Convolutional Neural Networks

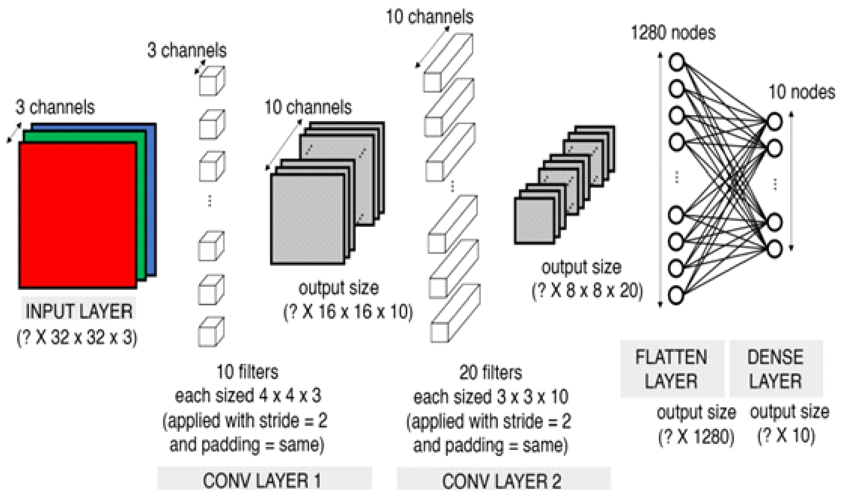
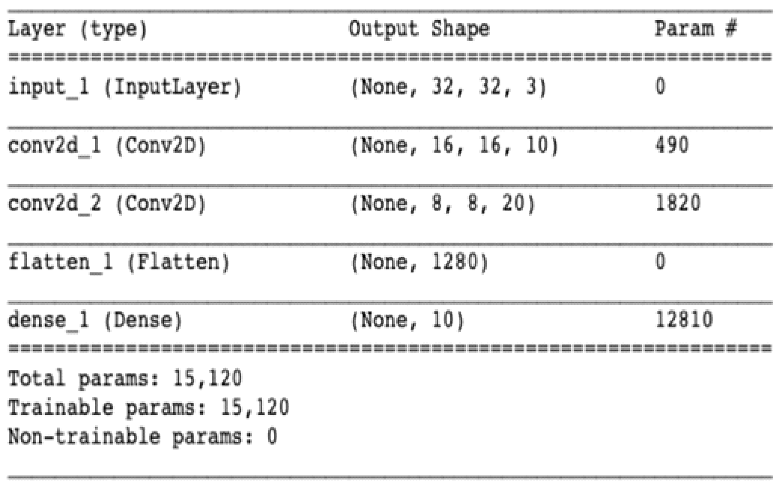


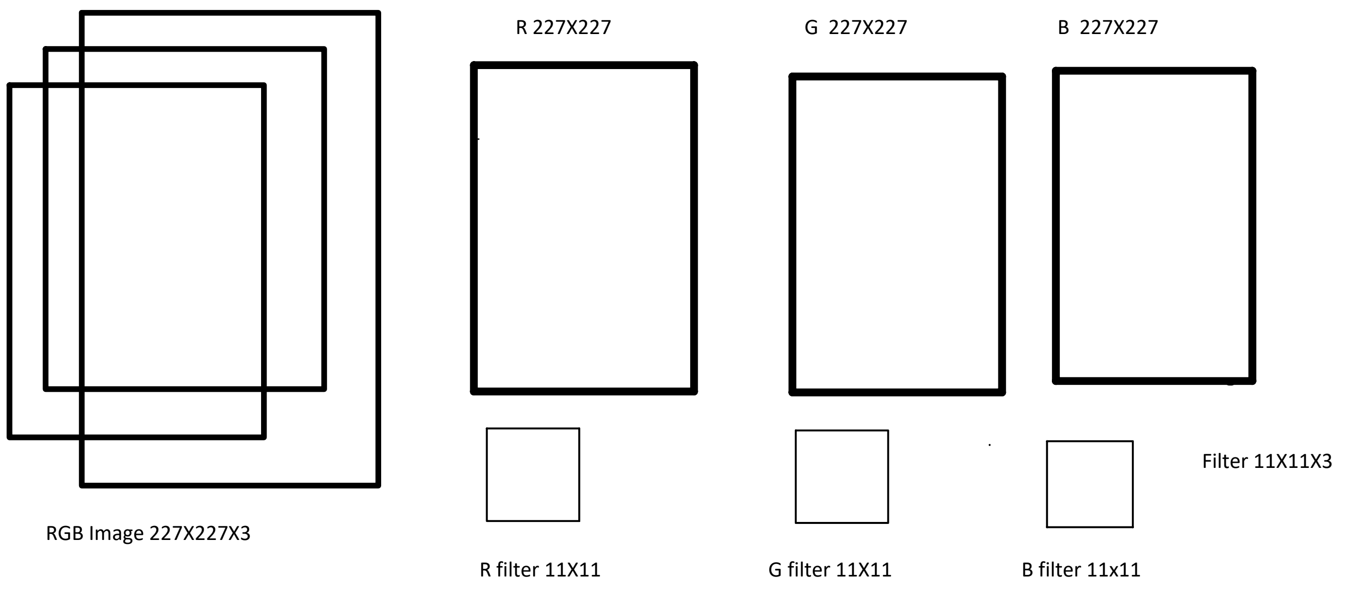
Figure 2-13. A diagram of a convolutional neural network

