Unit 4

CNN

INTRODUCTION:

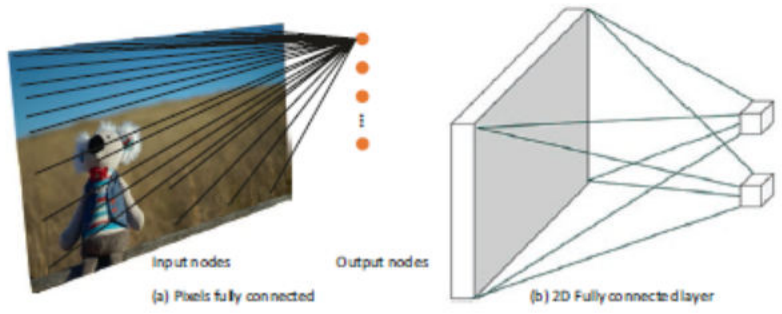
The network model based on the multilayer perceptron (MLP) mathematical model of biological neurons was called a neural network.
Due to factors such as limited computing power and small data size at the time, neural networks were generally only able to train to a small number of layers. We call this type of neural network a shallow neural network (shallow neural network).
It is not easy for shallow neural networks to extract high-level features from data, and the general
expression ability is not good.

PROBLEM WITH FULLY CONNECTED LAYER:

- The problem with using fully connected layers (also known as dense layers) directly on image data is that they don't take into account the spatial structure of the data.
- In an image, pixels that are close to each other are often highly correlated and form meaningful patterns, such as edges, textures, and shapes.
- Fully connected layers treat each pixel as an independent feature, ignoring the spatial relationships between them.
- This can result in a large number of parameters, making the model computationally expensive and prone to overfitting, especially when dealing with high-resolution images.

LOCAL CORRELATION:

Fully connected layers typically connect every neuron to every neuron in the previous layer, and they do not inherently capture local correlations. Each neuron in a fully connected layer receives information from every neuron in the preceding layer, without considering the spatial relationships between them.



It can be seen that each output node of the network layer is connected to all input nodes for extracting the feature information of all input nodes. This dense

connection method is the root cause of the large number of parameters and the high computational cost of the fully connected layer.
The fully connected layer is also called dense connection layer (dense layer), and the relationship between output and input is:

$$o_j = \sigma\left(\sum_{i \in nodes(I)} w_{ij} x_i + b_j \right)$$

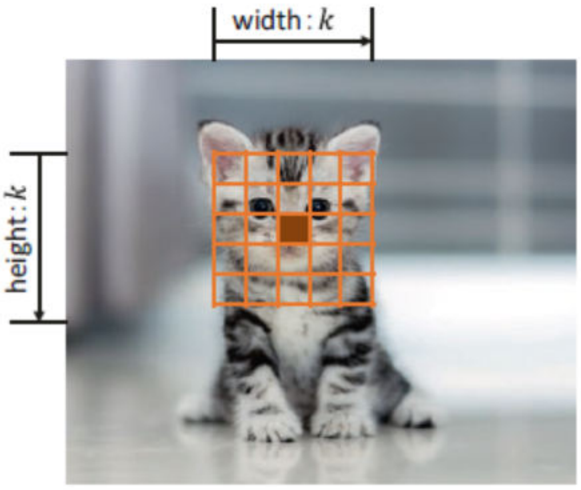
where nodes(I) represents the set of nodes in layer I.

IS THERE AN APPROXIMATE SIMPLIFIED MODEL?

In real life, there are a lot of data that use location or distance as a measure of importance distribution. For example, people who live closer to themselves are more likely to have greater influence on themselves (location correlation), and stock trend predictions should pay more attention to the recent trend (time correlation); each pixel of the picture is more related to the surrounding pixels (location correlation).

Taking 2D image data as an example, if we simply think that the pixels with Euclidean distance from the current pixel is less than or equal to *k*/root(2) are more important, and those with the Euclidean distance is greater than *k*/root(2) are less important, then we can easily simplify the problem of finding the importance distribution of each pixel.

Consider the below picture, the solid grid is located are used as reference points and the pixels
whose Euclidean distance is less than or equal to *k/root(*2) are represented by a rectangular grid.
The pixels in the grid are more important, and the pixels outside the grid are less important. This window is called the **receptive field,** which characterizes the importance distribution of each pixel to the central pixel.
The pixels within the grid will be considered, and the pixels outside the grid will be ignored for the central pixel. This hypothetical characteristic of distance-based importance distribution is called **LOCAL CORRELATION**. It only focuses on some nodes that are close to itself and ignores nodes that are far away.



WEIGHT SHARING:

- Each output node is only connected to $k \times k$ input nodes in the receptive field, and the number of output layer nodes is $\|J\|$. So the number of the parameters of the current layer is $k \times k \times \|J\|$.
- Comparing to the fully connected layer, because k is usually small, such as 1, 3, and 5, so
  $k \times k \ll \|I\|$, which means it successfully reduced the amount of parameters.

Can the amount of parameters be further reduced, for example, can we only need $k \times k$ parameters to complete the calculation of the current layer?

The answer is yes. Through the idea of weight sharing, for each output node *oj*, the same weight matrix W is used, then no matter how many output nodes $\|J\|$ will be, the number of network layer parameters is always $k \times k$.

NOTE:
By applying the idea of local correlation and weight sharing, we have successfully reduced the number of network parameters from $\|I\| \times \|J\|$ to $k \times k$ (to be precise, under the conditions of a single input channel and a single convolution kernel). This kind of weighted "local connection layer" network is actually a convolutional neural network.

CONVOLUTION OPERATION:

- Under the a priori of local correlation, we propose a simplified "local connection layer." For all pixels in the window $k \times k$, feature information is extracted by multiplying and accumulating weights, and each output node extracts features corresponding to the receptive field area. information. This operation is actually a standard operation in the field of signal processing: discrete convolution operation.

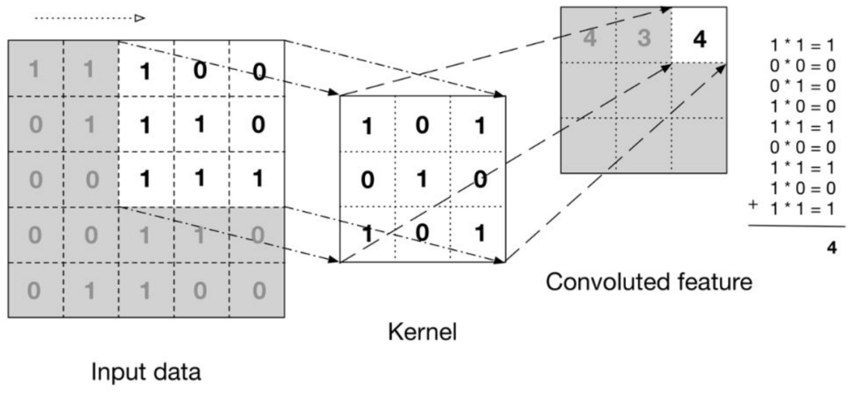
$$(f \otimes g)(n) = \sum_{\tau=-\infty}^{\infty} f(\tau) g(n-\tau)$$

- In deep learning, a **convolutional neural network** (**CNN/ConvNet**) is a class of deep neural networks, most commonly applied to analyze visual imagery.
- There are 4 main components of CNN:
  1. Convolution
  2. Non- linearity
  3. Pooling/ sub-sampling
  4. Classification(fully connected layer)

- BASIC TERMINOLOGY:
  - **Filter, Kernel, or Feature Detector** is a small matrix used for features detection.
  - **Convolved Feature, Activation Map or Feature Map** is the output volume formed by sliding the filter over the image and computing the dot product.
  - **Receptive field** is a local region of the input volume that has the same size as the filter.
  - **Depth** is the number of filters.
  - **Stride** has the objective of producing smaller output volumes spatially.

