Exercise Sheet 4: Convolutional Neural Networks

Due on 06.06.2025, 10:00

Ulrich Prestel , Tao Hu (tao.hu@lmu.edu)

Task 1: Convolutions Revisited (8P)

In this task, you will compute the output of a 2D-convolution operation with 3 different filters of kernel size 2×2 defined by the following weight matrices and biases:

$$w_1 = \begin{bmatrix} 1. & 0. \\ 0. & 1. \end{bmatrix} \qquad w_2 = \begin{bmatrix} 1. & 0. \\ 1. & 0. \end{bmatrix} \qquad w_3 = \begin{bmatrix} 1. & 1. \\ 0. & 0. \end{bmatrix}$$

$$b_1 = [-1] \qquad b_2 = [-1] \qquad b_3 = [-1]$$

Furthermore, let the following matrix be your input "image":

$$x = \begin{bmatrix} 1. & 0. & 0. & 1. & 1. & 0. \\ 0. & 1. & 0. & 0. & 0. & 1. \\ 0. & 0. & 1. & 1. & 0. & 0. \\ 1. & 0. & 0. & 0. & 1. & 0. \\ 1. & 0. & 0. & 0. & 0. & 1. \end{bmatrix}$$

1. Using $s = 1$ (stride) and $p = 0$ (padding), calculate by hand the output of the convolution operation followed by the ReLU activation function. (2P)
2. Think about and describe how the input image patch, the convolution kernel, and the ReLU activation function interact, and how you can avoid manual computation and directly infer the output for each patch. (1P)
3. Now repeat (a) with $s = 2$. Please only report the final outputs, not the intermediate ones before the activation. (1P)
4. What might be the purpose of using $s = 2$? What other operation can be used instead for the same purpose? **Hint**: Think about compression. (2P)
5. Please derive the general formula to compute the output size of a convolution operation (size of the feature map), depending on the kernel size, the stride, and padding. It is sufficient to derive it for the 1D case. (2P)

Task 2: Receptive Field (6P)

The receptive field of a neuron in a Convolutional Neural Network (CNN) is the size of the image region that is connected to the neuron.

1. Suppose that we have a CNN with 3 convolutional layers with a kernel size of 3×3 and $s = 1$. Calculate the receptive field of a neuron after the last convolutional layer. (2P)
2. Suppose that we have a CNN with 2 convolutional layers with a kernel size of 4×4 and $s = 2$. Calculate the receptive field of a neuron after the last convolutional layer. (2P)
3. Based on your findings, how is the depth related to the receptive field of a CNN? When keeping the depth fixed, can you infer any strategies to still increase the receptive field of a CNN? (2P)

Task 3: Convolutional Neural Network (14P)

Now, we want to move on to a more complex dataset and use a Convolutional Neural Network for classification.

1. *Dataset.* We will train our classifier on the canonical CIFAR10[1] dataset. Please use `torchvision.datasets.CIFAR10` to load the dataset and familiarize yourself with it. By setting the flag `download=True`, the dataset can be automatically downloaded. Plot 10 example images per class and make sure to also include the class label in your plot. (2P)
2. *Network.* Implement the following network as a PyTorch `torch.nn.Module`. In particular, implement its `__init__` and `forward` function. Finally, please print the number of trainable parameters. (4P)
   - Convolutional Layers: 6 filters of size 5×5 with padding 0 and stride 1 + ReLU layer.
   - Max-Pooling Layer: Size 2×2 and stride 2.
   - Convolutional Layer: 16 filters of size 5×5 with padding 0 and stride 1 + ReLU layer.
   - Max-Pooling Layer: Size 2×2 and stride 2.
   - Fully connected layer: 120 neurons + ReLU layer.
   - Fully connected layer: 84 neurons + ReLU layer.
   - Fully connected layer: 10 neurons.
3. *Training.* Train the above-defined CNN classifier using Cross-Entropy Loss on the training set of the CIFAR10 dataset (image size 32^2). (4P)

[1] https://www.cs.toronto.edu/~kriz/cifar.html

- Don't forget to normalize your data to range $[0, 1]$ and activate data shuffling while training.
- Train your model with a batch size of 16 and Adam optimizer with a learning rate of 0.001 for 10 epochs.
- During training set your model to train mode (i.e. `model.train()`) and for evaluation set your model to eval mode (i.e. `model.eval()`).
- Plot how the accuracy (i.e. percentage of correctly classified images) of your model evolves for both, the training- and test-set of CIFAR10. Please make sure to add meaningful axis labels for your plot (*y-axis* as the accuracy and *x-axis* as the epoch number).
- What do you observe in the accuracy plot, in particular if you compare the train and test accuracy? Do you have an explanation for that?

