50.039 Theory and Practice of Deep Learning

W4-S2 Introduction to Computer Vision and Convolutional Neural Networks

Matthieu De Mari

About this week (Week 4)

1. What are **images** and how is this datatype represented?
2. What is a **pixel** and how can **its information be interpreted**?
3. What is the **spatial dependence property** of pixels?
4. What is the **homophily property** of pixels?
5. Why is the **linear processing operation failing** on images?
6. What is the **convolution** operation?
7. How can we perform **image processing using convolutions**?
8. What is **padding** in convolutions?

About this week (Week 4)

9. What is **stride** in convolution?
10. What is **dilation** in convolution?
11. How does convolution apply to **higher dimensional images**?
12. How to write **our own convolutional processing layer** in PyTorch?
13. What is **Conv2d** in PyTorch?
14. What is a **Convolutional Neural Network** (CNN)?
15. What is the **intuition** behind using Convolutional layers in a Convolutional Neural Network?

About this week (Week 4)

16. How to **train** our first Convolutional Neural Network on MNIST?
17. What are **other typical image processing layers** in Computer Vision and how are they implemented?
18. What is the **pooling layer**?
19. What is the **batchnorm layer**?
20. What is the **dropout layer**?
21. What is **data augmentation** in Computer Vision?
22. What are some **milestone Computer Vision models** and their **contributions** to the field of Deep Learning?

About this week (Week 4)

23. What are the **AlexNet** and **VGG models**?
24. What is a **skip connection/residual**? What is its effect on a Neural Network and **how does it help with vanishing gradient problems**?
25. What are **ResNet** and **DenseNet models**?
26. What is the **Inception model**?
27. What is the **EfficientNet model**?
28. What is **transfer learning** and its uses?
29. How to **freeze** and **fine-tune layers** in a Neural Network?

Implementing a custom Conv in PyTorch

```python
def convolution_with_stride_and_padding_torch(image, kernel, stride = 1, padding = 0):
    # Convert image and kernel to PyTorch tensors
    image = torch.from_numpy(image)
    kernel = torch.from_numpy(kernel)

    # Flip the kernel (optional)
    kernel = torch.flip(torch.flip(kernel, [0]), [1])

    # Add padding to the image
    image = torch.nn.functional.pad(image, (padding, padding, padding, padding))

    # Set the output image to the correct size
    output_rows = (image.shape[0] - kernel.shape[0]) // stride + 1
    output_cols = (image.shape[1] - kernel.shape[1]) // stride + 1
    output = torch.zeros((output_rows, output_cols))

    # Convolve using PyTorch
    for i in range(0, output_rows, stride):
        for j in range(0, output_cols, stride):
            output[i, j] = (kernel * image[i:i + kernel.shape[0], j:j + kernel.shape[1]]).sum()

    return output
```

The Conv2d in PyTorch

PyTorch has a Conv2d function implementing said convolution.

We will rely on it from now on.

```python
def convolution_batch_torch_conv2d(images, kernel, stride = 1, padding = 0):

    # Convert kernel to PyTorch tensor, if needed
    kernel = torch.from_numpy(kernel)
    kernel = kernel.view(1, 1, kernel.shape[0], kernel.shape[1])
    kernel = kernel.float()

    # Flip the kernel (optional)
    kernel = torch.flip(torch.flip(kernel, [2]), [3])

    # Create a convolutional layer
    conv = torch.nn.Conv2d(in_channels = images.shape[1], \
                           out_channels = 1, \
                           kernel_size = kernel.shape[2:], \
                           stride = stride, \
                           padding = padding)

    # Assign the kernel to the layer
    conv.weight = torch.nn.Parameter(kernel)
    conv.bias = torch.nn.Parameter(torch.tensor([0.0]))

    # Perform convolution
    output = conv(images)

    return output
```

The Conv2d in PyTorch

Note, however that this layer has **two trainable parameters**:

- **Weight** (which is our kernel, whose values will be decided later during training).
- **Bias** (which was not there before?)

```python
def convolution_batch_torch_conv2d(images, kernel, stride = 1, padding = 0):

    # Convert kernel to PyTorch tensor, if needed
    kernel = torch.from_numpy(kernel)
    kernel = kernel.view(1, 1, kernel.shape[0], kernel.shape[1])
    kernel = kernel.float()

    # Flip the kernel (optional)
    kernel = torch.flip(torch.flip(kernel, [2]), [3])

    # Create a convolutional layer
    conv = torch.nn.Conv2d(in_channels = images.shape[1], \
                           out_channels = 1, \
                           kernel_size = kernel.shape[2:], \
                           stride = stride, \
                           padding = padding)

    # Assign the kernel to the layer
    conv.weight = torch.nn.Parameter(kernel)
    conv.bias = torch.nn.Parameter(torch.tensor([0.0]))

    # Perform convolution
    output = conv(images)

    return output
```

The Conv2d in PyTorch

Note, however that this layer has **two trainable parameters**:

- **Weight** (which is our kernel, whose values will be decided later during training).
- **Bias** (which was not there before?)

Bias is simply added to our convolution operation, following the intuition of the $WX + b$ operation from earlier.

$$Y_{i,j} = \cancel{\frac{1}{k^2}} \sum_{m=1}^{k} \sum_{n=1}^{k} X_{i+m-1,\, j+n-1}\, K_{m,n} + b$$

The Conv2d in PyTorch

In the PyTorch Conv2d operations however, the input images X need to be formatted as a 4D tensor of size $N_s \times c \times h \times w$, with

- N_s, the number of samples/images in batch of data X,
- c , the number of channels in images of batch X (e.g. greyscale = 1, RGB = 3),
- h , the height for images batch of data X,
- w , the width for images batch of data X.

Same thing for the kernel

The Conv2d in PyTorch

This is the reason for reshaping the kernel

- from a 2D NumPy array of size (k, k),
- to a 4D PyTorch tensor of size $(1, 1, k, k)$.

And eventually flipping the kernel along the last two dimensions.

```python
def convolution_batch_torch_conv2d(images, kernel, stride = 1, padding = 0):

    # Convert kernel to PyTorch tensor, if needed
    kernel = torch.from_numpy(kernel)
    kernel = kernel.view(1, 1, kernel.shape[0], kernel.shape[1])
    kernel = kernel.float()

    # Flip the kernel (optional)
    kernel = torch.flip(torch.flip(kernel, [2]), [3])

    # Create a convolutional layer
    conv = torch.nn.Conv2d(in_channels = images.shape[1], \
                           out_channels = 1, \
                           kernel_size = kernel.shape[2:], \
                           stride = stride, \
                           padding = padding)

    # Assign the kernel to the layer
    conv.weight = torch.nn.Parameter(kernel)
    conv.bias = torch.nn.Parameter(torch.tensor([0.0]))

    # Perform convolution
    output = conv(images)

    return output
```

If we use the MNIST dataloader…

```
# Define transform to convert images to tensors and normalize them
transform_data = Compose([ToTensor(),
                          Normalize((0.1307,), (0.3081,))])

# Load the data
batch_size = 256
train_dataset = MNIST(root='./mnist/', train = True, download = True, transform = transform_data)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size = batch_size, shuffle = True)

# Blur
kernel = np.array([[1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1]])/25

# Draw a single batch and try
# (Very slow!)
for inputs, _ in train_loader:
    outputs = convolution_batch_torch(inputs, kernel)
    outputs2 = convolution_batch_torch_conv2d(inputs, kernel)
    print(inputs.shape)
    print(outputs.shape)
    print(outputs2.shape)
    break
```

```
torch.Size([256, 1, 28, 28])
torch.Size([256, 1, 24, 24])
torch.Size([256, 1, 24, 24])
```

