COMP 6130 DATA MINING

Graduate Group 13

Deep Residual Learning for Image Recognition

Sajith Muralidhar
szm0227@auburn.edu

Nihitha Reddy S
nrs0041@auburn.edu

Paavana Raghunandan V
pzv0022@auburn.edu

By tackling the problem of training extremely deep neural networks for image recognition tasks, ResNet design revolutionized deep neural networks. With the introduction of residual connections into the architecture, the network was given the ability to learn residual functions and overcome the significant difficulty of vanishing gradients, which arises with increasing depth. By surpassing other cutting-edge architectures and even outperforming the accuracy of human experts on the ImageNet dataset, the ResNet architecture achieved extraordinary success. Additionally, the ResNet architecture demonstrated cutting-edge performance on the CIFAR-10 and CIFAR-100 datasets and translated well to other datasets. As a result, residual connections from the ResNet design are now frequently used in a variety of applications outside of image identification.

Environmental Settings

It is advised to train deep residual networks using high-performance computing tools like GPUs or TPUs. Since we cannot run GPU, we used google colaboratory to run our code. Since it is open source available online, we can run on any operating system. Learning rate, batch size and no of epochs are the hyperparameters which are used in the project.

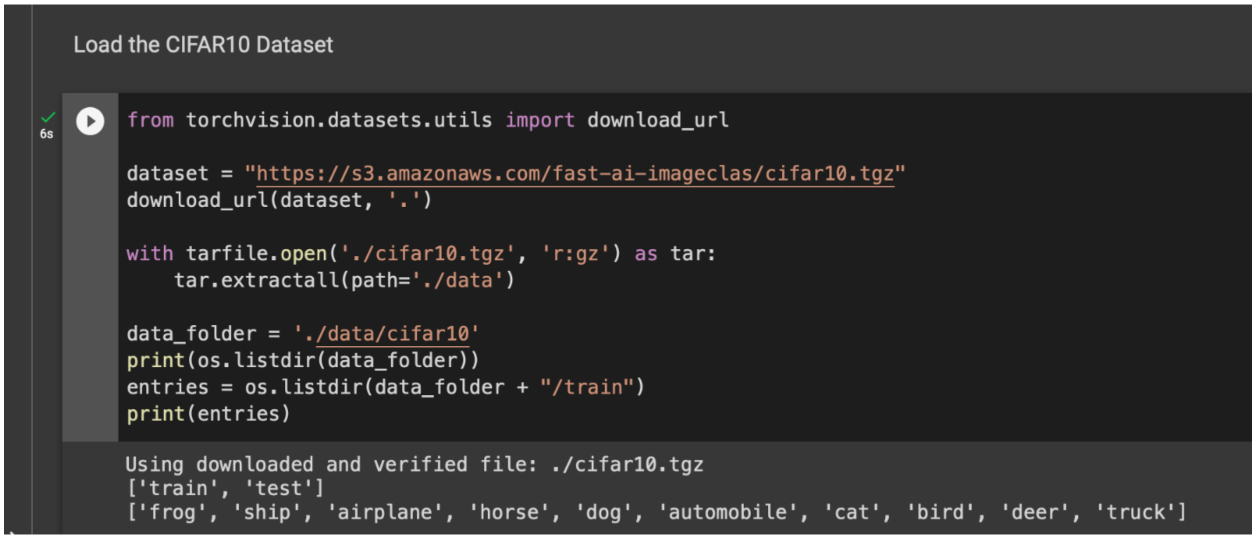
Obtaining Dataset:

We obtained the CIFAR10 dataset from Kaggle and executed it on Google colab which is an online platform. The data is already preprocessed in Kaggle.
Before loading the dataset, Select the option “Runtime > Change Runtime type” and select the “GPU” from the “Hardware Accelerator” dropdown.

<u>Dataset</u>:

The CIFAR-10 dataset, which has 60,000 32x32 color images in 10 classifications, is smaller than ImageNet. Due of the small, low-resolution photos and complicated scenes, it is a difficult dataset for image classification applications. On the CIFAR-10 dataset, the ResNet design performed at the cutting edge, demonstrating the value of residual connections in enhancing accuracy and the generalizability of the architecture.

