# Deep Learning Course: Lab 5

Alex Stergiou, Esther Rodrigo, Fawaz Sammani

7th November 2023

Traditionally, Convolutional Neural Networks (CNNs) are used to analyze images and are made up of one or more convolutional layers, followed by one or more linear layers. The convolutional layers use kernels (also called filters). Instead of assigning individual weights per pixel, convolutions apply kernels sequentially over patches within an image. After applying the kernel, the convolved patches are summed over their width and height. See fig. 1.

This output of a convolution layer can also be fed into another convolutional layer or it can be vectorized and fed into a linear layer. Apart from defining the number of kernels, we also need to define the height and width. For instance, taking the 3x3 kernel from the figure: The kernel has a height of 3 pixels and a width of 3 pixels. The size of the kernel determines the number of weights, so a 3x3 kernel would have 9 weights. In traditional image processing, these weights were specified by hand by engineers. However, the main advantage of the convolutional layers in neural networks is

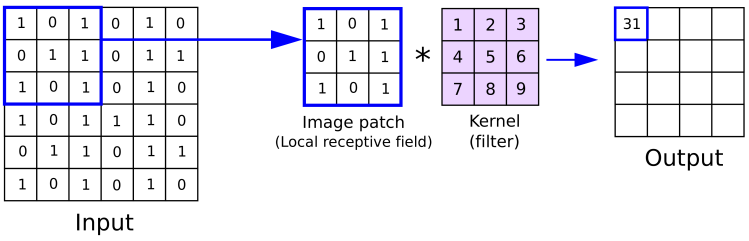


Figure 1: Applying a convolutional kernel

that these weights are learned via backpropagation.

The intuitive idea behind learning the weights is that your convolutional layers act like feature extractors, extracting parts of the image that are most important for your CNN's goal, e.g. if using a CNN to detect faces in an image, the CNN may be looking for features such as the existence of a nose, mouth or a pair of eyes in the image [**?**, **?**].

# 1 CNNs Warmup exercises

In this section, you can find some warmup exercises before actually building your own CNN network. We will start with exploring Max Pooling funtions in PyTorch.

- Start by creating a random tensor with height and width of $5 \times 5$ and 3 channels. Remember, PyTorch assumes the channel dimension to be first.
- What will be the output when using nn.MaxPool2d with a kernel size of 3x3 and a stride of 2?
- The nn.Conv2Dfunction takes the following parameters as input arguments: input channels (e.g. 3 if **the input to the layer** is an RGB image), output channels (i.e. the number of operations that their outputs will be stacked), kernel size (in height $\times$ width format), stride (how much the kernel is to be shifted), padding (to resize the input by appending zeros), and dilation (if the weights are to applied sporadically).
- Create a new random RGB tensor of size $10 \times 10$
- Next, apply a convolutional layer with 5 filters, each of size 3x3, with a stride of 1. What will be the shape?
- The spatial dimension has reduced to $8 \times 8$. Why? Can you fix it so that the spatial dimension does not change? Recall: Formula for **same padding**: $\frac{\mathrm{kernelsize} - 1}{2}$
- What is the shape of the weights for the convolutional layer?

- Now assume you have an input with a batch size of 16, 20 channels, and a spatial dimension of $100 \times 60$. You want your resulting output tensor after Max Pooling to be of shape (16, 20, 25, 25). What parameters of the Max Pooling would you choose?
- Can you achieve the same thing but using the stride of a convolution operator instead?

# 2 CNNs with Pytorch using image data

## 2.1 Data Pre-processing

Now we will explore using data transforms. We will work with the CIFAR10 dataset. It is a dataset of 60,000 32x32 color images in 10 classes, with 6000 images per class. There are 50,000 training images and 10,000 test images. We will use torchvision.transforms as a reference. You will apply data transformations to the training images such that you:

- Crop a random portion of the image and resize it to a size of $32 \times 32$
- Apply random horizontal flipping as a data augmentation strategy. You can check other data augmentations here
- Transform it to a PyTorch Tensor using the function ToTensor.
- Standardize it such that it has a mean of zero and standard deviation of 1. You can use the function Normalize for this. Note that the mean of the CIFAR dataset for each of the RGB Channels is: (0.4914, 0.4822, 0.4465) and the standard deviation of the CIFAR dataset for each of the RGB channels is: (0.2023, 0.1994, 0.2010).
- For the test images, only perform the last two steps.
- Load (and download if you haven't already done so) the training and testing datasets using torchvision.datasets. Specify the transforms that you have already defined in the previous step.
- Define your data loader. Use a batch size of 128 for training and 100 for testing. Make sure you shuffle the data loading process in the training set.

- Visualize some of the images from the dataset (the code is already in the jupyter notebook file)

### 2.2 Building the Convolutional Neural Network (CNN)

We will implement a CNN architecture as described below:

- Use three convolutional layers, each with a filter size of 3x3. Make sure you preserve the spatial dimension after each convolutional operation. Use ReLU as the activation function. After the first two convolutions define a Max Pooling layer of kernel size $2 \times 2$ with a stride of 2.
- The number of channels of each output convolutional layer should gradually increase to 126.
- After the final convolution layer, define an Average Pooling layer that will average over **the entire tensor**. You can use nn.AdaptiveAvgPool2d that only requires you to define the output shape (in this case $1 \times 1$) and automatically finds the correct kernel size, stride, and padding. Averaging the entire tensor is commonly referred to as **global** average pooling.
- The resulting tensor should then be fed into a fully-connected layer with 64 output channels (use ReLU and Dropout with a drop probability of 0.1)
- The resulting tensor should then be fed into a fully-connected layer which will be the output layer for our network. The layer should take as input a 64-channel vector and output a vector of 10 channels ( a class probability distribution over the 10 classes of CIFAR 10).
- You can define your network as either an instance of a custom class that implements nn.Module OR by wrapping all layers with a nn.Sequential.
- After you have constructed your network, run a forward pass with a random tensor to ensure there is no error. Also check that the output shape is (N, 10) where N is your batch size. Note: If you are using a GPU, make sure to move both your model AND input tensor to the GPU device

### 2.3 Optimization-Related Definitions

Now we will define the optimizer, the learning rate scheduler and the loss function.

- Refer to the torch.optim library. Define the optimizer and learning rate. Use SGD with momentum, with an initial learning rate of 0.01
- Now define the loss function. We will use the cross-entropyloss (also see the input shapes that this loss expects to know how to call it later). Note that this loss applied softmax on the network output directly. So there is no need to manually define/apply the softmax to the output.

### 2.4 Training and Testing Funtions

Now we will write the training loop function for one iteration.

- Set the model to training with `.train()`.
- Iterarate through your data loader. Pass the inputs to your model and get the outputs, calculate the loss, average the loss over the batch dimension, backpropogate the loss to calculate gradients, and then update the weights. Remember to zero out accumulated gradients after each update!
- At each iteration over the dataloader, print the averaged loss and accuracy. You should notice some fluctuations in the metrics at each batch.
- Write a test function to also iterate over the test set. Keep track of the accuracy and loss on your whole test set.

### 2.5 Run!

Loop over all the epochs and perform the following:

- Run the training function
- Save the losses and accuracies returned at each epoch
- Run the test function

- Save the model checkpoint. You can use torch.save. Overwrite the previously saved checkpoint only if the accuracy improved from the last epoch.

As a reference for this model, you should get a test accuracy of around 76% after training for the first 50 epochs.

## 3 Next Steps (advanced)

The architecture we implemented is at its very basic and dates back to 1998. Since 2012, there has been a tremendous amount of new CNN architectures, better regularization techniques, and better activations. this repo offers most of the models, and reports their accuracy on CIFAR10. If you are a pro, check out this also!