- **convolution** is a mathematical operation on two functions that produces a third function that expresses how the shape of one is modified by the other.
- **Why to use convolution operation??**

  Preserves the spatial relationship between pixels by learning image features using small
  squares of input data. It also Detects small, meaningful features such as edges with kernels



- Common kernels used in convolutional neural networks (CNNs) are designed to capture specific features or patterns in the input data. Here are some common kernels and their applications:

1. Identity Kernel:

```
0 0 0
0 1 0
0 0 0
```

Application: The identity kernel has no effect on the input and is used when no feature extraction or modification is desired. It is often used in the early layers of a CNN to maintain the spatial dimensions of the input.

2. Vertical Edge Detection:

```
-1 0 1
-1 0 1
-1 0 1
```

Horizontal Edge Detection:

```
-1 -1 -1
 0  0  0
 1  1  1
```

Application: These kernels are used to detect vertical and horizontal edges in images, respectively. They highlight changes in intensity along the specified direction.

3. Blur (Box Blur) Kernel:

```
1 1 1
1 1 1
1 1 1
```

Application: The blur kernel is used for smoothing and blurring an image. It helps reduce noise and can be used for applications such as image denoising.

4. Sharpening Kernel:

```
0 -1  0
-1  5 -1
 0 -1  0
```

Application: The sharpening kernel enhances edges and fine details in an image. It is used to make features more prominent.

5. Emboss Kernel:

-2 -1 0
-1 1 1
 0 1 2
Application: The emboss kernel creates a 3D effect by emphasizing changes in intensity and directional lighting in an image.

6. Gaussian Blur Kernel:

1 2 1
2 4 2
1 2 1
Application: The Gaussian blur kernel provides a smoother blur than the box blur and is commonly used for image smoothing and blurring.

7. Sobel Kernels (Edge Detection):

Sobel Vertical Kernel:

-1 0 1
-2 0 2
-1 0 1
Sobel Horizontal Kernel:
-1 -2 -1
 0 0 0
 1 2 1
Application: Sobel kernels are used for edge detection in images. They highlight changes in intensity along both vertical and horizontal directions.

These are just a few examples of common kernels used in CNNs. Depending on the specific task, researchers may design custom kernels or use pre-designed ones to extract features relevant to the problem at hand. The choice of kernels can significantly impact the network's ability to learn and represent features in the input data.

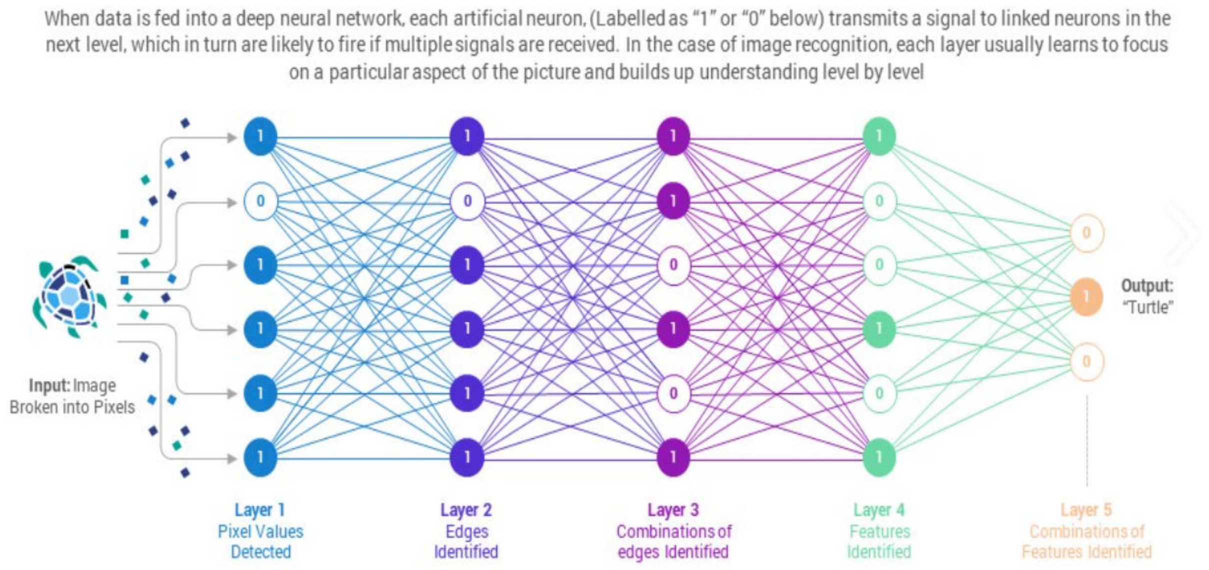
CONVOLUTIONAL NEURAL NETWORK:

The convolutional neural network makes full use of the idea of local correlation and weight sharing, which greatly reduces the amount of network parameters, thereby improving training efficiency and making it easier to realize ultra-large-scale deep networks.
CNNs are specifically designed to address the drawbacks of standard ANNs when it comes to processing visual data.

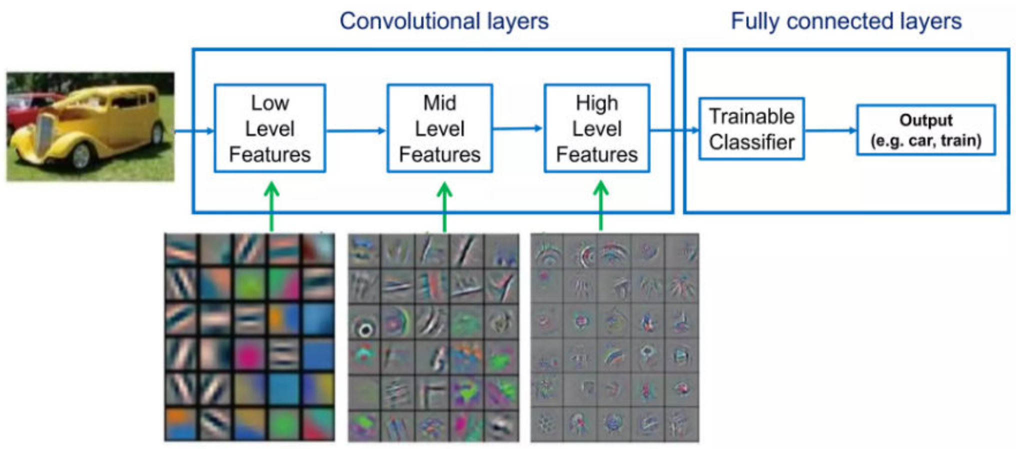
Drawbacks of ANN:

- High Dimensionality
- Lack of spatial awareness
- Computational complexity
- Over fitting
- Lack of weight sharing etc.

Glance at CNN:


When data is fed into a deep neural network, each artificial neuron, (Labelled as "1" or "0" below) transmits a signal to linked neurons in the next level, which in turn are likely to fire if multiple signals are received. In the case of image recognition, each layer usually learns to focus on a particular aspect of the picture and builds up understanding level by level
Input: Image Broken into Pixels
Output: "Turtle"
Layer 1 Pixel Values Detected
Layer 2 Edges Identified
Layer 3 Combinations of edges Identified
Layer 4 Features Identified
Layer 5 Combinations of Features Identified

Learning through CNN:



CNN COMPONENTS:

There are 4 main components of CNN:

1. Convolution
2. Non- linearity
3. Pooling/ sub-sampling
4. Classification(fully connected layer)

CNN LAYERS – 1

CONVOLUTIONAL LAYER:

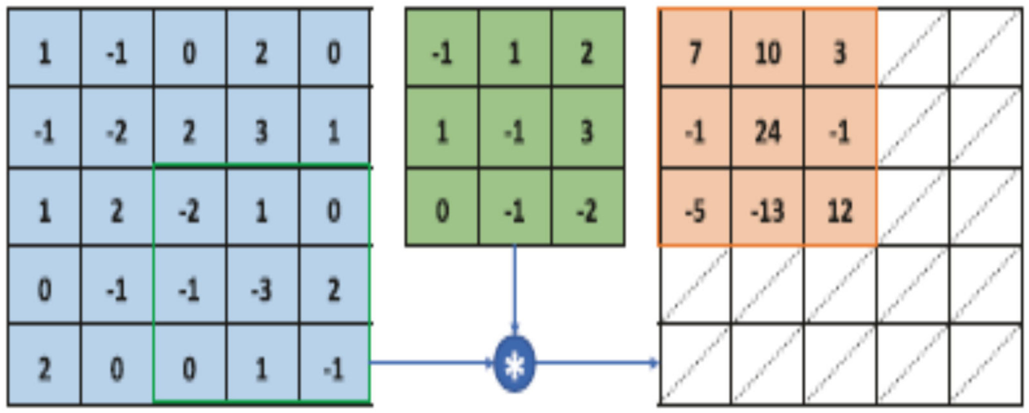
It is the Base layer of CNN and is responsible for determining the features of the pattern.The input image is passed through a filter which gives feature map as output.This layer applies some kernels that slide through the pattern to extract low- and high-level features in the pattern.
let’s introduce the specific calculation process of the convolutional neural network layer. Taking 2D image data as an example, the convolutional layer accepts input feature maps *X* with
height *h* and width *w*, and the number of channels *cin*.
Under the action of *cout* convolution kernels with height *h* and width *w* and the number of channels *cin*, feature maps with the height *h'* and width *w'* and *cout* channels are generated. It should be noted that the height and width of the convolution kernel can be unequal. In order to simplify the discussion, we only consider the equal height and width cases, and then it can be easily extended to the case of unequal height and width.