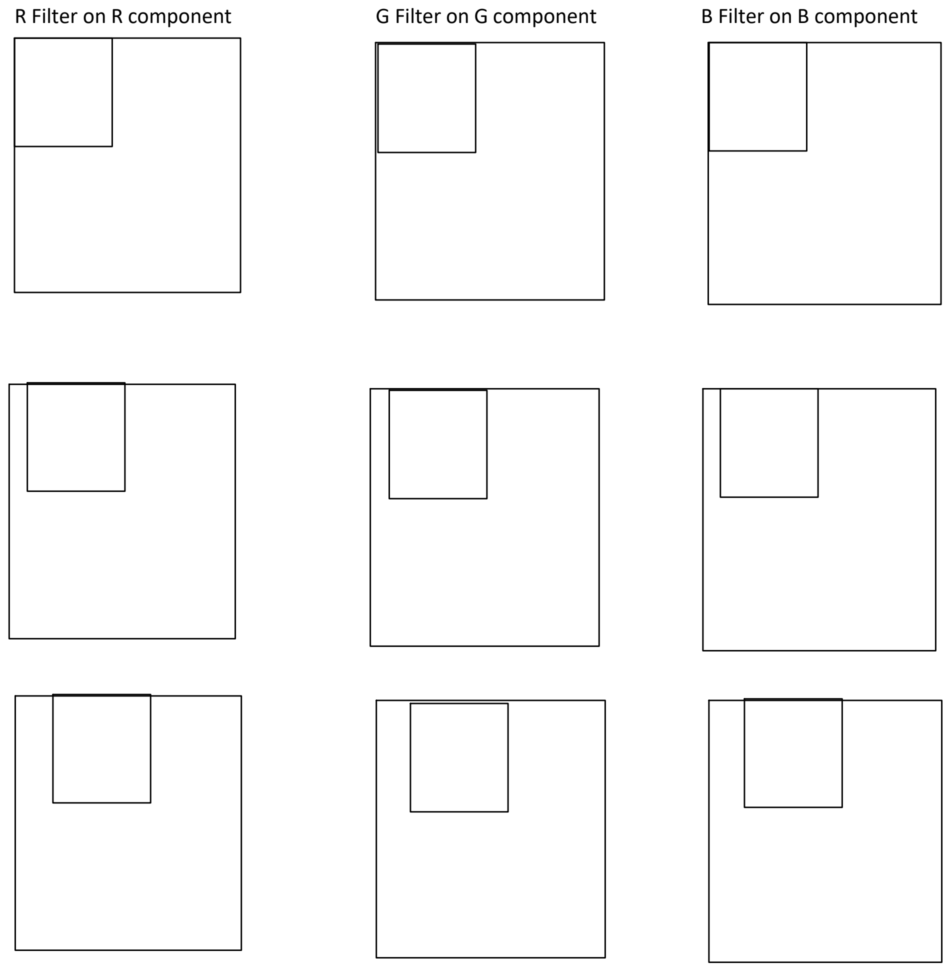
We can use the `model.summary()` method to see the shape of the tensor as it passes through the network (Figure 2-14).



