CS 464

Introduction to Machine Learning

Fall 2023

Homework 3

Due: December 29, 2023 23:59 (GMT+3)

CIFAR-100 Inpainting

Homework Description

In this assignment, you are asked to design and train a convolutional neural network model for the image inpainting task. In short, inpainting is a process of filling in the missing parts of an image. You will be applying this task on the preprocessed CIFAR-100 dataset, which is created for this homework by processing the original CIFAR-100 images. It contains RGB real-life images with the size of 28x28 pixel resolution. You can see a subset of the dataset below. The download link of the dataset is provided in the following parts.

Using PyTorch is mandatory for this assignment. You are requested to **submit only a single *.ipynb file** in your submissions (no report needed). If you want to provide further explanations about your work, you can add Markdown cells for this purpose. From this link, you can get familiar with the Markdown syntax if you need. Upload your homework with the following filename convention: **<BilkentID>_<Name>_<Surname>.ipynb**

Note that this assignment needs a CUDA-enabled GPU to be able to train the models in a reasonable time. If you do not have one, it is suggested to use the Colab environment.

Contact: Ahmet Burak Yıldırım

Importing the Libraries

In the cell below, some utilities that you can make use of in this assignment are imported. You can edit these imports considering your implementation as long as you use PyTorch.

```
import torch
from torch import nn, optim
from torch.utils.data import DataLoader, Dataset
import torchvision
from torchvision import transforms
from PIL import Image
#from tqdm import tqdm
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import random
import os
```

Environment Check

In the cell below, you can test whether hardware acceleration with GPU is enabled in your machine or not. If it is enabled, the printed device should be 'cuda'.

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print('Current device:', device)

if device.type == 'cuda':
    print('GPU Name:', torch.cuda.get_device_name(0))
    print('Total GPU Memory:', round(torch.cuda.get_device_properties(0).total_memory/1024**3,1), 'GB')
```

```
Current device: cuda
GPU Name: Tesla T4
Total GPU Memory: 14.7 GB
```

Setting Library Seeds for Reproducibility

DO NOT CHANGE

To make a fair evaluation, the seed values are set for random sampling methods in PyTorch, NumPy, and Python random library. Please do not change these values.

```
def seed_everything(seed):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.backends.cudnn.deterministic = True
```

```
seed_everything(464)
```

Preparing the Dataset

The CIFAR-100 dataset is downloadable from this link. If you are using Colab or a Linux machine, you can uncomment and run the below cell to download and extract the dataset automatically.

```
import gdown # Library to download files from Google Drive
!gdown 1KiymtjUjuEJjUTO_qB9ifpLC_UvJEhoL # Google Drive ID of the zip file to be downloaded
!unzip -oq cifar100.zip # Unzip the file downloaded. Options -o and -q overwrites the files if exists already and d
```

```
Downloading...
From: https://drive.google.com/uc?id=1KiymtjUjuEJjUTO_qB9ifpLC_UvJEhoL
To: /content/cifar100.zip
100% 64.1M/64.1M [00:00<00:00, 118MB/s]
```

Implementing a Custom Dataset [25 Points]

In this part, you are requested to implement a custom PyTorch dataset class that reads CIFAR-100 images from a dataset split folder. There are two split folders called train and test in the dataset. The model class should take the root directory of a split in the __init__ function and read the images accordingly. Before returning the requested images, you should apply the following steps:

- Apply bicubic interpolation using PIL to resize the images from (28,28) to (32,32) resolution.
- Convert images to Tensor object
- Normalize tensor values to scale them in the range of (-1,1)

Note that reading images in the __getitem__ function makes the training process slow for this dataset because reading such small-sized images as a batch is slower than the forward pass process of a simple neural network. Therefore, it is suggested to read and store the images in an array in the __init__ function and return them in the __getitem__ function when they are requested by the DataLoader object.

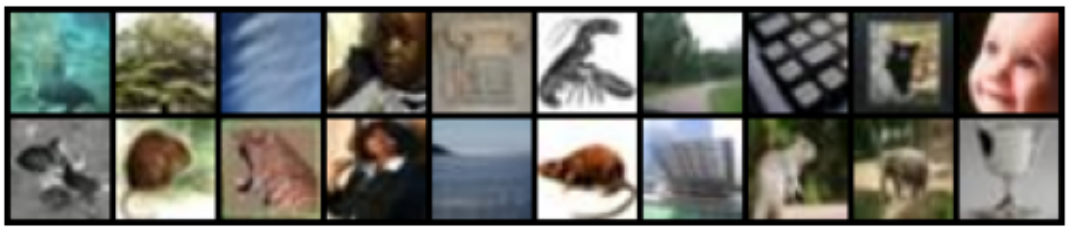
```
class CifarDataset(Dataset):
    def __init__(self, root_dir):
        self.root_dir = root_dir
        self.images = []
        self.labels = []

