Implementation Report for exercise 3

Homework 4

By: Katayoun Kebrai

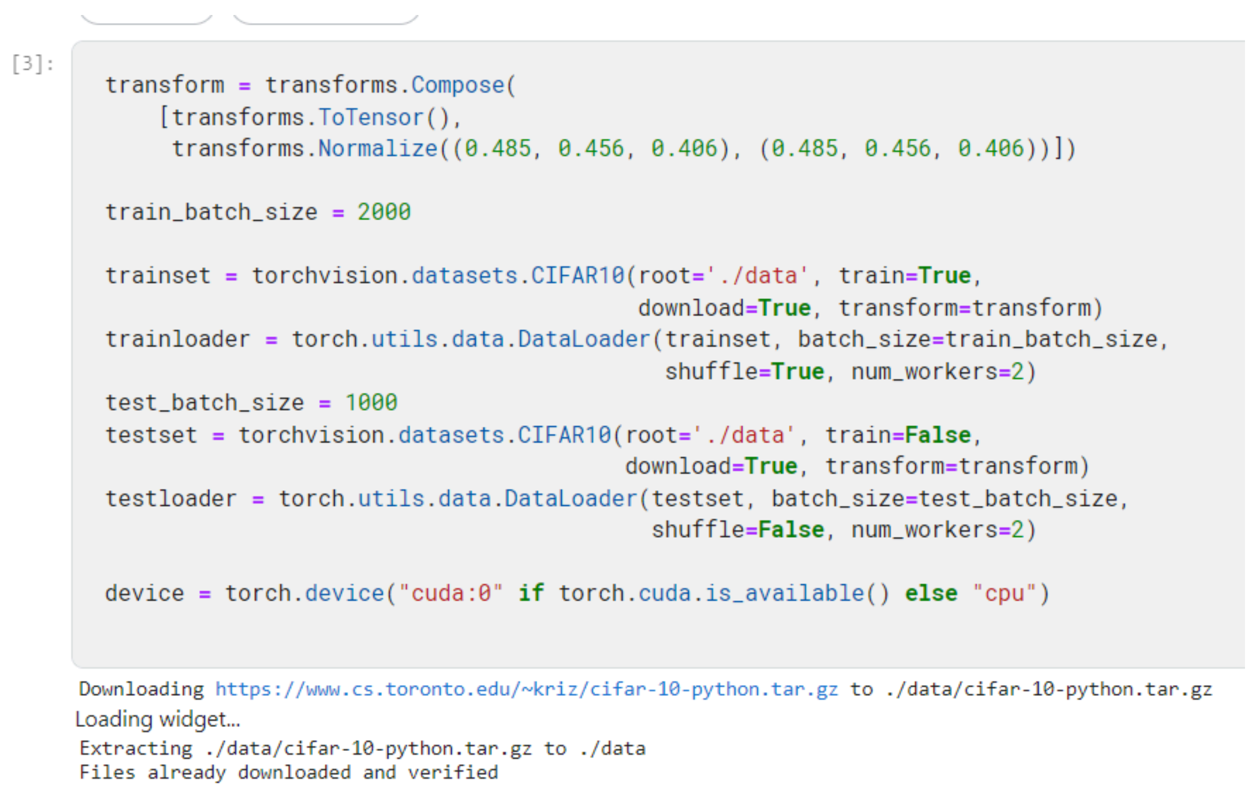
In this exercise, we worked with the CIFAR-10 dataset, and the task was to implement an autoencoder capable of separating and distinguishing composite images and returning them as outputs. The key difference between this exercise and previous ones was that, in addition to the accuracy and error metrics, the desired output in this case was the correctly distinguished and reconstructed images.

Step 1: Loading the Dataset

We started by loading the dataset. Since the CIFAR-10 dataset is available in the TorchVision library, it was easily accessible. The important thing to note about this method of loading data is that we can specify a batch size. The exercise required us to work with 1000 data points, so we set the batch size accordingly and used the shuffle mode to randomly load a subset of the data.

Step 2: Normalizing the Data

As with any dataset, the data needs to be normalized. Using the Torch library's transforms, we normalized all the data and converted it into tensors at the same time. After applying the transformations, we passed the data to a DataLoader to store the limited dataset, and the transforms were applied to this object as well. We repeated the same process for the test data.