Download the dataset from https://s3.amazonaws.com/fast-ai-imageclas/cifar10.tgz



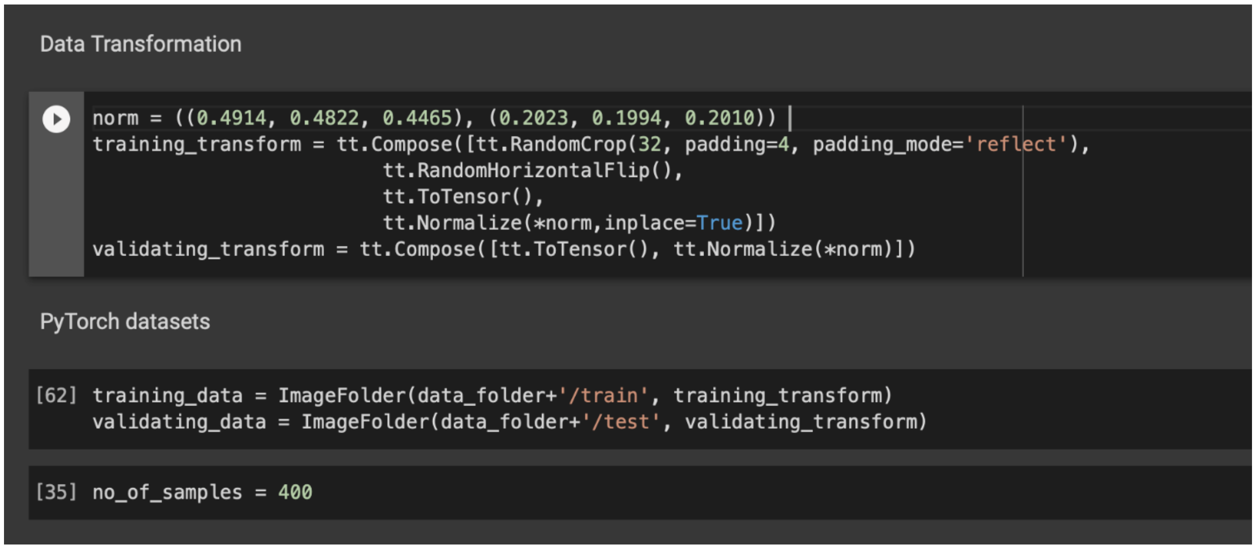
```
from torchvision.datasets.utils import download_url

dataset = "https://s3.amazonaws.com/fast-ai-imageclas/cifar10.tgz"
download_url(dataset, '.')

with tarfile.open('./cifar10.tgz', 'r:gz') as tar:
    tar.extractall(path='./data')

data_folder = './data/cifar10'
print(os.listdir(data_folder))
entries = os.listdir(data_folder + "/train")
print(entries)
```

```
Using downloaded and verified file: ./cifar10.tgz
['train', 'test']
['frog', 'ship', 'airplane', 'horse', 'dog', 'automobile', 'cat', 'bird', 'deer', 'truck']
```

<u>RESULTS</u>:

<u>Data transformation</u>:

For data transformation, we used techniques such as data augmentation and data normalization. For the training and validation datasets, we defined two distinct data transformations in the code. Random cropping, horizontal flipping, conversion to tensor format, and normalization using the mean and standard deviation values supplied by the norm variable are all included in the training transformation. Only the tensor format conversion and normalizing using the norm's supplied mean and standard deviation values are included in the validation transformation.


Data Transformation

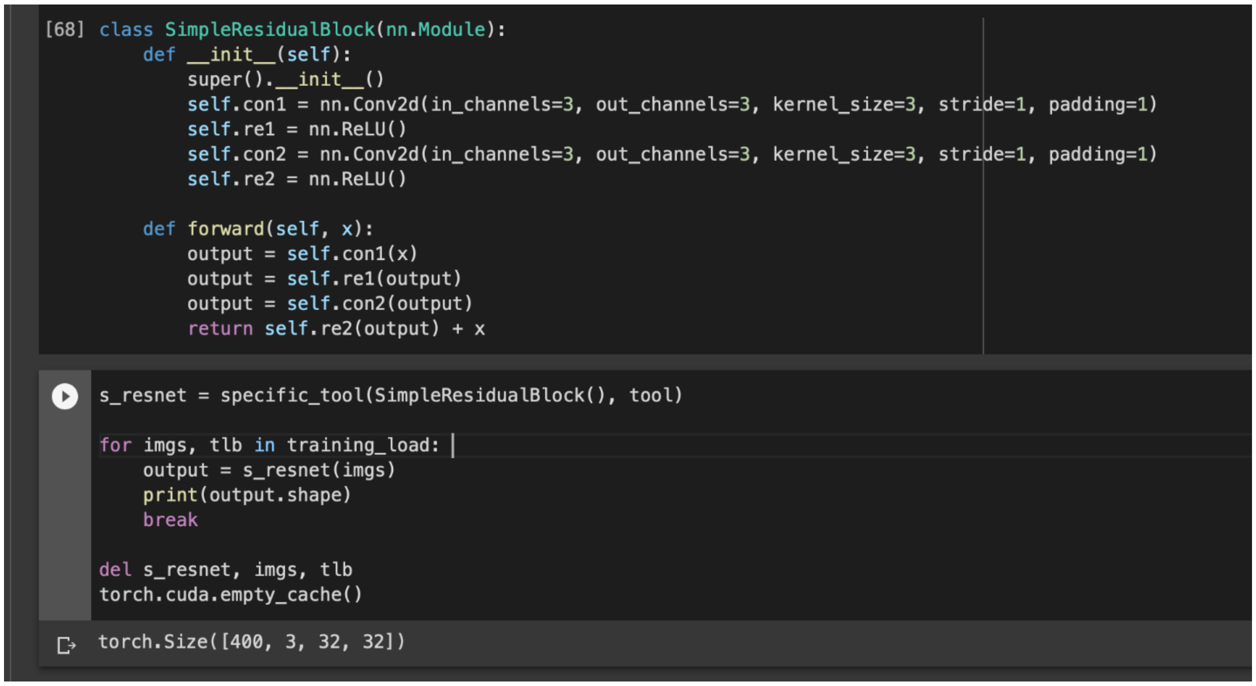
```
norm = ((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
training_transform = tt.Compose([tt.RandomCrop(32, padding=4, padding_mode='reflect'),
                         tt.RandomHorizontalFlip(),
                         tt.ToTensor(),
                         tt.Normalize(*norm,inplace=True)])
validating_transform = tt.Compose([tt.ToTensor(), tt.Normalize(*norm)])
```

