BBM418 Computer Vision Laboratory
Assignment 3 - Image Classification with CNNs

Kayla Akyüz
21726914
Department of Computer Engineering
Hacettepe University
Ankara, Turkey
b21726914@cs.hacettepe.edu.tr

Overview

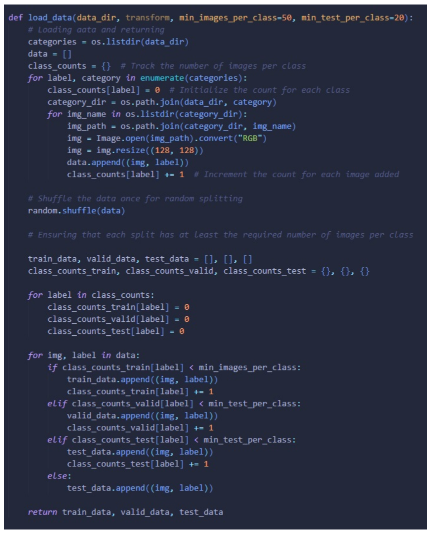
The key point of this assignment is getting familiar with CNNs by completing various tasks.

There are two parts, first is training different structures of CNNs and second is transfer learning. We can use PYTorch and other libraries.

I have created a notebook file! With in there are results, plots and explanations. To ease viewing redundant and big functions are bundled in kayla_tools.py. Please review my notebook file as it is created with care to be representable. I added as much as code screen shots here but you can easily browse them in the notebook.

Dataset

As explained in notebook, I had to maximize performance so I resized pictures to 64x64 sadly. Then also I need to minimize training and validation so I created a function that fills these sets with the minimum requirement which is 50 for each class for training set and 20 for each class in validation set. Rest is in training set as it is least performance impacting. So the the amount is: 400, 160, 229. The function simply reads the image and puts it in training or validation or test in that order and increases the increment, if count is sufficient the next set is filled.



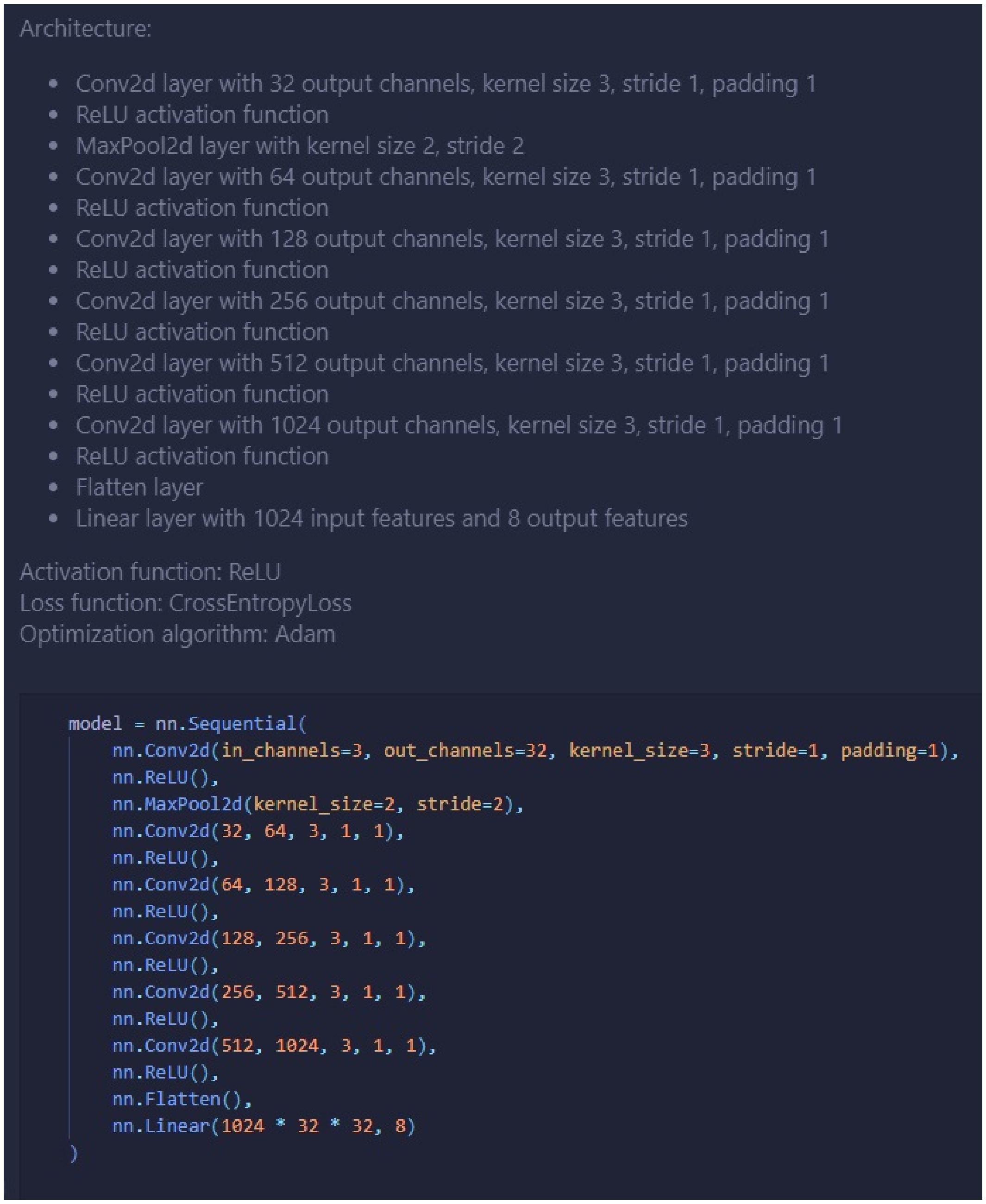
```python
def load_data(data_dir, transform, min_images_per_class=50, min_test_per_class=20):
    # Loading aata and returning
    categories = os.listdir(data_dir)
    data = []
    class_counts = {}  # Track the number of images per class
    for label, category in enumerate(categories):
        class_counts[label] = 0  # Initialize the count for each class
        category_dir = os.path.join(data_dir, category)
        for img_name in os.listdir(category_dir):
            img_path = os.path.join(category_dir, img_name)
            img = Image.open(img_path).convert("RGB")
            img = img.resize((128, 128))
            data.append((img, label))
            class_counts[label] += 1  # Increment the count for each image added

    # Shuffle the data once for random splitting
    random.shuffle(data)

    # Ensuring that each split has at least the required number of images per class

    train_data, valid_data, test_data = [], [], []
    class_counts_train, class_counts_valid, class_counts_test = {}, {}, {}

    for label in class_counts:
        class_counts_train[label] = 0
        class_counts_valid[label] = 0
        class_counts_test[label] = 0

    for img, label in data:
        if class_counts_train[label] < min_images_per_class:
            train_data.append((img, label))
            class_counts_train[label] += 1
        elif class_counts_valid[label] < min_test_per_class:
            valid_data.append((img, label))
            class_counts_valid[label] += 1
        elif class_counts_test[label] < min_test_per_class:
            test_data.append((img, label))
            class_counts_test[label] += 1
        else:
            test_data.append((img, label))

    return train_data, valid_data, test_data
```

Part 1 - Modeling and Training a CNN classifier from Scratch

Creating the normal CNN (without residual) was easy, I added layers according to instructions. For the CNN with residual layers I used class structure to ease adding residual blocks.

The explanations and plots are also in the notebook.

Here is CNN without residual:



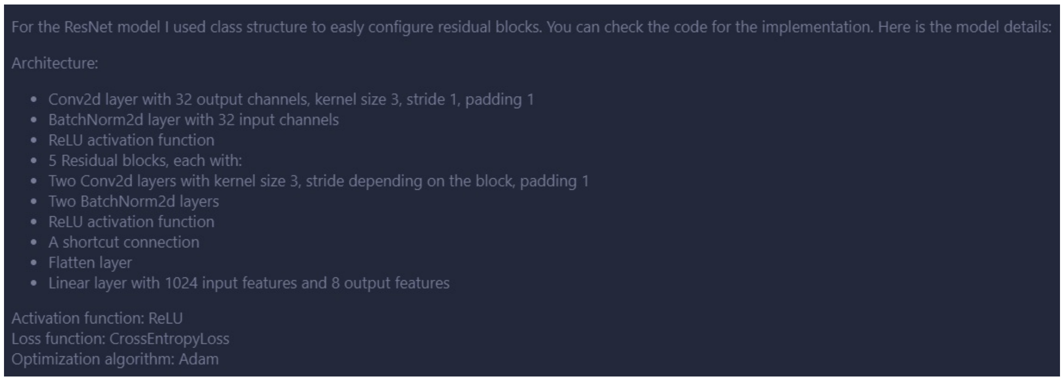
Architecture:

- Conv2d layer with 32 output channels, kernel size 3, stride 1, padding 1
- ReLU activation function
- MaxPool2d layer with kernel size 2, stride 2
- Conv2d layer with 64 output channels, kernel size 3, stride 1, padding 1
- ReLU activation function
- Conv2d layer with 128 output channels, kernel size 3, stride 1, padding 1
- ReLU activation function
- Conv2d layer with 256 output channels, kernel size 3, stride 1, padding 1
- ReLU activation function
- Conv2d layer with 512 output channels, kernel size 3, stride 1, padding 1
- ReLU activation function
- Conv2d layer with 1024 output channels, kernel size 3, stride 1, padding 1
- ReLU activation function
- Flatten layer
- Linear layer with 1024 input features and 8 output features

Activation function: ReLU
Loss function: CrossEntropyLoss
Optimization algorithm: Adam

```
model = nn.Sequential(
    nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, stride=1, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Conv2d(32, 64, 3, 1, 1),
    nn.ReLU(),
    nn.Conv2d(64, 128, 3, 1, 1),
    nn.ReLU(),
    nn.Conv2d(128, 256, 3, 1, 1),
    nn.ReLU(),
    nn.Conv2d(256, 512, 3, 1, 1),
    nn.ReLU(),
    nn.Conv2d(512, 1024, 3, 1, 1),
    nn.ReLU(),
    nn.Flatten(),
    nn.Linear(1024 * 32 * 32, 8)
)
```