MNIST dataloader will give batches of data as 4D tensors of shape $N_s \times c \times h \times w$, with:

- **N_s, being batch size (here, 256),**
- **c , the number of channels (here, 1),**
- **h , the height for images (here, 28),**
- **w , the height for images (here, 28).**

If we use the MNIST dataloader…

```python
# Define transform to convert images to tensors and normalize them
transform_data = Compose([ToTensor(),
                          Normalize((0.1307,), (0.3081,))])

# Load the data
batch_size = 256
train_dataset = MNIST(root='./mnist/', train = True, download = True, transform = transform_data)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size = batch_size, shuffle = True)

# Blur
kernel = np.array([[1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1],
                   [1, 1, 1, 1, 1]])/25

# Draw a single batch and try
# (Very slow!)
for inputs, _ in train_loader:
    outputs = convolution_batch_torch(inputs, kernel)
    outputs2 = convolution_batch_torch_conv2d(inputs, kernel)
    print(inputs.shape)
    print(outputs.shape)
    print(outputs2.shape)
    break
```

```
torch.Size([256, 1, 28, 28])
torch.Size([256, 1, 24, 24])
torch.Size([256, 1, 24, 24])
```

Not using a padding, stride 1, and kernel size is 5.

Our magic formula from earlier then gives:

$$28 + 2 \times 0 - 5 + 1 = 24.$$

A quick word on higher dimensions

So far, we have implemented the convolution operation on images that were greyscale (either our flower image or the MNIST dataset).

What happens when playing with higher dimension images, i.e. images that have more than one channel (e.g. RGB)?

Convolution on high dim

In the case of RGB images, the resulting matrix Y is then of size $h' \times w' \times 3$.

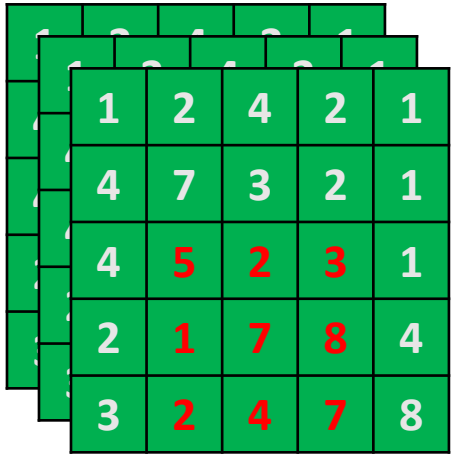
The pixel values $y_{i,j,l}$ for image Y are calculated using the convolution operation, on **each channel separately**.

We then define pixels $Y_{i,j,l}$,
$\forall i \in [1, h'], j \in [1, w'], \forall l \in [1, 3]$:

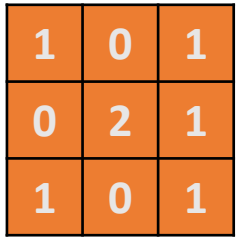
$$Y_{i,j,l} = \frac{1}{k^2} \sum_{m=1}^{k} \sum_{n=1}^{k} X_{i+m-1,\, j+n-1,\, l}\, K_{m,n} + b$$

This operation preserves the original number of channels of original image X, producing a new RGB image Y as a result.

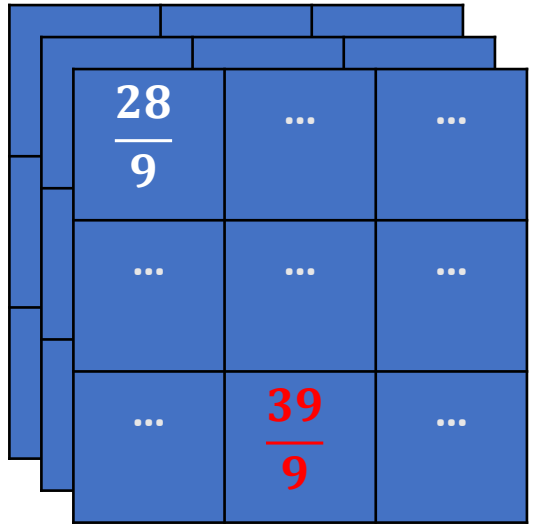
X



K



Y



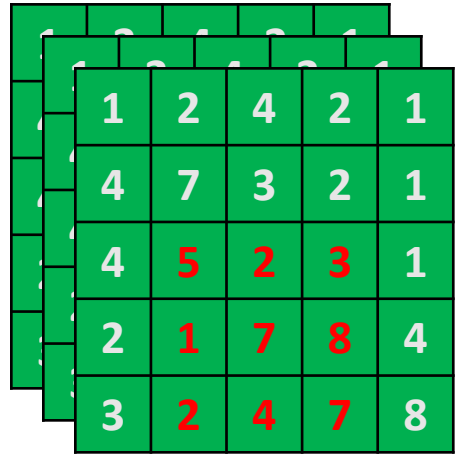
Convolution on high dim

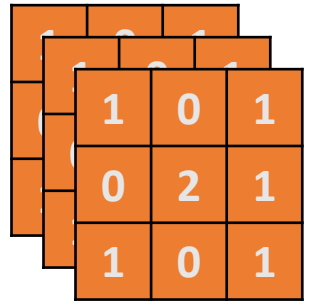
More generally, however: if you define the kernel as a tensor with its own number of channels $k_d \neq 1$…

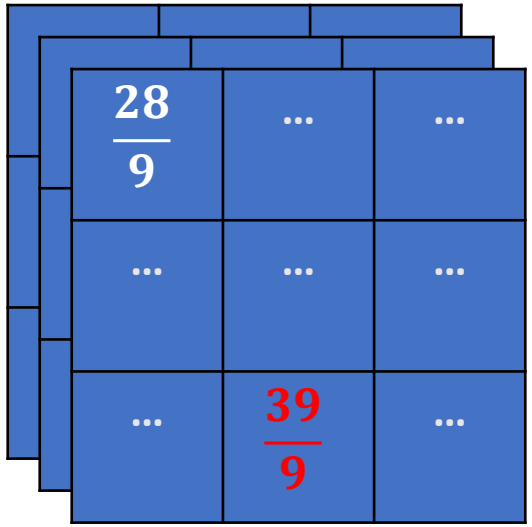
We then define pixels $Y_{i,j,l}$,
$\forall i \in [1, h'], j \in [1, w'], \forall l \in [1, k_d]$:

$$Y_{i,j,l} = \frac{1}{k^2} \sum_{m=1}^{k} \sum_{n=1}^{k} \sum_{l'=1}^{3} X_{i+m-1,\, j+n-1,\, l'} \, K_{m,n,l} + b$$

This operation will then produce a new image Y, whose number of channels matches the one of the kernel K, as a result, i.e. $h' \times w' \times k_d$.







Conv2d layer

Definition (the Conv2d layer):

The **Conv2d layer** is PyTorch's implementation of the convolution operation we have seen earlier.

The first two integer values correspond to the number of channels of input images X and the number of channels in kernel.

The convolution layer below then expects grayscale images X with only one channel and will produce images Y with 32 channels.

→ What is this image with 32 channels?

```
nn.Conv2d(1, 32, kernel_size = 3, stride = 1, padding = 1)
```

Convolution on high dim

→ What is this image with 32 channels?

- We could visualize that as 32 convolutions operations on our grayscale images X, being run on parallel with 32 different kernels.
- Kernels could correspond to different operations (blur, sharpen, edge detection, etc.).

```
nn.Conv2d(1, 32, kernel_size = 3, stride = 1, padding = 1)
```

Chaining convolutions

→ What is this image with 32 channels?

