# Work Report

ANN 05 assignment – Question 3 and 4

Hesam Koucheki

403723987

# Problem 3

## Part a

In this question we have worked with the CIFAR-10 dataset which cotains images from 10 different classes.

```
# DataLoaders going to help us. What are they doing?
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True)
valloader = torch.utils.data.DataLoader(valset, batch_size=64, shuffle=True)
```

The DataLoaders are really helpful because they make it easier to load the data in smaller parts instead of all at once. In the code, the trainloader is used to load the training data, and the valloader is used to load the validation data. Both of these loaders take the datasets (trainset and valset) and split them into smaller groups, called batches, with each batch having 64 images. This is what the batch_size=64 means. By loading the data in batches, it saves memory and makes the training process faster.

Another thing the DataLoaders do is shuffle the data when shuffle=True is used. Shuffling is important because it mixes up the data so the model doesn't learn patterns just based on the order of the images. This is especially useful for training because it helps the model generalize better to new data. Overall, DataLoaders make handling data simpler and more efficient, and they are an important part of training and validating a model.

```
dataiter = iter(trainloader)
images ,labels = next(dataiter)
print(images.shape)
print(labels.shape)
```

In this part of the code, we are creating a data iterator from the trainloader by using the iter function. This iterator helps us go through the training data one batch at a time. Then, the next function is used on the iterator to get the next batch of data, which gives us two things, images and labels. The images are the actual input data, and the labels are the corresponding class labels for those images.

When we print the shapes of images and labels, we can check if everything is working properly. For example, the shape of images might be (64, 3, 32, 32) because each batch has 64 images, and each image has 3 color channels (red, green, blue) and a size of 32 by 32 pixels. The labels shape would be (64), which means there are 64 labels in the batch, one for each image. This step is useful to confirm that the DataLoader is giving the data in the right format before it is used for training the model.

```
# We used data augmentation here, what is data augmentation?
train_transform = Compose([
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor(),
    transforms.Normalize([0, 0, 0], [1, 1, 1])
])
test_transform = Compose([
    transforms.ToTensor(),
    transforms.Normalize([0, 0, 0], [1, 1, 1])
])
trainset = datasets.CIFAR10(train_path, download=True, train=True, transform=train_transform)
valset = datasets.CIFAR10(test_path, download=True, train=False, transform=test_transform)
```

In this code, we are using something called data augmentation, which is a technique used to artificially increase the size and variety of the dataset by applying transformations to the images. Data augmentation helps the model learn better by making it more robust to different variations in the input data, like rotations, flips, or changes in brightness. This is especially useful when the dataset is small or lacks enough diversity to train the model effectively.

For the training set, we use several transformations through the train_transform. First, we apply a random horizontal flip with a probability of 0.5, meaning half of the images will be flipped horizontally. Then, we use random cropping with padding to slightly shift and crop the images, which simulates different perspectives. Finally, we convert the images to tensors using ToTensor and normalize their pixel values using Normalize. These steps ensure that the model sees different versions of the same data during training, improving its generalization.

For the test set, we don't use augmentation since it's meant for evaluation, so we only apply ToTensor and Normalize. These transformations prepare the data in the correct format for the model but keep it consistent with the original images. The CIFAR-10 datasets are loaded with these transformations applied, making the trainset augmented and the valset ready for testing. This way, the model is trained with diverse data but evaluated on clean and unchanged test data.

```
#Our CNN implement, 3 conv layer, pooling before fully connected Then fully connected layer for outputing our 10 classes
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3,   64,  3)
        self.conv2 = nn.Conv2d(64,  128, 3)
        self.conv3 = nn.Conv2d(128, 256, 3)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 4 * 4, 128)
        self.fc2 = nn.Linear(128, 256)
        self.fc3 = nn.Linear(256, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = x.view(-1, 64 * 4 * 4)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return F.log_softmax(x, dim=1)


model = Net()
```

This code defines a convolutional neural network called Net for classifying images into 10 classes. It has three convolutional layers (conv1, conv2, conv3) that extract features from the images, followed by pooling layers to reduce dimensions. The convolutional layers increase feature maps from 3 to 64, 128, and 256, respectively. After pooling, the feature maps are flattened and passed through three fully connected layers (fc1, fc2, fc3). The final layer outputs 10 values, one for each class, using log_softmax. ReLU is used after each layer to add non-linearity. The model is built to effectively process CIFAR-10 images and classify them.

```
# Model Hyperparameters, you can change every one of them to trying it with new parameters
criterion = nn.CrossEntropyLoss() # also can use "Mean Squared Error Loss"
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)#also can use "Adam Optimizer"
val_per_epoch = 5
n_epochs = 30
batch_size = 4


#tuning Hyperparameters is important and difficult.
```