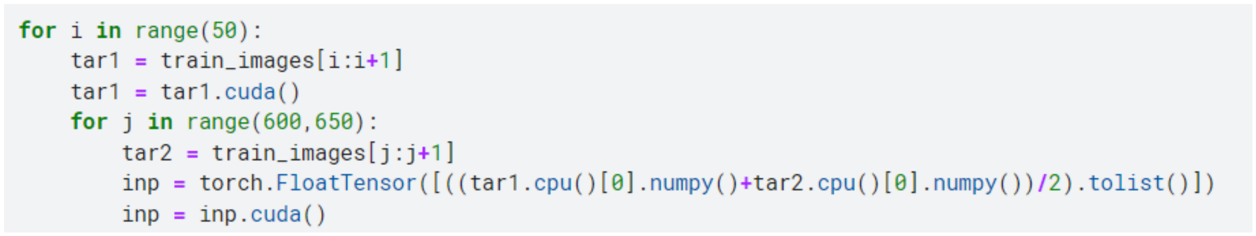


```
[3]:
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.485, 0.456, 0.406), (0.485, 0.456, 0.406))])

train_batch_size = 2000

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=train_batch_size,
                                          shuffle=True, num_workers=2)
test_batch_size = 1000
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=test_batch_size,
                                         shuffle=False, num_workers=2)

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

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
Loading widget...
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

Step 3: Creating Composite Images

Once the dataset was properly loaded, we needed to generate the required input data. The problem stated that the input data should be the average of two images from the original dataset. There are different methods to achieve this. For example, you could select two sets of 2000 images from the original dataset and compute the element-wise average of corresponding pairs. In our implementation, however, we selected two smaller sets of 50 images and iteratively computed the average of each image with all 50 members of the other set, saving the results.



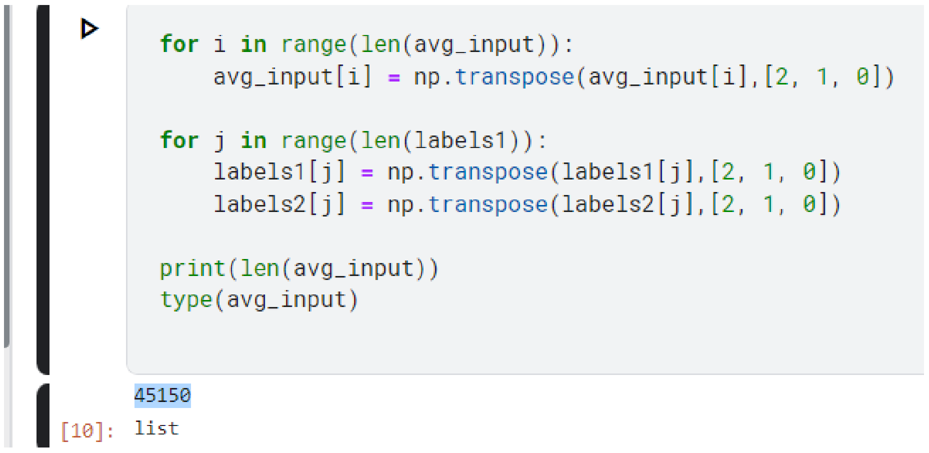
```
for i in range(50):
    tar1 = train_images[i:i+1]
    tar1 = tar1.cuda()
    for j in range(600,650):
        tar2 = train_images[j:j+1]
        inp = torch.FloatTensor([((tar1.cpu()[0].numpy()+tar2.cpu()[0].numpy())/2).tolist()])
        inp = inp.cuda()
```

Step 4: Moving to GPU

Since we were working with a GPU, we transferred all model inputs to CUDA.

Step 5: Visualizing the Data

To visualize some examples of the generated composite images, the input needed to be reshaped from 32×32×3 to 3×32×32, making it compatible with the plotting function.

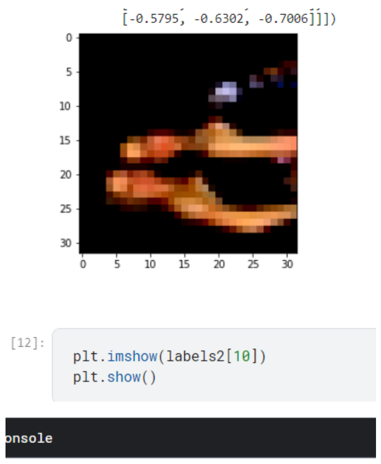


```
for i in range(len(avg_input)):
    avg_input[i] = np.transpose(avg_input[i],[2, 1, 0])

for j in range(len(labels1)):
    labels1[j] = np.transpose(labels1[j],[2, 1, 0])
    labels2[j] = np.transpose(labels2[j],[2, 1, 0])

print(len(avg_input))
type(avg_input)
```

```
45150
[10]: list
```

Below are some of the composite images created by averaging pairs of images from the original dataset.

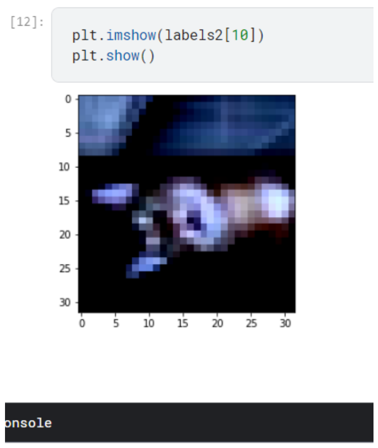
- **Original Images:**