Here is the code for residual CNN:

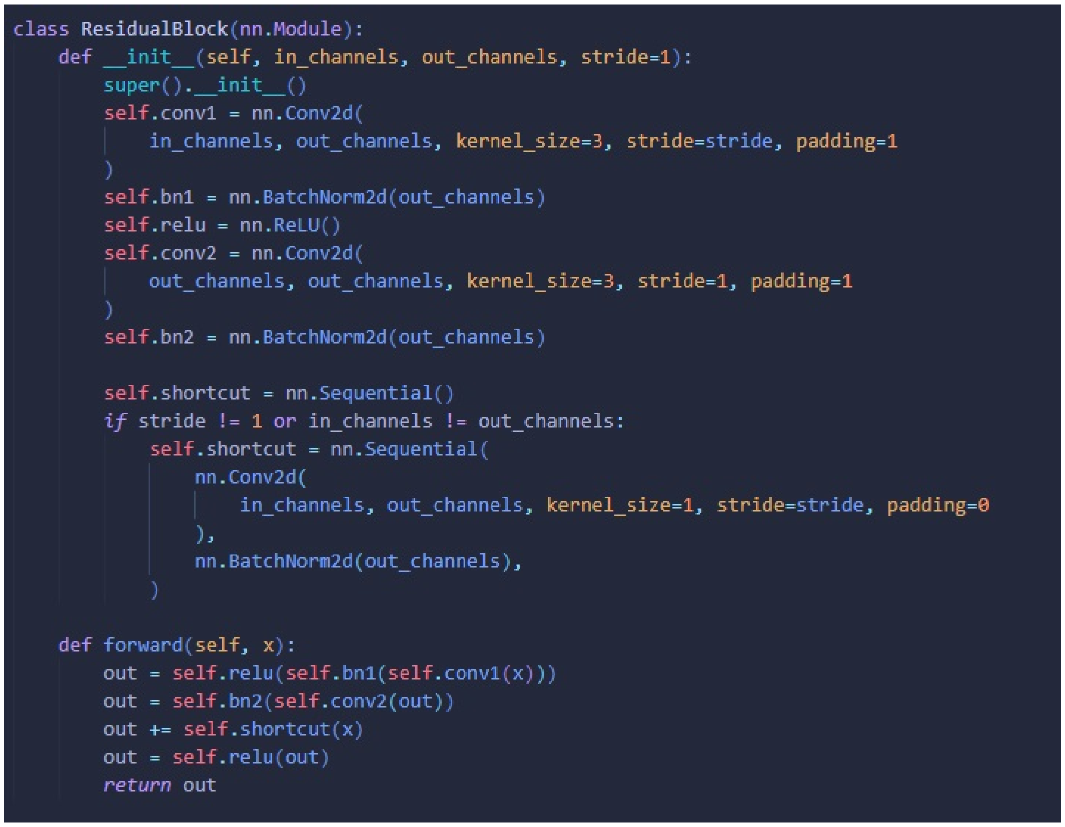


For the ResNet model I used class structure to easly configure residual blocks. You can check the code for the implementation. Here is the model details:

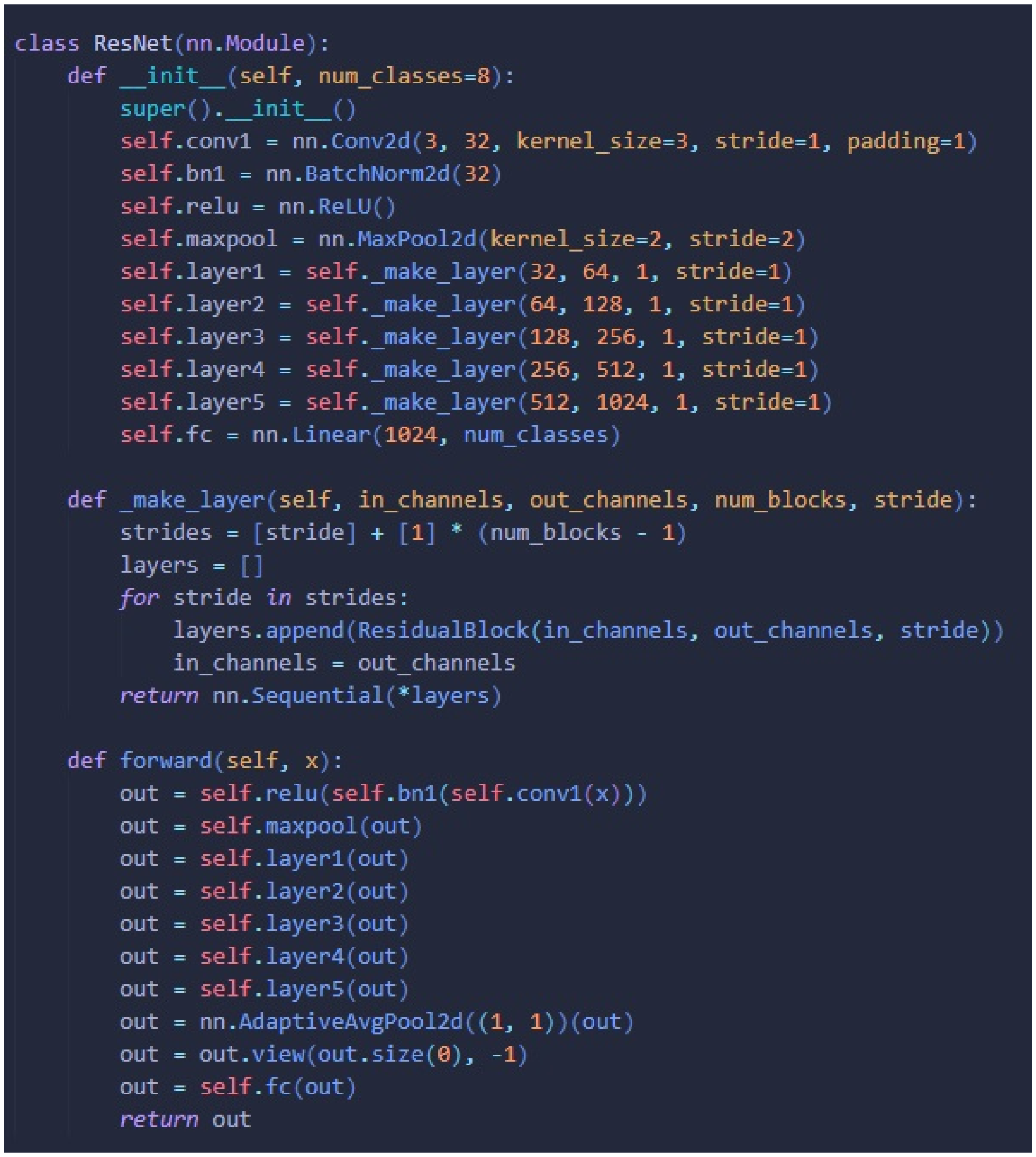
Architecture:

- Conv2d layer with 32 output channels, kernel size 3, stride 1, padding 1
- BatchNorm2d layer with 32 input channels
- ReLU activation function
- 5 Residual blocks, each with:
- Two Conv2d layers with kernel size 3, stride depending on the block, padding 1
- Two BatchNorm2d layers
- ReLU activation function
- A shortcut connection
- Flatten layer
- Linear layer with 1024 input features and 8 output features

Activation function: ReLU
Loss function: CrossEntropyLoss
Optimization algorithm: Adam



```python
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()
        self.conv1 = nn.Conv2d(
            in_channels, out_channels, kernel_size=3, stride=stride, padding=1
        )
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv2d(
            out_channels, out_channels, kernel_size=3, stride=1, padding=1
        )
        self.bn2 = nn.BatchNorm2d(out_channels)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(
                    in_channels, out_channels, kernel_size=1, stride=stride, padding=0
                ),
                nn.BatchNorm2d(out_channels),
            )

    def forward(self, x):
        out = self.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = self.relu(out)
        return out
```



```python
class ResNet(nn.Module):
    def __init__(self, num_classes=8):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.relu = nn.ReLU()
        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.layer1 = self._make_layer(32, 64, 1, stride=1)
        self.layer2 = self._make_layer(64, 128, 1, stride=1)
        self.layer3 = self._make_layer(128, 256, 1, stride=1)
        self.layer4 = self._make_layer(256, 512, 1, stride=1)
        self.layer5 = self._make_layer(512, 1024, 1, stride=1)
        self.fc = nn.Linear(1024, num_classes)

    def _make_layer(self, in_channels, out_channels, num_blocks, stride):
        strides = [stride] + [1] * (num_blocks - 1)
        layers = []
        for stride in strides:
            layers.append(ResidualBlock(in_channels, out_channels, stride))
            in_channels = out_channels
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.relu(self.bn1(self.conv1(x)))
        out = self.maxpool(out)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = self.layer5(out)
        out = nn.AdaptiveAvgPool2d((1, 1))(out)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out
```

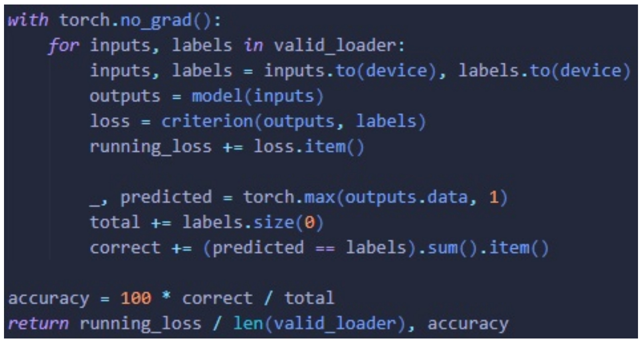
1. Draw a graph of loss and accuracy change for two different learning rates and two batch sizes:

The evolutions are done with:

learning_rates = [0.0001, 0.001]
batch_sizes = [16, 32]

I created evaluation functions to run the tests. While running the test the model with the best parameters and it's state dictionary are stored. This is returned so in the next step we got best model. Then new models with drop out layers are created and these weights from the best model are copied. Then the drop out models are tested and the result is printed as well. The best drop out model's confusion matrix is also printed. These are all in notebook file.