This part of the code sets the model's hyperparameters, which can be changed to test different configurations and improve performance. The criterion is set to CrossEntropyLoss, which is commonly used for classification tasks, but Mean Squared Error Loss can also be tried. The optimizer is set to SGD with a learning rate of 0.001 and momentum of 0.9, although the Adam optimizer can be used as an alternative.

The val per epoch parameter determines how often validation is performed during training, set to 5 times per epoch here. The n epochs parameter sets the total number of epochs, or full passes through the training data, which is 30 in this case. The batch size specifies how many samples are processed at a time, set to 4.  While training that for myself, I observed that the training process is too slow while not utilizing all the resources such as ram or GPU, thus, I changed the batch size and increased it to 128. That increasingly improved the model training speed.

```
#Training loop


history = dict()
history['train_loss']= list()
history['train_acc']= list()
history['val_loss']= list()
history['val_acc']= list()
for epoch in range(n_epochs):
    running_loss = 0
    running_acc = 0

    model.train()
    for idx, (X, y) in enumerate(tqdm(trainloader)):
        optimizer.zero_grad()
        X, y = X.to(device), y.to(device)
        y_hat = model(X)
        loss = criterion(y_hat, y)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        running_acc += accuracy(y_hat, y)
    else:
        running_loss = running_loss/ len(trainloader)
        running_acc = running_acc / len(trainloader)
        history['train_loss'].append(running_loss)
        history['train_acc'].append(running_acc)
        print(f'Epoch {epoch+1}/{n_epochs} : training loss: {running_loss}, training acc: 
{running_acc}')





    if epoch% val_per_epoch == val_per_epoch-1:
        running_loss = 0
        running_acc = 0
        model.eval()
        with torch.no_grad():
            for idx, (X, y) in enumerate(tqdm(valloader)):
                X, y = X.to(device), y.to(device)
                y_hat = model(X)
                loss = criterion(y_hat, y)

                running_loss += loss.item()
                running_acc += accuracy(y_hat, y)
            else:
                running_loss = running_loss/ len(valloader)
                running_acc = running_acc / len(valloader)
                history['val_loss'].append(running_loss)
                history['val_acc'].append(running_acc)
                print(f'Epoch {epoch+1}/{n_epochs} : validation loss: {running_loss},
validation acc: {running_acc}')
```

This code defines the training loop for the model, where the network is trained over multiple epochs. During each epoch, the model processes the training data to update its weights and improve its predictions, and validation is performed periodically to evaluate its performance on unseen data.

At the start, a dictionary called history is created to store the training and validation losses and accuracies for each epoch. For every epoch, the model is set to training mode using model.train. A loop runs through batches of training data from the trainloader. For each batch, the optimizer's gradients are reset, and the input data and labels are sent to the device (CPU or GPU). The model makes predictions, computes the loss using the chosen criterion, and backpropagation is performed to update the weights. The running loss and accuracy are accumulated for the entire training set and averaged at the end of the epoch. The results are stored in history and printed to show progress.

Every few epochs, as specified by val per epoch, the model switches to evaluation mode using model.eval, and validation is performed using the valloader. No gradient calculations are done during validation, as indicated by torch.no_grad. The same process is followed to compute the validation loss and accuracy, which are also averaged and stored in history. These periodic evaluations ensure that the model’s performance on unseen data is tracked during training.

This loop is essential for improving the model's performance while monitoring how well it generalizes to data it has not seen before.

```
# Review results
```

```
from sklearn.metrics import confusion_matrix ,f1_score ,precision_score ,recall_score,
accuracy_score

def test_label_predictions(model, device, test_loader):
    model.eval()
    actuals = []
    predictions = []
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            prediction = output.argmax(dim=1, keepdim=True)
            actuals.extend(target.view_as(prediction))
            predictions.extend(prediction)
    return [i.item() for i in actuals], [i.item() for i in predictions]

actuals, predictions = test_label_predictions(model, device, valloader)
print('Accuracy: %f' % accuracy_score(actuals, predictions))
print('Confusion matrix:')
print(confusion_matrix(actuals, predictions))
print('F1 score: %f' % f1_score(actuals, predictions, average='weighted'))
print('Precision: %f' % precision_score(actuals, predictions, average='weighted'))
print('Recall: %f' % recall_score(actuals, predictions, average='weighted'))
```

This code evaluates the trained model's performance on the validation set using several metrics such as accuracy, confusion matrix, f1 score, precision, and recall. These metrics help us understand how well the model is performing and how well it distinguishes between the different classes.