        # for folders in os.listdir(self.root_dir):
        folders = os.listdir(self.root_dir)

        # Assuming you have a list of unique class names
        class_names = folders
        # need to remove .DS_Store

        #class_names.remove('.DS_Store') # need this for mac
        # Create a mapping from class names to numerical labels
        class_name_to_label = {class_name: label for label, class_name in enumerate(set(class_names))}
        for folder in folders:
            # remove folders starts with . (like .ipynb_checkpoints)
            if folder.startswith('.'):
                folders.remove(folder)
                continue
            files = os.listdir(os.path.join(self.root_dir, folder))
            for file in files:
                if file.endswith(".jpg"):  # Images are in jpg format
                    file_path = os.path.join(self.root_dir, folder, file)
                    label_str = folder  # Extract label from the filename
                    label = class_name_to_label[label_str] # Numerical label
                    self.labels.append(label)
                    self.images.append(file_path)

    # Define transformations
    self.transforms = transforms.Compose([
        transforms.Resize((32, 32), interpolation=Image.BICUBIC),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

    # For faster training process, we can pre-load all the images and labels into memory as suggested
    for index in range(len(self.images)):
        img_path = self.images[index]
        label = self.labels[index]
        img = Image.open(img_path).convert('RGB')
        img = self.transforms(img)
        self.images[index] = img

def __len__(self):
    return len(self.images)

def __getitem__(self, data_id):
    return self.images[data_id]
```

Create a dataset and a data loader object for training and test splits. Set batch sizes to 64 and 512 for training and test data loaders, respectively. Enable shuffling in the training data loader and disable it in the test data loader.

```
train_dataset = CifarDataset('cifar100/train') # TODO
train_dataloader = DataLoader(train_dataset, batch_size=64, shuffle=True) # TODO

test_dataset = CifarDataset('cifar100/test') # TODO
test_dataloader = DataLoader(test_dataset, batch_size=512, shuffle=False) # TODO
```

Do not change the below code. If your implementation is correct, you should be seeing a grid of CIFAR-100 images properly.

```
# Uncomment the cell when the dataloader is ready

images = next(iter(train_dataloader)) # Taking one batch from the dataloader
images = (images + 1) / 2
grid_img = torchvision.utils.make_grid(images[:20], nrow=10)
plt.axis('off')
plt.imshow(grid_img.permute(1, 2, 0))
plt.show()
```



Constructing Convolutional Autoencoder Network [35 Points]

Autoencoder networks learn how to compress and reconstruct input data. It consists of two networks called the encoder and the decoder. The encoder network compresses the input data, and the decoder network regenerates the data from its compressed version. In this part, you are requested to implement an autoencoder model using convolutional layers. The architecture of the convolutional autoencoder is shown in the below figure.

The (in_channel, out_channel) pairs of the layers should be defined as follows:

Encoder:

- (3, 16)
- (16, 32)
- (32, 64)

Decoder:

- (64, 32)
- (32, 16)
- (16, 3)

You are free to choose the kernel and padding sizes of the layers. In each layer, 2D batch normalization should be applied and the resulting values should be passed through a LeakyReLU layer with slope 0.2, which is the activation function. Since the image pixel value range is set to [-1,1] in the dataset, the outputs should be bounded so. Therefore, you should be using a Tanh activation function in the last layer instead of the normalization and LeakyReLU layers.

In the encoder part of the network, use max pooling in each layer for decreasing the resolution by half. The stride size should be set to one for the convolution layers. In the decoder network, use transposed convolution (deconvolution) layers with stride two for increasing the resolution back.