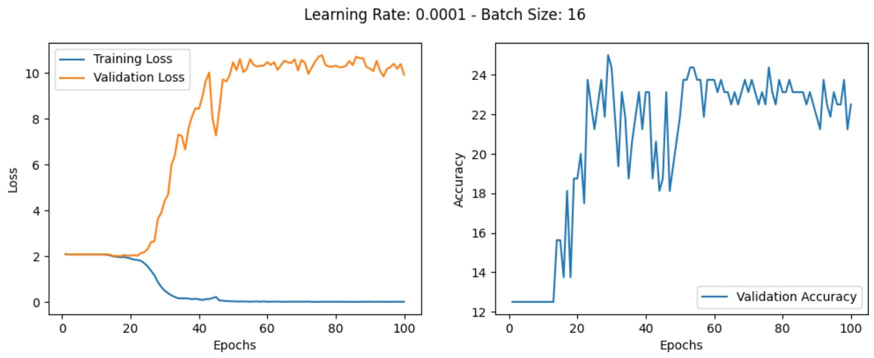
Here is my accuracy function which simply divides the true prediction to total percentage:

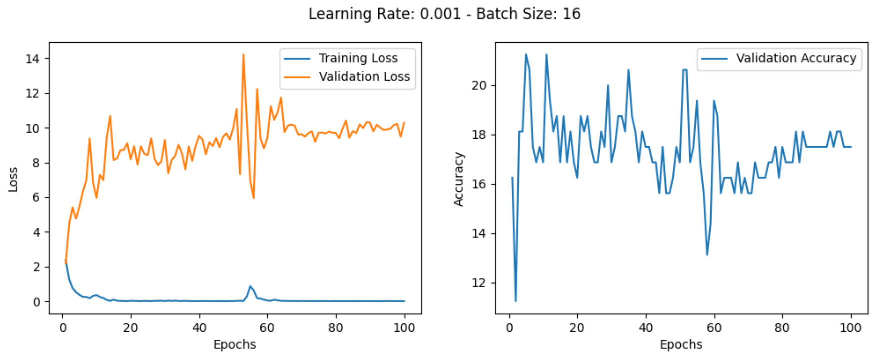


```
with torch.no_grad():
    for inputs, labels in valid_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        running_loss += loss.item()

        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

accuracy = 100 * correct / total
return running_loss / len(valid_loader), accuracy
```

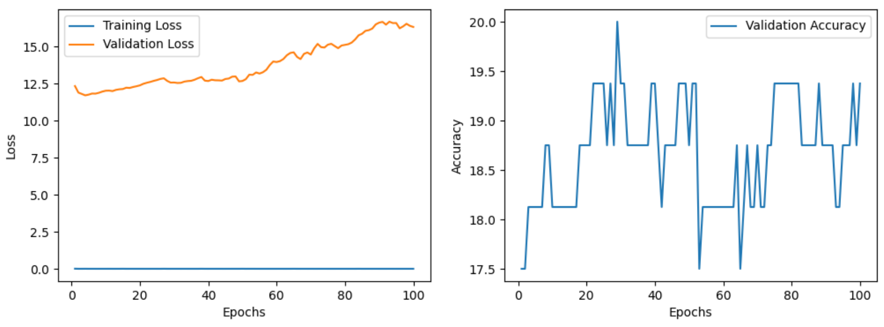
I also used accuracy_score from sklearn.metrics.

Here are the graphs starting from model without residual:

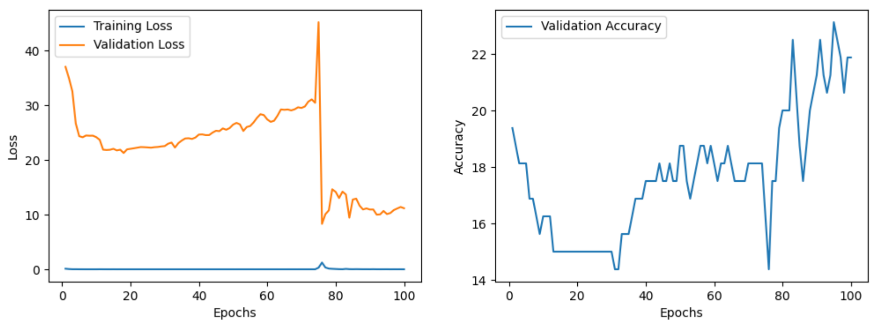




Learning Rate: 0.0001 - Batch Size: 32

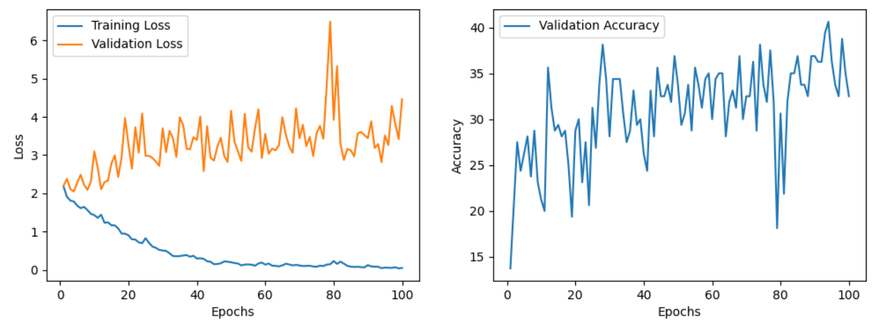


Learning Rate: 0.001 - Batch Size: 32

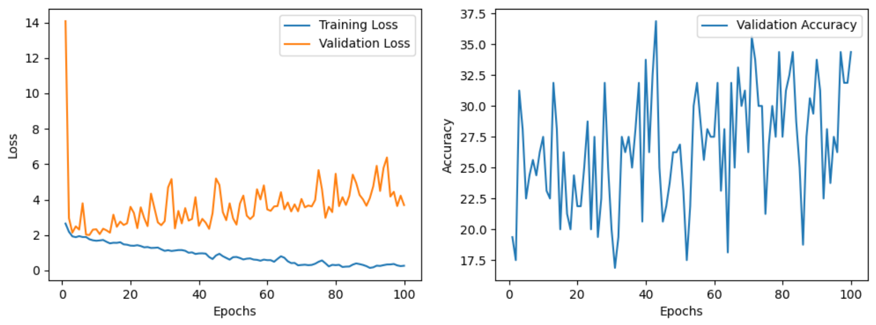


Now moving on for the one with residual blocks:

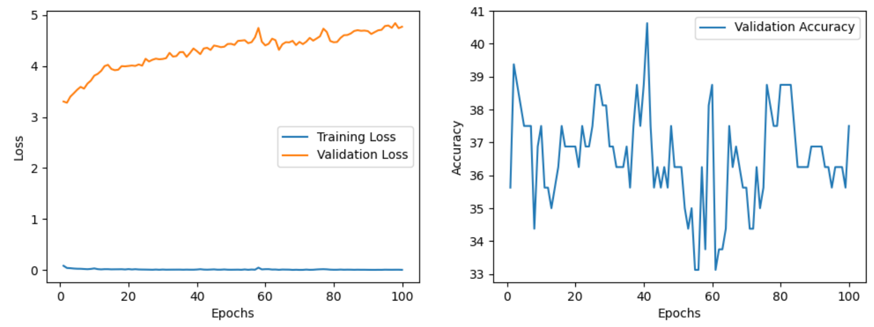
Learning Rate: 0.0001 - Batch Size: 16



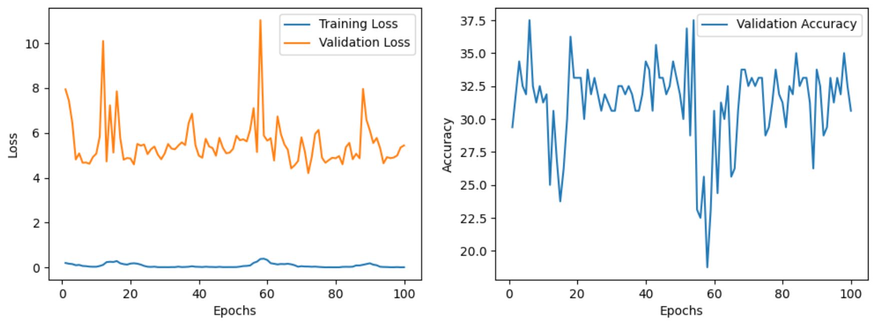
Learning Rate: 0.001 - Batch Size: 16



Learning Rate: 0.0001 - Batch Size: 32

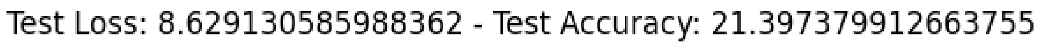


Learning Rate: 0.001 - Batch Size: 32

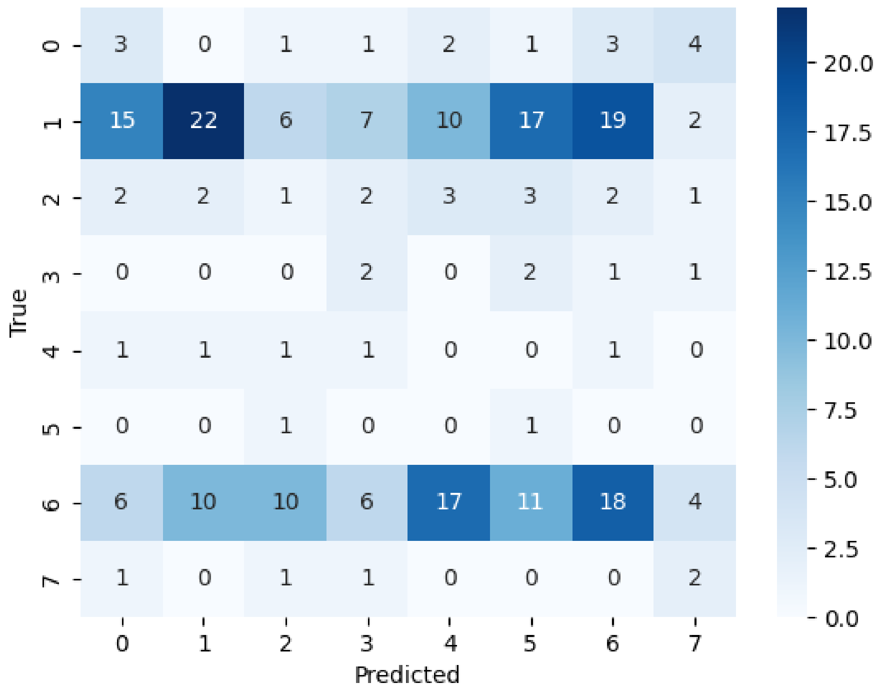


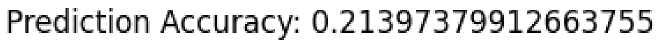
2. Select your best model with respect to validation accuracy and give test accuracy result:

Here is the confusion matrix for my best model's predictions, first non residual then residual model:



```
Test Loss: 8.629130585988362 - Test Accuracy: 21.397379912663755
```

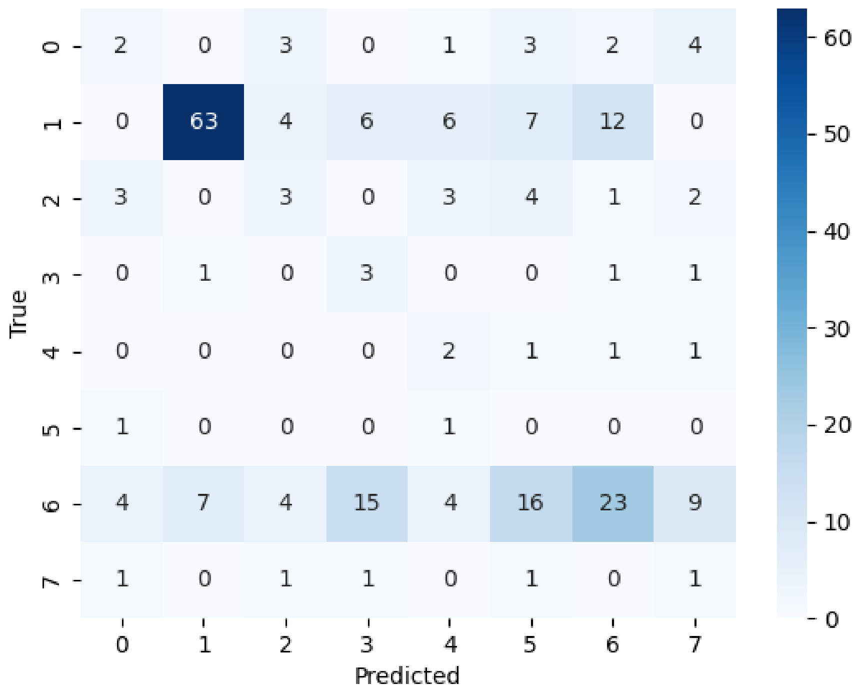




```
Prediction Accuracy: 0.21397379912663755
```

Best residual model:

```
Test Loss: 5.252290606498718 - Test Accuracy: 42.35807860262009
```



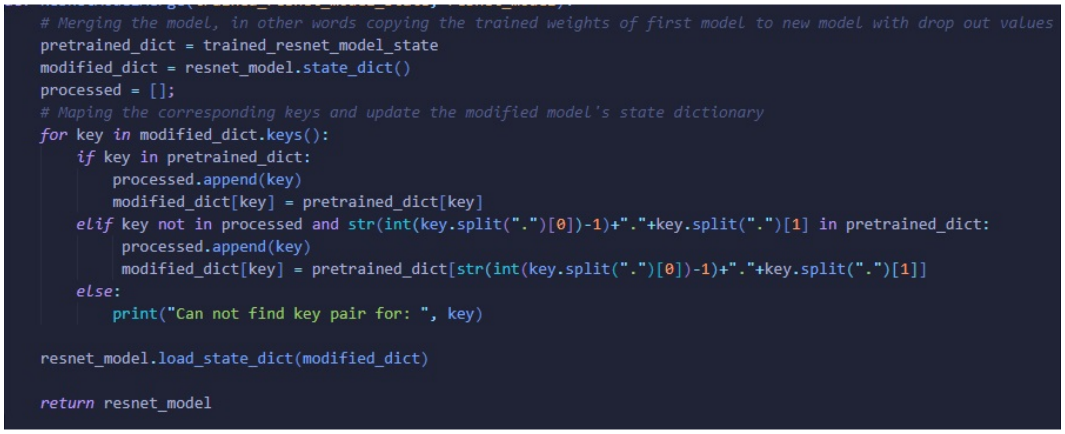
```
Prediction Accuracy: 0.42358078602620086
```

3. Integrate dropout to your best models (best model should be selected with respect to best validation accuracy. You need to integrate to both of your best models). In which part of the network you add dropout and why? Explore four different dropout values and give new validation and test accuracies:

I researched where to add drop out and it was "used on each of the fully connected (dense) layers before the output; it was not used on the convolution layers."[4], according to this in my code for the non residual model I add before the second convolution layer.

For the residual model, I created new classes with drop out layers applied after the first convolution layer and the first batch normalization layer. So overall, drop out layers are added after the first convolution layer in each ResidualBlockDropout and applied throughout the layer1 to layer5 sections of the ResNetDropout model.

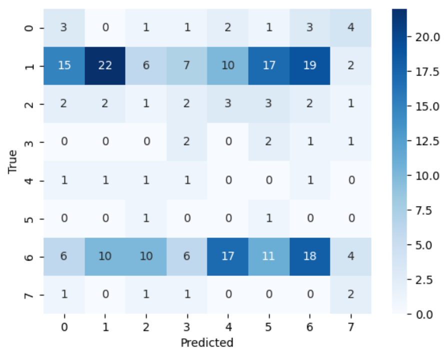
How the results are implemented is, I first create a new model with drop out layer, then transfer the weights, after that I print out results, then do training of 100 epoch and print results again. Because I wanted to get meaningful results and to analyse drop out layer better I first give confusion matrix and values right after inserting drop out layer, then I train for 100 epoch and give confusion matrix and values again. This is done for all 4 drop out rates and 2 models. So total of 16 confusion matrices provided.



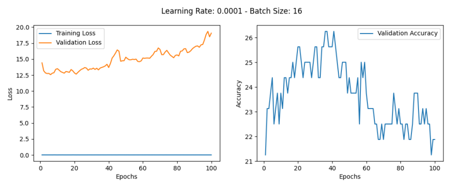
```
    # Merging the model, in other words copying the trained weights of first model to new model with drop out values
    pretrained_dict = trained_resnet_model_state
    modified_dict = resnet_model.state_dict()
    processed = [];
    # Maping the corresponding keys and update the modified model's state dictionary
    for key in modified_dict.keys():
        if key in pretrained_dict:
            processed.append(key)
            modified_dict[key] = pretrained_dict[key]
        elif key not in processed and str(int(key.split(".")[0])-1)+"."+key.split(".")[1] in pretrained_dict:
             processed.append(key)
             modified_dict[key] = pretrained_dict[str(int(key.split(".")[0])-1)+"."+key.split(".")[1]]
        else:
            print("Can not find key pair for: ", key)

    resnet_model.load_state_dict(modified_dict)

    return resnet_model
```

Here is the results of four different drop out values and give new validation and test accuracies and the confusion matrix also the results after training drop out model 100 epoch, for the training the best model's learning rate and batch size used:

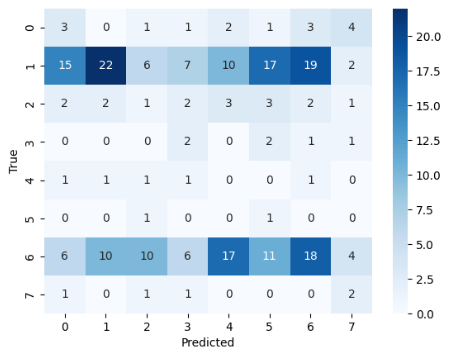
Details For Dropout Value: 0.01
Validation Loss: 11.17989149093628 - Validation Accuracy: 21.875
Test Loss: 8.629130585988362 - Test Accuracy: 21.397379912663755
The Accuracy: 0.21397379912663755



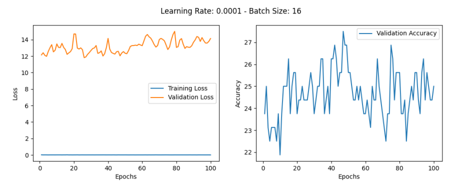
	0	1	2	3	4	5	6	7
0	3	0	1	1	2	1	3	4
1	15	22	6	7	10	17	19	2
2	2	2	1	2	3	3	2	1
3	0	0	0	2	0	2	1	1
4	1	1	1	1	0	0	1	0
5	0	0	1	0	0	1	0	0
6	6	10	10	6	17	11	18	4
7	1	0	1	1	0	0	0	2



