Convolutional neural networks

Deep learning for media technology (TNM112), lab 2

Abstract

The second lab is focused on getting familiar with the convolutional neural network (CNN) and strategies for improving generalization performance. We will both look at implementing convolutional layers in Python, as well as performing regularization and improving model design to increase the performance in image classification problems. We will consider a simple task of classification of handwritten numbers, but also test a more challenging digital pathology dataset.

1 Introduction

The convolutional neural network (CNN) is an effective design for processing structured data with spatial correlations. It has been the de facto standard within deep learning for imaging for more than a decade, only recently being challenged by visual transformer models. However, CNNs are still the best choice in scenarios with limited data and computational resources.

In this lab we will first look at implementing the building blocks of a CNN (Task 1) – the *convolutional*, *pooling*, and *flattening* layers. In doing so, you will gain a thorough understanding for how CNNs are constructed. Second, we will look at how to use regularization for improving generalization performance in a scenario with limited training data (Task 2), where overfitting is a prominent issue. Finally, we will explore strategies for improving the performance in a more difficult scenario on a medical imaging dataset (Task 3).

For more information on how to report your results, please refer to the LaTeX lab template. It provides information on how to structure the lab report, as well as how results should be reported (both in lab report and provided code).

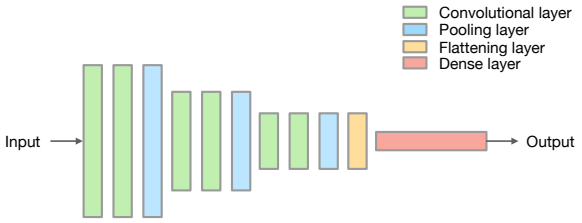

Figure 1: Layers of a convolutional neural network.

2 Method

A CNN for image classification is typically built by composing blocks of convolutional layers followed by pooling layers, as depicted in Figure 1. These layers extract image features that are relevant for solving the task formulated by the objective function. At the end of the model, we use dense, or fully-connected, layers to perform the classification. In order to do this, we need to vectorize the feature maps extracted by the convolutional layers, by means of a flattening layer.

2.1 Convolutional layers

A convolutional layer can be illustrated as in Figure 2, mapping from C_I output channel activations $H_i^{(l-1)}$ in layer $l-1$ to C_O channel activations $H_j^{(l)}$ in layer l. The feed-forward of information between the layers can be described as

$$H_j^{(l)} = \sigma\left(\sum_{i=1}^{C_I} W_{i,j}^{(l)} * H_i^{(l-1)} + b_j^{(l)}\right), \qquad (1)$$

where $H^{(l)}$ is a $[D_y \times D_x \times C_O]$ tensor with the channel activations of layer l, i.e. C_O images of height D_y and width D_x containing the features extracted at this layer. $W^{(l)}$ is a $[R \times S \times C_I \times C_O]$ tensor of learnable weights, arranged as $C_I \times C_O$ filter kernels of size $R \times S$. $H^{(l-1)}$ is a $[D_y \times D_x \times C_I]$ tensor with the channel activations of layer $l-1$, and $b^{(l)}$ is a

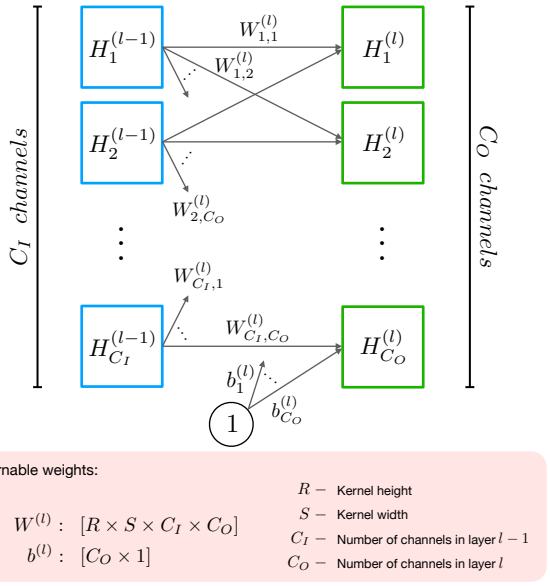

Figure 2: One layer in the CNN, mapping from C_I channels in layer $l-1$ to C_O channels in layer l.

