Deep Learning

Practical Exercises - Week 12

Convolutional Neural Network

ICAI

FS 2024

1 Topic

In the last three lab sessions we have implemented fully connected neural networks and tested various layers, such as dropout and batch normalization. With fully connected neural networks, however, the fact that the input vector is actually an image is neglected. All information contained in the spatial arrangement of the pixels is discarded. To further improve performance, the spatial arrangement of the pixels (grid-like topology) should also be taken into account. This is where convolutional neural networks (CNNs) come into play.

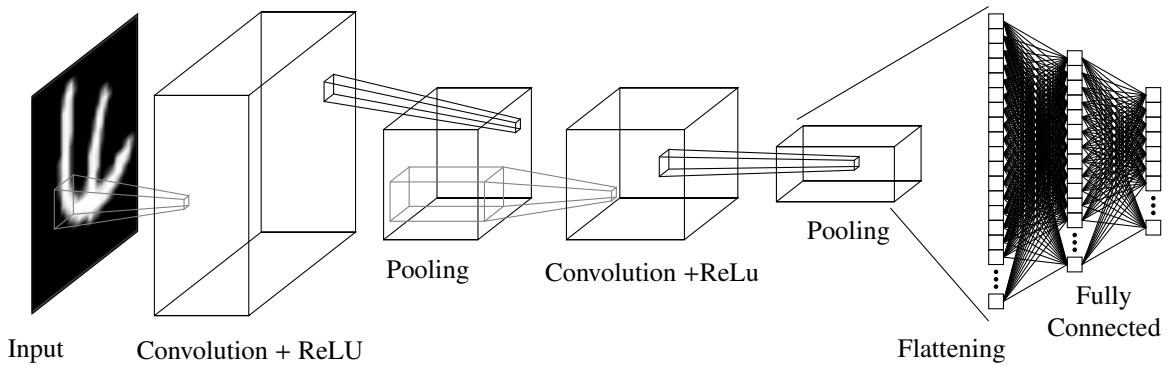

Figure 1: Architecture of the convolutional neural network used to classify handwritten digits of the MNIST data set.

2 Convolutional Neural Network

In this lab session we will build a convolutional neural network that classifies the handwritten digits of the MNIST test dataset with an accuracy of 99.5%. The model used for this is shown in Figure 1.

Input:

The input of the network consists of 28 by 28 pixel grayscale images of handwritten digits. In order for the convolutional layers to be able to process the images, they must have a shape of `[batch_size, image_height, image_width, channels]`, where for grayscale images, `channels` (the number of color channels) is set to 1. Hence, for the MNIST data set the desired shape for the input layer is `[batch_size, 28, 28, 1]`.
Reshaping and preprocessing the input data is generally part of the input pipeline and not of the model itself.

First Convolutional Layer:

In the first convolutional stage, 32 filters of size 5×5 (extracting 5×5-pixel subregions) are applied to the input image `x` by calling the following object:

```
def __init__(self):
    self.conv_1 = Conv2D(filters=32, kernel_size=5, padding='same')
def call(self, x, training=False):
    t_conv_1 = self.conv_1(x)
```

The zero-padding option is set to `'SAME'`, which specifies that the output tensor should have the same height and width as the input tensor. For more information on the `Conv2D` layer go to the corresponding TensorFlow description.

After the convolution layer a ReLU activation function is applied as follows

```
def __init__(self):
    self.relu_1 = ReLU()
def call(self, x, training=False):
    t_relu_1 = self.relu_1(t_conv_1)
```

Finally, the MaxPool2D is added to the first convolutional stage, which performs max-pooling in a non-overlapping neighborhood of 2×2 as follows

```
def __init__(self):
    self.pool_1 = MaxPool2D(pool_size=(2, 2))
def call(self, x, training=False):
    t_pool_1 = self.pool_1(t_relu_1)
```

The `pool_size` argument specifies the size of the max-pooling neighborhood by specifying the size of the window for each dimension of the input tensor.
This pooling size results in a downsampling of the feature map by a factor of 2 in both spatial directions. The output tensor `t_pool_1` produced by `self.pool_1()`, thus, has a shape of `[batch_size, 14, 14, 32]`.

Second Convolutional Layer:

In the second convolutional stage, 64 filters of size 5×5 are applied and the ReLU activation function is used as non-linearity. Subsequently, max-pooling is applied

in a non-overlapping neighborhood of 2×2. The resulting output tensor `t_pool_2` of the second convolutional stage has a shape of `[batch_size, 7, 7, 64]`.

Fully Connected Layers:
To perform classification on the features extracted by the convolutional layers, two fully-connected layers are added to the network. However, before the fully-connected layers can be connected, the feature map `t_pool_2` has to be flattened to a shape of `[batch_size, 7·7·64]`, so that the input to the fully-connected layers becomes one-dimensional (or two-dimensional for a mini-batch of examples):

```
def __init__(self):
    self.flat_1 = Flatten()
def call(self, x, training=False):
    t_flat_1 = self.flat_1(t_pool_2)
```

Now the fully-connected layer can be added: A hidden layer with 1024 neurons and the ReLU activation function and an output layer with 10 neurons (one for each digit target class 0–9) and the softmax activation function. Additionally, dropout is applied before both fully-connected layers.

Most parts of the training process are already implemented in the following Python modules:

main.py
Loads the datasets and starts the training.

train.py
Implements the training, validation and testing procedures and logs some metrics which can be displayed with TensorBoard.

data.py
Loads and preprocesses the MNIST images and prepares them as TF Dataset.

model.py
Implements the convolutional neural network.

Your task is to implement the convolutional neural network in `model.py` according to the description above. You are free to change the proposed architecture and test other layers. Use the following TensorFlow tutorials as a source of inspiration:

- `https://www.tensorflow.org/guide/keras/sequential_model`
- `https://www.tensorflow.org/guide/keras/functional`
- `https://www.tensorflow.org/guide/keras/custom_layers_and_models`

If you correctly implement the CNN, you should be able to achieve an accuracy of approximately 99.5% on the test dataset.
Hint: Keep in mind that the dropout behaves differently during training than at test time.