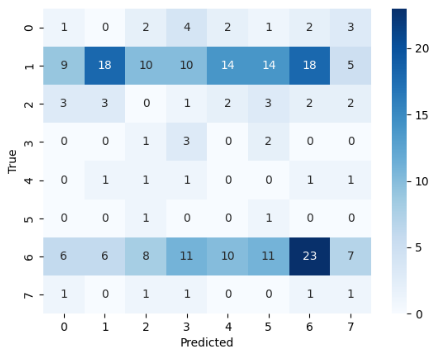
Details For Dropout Value: 0.1
Validation Loss: 11.17989149093628 - Validation Accuracy: 21.875
Test Loss: 8.629130585988362 - Test Accuracy: 21.397379912663755
The Accuracy: 0.21397379912663755



	0	1	2	3	4	5	6	7
0	3	0	1	1	2	1	3	4
1	15	22	6	7	10	17	19	2
2	2	2	1	2	3	3	2	1
3	0	0	0	2	0	2	1	1
4	1	1	1	1	0	0	1	0
5	0	0	1	0	0	1	0	0
6	6	10	10	6	17	11	18	4
7	1	0	1	1	0	0	0	2

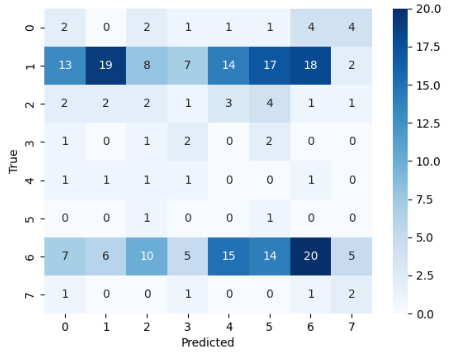


Details For After Training Dropout Value: 0.01
Validation Loss: 19.059549236297606 - Validation Accuracy: 21.875
Test Loss: 14.759876696268718 - Test Accuracy: 20.52401746724890
The Accuracy: 0.2052401746724891



	0	1	2	3	4	5	6	7
0	1	0	2	4	2	1	2	3
1	9	18	10	10	14	14	18	5
2	3	3	0	1	2	3	2	2
3	0	0	1	3	0	2	0	0
4	0	1	1	1	0	0	1	1
5	0	0	1	0	0	1	0	0
6	6	6	8	11	10	11	23	7
7	1	0	1	1	0	0	1	1

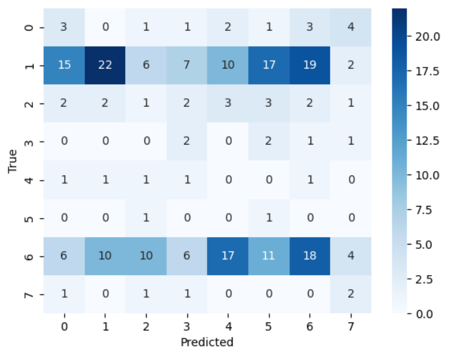
Details For After Training Dropout Value: 0.1
Validation Loss: 14.141003131866455 - Validation Accuracy: 25.0
Test Loss: 10.974254703521728 - Test Accuracy: 20.96069868995633
The Accuracy: 0.2096069868995633



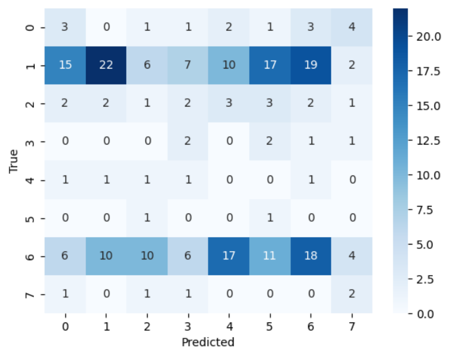
	0	1	2	3	4	5	6	7
0	2	0	2	1	1	1	4	4
1	13	19	8	7	14	17	18	2
2	2	2	2	1	3	4	1	1
3	1	0	1	2	0	2	0	0
4	1	1	1	1	0	0	1	0
5	0	0	1	0	0	1	0	0
6	7	6	10	5	15	14	20	5
7	1	0	0	1	0	0	1	2

Details For Dropout Value: 0.25
Validation Loss: 11.17989149093628 - Validation Accuracy: 21.875
Test Loss: 8.629130585988362 - Test Accuracy: 21.397379912663755
The Accuracy: 0.21397379912663755

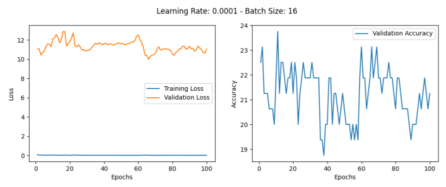
Details For Dropout Value: 0.5
Validation Loss: 11.17989149093628 - Validation Accuracy: 21.875
Test Loss: 8.629130585988362 - Test Accuracy: 21.397379912663755
The Accuracy: 0.21397379912663755

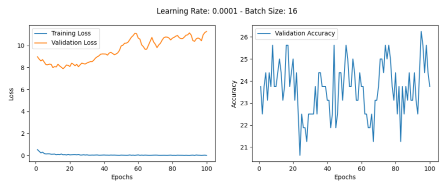


	0	1	2	3	4	5	6	7
0	3	0	1	1	2	1	3	4
1	15	22	6	7	10	17	19	2
2	2	2	1	2	3	3	2	1
3	0	0	0	2	0	2	1	1
4	1	1	1	1	0	0	1	0
5	0	0	1	0	0	1	0	0
6	6	10	10	6	17	11	18	4
7	1	0	1	1	0	0	0	2



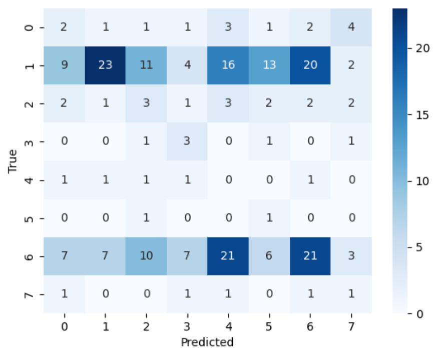
	0	1	2	3	4	5	6	7
0	3	0	1	1	2	1	3	4
1	15	22	6	7	10	17	19	2
2	2	2	1	2	3	3	2	1
3	0	0	0	2	0	2	1	1
4	1	1	1	1	0	0	1	0
5	0	0	1	0	0	1	0	0
6	6	10	10	6	17	11	18	4
7	1	0	1	1	0	0	0	2



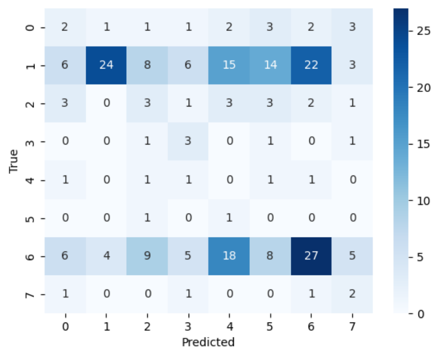


Details For After Training Dropout Value: 0.25
Validation Loss: 11.068023777008056 - Validation Accuracy: 21.25
Test Loss: 8.750902016957602 - Test Accuracy: 23.580786026200872
The Accuracy: 0.23580786026200873

Details For After Training Dropout Value: 0.5
Validation Loss: 11.275317335128785 - Validation Accuracy: 23.75
Test Loss: 8.657368914286296 - Test Accuracy: 26.637554585152838
The Accuracy: 0.2663755458515284



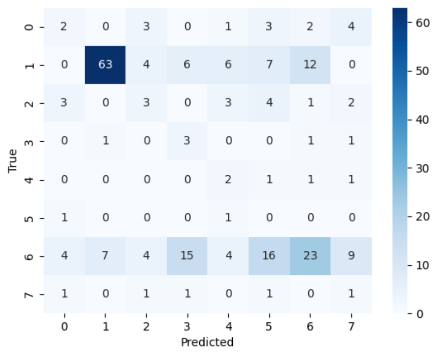
	0	1	2	3	4	5	6	7
0	2	1	1	1	3	1	2	4
1	9	23	11	4	16	13	20	2
2	2	1	3	1	3	2	2	2
3	0	0	1	3	0	1	0	1
4	1	1	1	1	0	0	1	0
5	0	0	1	0	0	1	0	0
6	7	7	10	7	21	6	21	3
7	1	0	0	1	1	0	1	1



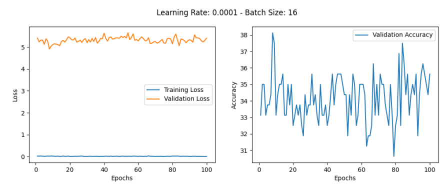
	0	1	2	3	4	5	6	7
0	2	1	1	1	2	3	2	3
1	6	24	8	6	15	14	22	3
2	3	0	3	1	3	3	2	1
3	0	0	1	3	0	1	0	1
4	1	0	1	1	0	1	1	0
5	0	0	1	0	1	0	0	0
6	6	4	9	5	18	8	27	5
7	1	0	0	1	0	0	1	2

The drop out graphs for the model with residual blocks, again there are matrices right after adding the drop out layer and then training 100 epoch:

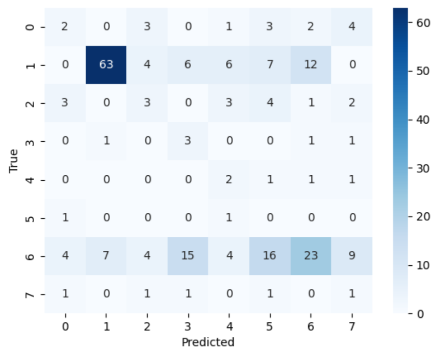
For Dropout Value: 0.01
Validation Loss: 5.441906762123108 - Validation Accuracy: 30.625
Test Loss: 5.252290606498718 - Test Accuracy: 42.35807860262009
The Accuracy: 0.42358078602620086