- We could visualize that as 32 convolutions operations on our grayscale images X, being run on parallel with 32 different kernels.
- Kernels could correspond to different operations (blur, sharpen, edge detection, etc.).
- Eventually chaining several convolutions, will lead to combining operations together.

```
nn.Conv2d(1, 32, kernel_size = 3, stride = 1, padding = 1)
nn.Conv2d(32, 64, kernel_size = 3, stride = 1, padding = 1)
```

Chaining convolutions

- **For instance:** recognizing a cat ear = two edge detections, then combined to recognize a triangular shape or some sort?

The core idea behind CNN

Definition (Convolutional Neural Networks):

Introduced in [LeCun1998], **Convolutional Neural Networks** (or **CNNs** in short) are Neural Networks which integrate convolutions as the processing operations to be used on input images, in place of the Linear ones.

Training a CNN is then equivalent to **letting the Neural Network decide on which kernels values** (and therefore image processing operations) to use in the convolution operations run in both parallel and sequence.

Training a CNN

- CNNs have shown much better performance at processing images data types, compared to Neural Networks implementing Linear operations only.
- For this reason, **Convolutions (and CNNs) are indeed the preferred processing method when inputs are images.**

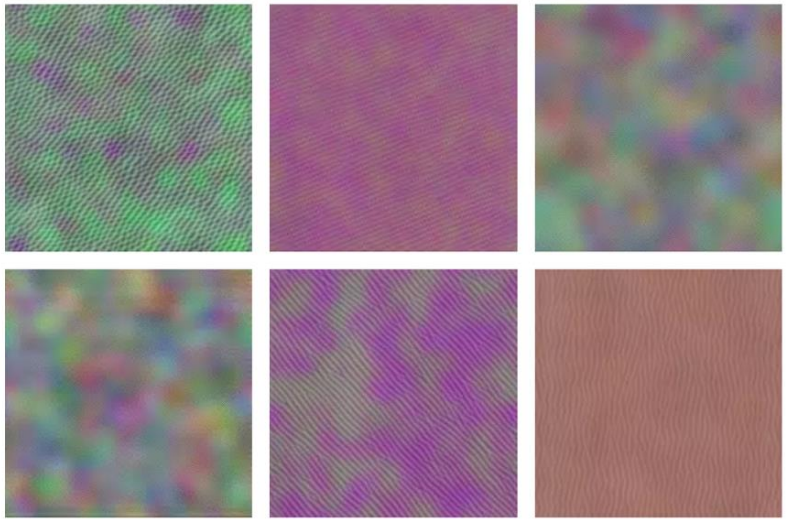
- In addition, as the convolution is “just another weird matrix multiplication of some sort”, this means that **the trainable parameters (weight/kernel and bias)** could be trained **using the same backpropagation as earlier**!
- *(We will however not implement it and rely on autograd!)*

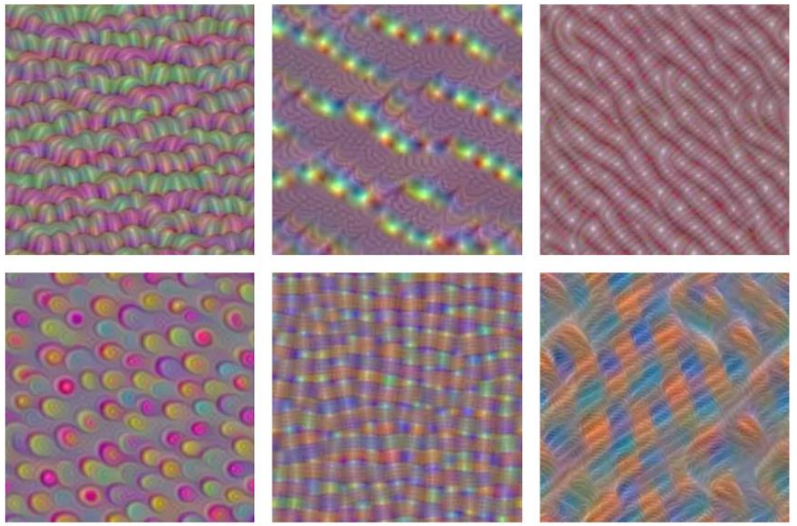
Chaining convolutions

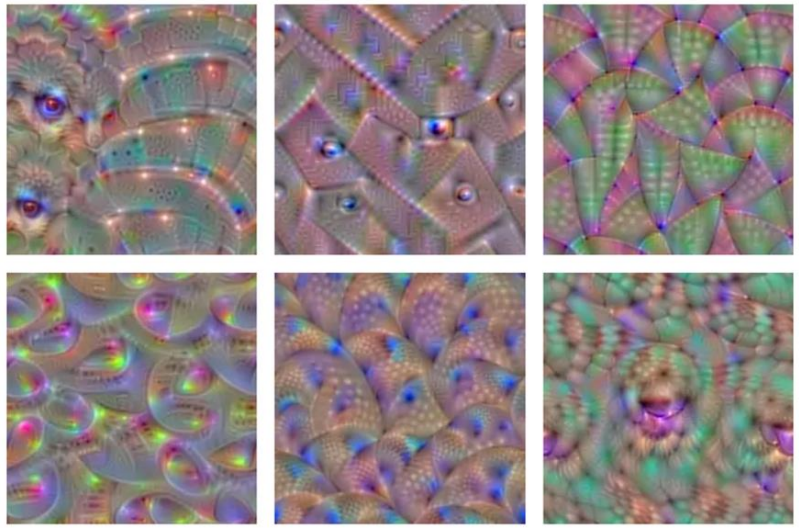
- **For instance:** recognizing a cat ear = two edge detections, then combined to recognize a triangular shape or some sort?

Chaining convolutions

- **Week 12-13 (explainability on CNNs):** We will show during this W12-13 lecture how the first layers are typically implementing basic shapes and colors detection, whereas deeper layers are progressively recognizing more and more sophisticated patterns.







Our first CNN on MNIST

Using previous concepts and intuitions, we will therefore build our first Convolutional Neural Network (CNN).

- This network will have the same logic as the previous DNN from last week, but it will **replace the first few Linear layers with Conv2d operations instead**.
- By doing so, we are hoping that **the CNN will be able to figure out which kernel values (and image processing operations) to use**, for the MNIST classification task.

