Super_Resolution_CI_Model

February 10, 2021

1 Super-resolución 4x:

1.1 2.- Modelado y Entranamiento:

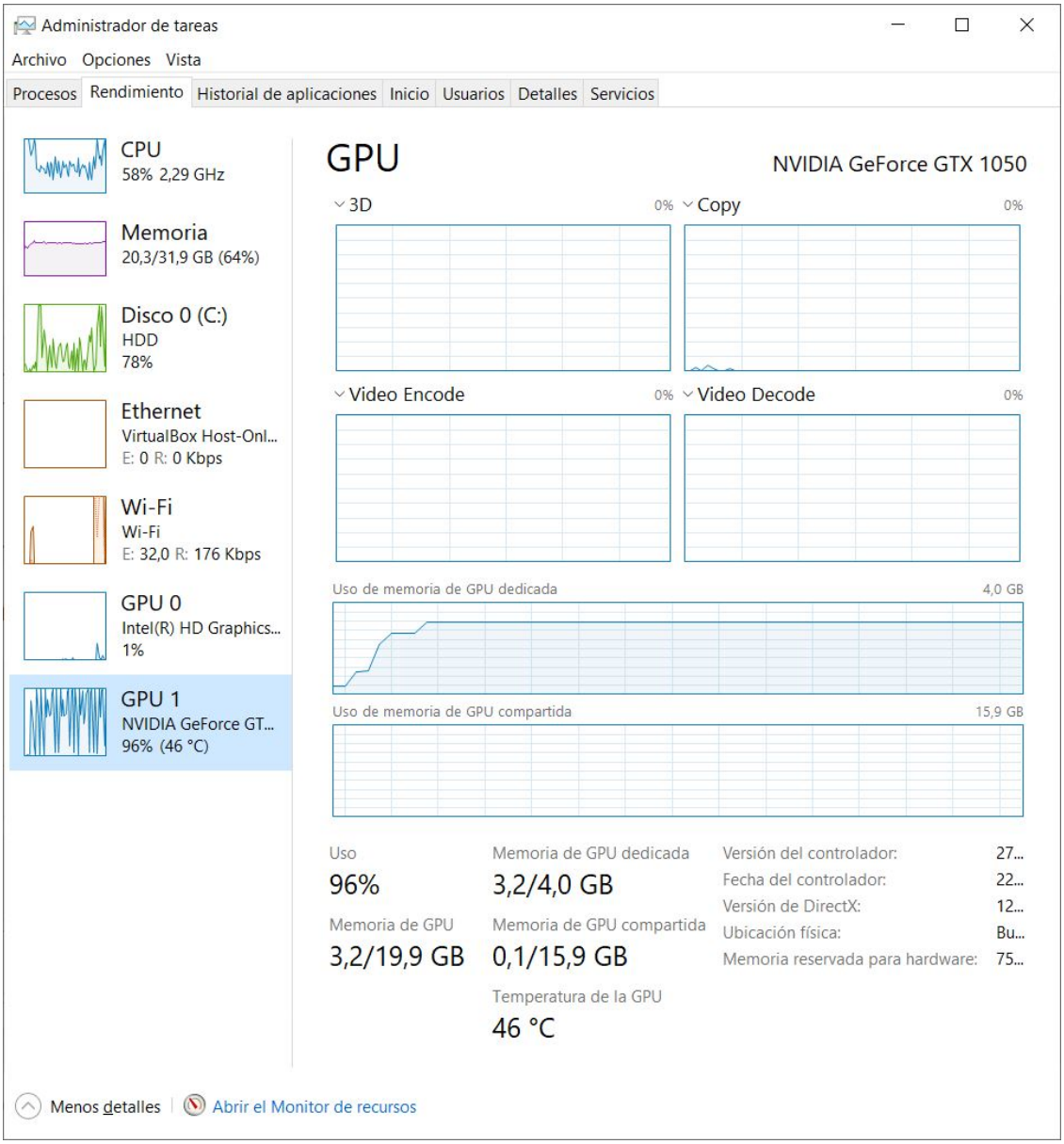
Ejercicio de curso para la asignatura de Computación Inteligente perteneciente al Máster Universitario en Sistemas Inteligentes y Aplicaciones Numéricas para la Ingeniería (MUSIANI) en el curso 2020/21, realizado por Juan Sebastián Ramírez Artiles.

El ejercicio consiste en implementar el método de superresolución en imágenes descrito en el paper *Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network*. Concretamente, en este notebook se ha implementado un escalado de cuatro aumententos. El dataset usado fue el DIV2X de libre descarga. Las imgágenes usadas son las del track1 con un tamaño para las imágenes de alta resolución recortadas a 2040x1304.