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 32, 32, 3)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 16, 16, 10)        490
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 8, 8, 20)          1820
_________________________________________________________________
flatten_1 (Flatten)          (None, 1280)              0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                12810
=================================================================
Total params: 15,120
Trainable params: 15,120
Non-trainable params: 0
_________________________________________________________________
```

Figure 2-14. A convolutional neural network summary



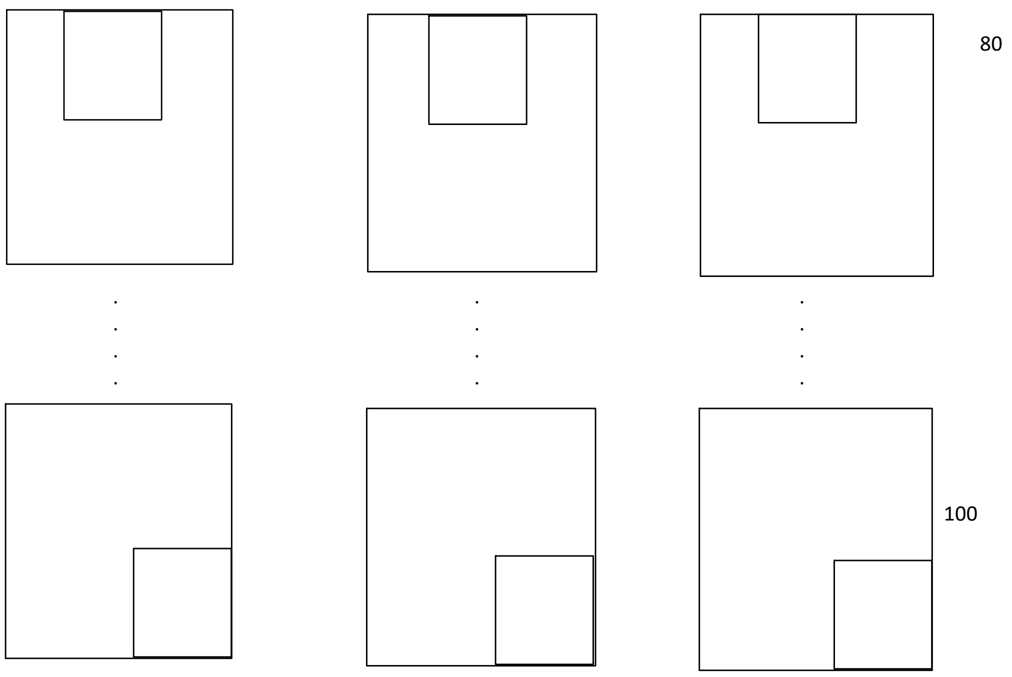


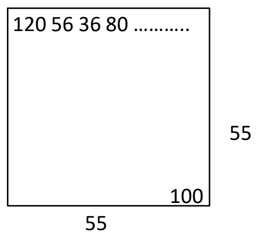
R1F1+R2F2+R3F3+.........+R121F121+
G1F1+G2F2+G3F3+........+G121F121+
B1F1+B2F2+B3F3+..........+B121F121

120

56

36





Output dimension

= (W-F+2P)/s + 1

= 55

Here W = 227, F = 11, P = 0 and S = 4

If we use 90 filters of size 11X11X3 we will produce 90 outputs each having size 55X55

Each convolution operation is performed by a neuron. It is connected with 11X11X3 = 363 R, G, B values through links having weights equal to the filter values. The initial weight values are set randomly and learned during training through gradient descent approach.