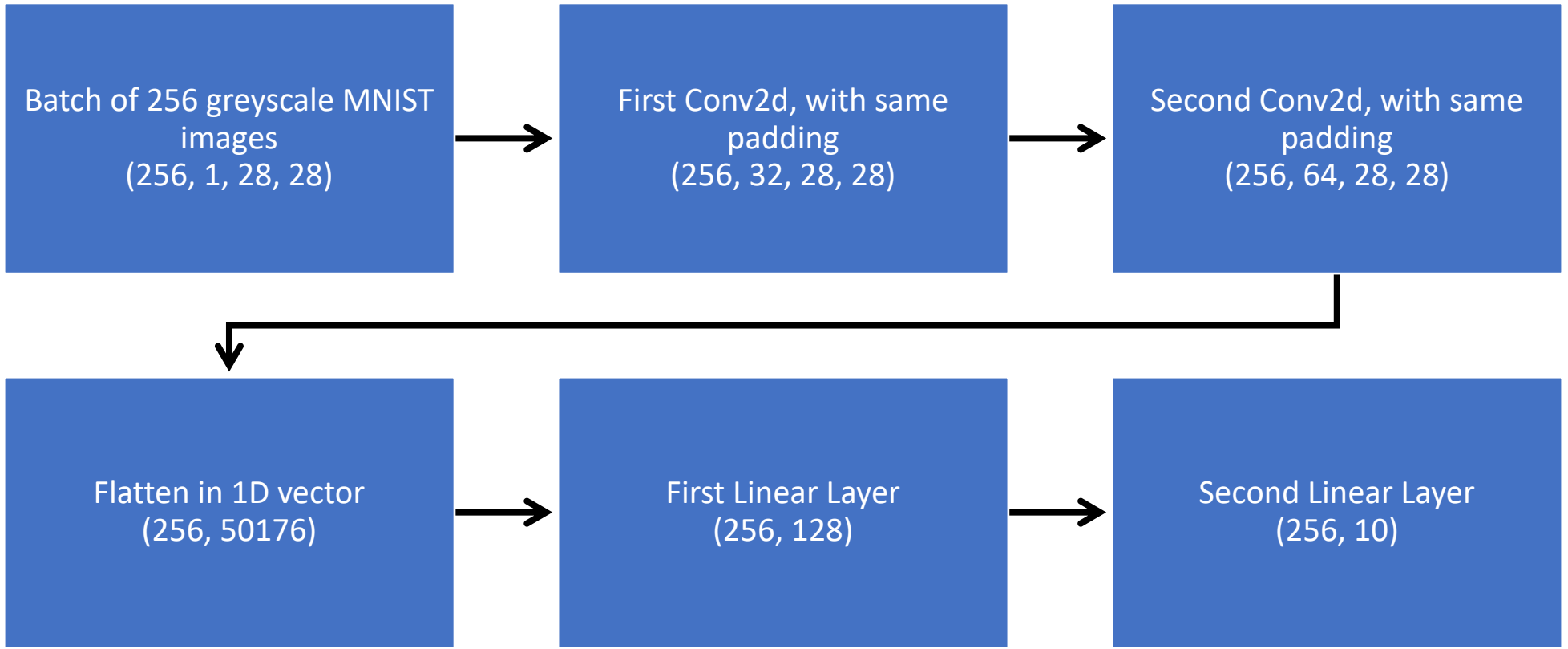
Our first CNN on MNIST

We will therefore build our first Convolutional Neural Network (CNN).

- The **final layers must however consist of Linear layers**, whose purpose is to eventually produce a 1D vector of 10 values, consisting of the probabilities for each MNIST class.
- Between the last Conv2d (which produces a 2D image) and the first fully connected vector (which expects a 1D vector), we will implement a **flattening** of the image, as before.

Our first CNN on MNIST

Our CNN: 2 Conv2d layers + Flatten + 2 Linear layers



Our first CNN on MNIST

Our CNN: 2 Conv2d layers + Flatten + 2 Linear layers

```
class MNIST_CNN(nn.Module):
    def __init__(self):
        super(MNIST_CNN, self).__init__()

        # Two convolutional layers
        self.conv1 = nn.Conv2d(1, 32, kernel_size = 3, stride = 1, padding = 1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size = 3, stride = 1, padding = 1)

        # Two fully connected layers
        self.fc1 = nn.Linear(64*28*28, 128) # 64*28*28 = 50176
        self.fc2 = nn.Linear(128, 10)
```

Our first CNN on MNIST

Our CNN: 2 Conv2d layers + Flatten + 2 Linear layers

- Define the sequence of operations in the forward method as before.
- **Good practice:** We will print the shape of each layer output to confirm sizes produced by each layer.

```python
def forward(self, x):
    # Display initial shape
    print("Initial: ", x.shape)

    # Pass input through first convolutional layer
    x = self.conv1(x)
    x = F.relu(x)
    print("After conv1: ", x.shape)

    # Pass output of first conv layer through
    # second convolutional layer
    x = self.conv2(x)
    x = F.relu(x)
    print("After conv2: ", x.shape)

    # Flatten output of second conv layer
    x = x.view(-1, 64*28*28)
    print("After flatten: ", x.shape)

    # Pass flattened output through first Linear layer
    x = self.fc1(x)
    x = F.relu(x)
    print("After FC1: ", x.shape)

    # Pass output of first Linear layer to second Linear layer
    x = self.fc2(x)
    print("After FC2: ", x.shape)
    return x
```

Our first CNN on MNIST

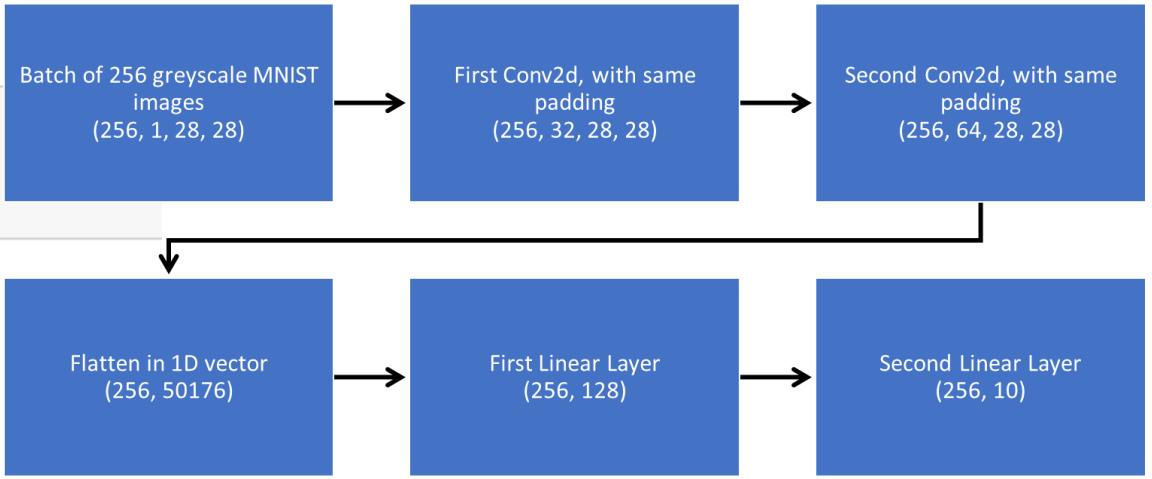
Good practice: We will show the modules/layers in our CNN, and print the shape of each layer output to confirm sizes produced by each layer.

```
model = MNIST_CNN()
print(model.modules)
```

```
<bound method Module.modules of MNIST_CNN(
  (conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=50176, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=10, bias=True)
)>
```

```
for inputs, labels in train_loader:
    out = model(inputs)
    break
```

```
Initial:  torch.Size([256, 1, 28, 28])
After conv1:  torch.Size([256, 32, 28, 28])
After conv2:  torch.Size([256, 64, 28, 28])
After flatten:  torch.Size([256, 50176])
After FC1:  torch.Size([256, 128])
After FC2:  torch.Size([256, 10])
```



Writing a trainer function for CNNs

As before, we need a **trainer** and **evaluation function** for our Convolution Neural Network model.

- We will be using a **stochastic mini-batch Adam optimizer**, like before.
- **Loss** is **cross entropy**, like before.
- We will keep track of the **train/test losses**, **train/ test accuracies**, and display them in training performance curves later.
- Over a given number of iterations, using the power of the **backward**() and **optimizer.step()** methods to update parameters automatically for us in all the Conv2d and Linear layers.
- We will then set the model on **eval mode** and **compute losses** and **accuracies** on the testing set, one final time.