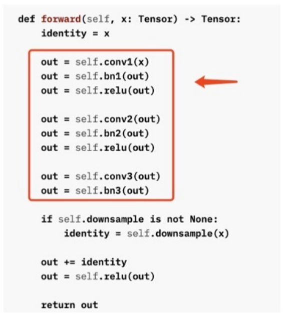
4. *Augmentation.* Now we will try to improve our model by using data augmentation strategies. In PyTorch, augmentations can be easily realized with the `torchvision.transforms` library. (4P)
   - Please apply random horizontal flipping and random cropping with `padding = 4` to the CIFAR10 images and visualize a few pairs of augmented images against their original. Feel free to also try out other augmentations, e.g. ColorJitter.
   - Train your model using the same setup as above, however, with **additional normalization** (normalize your input images channel-wise using $\mu = 0.5$ and $\sigma = 0.5$) and **data augmentation**.
   - Plot the test accuracy of this model and the one you trained before over time. Make sure to use proper axis labels and a meaningful legend.
   - Based on your results, provide a brief discussion of why the extra normalization and augmentation improves/harms the performance of your model.

Task 4: Activation and Saliency Maps (10P)

In this task, we will try to visualize what CNNs learn by means of the activation map, which is considered to be more informative than just visualizing e.g. the kernel weights.

1. *Instantiate a pre-trained ResNet18.* Apart from implementing your own neural networks, `torchvision.models` provides a bunch of often-used network architectures with pre-trained weights (usually pre-trained on an ImageNet classification task). Please instantiate a ResNet-18 and print out its architecture. (1P)

2. *Intermediate activation maps.* When you define your `model.forward()` function, you will have intermediate features after different layers, like the variable `out` in the code snippet below. Please extract intermediate activations for the `pug.jpg` image provided for at least 2 different layers (e.g., the `out` after `conv1` and the `out` after `conv3` below), and print out their shapes. (3P)



```
def forward(self, x: Tensor) -> Tensor:
    identity = x

    out = self.conv1(x)
    out = self.bn1(out)
    out = self.relu(out)

    out = self.conv2(out)
    out = self.bn2(out)
    out = self.relu(out)

    out = self.conv3(out)
    out = self.bn3(out)

    if self.downsample is not None:
        identity = self.downsample(x)

    out += identity
    out = self.relu(out)

    return out
```

Hints:

- There are multiple ways of obtaining these features. The most straightforward one would be to directly store them when you define your forward function. However, this is not the case for a pre-trained model. A workaround is to define a new network with the same architecture and weights as the pre-trained one and define your own `forward()` function[2].
- Feel free to use other methods like forward hooks or even some packages. The search engine of your preference will help you with that :)

3. *Visualize the Activation Maps.* Now visualize these activation maps you just got and report/discuss what you see. Most likely you will be facing at least one problem here, as the channel for the activation maps are some different numbers other than 1 (grayscale) or 3 (RGB), which you cannot directly visualize using `plt.imshow()`. So you need some **Dimensionality Reduction technique** to proceed with this task, for instance, PCA, but you can use whatever method you prefer. Even something like the mean value of the first 10 channels will be accepted, as soon as the visualization makes sense and you can explain it. (2P)

4. *Plot both the activation map and the original image.* Show the activation map on top of our original image. Check whether this matches a human's intuition - how would you recognize the given image? (1P)

   Hint: You need to use image re-sampling to match the spatial resolutions of the activation map and the original image.

[2] https://pytorch.org/vision/stable/feature_extraction.html

5. *Saliency Maps.* Another way to visualize what your network has learned is called *Saliency Map* (also known as Pixel Attribution map) — a simple heatmap that highlights pixels of the input image that most caused the output classification. The idea is that we calculate the gradient of the loss function for the class we are interested in with respect to the input pixels. This gives us a map of the size of the input features with negative to positive values. Please use the same pre-trained network from the task above for this task. (3P)

   (a) Complete the saliency map function provided in `ex4.ipynb` under section **4. a)**. (1P)

   (b) Visualize the original image with the saliency maps together and provide a short discussion on why different ground truth labels with even the same input image would yield different saliency maps, for instance, the `catdog_243.png` image with $y = 243$ (bull mastiff) and $y = 285$ (Egyptian cat). (2P)