The function test label predictions is used to test the model. It sets the model to evaluation mode with model.eval so no gradients are calculated. For each batch of data in the validation loader, the model makes predictions, and the actual and predicted labels are stored. Predictions are taken from the output by selecting the index of the highest value using argmax.

After generating predictions, various metrics are calculated using sklearn. The accuracy is shown to be 82.16 percent, which means that 82 percent of the predictions are correct. The confusion matrix shows the detailed performance for each class, revealing which classes are more likely to be misclassified. The f1 score, precision, and recall are all about 82 percent, which shows that the model performs consistently across different classes.

Overall, the results show that the model is doing well but still has some room for improvement, especially in specific classes that have more errors. The confusion matrix can help identify these areas, and the insights can guide further model tuning or data adjustments to improve accuracy.

```
# Adding DropOut here. What is DropOut?, what is advantages and disadvantages?, how result
going to be different?
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3,   64,  3)
        self.conv2 = nn.Conv2d(64,  128, 3)
        self.conv3 = nn.Conv2d(128, 256, 3)
        self.pool  = nn.MaxPool2d(2, 2)
        self.do1 = nn.Dropout(0.2)  # 20% Probability
        self.fc1 = nn.Linear(64 * 4 * 4, 128)
        self.do2 = nn.Dropout(0.2)  # 20% Probability
        self.fc2 = nn.Linear(128, 256)
        self.do3 = nn.Dropout(0.1)  # 10% Probability
        self.fc3 = nn.Linear(256, 10)

    def forward(self, x):

        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = x.view(-1, 64 * 4 * 4)
        x = self.do1(x)
        x = F.relu(self.fc1(x))
        x = self.do2(x)
        x = F.relu(self.fc2(x))
        x = self.do3(x)
        x = self.fc3(x)
        return F.log_softmax(x, dim=1)


model = Net()
```
Dropout is a regularization technique used in neural networks to prevent overfitting during training. It works by randomly "dropping out" (setting to zero) a fraction of the neurons in the network during each forward pass. This ensures that the model does not rely too heavily on specific neurons and instead forces it to learn more general patterns in the data.

In this updated code, Dropout has been added after the fully connected layers. For example, do1, do2, and do3 represent dropout layers with probabilities of 20% and 10%. This means that during training, 20% of the neurons in the first and second fully connected layers and 10% in the final layer will be randomly set to zero. However, during evaluation or testing, dropout is turned off, and all neurons are used for predictions.

The advantages of dropout include its ability to reduce overfitting, improve generalization, and help the network learn robust features that do not rely too much on specific neurons. By introducing randomness during training, it can make the network less sensitive to noise in the data.

The disadvantages are that it increases training time since the network has to learn despite having fewer active neurons in each forward pass. It can also make convergence slower and may require careful tuning of dropout rates to avoid underfitting (dropping too many neurons).

The results after adding dropout will likely show better generalization on unseen data, meaning the model may perform better on the validation and test sets. However, the training accuracy might decrease slightly because the network is learning with fewer active neurons during training. This tradeoff is often acceptable since the goal is to improve performance on data the model has not seen before. Dropout is particularly useful when dealing with large networks or small datasets prone to overfitting.

The other parts of this code are exactly the same as the previous one (whitout dropout), so, I do not repeat the report for them.

## Part b

MNIST is a simple and well-known dataset of handwritten digits, containing 60,000 training images and 10,000 testing images. Each image is grayscale and has a size of 28x28 pixels. The goal is to classify these digits into 10 classes (0 through 9).