$[C_O \times 1]$ vector with the biases of layer l. Note that we assume the same size of the channels in layer $l-1$ and l. This does not need to be the case if we do not perform zero padding before convolving, or if we use, e.g., strided convolutions. However, for this lab we will for simplicity assume that a convolutional layer does not change the resolution of the channels/feature maps. We only modify the resolution by performing max pooling operations.

2.2 Pooling layers

To reduce the dimensionality of the images, or feature maps, we apply pooling layers. This both reduces the number of elements that will be passed to the dense layers, and increases the receptive field of the network. The most common pooling operation is max pooling in neighborhoods of 2×2 pixels, reducing the resolution of feature maps by a factor of 2, as illustrated in Figure 3. The operation is applied to each of the channels/feature maps of a layer.

2.3 Flattening layer

The flattening layer simply reshapes the channel activations $H^{(l)}$ from a tensor with dimensions

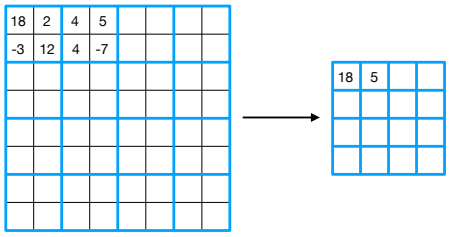

Figure 3: Max pooling operation.

$[D_y \times D_x \times C]$ to a vector of size $[D_y D_x C \times 1]$.

2.4 Dense layer

The dense, or fully-connected, layer is the same formulation we used in Lab 1 for the MLP,

$$h^{(l)} = \sigma\left(W^{(l)} h^{(l-1)} + b^{(l)}\right), \tag{2}$$

where $W^{(l)}$ now is a weight matrix, mapping activation vectors from the previous layer. If we use this after a flattening layer, the input activation vector will be the flattened/vectorized channel activations from the last convolutional layer, potentially reduced in size after max pooling.

3 Datasets

We will use low-resolutional image datasets, starting with the simple MNIST dataset. This contains 60K grayscale training images in 28×28 pixels resolution, where a few examples are shown in Figure 4. These can be loaded through the generator in `data_generator.py` by providing `dataset='mnist'` to the generator function.

In Task 3, we will look at tumor classification in a digital pathology dataset. The dataset is provided with the lab material and is a cropped and subsampled (100K images in 32×32 pixels resolution) version of the PatchCamelyon dataset[1] [1], where a few training samples are shown in Figure 5. PatchCamelyon, in turn, has been derived from the Camelyon dataset, which contains the original scanned slides[2]. The PatchCamelyon

[1] https://github.com/basveeling/pcam

[2] For more information, you can have a look at the Camelyon webpage: https://camelyon16.grand-challenge.org/

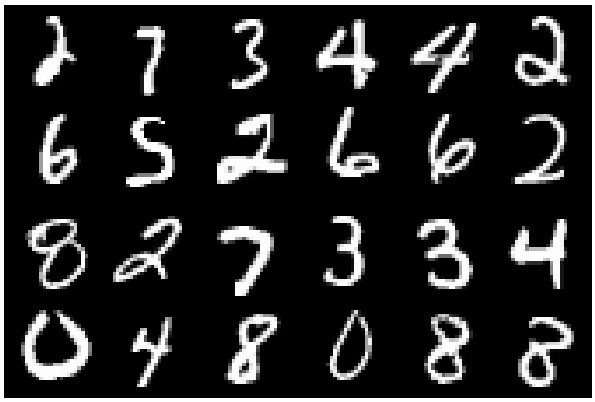

Figure 4: Examples of MNIST training images.

