CAI Classification Problem
Report 2: Implementation 1

Nil Crespo-Peiró, Paco Rahn, Amin Ranem

12. January 2021

1 Introduction

Once the planning phase has been done, our next steps were to load the dataset, to transform it from 25 fps (frames per second) to 1 fps and to set up the boilerplate code.
In the following sections the implementation details regarding the dataset structure and the boilerplate code will be discussed.

2 Implementation 1

In this section the first implementations will be described, beginning with a granular description of the data preprocessing and the dataset structure itself, followed by the model and agents that will be necessary to train a desired classifier. With the help of the dataset structure and the transformation method that has been used, it is possible to load such a big dataset like the `Cholec80` dataset consisting of 80 videos resulting in 70GB of data to train different models based on the videos. As stated in the previous reports, the source code used in this project can be found at the GitHub repository.

2.1 Preprocessing of the data

The first step was to download the whole `Cholec80` dataset and to get familiar with the structure in order to be able to load the videos and corresponding labels to preprocess them in the following ways:

1. Transform all videos from 25 fps to 1 fps
2. Transform the labels from `.txt` files into `.json` files in order to load the labels after the preprocessing step in a more generic way.

2.1.1 Transform Cholec80 dataset

The Cholec80 dataset has been loaded using the Python library MoviePy. With this library, a video can first be loaded by specifying its path, then be transformed by resetting the fps and finally be saved as a more or less new, modified video, in order to load these preprocessed videos before training. This approach is used to only transform the videos once, so the preprcoessed images can be used anytime, without preprocessing them on the fly:

```
import os
import moviepy.editor as mpy

# Define videos_path to videos and target_path
videos_path = '.../Cholec80/videos/'
target_path = '.../Cholec80_fps1/'

# Create directories if not existing
if not os.path.isdir(target_path):
    os.makedirs(target_path)

# Extract the filenames of the videos, ending with '.mp4'
filenames = [x.split('.')[0] for x in os.listdir(videos_path) if
'.mp4' in x and '._' not in x]

# Loop through extracted filenames
for filename in filenames:
    # Extract the video from videos_path (ending with '.mp4')
    video = mpy.VideoFileClip(os.path.join(videos_path,
                                           filename+'.mp4'))
    # Reduce fps to 1
    video = video.set_fps(1)
    # Save transformed video to target_path (ending with '.mp4')
    video.write_videofile(os.path.join(target_path,
                                       filename+'.mp4'))
```

2.1.2 Transform labels into JSON format

With this step in the preprocessing of the data, all the transformations need only to be done once, for instance, the right parsing of the one-hot vectors provided in the .txt file and transforming them into lists to be able to transfer them to torch.tensors can be easily done using .json files. Further, the number of frames needed to be divided by 25, since the videos have been transformed from 25 fps to 1 fps, which also needs to be considered in the labeling of the different one-hot vectors:

```
import os
import json
```

```python
# Define videos_path to videos and target_path
videos_path = '.../Cholec80/videos/'
target_path = '.../Cholec80_fps1/'

# Create directories if not existing
if not os.path.isdir(target_path):
    os.makedirs(target_path)

# Extract the filenames of the videos, ending with '.mp4'
filenames = [x.split('.')[0] for x in os.listdir(videos_path) if
'.mp4' in x and '._' not in x]

# Loop through extracted filenames
for filename in filenames:
    # Extract labels and save it
    label = open(os.path.join(labels_path, filename+'-tool.txt'))
    label_dict = dict()
    # Skip first line (Header)
    next(label)
    # Loop through file
    for line in label:
        line_split = line.split('\t')
        res = list()
        for elem in line_split:
            res.append(int(elem))
        # Divide number of fps by 25 since the fps has been
        # reduced from 25 to 1!
        label_dict['Frame: '+str(int(int(res[0])/25))] = res[1:]

    # Save json file
    with open(os.path.join(target_path, filename+'-tool.json'),
              'w') as fp:
        json.dump(label_dict, fp, sort_keys=False, indent=4)
```

With this method, an exemplary label file for a single video looks like the following:

```
{
      "Frame: 0": [1, 0, 1, 0, 0, 0, 0],
      "Frame: 1": [1, 0, 1, 0, 0, 0, 0],
          .
          .
          .
      "Frame: N": [0, 0, 0, 0, 0, 0, 0]
}
```

whereas the one-hot vector represents the tools that are currently present within the frame:

`[Grasper, Bipolar, Hook, Scissors, Clipper, Irrigator, Specimenbag]`

It is important to note, that the code snippets provided in this section can be retrieved from the GitHub repository, and that the video and label preprocessing step has been made in a combined for loop, not separately as visualised in this section.

2.2 Boilerplate code

In this section, the created boilerplate code will be discussed, i.e. the structure of the dataset will be presented, as well as the `PyTorch` dataset that is needed for training the models on a GPU using the `PyTorch Dataloader`.

2.2.1 Dataset structure (`Dataset, Instance`)

The dataset structure itself is a `Dataset` class that has been implemented to hold different `Instances`. In this case the `Dataset` would represent the `Cholec80` dataset, whereas a single video with its corresponding labels would represent an `Instance` of the `Dataset` that will be discussed in the following and can be retrieved from `dataset_classification.py`.

