Deep Learning Exercises [DL E]
Exercise 4
PyTorch for Classification

DL Tutors Team
January 16, 2021

Important: As the training of Neural Networks requires a lot of computational effort, we strongly recommend to use machines with GPUs. The university does provide such machines, however since the CIP-Pools are still closed, the computers have to be accessed remotely. Information on this is outlined in the **"troubleshooting_cip.pdf"** file to this exercise which can be found on StudOn.

PyTorch for Classification

In the previous exercises, we have implemented the constituents of a deep learning framework. Now we will use the knowledge we gathered while doing so. We will implement a version of the common convolutional neural network architecture **ResNet** and the necessary workflow surrounding deep learning algorithms using the open source library **PyTorch**.

PyTorch allows to build data flow graphs for numerical computations. A graph consists of **nodes**, which represent mathematical operations (e.g. convolutions), and edges, which represent the data arrays (tensors).

We will use our implementation to detect defects on solar cells. To this end, a dataset containing images of solar cells is provided along with the corresponding labels. We will complement the baseline implementation task with an open **classification challenge**. During the challenge period, the best results of each team will be listed in an online leader board.

1 Dataset

Solar modules are composed of many solar cells that are subject to degradation, causing different types of defects. Defects may occur during transportation or installation of modules as well as during operation, for example due to wind, snow load or hail. Many of the defect types can be found by visual inspection. A common approach is to take electroluminescence images of the modules. To this end, a current is applied to the module, causing the silicon layer to emit light in the near infrared spectrum. This light is then captured by specialized cameras.

In this exercise, we focus on two different types of defects (see Fig. 1):

1. **Cracks**: The size of cracks may range from very small cracks (a few pixels in our case) to large cracks that cover the whole cell. In most cases, the performance of the cell is unaffected by this type of defect, since connectivity of the cracked area is preserved.

2. **Inactive regions**: These regions are mainly caused by cracks. It can happen when a part of the cell becomes disconnected and therefore does not contribute to the power production. Hence, the cell performance is decreased.

Of course, the two defect types are related, since an inactive region is often caused by cracks. However, we treat them as independent and only classify a cell as cracked if the cracks are visible (see Fig. 1, right example).

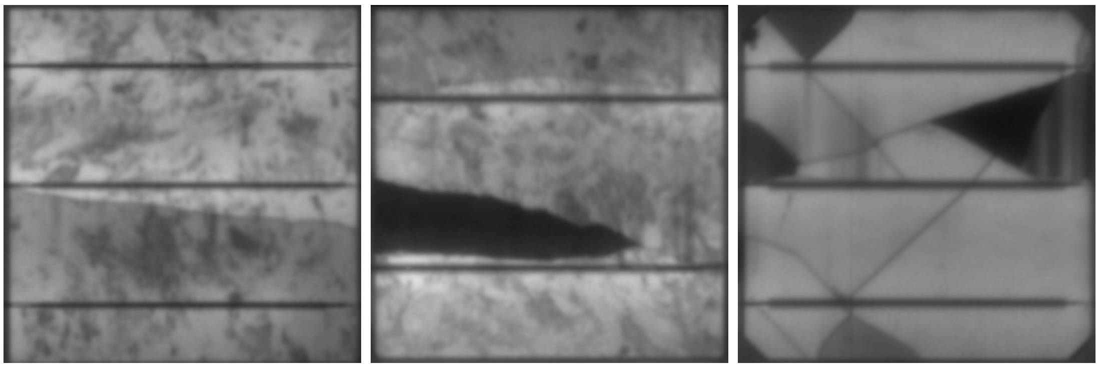

Figure 1: Left: Crack on a polycrystalline module; Middle: Inactive region; Right: Cracks and inactive regions on a monocrystalline module

The images are provided in `png`-format, and are collected in a zip folder. The filenames and the corresponding labels are listed in the csv-file *data.csv*. Each row in the csv-file contains the path to an image in our dataset and two numbers indicating if the solar cell shows a "crack" and if the solar cell can be considered "inactive".

2 Preparation

We provide a skeleton that you will complete during this assignment and a few unit tests as an additional guide line. Furthermore, this skeleton contains a *environment.yml* file that lists the necessary software packages for a conda environment (see conda documentation).

Keep in mind though, some of the listed packages in "environment.yml" file may require hardware support. Thus, if you work on your own machine, not all functionality of your program might be able to be tested. Therefore, we recommend using your own machine for testing and debugging and accessing a computer of the university to actually train your model. For specifics, please read the file **"troubleshooting_cip.pdf"** .

The computers in the **CIP-Pools** have already the required packages installed. However, before using **PyTorch**, one has to first activated via the terminal. E.g. by issuing

```
module load python3
module load torch
```

3 Data and Evaluation

When you start working on a new machine learning problem, you have to deal with the format of the given data collection and implement a pipeline to load, preprocess and augment the data.

Task

- Implement a container for our dataset in the file **data.py**, i.e., a class `ChallengeDataset` which inherits from the `torch.utils.data.Dataset` class and provides some basic functionalities. Example code for this can be found here.
- The constructor of `ChallengeDataset` receives two parameters: First, a pandas.dataframe "data" – a container structure that stores the information found in the file "data.csv". The second parameter is a flag "mode" of type `String` which can be either "val" or "train".
  Furthermore, create a member "self._transform" of type "tv.transforms.Compose". It takes a list of `torchvision.transforms` as a parameter, which should include at least the following: `ToPILImage()`, `ToTensor()` and `Normalize()`.
- Overwrite the method `__len__(self)`. It returns the length of the currently loaded data.
- Overwrite the method `__getitem__(self,index)`, which returns the sample as a tuple: the image and the corresponding label. Since our raw data is grayscale you need to convert the image to rgb using the `skimage.color.gray2rgb(*args)` function. Before returning the sample, perform the transformations specified in the transform member. The two return values need to be of type `torch.tensor`.