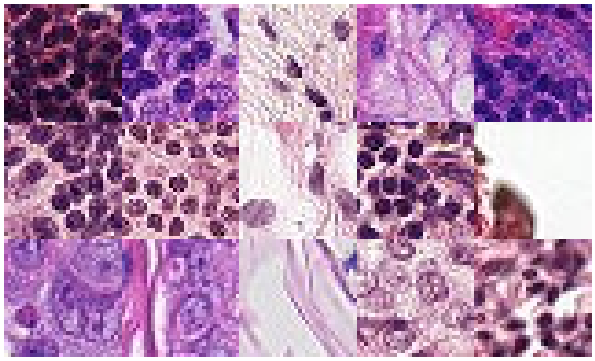

Figure 5: Examples of PatchCamelyon training images.

images have been sampled from the original digitized whole slide images of stained tissue samples of breast lymph node sections, and they are labeled as either healthy or containing tumor tissue. Without background in medicine, it can be difficult to distinguish between the classes, but we will let the CNN do this for us. However, this is a much more challenging task compared to classification of the MNIST dataset. The dataset can be loaded through the same data loader by providing `dataset='patchcam'` to the generator function.

4 Assignments

You will test the results of the different tasks using the provided Jupyter notebook, `lab02.ipynb`. In addition to this, there are three python scripts, `data_generator.py`, `util.py`, and `cnn.py`. For this lab, you will not need to do changes in the `data_generator.py` and `util.py`. You will work in the Jupyter notebook, and do some implementations in `cnn.py`.

Libraries

For the different assignments, you might find it useful to install `scipy` and `scikit-image`, which provides functions for implementing, e.g., the convolution and pooling operations. Also, for Task 3 you should install `pandas`, which is used to export your predictions to CSV. All these libraries are readily available through pip:

```
pip3 install scipy scikit-image pandas
```

Task 1 – Implementing a CNN

Implement the following functions in `cnn.py`: `activation`, `conv2d_layer`, `pool2d_layer`, `flatten_layer`, `dense_layer`, and `evaluate`. Additionally, compute the number of weights in the model, in the function `setup_model`.

For the `activation` and `dense_layer` functions you can use code that you wrote in Lab 1. For the `evaluate` function you can use the code from Lab 1 for the accuracy. However, now we have trained with the cross-entropy loss, so you should evaluate this loss function through our CNN.

When you have finished the implementation, you should be able to run the code provided in the Jupyter notbook and get the same result as with the Keras model. However, our code is not optimized, nor parallelized, so it will be rather slow to process the full dataset. In order to facilitate comparisons when you implement the functions, you can use the cell in the Jupyter notebook called 'Evaluation of our CNN layers'. This sets up a Keras model with randomly initialized weights, where you can select which type of layer it should use. By feeding a single image through the Keras model with random weights, we can compare the layer output with our CNN. To run a single image through our model, we can use the `feedforward_sample` function. Evaluate the different layers you implement, to verify that you get the same output as with the Keras model. When you are finished with the implementation, you can run the evaluation in the cell called 'Our CNN' to

verify that you get the same loss and accuracy as with the Keras model.

The instructions in the code in `cnn.py` should be sufficient to understand what you are supposed to add in terms of code. However, here are some pointers to facilitate the task:

- Make sure you understand the operations in Equation 1, summing over convolved images.
- Have a look at the `get_weights` function in `util.py` to understand how weights and biases are arranged in lists when these are extracted from the Keras model.
- Even if we describe the network as convolutional, it is using cross-correlation. This means that we need to flip our kernel vertically and horizontally before performing the convolution.
- The weights for a convolutional layer extracted from Keras use the same indexation as in Figure 2, i.e. with the first two dimensions of the tensor being the kernel, and where 3rd and 4th dimensions are indexing to the previous and current layer channel, respectively.
- For the convolutional operation, you can e.g. use `convolve2d` through `scipy`:

```
from scipy import signal
z = signal.convolve2d(...)
```