Single-Channel Input and Single Convolution Kernel:

When dealing with a single-channel input (grayscale image) and a single convolution kernel, the convolution operation simplifies. Here's how the convolution would be applied:
Let's represent a 3x3 grayscale image as follows:
2 4 1
5 9 7
3 8 6
For illustration, consider a simple 3x3 convolution kernel:
-1 0 1
-1 2 2
0 1 -1
The convolution operation involves sliding the kernel over the input image, computing the element-wise multiplication at each position and summing the results to produce a feature map. Let's compute the convolution at the top-left corner of the image:
Convolution at (1,1):
(2 * -1) + (4 * 0) + (1 * 1) + (5 * -1) + (9 * 2) + (7 * 2) + (3 * 0) + (8 * 1) + (6 * -1) = 29
As the kernel slides across the entire image, similar computations are performed at each position, resulting in a new matrix known as the feature map. The size of the feature map depends on factors such as the size of the input image, the size of the kernel, and the stride used in the convolution.

Note: In practical applications, a bias term would typically be added to each computed value in the feature map, and an activation function may be applied to introduce non-linearity.

Another example:



Multi-channel Input and Single Convolution Kernel:
Multi-channel input convolutional layers are more common. For example, a color image contains three channels (R/G/B). The pixel value on each channel indicates the intensity of the R/G/B color. In the following, we take three-channel input and a single convolution kernel as an example to extend the convolution operation of single-channel input to multichannel.
When dealing with multi-channel input (for example, a color image with three channels - red, green, and blue) and a single convolution kernel, the convolution operation is extended to handle each channel separately. This is known as depth-wise convolution.

In the case of multi-channel input, the number of channels of the convolution kernel needs to match the number of input channels. The *ith* channel of the convolution kernel and the *ith* channel of the input *X* are calculated to obtain the first intermediate matrix, which can be then regarded as the case of single input and single convolution kernel. The corresponding elements of the intermediate matrix of all channels are added again as the final output.

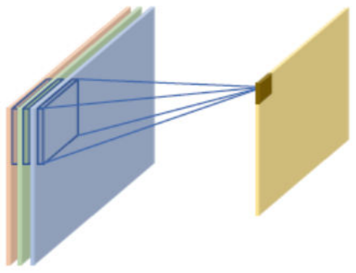


Figure 10-20. Multi-channel input and single convolution kernel diagram

Here's how the convolution would be applied with a single convolution kernel across a multi-channel input:

Multi-channel Input:
Let's represent a 3x3 RGB image with three channels:
R: 2 4 1
G: 5 9 7
B: 3 8 6
Single Convolution Kernel:
Consider a single 3x3 convolution kernel that will be applied to each channel:
-1 0 1
-1 2 2
0 1 -1
Convolution Operation:
The convolution operation is applied independently to each channel, and the results are summed to produce a single output value for each position in the feature map.
Convolution at (1,1) for the Red channel:
(2 * -1) + (4 * 0) + (1 * 1) + (5 * -1) + (9 * 2) + (7 * 2) + (3 * 0) + (8 * 1) + (6 * -1) = 29
Convolution at (1,1) for the Green channel:
(2 * -1) + (4 * 0) + (1 * 1) + (5 * -1) + (9 * 2) + (7 * 2) + (3 * 0) + (8 * 1) + (6 * -1) = 29
Convolution at (1,1) for the Blue channel:
(2 * -1) + (4 * 0) + (1 * 1) + (5 * -1) + (9 * 2) + (7 * 2) + (3 * 0) + (8 * 1) + (6 * -1) = 29
Summing Results:
The results of the individual channel convolutions are summed to produce the final value for that position in the feature map:
29 (Red) + 29 (Green) + 29 (Blue) = 87
As the kernel slides across the entire image, similar computations are performed at each position for each channel, resulting in a new matrix known as the feature map.
Generally speaking, a convolution kernel can only complete the extraction of a certain logical feature. When multiple logical features need to be extracted at the same time, it can be achieved by adding multiple convolution kernels to improve the expression ability of the neural
network. This is the case of multi-channel input and multi-convolution kernels.

Multi-channel Input and Multi-convolution Kernel:
Multi-channel input and multi-convolution kernels are the most common forms of convolutional neural networks. We have already introduced the operation

process of single convolution kernels. Each convolution kernel and input are convolved to obtain an output matrix. When there are multiple convolution kernels, the *i*th ($i \in [1, n]$, n is the number of convolution kernels) convolution kernel and input X get the *i*th output matrix (also called the channel i of output tensor O), and finally all the output matrix in the channel dimension stitch together (stack operation to create a new dimension – the number of output channels) to generate an output tensor O that contains n channels.

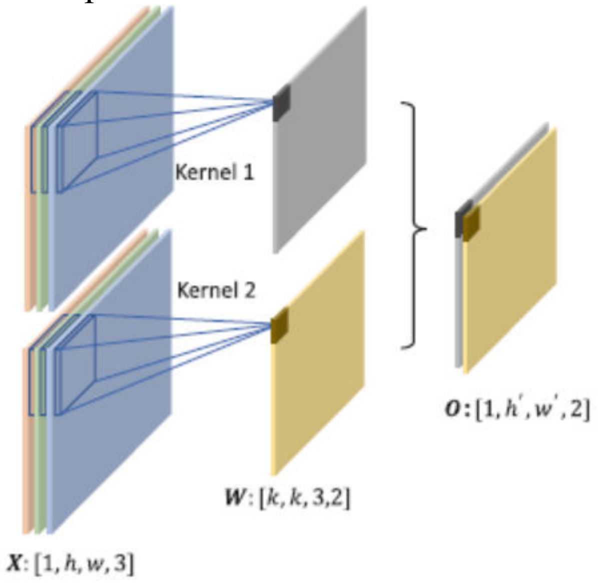


Figure 10-21. Diagram of multi-convolution kernels

When dealing with a multi-channel input and multiple convolution kernels, the convolution operation becomes more complex. Each convolution kernel is applied independently to its corresponding channel, and the results are summed across all channels to produce a single value for each position in the feature map. This is known as cross-channel or cross-correlation.

Here's how the convolution would be applied with multiple convolution kernels across a multi-channel input:

Multi-channel Input:
Let's represent a 3x3 RGB image with three channels:
R: 2 4 1
G: 5 9 7
B: 3 8 6
Multiple Convolution Kernels:
Consider two 3x3 convolution kernels, one for detecting vertical edges and another for detecting horizontal edges:
Vertical Edge Detection Kernel:
-1 0 1
-1 0 1
-1 0 1

Horizontal Edge Detection Kernel:
-1 -1 -1
0 0 0
1 1 1
Convolution Operation:
For each convolution kernel, the convolution operation is applied independently to each channel of the input, and the results are summed across all channels to produce a single output value for each position in the feature map.