Trainer/Eval function (setup)

```
def train(model, train_loader, test_loader, epochs = 10, lr = 0.001):
    # Use Adam optimizer to update model weights
    optimizer = optim.Adam(model.parameters(), lr = lr)
    # Use cross-entropy loss function
    criterion = nn.CrossEntropyLoss()
    # Performance curves data
    train_losses = []
    train_accuracies = []
    test_losses = []
    test_accuracies = []
```

Trainer/Eval function (training part)

```python
for epoch in range(epochs):
    # Set model to training mode
    model.train()
    # Initialize epoch loss and accuracy
    epoch_loss = 0.0
    correct = 0
    total = 0
    # Iterate over training data
    for batch_number, (inputs, labels) in enumerate(train_loader):
        # Get from dataloader and send to device
        inputs = inputs.to(device)
        labels = labels.to(device)
        # Zero out gradients
        optimizer.zero_grad()
        # Compute model output and loss
        outputs = model(inputs)
        _, predicted = torch.max(outputs.data, 1)
        loss = criterion(outputs, labels)
        # Backpropagate loss and update model weights
        loss.backward()
        optimizer.step()
        # Accumulate loss and correct predictions for epoch
        epoch_loss += loss.item()
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
        print(f'Epoch {epoch+1}/{epochs}, Batch number: {batch_number}, Cumulated accuracy: {correct/total}')
    # Calculate epoch loss and accuracy
    epoch_loss /= len(train_loader)
    epoch_acc = correct/total
    train_losses.append(epoch_loss)
    train_accuracies.append(epoch_acc)
    print(f'--- Epoch {epoch+1}/{epochs}: Train loss: {epoch_loss:.4f}, Train accuracy: {epoch_acc:.4f}')
```

Trainer/Eval function (evaluation part)

```
        # Set model to evaluation mode
        model.eval()
        # Initialize epoch loss and accuracy
        epoch_loss = 0.0
        correct = 0
        total = 0
        # Iterate over test data
        for inputs, labels in test_loader:
            # Get from dataloader and send to device
            inputs = inputs.to(device)
            labels = labels.to(device)
            # Compute model output and loss
            # (No grad computation here, as it is the test data)
            with torch.no_grad():
                outputs = model(inputs)
                _, predicted = torch.max(outputs.data, 1)
                loss = criterion(outputs, labels)
            # Accumulate loss and correct predictions for epoch
            epoch_loss += loss.item()
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
        # Calculate epoch loss and accuracy
        epoch_loss /= len(test_loader)
        epoch_acc = correct/total
        test_losses.append(epoch_loss)
        test_accuracies.append(epoch_acc)
        print(f'--- Epoch {epoch+1}/{epochs}: Test loss: {epoch_loss:.4f}, Test accuracy: {epoch_acc:.4f}')

    return train_losses, train_accuracies, test_losses, test_accuracies
```

Trainer/Eval function (running)

```
model = MNIST_CNN().to(device)
train_losses, train_accuracies, test_losses, test_accuracies = train(model, \
                                                                     train_loader, \
                                                                     test_loader, \
                                                                     epochs = 3, \
                                                                     lr = 1e-3)
```

```
Epoch 1/3, Batch number: 0, Cumulated accuracy: 0.06640625
Epoch 1/3, Batch number: 1, Cumulated accuracy: 0.10546875
Epoch 1/3, Batch number: 2, Cumulated accuracy: 0.15494791666666666
Epoch 1/3, Batch number: 3, Cumulated accuracy: 0.2421875
Epoch 1/3, Batch number: 4, Cumulated accuracy: 0.29296875
Epoch 1/3, Batch number: 5, Cumulated accuracy: 0.3502604166666667
Epoch 1/3, Batch number: 6, Cumulated accuracy: 0.40457589285714285
Epoch 1/3, Batch number: 7, Cumulated accuracy: 0.44580078125
Epoch 1/3, Batch number: 8, Cumulated accuracy: 0.4830729166666667
Epoch 1/3, Batch number: 9, Cumulated accuracy: 0.518359375
Epoch 1/3, Batch number: 10, Cumulated accuracy: 0.5461647727272727
Epoch 1/3, Batch number: 11, Cumulated accuracy: 0.5738932291666666
Epoch 1/3, Batch number: 12, Cumulated accuracy: 0.5964543269230769
Epoch 1/3, Batch number: 13, Cumulated accuracy: 0.6177455357142857
Epoch 1/3, Batch number: 14, Cumulated accuracy: 0.63671875
Epoch 1/3, Batch number: 15, Cumulated accuracy: 0.651123046875
Epoch 1/3, Batch number: 16, Cumulated accuracy: 0.6659007352941176
Epoch 1/3, Batch number: 17, Cumulated accuracy: 0.6796875
Epoch 1/3, Batch number: 18, Cumulated accuracy: 0.6903782894736842
```

Trainer/Eval function (running)

- In only three iterations, we manage to achieve a higher accuracy than the equivalent Linear model we have trained last week for almost 1000 iterations!
- Confirmed!
- **CNNs and Convolution operations are indeed better than Linear operations when it comes to processing images!**

```
Epoch 3/3, Batch number: 229, Cumulated accuracy: 0.9927649456521739
Epoch 3/3, Batch number: 230, Cumulated accuracy: 0.9927793560606061
Epoch 3/3, Batch number: 231, Cumulated accuracy: 0.9928104795258621
Epoch 3/3, Batch number: 232, Cumulated accuracy: 0.9928078057939914
Epoch 3/3, Batch number: 233, Cumulated accuracy: 0.9927884615384616
Epoch 3/3, Batch number: 234, Cumulated accuracy: 0.9928
--- Epoch 3/3: Train loss: 0.0227, Train accuracy: 0.9928
--- Epoch 3/3: Test loss: 0.0410, Test accuracy: 0.9865
```

Pooling layers

Definition (pooling layers):

A **pooling layer** in a CNN is a **down-sampling operation** that strongly reduces the spatial dimensions (**width** and **height**) of the successive images or feature maps, while retaining the important information.

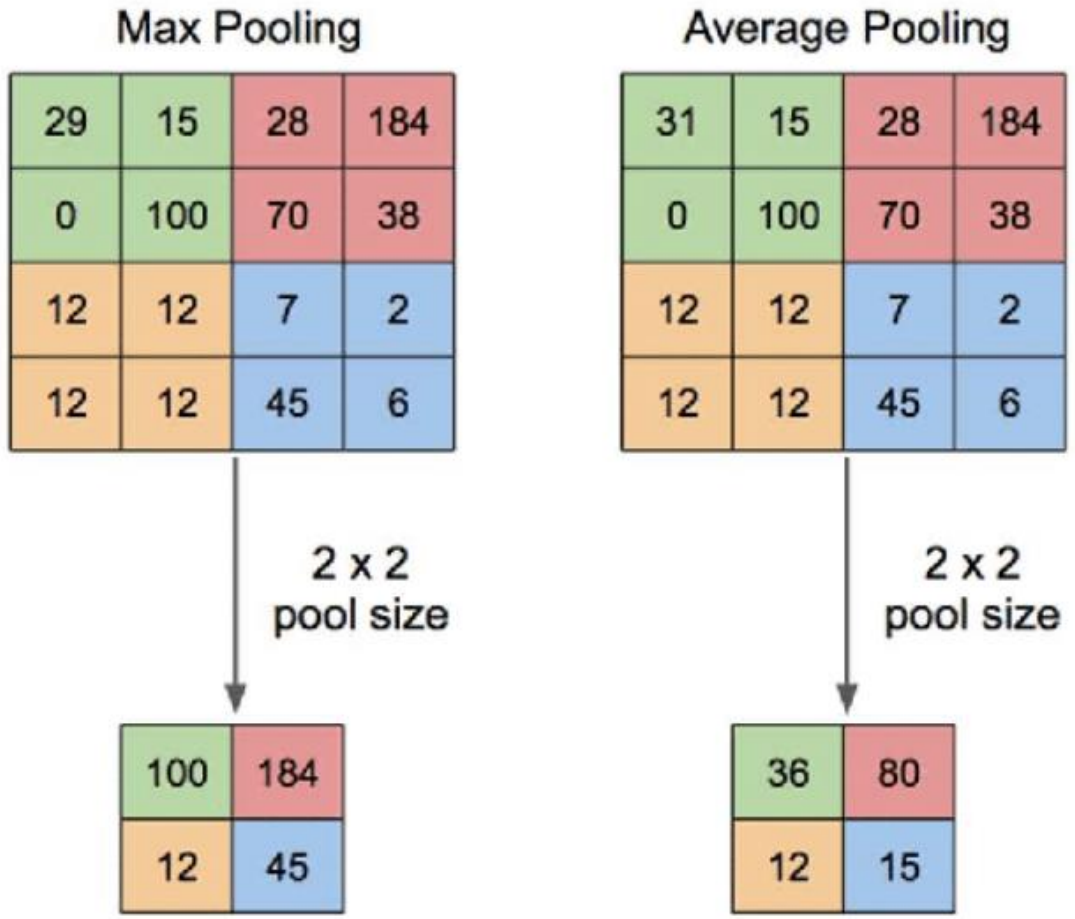
The purpose of pooling is to reduce the computation and memory required by the network, as well as to increase the invariance of the features with respect to small translations (from the same paper, [LeCun1998]).