PyTorch datasets

```
[62] training_data = ImageFolder(data_folder+'/train', training_transform)
     validating_data = ImageFolder(data_folder+'/test', validating_transform)
```

```
[35] no_of_samples = 400
```

Residual block and Block Normalization:



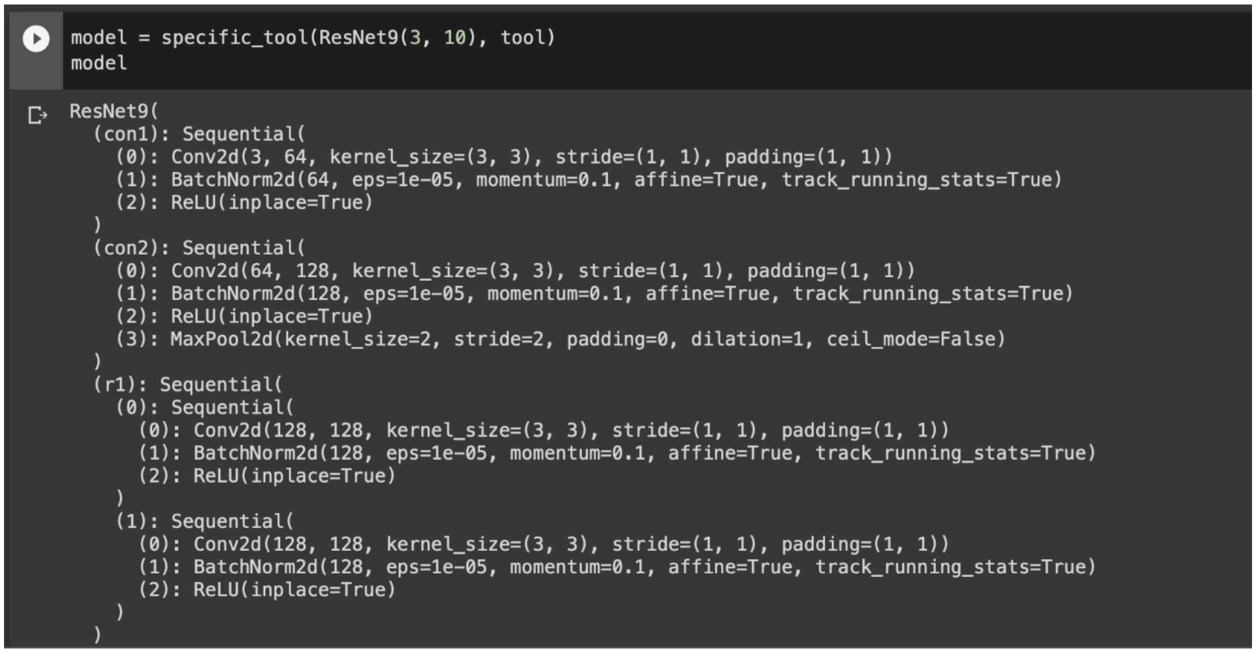
```
[68] class SimpleResidualBlock(nn.Module):
         def __init__(self):
             super().__init__()
             self.con1 = nn.Conv2d(in_channels=3, out_channels=3, kernel_size=3, stride=1, padding=1)
             self.re1 = nn.ReLU()
             self.con2 = nn.Conv2d(in_channels=3, out_channels=3, kernel_size=3, stride=1, padding=1)
             self.re2 = nn.ReLU()

         def forward(self, x):
             output = self.con1(x)
             output = self.re1(output)
             output = self.con2(output)
             return self.re2(output) + x
```

```
s_resnet = specific_tool(SimpleResidualBlock(), tool)

for imgs, tlb in training_load:
    output = s_resnet(imgs)
    print(output.shape)
    break

del s_resnet, imgs, tlb
torch.cuda.empty_cache()
```

```
torch.Size([400, 3, 32, 32])
```

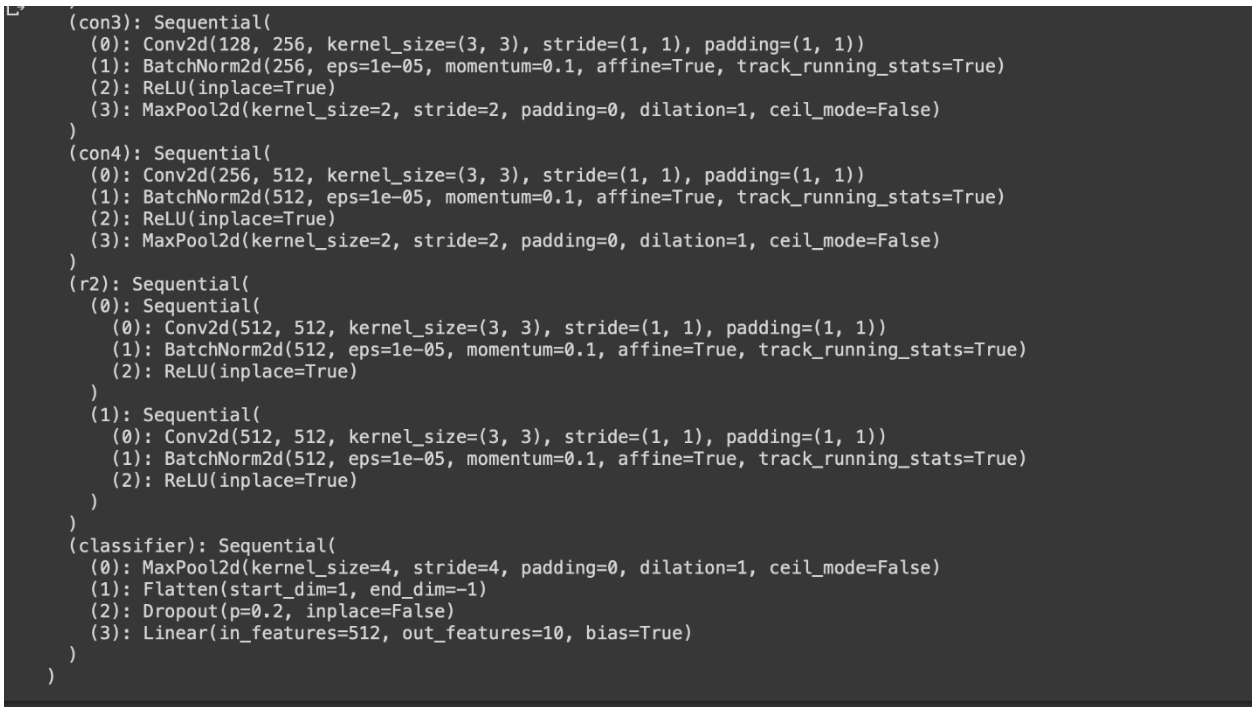
Introducing of Restnet9 Architecture:

For classifying images, the provided code defines the ResNet-9 architecture. Multiple convolutional blocks (c_block) and two residual blocks (r1 and r2) make up the architecture, which helps accuracy by enabling the model to learn more intricate characteristics. A max-pooling layer, a flattening layer, a dropout layer for regularization, and a fully connected layer with num_classes output nodes make up the classifier portion of the network. The forward function carries input through the residual and convolutional blocks, the classifier, and finally the output to execute the forward pass via the ResNet-9 architecture.