[1]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
import matplotlib.pyplot as plt
import torch
from torchvision import datasets, transforms
import torchvision.utils as vutils
from torch.utils.data import DataLoader, TensorDataset
import torch.nn.functional as F
import numpy as np
import torch.nn as nn
import torch.optim as optim
```

Las imágenes se cargan de cuatro directorios, cada directorio debe contener un subdirectorio de etiqueta donde se situarán las imágenes, en este caso sólo habrá un subdirectorio, por lo tanto una etiqueta cada uno. Las etiquetas de los dataset se ignorarán ya que no se van a usar. Las imágenes reales se situan en train_y_hr y valid_y_hr y tienen un tamaño de 2040x1304. Las imágenes reducidas tienen un tamaño cuatro veces menor. Para el acondicionamiento de las imágenes se usó el notebook Super_Resolution_CI_Preprocessed.

El programa se ejecutó en una máquina con un procesador Intel Core i7-7700HQ a 2.80GHz con una tarjeta de vídeo NVIDIA GeForce GTX 1050 de 4GB de memoria dedicada y con 32 GB de memoria RAM.



[2]:
```
workers = 8
ngpu = 1
beta1 = 0.5
lr = 0.2
bs = 14
epochs = 20

path_train_x = "images/train/train_x"
path_train_y = "images/train/train_y_hr"

path_valid_x = "images/valid/valid_x"
path_valid_y = "images/valid/valid_y_hr"
```

[3]:
```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

imgs_train_x = datasets.ImageFolder(path_train_x, transform = transform)
imgs_train_y = datasets.ImageFolder(path_train_y, transform = transform)

imgs_valid_x = datasets.ImageFolder(path_valid_x, transform = transform)
imgs_valid_y = datasets.ImageFolder(path_valid_y, transform = transform)
```

El dataset se dividió en 611 imágenes para entrenamiento y 75 imágenes para validación.

[4]:
```
print(len(imgs_train_x))
print(len(imgs_train_y))
#imgs_train_x.classes
#train_ds = TensorDataset(imgs_train_x, imgs_train_y)
```

```
611
611
```

[5]:
```
imgs_train_x_dl = DataLoader(imgs_train_x, batch_size = bs, num_workers =␣
↪workers)
imgs_train_y_dl = DataLoader(imgs_train_y, batch_size = bs, num_workers =␣
↪workers)

imgs_valid_x_dl = DataLoader(imgs_valid_x, batch_size = bs, num_workers =␣
↪workers)
imgs_valid_y_dl = DataLoader(imgs_valid_y, batch_size = bs, num_workers =␣
↪workers)
```

[6]:
```
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else␣
↪"cpu")
```

El modelo hace uso de la capa PixelShuffle que se encarga de realizar el escalado a nivel subpíxel tomando como entrada un tensor de tamaño $[Batch, Channels * R^2, H, W]$ y reordenándolo como $[Batch, Channels, H * R, W * R]$

Se probaron una variedad de modelos diferentes. Se usaron kernels de 3x3, de 5x5 y de 7x7, siendo estos últimos los que mejor resultados dieron. También se probaron diferentes configuraciones de red, añadiendo capas convolutivas y modificando las funciones de activación. Esta configuración resultó la más adecuada.

En el proceso de entrenamiento se usaron gran variedad de combinaciones de tamaños de batches y de cantidad de épocas. Se empezó con tamaños de batch de 8, se fue subiendo hasta 16 y finalmente se dejó en 14, más de esta cantidad desborda la memoras de la GPU. En cuanto al número de épocas, se empezó con 20 iteraciones y se fue subiendo hasta la cantidad de 80. No obstante, al variar el learning rate se aceleró la convergencia, con lo que volví a la veinte iteraciones.

[7]:
```
class SuperResolution(nn.Module):
    def __init__(self):
        super().__init__()

        self.conv1 = nn.Conv2d(3, 12, kernel_size = 7, padding = 3)
        self.conv2 = nn.Conv2d(12, 12, kernel_size = 7, padding = 3)
        self.upsample = nn.PixelShuffle(upscale_factor = 2)

    def forward(self, xb):

        xb = torch.tanh(self.conv1(xb))
        xb = torch.tanh(self.conv2(xb))
        xb = self.upsample(xb)
        xb = torch.tanh(self.conv1(xb))
        xb = torch.sigmoid(self.conv2(xb))

        return self.upsample(xb)
```

[8]:
```
def preprocess(x, y):
    return x.to(device), y.to(device)
```

[9]:
```
def get_model():
    model = SuperResolution().to(device)
    return model, optim.SGD(model.parameters(), lr=lr, momentum=0.9)

loss_func = nn.MSELoss(reduction='mean')
```

[10]:
```
class WrappedDataLoader:
    def __init__(self, dl_x, dl_y, func):
        assert len(dl_x) == len(dl_y)
        self.dl_x = dl_x
        self.dl_y = dl_y
        self.func = func