A pooling layer, like the flattening operation, does not have any trainable parameters, and is therefore not a trainable layer.

Types of pooling layers

In PyTorch, there are different types of **pooling layers**, all worth trying, including:

- **Max pooling:** Selects the maximum value from the window,
- **Average pooling:** Takes the average of all values,
- **Sum pooling:** Takes the sum of all values in the window.
- Etc.



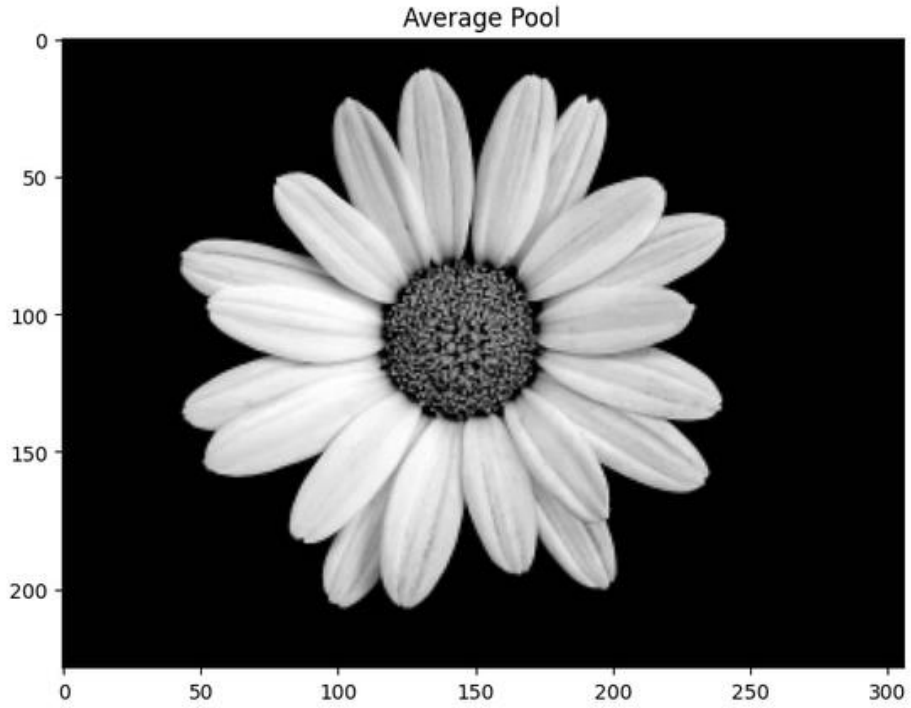
Types of pooling layers

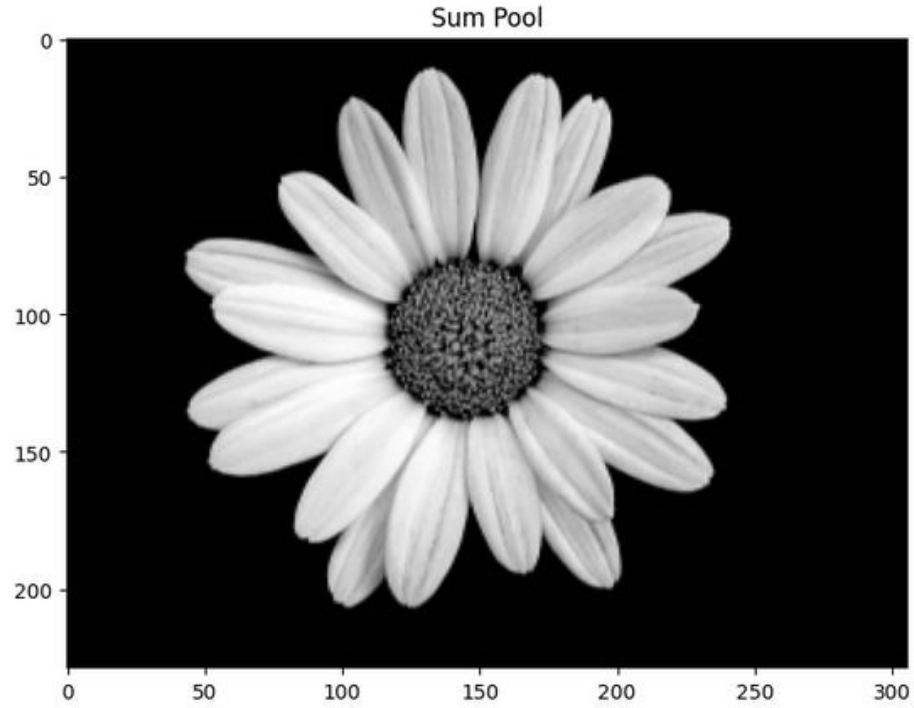
```python
# Seeing the effect of some pooling layers on data
# Open the image and convert it to grayscale
im = Image.open('flower.jpg').convert('L')
# Convert the image to a Tensor
im_tensor = torch.from_numpy(np.array(im)).float().reshape(1, 459, 612)
print(im_tensor.shape)

# Define the pooling layers
max_pool = nn.MaxPool2d(kernel_size = 2, stride = 2)
avg_pool = nn.AvgPool2d(kernel_size = 2, stride = 2)
sum_pool = nn.AdaptiveAvgPool2d(output_size = (229, 306))

# Apply the pooling layers to the image tensor
img_max_pool = max_pool(im_tensor)
img_avg_pool = avg_pool(im_tensor)
img_sum_pool = sum_pool(im_tensor)

# Print the shapes of the pooled images
print(img_max_pool.shape)
print(img_avg_pool.shape)
print(img_sum_pool.shape)
```

```
torch.Size([1, 459, 612])
torch.Size([1, 229, 306])
torch.Size([1, 229, 306])
torch.Size([1, 229, 306])
```