```
[-0.5795, -0.6302, -0.7006]]])
```

```
[12]:
plt.imshow(labels2[10])
plt.show()
```

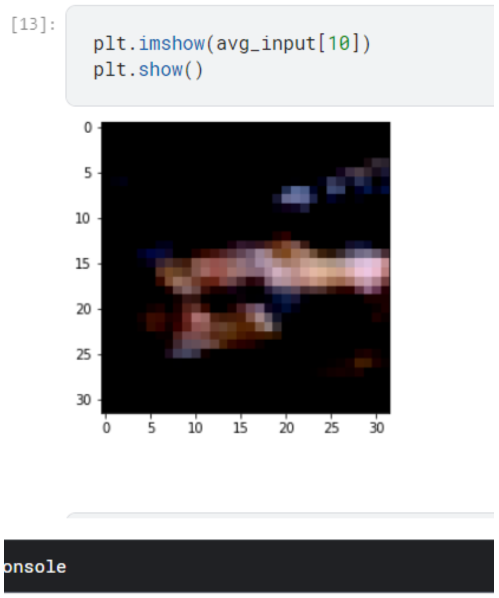
```
onsole
```



```
[12]:
plt.imshow(labels2[10])
plt.show()
```

```
onsole
```

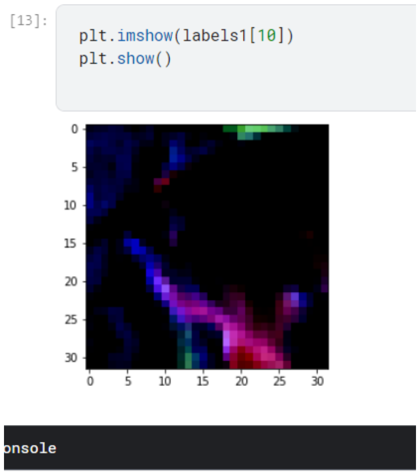
- **Generated Composite Images:**



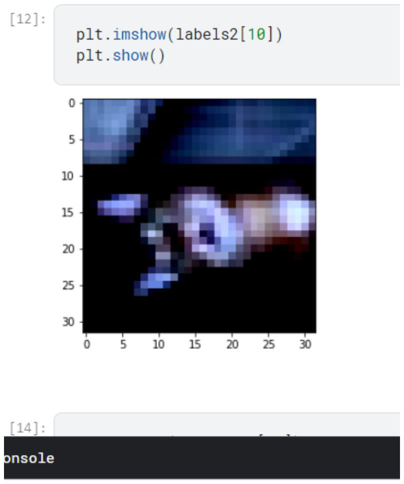
```
[13]:
plt.imshow(avg_input[10])
plt.show()
```