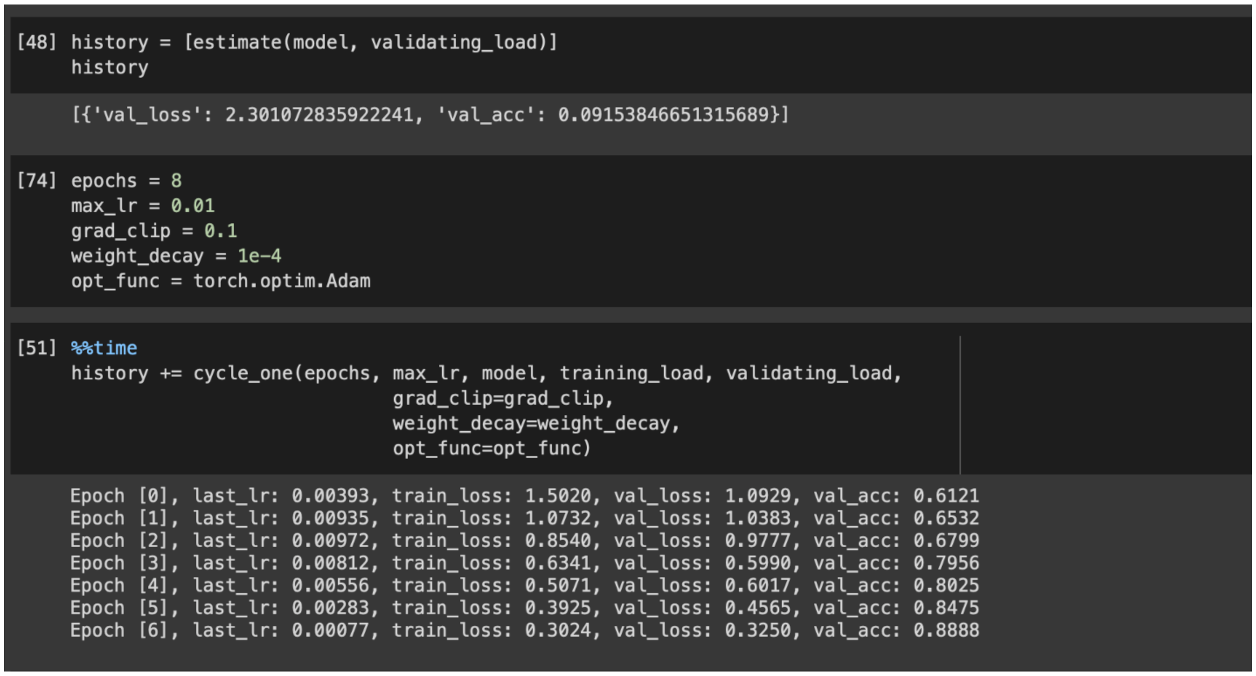
```
model = specific_tool(ResNet9(3, 10), tool)
model
```

```
ResNet9(
  (con1): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (con2): Sequential(
    (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (r1): Sequential(
    (0): Sequential(
      (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (1): Sequential(
      (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
  )
```



```
  (con3): Sequential(
    (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (con4): Sequential(
    (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (r2): Sequential(
    (0): Sequential(
      (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (1): Sequential(
      (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
  )
  (classifier): Sequential(
    (0): MaxPool2d(kernel_size=4, stride=4, padding=0, dilation=1, ceil_mode=False)
    (1): Flatten(start_dim=1, end_dim=-1)
    (2): Dropout(p=0.2, inplace=False)
    (3): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

Training the model:

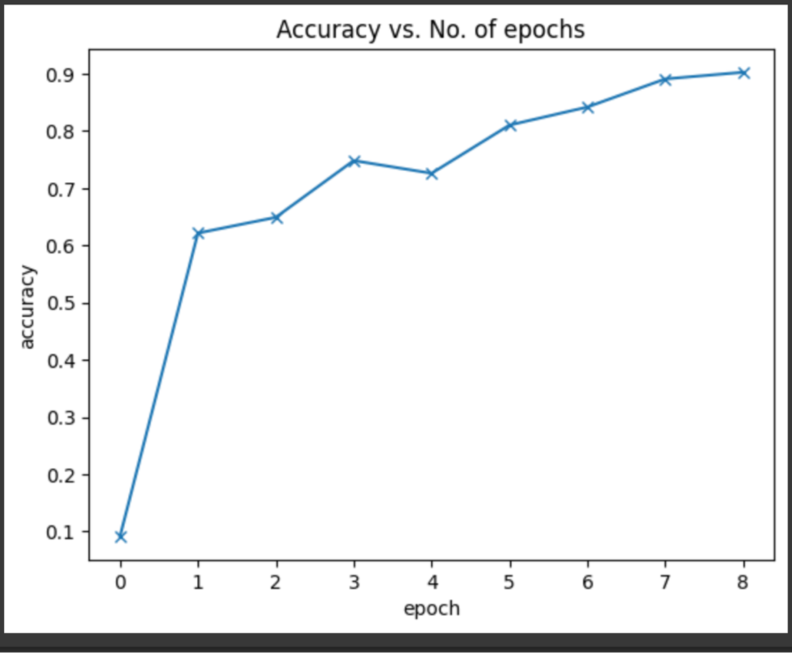