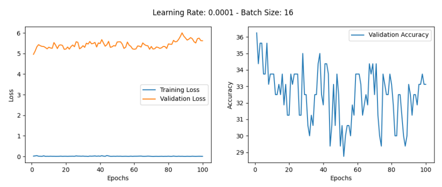
	0	1	2	3	4	5	6	7
0	2	0	3	0	1	3	2	4
1	0	63	4	6	6	7	12	0
2	3	0	3	0	3	4	1	2
3	0	1	0	3	0	0	1	1
4	0	0	0	0	2	1	1	1
5	1	0	0	0	1	0	0	0
6	4	7	4	15	4	16	23	9
7	1	0	1	1	0	1	0	1



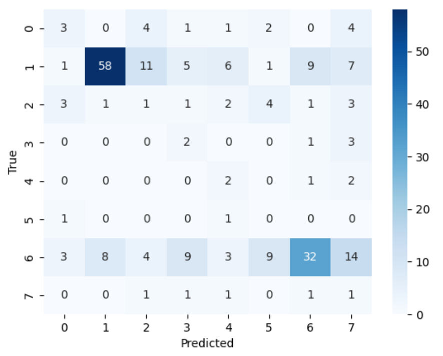
For Dropout Value: 0.1
Validation Loss: 5.441906762123108 - Validation Accuracy: 30.625
Test Loss: 5.252290606498718 - Test Accuracy: 42.35807860262009
The Accuracy: 0.42358078602620086



	0	1	2	3	4	5	6	7
0	2	0	3	0	1	3	2	4
1	0	63	4	6	6	7	12	0
2	3	0	3	0	3	4	1	2
3	0	1	0	3	0	0	1	1
4	0	0	0	0	2	1	1	1
5	1	0	0	0	1	0	0	0
6	4	7	4	15	4	16	23	9
7	1	0	1	1	0	1	0	1

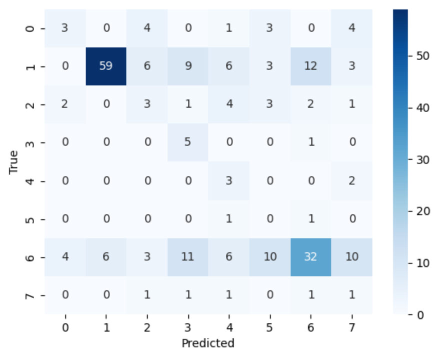


Details For After Training Dropout Value: 0.01
Validation Loss: 5.406956386566162 - Validation Accuracy: 35.625
Test Loss: 5.256799554824829 - Test Accuracy: 43.23144104803494
The Accuracy: 0.43231441048034935



	0	1	2	3	4	5	6	7
0	3	0	4	1	1	2	0	4
1	1	58	11	5	6	1	9	7
2	3	1	1	1	2	4	1	3
3	0	0	0	2	0	0	1	3
4	0	0	0	0	2	0	1	2
5	1	0	0	0	1	0	0	0
6	3	8	4	9	3	9	32	14
7	0	0	1	1	1	0	1	1

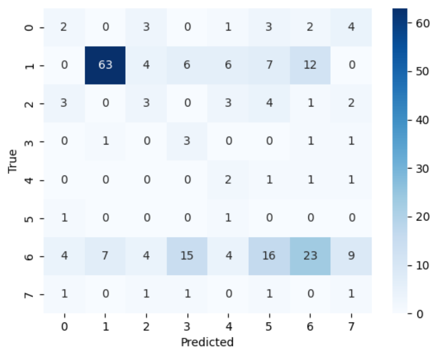
Details For After Training Dropout Value: 0.1
Validation Loss: 5.62065737247467 - Validation Accuracy: 33.125
Test Loss: 5.007287915547689 - Test Accuracy: 46.2882096069869
The Accuracy: 0.462882096069869

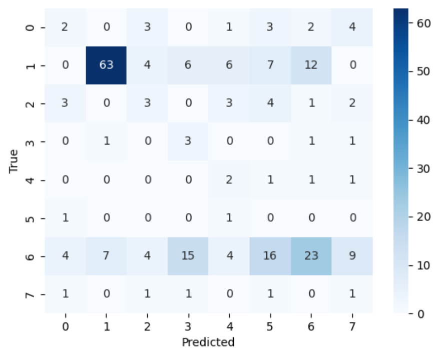


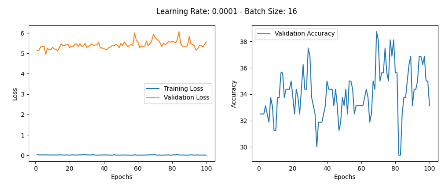
	0	1	2	3	4	5	6	7
0	3	0	4	0	1	3	0	4
1	0	59	6	9	6	3	12	3
2	2	0	3	1	4	3	2	1
3	0	0	0	5	0	0	1	0
4	0	0	0	0	3	0	0	2
5	0	0	0	0	1	0	1	0
6	4	6	3	11	6	10	32	10
7	0	0	1	1	1	0	1	1

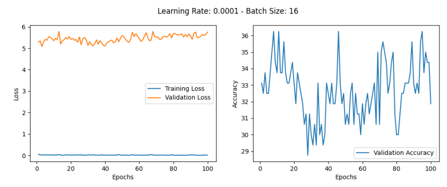
For Dropout Value: 0.25
Validation Loss: 5.441906762123108 - Validation Accuracy: 30.625
Test Loss: 5.252290606498718 - Test Accuracy: 42.35807860262009
The Accuracy: 0.42358078602620086

For Dropout Value: 0.5
Validation Loss: 5.441906762123108 - Validation Accuracy: 30.625
Test Loss: 5.252290606498718 - Test Accuracy: 42.35807860262009
The Accuracy: 0.42358078602620086



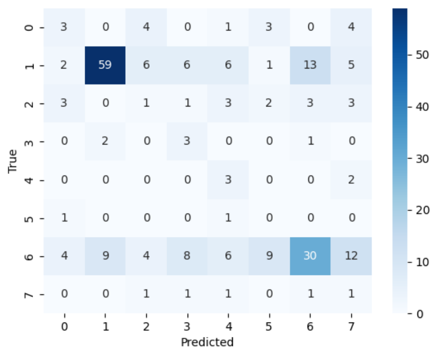


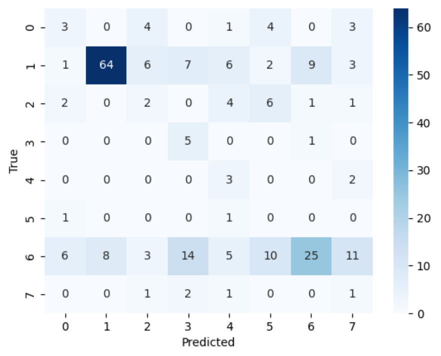




Details For After Training Dropout Value: 0.25
Validation Loss: 5.557058358192444 - Validation Accuracy: 33.125
Test Loss: 5.18019007841746 - Test Accuracy: 43.66812227074236
The Accuracy: 0.4366812227074236

Details For After Training Dropout Value: 0.5
Validation Loss: 5.76018123626709 - Validation Accuracy: 31.875
Test Loss: 5.598297627766927 - Test Accuracy: 44.97816593886463
The Accuracy: 0.4497816593886463



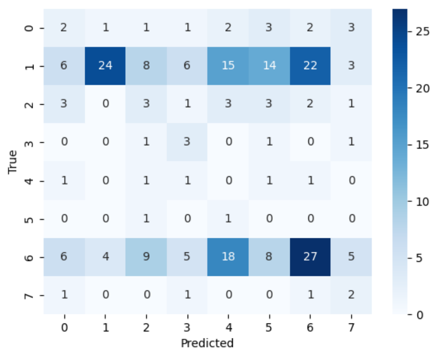


4. Plot a confusion matrix for your best model's predictions:

Best models are below:

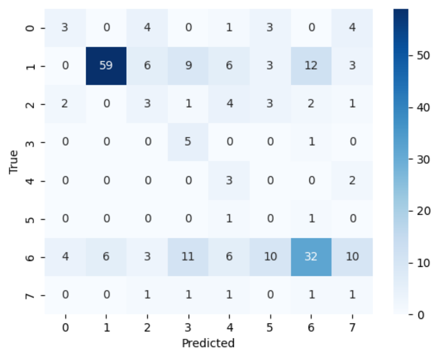
```
Best model stats for CNN:
```

```
The Accuracy: 0.2663755458515284
```



```
Best model stats for residual CNN:
```

```
The Accuracy: 0.462882096069869
```



5. Explain and analyze your findings and results:

Before I start the explanation I urge you to check the notebook for better representation and detailed results. It would be better to understand the implementation

My explanation and analysis is:

We can clearly see there is a tiny peak at loss when the epoch reaches 90, that is because of over fitting. Considering learning rate, 0.001 seems to be giving better results as 0.0001 gets linear increase in loss and can not get good loss and accuracy for validation. That is also because of over fitting. However for the end result the higher learning rate seems to be getting better validation accuracy. So the 0.0001 learning rate seems to be giving better accuracy.

Considering batch size, higher batch size also seems to be fluctuating a lot while lower batch size, 16, is giving more stable improvement over the epoch. However the peaks of 32 batch size seems to be able to hit higher accuracies.

Considering residual blocks, it is obvious that residual model is getting at least double the accuracy thanks to the residual layers.

Now, regarding drop out, it is a regularization technique commonly used in neural networks to prevent over fitting. Drop out randomly sets a fraction of input units to zero during training, effectively "dropping out" those units from the network. In my results the drop outs seem to be resulting in the increased accuracy however I am not sure if that means loss of patterns in data, this might be due that a higher drop out rate and it's position which is right after the first ReLU layer. What I mean is, my test set has more of some classes so the accuracy being higher might mean over fitting those two layers, still for sure it is not over fitting one class as the accuracy for both classes and all other classes increase as well.

So it seems there is no over fitting with drop out, in contrary it prevents over fitting and makes test accuracy higher.

I tested with many drop out vales and positions. So I will comment on the results, it seems changing the drop out value corresponds to loss of data about different classes and the magnitude of drop out increases. Putting the drop out layer at the end results on less loss of data while putting it on the top results in bigger changes.