```
onsole
```

- **Original Images:**

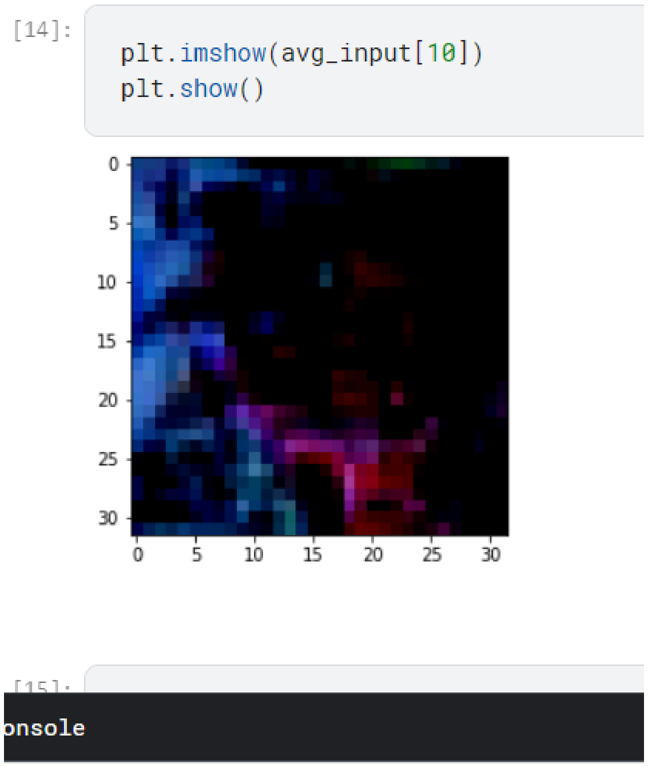


```
[13]: plt.imshow(labels1[10])
      plt.show()
```



```
[12]: plt.imshow(labels2[10])
      plt.show()
```

- **Generated Composite Images:**

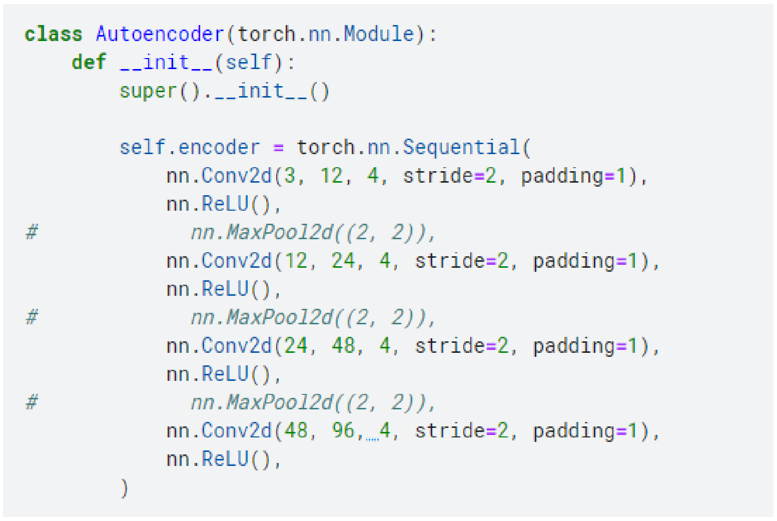


```
[14]: plt.imshow(avg_input[10])
      plt.show()
```

Step 6: Model Design

Our code creates an object that contains the three main components of the task: an encoder and two parallel decoders. Each component is designed similarly to previous models, by selecting a series of linear or convolutional layers followed by the appropriate activation function. The key point here is that the object contains two parallel decoders. Both decoders receive the output of the encoder, re-encode it, and then decode it again to reconstruct the original images. Based on experience, we opted for convolutional layers instead of linear layers for better results.

- The encoder takes the input image, processes it through layers where each successive layer is doubled, and passes it to the next layer. While we could have added more layers, excessive depth in the encoder tends to degrade results.
- These encoded layers are then passed to the decoders, which expand them layer by layer, halving the dimensions each time, until the original images are reconstructed. Each decoder ultimately returns one reconstructed image.



```
class Autoencoder(torch.nn.Module):
    def __init__(self):
        super().__init__()

        self.encoder = torch.nn.Sequential(
            nn.Conv2d(3, 12, 4, stride=2, padding=1),
            nn.ReLU(),
#             nn.MaxPool2d((2, 2)),
            nn.Conv2d(12, 24, 4, stride=2, padding=1),
            nn.ReLU(),
#             nn.MaxPool2d((2, 2)),
            nn.Conv2d(24, 48, 4, stride=2, padding=1),
            nn.ReLU(),
#             nn.MaxPool2d((2, 2)),
            nn.Conv2d(48, 96, 4, stride=2, padding=1),
            nn.ReLU(),
        )
        self.decoder1 = torch.nn.Sequential(
            nn.ConvTranspose2d(96, 48, 4, stride=2, padding=1),
#             nn.Conv2d(96, 48, 4, stride=2, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(48, 24, 4, stride=2, padding=1),
#             nn.Conv2d(48, 24, 4, stride=2, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(24, 12, 4, stride=2, padding=1),
#             nn.Conv2d(24, 12, 4, stride=2, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(12, 3, 4, stride=2, padding=1),
#             nn.Conv2d(12, 6, 4, stride=2, padding=1),
            nn.Sigmoid(),
        )

        self.decoder2 = torch.nn.Sequential(
            nn.ConvTranspose2d(96, 48, 4, stride=2, padding=1),
#             nn.Conv2d(96, 48, 4, stride=2, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(48, 24, 4, stride=2, padding=1),
#             nn.Conv2d(48, 24, 4, stride=2, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(24, 12, 4, stride=2, padding=1),
#             nn.Conv2d(24, 12, 4, stride=2, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(12, 3, 4, stride=2, padding=1),
#             nn.Conv2d(12, 6, 4, stride=2, padding=1),
            nn.Sigmoid(),
        )
```

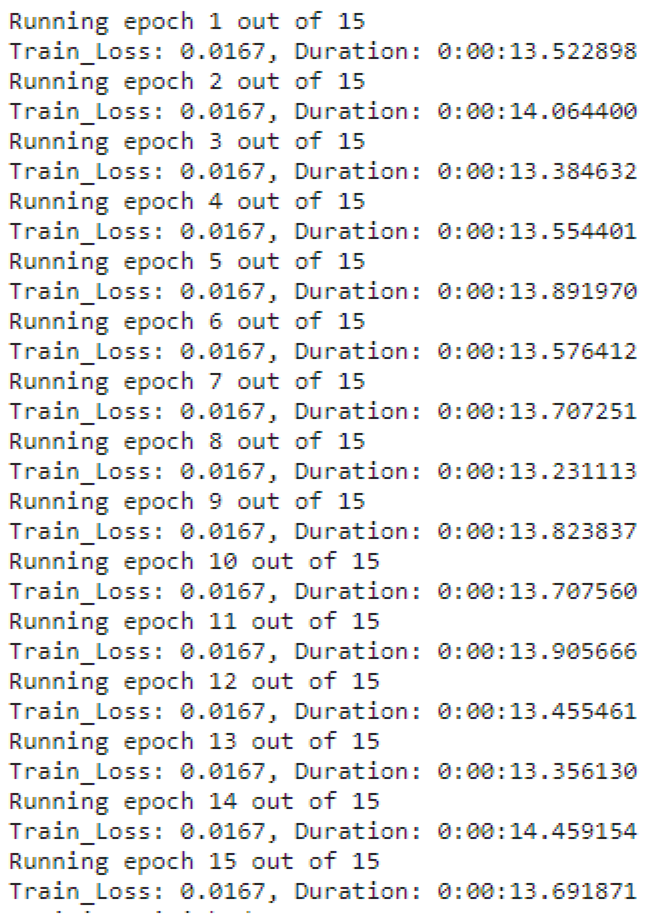
Like the input data, we also moved the model to the GPU for faster computation.

We used the Adam optimizer and the MSE (Mean Squared Error) loss function to obtain the best possible results.

