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

**K. Breininger S. Vesal**
**M. Gu Z. Pan N. Maul**
**Z. Yang L. Reeb**
**F. Thamm T. Würfl**
**Jan 15, 2020**

# 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 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. The solar cells are subject to degradation, causing many 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 electroluminescense 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).

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 *train.csv*. We will use only the labels "crack" and "inactive". You may ignore the additional label "poly_wafer".

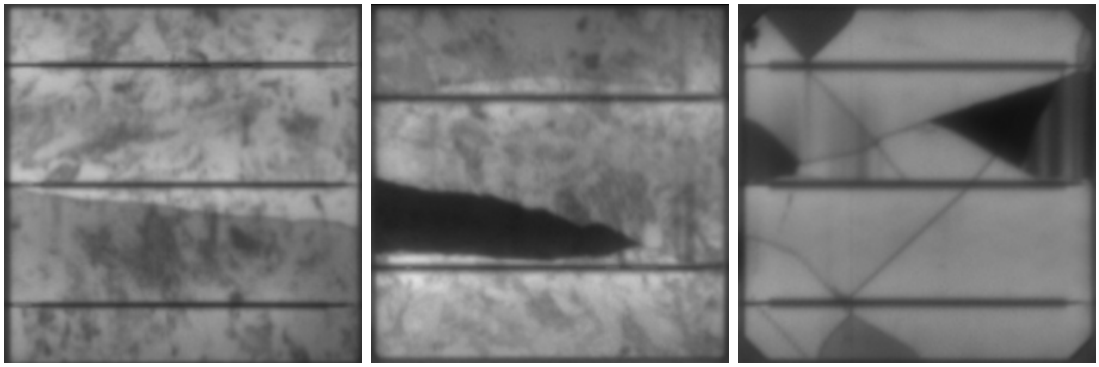

Figure 1: Left: Crack on a polycristalline module; Middle: Inactive region; Right: Cracks and inactive regions on a monocristalline module

## 2 Preparation

We provide a skeleton that you will complete during this assignment. This skeleton contains a *environment.yml* file that lists the necessary software packages for a conda environment (see conda documentation).

In the **CIP pool**, the required packages are already installed. However, you need to enable PyTorch 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 and preprocess the data correctly. You also have to implement evaluation routines that you need to assess the performance of your model.

### Task

- Implement a dataset loader 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 four parameters: A mode flag of type `String` which can be either "val" or "train", the path to the csv file, a parameter which controls the split between your train and validation data, and an object of type `torchvision.transforms.Compose` with a default value of `tv.transforms.Compose(tv.transforms.ToTensor()])`.
  **Note:** 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.

- The *train.csv* file consists of a table with the local file paths and the labels for each data sample. It can for example be read using the pandas library.

- 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`.

- Overwrite the method `__len__(self)`. It returns the length of the currently loaded data split.

- Additionally, implement a method `pos_weight()`. It should return a `torch.tensor` that contains weights

$$w_i = \frac{\sum_j 1 - y_{i,j}}{\sum_j y_{i,j}} \tag{1}$$

  for the positive samples of each class $i$, calculated over the ground truth labels $y_{i,j}$ of all samples $j$ in the training set. In other words it is the ratio of negative to positive samples. It allows to deal with unbalanced classes, as positive and negative samples are

| Model | Layers | Notes |
|---|---|---|
| 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)` | |

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)`.

weighted differently. You will need this value as an input for your loss function to ease convergence.

- Implement two functions `get_train_dataset()` and `get_validation_dataset()` **outside** of the `ChallengeDataset` class. First, create a `torchvision.transforms.Compose` object which includes at least the following `torchvision.transforms`: `ToPILImage()`, `ToTensor()` and `Normalize()`. Both functions return an object of type `ChallengeDataset` in the respective mode and initialized with the previously created `Compose` object.

  **Note:** The `Normalize` transformation requires the mean and standard deviation of your data. Both are given in the skeleton. You can test whether you normalized correctly by using the *test_dataloading.py* file.

**Hint:** You can use `sklearn.model_selection.train_test_split(*args)` for splitting the data into a train and a validation set. Set the parameter `random_state` to obtain a reproducible data split.

## 4 Architectures

In this exercise, you will implement a *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 for `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 $1 \times 1$ 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/resnet.py". The model should inherit from the base class `torch.nn.Module`. Overwrite the necessary methods to allow training your model.

## 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 will be continued continued. We will decide automatically if the training process is continued or cancelled. A common stopping criterion is to stop training if the validation loss did not decrease for a specified number of epochs. We will use that criterion in our implementation. It is defined in the file *stopping.py*.

**Task**

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

## 6 Put it together

In order to train the model, we need to decide for a loss function. In multi-class problems, *SoftMax*-loss is often used as loss function. This assumes that the class assignments are mutually **exclusive**. However, this assumption does not hold for multi-label problems.

**Task**

Make yourself familiar with loss functions that can be used to train for multi-label problems. Then, 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.

## 8 Save model and submit

With the skeleton, we provide the ability to save a trained model using the _save_onxx(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 January 26 on, we will make the online leaderbord available on https://lme156.informatik.uni-erlangen.de/dl-challenge. Note that you must be in the **university network** to access the service (VPN is sufficient).

### Task

Create an account for the online leaderboard. In order to submit jobs, 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.

In order to pass this exercise, you need to submit a model that reaches a **mean F1 score of at least 0.6** until the deadline mentioned in StudOn.

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