Increasing the drop out rate increases gain towards the accuracy however increasing it too much makes model's predict wrong.

Also adding multiple drop out layers increases the magnitude of effect.

We can see the performance for the model with residual blocks is generally better and it follow to trends explained about the learning rate and batch size and drop out.

Part 2 - Transfer Learning with CNNs

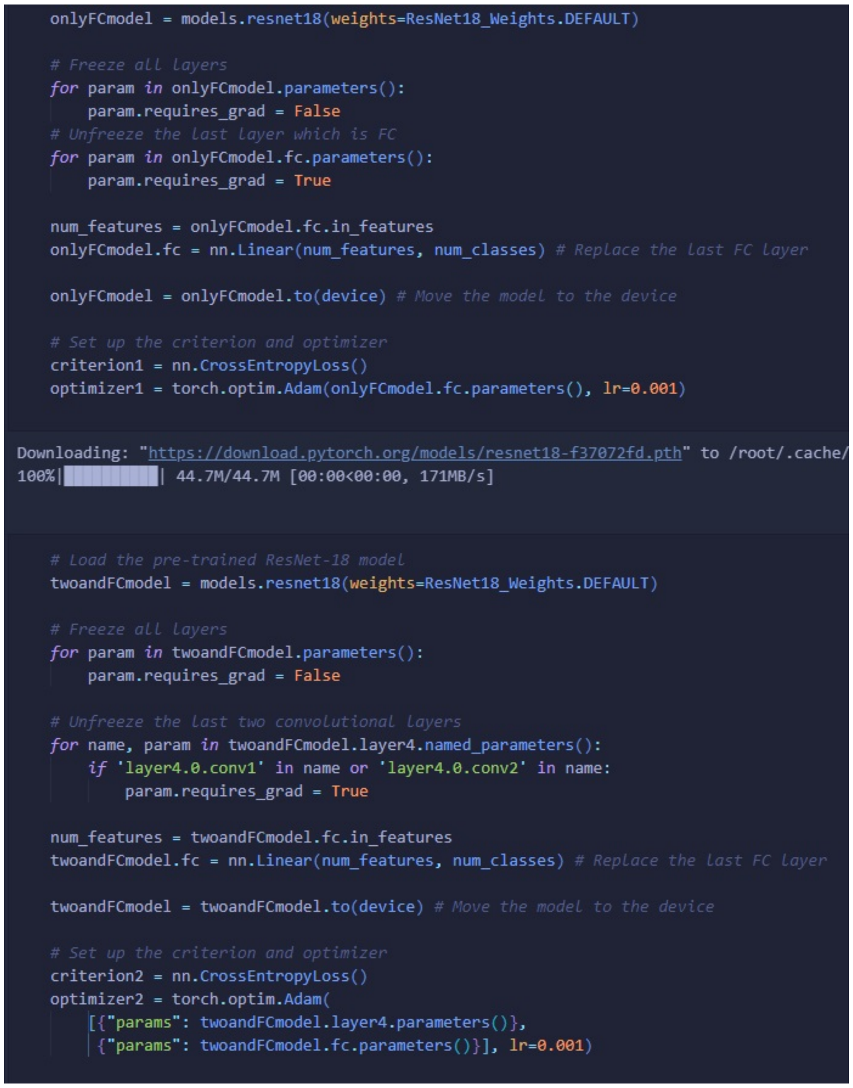
1. What is fine-tuning? Why should we do this? Why do we freeze the rest and train only FC layers? Give your explanation in detail with relevant code snippet:

Fine tuning is a process commonly used in transfer learning, where a pre-trained neural network model is further trained on a new task or dataset. This is due the computational requirements to train a model and trying to ease it. For example there are amazing models with huge training in image generation or image detection which has the features of general images. We fine tune on our dataset.

We do this to reduce the training time and power required by training a general model. Then we train it on our dataset to fine tune the model.

When performing fine-tuning, it is common to freeze the weights of the early layers of the pre-trained model while only training the FC layers or the final layers specific to the new task. This is to keep the trained weight and prevent over fitting.

Here is my code:



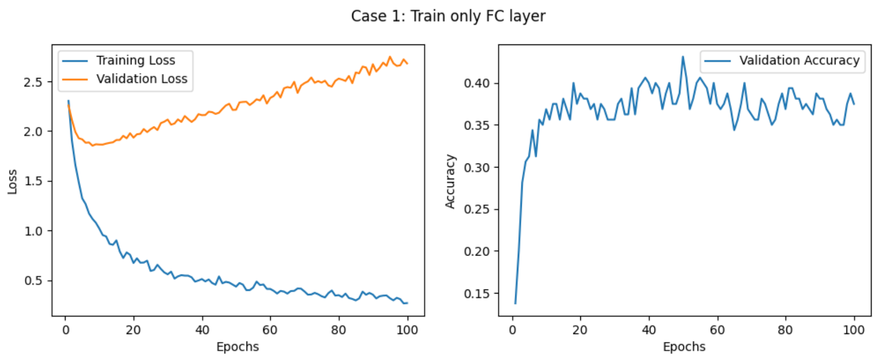
```
onlyFCmodel = models.resnet18(weights=ResNet18_Weights.DEFAULT)

# Freeze all layers
for param in onlyFCmodel.parameters():
    param.requires_grad = False
# Unfreeze the last layer which is FC
for param in onlyFCmodel.fc.parameters():
    param.requires_grad = True

num_features = onlyFCmodel.fc.in_features
onlyFCmodel.fc = nn.Linear(num_features, num_classes) # Replace the last FC layer

onlyFCmodel = onlyFCmodel.to(device) # Move the model to the device

# Set up the criterion and optimizer
criterion1 = nn.CrossEntropyLoss()
optimizer1 = torch.optim.Adam(onlyFCmodel.fc.parameters(), lr=0.001)
```

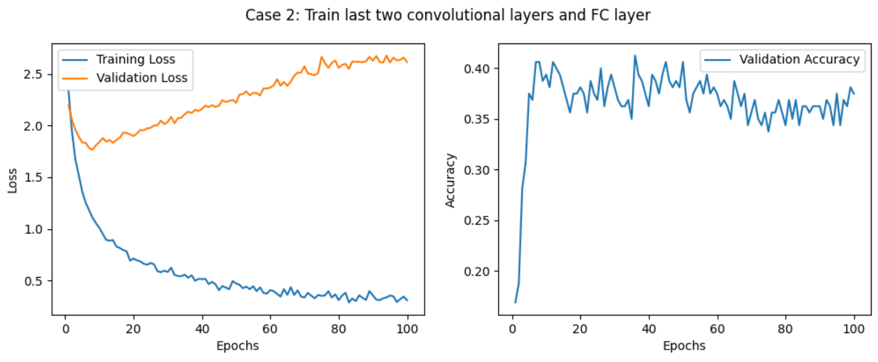
```
Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pth" to /root/.cache/
100%|██████████| 44.7M/44.7M [00:00<00:00, 171MB/s]
```

```
# Load the pre-trained ResNet-18 model
twoandFCmodel = models.resnet18(weights=ResNet18_Weights.DEFAULT)

# Freeze all layers
for param in twoandFCmodel.parameters():
    param.requires_grad = False

# Unfreeze the last two convolutional layers
for name, param in twoandFCmodel.layer4.named_parameters():
    if 'layer4.0.conv1' in name or 'layer4.0.conv2' in name:
        param.requires_grad = True

num_features = twoandFCmodel.fc.in_features
twoandFCmodel.fc = nn.Linear(num_features, num_classes) # Replace the last FC layer

twoandFCmodel = twoandFCmodel.to(device) # Move the model to the device

# Set up the criterion and optimizer
criterion2 = nn.CrossEntropyLoss()
optimizer2 = torch.optim.Adam(
    [{"params": twoandFCmodel.layer4.parameters()},
     {"params": twoandFCmodel.fc.parameters()}], lr=0.001)
```

2. Explore training with two different cases; train only FC layer and freeze rest, train last two convolutional layers and FC layer and freeze rest. Tune your parameters accordingly and give accuracy on validation set and test set. Compare and analyze your results. Give relevent code snippet.

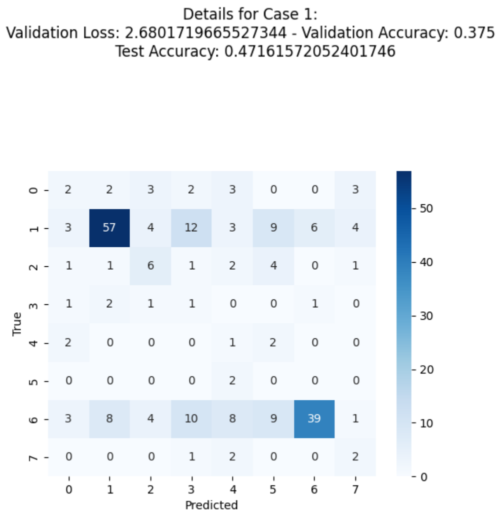
Results for the cases are:

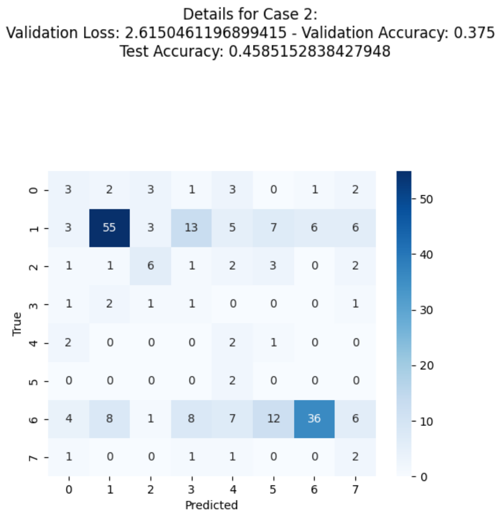




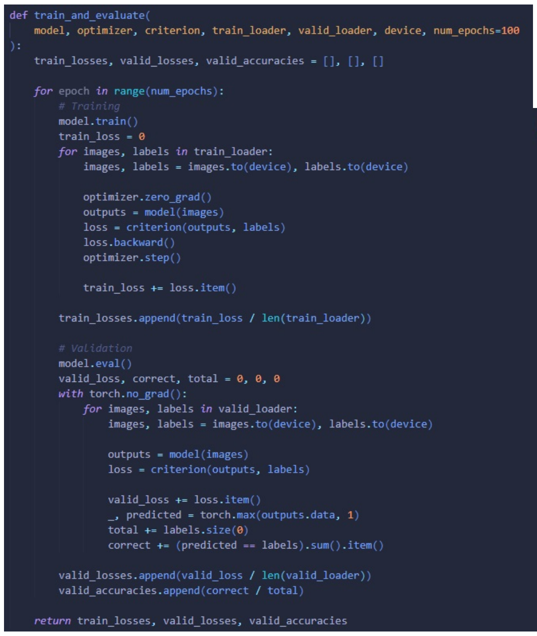
Analysing the results, training with FC layer and last two layers seems to get better accuracy. The higher epoch still tends to over fitting, while around 40 epoch the training hits it's best fit. We can see the fluctuations to the accuracy due the batch size for both results, however while FC only model gets a quick and high result around 40 epoch the FC and last two layers model seem to get stable and better results with higher epochs.