```
model = create_model()
print(model)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters())
```

Finally, we start to train the model.

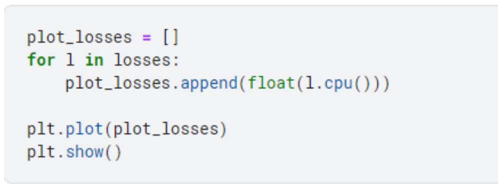
```python
epochs = 15
losses = []
for epoch in range(epochs):
    t0 = datetime.now()
    running_loss = 0.0
    print(f"Running epoch {epoch+1} out of {epochs}")

    for i in range(50):
        tar1 = train_images[i:i+1]
        tar1 = tar1.cuda()
        for j in range(600,650):
            tar2 = train_images[j:j+1]
            inp = torch.FloatTensor([((tar1.cpu()[0].numpy()+tar2.cpu()[0].numpy())/2).tolist()])
            inp = inp.cuda()
            tar2 = tar2.cuda()


            out1, out2 = model(inp)
            loss_1 = criterion(out1, tar1)
            loss_2 = criterion(out2, tar2)
            loss = loss_1 + loss_2
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            running_loss += loss.data
    losses.append(running_loss/40000)

    dt = datetime.now() - t0
    print(f'Train_Loss: {losses[-1]:.4f}, Duration: {dt} ')
```



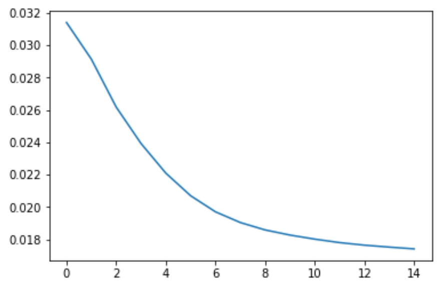
```
Running epoch 1 out of 15
Train_Loss: 0.0167, Duration: 0:00:13.522898
Running epoch 2 out of 15
Train_Loss: 0.0167, Duration: 0:00:14.064400
Running epoch 3 out of 15
Train_Loss: 0.0167, Duration: 0:00:13.384632
Running epoch 4 out of 15
Train_Loss: 0.0167, Duration: 0:00:13.554401
Running epoch 5 out of 15
Train_Loss: 0.0167, Duration: 0:00:13.891970
Running epoch 6 out of 15
Train_Loss: 0.0167, Duration: 0:00:13.576412
Running epoch 7 out of 15
Train_Loss: 0.0167, Duration: 0:00:13.707251
Running epoch 8 out of 15
Train_Loss: 0.0167, Duration: 0:00:13.231113
Running epoch 9 out of 15
Train_Loss: 0.0167, Duration: 0:00:13.823837
Running epoch 10 out of 15
Train_Loss: 0.0167, Duration: 0:00:13.707560
Running epoch 11 out of 15
Train_Loss: 0.0167, Duration: 0:00:13.905666
Running epoch 12 out of 15
Train_Loss: 0.0167, Duration: 0:00:13.455461
Running epoch 13 out of 15
Train_Loss: 0.0167, Duration: 0:00:13.356130
Running epoch 14 out of 15
Train_Loss: 0.0167, Duration: 0:00:14.459154
Running epoch 15 out of 15
Train_Loss: 0.0167, Duration: 0:00:13.691871
```

Step 10: Visualizing the Loss

The obtained loss values were quite low and suitable for the task. Plotting the loss curve shows that it decreases significantly, indicating that the model is functioning correctly and is able to extract the two .original images from the composite with minimal error

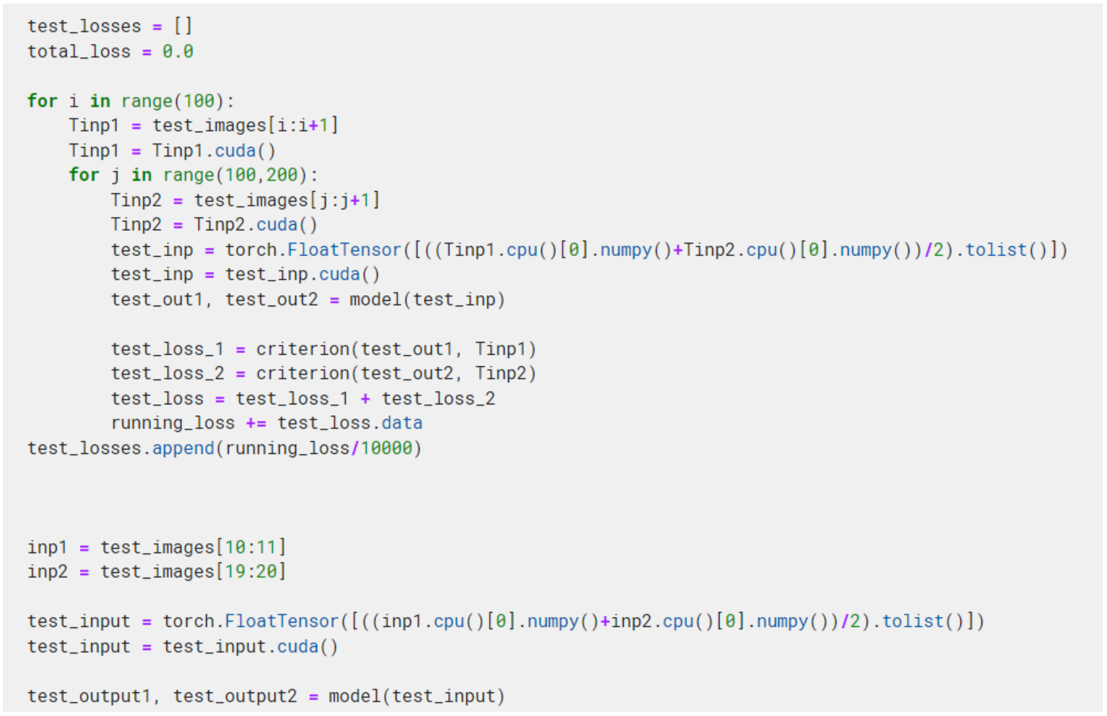