To cover the whole image we need 55 convolution operations horizontally and 55 operations vertically by sliding the filters across the image horizontally and vertically. Thus the output is a single feature map of dimension 55X55.

Each separate filter is supposed to extract a unique kind of feature. That is why all the 55X55 image segments are convolved with the same filter having same weights. Thus to produce a feature map using a 11X11X3 filter we need 55X55 neurons having a single set of weights 11X11X3.

If we use 90 filters each having dimension 11X11X3, we will produce 90 different feature maps each having size 55X55. Total number of neuros will be 55X55X90 and the total number of shared weights are 90X11X11X3 and 90 biases

In general feature map generated from convolution operations has a size less than the original input. In order to keep the size equal, the input image/feature map is padded with zeros around it. It is called zero padding. Sometimes the outer most rows and columns are copied around more accuracy.

Image = W1XH1XD1, Filter = FXFXD1, Number of filters = K, Stride = S, P = Number of Zero padding

Output = W2XH2XD2 where W2 = (W1-F+2P)/S + 1, H2 = (H1-F+2P)/S + 1 and D2 = K

To apply K filers we need FXFXD1 weights per filter, a total KXFXFXD1 weights and K biases

Convolution operation as matrix multiplication

Image : 11x11x3 is the number of pixels convolved with the filter as the filter slides through the image. The image can be divided into 55x55 = 3,025 blocks where each block consists of 11x11x3 = 363 pixels. So the input image 227x227x3 can be represented by 363x3025 image matrix where each column represents a convolution block.

Each filter can be represented by a row of 11x11x3 = 363 values (weights) and we have 90 such filters. So the filters can be represented by 90x363 filter matrix

Now Output feature maps = Filter matrix x Image matrix = (90x363)x(363x3025)
= 90x3025

Pooling Layer

$$\left[\begin{array}{cc|cc} 1 & 8 & 2 & 6 \\ 2 & 6 & 0 & 9 \\ \hline 3 & 5 & 7 & 8 \\ 0 & 1 & 2 & 0 \end{array}\right] \qquad \begin{bmatrix} 8 & 9 \\ 5 & 8 \end{bmatrix}$$

Max Pooling (2X2 and S = 2). Pooling halves the input feature map. The discarded values are marked and all the backward connections through these discarded values are ignored while training through backpropagation is done. There is another variation of pooling which is called average pooling. Sometimes pooling is avoided by applying bigger strides

Normalization Layer

Normalization in Convolutional Neural Networks (CNNs) is important for several reasons:

1. **Stability During Training:**
   - Normalization helps in stabilizing the learning process by maintaining a consistent scale of inputs throughout the network, reducing the possibility of fluctuations in parameter updates.
2. **Improved Convergence:**
   - By ensuring that inputs to each layer have a zero mean and unit variance (or similar targets), normalization can significantly speed up the convergence of the network during training, enabling faster learning.
3. **Reduced Sensitivity to Initialization:**
   - Normalization can lessen the sensitivity of the network to the initialization of weights. This means

models are less reliant on choosing "perfect" initial weights and are more likely to converge to good solutions starting from a broader range of initializations.

4. **Mitigation of Internal Covariate Shift:**

- Internal covariate shift refers to changes in the distribution of inputs to a layer as the network learns. Normalization helps in reducing these shifts, maintaining consistent distribution of inputs across layers as the network learns.

5. **Enhanced Generalization:**

- Networks with normalization tend to generalize better to new data because they are less prone to overfitting. Normalization acts as a form of regularization.

6. **Allowance for Higher Learning Rates:**

- By controlling the scale dynamics within the network, normalization often permits the use of higher learning rates, which can accelerate the training process.