Notes

- The `Normalize` transformation requires the mean and standard deviation of your data. Both are given in the skeleton.
- The Compose object allows to easily perform a chain of transformations on the data. Among other aspects, this is interesting for data augmentation. In the transpose package, you can find different augmentation strategies. Consider creating two different transforms based on whether you are in the training or validation dataset.
- You can test your implementation using the corresponding test in the *PytorchChallengeTests.py* file.

4 Architecture

In this exercise, you will implement a variant of the *ResNet* architecture. The details of the architecture are specified in Tab. 1.

The main component of *ResNet* are blocks that are augmented by skip connections. We name those blocks `ResBlock(out_channels, stride)`. For the our variant of *ResNet*, each `ResBlock` consists of a sequence of (`Conv2D`, `BatchNorm`, `ReLU`) that is repeated twice.

Within the `ResBlock`, the number of output channels for `Conv2D` is given by the argument `out_channels`. The stride of the first `Conv2D` is given by `stride`. For the second convolution, no stride is used.

All `Conv2D` layers within a `ResBlock` have a filter size of 3.

Finally, the input of the `ResBlock` is added to the output. Therefore, the size and number of channels needs to be adapted. To this end, we recommend to apply a 1×1 convolution to the input with stride and channels set accordingly. Also, we recommend to apply a batchnorm layer before you add the result to the output.

Task

Implement the *ResNet* architecture according to the specification in Tab. 1 in the file "model.py". The model should inherit from the base class `torch.nn.Module`. Overwrite the necessary methods to allow training your model.

Model	Layers
ResNet	
	`Conv2D(64, 7, 2)`
	`BatchNorm()`
	`ReLU()`
	`MaxPool(3, 2)`
	`ResBlock(64, 1)`
	`ResBlock(128, 2)`
	`ResBlock(256, 2)`
	`ResBlock(512, 2)`
	`GlobalAvgPool()`
	`Flatten()`
	`FC(2)`
	`Sigmoid()`

Table 1: Architectural details for our *ResNet*. Convolutional layers are denoted by `Conv2D(out_channels, filter_size, stride)`. Max pooling is denoted `MaxPool(pool_size, stride)`. `ResBlock(channels_out, stride)` denotes one block within a residual network. Fully connected layers are represented by `FC(out_features)`.

5 Training

The training process consists of alternating between training for one epoch on the training dataset (*training step*) and then assessing the performance on the validation dataset (*validation step*). After that, a decision is made if the training process should be continued. A common stopping criterion is called EarlyStopping with the following behaviour: If the validation loss does not decrease after a specified number of epochs, then the training process will be stopped. This criterion will be used in our implementation and should be realised in *trainer.py*.

Task

Implement the class **Trainer** in “trainer.py” according to the comments.

6 Put it together

Now that we have all of the individual parts of our pipeline ready, we can assemble them in "train.py". This means, we have to first load our data, split the data into train and test set, create our model and set an optimizer and the loss function, before we can start training.

There are many different classification tasks requiring also different loss-functions. In a multi-class setting, the *SoftMax*-loss is often used. This loss assumes that the class assignments are **mutually exclusive**. This however does not hold for our problem, as your classes "crack" and "inactive" are not **mutually exclusive**.

Task

Make yourself familiar with loss functions that are suitable to use in a multi-label setting. Keep in mind, that some loss-functions perform internally a "sigmoid" operation. However, we already included this in our model. Implement the missing parts in "train.py" according to the comments.

7 Train and tune hyperparameters

At this point, we are able to start training the model. We might need to adjust the hyperparameters to obtain good results.

Task

Train the model and watch the evaluation measures on the validation split. Observe and document how changes in hyperparameters affect the performance. Determine good hyperparameter settings by experiment. Most commonly you start with the default values and alternate for example the learning rate by a change factor of 10. Note that the learning rate and the batch size are highly dependent on each other. For the ratio between training and validation data try to get a validation set which is as small as possible while still being a representative subset.

8 Save model and submit

With the skeleton, we provide the ability to save a trained model using the _save_onnx(epoch) function. An example file on how to use it is defined in export_onnx.py. In it, a onnx-file is automatically created that can be submitted to the online evaluation server. To make this work, it is required that the in- and outputs of your model have a fixed name. Please do not change the specified names in "train.py".

From TBD we will make the online leaderbord available on http://lme156.informatik.uni-erlangen.de/dl-challenge. Note that you must be in the **university network** to access the service (VPN is sufficient).

Task

When everything is done, upload your code on StudOn and create an account for the online leaderboard and submit your model.

In order to pass this exercise, you need to submit a model that reaches a **mean F1 score of at least 0.60** until the deadline mentioned in StudOn.

In order to submit a model, you need to be part of a team. Open the team-page to create a team or join an existing team. Note that only one group member should create the team. The other person can then join the team. If you are working alone, you need to create a one-person team.

Finally, you can start submitting jobs to the test server. Please note that we will do the final evaluation on a second test set that will not be available on the evaluation service during the challenge period. Therefore, you should avoid optimizing your parameters using the feedback from the evaluation server. You should use your validation split for that.

After the deadline, we will have an open challenge. You may experiment with alternative architectures, pretraining, different loss functions, and much more. Be creative!