```
plot_losses = []
for l in losses:
    plot_losses.append(float(l.cpu()))

plt.plot(plot_losses)
plt.show()
```

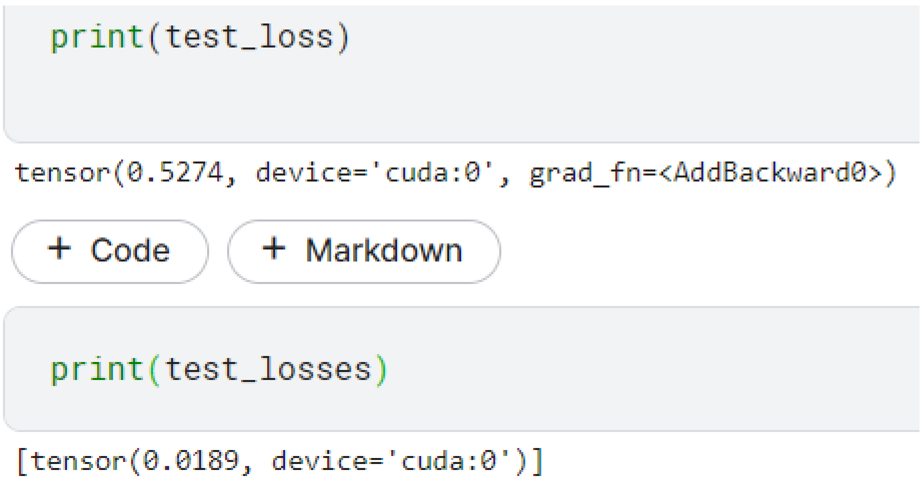


Step 11: Testing the Model

To test the model, we passed test data through it to generate two output images. We randomly selected 100 samples from the test data and fed them into the model to calculate their losses. After generating the data and obtaining an array of losses, we computed the average of this array to represent the overall test loss for the model.