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

train_dataset = datasets.MNIST(root='./data', train=True, transform=transform, download=True)
test_dataset = datasets.MNIST(root='./data', train=False, transform=transform, download=True)

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=128, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=128, shuffle=False)
```

This code is used to prepare the MNIST dataset for training and testing. First, transformations are applied to the dataset. The images are converted into tensors and then normalized to have values between -1 and 1, which helps the model learn better.

The training dataset is loaded with train dataset and the test dataset with test dataset. The train dataset uses the training images, while the test dataset uses the test images. Both apply the same transformations, and if the dataset is not already present, it will be downloaded.

Finally, data loaders are created. The train loader splits the training data into batches of 64 and shuffles them to avoid bias in training. The test loader also divides the test data into batches but does not shuffle them since order does not matter during testing. This setup organizes the data for efficient training and testing.

```python
import torch.nn.functional as F

class MNISTClassifier(nn.Module):
    def __init__(self):
        super(MNISTClassifier, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 5 * 5, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 64 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

model = MNISTClassifier()
```

This code defines a simple convolutional neural network called MNISTClassifier to classify handwritten digits from the MNIST dataset. The network has two convolutional layers and two fully connected layers.

In the init method, the first convolutional layer takes input with one channel (grayscale images) and outputs 32 feature maps using a 3x3 filter. The second convolutional layer takes these 32 feature maps and outputs 64 feature maps, also using a 3x3 filter. A max-pooling layer follows each convolutional layer, reducing the spatial dimensions by half, making the network more efficient.

After the convolutional and pooling layers, the output is flattened into a one-dimensional tensor to feed into the fully connected layers. The first fully connected layer reduces the features to 128, and the second fully connected layer produces 10 outputs, one for each digit class (0 to 9). The ReLU activation function is applied after each layer except the final one to introduce non-linearity.

The forward method defines how the data flows through the network during training and prediction. Finally, an instance of the MNISTClassifier is created and stored in the model variable, ready for training and evaluation.

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

This code sets up the loss function and optimizer for training the model. The criterion is set to CrossEntropyLoss, which is a commonly used loss function for classification tasks. It calculates the difference between the predicted probabilities and the actual class labels, helping the model adjust its weights to make better predictions.

The optimizer is set to Adam, a popular optimization algorithm that combines the benefits of both momentum and adaptive learning rates. It updates the model's parameters during training to minimize the loss. The learning rate is set to 0.001, which controls the size of the weight updates. This combination of CrossEntropyLoss and Adam optimizer is efficient and commonly used for tasks like MNIST classification.

```
from tqdm import tqdm

num_epochs = 20
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model.to(device)

for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0

    # Wrap train_loader with tqdm to show a progress bar
    with tqdm(train_loader, unit="batch") as train_progress:
        train_progress.set_description(f"Epoch {epoch+1}/{num_epochs}")
        for inputs, labels in train_progress:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()  # Reset gradients
            outputs = model(inputs)  # Forward pass
            loss = criterion(outputs, labels)  # Compute loss
            loss.backward()  # Backward pass
            optimizer.step()  # Update weights

            running_loss += loss.item()

            # Update tqdm description with loss for the current batch
            train_progress.set_postfix(loss=loss.item())

    # Average loss for the epoch
    epoch_loss = running_loss / len(train_loader)
    print(f"Epoch {epoch+1}/{num_epochs}, Average Loss: {epoch_loss}")
```

This code defines the training loop for the model and uses tqdm to display a progress bar for better visualization of the training process. The loop runs for 20 epochs, where each epoch represents a complete pass through the training data.

The model is set to training mode with model.train(), ensuring that training-specific operations like dropout (if present) are applied. Inside the loop, tqdm is wrapped around the train_loader to show the progress of batch processing during training. For each batch, the input data and labels are sent to the device and the optimizer resets the gradients with optimizer.zero_grad().

The model performs a forward pass to make predictions, and the loss is calculated using the criterion. Then, the loss.backward() function computes the gradients, and optimizer.step() updates the model's weights to minimize the loss. The loss for each batch is added to running_loss, which is averaged at the end of the epoch.

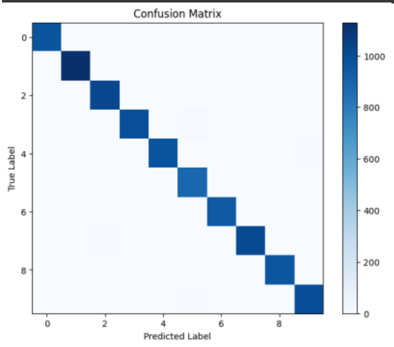
After each epoch, the average loss is calculated and printed to track the model's performance over time.

Result after 20 epochs:

```
Epoch [1/10], Loss: 0.0243
Epoch [2/10], Loss: 0.0240
Epoch [3/10], Loss: 0.0016
Epoch [4/10], Loss: 0.0019
Epoch [5/10], Loss: 0.0555
Epoch [6/10], Loss: 0.0006
Epoch [7/10], Loss: 0.0133
Epoch [8/10], Loss: 0.0012
Epoch [9/10], Loss: 0.0001
Epoch [10/10], Loss: 0.0018
```

```python
# Evaluate the model on GPU
model.eval()
correct = 0
total = 0
all_preds = []
all_labels = []
with torch.no_grad():
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)

        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())

accuracy = 100 * correct / total
error_rate = 100 - accuracy
precision = precision_score(all_labels, all_preds, average='weighted')
recall = recall_score(all_labels, all_preds, average='weighted')
conf_matrix = confusion_matrix(all_labels, all_preds)

print('Accuracy of the model on the test images: {} %'.format(accuracy))
print('Error Rate: {} %'.format(error_rate))
print('Precision: {}'.format(precision))
print('Recall: {}'.format(recall))
print('Confusion Matrix:\n', conf_matrix)

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
plt.imshow(conf_matrix, interpolation='nearest', cmap=plt.cm.Blues)
plt.title('Confusion Matrix')
plt.colorbar()
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```

```
Accuracy of the model on the test images: 99.02 %
Error Rate: 0.980000000000004 %
Precision: 0.9902637516949507
Recall: 0.9902
Confusion Matrix:
 [[ 970    0    3    0    0    3    3    0    0    1]
 [    0 1129    1    2    0    0    0    2    1    0]
 [    2    0 1027    0    1    0    0    2    0    0]
 [    0    0    1  995    0    9    0    1    4    0]
 [    0    0    1    0  968    0    3    2    0    8]
 [    0    0    0    1    0  889    2    0    0    0]
 [    2    2    2    0    1    2  948    0    1    0]
 [    0    0   11    0    0    0    0 1014    0    3]
 [    0    0    1    1    3    3    0    0  965    1]
 [    0    0    1    0    2    6    0    3    0  997]]
```



## Problem 4

```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
n_examples = 50000


X1_train = []
X1_test = []
X2_train = []
X2_test = []
Y1_train = []
Y1_test = []
Y2_train = []
Y2_test = []

for ix in range(n_examples):
    if y_train[ix] < 5:
        # put data in set 1
        X1_train.append(x_train[ix]/255.0) #Dividing by 255 is for normalization
        Y1_train.append(y_train[ix])
    else:
        # put data in set 2
        X2_train.append(x_train[ix]/255.0)
        Y2_train.append(y_train[ix])

for ix in range(y_test.shape[0]):
    if y_test[ix] < 5:
        # put data in set 1
        X1_test.append(x_test[ix]/255.0)
        Y1_test.append(y_test[ix])
    else:
        # put data in set 2
        X2_test.append(x_test[ix]/255.0)
        Y2_test.append(y_test[ix])
```

In this part of the code, we load the CIFAR-10 dataset, which has 50,000 training images and 10,000 test images. The dataset is split into two separate sets based on the labels of the images. To do this, we first normalize the data by dividing each pixel value by 255. This makes the pixel values range between 0 and 1 instead of 0 to 255. Normalizing the data helps the model train better and faster because it keeps the numbers small and more consistent.

After loading and normalizing the data, we separate the images into two groups. The first group, called Set 1, contains all images with labels less than 5. These are classes 0, 1, 2, 3, and 4. The second group, called Set 2, contains images with labels 5 or more, which are classes 5, 6, 7, 8, and 9. For each image in the training data, we check its label, and if it belongs to Set 1, we add it to the list for Set 1. If it belongs to Set 2, we add it to the list for Set 2. The same process is repeated for the test data, so we have separate training and testing sets for each group.