Vertical Edge Detection:

Convolution at (1,1) for the Vertical Edge Detection:
(2 * -1) + (4 * 0) + (1 * 1) + (5 * -1) + (9 * 0) + (7 * 1) + (3 * -1) + (8 * 0) + (6 * 1) = 4
Horizontal Edge Detection:
Convolution at (1,1) for the Horizontal Edge Detection:
(2 * -1) + (4 * -1) + (1 * -1) + (5 * 0) + (9 * 0) + (7 * 0) + (3 * 1) + (8 * 1) + (6 * 1) = 4

Summing Results:
The results of the individual convolution operations are summed to produce the final value for that position in the feature map:
4 (Vertical Edge Detection) + 4 (Horizontal Edge Detection) = 8

his value, 8, represents the output for the corresponding position in the feature map.

Resulting Feature Map:
As the convolution kernels slide across the entire image, similar computations are performed at each position for each channel, and the results are summed across channels, resulting in a new matrix known as the feature map.

In practical applications, biases may be added, and activation functions may be applied after the summation to introduce non-linearity to the feature map. The use of multiple channels and multiple kernels allows convolutional neural networks to capture a wide variety of features from the input data.

STRIDE SIZE:
In convolution operation, how to control the density of receptive field layout? For inputs with high information density, such as pictures with a large number of objects, in order to maximize the useful information, it is desirable to arrange the receptive field windows more densely during network design.

For inputs with lower information density, such as a picture of the ocean, we can reduce the number of receptive fields in an appropriate amount. The control method of receptive field density is
generally realized by moving strides.
The stride size is a parameter used in the convolution operation of a neural network, and it determines the step size at which the convolutional kernel moves across the input data. The stride affects the size of the resulting feature map and, consequently, the computational complexity of the network.

Here's how the stride size works:

Stride Size Definition:
The stride size is the number of pixels the convolutional kernel moves at each step or slide across the input data.
A stride of 1 means the kernel moves one pixel at a time, while a larger stride (e.g., 2) means the kernel skips pixels, resulting in a larger step.

Effect on Feature Map Size:
Smaller stride sizes result in larger feature maps because the convolutional kernel overlaps more and covers the input data more comprehensively.
Larger stride sizes result in smaller feature maps because the convolutional kernel skips over parts of the input, reducing the coverage.

Mathematical Relationship:
The size of the feature map (output size) can be computed using the formula:

$$\text{Output size} = \frac{\text{Input size} - \text{Kernel size}}{\text{Stride}} + 1$$

Here, "Input size" refers to the spatial dimensions of the input data (e.g., width or height), "Kernel size" is the size of the convolutional kernel, and "Stride" is the stride size.

Impact on Computational Complexity:
Smaller stride sizes result in more computations because the convolutional kernel processes more overlapping regions of the input data.
Larger stride sizes reduce the number of computations but may lead to loss of fine-grained spatial information.

Use Cases:
Smaller strides are often used in the early layers of convolutional neural networks (CNNs) to capture fine details and spatial information.

Larger strides may be used in later layers to reduce the spatial dimensions and computational load.

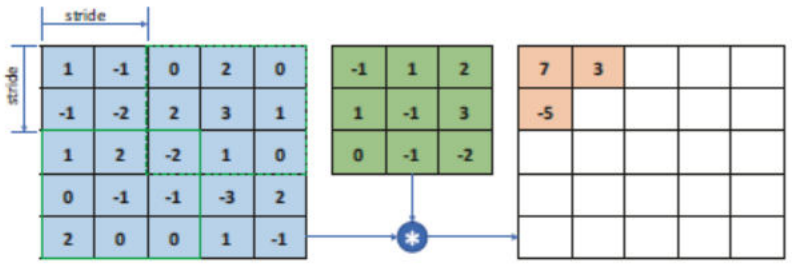
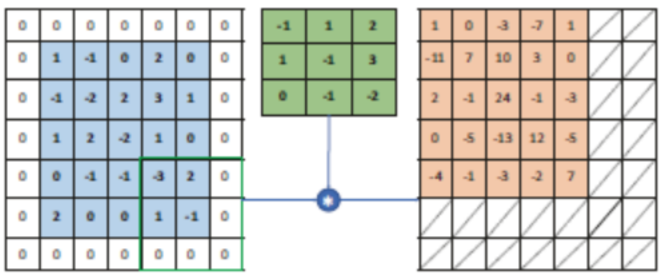


Figure 10-23. Convolution operation stride size demnostration-1

PADDING:
As we move onto the next convolutional layer the details become more complex. There is potential for errors with convolutional layers including image shrinking or the edges becoming less clear. To help fight against this the data scientist can use something called padding.
After the convolution operation, the height and width of the output will generally be smaller than the height and width of the input. Even when the stride size is 1, the height and width of the output will be slightly smaller than the input height and width. When designing a network model, it is sometimes desired that the height and width of the output can be the same as the height and width of the input, thereby facilitating the design of network parameters and residual connection. In order to make the height and width of the output equal to that of the input, it is common to increase the input by padding several invalid elements on the height and width of the original input. By carefully designing the number of filling units, the height and width of the output after the convolution operation can be
equal to the original input, or even larger.
Padding is basically adding a layer of pixels around the edge of an image in order to preserve the true size of the image when convolution is occurring.
Padding is a technique used in the context of convolutional neural networks (CNNs) to address the reduction in spatial dimensions that occurs when applying convolutional or pooling layers to an input. Padding involves adding extra pixels around the boundaries of the input data, effectively creating a "border" or "padding" of zeros.



Here's how padding works and why it's used:

Spatial Dimension Preservation:
When a convolutional kernel slides across the input data, pixels at the edges may not be fully covered by the kernel, leading to a reduction in the spatial dimensions of the feature map.
Padding helps preserve the spatial dimensions by adding extra pixels around the input, ensuring that the convolutional kernel covers all parts of the input.

Mathematical Representation:
The size of the feature map after convolution (or pooling) can be calculated using the formula:

$$\text{Output size} = \frac{\text{Input size} - \text{Kernel size} + 2 \times \text{Padding}}{\text{Stride}} + 1$$

Here, "Input size" refers to the spatial dimensions of the input data, "Kernel size" is the size of the convolutional kernel, "Padding" is the size of the zero-padding, and "Stride" is the stride size.

Types of Padding:
Valid (No Padding): No padding is applied (Padding=0). This can lead to a reduction in spatial dimensions.
Same (Zero Padding): Padding is applied to keep the output size the same as the input size. Padding is calculated as $\text{Padding} = \frac{\text{Kernel size} - 1}{2}$ for a stride of 1.

Benefits of Padding:
Preservation of Edge Information: Padding helps preserve edge information in the input, especially in the early layers of the network where fine details are crucial.
Mitigating Border Effects: Without padding, pixels at the borders of the input may be under represented in the feature map, leading to a loss of information. Padding helps mitigate this effect.

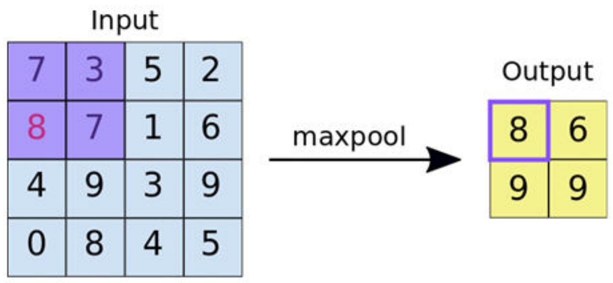
Use Cases:
Padding is often used in the early layers of a CNN to maintain spatial information, and it may be reduced or eliminated in later layers to reduce computational cost.Pooling layers, like max pooling or average pooling, can also benefit from padding to avoid size reduction.

POOLING LAYER:

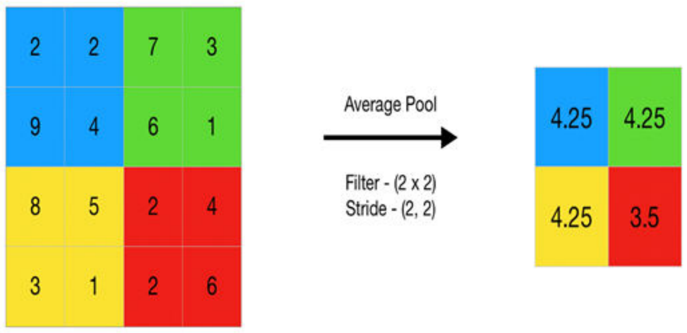
- Convolutional layers provide activation maps.
- Pooling layer applies non-linear down sampling on activation maps.
- It is responsible for reducing the spatial size of the Convolved Feature.
- This is to **decrease the computational power required to process the data** by reducing the dimensions.
- Pooling layers are a crucial component of convolutional neural networks (CNNs) that contribute to spatial down-sampling and feature selection.
- Pooling layers are typically inserted between convolutional layers to progressively reduce the spatial dimensions of the input data, leading to a more compact representation.