Original Image
0
100
200
300
400
0
100
200
300
400
500
600


Max Pool
0
50
100
150
200
0
50
100
150
200
250
300


Average Pool
0
50
100
150
200
0
50
100
150
200
250
300

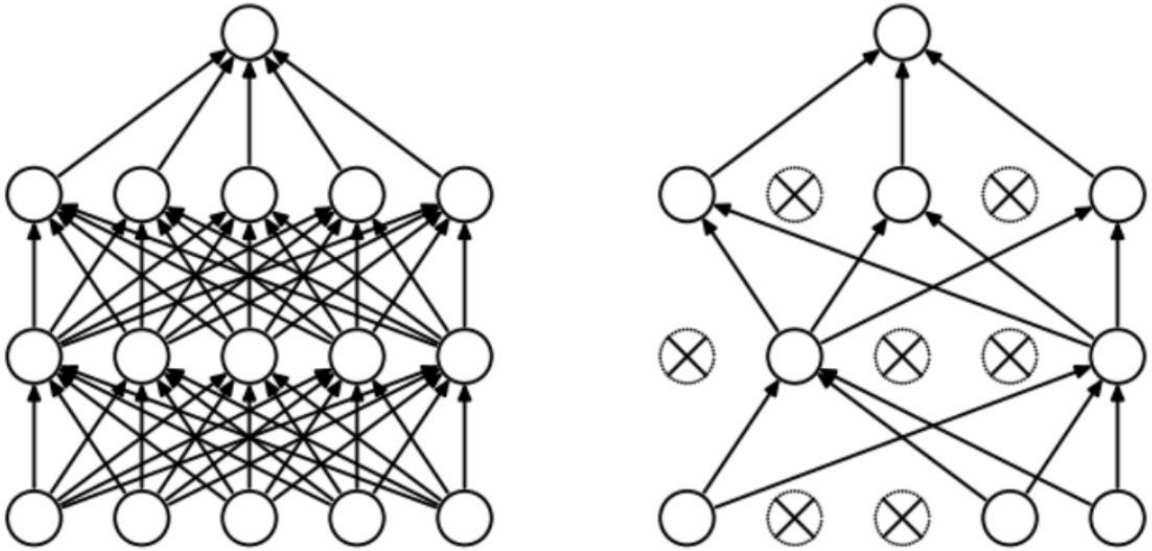

Sum Pool
0
50
100
150
200
0
50
100
150
200
250
300

Dropout layers

Definition (dropout layers):

Introduced in [Srivastava2014], the **dropout layer** is a **regularization technique** used in deep learning to **prevent overfitting.**

The dropout layer **randomly drops out, or sets to zero, some of the neurons** in the network **during each training iteration**.



(a) Standard Neural Net

(b) After applying dropout.

Dropout layers

Definition (dropout layers):

Introduced in [Srivastava2014], the **dropout layer** is a **regularization technique** used in deep learning to **prevent overfitting.**

The dropout layer **randomly drops out, or sets to zero, some of the neurons** in the network **during each training iteration**.

In PyTorch, dropout is applied using the **nn.Dropout** module, and it is typically inserted

- **after a linear or convolutional layer**,
- and **before a non-linear activation** function.

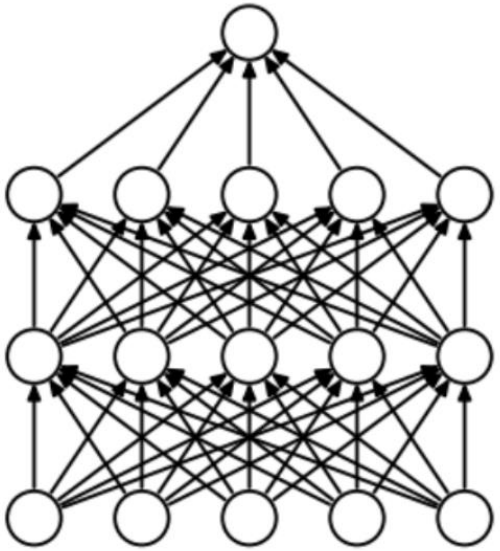
The nn.Dropout takes one argument: the probability p of an element to be zeroed.

Dropout layers

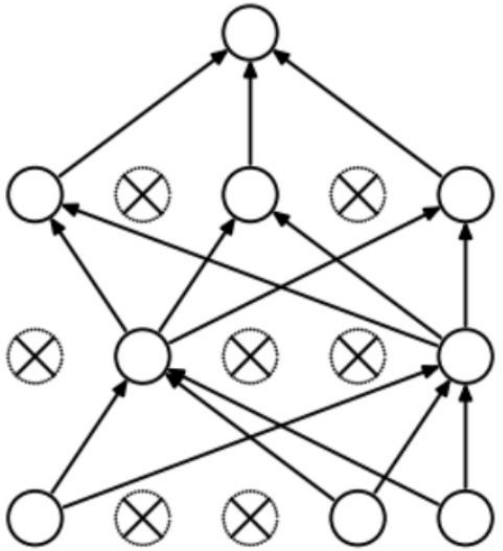
E.g. using $p = 0.5$ means that during the forward pass, half of the neurons will be set to zero.

The Dropout layer is **not trainable** as the parameter p is to be manually decided, usually with a value between 0.1 and 0.5.

It is therefore **another hyperparameter** to take into account!



(a) Standard Neural Net

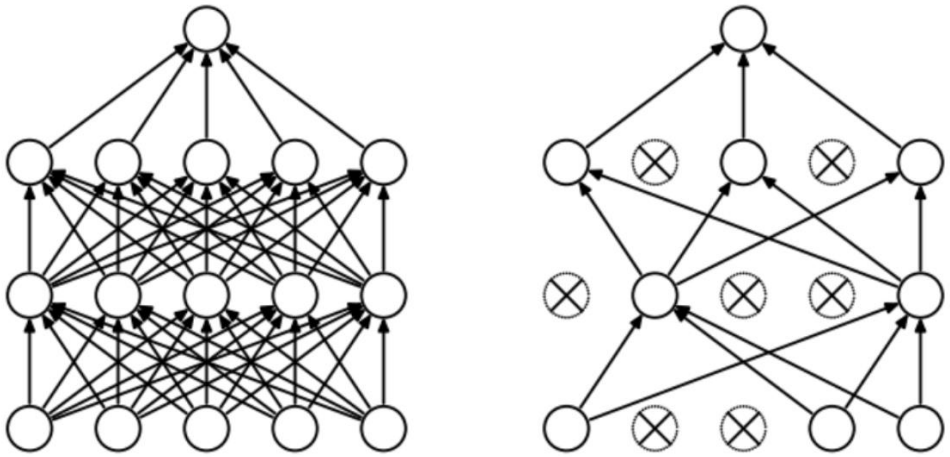


(b) After applying dropout.

A very important note, however:

Dropout is the first operation we see to have **two different behaviours**.

- During **training mode** (i.e. when model is set to **train()** mode), the dropout layer **randomly drops out a specified portion of the neurons**, as said earlier.



(a) Standard Neural Net

(b) After applying dropout.

- During **evaluation mode** (i.e. when model is set to **eval()** mode), it does **not drop out any neurons** anymore. Instead, it passes all activations from the previous layer to the next layer in the network, in an attempt to maximize performance on the test set.

Batchnorm layer

Definition (the batchnorm layer):

Batch normalization (or **batchnorm**) is a technique used to improve the stability and performance of neural networks. It was presumably introduced in [Ioffe2015].

It **normalizes the activations** produced by each layer for each mini-batch. The purpose of batchnorm is to reduce internal covariate shift, which refers to the change in the distribution of network activations due to the change in network parameters during training.

This can make the training process faster and more stable.