This setup creates two distinct datasets that we can use for experiments, like transfer learning. Set 1 has images from the first five classes, and Set 2 has images from the other five classes. This division allows us to train a model on one dataset, like Set 1, and then adapt or fine-tune it for the other dataset, Set 2. The splitting and normalization prepare the data properly for these kinds of experiments.

```
X1_train = np.asarray(X1_train).reshape((-1, 32, 32, 3))
X1_test = np.asarray(X1_test).reshape((-1, 32, 32, 3))
X2_train = np.asarray(X2_train).reshape((-1, 32, 32, 3))
X2_test = np.asarray(X2_test).reshape((-1, 32, 32, 3))

Y1_train = to_categorical(np.asarray(Y1_train), 5)
Y1_test = to_categorical(np.asarray(Y1_test), 5)

Y2_train = to_categorical(np.asarray(Y2_train), 10)
Y2_test = to_categorical(np.asarray(Y2_test), 10)
print (X1_train.shape, X1_test.shape)
print (Y1_train.shape, Y1_test.shape)
```

In this part of the code, the data is reshaped and prepared for use in training and testing. After separating the CIFAR-10 dataset into two sets, Set 1 and Set 2, as described earlier, the images and labels are converted into appropriate formats for the model.

The X1_train, X1_test, X2_train, and X2_test variables, which were initially lists of image data, are converted into numpy arrays using np.asarray. The reshape method is then used to ensure that each array has the correct dimensions of 32x32x3, which corresponds to the size of the images (32 pixels by 32 pixels with 3 color channels for RGB). The -1 in the reshape function automatically calculates the number of samples based on the total data size.

The labels for the datasets are also processed to make them compatible with the neural network. For Set 1, the labels are one-hot encoded using to_categorical with 5 classes, since Set 1 contains classes 0 through 4. Similarly, for Set 2, the labels are one-hot encoded with 10 classes, because Set 2 includes classes 5 through 9, but they are still represented in a 10-class format for consistency. One-hot encoding transforms the labels into vectors where the correct class is represented by a 1, and all other classes are represented by 0s. This format is required for classification tasks in neural networks.

Finally, the shapes of the training and testing data for Set 1 are printed. This is to verify that the data has been reshaped correctly and that the labels are in the expected one-hot encoded format. This step ensures that the input data is ready for training and testing the model, with the proper structure and compatibility for further processing.

```
split1 = int(0.8 * X1_train.shape[0])
split2 = int(0.8 * X2_train.shape[0])


x1_val = X1_train[split1:]
x1_train = X1_train[:split1]
```

```
y1_val = Y1_train[split1:]
y1_train = Y1_train[:split1]

x2_val = X2_train[split2:]
x2_train = X2_train[:split2]
y2_val = Y2_train[split2:]
y2_train = Y2_train[:split2]
```

This code splits the training datasets for Set 1 and Set 2 into training and validation sets. Splitting the data in this way ensures that part of the training data is held back for validation purposes, allowing the model's performance to be evaluated during training on unseen data.

First, the number of samples to be included in the training subset is calculated using int(0.8 * X1_train.shape[0]) for Set 1 and int(0.8 * X2_train.shape[0]) for Set 2. This takes 80% of the total training samples for each set, leaving 20% for validation.

For Set 1, the first 80% of the data (X1_train[:split1] and Y1_train[:split1]) is assigned to the training set (x1_train and y1_train). The remaining 20% (X1_train[split1:] and Y1_train[split1:]) is assigned to the validation set (x1_val and y1_val).

Similarly, for Set 2, the first 80% of the data (X2_train[:split2] and Y2_train[:split2]) becomes the training set (x2_train and y2_train), and the last 20% (X2_train[split2:] and Y2_train[split2:]) becomes the validation set (x2_val and y2_val).

This process ensures that the model can be trained on the majority of the data while being validated on a separate, unseen portion. This validation data is crucial for monitoring overfitting and ensuring that the model generalizes well to unseen data. The split ratios can also be adjusted if needed, but 80% for training and 20% for validation is a common choice.

```
start = datetime.datetime.now()
hist1 = model.fit(x1_train, y1_train,
         epochs=10,
         shuffle=True,
         batch_size=100,
         validation_data=(x1_val, y1_val), verbose=2)

time_taken = datetime.datetime.now() - start
print ('\n'*2, '-'*20, '\n')
print ('Time taken for first training: ', time_taken)
print ('\n', '-'*20, '\n'*2)
```

In this part of the code, the model is trained on the first dataset, Set 1. The training starts by recording the current time using datetime.datetime.now(). This is done so that the total time taken for training can be calculated later. The model.fit function is then used to train the model. It trains the model on the x1_train data and the corresponding labels y1_train for 10 epochs. The epochs parameter is set to 10, which means the model will go through the entire training data 10 times.