There are two common types of pooling layers:

Max Pooling:



Operation:
For each local region in the input (typically non-overlapping), the max pooling operation selects the maximum value.
Effect:
Max pooling helps retain the most salient features in a local neighborhood, emphasizing the presence of specific patterns or activations.
Average Pooling:



Operation:
For each local region in the input (typically non-overlapping), the average pooling operation computes the average value.
Effect:
Average pooling provides a smoother down-sampling, reducing the impact of outliers or extreme values.

Purpose of Pooling Layers:

- Spatial Down-Sampling: Pooling layers reduce the spatial dimensions of the input data, which helps in reducing the computational complexity of the network.Down-sampling also helps in creating a more abstract and hierarchical representation of the input.
- Translation Invariance: Pooling layers contribute to translation invariance, making the network less sensitive to small translations or shifts in the input data.
- Reducing Overfitting: Pooling can help prevent overfitting by reducing the spatial dimensions and the number of parameters in the network. This encourages the learning of more general features.
- Feature Selection: By selecting the maximum or average values in local regions, pooling layers emphasize important features and help discard less relevant information.

DROPOUT LAYER:
Dropout is a regularization technique commonly used in neural networks, including convolutional neural networks (CNNs), to prevent overfitting and improve the generalization performance of the model. It involves randomly setting a fraction of input units to zero during training.

- Dropout Operation: During training, for each update, a random subset of neurons (both input and/or hidden neurons) is set to zero with a certain probability.This process is applied independently for each training example and each mini-batch.
- Fraction of Dropped Units:The fraction of neurons that are dropped out is a hyperparameter known as the dropout rate. Common dropout rates are

between 0.2 and 0.5, meaning that, on average, 20% to 50% of the neurons are dropped out during training.

- Effect during Training: Dropout introduces a form of model averaging over many different architectures. The network becomes less reliant on specific neurons and learns more robust features.It helps prevent overfitting by avoiding the reliance on a particular set of neurons, making the model more adaptable to different data.
- No Dropout during Inference: During the inference or testing phase, the dropout is usually turned off, and the full network is used for making predictions.

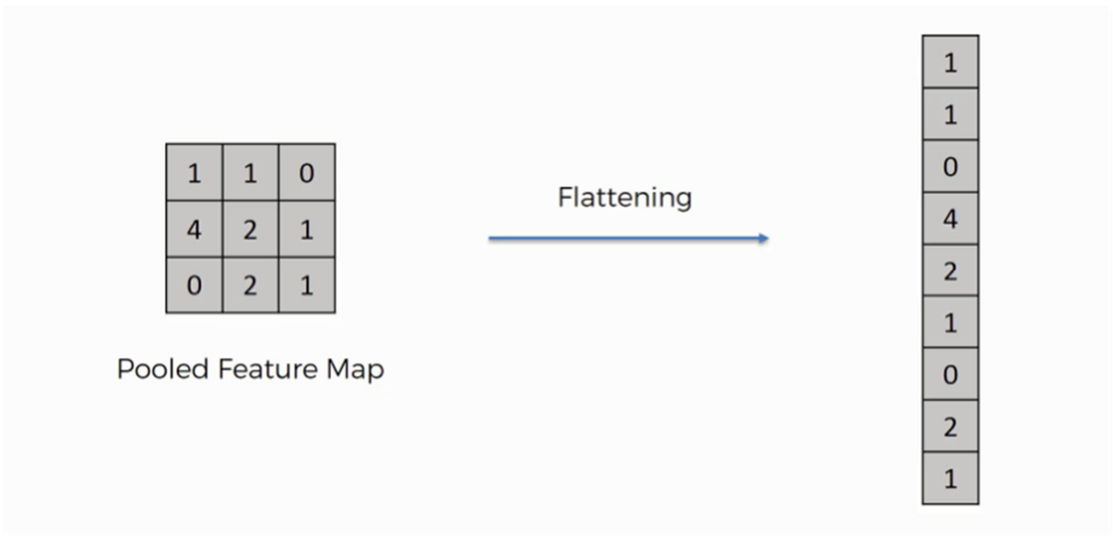
Sample Code:

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28,
28, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))  # Dropout layer with a dropout rate of 0.5
model.add(Dense(10, activation='softmax'))
```

FLATTEN LAYER:

The Flatten layer is a component used in neural network architectures, particularly in deep learning frameworks like TensorFlow and Keras. Its purpose is to transform the input data from a multidimensional array (such as a 2D or 3D array) into a one-dimensional array. This is often necessary when transitioning from convolutional layers to fully connected layers in a neural network.



DENSE LAYER:

The Dense layer, also known as a fully connected layer, is a fundamental building block in neural networks. It connects each neuron in one layer to every neuron in the next layer, forming a densely connected network. The Dense layer is a versatile layer that is commonly used in various types of neural network architectures, including feedforward neural networks and convolutional neural networks (CNNs).

- Full Connectivity: In a Dense layer, each neuron is connected to every neuron in the previous and next layers. The term "dense" refers to the dense connections between neurons.
- Weights and Biases: Each connection between neurons has an associated weight parameter, which the network learns during training.Additionally, each neuron has a bias term.
- Activation Function: A non-linear activation function is typically applied to the output of each neuron in the Dense layer. Common activation functions include ReLU (Rectified Linear Unit), sigmoid, or tanh.
- Output Size: The number of neurons in the Dense layer determines the output size of the layer. The output size is a hyperparameter that depends on the task and the architecture of the neural network.
- Training and Learning: During training, the weights and biases of the Dense layer are optimized using techniques like gradient descent and backpropagation to minimize a defined loss function.
- Common Usage: Dense layers are often used in the final stages of a neural network architecture for tasks like classification or regression, where they map the learned features to the desired output.

Purpose of Dense Layers:

- Feature Learning and Representation:Dense layers are responsible for learning complex patterns and representations from the features extracted by preceding layers.They transform the input into a format suitable for the desired output task.
- Non-Linearity and Model Complexity:The non-linear activation functions in Dense layers enable the network to learn and represent complex, non-linear relationships in the data.
- Classification and Regression: In classification tasks, the output of a Dense layer may represent class probabilities.In regression tasks, it might predict continuous values.
- Integration in Neural Architectures: Dense layers are often used in conjunction with other layers, such as convolutional layers or recurrent layers, in various neural network architectures.

SAMPLE CODE FOR IMAGE CLASSIFICATION USING CNN:

In this example, we're using the MNIST dataset for simplicity, which consists of grayscale images of handwritten digits (0-9). The CNN architecture consists of convolutional layers followed by max-pooling layers, and then fully connected layers. The model is trained using the Adam optimizer and categorical cross-entropy loss. After training, the model is evaluated on the test set.

