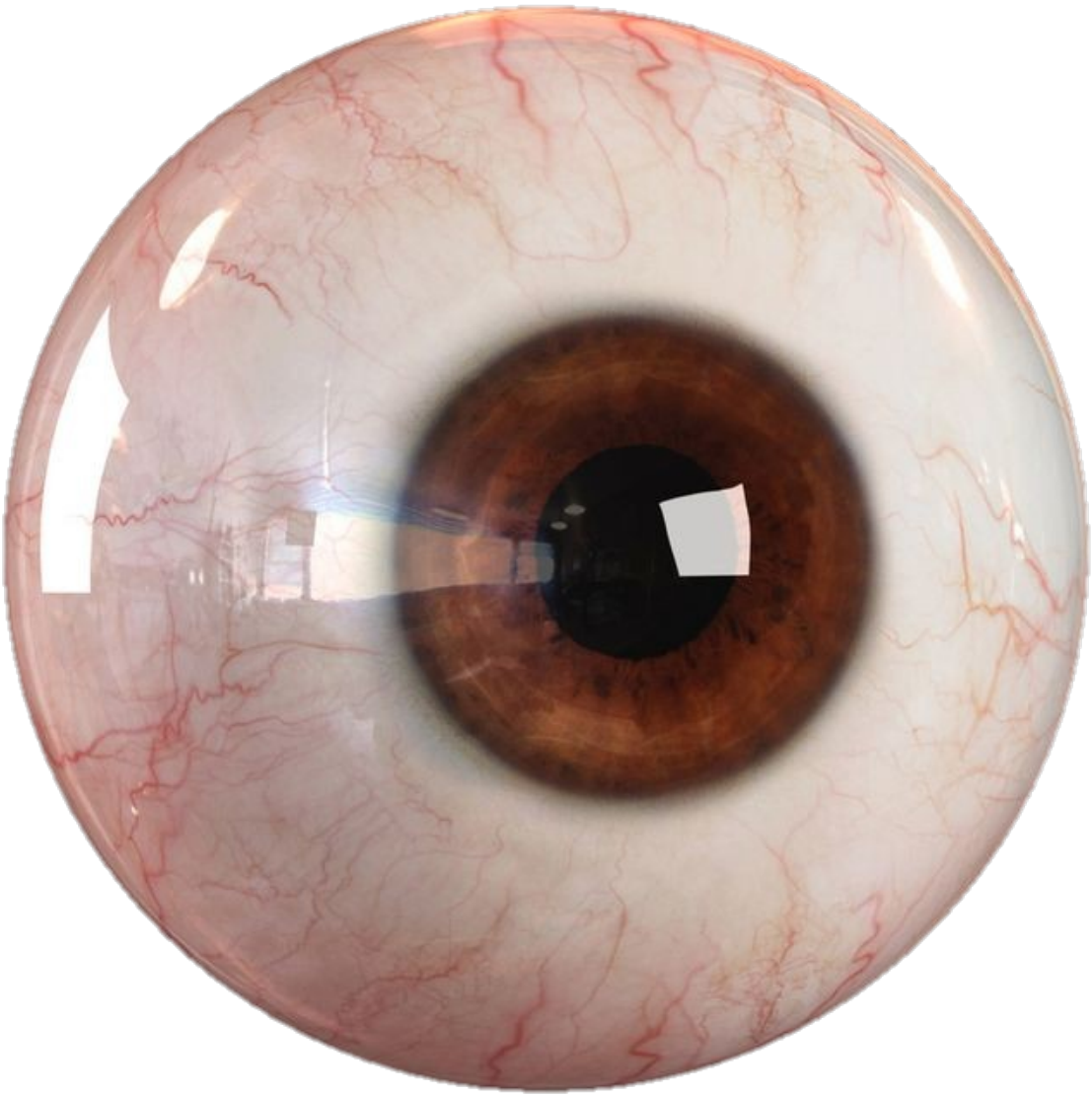
CONVNET REVISITED

A R I H A R A S U D H A N



A STORY OF CONV

Once upon a time in the land of Deep Learning Forest, there lived a wise and ancient tortoise named Conv. Conv was known far and wide for his slow yet steady approach to solving complex problems. He had an insatiable curiosity for learning patterns and recognizing shapes in the environment around him.

One day, as he was wandering through the forest, Conv came across a beautiful collection of images scattered on the forest floor. These images had various objects, creatures, and landscapes, and they seemed to be filled

with fascinating patterns that intrigued Conv's inquisitive mind. Curious to learn more about the images, Conv decided to embark on a journey to uncover the hidden secrets within them. He knew that his wisdom and slow pace would be a perfect match for this daunting task. As he took his first step, Conv encountered a group of young creatures known as

Filters. These Filters were specialized in detecting specific patterns in the images, such as edges, colors, and textures. They could slide across the images, peeking at small portions at a time, and signaling their findings to Conv. With the help of the Filters, Conv learned about the low-level features of the images, the basic building blocks that formed the

objects and shapes. These low-level features were fascinating, but Conv knew he needed to go deeper into the forest of knowledge. As Conv delved further, he met the powerful Convolution Layer, a magnificent structure with a vast array of Filters. The Convolution Layer could analyze the entire image, using multiple Filters to capture different patterns

simultaneously. The Filters' collective wisdom helped Conv recognize more complex shapes and objects within the images. But the forest held even more secrets. Conv stumbled upon a mysterious creature called Activation Function, a mystical gatekeeper that had the power to determine which features should be emphasized and which should

be discarded. Activation Function guided Conv in his exploration, making sure he stayed on the right path. In the heart of the forest, Conv encountered the Pooling Ponds, tranquil places where he could rest and reflect. The Pooling Ponds had a unique ability - they could reduce the size of the feature maps, making the images more manageable. This allowed

Conv to focus on the most important aspects while discarding the less crucial details. As he moved forward, Conv felt he was getting closer to the ultimate truth hidden within the images. He arrived at the Fully Connected Forest, a place where the features he had learned so far were combined and analyzed, leading him to make decisions and

predictions about the images' contents. At the end of his long and exciting journey, Conv realized that he had achieved something remarkable. By combining the knowledge gained from Filters, Convolution Layers, Activation Functions, and Pooling Ponds, he had constructed a remarkable structure - a Convolutional Neural Network or CNN.