```
[48] history = [estimate(model, validating_load)]
     history

     [{'val_loss': 2.301072835922241, 'val_acc': 0.09153846651315689}]
```

```
[74] epochs = 8
     max_lr = 0.01
     grad_clip = 0.1
     weight_decay = 1e-4
     opt_func = torch.optim.Adam
```

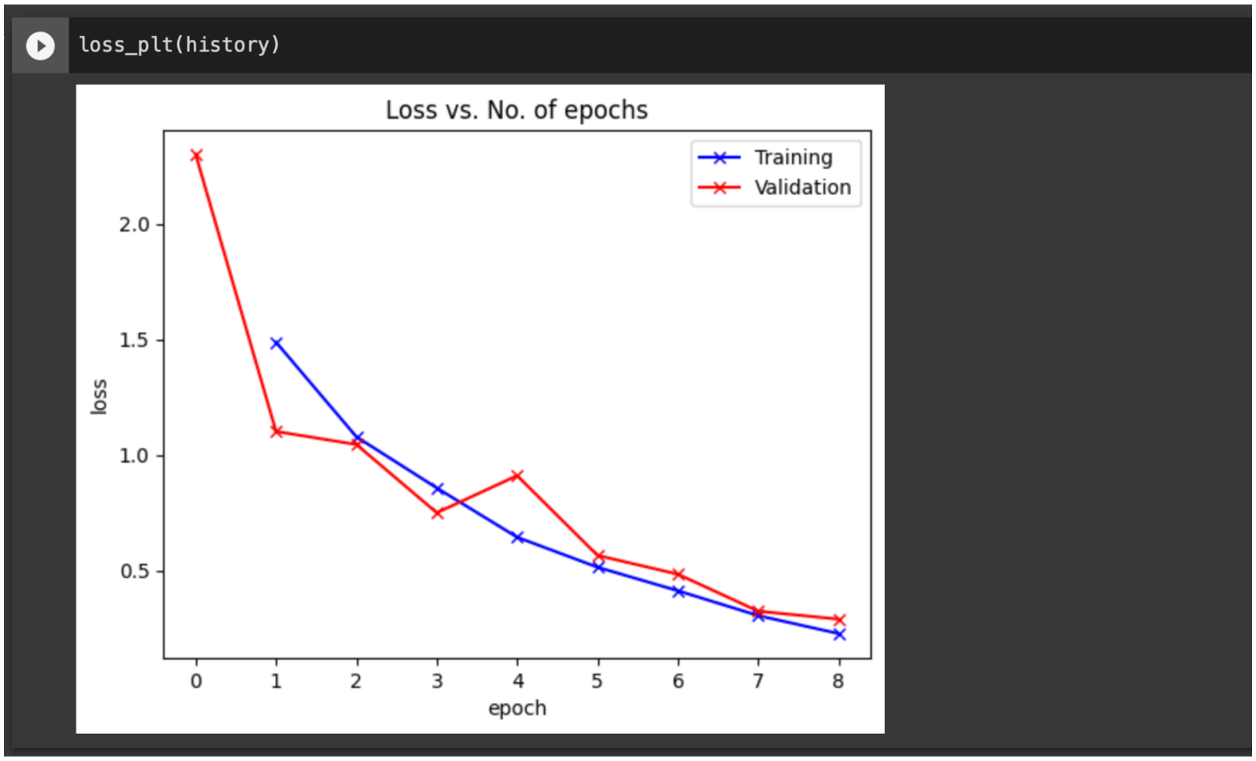
```
[51] %%time
     history += cycle_one(epochs, max_lr, model, training_load, validating_load,
                              grad_clip=grad_clip,
                              weight_decay=weight_decay,
                              opt_func=opt_func)

     Epoch [0], last_lr: 0.00393, train_loss: 1.5020, val_loss: 1.0929, val_acc: 0.6121
     Epoch [1], last_lr: 0.00935, train_loss: 1.0732, val_loss: 1.0383, val_acc: 0.6532
     Epoch [2], last_lr: 0.00972, train_loss: 0.8540, val_loss: 0.9777, val_acc: 0.6799
     Epoch [3], last_lr: 0.00812, train_loss: 0.6341, val_loss: 0.5990, val_acc: 0.7956
     Epoch [4], last_lr: 0.00556, train_loss: 0.5071, val_loss: 0.6017, val_acc: 0.8025
     Epoch [5], last_lr: 0.00283, train_loss: 0.3925, val_loss: 0.4565, val_acc: 0.8475
     Epoch [6], last_lr: 0.00077, train_loss: 0.3024, val_loss: 0.3250, val_acc: 0.8888
```

```
acc_plt(history)
```



The accuracy turns out to be 90 percent with 8 epochs.

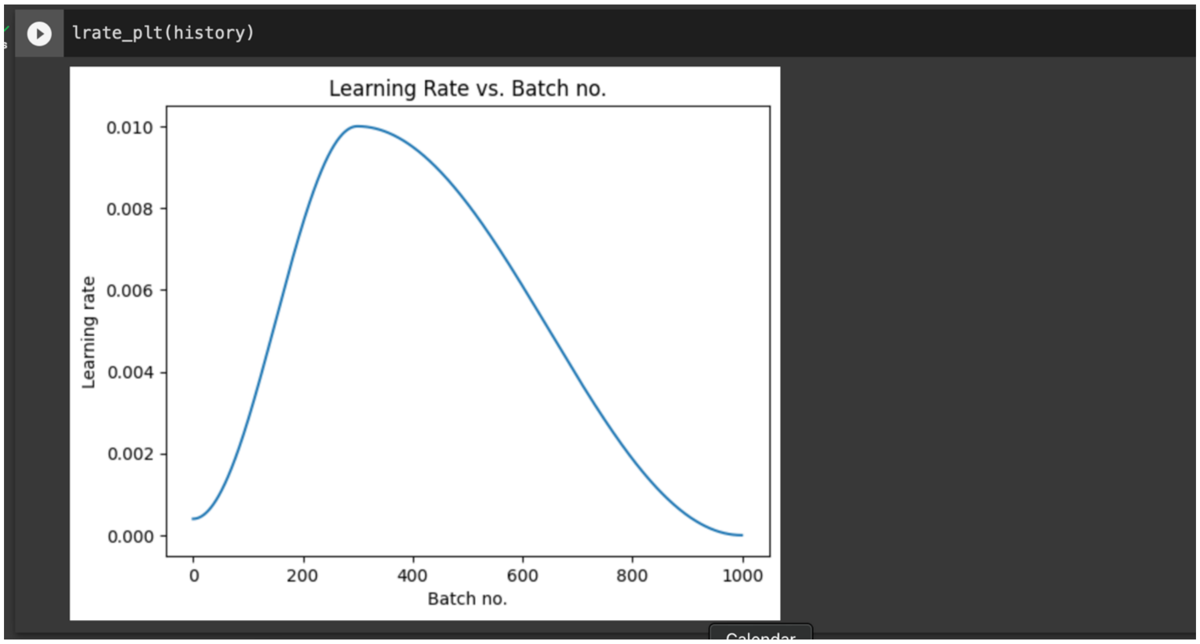
<u>Plotting the Training and Validation losses</u>:



```
loss_plt(history)
```

The model is not overfitting the training data.

<u>Learning Rate</u>:

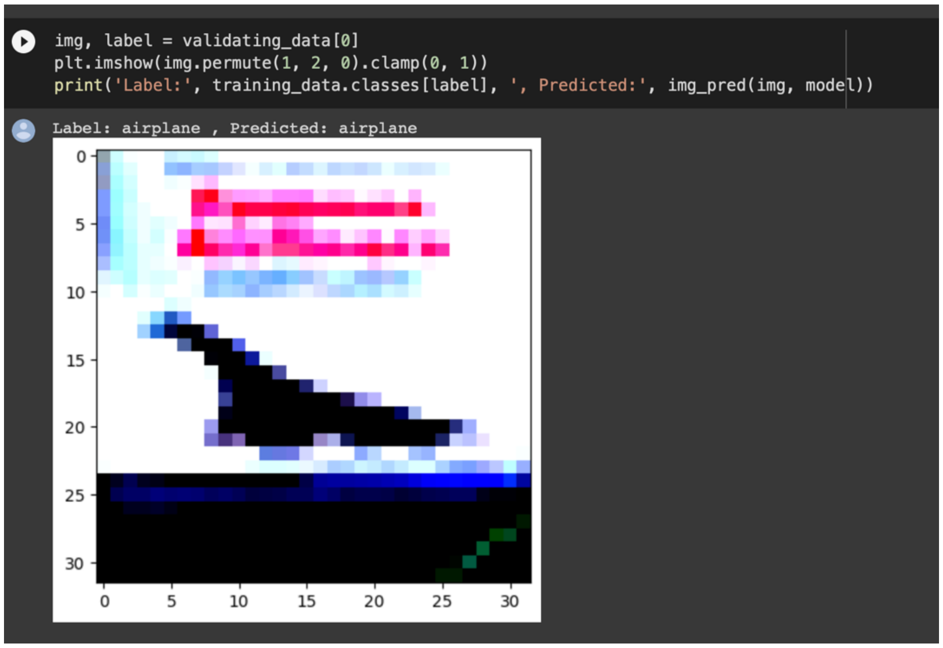


```
lrate_plt(history)
```

The learning was low initially then gradually increased to 0.01 for 30 % iterations and then decreased to lower value.

Testing on individual images:

1.

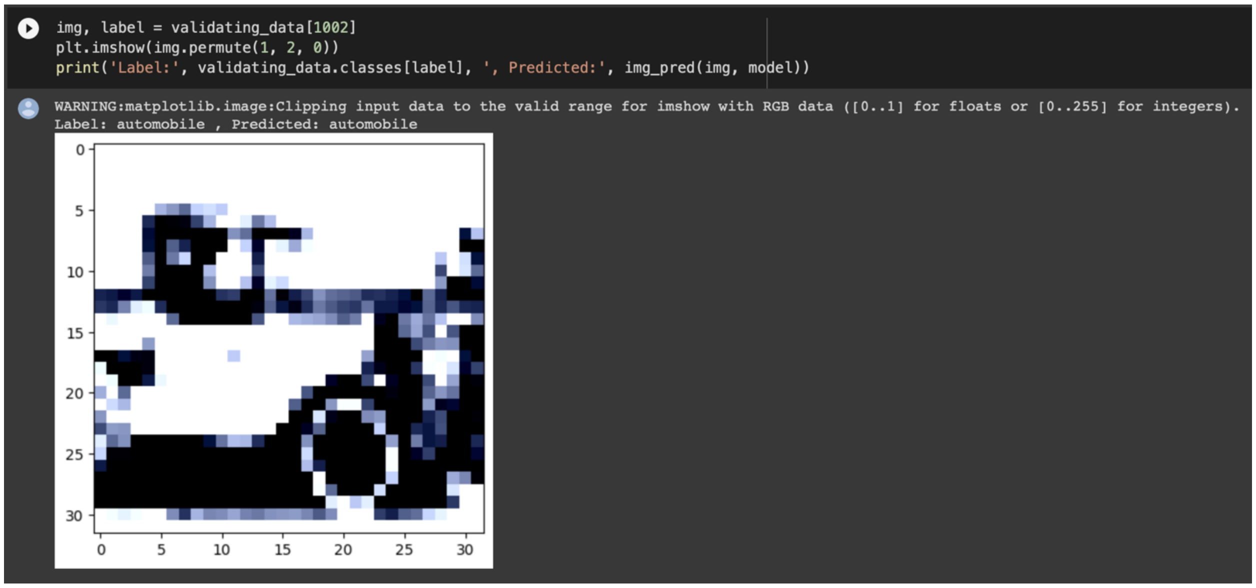