```
# Import necessary libraries
import numpy as np
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from keras.datasets import mnist
from keras.utils import to_categorical

# Load and preprocess the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalize pixel values to be between 0 and 1
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# One-hot encode the labels
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Add a channel dimension to the images (required for convolutional layers)
x_train = np.expand_dims(x_train, axis=-1)
x_test = np.expand_dims(x_test, axis=-1)

# Build the CNN model
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Flatten())
model.add(Dense(64, activation='relu'))
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
```

```
# Train the model
model.fit(x_train, y_train, epochs=5, batch_size=64, validation_data=(x_test,
y_test))

# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f'Test Accuracy: {test_accuracy * 100:.2f}%')
```

LeNet5 NETWORK:
LeNet-5 is a convolutional neural network (CNN) architecture designed for handwritten and machine-printed character recognition. It was proposed by Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner in 1998. LeNet-5 was one of the pioneering architectures in the field of deep learning and played a significant role in the development of convolutional neural networks.
LeNet-5 was originally designed for the recognition of handwritten digits, but its principles laid the foundation for more advanced architectures. Modern CNN architectures, such as AlexNet, VGG, and ResNet, have built upon the ideas introduced by LeNet-5, incorporating deeper networks and more sophisticated components.
In practice, for more recent applications, especially on larger and more complex datasets, deeper and more sophisticated architectures are often preferred. LeNet-5, however, remains a historic milestone in the evolution of convolutional neural networks.

Network Structure Diagram of LeNet5:
It accepts digital and character pictures of size 32 × 32 as input and then passes through the first convolution layer to obtain the tensor with shape [*b*, 28,28,6]. After a downsampling layer, the tensor size is reduced to [*b*, 14,14,6]. After the second convolutional layer, the tensor shape becomes [*b*, 10,10,16]. After similar downsampling layer, the tensor size is reduced to [*b*, 5, 5, 16]. Before entering the fully connected layer, the tensor is converted to shape [*b*, 400] and feed into two fully connected layers with the number of input nodes 120 and 84, respectively. A tensor with shape [*b*, 84] is obtained and finally goes through the Gaussian connections layer.

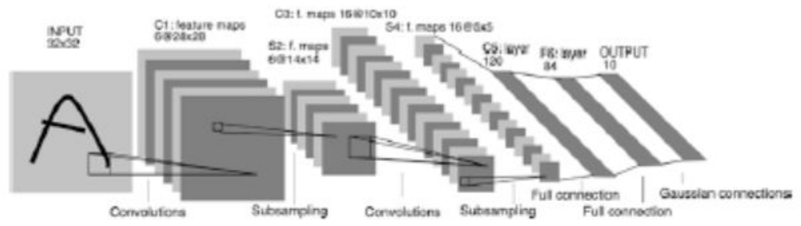


Figure 10-30. LeNet-5 structure [4]

LeNet-5 Architecture:
The LeNet-5 architecture consists of seven layers, including convolutional layers, pooling layers, and fully connected layers. Here's a high-level overview:

Input Layer:
The network takes a grayscale image of size 32x32 as input.
Convolutional Layers:
Convolutional layers are used for feature extraction. LeNet-5 has two convolutional layers:
Convolutional Layer 1:
Filters: 6 filters of size 5x5
Activation: Tanh
Input Size: 32x32
Output Size: 28x28
Average Pooling Layer 1:
Pooling: 2x2, Stride: 2
Output Size: 14x14
Convolutional Layer 2:
Filters: 16 filters of size 5x5
Activation: Tanh
Output Size: 10x10
Average Pooling Layer 2:
Pooling: 2x2, Stride: 2
Output Size: 5x5
Flatten Layer:
The output of the second pooling layer is flattened into a vector.
Fully Connected Layers:
LeNet-5 has three fully connected layers:
Fully Connected Layer 1:
Neurons: 120
Activation: Tanh
Fully Connected Layer 2:
Neurons: 84
Activation: Tanh
Output Layer:
Neurons: 10 (for 10 classes in the case of digit recognition)
Activation: Softmax
Activation Function: Tanh
The activation function used in the convolutional and fully connected layers of LeNet-5 is the hyperbolic tangent function (tanh), which squashes the output to the range [-1, 1].
Output Layer: Softmax
The output layer uses the softmax activation function to convert the network's raw output into probabilities for each class. It is commonly used in multi-class classification problems.

HANDSON LENET5:

```
import numpy as np
from keras.models import Sequential
from keras.layers import Conv2D, AveragePooling2D, Flatten, Dense
from keras.datasets import mnist
from keras.utils import to_categorical

# Load and preprocess the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalize pixel values to be between 0 and 1
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Add a channel dimension to the images (required for convolutional layers)
x_train = np.expand_dims(x_train, axis=-1)
```

```
x_test = np.expand_dims(x_test, axis=-1)

# One-hot encode the labels
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Build LeNet-5 model
model = Sequential()

# Convolutional Layer 1
model.add(Conv2D(6, kernel_size=(5, 5), activation='tanh', input_shape=(28, 28, 1)))
model.add(AveragePooling2D(pool_size=(2, 2), strides=(2, 2)))

# Convolutional Layer 2
model.add(Conv2D(16, kernel_size=(5, 5), activation='tanh'))
model.add(AveragePooling2D(pool_size=(2, 2), strides=(2, 2)))

# Flatten Layer
model.add(Flatten())

# Fully Connected Layer 1
model.add(Dense(120, activation='tanh'))

# Fully Connected Layer 2
model.add(Dense(84, activation='tanh'))

# Output Layer
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model
model.fit(x_train, y_train, epochs=10, batch_size=128, validation_data=(x_test, y_test))

# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f'Test Accuracy: {test_accuracy * 100:.2f}%')
```

REPRESENTATION LEARNING:

- The complex convolutional neural network model is also based on the stacking of convolutional layers. In the past, researchers have discovered that the deeper the network layer, the stronger the model's expressive ability, and the more likely it is to achieve better performance.
- The image recognition process is generally considered to be a representation learning process. Starting from the original pixel features received, it gradually extracts low-level features such as edges and corners, then mid-level features such as textures, and then high-level features such as object parts. The last network layer learns classification logic based on these learned abstract feature representations.
- The higher the layer and the more accurate the learned features, the more favorable the classification of the classifier is, thereby obtaining better performance.
- From the perspective of representation learning, convolutional neural networks extract features layer by layer, and the process of network training can be considered as a feature learning process.
- Based on the learned high-level abstract features, classification tasks can be conveniently performed. Applying the idea of representation learning, a well-trained convolutional neural network can often learn better features.

Here's a breakdown of how this typically works:

- Convolutional Layers: CNNs use convolutional layers to apply filters (kernels) to input data, which helps detect local patterns and features. These convolutional operations allow the network to learn low-level features like edges, textures, and basic shapes.

- Pooling Layers: Pooling layers are often used after convolutional layers to reduce the spatial dimensions of the data and retain the most important information. Max pooling, for example, selects the maximum value from a group of values in a local region, further emphasizing significant features.

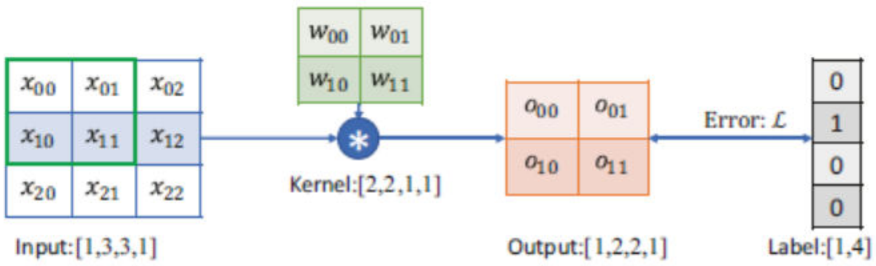
- High-level Features: As the data passes through multiple convolutional and pooling layers, the network learns to combine low-level features into higher-level representations. These higher-level features capture more complex patterns and relationships in the input data.

- Fully Connected Layers: After the convolutional and pooling layers, CNNs often have one or more fully connected layers. These layers further refine the learned representations and are typically used for the final classification or regression tasks.

  The key advantage of representation learning in CNNs is that the network automatically discovers relevant features from the raw input data during the training process. The hierarchical nature of these features allows the model to understand increasingly abstract and complex information, ultimately leading to improved performance on tasks like image recognition.

GRADIENT PROPOGATION IN CNN:

Consider a simple case where the input is a 3×3 single-channel matrix, and a 2×2 convolution kernel is used to perform the convolution operation. We then calculate the error between the flattened output and the corresponding label, as shown:



First derive the expression of the output tensor *O*:

$$o_{00} = x_{00}w_{00} + x_{01}w_{01} + x_{10}w_{10} + x_{11}w_{11} + b$$

$$o_{01} = x_{01}w_{00} + x_{02}w_{01} + x_{11}w_{10} + x_{12}w_{11} + b$$

$$o_{10} = x_{10}w_{00} + x_{11}w_{01} + x_{20}w_{10} + x_{21}w_{11} + b$$

$$o_{11} = x_{11}w_{00} + x_{12}w_{01} + x_{21}w_{10} + x_{22}w_{11} + b$$

Taking w_{00} gradient calculation as an example, decompose by chain rule:

$$\frac{\partial L}{\partial w_{00}} = \sum_{i\in\{00,01,10,11\}} \frac{\partial L}{\partial o_i}\frac{\partial o_i}{\partial w_{00}}$$

Let's consider:

$$\frac{\partial o_{00}}{\partial w_{00}} = \frac{\partial\left(x_{00}w_{00} + x_{01}w_{01} + x_{10}w_{10} + x_{11}w_{11} + b\right)}{w_{00}} = x_{00}$$

$$\frac{\partial o_{01}}{\partial w_{00}} = \frac{\partial\left(x_{01}w_{00} + x_{02}w_{01} + x_{11}w_{10} + x_{12}w_{11} + b\right)}{w_{00}} = x_{01}$$

$$\frac{\partial o_{10}}{\partial w_{00}} = \frac{\partial\left(x_{10}w_{00} + x_{11}w_{01} + x_{20}w_{10} + x_{21}w_{11} + b\right)}{w_{00}} = x_{10}$$

$$\frac{\partial o_{11}}{\partial w_{00}} = \frac{\partial\left(x_{11}w_{00} + x_{12}w_{01} + x_{21}w_{10} + x_{22}w_{11} + b\right)}{w_{00}} = x_{11}$$

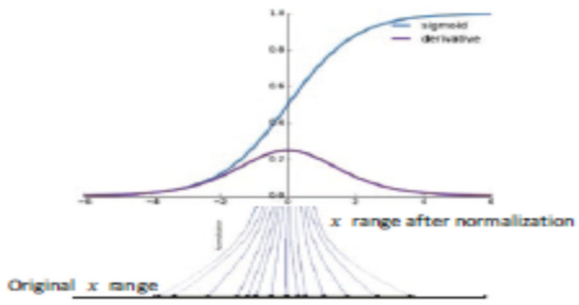
NOTE : It can be observed that the method of cyclically moving the receptive field does not change the derivatization of the network layer, and the derivation of the gradient is not complicated. But when the number of network layers increases, the artificial gradient derivation will become very cumbersome. But don't worry, the deep learning framework can help us automatically complete the gradient calculation and update of all parameters, we only need to design the network structure.

BATCHNORM LAYER:

With the advent of convolutional neural networks, the amount of network parameters has been greatly reduced, making it possible for deep networks with dozens of layers. However, before the emergence of the residual network, the increasing number of neural network layers makes the training very unstable, and sometimes the network does not update or even does not converge for a long time. At the same time, the network is more sensitive to hyperparameters, and the slight change of hyperparameters will change training trajectory of the network completely.

In 2015, Google researcher Sergey Ioffe et al. proposed a method of parameter normalization and designed the Batch Normalization (abbreviated as BatchNorm, or BN) layer. The proposal of the BN layer makes the setting of network hyperparameters more free, such as a largerlearning rate, and more random network initialization. In the meantime, the network has a faster convergence speed and better performance.

An example of the benefit of normalization: Consider the Sigmoid activation function and its gradient distribution. As shown in Figure the derivative value of the Sigmoid function in the interval $x \in [-2, 2]$ is distributed in the interval [0.1, 0.25]. When x > 2 or x < -2, the derivative of the Sigmoid function becomes very small, approaching 0, which is prone to gradient dispersion. In order to avoid the gradient dispersion phenomenon of the Sigmoid function due to too large or too small input, it is very important to normalize the function input to a small interval near 0. It can be seen from Figure, that after normalization, the value is mapped near 0, and the derivative value here is not too small, so that gradient dispersion is not easy to appear.



Implementation of BN layer:
Take the network model of LeNet-5 as an example, add the BN layer after the convolutional layer; the code is as follows:

```
network = Sequential([
layers.Conv2D(6,kernel_size=3,strides=1),
# Insert BN layer
layers.BatchNormalization(),
layers.MaxPooling2D(pool_size=2,strides=2),
layers.ReLU(),
layers.Conv2D(16,kernel_size=3,strides=1),
# Insert BN layer
layers.BatchNormalization(),
layers.MaxPooling2D(pool_size=2,strides=2),
layers.ReLU(),
layers.Flatten(),
layers.Dense(120, activation='relu'),
layers.Dense(84, activation='relu'),
layers.Dense(10)
])
```

In the training phase, you need to set the network parameter training=True to distinguish whether the BN layer is a training or testing model. The code is as follows:

```
with tf.GradientTape() as tape:
# Insert channel dimension
x = tf.expand_dims(x,axis=3)
# Forward calculation, [b, 784] => [b, 10]
out = network(x, training=True)
```

In the testing phase, you need to set training=False to avoid wrong behavior in the BN layer. The code is as follows:

```
for x,y in db_test:
# Insert channel dimension
x = tf.expand_dims(x,axis=3)
# Forward calculation
```