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

    def __iter__(self):
        batches_x = iter(self.dl_x)
        batches_y = iter(self.dl_y)

        for b_x, _ in batches_x:
            b_y, _ = batches_y.next()
            yield (self.func(b_x, b_y))
```

[11]:
```
def loss_batch(model, loss_func, xb, yb, opt=None):
    loss = loss_func(model(xb), yb)
    if opt is not None:
        loss.backward()
        opt.step()
        opt.zero_grad()

    return loss.item(), len(xb)
```

[12]:
```
def fit(epochs, model, loss_func, opt, train_dl, valid_dl, val_losses):
    for epoch in range(epochs):
        model.train()
        for xb, yb in train_dl:
            loss_batch(model, loss_func, xb, yb, opt)

        model.eval()
        with torch.no_grad():
            losses, nums = zip(
                *[loss_batch(model, loss_func, xb, yb) for xb, yb in valid_dl]
            )
        val_loss = np.sum(np.multiply(losses, nums)) / np.sum(nums)
        val_losses.append(val_loss)

        print(epoch, val_loss)
```

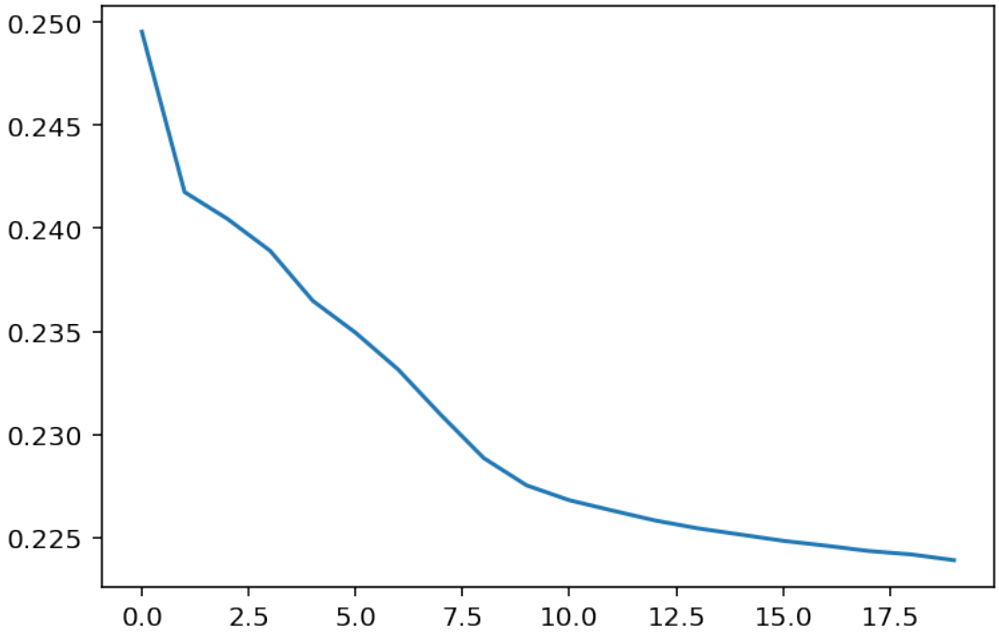
[13]:
```
train_dl = WrappedDataLoader(imgs_train_x_dl, imgs_train_y_dl, preprocess)
valid_dl = WrappedDataLoader(imgs_valid_x_dl, imgs_valid_y_dl, preprocess)

val_losses = []

model, opt = get_model()
fit(epochs, model, loss_func, opt, train_dl, valid_dl, val_losses)
```

```
0 0.249540270169576
1 0.2417554489771525
2 0.24045846700668336
3 0.23890738646189372
4 0.23647509972254435
5 0.23492557724316915
6 0.23313244104385375
7 0.230927152633667
8 0.22883734265963238
9 0.2275152798493703
10 0.22679356972376505
11 0.2262951143582662
12 0.22581315398216248
13 0.22543023506800333
14 0.2251266622543335
```

```
15 0.2248228359222412
16 0.2245865229765574
17 0.22432732860247295
18 0.22416173100471495
19 0.22388007005055746
```

Se puede observar en el gráfico que la red aprende bien.

[14]:
```
plt.plot(val_losses)
plt.show()
```



Para finalizar, salvo el modelo para poder usarlo en el notebook SR_restore_model.ipynb que se encargará de escalar las 75 imágenes de validación.

[15]:
```
torch.save(model, "SR_model_3.4.ml")
```

[ ]:
```
%%javascript
Jupyter.notebook.session.delete();
```

```
<IPython.core.display.Javascript object>
```

[ ]: