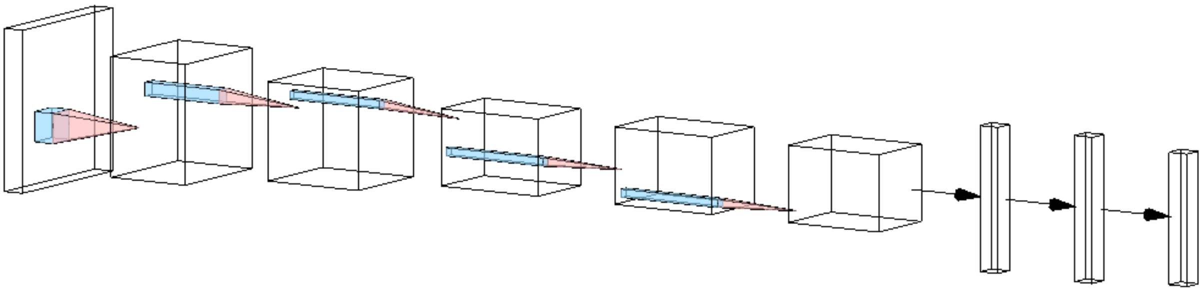
Convolutional Neural Networks

Disclaimer: large parts of the lab are taken from here.

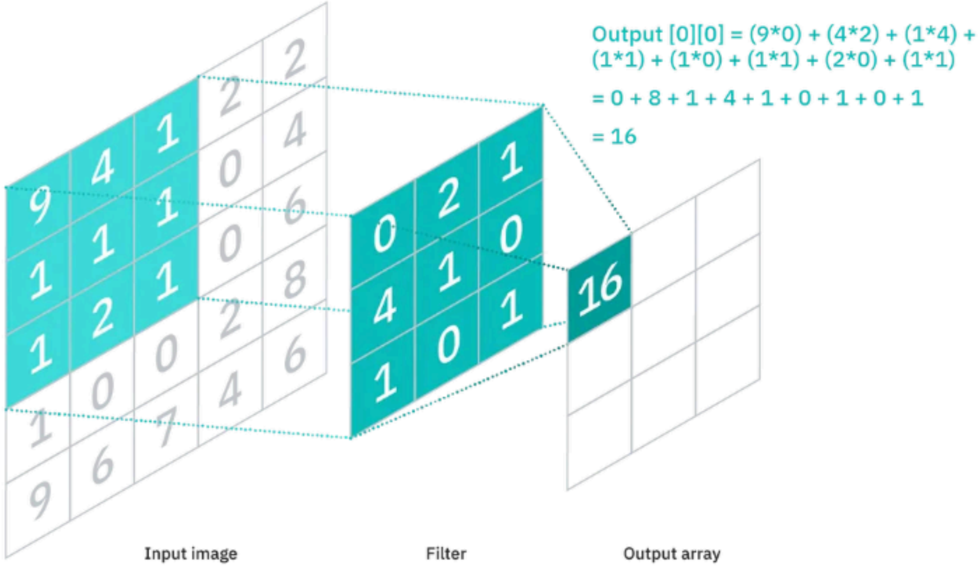
Definition: convolutional neural networks (CNNs) are usually used for image recognition. A network is a convolutional network if it uses *convolution* at least in one layer (the convolutional layer).

IDEA: start from an immagise, and color channel goes the right color of the pixel that you are analyzing. You enlarge the space so that the NET is capable to understed which are the most important & usefull features. The NET has fullfill the space, so that he can analyze better and find the features. At the end, I have to have only a vector - containing classes.



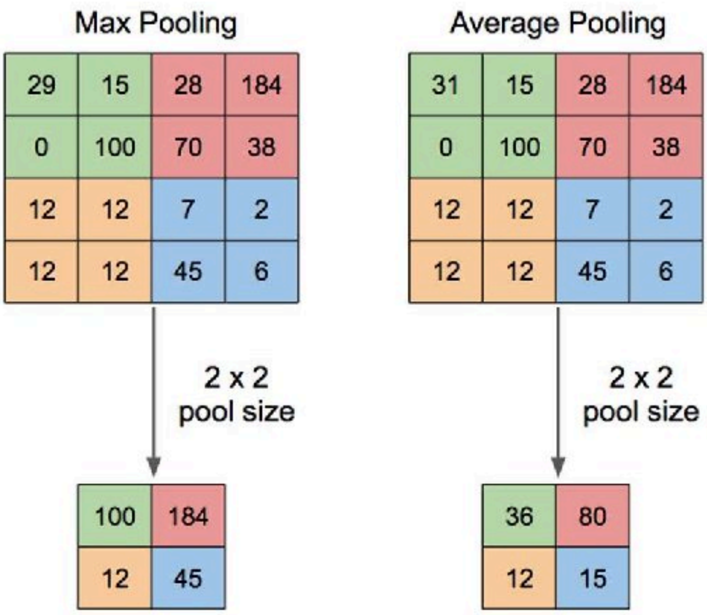
The convolutional layer: the **convolutional layer *extracts features*** (sharpen, blur...) from the pixels. It is a mathematical operation between the input image and the kernel (filter). --> application of the kernel

This is an example of **poling** filter that compress the information that I have. The idea is that I enlarge at the beginning, but after I have to reduce the dimension so that I can effort the computation cost. Here, i.e., we are applying moltiplication (as we can see in the photo).



Credits

The pool layer: pooling layers reduce the dimensionality of the previous layer, mantaining the most important features. Common choices are the max and the average value. --> apply only in the right example: you have to understand wich layer put in your net in relation to what you have to do, what is your purpose.



Credits

Let us try to build our first CNN! As usual, first thing first, we import the libraries and we ask for GPUs.

AIM Works with colored images.

In [1]:
```
# Load in relevant libraries, and alias where appropriate
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms


# Device will determine whether to run the training on GPU or CPU.
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

Let us define some useful parameters!

In [2]:
```
# parameters
batch_size = 64
num_classes = 10
learning_rate = 0.001
num_epochs = 20
```

Let us import the Dataset. The below code:

- defines a transformation over the data (resizes, converts and normalizes),
- downloads the data,
- defines dataloaders which load the data in batches (the RAM does not suffer).

In [3]:
```
# Use transforms.compose method to reformat images for modeling,
# and save to variable all_transforms for later use

# We compose the transformation, we have 3 step in it: Resize, ToTensor and Normalize (3 because one for each color - channel)
all_transforms = transforms.Compose([transforms.Resize((32,32)),
                                     transforms.ToTensor(),
                                     transforms.Normalize(mean=[0.4914, 0.4822, 0.4465],
                                                          std=[0.2023, 0.1994, 0.2010])
                                     ])
# Create Training dataset
train_dataset = torchvision.datasets.CIFAR10(root = './data',
                                             train = True,
                                             transform = all_transforms,
                                             download = True)

# Create Testing dataset
test_dataset = torchvision.datasets.CIFAR10(root = './data',
                                            train = False,
                                            transform = all_transforms,
                                            download=True)

# Instantiate loader objects to facilitate processing
train_loader = torch.utils.data.DataLoader(dataset = train_dataset,
                                           batch_size = batch_size,
                                           shuffle = True)


test_loader = torch.utils.data.DataLoader(dataset = test_dataset,
                                           batch_size = batch_size,
                                           shuffle = True)
```

```
100%|████████████████████████████████████████████████████████████████| 170M/170M [00:17<00:00, 9.88MB/s]
```

We have to create the CNN and, as usual, we extend the `nn.Module` and define the layers in the constructor.

Then, we have to define the `forward` method. The input channel is 3 (3 channels: red, green and blue). We are "enlarging" the third dimension to captures more features (opacity, sharpness...).

Then we pool. You define how many pixel you want to consider (2x2), filter 2-by-2, and the *stride*, i.e. how do you move along the pixels.

In the end, a fully connected layer is defined (nothing new).

IMP - Courious about how to match dimensions of the CNN? Give a look

In [ ]:
```
# Creating a CNN class
class ConvNeuralNet(nn.Module):
        #  Determine what layers and their order in CNN object
    def __init__(self, num_classes):
        super(ConvNeuralNet, self).__init__()

        # Different dimension that are power of 2, depends on the image and channel that you want
        # NB You have to match the dimension of the matrix moltiplication in each step of the convolutional layer

        # Build the layer
        # Convolutional layer, stride omitted == takes all the stuff
        self.conv_layer1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3)
        self.conv_layer2 = nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3) # Convolutional layer

        # Max poll layer: kernel_size is the dimension of the filter, stride represent how many step I'm doing through the kernel
        self.max_pool1 = nn.MaxPool2d(kernel_size = 2, stride = 2)

        self.conv_layer3 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3) # Convolutional layer "out_channels"
                                                                                     # here we enlarge the dimension
        self.conv_layer4 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3) # Convolutional layer

        self.max_pool2 = nn.MaxPool2d(kernel_size = 2, stride = 2) # Max pool

        self.fc1 = nn.Linear(1600, 128) # Linear layer - tricky part to match the dimension

        self.relu1 = nn.ReLU() # Activation function - you have to call it to activate the layer

        self.fc2 = nn.Linear(128, num_classes) # Linear layer

    # Progresses data across layers
    def forward(self, x):
        # Structur of the net - see the different layer
        out = self.conv_layer1(x)
        out = self.conv_layer2(out)

        out = self.max_pool1(out)

        out = self.conv_layer3(out)
        out = self.conv_layer4(out)

        out = self.max_pool2(out)

        out = out.reshape(out.size(0), -1)

        out = self.fc1(out)

        out = self.relu1(out)

        out = self.fc2(out)

        return out
```

Let us define the loss and the optimizer.

In [ ]:
```
seed = 0
torch.manual_seed(seed)

model = ConvNeuralNet(num_classes)

# Set Loss function with criterion
criterion = nn.CrossEntropyLoss() # We select Entropy because we're working with classification problem

# Set optimizer with optimizer
# Stocastic gradient descent
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, weight_decay = 0.005, momentum = 0.9)
# weight_decay := the parameter of the penalication of the weight (you do NOT want the weight increase too mucn)
# --> otherwise, when you apply the loss
# maybe you are just focusing on the value of the weigths and not on the athore part of the minimization function
# momentum := related to the gradient computed in the prevoius step

total_step = len(train_loader)
```

Let us train the model and test it!

In [ ]:
```
# We use the pre-defined number of epochs to determine how many iterations to train the network on
for epoch in range(num_epochs):

        #Load in the data in batches using the train_loader object
```

```
    for i, (images, labels) in enumerate(train_loader):
        # Move tensors to the configured device
        images = images.to(device) # Using cpu or gpu in this data
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad() # Put zero the value at the beginning
        loss.backward()
        optimizer.step() # Optimize

    print('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1, num_epochs, loss.item()))
```

```
Epoch [1/20], Loss: 1.6079
Epoch [2/20], Loss: 1.6913
Epoch [3/20], Loss: 1.7127
Epoch [4/20], Loss: 1.6160
Epoch [5/20], Loss: 1.1036
Epoch [6/20], Loss: 1.1968
Epoch [7/20], Loss: 1.3355
Epoch [8/20], Loss: 0.9578
Epoch [9/20], Loss: 1.4896
Epoch [10/20], Loss: 0.7365
Epoch [11/20], Loss: 0.7428
Epoch [12/20], Loss: 0.8093
Epoch [13/20], Loss: 0.5475
Epoch [14/20], Loss: 0.7854
Epoch [15/20], Loss: 0.6036
Epoch [16/20], Loss: 0.7846
Epoch [17/20], Loss: 0.8605
Epoch [18/20], Loss: 0.7052
Epoch [19/20], Loss: 0.4417
Epoch [20/20], Loss: 0.6482
```

In [ ]:

```
# How we compute the accurcy of the network
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in train_loader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Accuracy of the network on the {} train images: {} %'.format(50000, 100 * correct / total))
```

```
Accuracy of the network on the 50000 train images: 82.378 %
```