Advanced Computer Vision

Exercise Sheet 5

Winter Term 2023
Prof. Dr. Niels Landwehr
Dr. Ujjwal

Available: 05.12.2023
Hand in until: 12.12.2023
Exercise session: 15.12.2023

Task 1 – Number of Multiply-Adds and Parameters in Convolutional Neural Network Architectures [15 points]

Assume a convolutional neural network with the following structure:

Input: 47×47 pixel RGB image
Convolution layer: 5×5 kernel, 32 filters, stride 1
Convolution layer: 3×3 kernel, 32 filters, stride 2
Convolution layer: 3×3 kernel, 32 filters, stride 1
Convolution layer: 3×3 kernel, 32 filters, stride 2
Convolution layer: 3×3 kernel, 32 filters, stride 1
Flattening layer
Fully connected layer with 128 output nodes
Fully connected layer with 10 output nodes

The convolution layers do not employ any padding. Compute the output size, the number of parameters, and the number of multiply-add operations at each layer in the network. Can you give a general formula for the output size of a convolution layer as a function of the input size, kernel size, and stride? What is the overall number of parameters and the overall number of multiply-add operations for this network?

Task 2 – Batch Normalization in Python [20 points]

In this exercise, we implement and experiment with the batch normalization layer. The notebook *batchnorm.ipynb* defines a convolutional neural network for the Fashion-MNIST data set. The network contains $L = 3$ blocks of K convolution layers each. For small K, e.g. $K = 3$, the network quickly converges, however, for e.g. $K = 20$ training basically stalls.
Define a custom layer for the network that implements batch normalization, with trainable parameters α and β as discussed in the lecture, without using the Tensorflow/Keras implementation of batch normalization. Add this custom layer before each convolution layer as indicated in the notebook. Verify that after adding your custom batch normalization layer, the network converges to a reasonable training loss and accuracy also for $K = 20$. Also implement a version of batch normalization in which the mean and variance for normalization are computed over all channels jointly, rather than separately for each channel.

Hints (assuming a Tensorflow implementation, you can of course also use PyTorch):

- See `https://keras.io/guides/making_new_layers_and_models_via_subclassing/` for how to define custom layers with trainable weights in Keras.
- Use the functions *tf.math.reduce_mean* and *tf.math.reduce_std*.

- To simplify your code, make use of the broadcasting rules in Tensorflow, which automatically tile tensors to match another tensor in a pointwise operation. For example, you can subtract a tensor of size $(n, k, 1)$ from a tensor of size (n, k, m), in which case the first tensor is tiled (duplicated) m times in the third dimension such that it matches the second tensor.

Task 3 – Residual Connections in Python [15 points]

In this exercise, we extend the convolutional neural network defined in *batchnorm.ipynb* to a residual architecture.

Take the original architecture defined in the notebook, and add residual layers to it as shown in the lecture. Specifically, starting from the input layer, add shortcut connections that each bridge M layers, where M is a hyperparameter in the model definition.

Remember that when adding a shortcut connection and the output of the main branch,

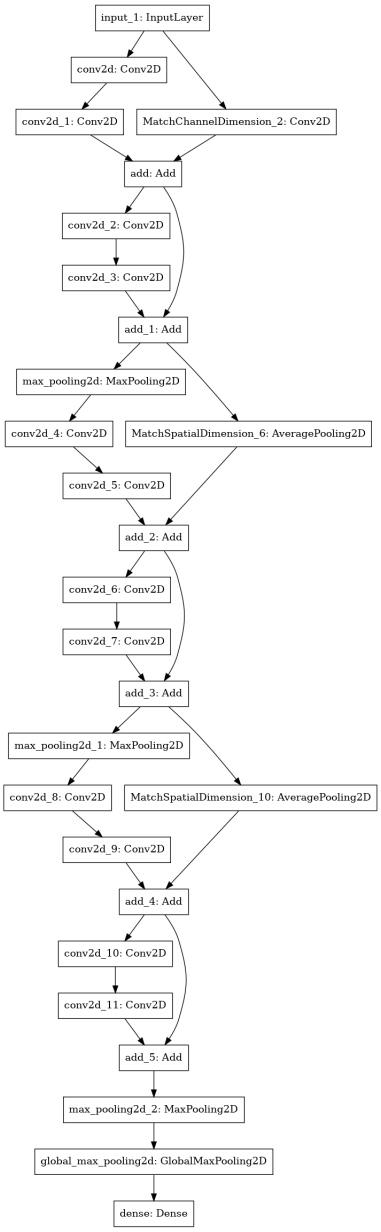

Figure 1: Example of model with shortcut connections

you may need to perform spatial subsampling (you can use an average pooling layer) to

match the spatial resolution of the layers, and/or a 1×1-convolution to match the number of channels. The figure 1 shows an example graph of a model with shortcut connections, for $L = 3$, $K = 4$, and $M = 2$. Consider this is as an example of an implementation. In your own implementation, the exact node names and types might change a bit. Confirm that with shortcut connections, the model achieves reasonable training loss and accuracy, even for very deep architectures and even without batch normalization layers.