```python
test_losses = []
total_loss = 0.0

for i in range(100):
    Tinp1 = test_images[i:i+1]
    Tinp1 = Tinp1.cuda()
    for j in range(100,200):
        Tinp2 = test_images[j:j+1]
        Tinp2 = Tinp2.cuda()
        test_inp = torch.FloatTensor([((Tinp1.cpu()[0].numpy()+Tinp2.cpu()[0].numpy())/2).tolist()])
        test_inp = test_inp.cuda()
        test_out1, test_out2 = model(test_inp)

        test_loss_1 = criterion(test_out1, Tinp1)
        test_loss_2 = criterion(test_out2, Tinp2)
        test_loss = test_loss_1 + test_loss_2
        running_loss += test_loss.data
test_losses.append(running_loss/10000)



inp1 = test_images[10:11]
inp2 = test_images[19:20]

test_input = torch.FloatTensor([((inp1.cpu()[0].numpy()+inp2.cpu()[0].numpy())/2).tolist()])
test_input = test_input.cuda()

test_output1, test_output2 = model(test_input)
```



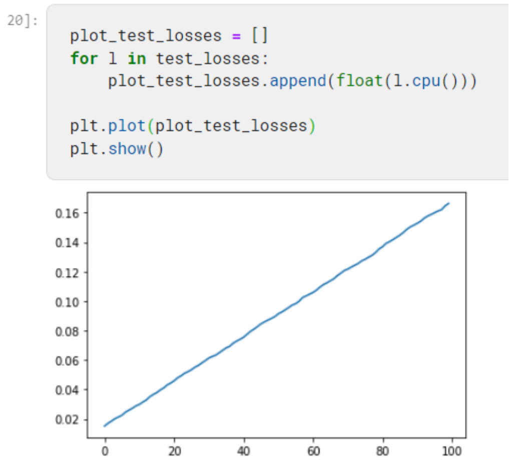
```python
print(test_loss)
```

```
tensor(0.5274, device='cuda:0', grad_fn=<AddBackward0>)
```

```python
print(test_losses)
```

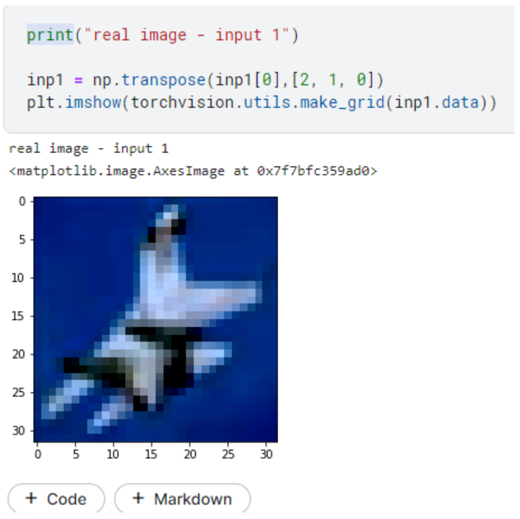
```
[tensor(0.0189, device='cuda:0')]
```

For better insights, we also plotted the loss curves:



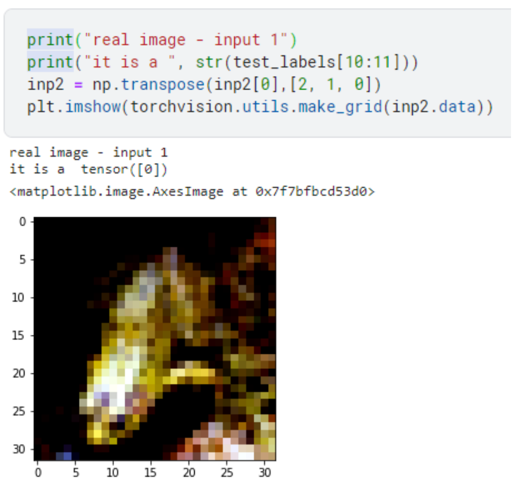
```
20]:
plot_test_losses = []
for l in test_losses:
    plot_test_losses.append(float(l.cpu()))

plt.plot(plot_test_losses)
plt.show()
```

Finally, we printed the outputs visually to verify how well the model performed.



```
print("real image - input 1")

inp1 = np.transpose(inp1[0],[2, 1, 0])
plt.imshow(torchvision.utils.make_grid(inp1.data))
```

```
real image - input 1
<matplotlib.image.AxesImage at 0x7f7bfc359ad0>
```



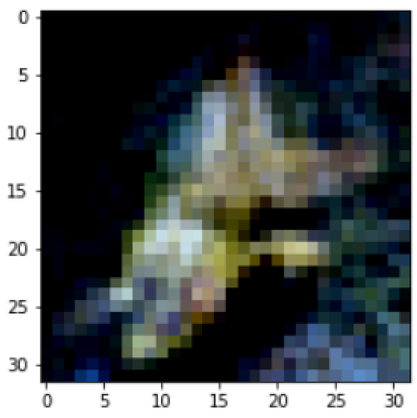
```
print("real image - input 1")
print("it is a ", str(test_labels[10:11]))
inp2 = np.transpose(inp2[0],[2, 1, 0])
plt.imshow(torchvision.utils.make_grid(inp2.data))
```

```
real image - input 1
it is a  tensor([0])
<matplotlib.image.AxesImage at 0x7f7bfbcd53d0>
```

- **Composite Image Generated:**

