L Mill N Ravikumar
T Würfl K Breininger
Y Xu F Denzinger
January 20, 2019

TensorFlow for Classification

We will implement different architectures of convolutional neural networks using the open-source library TensorFlow. With TensorFlow, one can build data flow graphs for numerical computations. Graphs consist of nodes which represent mathematical operations, e.g. convolutions, and edges, which represent the data arrays (tensors). The core of TensorFlow is implemented in C++ and has several APIs for developers. The Python API, which we will use, is the most complete at the moment.

For the architectures, we will start with the classical convolutional architecture "AlexNet". Then, we will implement "ResNet" as a modern convolutional architecture.

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 enlisted in an online leaderboard.

1 Dataset

Solar modules are composed of many solar cells. The solar cells are subject 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 figure 1):

1. Different kinds of 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 are mainly caused by cracks, where a part of the cell to becomes disconnected. This disconnected area 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 on its own (see figure 1, right example).

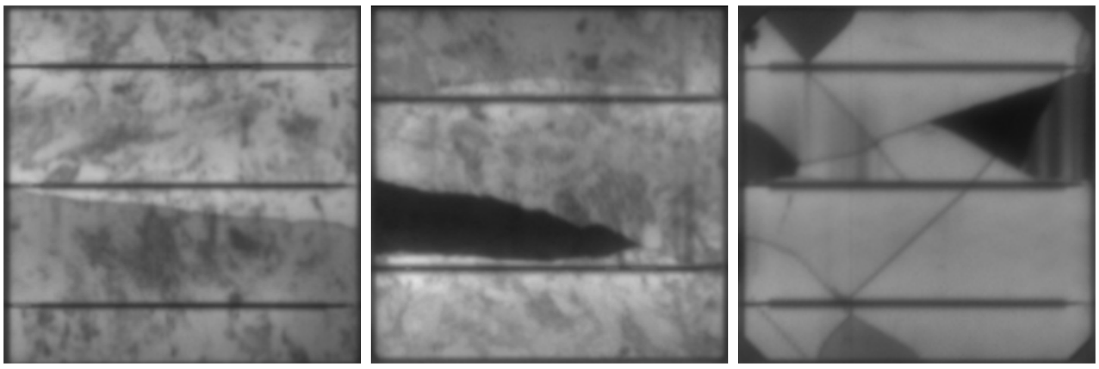
Figure 1: Left: Crack on a polycristalline module; Middle: Inactive region; Right: Cracks and inactive regions on a monocristalline module

2 Preparation

We provide a skeleton that we will use during the course of this exercise. This skeleton contains a *Pipfile* that lists dependent software packages. When working **at home**, you can use PIPENV[1], which combines the PIP package manager with VIRTUALENV to setup an environment that has all required dependencies installed.

In the **CIP pool**, the required packages are already installed and we do not recommend PIPENV here, since it is memory demanding and home directories are not allowed to grow beyond 2GB. However, you need to enable Tensorflow by issuing

```
addpackage tensorflow
```

Task (optional)

Install pipenv on your machine. Then, open a shell in the skeleton folder and issue the following command to install the required packages:

```
pipenv --three install
```

The installation might take some time. Finally, type

```
pipenv shell
```

to open a virtual shell that has the dependencies loaded. From here, you can use Python as you are used to.

[1]https://pipenv.readthedocs.io/

3 Data and evaluation

When you start working on a new machine learning problem, you have to deal with the format, the data is given in and implement a pipeline to load and preprocess the data. The same holds for evaluation routines that you need to assess the performance of your model. Since both tasks often require writing a lot of boilerplate code, we provide a dataset abstraction (in "data/dataset.py") and an implementation for the evaluation (in "evaluation/*") with the skeleton. For the data part, this includes loading, normalization, augmentation as well as a sampling strategy that makes sure that every class is sampled with the same probability. For the evaluation, this includes common evaluation metrics for classification problems. Please make yourself familar with the given implementation.

4 Architectures

In this exercise, you are supposed to implement a *ResNet18* and *Alexnet*. The details of both architectures are specified in Tab. 1 and Tab. 2. We implement a slightly modified version of *Alexnet*. In the original formulation, local response normalization hass been used to stabilize training. We resort to the common batchnorm instead. The main component of *ResNet* are blocks that are augmented by skip connections. We name those blocks $ResBlock(channels, stride)$. For the 18-layer variant of *ResNet*, each *ResBlock* consists of a sequence of $Conv2D$, $BatchNorm$, *ReLU* that is repeated twice. The number of output channels for $Conv2D$ is given by the argument $channels$. The stride of the first $Conv2D$ is given by $stride$. For the second convolution, no stride is used. 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 *Alexnet* and *ResNet18*-architectures in the files "model/alexnet.py" and "model/resnet.py" respectively according to the specification in Tab. **??**.

5 Training

The training process consists of alternately training for one epoch on the training dataset and then assessing the performance on the validation step. After that, a decision is made, if the training process is continued. We will automatically decide, if the training process is continued or cancelled. For that, we need a stopping criterion. A common approach is to stop training, if

Model	Layers	Notes
AlexNet		
	$Conv2D(96, 11, 4)$	
	$BatchNorm()$	
	$ReLU()$	
	$MaxPool(3, 2)$	
	$Conv2D(192, 5, 1)$	
	$BatchNorm()$	
	$ReLU()$	
	$MaxPool(3, 2)$	
	$Conv2D(384, 3, 1)$	
	$BatchNorm()$	
	$ReLU()$	
	$Conv2D(256, 3, 1)$	
	$BatchNorm()$	
	$ReLU()$	
	$Conv2D(256, 3, 1)$	
	$BatchNorm()$	
	$ReLU()$	
	$MaxPool(3, 2)$	
	$DropOut()$	
	$FC(4096)$	
	$ReLU()$	
	$DropOut()$	
	$FC(4096)$	
	$ReLU()$	
	$FC(2)$	

Table 1: Architectural details for the AlexNet. Convolutional layers are denoted by $Conv2D(channels, filter\_size, stride)$. Max pooling is denoted $MaxPool(pool\_size, stride)$.

Model	Layers	Notes
ResNet18		
	$Conv2D(64, 7, 2)$	
	$BatchNorm()$	
	$ReLU()$	
	$MaxPool(3, 2)$	
	$ResBlock(64, 1)$	
	$ResBlock(128, 2)$	
	$ResBlock(256, 2)$	
	$ResBlock(512, 2)$	
	$GlobalAvgPool()$	
	$FC(2)$	

Table 2: Architectural details for the models to implement. Convolutional layers are denoted by $Conv2D(channels, filter\_size, stride)$. Max pooling is denoted $MaxPool(pool\_size, stride)$. $ResBlock(filter\_size, stride)$ denotes one block within a residual network.

the validation loss did not decrease for a specified number of epochs. We will use that criterion in our implementation.

Task

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

6 Put it together

In order to train the model, we need to decide for a loss function. A loss function that is often used in multi-class problems is the *SoftMax*-loss. However, this assumes that the output shall be interpretable as a probability density function and hence sums up to one. This does not hold for multi-label problems, where classes are not mutally exclusive.

Task

Make yourself familar 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.

L Mill N Ravikumar
T Würfl K Breininger
Y Xu F Denzinger
January 20, 2019

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. To pass this exercise, you are supposed to reach a mean F1 score over both classes of 0.6 is required.

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 *saved model*-API of tensorflow. In addition, a zip-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 of the members of the group should create the team. All others should 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 February 15. Afterwards, we will have an open challenge.