- Make sure to use the same border mode as in Keras, padding in zeros before the convolution so that the resolution is not changed.
- For the pooling operation, you can e.g. use `measure.block_reduce()` through `scikit-image`:

```
import skimage
z = skimage.measure.block_reduce(...)
```

In the lab report, you should discuss around how a CNN is implemented. Go through all the steps you have implemented and explain how you have solved the problems in code. You should also provide the relevant code as supplementary material when submitting your lab report.

Task 2 – Regularization of a CNN

In this experiment, we select a random subset from MNIST, with only 128 images. This is a very minimal dataset, but it is interesting to see how overfitting can be prevented with regularization strategies.

Your task is to expand the given network with different regularization strategies. You are free to choose which combination of strategies you want to use, for example:

- Augmentation layers (you can look at different Keras options[3], and apply these directly after the input layer).
- Dropout (checkout `layers.Dropout()`, and apply, e.g., to your dense layers).
- Weight decay (you can, e.g., use the `kernel_regularizer` option in a layer specification[4]).
- Batch normalization (unclear if it will help in this task, but you can checkout `layers.BatchNormalization()`).

You can also expand the network with more layers, strided convolutions, or add skip-connections to facilitate optimization. Moreover, you can experiment with the number of training epochs and the batch size.

During your development, you test the performance on the validation set (`util.evaluate` with the flag `final=False`). When you have found a good setup, run evaluation on the test set (`final=True`). Do this for at least **5 runs** and report the **average** and the **variance** across models. Since the subset of 128 images is randomly selected and the optimization is stochastic, you will get slightly different values each time. Thus, averaging is important to get a robust indication of your model's performance. You can do this manually, or you can setup a for-loop to run a sequence of trainings, where you log the evaluation results for each model (some pointers for doing this is provided in the code).

For the final results, you should aim at having an average accuracy of at least 90% (which is quite

[3] https://keras.io/api/layers/preprocessing_layers/image_augmentation/

[4] https://keras.io/api/layers/regularizers/

good considering that we are only training on 128 images).

Task 3 – Tumor classification

In this task, we will look at a more difficult task, using the PatchCamelyon dataset. This contains tissue samples from breast lymph nodes (see Section 3), which could either be healthy or contain tumor tissue, i.e. this is a binary classification problem. For this reason, it is valuable to also evaluate the AU-ROC, which can be passed to Keras as an evaluation metric, 'AUC'.

You are provided with the training and validation set. These have been cropped to 32×32 pixels, to make it a bit less resource demanding to train. The test set is also provided, but without labels.

Your task is to setup a Keras model to achieve the best possible performance on the validation set. You are free to choose how to do this, e.g., by means of regularization and network specifications. You can, for example, use your results from Task 2 as a starting point, but you should also think around how this problem differs from the MNIST classification. For example, in this case the orientation of images has no meaning, so you can randomly flip images both horizontally and vertically. You can also, e.g., explore augmentation by means of changing the image contrast and brightness.

When you are finished with your development, you should run the `util.pred_test()`, to produce a CSV file with predictions of the test set. The exported CSV file should be uploaded to the Lab 2 Kaggle challenge[5]. You need to train the model and export a CSV file 5 times, in order to have a good estimate for your model's performance. There are no requirements on how well your model should perform, as long as it is well above random guessing (50% accuracy).

References

[1] Bastiaan S Veeling, Jasper Linmans, Jim Winkens, Taco Cohen, and Max Welling. Rotation equivariant CNNs for digital pathology. *arXiv preprint arXiv:1806.03962*, 2018.

[5] Challenge invitation: https://www.kaggle.com/t/7767161492544993833532e95127761f