```
print("avarage image - input")

avg = torch.FloatTensor([((inp2.cpu().numpy()+inp1.cpu().numpy())/2).tolist()])
avg = np.array(avg)
plt.imshow(avg[0])
```

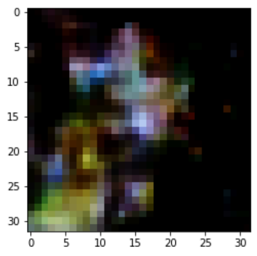
```
avarage image - input
<matplotlib.image.AxesImage at 0x7efc3efe7b50>
```



- **Model Outputs:**

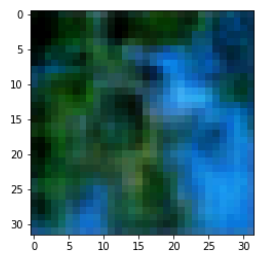
```
print("predected image - output 1")

type(test_output2[0])
index = test_output2[0].cpu()
index = index.detach().numpy()
# output1 = np.array(index)
type(index)
index.shape
test_output2 = np.transpose(index,[2, 1, 0])
plt.imshow(test_output2)
```

```
<matplotlib.image.AxesImage at 0x7f7bfc2c5390>
```



```
print("predected image - output 1")

index = test_output1[0].cpu()
index = index.detach().numpy()
test_output1 = np.transpose(index,[2, 1, 0])
plt.imshow(test_output1)
```

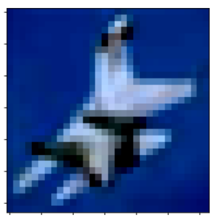
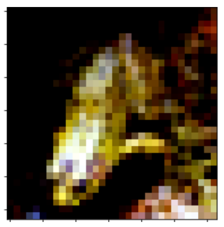
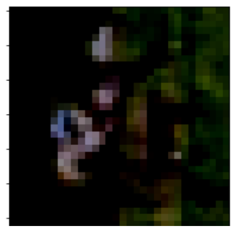
```
<matplotlib.image.AxesImage at 0x7f7bfc286c90>
```



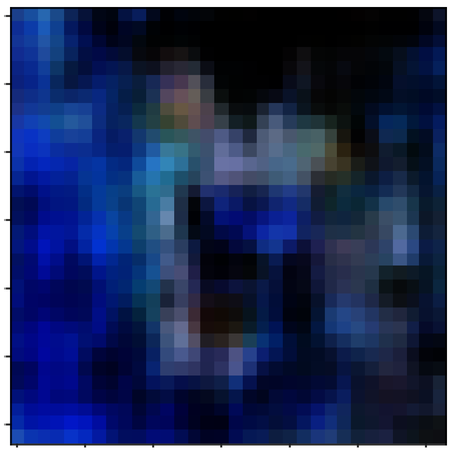
In this exercise, we experimented with different structures and larger datasets. For example, when we set the batch size to 100 or 200, we obtained results, but they were not optimal. In one such experiment, the loss reached 0.06730.06730.0673.

```
Running epoch 1 out of 20
Train_Loss: 0.0678, Duration: 0:00:54.968518
Running epoch 2 out of 20
Train_Loss: 0.0677, Duration: 0:00:55.110824
Running epoch 3 out of 20
Train_Loss: 0.0676, Duration: 0:00:54.897178
Running epoch 4 out of 20
Train_Loss: 0.0676, Duration: 0:00:55.145834
Running epoch 5 out of 20
Train_Loss: 0.0676, Duration: 0:00:55.057226
Running epoch 6 out of 20
Train_Loss: 0.0676, Duration: 0:00:55.244662
Running epoch 7 out of 20
Train_Loss: 0.0675, Duration: 0:00:54.640514
Running epoch 8 out of 20
Train_Loss: 0.0675, Duration: 0:00:55.070483
Running epoch 9 out of 20
Train_Loss: 0.0675, Duration: 0:00:55.224217
Running epoch 10 out of 20
Train_Loss: 0.0674, Duration: 0:00:55.243265
Running epoch 11 out of 20
Train_Loss: 0.0674, Duration: 0:00:54.732648
Running epoch 12 out of 20
Train_Loss: 0.0674, Duration: 0:00:55.171618
Running epoch 13 out of 20
Train_Loss: 0.0673, Duration: 0:00:55.543859
Running epoch 14 out of 20
Train_Loss: 0.0674, Duration: 0:00:55.367001
Running epoch 15 out of 20
Train_Loss: 0.0673, Duration: 0:00:54.837852
Running epoch 16 out of 20
Train_Loss: 0.0673, Duration: 0:00:55.792784
Running epoch 17 out of 20
Train_Loss: 0.0674, Duration: 0:00:55.161610
Running epoch 18 out of 20
Train_Loss: 0.0672, Duration: 0:00:55.225406
Running epoch 19 out of 20
Train_Loss: 0.0673, Duration: 0:00:55.333582
Running epoch 20 out of 20
Train_Loss: 0.0673, Duration: 0:00:55.327516
```

The model then reconstructed the images as:

( + ) / 2 = 

And the output was:

 ,