In summary, normalization is a vital component in modern neural network architectures, enabling more efficient and stable training processes, which are crucial for achieving high-performance models.

How normalization is done in CNN

Normalization in CNNs can be achieved through several techniques, each with its specific methodology and application. Here's a brief overview of how some of the most common normalization methods are performed:

1. **Batch Normalization:**

- **Calculation:** For each mini-batch, calculate the mean and variance of the inputs for each feature (or channel). This is done across the batch and spatial dimensions for CNNs.
- **Normalization:** Use these statistics to normalize the inputs to have zero mean and unit variance.
- **Scaling and Shifting:** After normalization, apply two learned parameters per feature: a scale parameter (gamma) and a shift parameter (beta), allowing the model to learn the optimal representational space.
- **Formula:** output = \left(\frac{\text{input} - \mu}{\sqrt{\sigma^2 + \epsilon}}\right) + \beta] where (\mu) is the mean, (\sigma^2) is the variance, and (\epsilon) is a small constant for numerical stability.

2. **Layer Normalization:**

- Like batch normalization, but the normalization is performed over all the features for each individual data sample, rather than across the batch. This makes it independent of the batch size.
- It calculates the mean and variance across the feature dimensions and normalizes the inputs for each sample.

3. **Instance Normalization:**

- This method normalizes each example independently. It calculates the mean and variance for each example individually, rather than using the batch statistics.
- It's commonly used in tasks like style transfer where maintaining individual image styles is important.

4. **Group Normalization:**

- Divides the channels into groups and computes mean and variance within each group for normalization.
- Unlike batch normalization, it is more flexible with varying batch sizes and is often a better choice when batch sizes are small.

Each of these methods adjusts the distribution of the inputs at different granularity levels (batch, instance, layer, or group), and the choice of method depends on the specific architecture and training conditions.

VGGNet in detail. Lets break down the VGGNet in more detail as a case study. The whole VGGNet is composed of CONV layers that perform 3x3 convolutions with stride 1 and pad 1, and of POOL layers that perform 2x2 max pooling with stride 2 (and no padding). We can write out the size of the representation at each step of the processing and keep track of both the representation size and the total number of weights:

```
INPUT: [224x224x3]        memory:  224*224*3=150K
weights: 0
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M
weights: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64]  memory:  224*224*64=3.2M
weights: (3*3*64)*64 = 36,864
POOL2: [112x112x64]  memory:  112*112*64=800K
weights: 0
CONV3-128: [112x112x128]  memory:  112*112*128=
1.6M   weights: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128]  memory:  112*112*128=
1.6M   weights: (3*3*128)*128 = 147,456
POOL2: [56x56x128]  memory:  56*56*128=400K
weights: 0
CONV3-256: [56x56x256]  memory:  56*56*256=800K
weights: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256]  memory:  56*56*256=800K
weights: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256]  memory:  56*56*256=800K
weights: (3*3*256)*256 = 589,824
POOL2: [28x28x256]  memory:  28*28*256=200K
weights: 0
CONV3-512: [28x28x512]  memory:  28*28*512=400K
weights: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512]  memory:  28*28*512=400K
weights: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512]  memory:  28*28*512=400K
weights: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512]  memory:  14*14*512=100K
weights: 0
CONV3-512: [14x14x512]  memory:  14*14*512=100K
weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512]  memory:  14*14*512=100K
weights: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512]  memory:  14*14*512=100K
```

```
weights: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512]  memory:  7*7*512=25K  weights: 0
FC: [1x1x4096]  memory:  4096  weights: 7*7*512*
4096 = 102,760,448
FC: [1x1x4096]  memory:  4096  weights: 4096*4096
= 16,777,216
FC: [1x1x1000]  memory:  1000 weights: 4096*1000 =
4,096,000

TOTAL memory: 24M * 4 bytes ~= 93MB / image (only
forward! ~*2 for bwd)
TOTAL params: 138M parameters
```