≈ **Reuses the same intuition as to why inputs in datasets should be normalized, but applies it to the inputs of all layers in network.**

Batchnorm layer

In PyTorch, **batchnorm** can be applied to a layer using the **nn.BatchNorm2d** or **nn.BatchNorm1d** module, depending on whether the data is 1D shaped (output of a linear layer) or 2D shaped (e.g. image output of a Convolutional layer)

Like the dropout, the module should be inserted

- **after a linear** or **convolutional layer** in the network,
- and **before the non-linear activation function**.

The batchnorm layer takes one argument: the number of channels in the output of the previous layer.

Normalization in BatchNorm

The equation for batch normalization (batchnorm) is as follows:

$$y = \frac{\gamma(x - \mu)}{\sigma} + \beta$$

Where:

- x (resp. y) is the **input (resp. output) to the** batchnorm layer,
- μ and σ are the **mean** and **standard** deviation of the batch, respectively, and are calculated separately for each feature dimension in the input batch x.
- γ and β are **trainable parameters** for the batchnorm layer, also known as **scale** and **shift** parameters.

Normalization in BatchNorm

```python
# Seeing the effect of batchnorm on data
# Initialize a random tensor with uniform distribution
# (definitely not normal with zero mean and variance 1).
x = torch.rand(size = (100, 1))
print("Mean before batchnorm: ", x.mean())
print("Variance before batchnorm: ", x.var())

# Create a batchnorm layer
batchnorm = nn.BatchNorm1d(1)

# Pass the tensor through the batchnorm layer
x = batchnorm(x)

# Print the mean and variance of the tensor after batchnorm
print("Mean after batchnorm: ", x.mean())
print("Variance after batchnorm: ", x.var())
```

```
Mean before batchnorm:  tensor(0.4873)
Variance before batchnorm:  tensor(0.0825)
Mean after batchnorm:  tensor(2.1458e-08, grad_fn=<MeanBackward0>)
Variance after batchnorm:  tensor(1.0100, grad_fn=<VarBackward0>)
```

About BatchNorm behaviours

Important: BatchNorm, just like Dropout **has two different behaviours in train and eval modes**.

- When the network is in **training mode**, the batch normalization layer updates its mean and variance estimates based on the current mini-batch of data.

  During the forward pass, it applies the normalization based on these mean and variance estimates.

- When the network is in **evaluation mode**, the batch normalization layer **uses the mean and variance estimates calculated during training to normalize the activations**.

  This ensures that the activations are normalized in the same way during training and evaluation.

Implementing all layers in our CNN

Let us first define Batchnorm, dropout and maxpool layers in __init__. Arbitrarily fixing the dropout probabilities (should be hyperparameter?).

```
class MNIST_CNN_all(nn.Module):
    def __init__(self):
        super(MNIST_CNN_all, self).__init__()
        # Two convolutional layers
        self.conv1 = nn.Conv2d(1, 32, kernel_size = 3, stride = 1, padding = 1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size = 3, stride = 1, padding = 1)
        # Two fully connected layers
        self.fc1 = nn.Linear(64*28*28, 128) # 64*28*28 = 50176
        self.fc2 = nn.Linear(128, 10)
        # Batch normalization layers
        self.batch_norm1 = nn.BatchNorm2d(32)
        self.batch_norm2 = nn.BatchNorm2d(64)
        self.batch_norm3 = nn.BatchNorm1d(128)
        # Dropout layers
        self.dropout1 = nn.Dropout2d(0.25)
        self.dropout2 = nn.Dropout2d(0.5)
        self.dropout3 = nn.Dropout(0.25)
        # MaxPool layers
        self.maxpool2d = F.max_pool2d
```

Implementing all layers in our CNN

- Using each layer in correct sequence of the forward() method.
- Also, one MaxPool layer after last Conv2d.
- 64*14*14 instead of 64*28*28 in Flatten due to MaxPool

```
def forward(self, x):
    # Pass input through first convolutional layer
    x = self.conv1(x)
    x = F.relu(x)
    x = self.batch_norm1(x)
    x = self.dropout1(x)
    # Pass output of first conv layer through second convolutional layer
    # Pooling only once on second layer (we could also do it on the first one)
    x = self.conv2(x)
    x = F.relu(x)
    x = self.batch_norm2(x)
    x = self.dropout2(x)
    x = self.maxpool2d(x, 2)
    # Flatten output of second conv layer
    x = x.view(-1, 64*14*14)
    # Pass flattened output through first fully connected layer
    x = self.fc1(x)
    x = F.relu(x)
    x = self.batch_norm3(x)
    x = self.dropout3(x)
    # Pass output of first fully connected layer through second fully connected layer
    x = self.fc2(x)
    return x
```

Implementing all layers in our CNN

```
model = MNIST_CNN_all()
print(model.modules)
```

```
<bound method Module.modules of MNIST_CNN_all(
  (conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=12544, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=10, bias=True)
  (batch_norm1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (batch_norm2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (batch_norm3): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (dropout1): Dropout2d(p=0.25, inplace=False)
  (dropout2): Dropout2d(p=0.5, inplace=False)
  (dropout3): Dropout(p=0.25, inplace=False)
)>
```

```
for inputs, labels in train_loader:
    out = model(inputs)
    print(out.shape)
    print(labels.shape)
    break
```

```
torch.Size([256, 10])
torch.Size([256])
```

A final note

The Dropout, Batchnorm and Pooling layers have been introduced in the Computer Vision field.

However, these days, every other field of Deep Learning (e.g. Natural Language Processing) is reusing them in their own models.

Important lesson: you might not necessarily be interested in Computer Vision, but you should still consider learning about it, as many intuitions and tools could be reused in other problems.

Learn more about these topics

Out of class, supporting papers, for those of you who are curious.

- [LeCun1998] Y. **LeCun**, L. **Bottou**, Y. **Bengio**, and P. **Haffner**, “Gradient-Based Learning Applied to Document Recognition”, 1998.
- [Srivastava2014] N. **Srivastava**, G. **Hinton**, A. **Krizhevsky**, I. **Sutskever**, and R. **Salakhutdinov**, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, 2014.
- [Ioffe2015] S. **Ioffe** and C. **Szegedy**, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”, 2015.

Learn more about these topics

Tracking important names (Track their works and follow them on Scholar, Twitter, or whatever works for you!)

- **Nitish Srivastava: Researcher** at **Google**.
  https://scholar.google.com/citations?user=s1PgoeUAAAAJ&hl=fr
  https://nitish2112.github.io/
- **Alex Krizhevsky: Former researcher** at **Google**. Now works for a company called **Dessa**, and **University of Toronto**.
  https://scholar.google.com/citations?user=xegzhJcAAAAJ&hl=fr

Learn more about these topics

Tracking important names (Track their works and follow them on Scholar, Twitter, or whatever works for you!)

- **Ilya Sutskever: Co-founder** and **Chief Scientist** at **OpenAI**.
  https://scholar.google.com/citations?user=x04W_mMAAAAJ&hl=en
  https://www.cs.toronto.edu/~ilya/
- **Ruslan Salakhutdinov: Professor** at **Carnegie Mellon University**.
  https://scholar.google.co.uk/citations?user=ITZ1e7MAAAAJ&hl=en
  https://www.cs.cmu.edu/~rsalakhu/

Learn more about these topics

Tracking important names (Track their works and follow them on Scholar, Twitter, or whatever works for you!)

- **Sergey Ioffe: Researcher** at **Google Brain**.
  https://scholar.google.com/citations?user=3QeF7mAAAAAJ&hl=en
  https://research.google/people/ChristianSzegedy
- **Christian Szegedy: Researcher** at **Google Brain**.
  https://scholar.google.com/citations?user=S5zOyIkAAAAJ&hl=en
  https://research.google/people/SergeyIoffe/