The shuffle=True makes sure the training data is shuffled during each epoch, which helps the model learn better by preventing it from seeing the data in the same order every time. The batch_size is set to 100, meaning the model processes 100 samples at a time before updating its weights. The validation data, x1_val and y1_val, is also provided to monitor the model's performance on unseen data during training. This is useful for checking if the model is overfitting.

The training progress is displayed using verbose=2, which gives detailed updates about each epoch and the training and validation losses. Once training is complete, the time taken is calculated by subtracting the start time from the current time. The total time taken for the training is printed along with some separators to make it clear and readable. This step helps in understanding how long the training process takes and provides a way to compare the time needed for different models or datasets.

```python
### Your code goes here ###
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense
from tensorflow.keras import Sequential

for layer in model.layers[:5]:
    layer.trainable = False

new_model = Sequential()

for layer in model.layers[:5]:
    new_model.add(layer)

new_model.add(Dense(128, activation='relu'))

new_model.add(Dense(10, activation='softmax'))

new_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

new_model.summary()
```

This code implements the convolutional neural network for training a model to classify classes 5 to 9 by leveraging the information gained from the previously trained model.

First, the code freezes the first five layers of the previous model. These layers include the convolutional and pooling layers, which have already learned useful features like edges and textures during training on Set 1 (classes 0 to 4). By setting trainable=False for these layers, their weights will not be updated during training, preserving the information learned earlier.

Next, a new model is created as a Sequential model. The first five layers from the previous model are added to this new model. These layers will act as the feature extractor for the new classification task. After these frozen layers, two new dense layers are added. The first dense layer has 128 neurons with ReLU activation, which allows the model to learn additional complex patterns from the extracted features. The final dense layer has 10 neurons with softmax activation to output probabilities for the 10 classes (5 to 9).

The model is compiled using the Adam optimizer, which adjusts the learning rate dynamically for efficient training. The loss function is set to categorical crossentropy, appropriate for multi-class classification problems. The metrics=['accuracy'] ensures that the accuracy is monitored during training and evaluation.

Finally, the new_model.summary() function displays the architecture of the new model. It will show that the first five layers are frozen (non-trainable) and the last two dense layers are trainable. This confirms that the model reuses the knowledge from the previous training while adapting to the new classification task for classes 5 to 9. This process demonstrates the concept of transfer learning, where a pre-trained model is repurposed for a new, related task.

```
start = datetime.datetime.now()
hist2 = trans_model.fit(x2_train, y2_train, epochs=10, shuffle=True, batch_size=100,
validation_data=(x2_val, y2_val), verbose=2)
time_taken = datetime.datetime.now() - start
print ('\n'*2, '-'*20, '\n')
print ('Time taken for final training: ', time_taken)
print ('\n', '-'*20, '\n'*2)
```

This code trains the new model, trans_model, on the dataset for classes 5 to 9 (Set 2). The training process begins by recording the current time using datetime.datetime.now() so that the total time taken for training can be calculated once the process is complete.

The fit function is used to train the model. The training data, x2_train and y2_train, is passed along with the validation data, x2_val and y2_val. The training is set to run for 10 epochs, meaning the model will process the entire training dataset 10 times. The data is shuffled at the start of each epoch using shuffle=True, which helps improve learning by preventing the model from seeing the data in the same order every time. The batch size is set to 100, meaning the model processes 100 samples at a time before updating the weights.

The verbose=2 parameter ensures that detailed logs of the training progress are displayed, including the loss and accuracy for both training and validation data after each epoch. Once the training is complete, the time taken is calculated by subtracting the start time from the current time. This value is printed along with separators to clearly indicate the total training time for the new model.

This process evaluates how well the new model, which utilizes the frozen layers from the previously trained model, adapts to the new task of classifying classes 5 to 9. It also provides an idea of the time efficiency of training with transfer learning compared to training a model from scratch.

## Explain what we did with transfer learning

With transfer learning, we reused the first five layers of the previous model, trained on classes 0 to 4, for the new task of classifying classes 5 to 9. These layers, which learned general features like edges and textures, were frozen to preserve their knowledge.

We added new trainable layers: a dense layer with 128 neurons and ReLU activation, and a final dense layer with 10 neurons and softmax activation for classification. This allowed the model to adapt to the new task while starting with a strong foundation, reducing training time and improving efficiency compared to training from scratch.