```
img, label = validating_data[0]
plt.imshow(img.permute(1, 2, 0).clamp(0, 1))
print('Label:', training_data.classes[label], ', Predicted:', img_pred(img, model))
```

```
Label: airplane , Predicted: airplane
```

The predicted model is a ship but labeled is an airplane.

2.

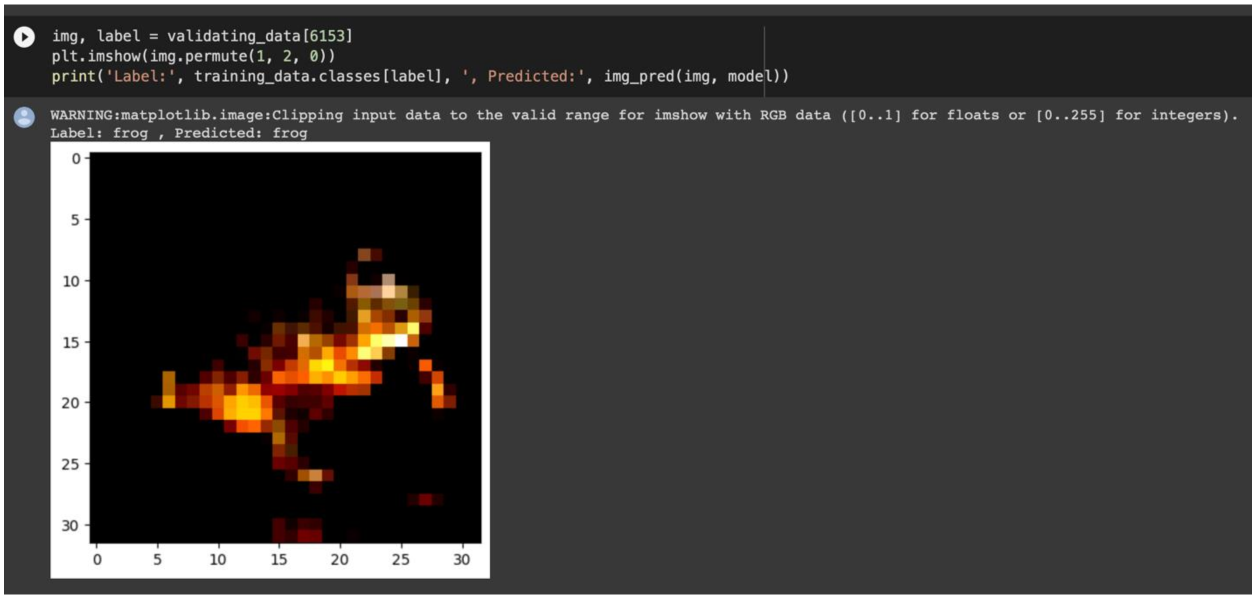


```
img, label = validating_data[1002]
plt.imshow(img.permute(1, 2, 0))
print('Label:', validating_data.classes[label], ', Predicted:', img_pred(img, model))
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Label: automobile , Predicted: automobile
```

Both the predicted model and the labeled turned out to be automobiles.

3.



```
img, label = validating_data[6153]
plt.imshow(img.permute(1, 2, 0))
print('Label:', training_data.classes[label], ', Predicted:', img_pred(img, model))
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Label: frog , Predicted: frog
```

The model predicted is frog and the labeled is also a frog.

Overall, the findings show that ResNet is a very successful architecture for image recognition tasks, and that it is especially well suited to jobs requiring high levels of abstraction due to its capacity to handle deep networks. The ResNet architecture has been widely employed in a variety of applications outside image recognition, and the residual connections that were used in it have subsequently become a standard deep learning technique.

References

https://ieeexplore.ieee.org/document/7780459

https://s3.amazonaws.com/fast-ai-imageclas/cifar10.tgz

https://s3.amazonaws.com/fast-ai-imageclas/cifar100.tgz

https://www.researchgate.net/publication/311609041_Deep_Residual_Learning_for_Image_Recognition