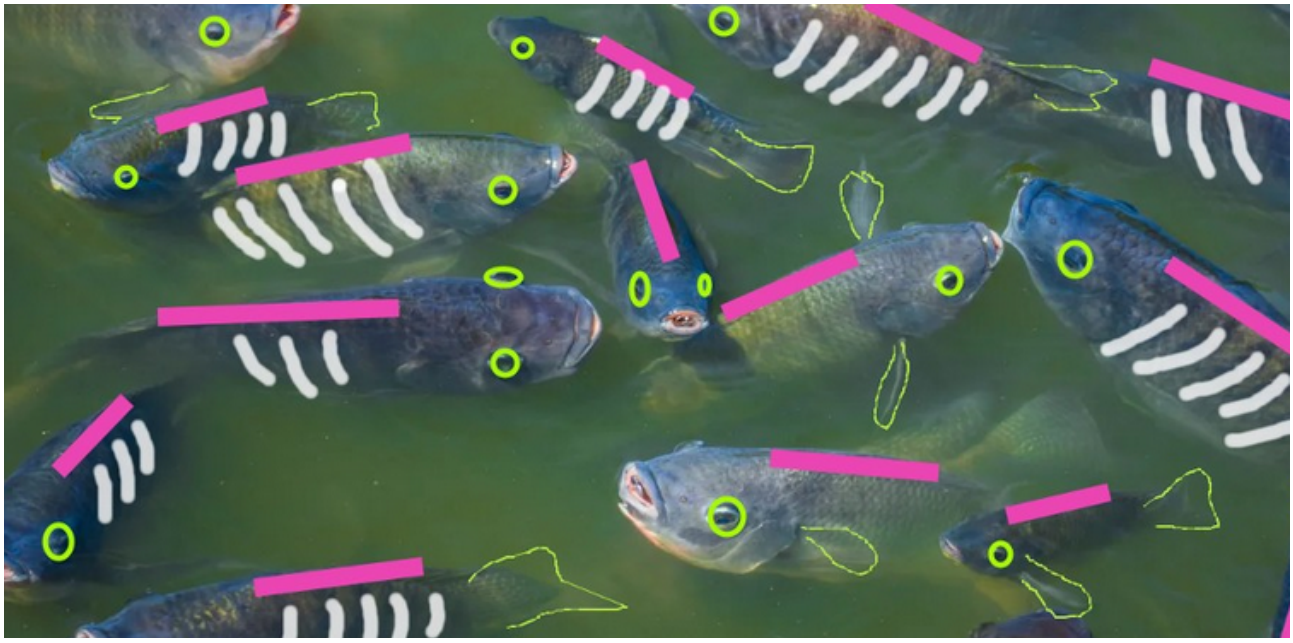
With his newfound wisdom, Conv could now analyze images, recognize objects, and even predict the class of the creatures or objects depicted in the images. He knew that his slow and methodical approach had led him to a deeper understanding of the world around him, and he was ready to share his knowledge with others.

From that day on, Conv continued his journey through the vast Deep Learning Forest, helping others and inspiring many to embrace the power of CNNs—the slow but mighty tortoises of the machine learning world. And so, the tale of Conv, the wise tortoise of CNN, spread far and wide, inspiring generations of learners to seek wisdom in the patterns of the world.

INTRODUCTION

Convolutional Neural Network (CNN) is a type of artificial neural network inspired by how the human brain processes visual information. It is mainly used for image recognition tasks, like identifying objects in pictures or videos. Imagine it as a detective for images: The CNN breaks down an image into smaller pieces and analyzes them

step by step. It looks for simple patterns, like edges or corners, in the first layers. Then, it combines these patterns into more complex features in deeper layers, like shapes and textures. Finally, it uses all this information to make a prediction about what's in the image. CNNs are great at understanding and classifying visual data.



oreochromis niloticus /
oreochromis tilapia

APPLICATIONS

Convolutional Neural Networks (CNNs) have proven to be incredibly versatile and powerful in various applications within the field of computer vision. Let’s discuss some exciting projects that we can achieve using CNNs :

@ Image Classification
@ Object Detection
@ Facial Recognition
@ Image Segmentation
@ Style Transfer

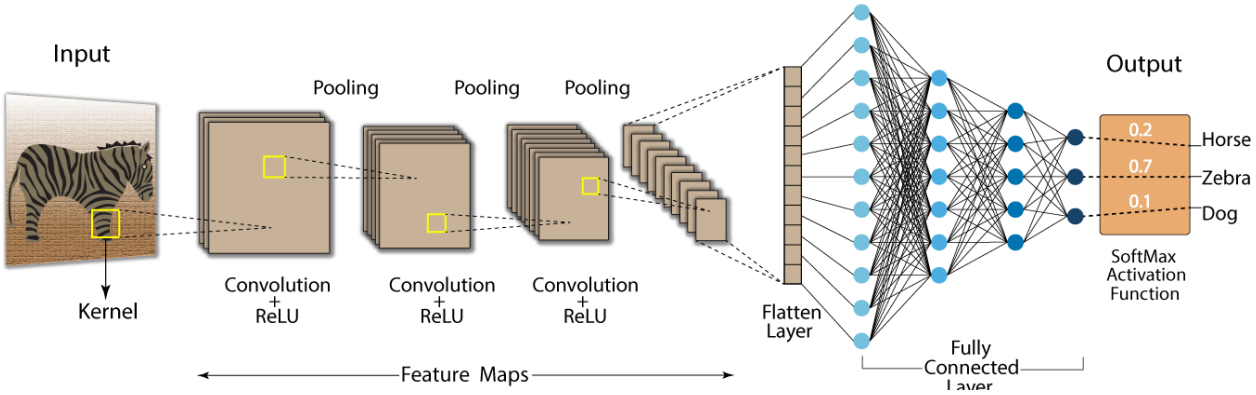
THE APPROACH

The process of training and using a Convolutional Neural Network (CNN) typically involves the following steps:

Data Collection : Gather a large dataset of labeled images relevant to our task. This dataset will be used to train and test the CNN.

