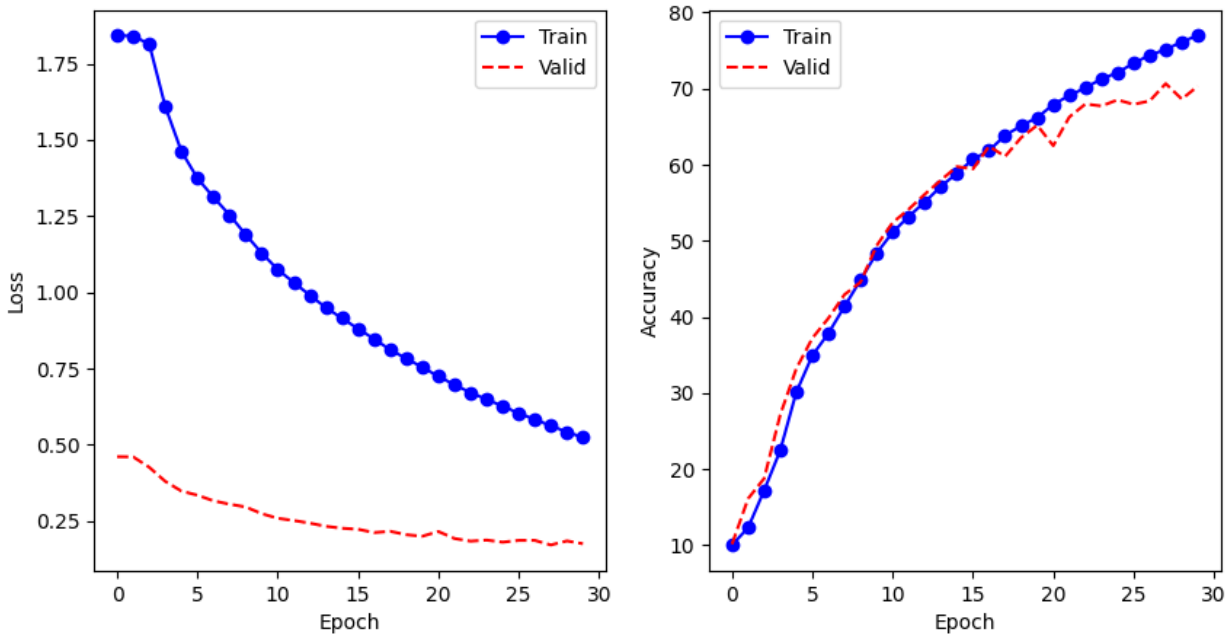
Performing the training process:

After implementing the CNN neural network and determining its loss function and optimizer, we start the training process and print the accuracy and loss on the training and validation data at each iteration. Finally, we save the best model for evaluation on the test data.

The training process is shown in the image below:



Accuracy and Loss plots on training and validation data during the CNN training process:



As you can see in the image above, the loss value on the training and validation data is decreasing in each iteration, and the accuracy value is increasing.

The accuracy of the implemented CNN model on the evaluation data is reported as 70% after the training process.

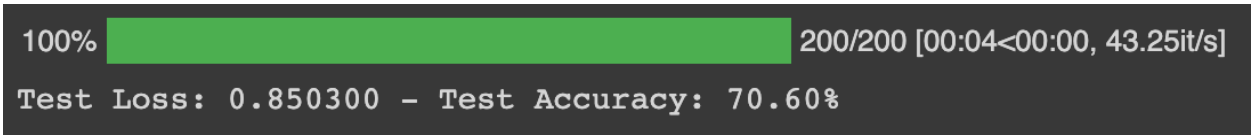
2- In this section, using the methods mentioned in the last descriptive questions, we increased the accuracy of the implemented network in the first section and reported and visualized the accuracies.

As you know, in the first section or the familiarity with the concepts and advantages of CNNs, several ways to increase the accuracy of CNNs were mentioned. Then, in the second section, by using methods such as changing the architecture of the neural network, adjusting hyperparameters, and changing the optimization system, we tried to implement a neural network that would produce better outputs than the previous network.

Performing the training process:
After implementing the CNN neural network and determining its loss function and optimizer, we start the training process and print the accuracy and loss on the training and validation data at each iteration. Finally, we save the best model for evaluation on the test data.

The training process is shown in the image below:

The accuracy and loss plots on training and validation data during the CNN training process:



As you can see in the image above, the loss value on the training and validation data is decreasing in each iteration, and the accuracy value is increasing.

After the training process, we evaluate the accuracy of the model on the test data. The accuracy of the implemented CNN model after applying the mentioned methods to increase its accuracy is reported as 85% on the evaluation data.

In conclusion, by applying the methods mentioned in the descriptive questions and optimizing the CNN model, we were able to achieve a higher accuracy of 85% on the evaluation data, which is a significant improvement compared to the initial accuracy of 70%.

2- In this section, using the methods mentioned in the last section of the descriptive questions, you are asked to increase the accuracy of the implemented CNN network in the first section and report and visualize the accuracies obtained. As you know, in the first section, we introduced the concepts and advantages of CNN neural networks and mentioned several ways to increase the accuracy of CNN networks. Then, in the second section, we attempted to implement a neural network by modifying the architecture structure, adjusting the hyperparameters, and changing the optimization system to achieve better outputs than the previous network.

