Politecnico di Torino

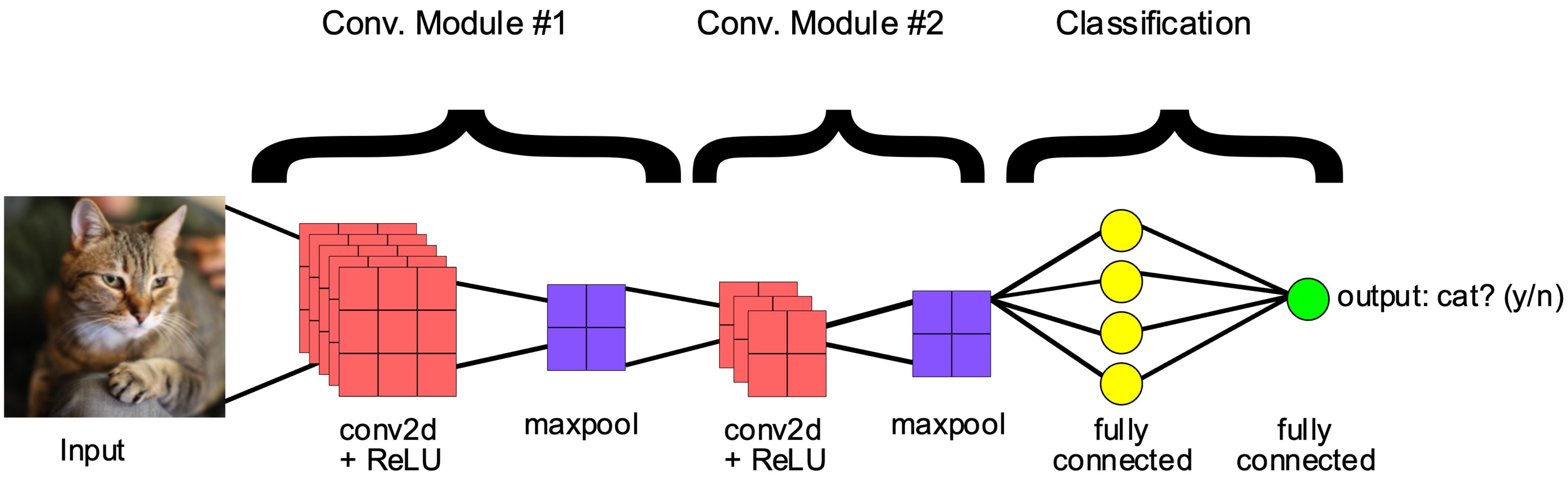
1859

Lab 5: AlexNet

Machine Learning and Deep Learning

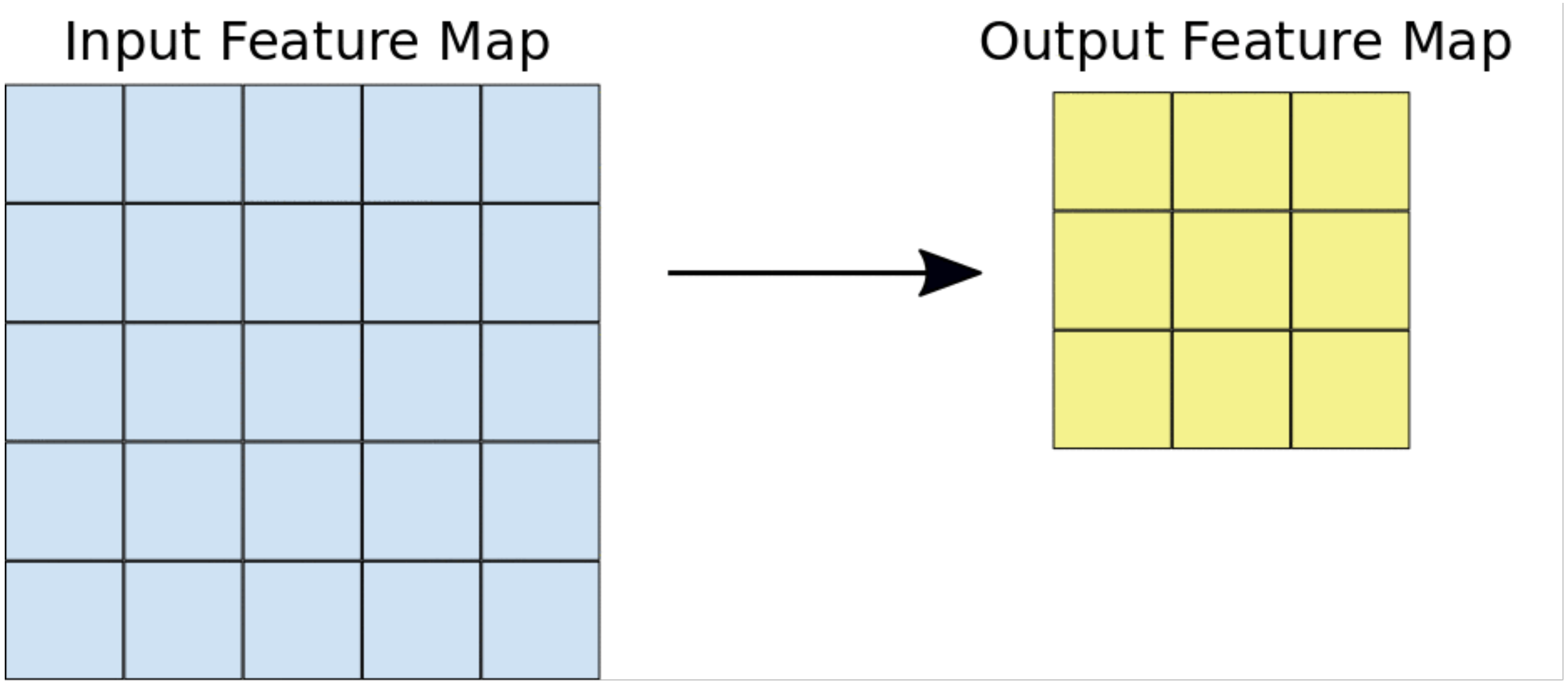
Niccolò Cavagnero, Chiara Plizzari

Convolutional Network

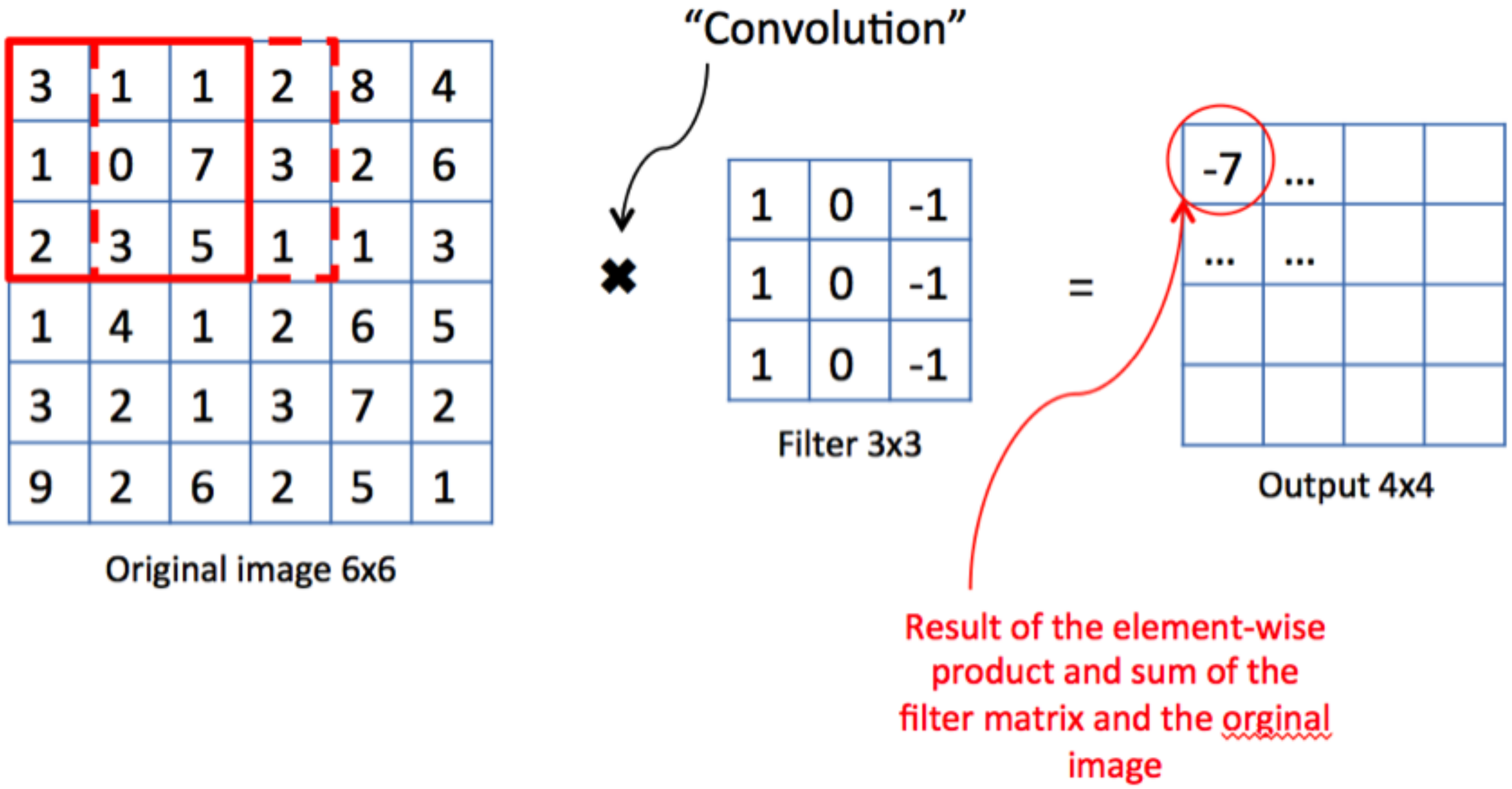


Convolutional Layer

- Every image is considered as a matrix of pixel values.
- A convolution layer has several filters that perform the convolution operation.
- Slide the filter matrix over the image and compute the dot product to get the convolved feature matrix.

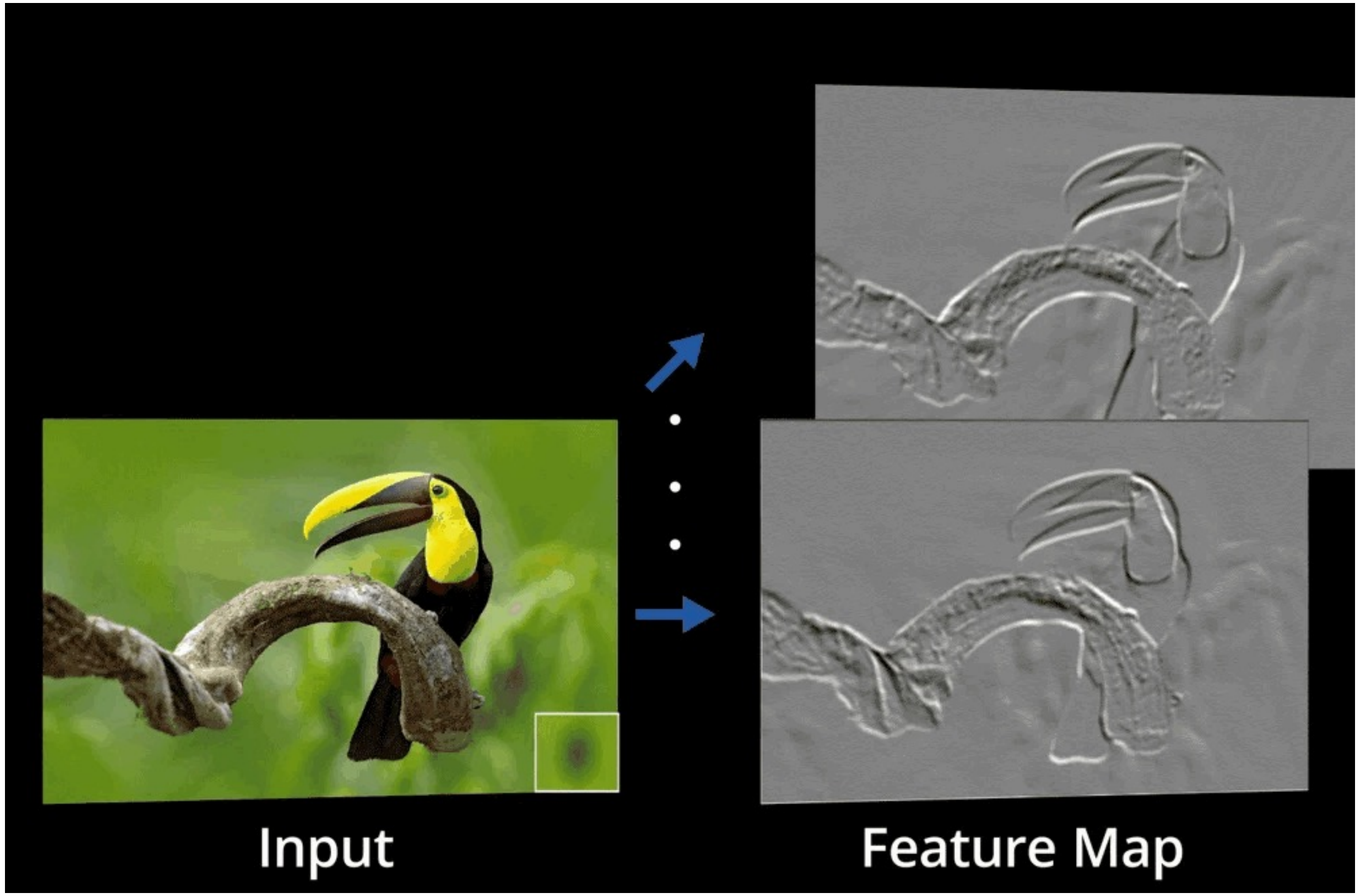


Convolutional Layer



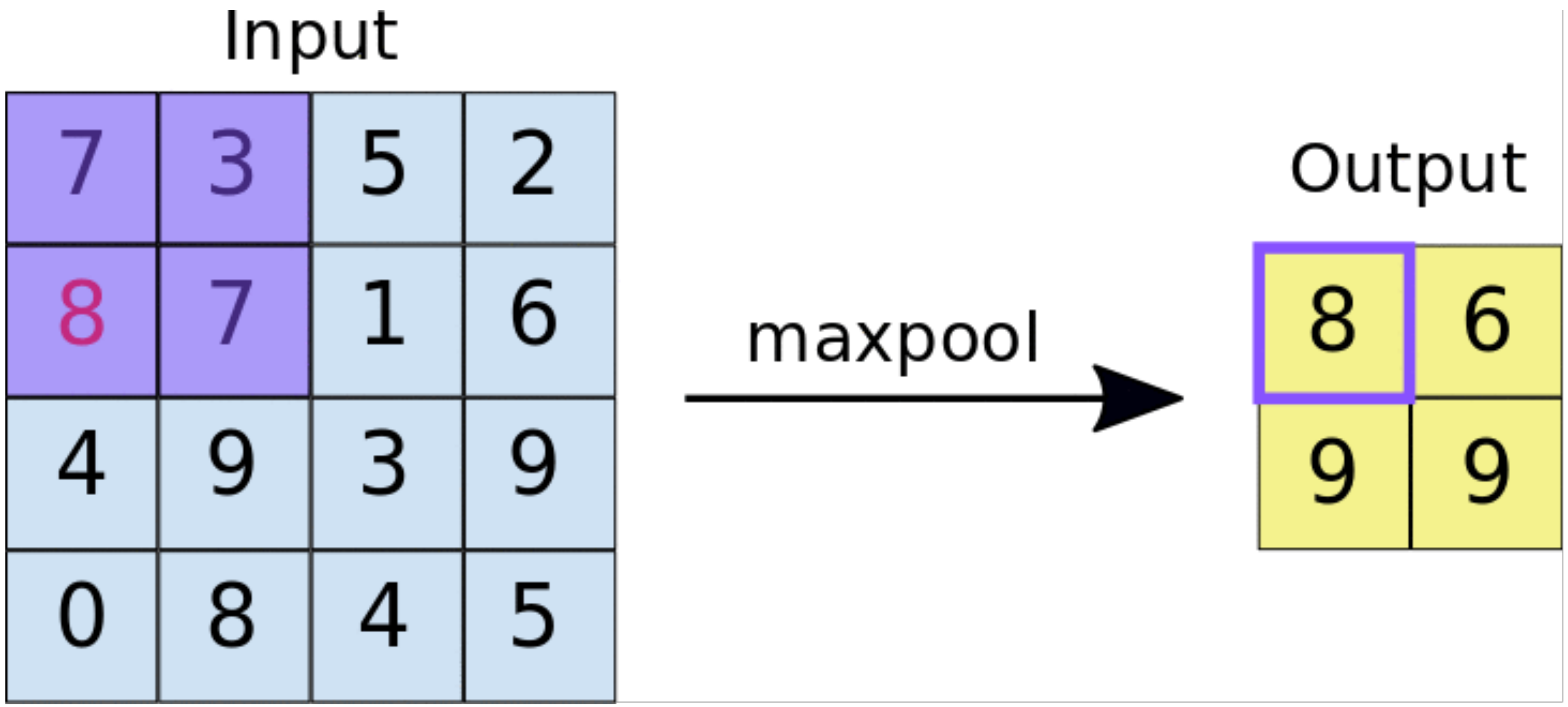
Convolutional Layer

- The result of the convolution operation is called **feature map**.

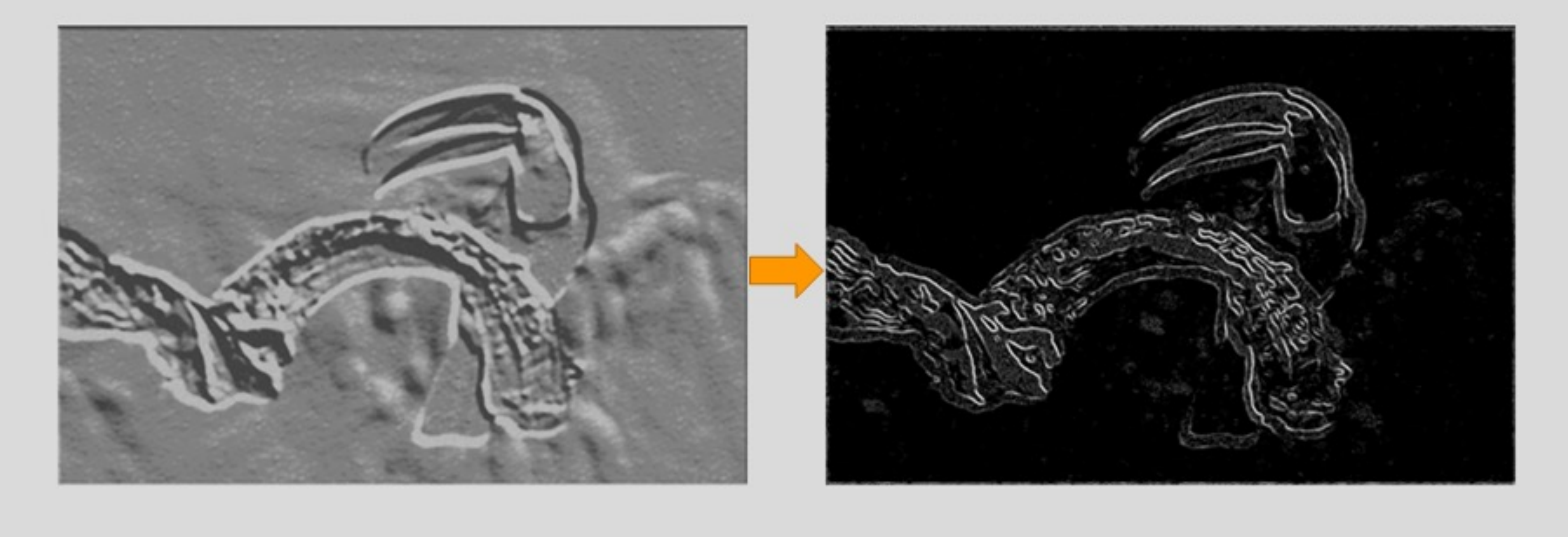


Pooling Layer

- Pooling is a down-sampling operation that reduces the dimensionality of the feature map

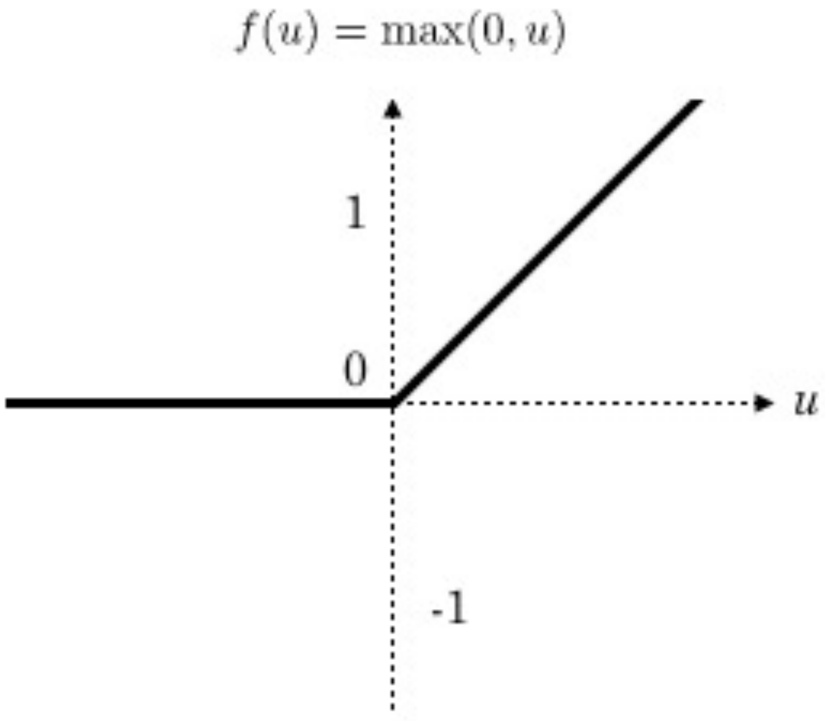


Pooling Layer



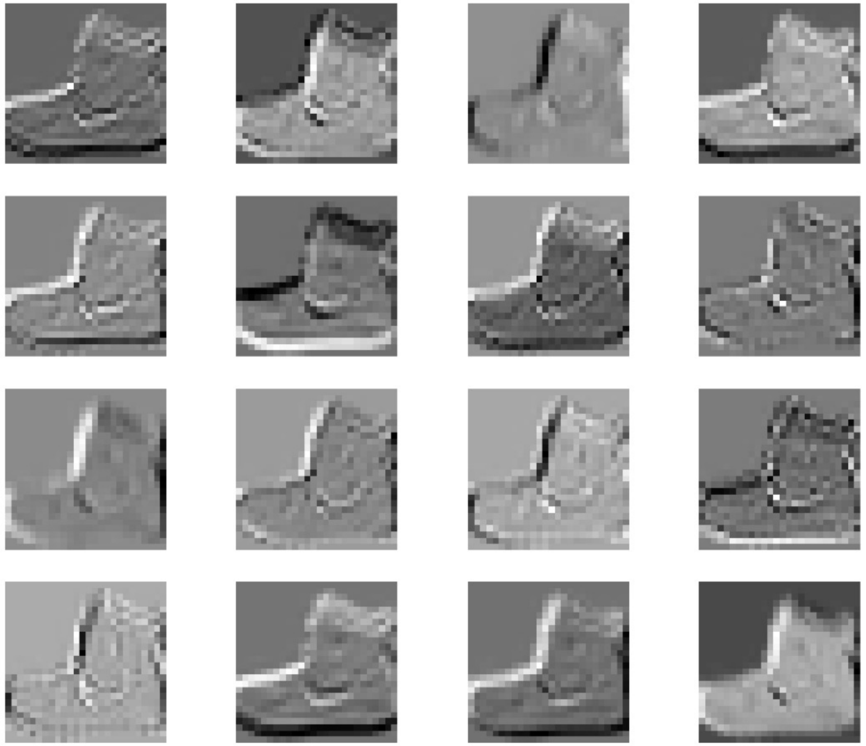
Activation function

- The Rectified Linear Unit (ReLU) Activation Function is commonly used within Convolutional Neural Networks to further process feature maps before they are sent further.
- ReLU introduces non-linearity into our model
- ReLU ensures that only nodes with a positive activation will send their values onwards in the CNN



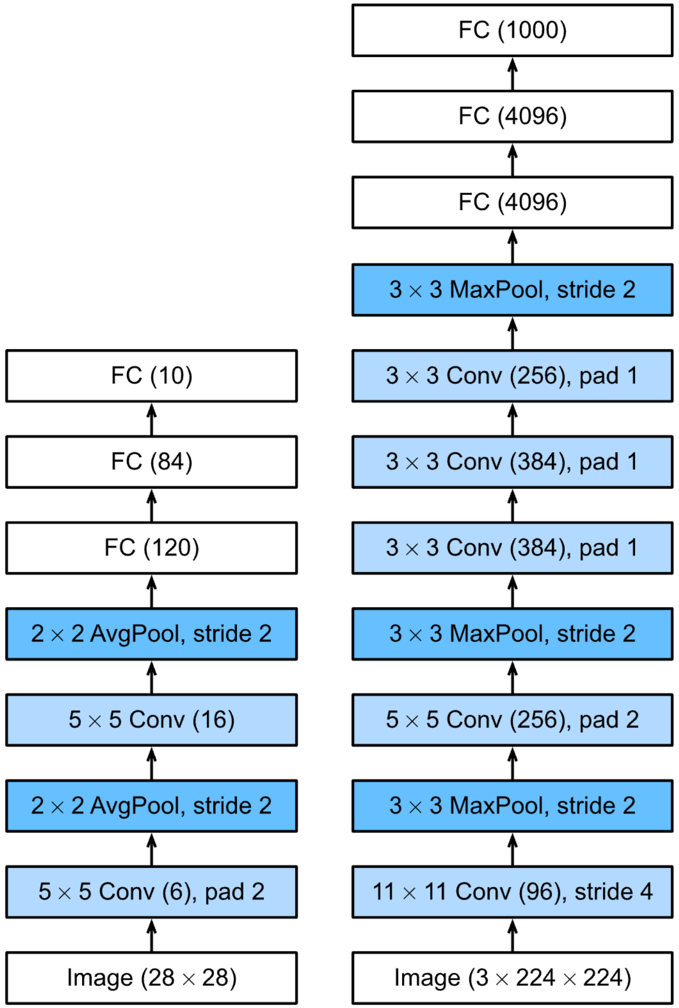
Activation function

Before ReLU Processing



After ReLU Processing

A common CNN: AlexNet



Exercise: let’s implement it!

Step 1: import necessary libraries

```
import torch
import torchvision
from torchvision
import transforms as T
import torch.nn.functional as F
```

Exercise: let’s implement it!

Step 1: import necessary libraries

```
import torch
import torchvision
from torchvision
import transforms as T
import torch.nn.functional as F
```

Step 2: define AlexNet

```
class LeNet(torch.nn.Module):
def __init__(self):
        super(LeNet, self).__init__()
        # TODO
def forward(self, x):
        # TODO
        return x
```

https://blog.paperspace.com/alexnet-pytorch/

Exercise: let’s implement it!

Step 3: define the loss function

```
def get_loss_function():
        loss_function = torch.nn.CrossEntropyLoss()
        return loss_function
```

Step 4: define the optimizer

```
def get_optimizer(net, lr, wd, momentum):
        optimizer = torch.optim.SGD(net.parameters(), lr=lr,
weight_decay=wd, momentum=momentum)
        return optimizer
```

Exercise: let’s implement it!

Step 5: train function

```
train(net,data_loader,optimizer,cost_function, device='cuda:0’):
        samples = 0.
        cumulative_loss = 0.
        cumulative_accuracy = 0.
        net.train() # Strictly needed if network contains layers which has different behaviours between train and test
        for batch_idx, (inputs, targets) in enumerate(data_loader): # Load data into GPU
                inputs = inputs.to(device)
                targets = targets.to(device) # Forward pass
                outputs = net(inputs) # Apply the loss
                loss = loss_function(outputs,targets) # Reset the optimizer
                # Backward pass
                loss.backward()
                # Update parameters
                optimizer.step()
                optimizer.zero_grad()
                samples+=inputs.shape[0]
                cumulative_loss += loss.item()
                _, predicted = outputs.max(1)
                cumulative_accuracy += predicted.eq(targets).sum().item()
        return cumulative_loss/samples, cumulative_accuracy/samples*100
```

Exercise: let’s implement it!

Step 6: test function

```
def test(net, data_loader, cost_function, device='cuda:0’):
        samples = 0.
        cumulative_loss = 0.
        cumulative_accuracy = 0.
        net.eval() # Strictly needed if network contains layers which has different behaviours between train and test
        with torch.no_grad():
                for batch_idx, (inputs, targets) in enumerate(data_loader):
                # Load data into GPU inputs = inputs.to(device)
                targets = targets.to(device)
                # Forward pass
                outputs = net(inputs)
                _, predicted = outputs.max(1)
                cumulative_accuracy += predicted.eq(targets).sum().item()
        return cumulative_loss/samples, cumulative_accuracy/samples*100
```

Exercise: let’s implement it!

Step 7: loading the data

```
def get_data(batch_size, test_batch_size=256):
    # Prepare data transformations and then combine them sequentially
    transform = list() transform.append(T.ToTensor())
    # converts Numpy to Pytorch Tensor
    transform.append(T.Normalize(mean=[0.5], std=[0.5]))
    # Normalizes the Tensors between [-1, 1]
    transform = T.Compose(transform)
    # Composes the above transformations into one.
    # Load data full_training_data = torchvision.datasets.CIFAR10('./data', train=True, transform=transform,
    download=True)
    test_data = torchvision.datasets.CIFAR10('./data', train=False, transform=transform, download=True)
    # Create train and validation splits
    num_samples = len(full_training_data)
    training_samples = int(num_samples*0.5+1)
    validation_samples = num_samples - training_samples
    training_data, validation_data = torch.utils.data.random_split(full_training_data, [training_samples,
    validation_samples])
    # Initialize dataloaders train_loader = torch.utils.data.DataLoader(training_data, batch_size, shuffle=True)
    val_loader = torch.utils.data.DataLoader(validation_data, test_batch_size, shuffle=False)
    test_loader = torch.utils.data.DataLoader(test_data, test_batch_size, shuffle=False)
    return train_loader, val_loader, test_loader
```

Let’s train!

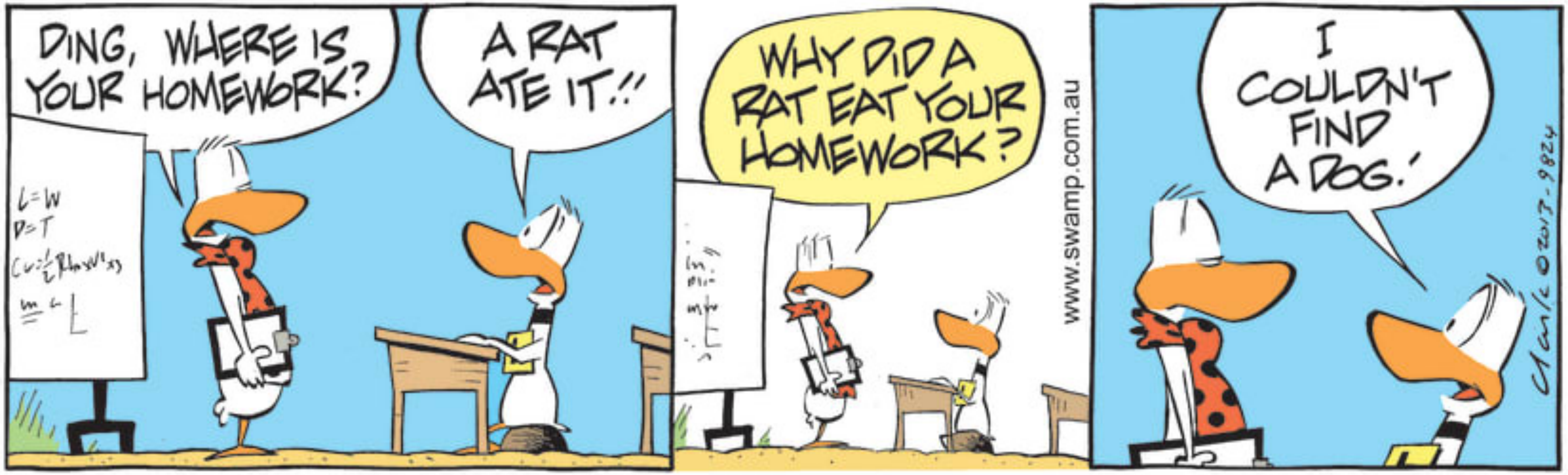
```
''' Input arguments batch_size: Size of a mini-batch device: GPU where you want to train your
network weight_decay: Weight decay co-efficient for regularization of weights momentum: Momentum
for SGD optimizer epochs: Number of epochs for training the network ''’

def main(batch_size=128, device='cuda:0', learning_rate=0.01, weight_decay=0.000001,
momentum=0.9, epochs=50):

    train_loader, val_loader, test_loader = get_data(batch_size)
    # TODO for defining  AlexNet
    optimizer = get_optimizer(net, learning_rate, weight_decay, momentum)
    loss_function = get_loss_function()
```

Continue…

Let’s train!

```
def main(batch_size=128, device='cuda:0', learning_rate=0.01, weight_decay=0.000001,
momentum=0.9, epochs=50):
    ...
    for e in range(epochs):
        train_loss, train_accuracy = train(net, train_loader, optimizer, loss_function)
        val_loss, val_accuracy = test(net, val_loader, loss_function)
        print('Epoch: {:d}'.format(e+1))
        print('\t Training loss {:.5f}, Training accuracy {:.2f}'.format(train_loss,
        train_accuracy))
        print('\t Validation loss {:.5f}, Validation accuracy {:.2f}'.format(val_loss,
        val_accuracy))
        print('-----------------------------------------------------')
        print('After training:')
        train_loss, train_accuracy = test(net, train_loader, loss_function)
        val_loss, val_accuracy = test(net, val_loader, loss_function)
        test_loss, test_accuracy = test(net, test_loader, loss_function)
        print('\t Training loss {:.5f}, Training accuracy {:.2f}'.format(train_loss,
        train_accuracy))
        print('\t Validation loss {:.5f}, Validation accuracy {:.2f}'.format(val_loss,
        val_accuracy))
        print('\t Test loss {:.5f}, Test accuracy {:.2f}'.format(test_loss, test_accuracy))
        print('-----------------------------------------------------')
```

Now it's your turn!



Slide credits:
https://github.com/mancinimassimiliano/DeepLearningLab/blob/master/Lab2/convolutional_neural_networks.ipynb