Data Preprocessing : Prepare the data for training.

Model Architecture : Design the CNN's architecture, which consists of multiple layers.



The key layers in a CNN are:
Convolutional Layers : These layers apply filters (kernels) to the input images to detect features like edges, corners, and textures.

Activation Layers : After each convolution, apply an activation function (like ReLU) to introduce non-linearity.

Pooling Layers : These layers downsample the feature maps, reducing the spatial dimensions and retaining important information.

Fully Connected Layers : Flatten the features and connect them to a neural network.

Model Compilation : Specify the loss function (e.g., cross-entropy for classification), an optimizer (e.g., Adam or SGD), and evaluation metrics (e.g., accuracy) to optimize and measure the model's performance during training.

Model Training : Feed the training data into the CNN, and let it adjust its parameters (weights and biases) through backpropagation.

Model Evaluation : Once the training is complete, evaluate the model's performance on a separate test dataset to assess its accuracy and generalization ability.

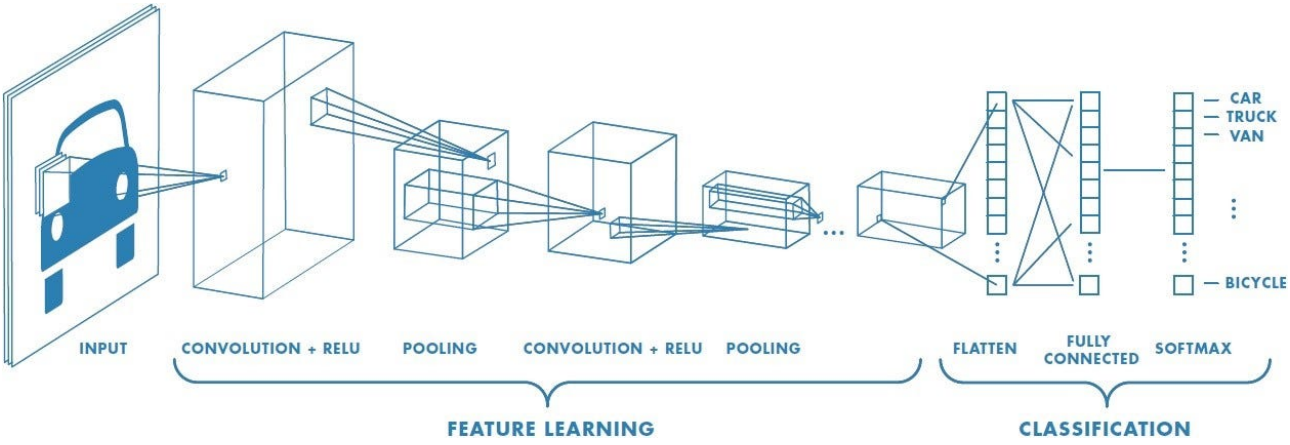
Hyperparameter Tuning : Experiment with different hyperparameters, like learning rate, batch size, and number of layers, to optimize the CNN's performance.

Prediction : Use the trained CNN to make predictions on new, unseen images. The model will classify or detect objects based on what it has learned during training.

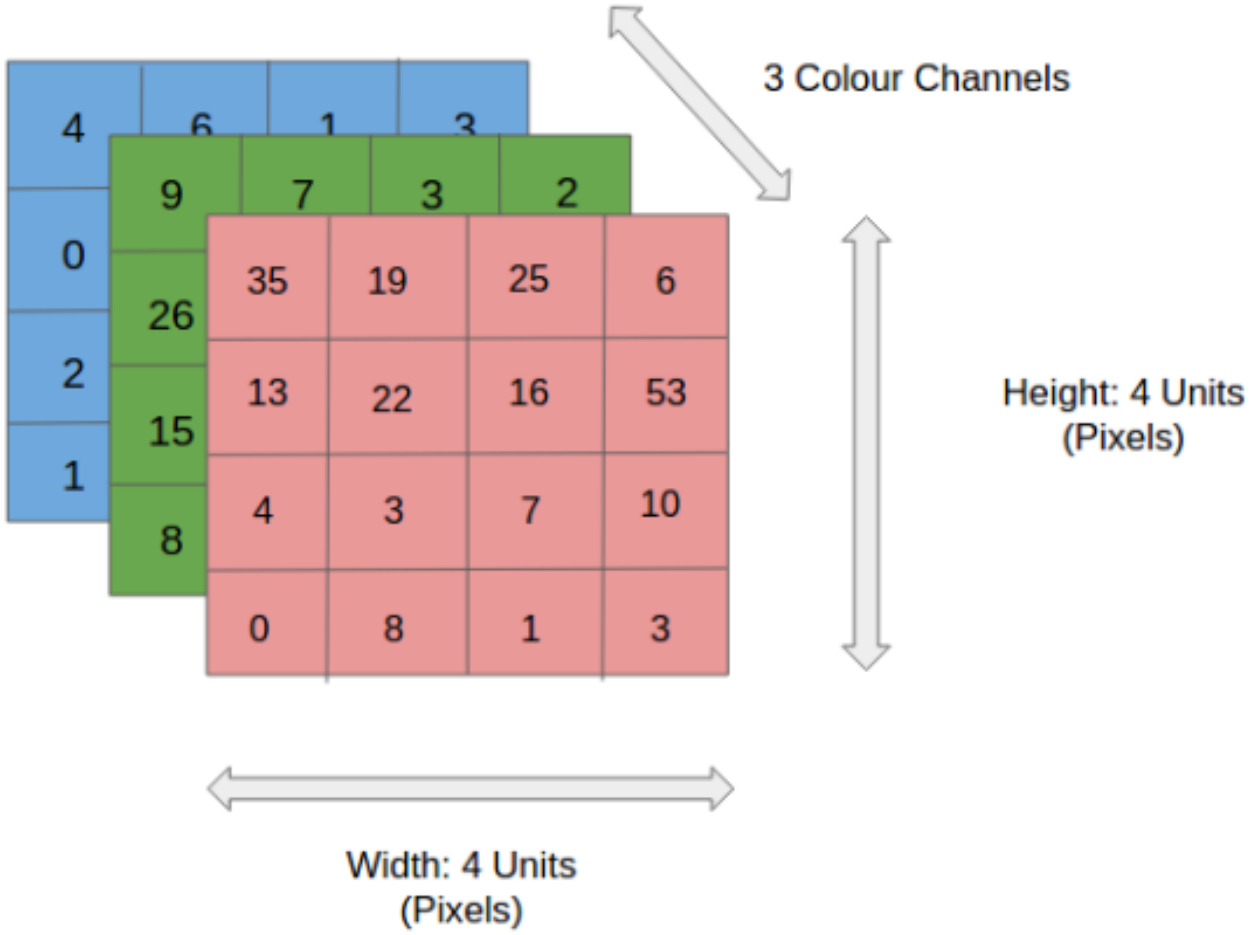
For some people, these words maybe quite easy to get through. But, for people like me, it won’t sound nice. So, just keep these topics in mind. We’ll explore each of them in this book.