Their confusion matrices and accuracy values are:

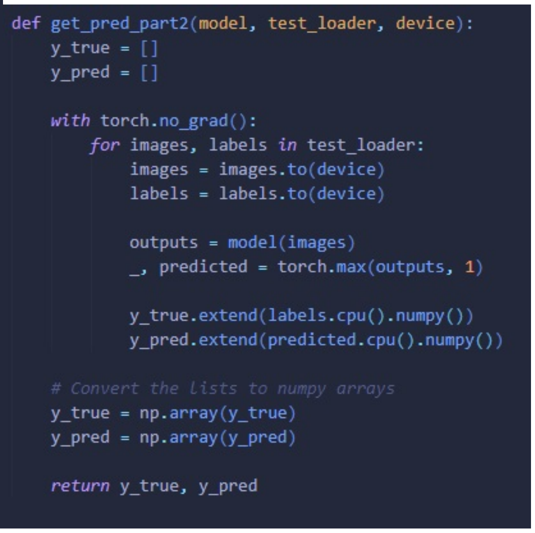




It seems the results are similar, and the tendency of overfitting is the same, but the confusion matrix for only FC seems better. Case 2 seems to get higher accuracy for both validation and test. This indicates the data stored to last two layers helps.



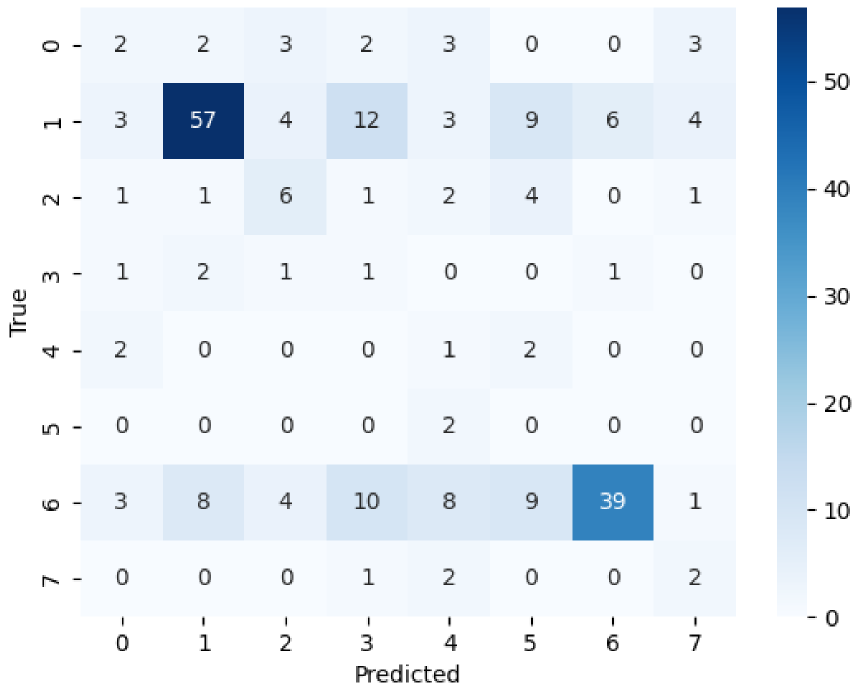
```python
def train_and_evaluate(
    model, optimizer, criterion, train_loader, valid_loader, device, num_epochs=100
):
    train_losses, valid_losses, valid_accuracies = [], [], []

    for epoch in range(num_epochs):
        # Training
        model.train()
        train_loss = 0
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            train_loss += loss.item()

        train_losses.append(train_loss / len(train_loader))

        # Validation
        model.eval()
        valid_loss, correct, total = 0, 0, 0
        with torch.no_grad():
            for images, labels in valid_loader:
                images, labels = images.to(device), labels.to(device)

                outputs = model(images)
                loss = criterion(outputs, labels)

                valid_loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        valid_losses.append(valid_loss / len(valid_loader))
        valid_accuracies.append(correct / total)

    return train_losses, valid_losses, valid_accuracies
```



```python
def get_pred_part2(model, test_loader, device):
    y_true = []
    y_pred = []

    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, 1)

            y_true.extend(labels.cpu().numpy())
            y_pred.extend(predicted.cpu().numpy())

    # Convert the lists to numpy arrays
    y_true = np.array(y_true)
    y_pred = np.array(y_pred)

    return y_true, y_pred
```

3. Plot confusion matrix for your best model and analyze results:

The confusion matrix with test data to best model is:

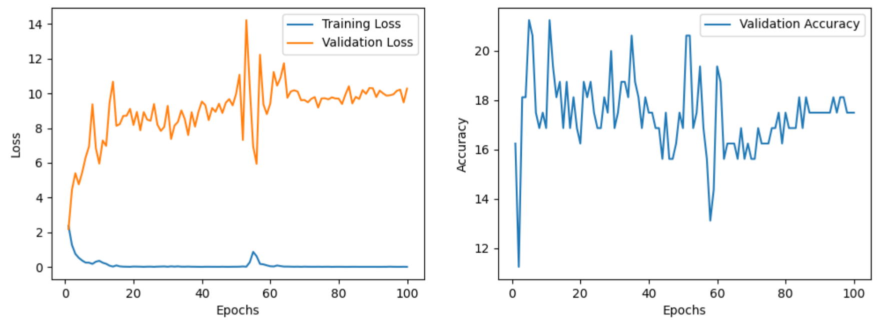


We can see the best model is performing decent, one thing to note is since with minimum 400-160 and rest is test, the test set seems to have more of some classes like one and three, still we can see at the diagonal line that most of the predictions are good. However the biggest misses are at classifying between class three and zero and one, it seems there are some test samples that are class one that is classified as class zero and one. Still there can be improvements.

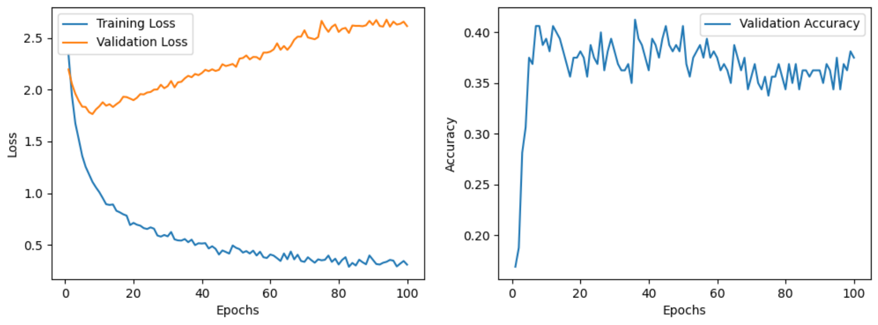
4. Compare and analyze your results in Part-1 and Part-2. Please state your observations clearly and precisely:

Comparing part-1 and part-2 it is obvious that part-2 easily catches high accuracy and ends with almost double the accuracy of part-1 while the part-1 fluctuates a lot. That is because part-2 uses weights of the pre trained ResNet model but the part-1 is trained from zero. The fluctuations are caused by over fittings due to learning rate and batch size. We can confirm this by seeing, train validation increasing and validation decreasing. This kind of behaviour does not happen at part-2 since the most layers are frozen.

Learning Rate: 0.001 - Batch Size: 16



Case 2: Train last two convolutional layers and FC layer



Also the performance and time it takes to run the code is greatly on the side of part 2. Not only the performance of model is better the time it takes to train is amazing as well.

Concluding the work, I realize the resizing of images to 64x64 because of computing power limitations lowered the quality of models. Due to this even the transfer learning is not performing good. Out side of experiments listed here I have tried to get better results for days with the variables I can change however testing with images resized 128x128 or higher was not possible.

References

[1] Assoc. Prof. Nazlı İkizler Cinbiş. Image Classification with Convolutional Neural Networks. BBM 418 - Computer Vision Laboratory, 2023.

[2] PyTorch Documentation. "Official documentation for PyTorch library." Retrieved May 9, 2023, from `https://pytorch.org/docs/stable/index.html`

[3] Goodfellow, I., Bengio, Y., & Courville, A. "Deep learning." MIT press. Retrieved May 10, 2023, from `https://www.deeplearningbook.org/`

[4] Stack Exchange. "Where should I place dropout layers in a neural network?". Retrieved May 11, 2023 from: `https://stats.stackexchange.com/questions/240305/where-should-i-place-dropout-layers-in-a-neural-network`

[5] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., ... & Berg, A. C. "ImageNet pre-trained models." Retrieved May 12, 2023, from `https://pytorch.org/hub/pytorch_vision_resnet/`