Instance Like previously mentioned, a single video with its corresponding labels have been internally defined as an `Instance` that takes a path to the video (`x_path`) and a path to the corresponding labels (`y_path`). Based on these informations, the video and labels will be loaded using the `torchvision` and `json` libraries from Python:

```
from cai.data.datasets.dataset import Dataset, Instance
import torchvision
import torch
import json

class ClassificationPathInstance(Instance):
    def __init__(self, x_path, y_path, name=None,
                 class_ix=0, group_id=None):
        r"""Classification instance.

            Args:
            x_path (str): path to image
            y_path (st): path to label for the tools present
            in video based on fps
        """
        assert isinstance(x_path, str)
        assert isinstance(y_path, str)
```

```
        # torchvision.io.read_video returns tuple of video,
        # audio and infos
        (x, _, _) = torchvision.io.read_video(x_path)
        # Resize tensor to (NUM_FRAMES x CHANNELS x HEIGHT
                            x WIDTH)
        x = torch.reshape(x, (x.size()[0],
                              x.size()[3],
                              x.size()[1],
                              x.size()[2]))
        with open(y_path, 'r') as fp:
            y = json.load(fp)

        # Transform label list into stacked torch.tensors.
        y_labels = dict()
        for key, value in y.items():
            y_labels[key] = torch.tensor(value)
        # Stack the one-hot vectors
        y_labels = torch.stack(list(y_labels.values()), dim=0)

        self.shape = x.shape
        super().__init__(x=x, y=y_labels, name=name,
                         class_ix=class_ix,
                         group_id=group_id)

    def get_subject(self):
        return self.x, self.y
```

Dataset A `Dataset` on the other hand consists of these instances that will be added one by one when the dataset will be initialized and loaded (see `ds_cholec80_classification.py` for clarification). The class is simply defined as follows while printing some metrics like the mean shape of the dataset:

```
class ClassificationDataset(Dataset):
    r"""Classification Dataset:
        A Dataset for classification tasks, that specific datasets
        descend from.

        Args:
        instances (list[ClassificationPathInstance]): a list of
        instances
        name (str): the dataset name
        mean_shape (tuple[int]): the mean input shape of the data,
        or None
        label_names (list[str]): list with label names, or None
        nr_channels (int): number input channels
        modality (str): modality of the data, e.g. MR, CT
```

```
        hold_out_ixs (list[int]): list of instance index to
        reserve for a  separate hold-out dataset.
        check_correct_nr_labels (bool): Whether it should be
        checked if the correct number of labels (the length
        of label_names) is consistent with the dataset.
    """
    def __init__(self, instances, name, mean_shape=None,
                 label_names=None, nr_channels=1,
                 modality='unknown', hold_out_ixs=[],
                 check_correct_nr_labels=False):
        # Set mean input shape and mask labels, if these
        # are not provided
        print('\nDATASET: {} with {} instances'.format(name,
                                                       len(instances)))
        if mean_shape is None:
            mean_shape, shape_std = du.get_mean_std_shape(instances)
            print('Mean shape: {}, shape std: {}'.format(mean_shape,
                                                         shape_std))
        self.mean_shape = mean_shape
        self.label_names = label_names
        self.nr_labels = 0 if label_names is None
                           else len(label_names)
        self.nr_channels = nr_channels
        self.modality = modality
        super().__init__(name=name, instances=instances,
            mean_shape=mean_shape, output_shape=mean_shape,
            hold_out_ixs=hold_out_ixs)
```

PyTorch Dataset Since the dataset described so far includes only the videos, the format needs to be transfered to PyTorch in order to be able to load the data into the `Dataloader` for training the models. The corresponding classes for 2 dimensional data, meaning image slices from a video and 3 dimensional data, i.e. whole videos as an input to the model can be retrieved from `pytorch_classification_dataset.py` and will not be discussed further at this point.

In the following, the models and agents will be briefly discussed.

2.2.2 Models and Agents

The implemented models that will be evaluated against can be retrieved from `CNN.py` as well as agents for different models (see here) that train a model based on an optimizer, loss function and dataloader are crucial for this Classification task and will be presented in the following.

Models In the `CNN.py` file, all models that will be used during this project will be defined in order to create a pool of models to select the best suited one based on the `Cholec80` dataset. Currently, 4 different models have been collected:

- AlexNet (for transfer learning)
- VGG19BN (for transfer learning)
- EndoNet (as a Baseline)
- CNN_Net2D (for 2 dimensional data, meaning one single image from a video with the shape: `nr_channels x width x height`)
- CNN_Net3D (for 3 dimensional data, meaning one a whole video with the shape: `nr_frames x nr_channels x width x height`)

Note that probably only the 2 dimensional models will be used for training, since training on whole video volumes leads to a high amount of memory usage and is respectively computationally heavy.

Agents Based on the models, a corresponding agent needs to be defined. In this case we have the following agents:

- AlexNetAgent (an agent for AlexNet models)
- ClassificationAgent (an agent for CNN models)

In each agent, the `train` function needs to be implemented in order to be able to train a specific model based on the transmitted options (dataset, optimizer, loss function, nr_epochs, etc.). Further a `test` function needs to be provided within these classes to be able to evaluate a pre-trained model using the transmitted test dataset.