EXPLAINED!

The architecture of CNN performs a better fitting to the image dataset due to the reduction in the number of parameters involved and the reusability of weights. The network can be trained to understand the sophistication of the image better.



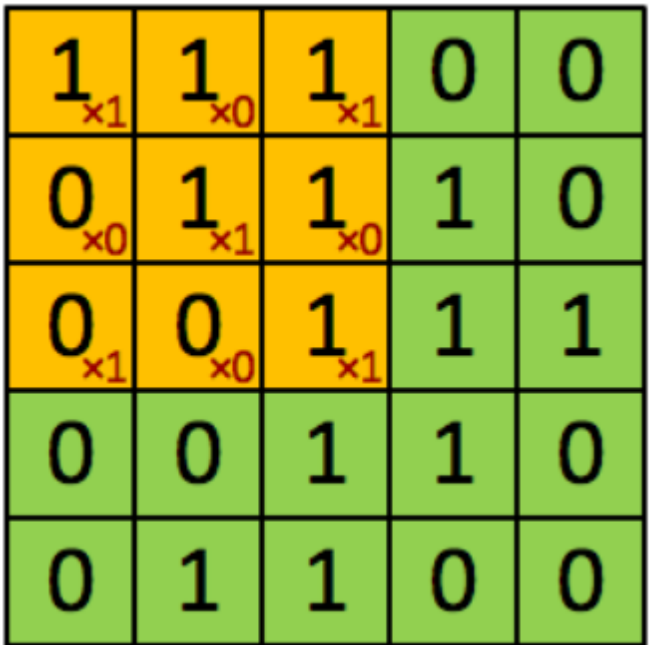
INPUT IMAGE

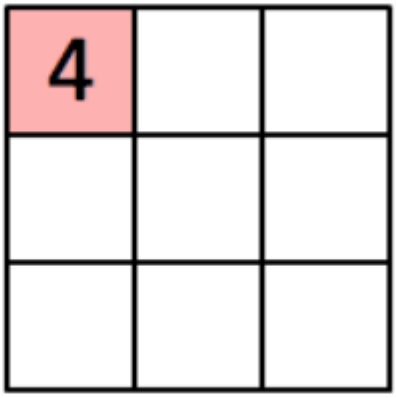


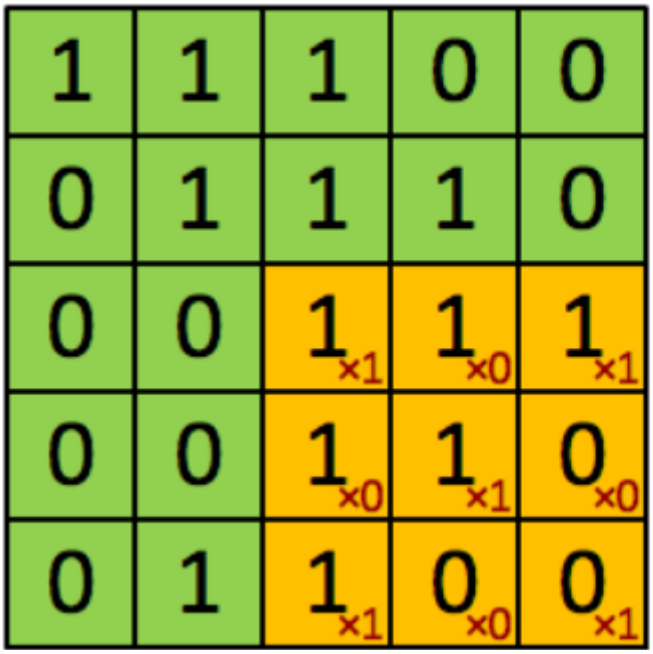
We have an RGB image that has been separated by its three color channels/planes (Red, Green, Blue) with the Height and Width size mentioned.

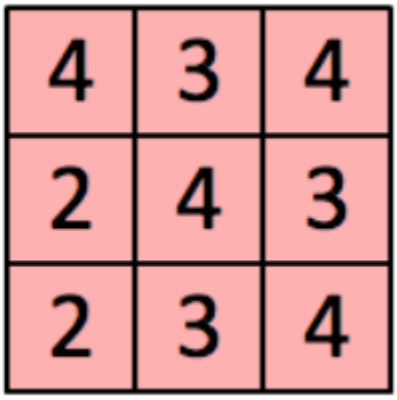
CONV LAYER

We convolve the input image with a kernel which captures patterns.

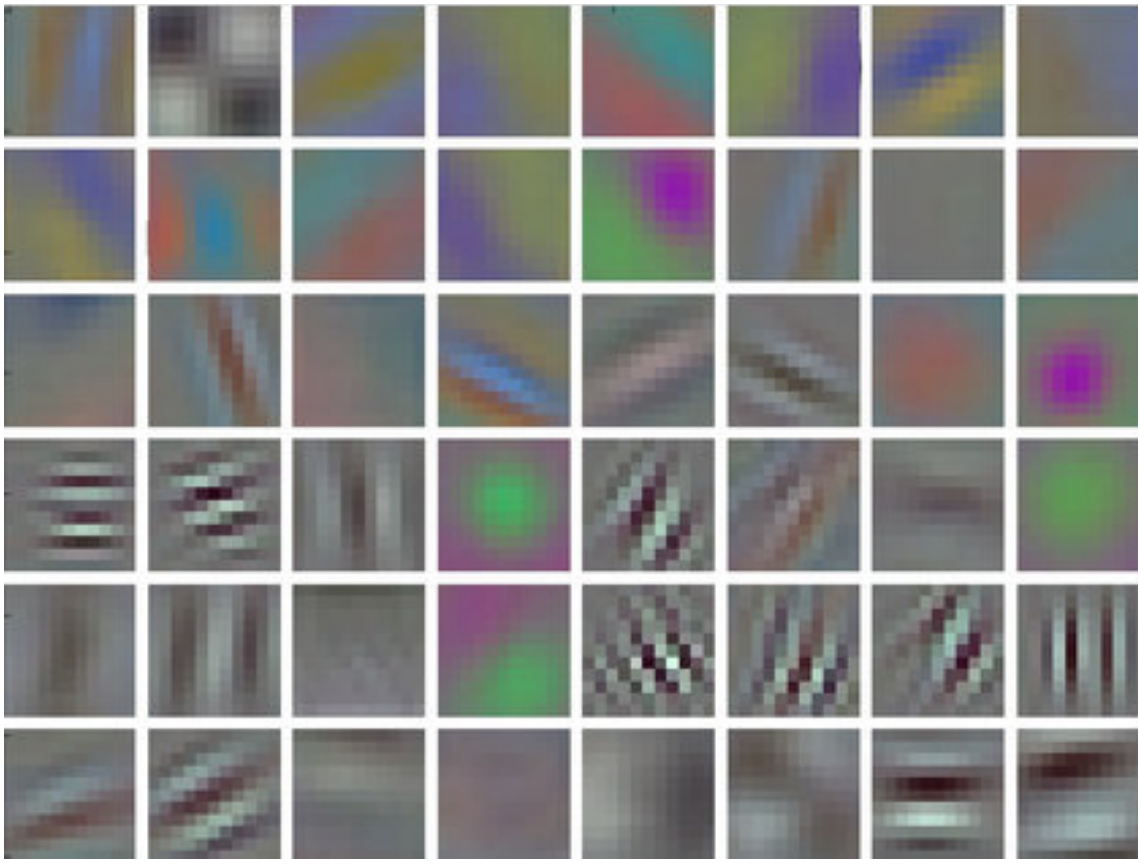


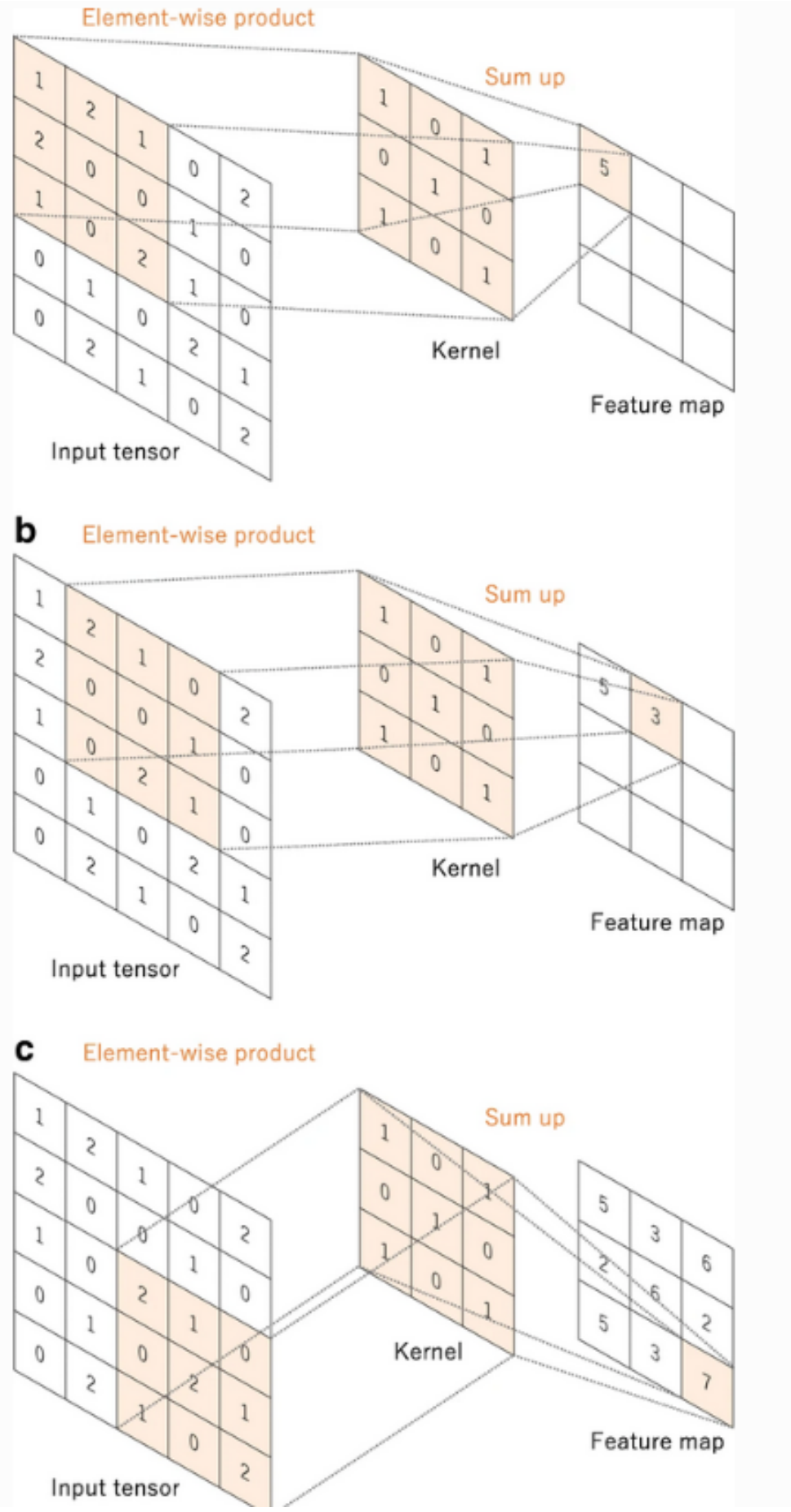






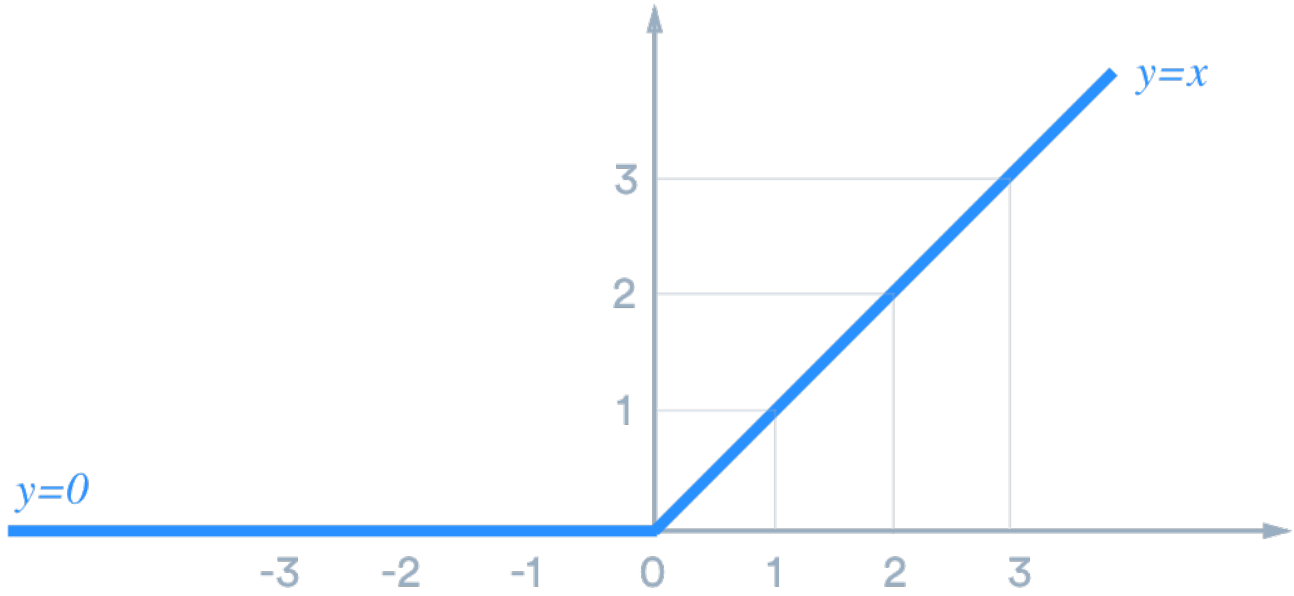
There are different kernels endowed with the ability of capturing various patterns of the given input. Some kernels may capture the patterns like horizontal lines. Some may go for vertical lines. Some can capture circular patterns and so on.




Element-wise product
Sum up
Kernel
Feature map
Input tensor
b
Element-wise product
Sum up
Kernel
Feature map
Input tensor
c
Element-wise product
Sum up
Kernel
Feature map
Input tensor

NON-LINEARITY

The outputs of a linear operation such as convolution are then passed through a nonlinear activation function. The most common nonlinear activation function used presently is the rectified linear unit (ReLU), which simply computes the function: $f(x) = \max(0, x)$.



POOLING LAYER

Pooling is an essential operation in ConvNets, primarily used for downsampling or reducing the spatial dimensions of feature maps while retaining important information. Pooling layers help to make the network more robust, reduce computational complexity, and increase its ability to generalize to new data. The main reasons for using pooling in ConvNets :

Dimensionality Reduction :

Convolutional layers can result in large feature maps, especially as we stack multiple layers. Pooling layers help reduce the spatial dimensions of these feature maps, leading to a more efficient representation. Smaller feature maps decrease the number of parameters and computations in subsequent layers.

Translation Invariance :

Pooling allows the network to be partially invariant to small translations in the input data. Since pooling aggregates information from local regions, slight shifts in the input can result in the same pooled output. This property makes the model more robust to small changes in the input data.

Feature Localization :

By reducing spatial dimensions, pooling helps to focus on the most important features. The pooling operation selects the most dominant features within each region, which can be helpful in identifying essential patterns and reducing the impact of noisy or less relevant information.

Two common types of pooling operations used in ConvNets are:

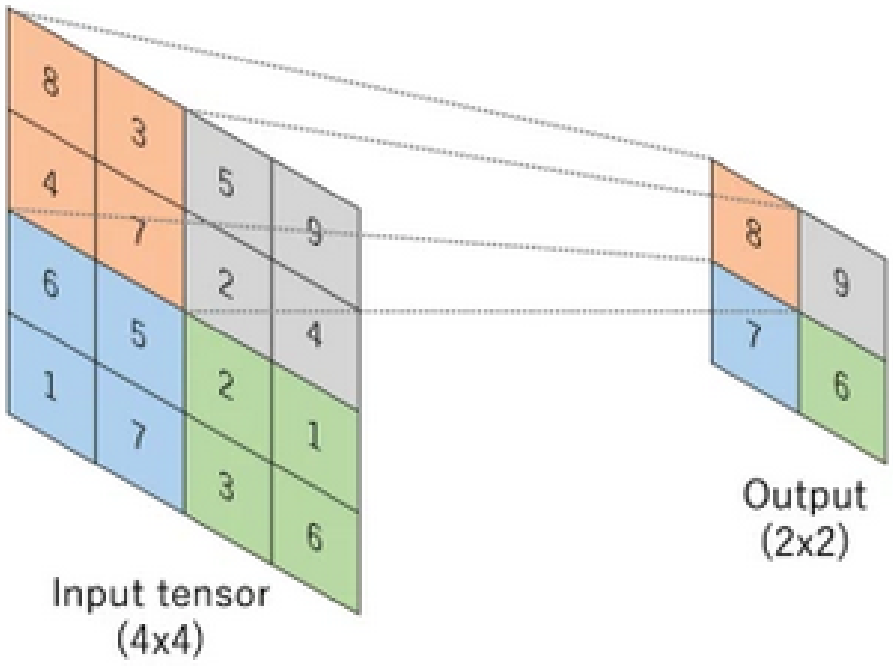
Max-Pooling :

In max pooling, the operation selects the maximum value within each pooling region (usually a small window). This captures the most salient feature within that region.

Average-Pooling :

In average pooling, the operation takes the average value within each pooling region. This provides a

smoother representation of the features and is more suitable for tasks where precise localization is not crucial. Pooling is typically applied after convolutional layers and can be followed by additional convolutional layers and fully connected layers to build deeper networks.



FULLY CONNECTED

The output feature maps of the final convolution or pooling layer is typically flattened, i.e., transformed into a one-dimensional (1D) array of numbers (or vector), and connected to one or more fully connected layers, also known as dense layers, in which every input is connected to every output by a learnable weight. Once the features extracted by the convolution layers and

downsampled by the pooling layers are created, they are mapped by a subset of fully connected layers to the final outputs of the network, such as the probabilities for each class in classification tasks. The final fully connected layer typically has the same number of output nodes as the number of classes. Each fully connected layer is followed by a nonlinear function, such as ReLU, as described above.

CLASSIFICATION

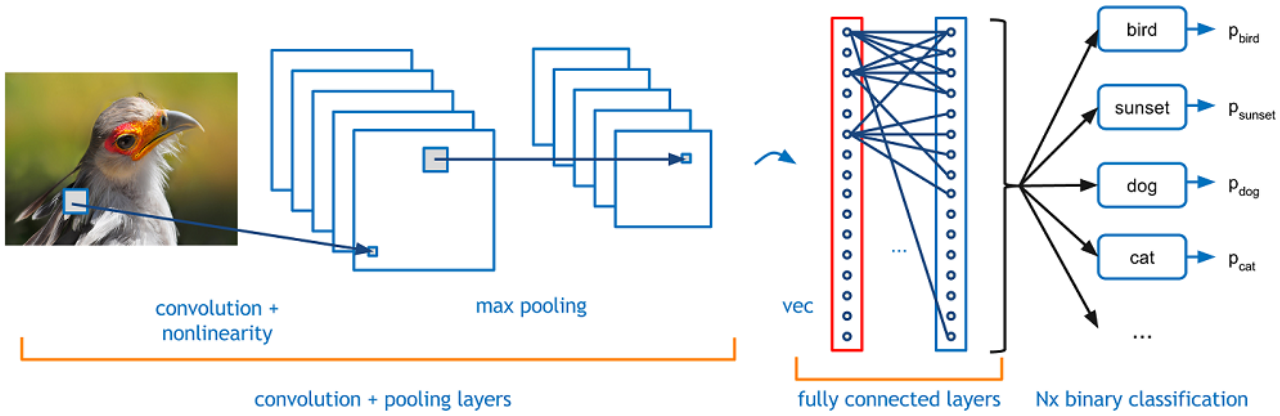
At last, a classification head can be used. We can use an activation function such as SoftMax for multiclass classification.

TRAIN

Training a CNN is the process of finding kernels in convolution layers and weights in fully connected layers which minimize differences between output

predictions and given ground truth labels on a training dataset. Backpropagation algorithm is the method commonly used for training neural networks where loss function and gradient descent optimization algorithm play essential roles. A model performance under particular kernels and weights is calculated by a loss function through forward propagation

on a training dataset, and learnable parameters, namely kernels and weights, are updated according to the loss value through an optimization algorithm called backpropagation and gradient descent.



ON MNIST

IMPORT LIBRARIES

```
import torch
from torchvision import datasets
import torchvision
from torchvision.transforms import ToTensor
from torch.utils.data import DataLoader
import numpy
```

LOAD TRAIN & TEST DATA

```
train_data = datasets.MNIST(
    root = 'data',
    train = True,
    transform = ToTensor(),
    download = True,
)
test_data = datasets.MNIST(
    root = 'data',
    train = False,
    transform = ToTensor()
)
```

DATA LOADERS

```
loaders = {
    'train' : DataLoader(train_data,
                         batch_size=100,
                         shuffle=True,
                         num_workers=1),
    'test'  : DataLoader(test_data,
                         batch_size=100,
                         shuffle=True,
                         num_workers=1), }
```

THE CNN

```
import torch.nn as nn

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels=1,
                out_channels=16,
                kernel_size=5,
                stride=1,
                padding=2,
            ),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, 5, 1, 2),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        self.out = nn.Linear(32 * 7 * 7, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)
        output = self.out(x)
        return output, x
```

LOSS FUNCTION & ACTIVATION FUNCTION

```
from torch import optim
cnn = CNN()
optimizer = optim.Adam(cnn.parameters(), lr = 0.01)
loss_func = nn.CrossEntropyLoss()
```

TRAIN THE NETWORK

```
from torch.autograd import Variable
num_epochs = 10
def train(num_epochs, cnn, loaders):
    cnn.train()
    total_step = len(loaders['train'])
    for epoch in range(num_epochs):
        for i, (images, labels) in enumerate(loaders['train']):
            b_x = Variable(images)   # batch x
            b_y = Variable(labels)   # batch y
            output = cnn(b_x)[0]
            loss = loss_func(output, b_y)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            if (i+1) % 100 == 0:
                print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'.format(epoch + 1, num_epochs, i + 1, total_ste

train(num_epochs, cnn, loaders)
```

LET'S TEST

```
with torch.no_grad():
    correct = 0
    total = 0
    accuracy = 0
    for data in loaders['test']:
        images,labels = data
        test_output, last_layer = cnn(images)
        pred_y = torch.max(test_output, 1)[1].data.squeeze()
        accuracy += (pred_y == labels).sum().item() / float(labels.size(0))
    print(accuracy/len(loaders['test']))
```

```
0.9858999999999999
```

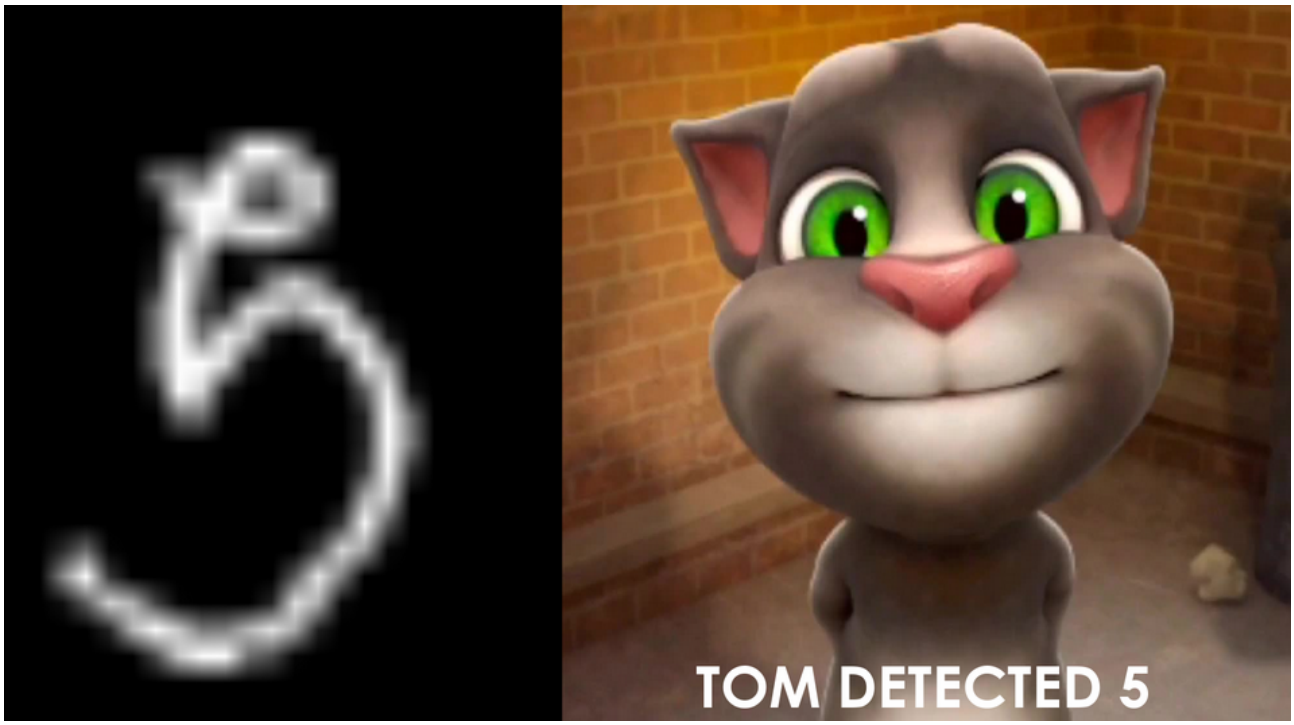
PREDICTIONS

```
sample = next(iter(loaders['test']))
imgs, lbls = sample
actual_number = lbls[:10].numpy()
test_output, last_layer = cnn(imgs[:10])
pred_y = torch.max(test_output, 1)[1].data.numpy().squeeze()
print(f'Prediction number: {pred_y}')
print(f'Actual number: {actual_number}')
```

```
Prediction number: [1 1 9 3 2 5 1 9 7 1]
Actual number: [1 1 9 3 2 5 1 9 7 1]
```

We can make it a bit interesting by calling Mr.TOM to tell what the digit is.

CHECK : https://github.com/arihara-sudhan/MNIST-Tom-NeuralNetwork



MERCI