```
class CifarAutoencoder(nn.Module):
    def __init__(self):
        super(CifarAutoencoder, self).__init__()
        # Encoder layers
        self.encoder = nn.Sequential(
            # Padding = (Kernel size - 1) / 2
            # This padding value used to keep dimensions same at convolutional layers
            # Used kernel sizes in increasing fashion in encoder and reverse in decoder.

            nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(16),
            nn.LeakyReLU(0.2),
            nn.MaxPool2d(kernel_size=2, padding=0),

            nn.Conv2d(16, 32, kernel_size=5, stride=1, padding=2),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(0.2),
            nn.MaxPool2d(kernel_size=2, padding=0),
```

```
            nn.Conv2d(32, 64, kernel_size=7, stride=1, padding=3),
            nn.BatchNorm2d(64),
            nn.LeakyReLU(0.2),
            nn.MaxPool2d(kernel_size=2, padding=0)
        )

        # Decoder layers
        self.decoder = nn.Sequential(
            # To correctly double dimensions, output_padding is used.
            # Padding = (Kernel size - 1) / 2
            # Output_padding = 1
            nn.ConvTranspose2d(64, 32, kernel_size=7, stride=2, padding=3, output_padding=1),
            nn.BatchNorm2d(32),
            nn.LeakyReLU(0.2),

            nn.ConvTranspose2d(32, 16, kernel_size=5, stride=2, padding=2, output_padding=1),
            nn.BatchNorm2d(16),
            nn.LeakyReLU(0.2),

            nn.ConvTranspose2d(16, 3, kernel_size=3, stride=2, padding=1, output_padding=1),
            nn.Tanh()  # Tanh activation
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x

model = CifarAutoencoder().to(device)

from torchsummary import summary

summary(model, (3, 32, 32))
# As we can see from summary, dimensions match correctly
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 16, 32, 32]             448
       BatchNorm2d-2           [-1, 16, 32, 32]              32
         LeakyReLU-3           [-1, 16, 32, 32]               0
         MaxPool2d-4           [-1, 16, 16, 16]               0
            Conv2d-5           [-1, 32, 16, 16]          12,832
       BatchNorm2d-6           [-1, 32, 16, 16]              64
         LeakyReLU-7           [-1, 32, 16, 16]               0
         MaxPool2d-8             [-1, 32, 8, 8]               0
            Conv2d-9             [-1, 64, 8, 8]         100,416
      BatchNorm2d-10             [-1, 64, 8, 8]             128
        LeakyReLU-11             [-1, 64, 8, 8]               0
        MaxPool2d-12             [-1, 64, 4, 4]               0
  ConvTranspose2d-13             [-1, 32, 8, 8]         100,384
      BatchNorm2d-14             [-1, 32, 8, 8]              64
        LeakyReLU-15             [-1, 32, 8, 8]               0
  ConvTranspose2d-16           [-1, 16, 16, 16]          12,816
      BatchNorm2d-17           [-1, 16, 16, 16]              32
        LeakyReLU-18           [-1, 16, 16, 16]               0
  ConvTranspose2d-19            [-1, 3, 32, 32]             435
             Tanh-20            [-1, 3, 32, 32]               0
================================================================
Total params: 227,651
Trainable params: 227,651
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 0.90
Params size (MB): 0.87
Estimated Total Size (MB): 1.78
----------------------------------------------------------------
```

Implementing the Training Loop [15 Points]

Define your training loop in this function. In the following parts, this function will be called to train the convolutional autoencoder. The input arguments are provided below. Apply the training progress and return a list of losses that are calculated on each epoch. You should sum the iteration losses up during an epoch and take the mean of them to calculate the running loss of that epoch.

To be able to learn inpainting, you should mask the input images as follows:

Simply, you should set the input tensor columns starting from 16 to 32 as -1 (black pixel). For the loss function, you should use the original image as the ground truth image so that the network learns how to fill the masked area of the input image and output the restored image. Before assigning the black pixels, do not forget to clone the original image to use it later in the loss function.

```
def train_model(model, train_dataloader, optimizer, loss_func, num_epochs):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)

    losses_per_epoch = []

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

        for batch in train_dataloader:
            inputs= batch

            # Check if the inputs is a tuple
            if isinstance(inputs, tuple):
                inputs = inputs[0]

            # Move inputs and labels to the device
            inputs = inputs.to(device)

            # Check if the labels is a tuple
            #if isinstance(labels, tuple):
            #    labels = labels[0]

            #labels = labels.to(device)

            # Mask the input images
            masked_inputs = inputs.clone()
            masked_inputs[:, :, :, 16:32] = -1  # Set the columns from 16 to 32 as black pixels

            masked_inputs = masked_inputs.to(device)

            optimizer.zero_grad()

            outputs = model(masked_inputs)
            loss = loss_func(outputs, inputs)  # Use the original image as ground truth

            loss.backward()
            optimizer.step()

            epoch_loss += loss.item()

        epoch_loss /= len(train_dataloader)
        losses_per_epoch.append(epoch_loss)

        print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {epoch_loss:.4f}")

    return losses_per_epoch
```

Implementing the Evaluation Function [15 Points]

Implement an evaluation function that returns the mean MSE calculated over the test dataset samples.

```
def evaluate_model(model, test_dataloader):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)

    criterion = nn.MSELoss()
    model.eval()

    total_loss = 0.0
    num_samples = 0

    with torch.no_grad():
        for batch in test_dataloader:
            inputs = batch

            # Check if the inputs is a tuple
            if isinstance(inputs, tuple):
                inputs = inputs[0]
            # Check if the labels is a tuple
            #if isinstance(labels, tuple):
            #    labels = labels[0]

            # Mask the input images
            masked_inputs = inputs.clone()
            masked_inputs[:, :, :, 16:32] = -1  # Set the columns from 16 to 32 as black pixels

            inputs, masked_inputs = inputs.to(device), masked_inputs.to(device)

            #labels = labels.to(device)

            outputs = model(masked_inputs)
            loss = criterion(outputs, inputs)  # Use the original image as ground truth

            total_loss += loss.item()
            num_samples += inputs.size(0)

    mean_loss = total_loss / num_samples
    return mean_loss
```

Inpainting Visualization Function

The below code will be used to visualize the outputs of the trained models later. **Do not change the codes in the cell**.

```
def visualize_inpainting(model, dataset):
    seed_everything(464)
    dataloader = DataLoader(dataset, batch_size=10, shuffle=True)
    images = next(iter(dataloader)) # Taking one batch from the dataloader
    images = images
    model.eval()
    with torch.no_grad():
      masked_images = images.clone()
      masked_images[:,:,:,16:] = -1
      inpainted_images = model(masked_images.cuda()).cpu()
    images = (images + 1) / 2
    masked_images = (masked_images + 1) / 2
    inpainted_images = (inpainted_images + 1) / 2
    images_concat = torch.cat((images, masked_images, inpainted_images), dim=2)
    grid_img = torchvision.utils.make_grid(images_concat, nrow=10)
    plt.axis('off')
    plt.imshow(grid_img.permute(1, 2, 0))
    plt.show()
```

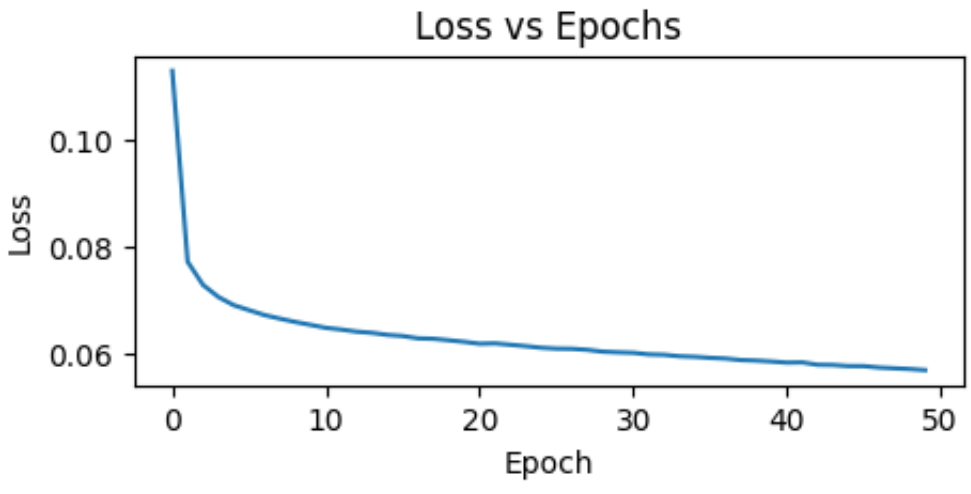
Training and Evaluating the Model [10 Points]

Define your loss function as MSE, set learning rate to 2e-4, create Adam optimizer, and set number of epochs to 50. Later, call the train_model function that you implemented. Visualize the returned losses on a plot (loss vs. epoch). Lastly, call evaluate_model function that you implemented and print the mean square error that your model reached on the test dataset. Also, call the visualize_inpainting function to observe the final inpainting results on the test dataset.

```
seed_everything(464)

# Set Model Parameters
model = CifarAutoencoder() ## Uncomment when the model is implemented
optimizer = optim.Adam(model.parameters(), lr=0.0002)
loss_func = nn.MSELoss()

# Train Model
history = train_model(model, train_dataloader, optimizer, loss_func, num_epochs=50)

# Evaluate Model
mse = evaluate_model(model, test_dataloader)
print(f'\nAverage MSE over Test Dataset is: {mse}')

# TODO
```

```
Epoch 1/50, Loss: 0.1131
Epoch 2/50, Loss: 0.0773
Epoch 3/50, Loss: 0.0729
Epoch 4/50, Loss: 0.0706
Epoch 5/50, Loss: 0.0691
Epoch 6/50, Loss: 0.0682
Epoch 7/50, Loss: 0.0672
Epoch 8/50, Loss: 0.0666
Epoch 9/50, Loss: 0.0659
Epoch 10/50, Loss: 0.0654
Epoch 11/50, Loss: 0.0648
Epoch 12/50, Loss: 0.0645
Epoch 13/50, Loss: 0.0641
Epoch 14/50, Loss: 0.0639
Epoch 15/50, Loss: 0.0635
Epoch 16/50, Loss: 0.0633
Epoch 17/50, Loss: 0.0629
Epoch 18/50, Loss: 0.0628
Epoch 19/50, Loss: 0.0625
Epoch 20/50, Loss: 0.0622
Epoch 21/50, Loss: 0.0619
Epoch 22/50, Loss: 0.0619
Epoch 23/50, Loss: 0.0617
Epoch 24/50, Loss: 0.0614
Epoch 25/50, Loss: 0.0611
Epoch 26/50, Loss: 0.0609
Epoch 27/50, Loss: 0.0609
Epoch 28/50, Loss: 0.0607
Epoch 29/50, Loss: 0.0604
Epoch 30/50, Loss: 0.0603
Epoch 31/50, Loss: 0.0602
Epoch 32/50, Loss: 0.0599
Epoch 33/50, Loss: 0.0598
Epoch 34/50, Loss: 0.0595
Epoch 35/50, Loss: 0.0594
Epoch 36/50, Loss: 0.0592
Epoch 37/50, Loss: 0.0591
Epoch 38/50, Loss: 0.0588
Epoch 39/50, Loss: 0.0587
Epoch 40/50, Loss: 0.0585
Epoch 41/50, Loss: 0.0583
Epoch 42/50, Loss: 0.0584
Epoch 43/50, Loss: 0.0579
Epoch 44/50, Loss: 0.0579
Epoch 45/50, Loss: 0.0577
Epoch 46/50, Loss: 0.0577
Epoch 47/50, Loss: 0.0574
Epoch 48/50, Loss: 0.0573
Epoch 49/50, Loss: 0.0571
Epoch 50/50, Loss: 0.0569

Average MSE over Test Dataset is: 0.00012868229448795317
```

```
# Plot the training loss history
plt.figure(figsize=(5,2))
plt.plot(history)
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss vs Epochs')
plt.show()
```




```
visualize_inpainting(model, test_dataset) ## Uncomment when the model is trained
```

```
# Results looks good, after trying different kernel sizes, this is best I can find.

# Too large kernel sizes cause longer train time and possibly overfitting.

# Too small kernel sizes cause underfitting.

# I think this works good.
```