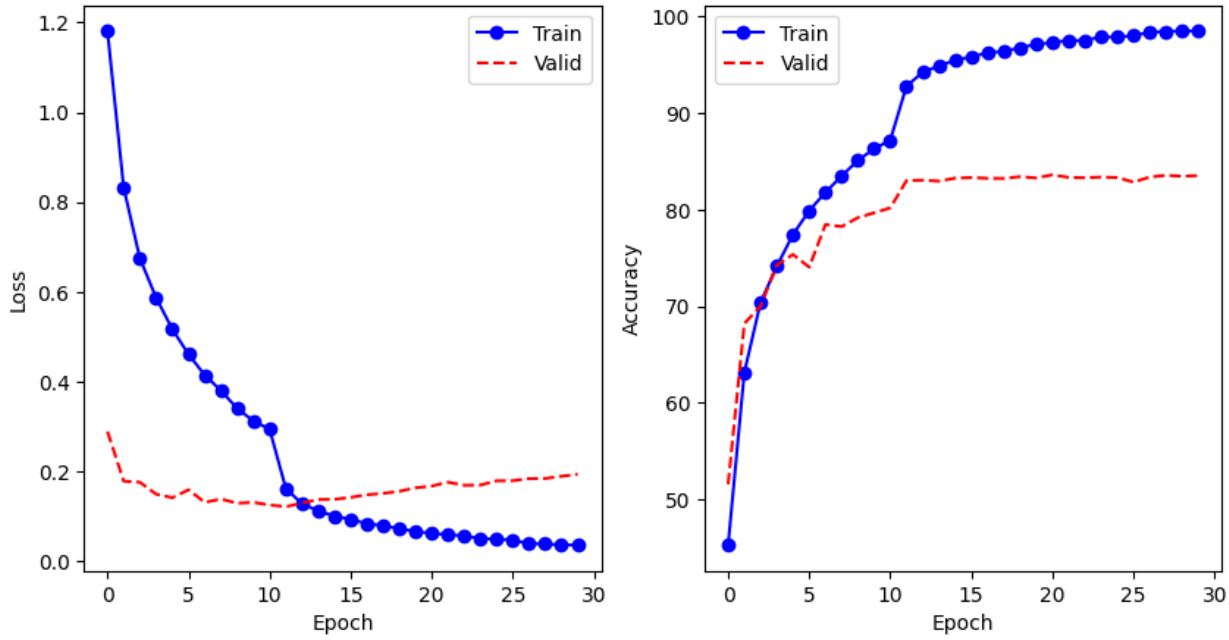
Training process:

After implementing the CNN neural network and determining its loss function and optimizer, we start the training process and print the accuracy and loss on the training and validation data at each iteration. Finally, we save the best model for evaluation on the test data.

The training process is shown in the image below:

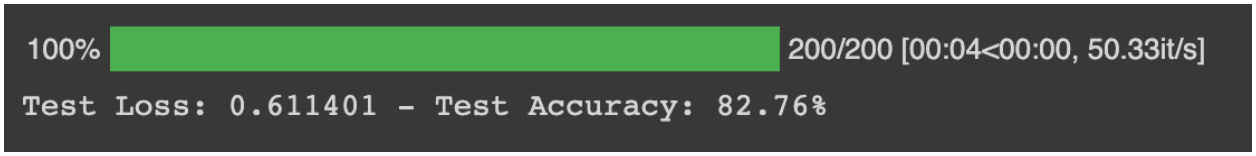
```
Epoch: 26       Training Loss: 0.047191 - Validation Loss: 0.179403 - Training Accuracy: 98.00% - Validation Accuracy: 82.85%
Epoch 27 ...
100% 800/800 [00:18<00:00, 46.18it/s]
100% 200/200 [00:03<00:00, 54.42it/s]
Epoch: 27       Training Loss: 0.039163 - Validation Loss: 0.184048 - Training Accuracy: 98.36% - Validation Accuracy: 83.37%
Epoch 28 ...
100% 800/800 [00:18<00:00, 44.19it/s]
100% 200/200 [00:03<00:00, 53.02it/s]
Epoch: 28       Training Loss: 0.038579 - Validation Loss: 0.184361 - Training Accuracy: 98.44% - Validation Accuracy: 83.55%
Epoch 29 ...
100% 800/800 [00:19<00:00, 43.84it/s]
100% 200/200 [00:03<00:00, 52.86it/s]
Epoch: 29       Training Loss: 0.036496 - Validation Loss: 0.189392 - Training Accuracy: 98.49% - Validation Accuracy: 83.46%
Epoch 30 ...
100% 800/800 [00:18<00:00, 45.83it/s]
100% 200/200 [00:04<00:00, 51.55it/s]
Epoch: 30       Training Loss: 0.036124 - Validation Loss: 0.193636 - Training Accuracy: 98.49% - Validation Accuracy: 83.52%
```

The accuracy and loss plots on training and validation data during the CNN training process:



As you can see in the image above, the loss value is decreasing and the accuracy is increasing on the training and validation data in each iteration.

After the training process, we evaluate the accuracy of the model on the test data. The accuracy of the implemented CNN network on the evaluation data is reported as 82%.



Therefore, we conclude that the methods of modifying the architecture structure, adjusting the hyperparameters, and changing the optimization system, as implemented in the second section, can increase the accuracy of the CNN neural network.