```
out = network(x, training=False)
```

CLASSICAL CONVOLUTIONAL NETWORKS:

- The network models before the emergence of AlexNet were all shallow neural networks, and the Top-5 error rate was above 25%.
- The AlexNet 8-layer deep neural network reduced the Top-5 error rate to 16.4%, and the performance was greatly improved.
- The subsequent VGG and GoogleNet models continued to reduce the error rate to 6.7%.
- The emergence of ResNet increased the number of network layers to 152 layers for the first time. The error rate is also reduced to 3.57%.

ALEXNET:
AlexNet is a deep convolutional neural network architecture that gained significant attention and popularity after winning the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012. It was developed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. The ImageNet competition involves classifying images into one of 1,000 object categories, and AlexNet demonstrated a remarkable improvement in accuracy over previous approaches.

Here are some key features of the AlexNet architecture:

Architecture:

- Convolutional Layers:

AlexNet consists of five convolutional layers. The convolutional layers use small receptive fields (typically 3x3) and are followed by max-pooling layers to downsample the spatial dimensions.

- Activation Function:

Rectified Linear Unit (ReLU) is used as the activation function throughout the network. ReLU helps mitigate the vanishing gradient problem and accelerates the convergence of the training process.

- Local Response Normalization (LRN):

LRN is applied after the first and second convolutional layers. It normalizes the responses across nearby channels to enhance the contrast between activations.

- Fully Connected Layers:

After the convolutional layers, there are three fully connected layers with 4096 neurons each. The last fully connected layer outputs predictions for the 1,000 ImageNet classes.

- Dropout:

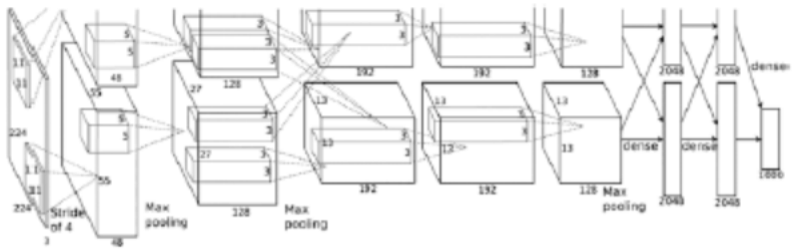
Dropout is applied to the first two fully connected layers during training. It helps prevent overfitting by randomly setting a fraction of the neurons to zero at each update.

- Softmax Activation:

The final layer uses the softmax activation function to convert the network's raw output into probability scores for each class.

Architecture Summary:
Input: 224x224x3 (RGB image)
Convolutional Layers: 5
Fully Connected Layers: 3
Output: 1,000 classes (ImageNet)
Training:
AlexNet was trained on a large dataset, ImageNet, which consists of millions of labeled images.
It utilized data augmentation and dropout to improve generalization.
Impact:
AlexNet significantly contributed to the resurgence of interest in neural networks and deep learning.

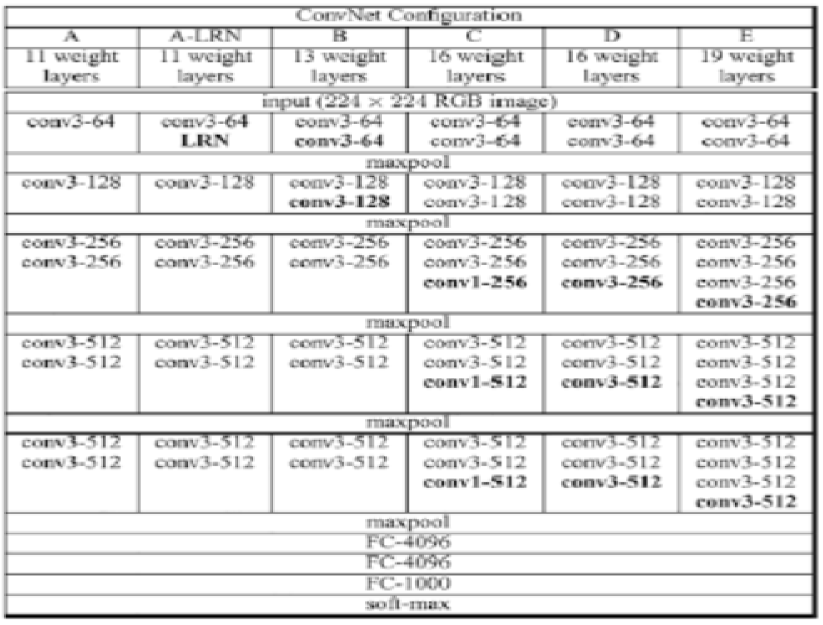
Its success demonstrated the effectiveness of deep learning for image classification tasks.



VGG SERIES:
he VGG (Visual Geometry Group) network is a series of convolutional neural network architectures that was developed by the Visual Geometry Group at the University of Oxford. The VGG architectures are known for their simplicity and uniform structure, consisting mainly of repeated blocks of convolutional layers with small 3x3 filters and max-pooling layers. This architecture makes VGG easy to understand and implement.

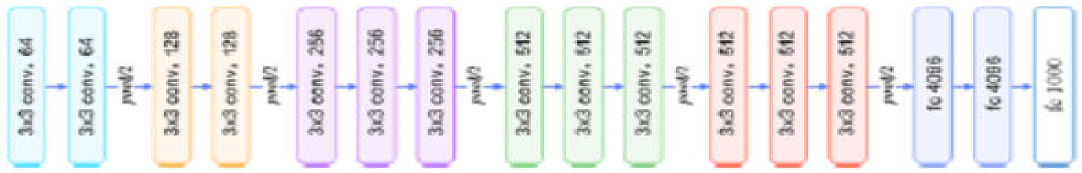
There are several variations of the VGG architecture, with the most commonly referenced ones being VGG16 and VGG19. The numbers in their names indicate the number of weight layers, including both convolutional and fully connected layers.

Key Characteristics of VGG Networks:
Simple Architecture:
VGG networks have a straightforward architecture with repeated blocks of convolutional layers. The convolutional layers use small 3x3 filters, and the max-pooling layers downsample the spatial dimensions.
Stacked Convolutional Layers:
The VGG architecture consists of stacking multiple convolutional layers with small receptive fields. This helps the network capture complex patterns.
Uniform Design:
The architecture follows a consistent design pattern. Convolutional blocks with the same number of filters are repeated, making it easy to understand and implement.
Fully Connected Layers:
The convolutional layers are followed by one or more fully connected layers. The final fully connected layer produces the network's output.
ReLU Activation:
Rectified Linear Unit (ReLU) is used as the activation function throughout the network.
Dropout:
Some variations of VGG, especially VGG16, include dropout in the fully connected layers to prevent overfitting during training.



ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

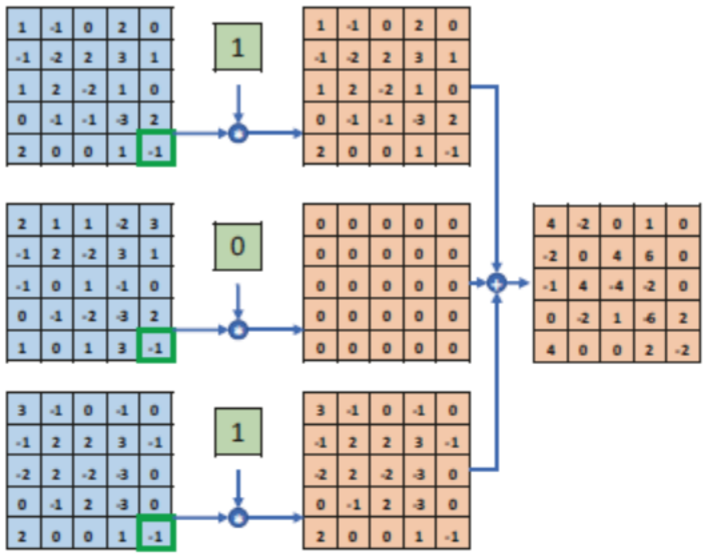
VGG16 ARCHITECTURE: It accepts color picture data with size of 224 × 224, and then passes through 2 Conv- Conv-Pooling units and 3 Conv-Conv-Conv-Pooling units, and finally outputs the probability of current picture belonging to 1000 categories through a 3 fully connected layers



Input: 224x224x3 (RGB image)
Convolutional Layers: 13 (with max-pooling)
Fully Connected Layers: 3
Output: 1,000 classes (ImageNet)

GoogLeNet:

1x1 kernel example:
The input is a threechannel 5x5 picture, and the convolution operation is performed with a single 1x1 convolution kernel. The data of each channel is calculated with the convolution kernel of the corresponding channel to obtain the intermediate matrix of the three channels, and the corresponding positions are added to get the final output tensor. For the input shape of [b, h, w, c_{in}], the output of the 1x1 convolutional layer is [b, h, w, c_{out}], where c_{in} is the number of channels of input data, c_{out} is the number of channels of output data, and is also the number of 1x1 convolution kernels. A special feature of the 1x1 convolution kernel is that it can only transform the number of channels without changing the width and height of the feature map.

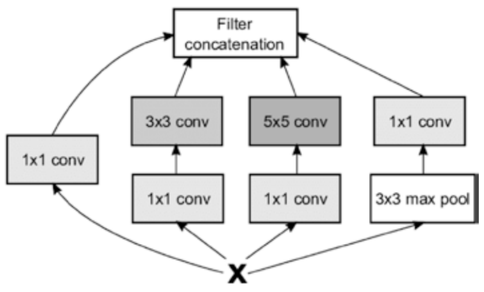


GoogleNet, also known as **Inception V1**, is a convolutional neural network architecture developed by researchers at Google. It was designed to address the challenge of training deep neural networks with a large number of parameters while maintaining computational efficiency. GoogleNet achieved this by introducing the concept of "inception modules" that allow the network to perform parallel processing at different spatial resolutions.

Key Features of GoogleNet:

- **Inception Modules:**

  The inception module consists of parallel convolutional and pooling operations, and the outputs of these operations are concatenated along the depth dimension. The primary components of the inception module include:



1x1 Convolutional Layer:
A 1x1 convolutional layer with a smaller number of filters is used to perform dimensionality reduction. It helps reduce the number of input channels and control the depth of the network.
3x3 Convolutional Layer:
A 3x3 convolutional layer captures spatial hierarchies and complex patterns. This filter size is effective at capturing medium-sized features in the input.
5x5 Convolutional Layer:
A 5x5 convolutional layer captures larger spatial features. It helps the network learn to recognize more global patterns in the input.
Max Pooling Layer:
Max pooling is applied to capture the most salient features in a local region. It helps with translation invariance and reduces spatial dimensions.
Concatenation:
The outputs of the 1x1, 3x3, 5x5 convolutions, and max pooling are concatenated along the depth dimension. This concatenated output becomes the input to the next layer.
The inception module allows the network to efficiently learn features at different scales without having to choose a specific filter size. This flexibility is beneficial for capturing information at various levels of abstraction

- **Global Average Pooling:**

Instead of fully connected layers at the end of the network, GoogleNet uses global average pooling. This involves taking the average of each feature map in the last convolutional layer across its spatial dimensions. Global average pooling reduces the number of parameters and helps with overfitting.

- **Auxiliary Classifiers**:

GoogleNet includes auxiliary classifiers at intermediate layers during training. These auxiliary classifiers are additional softmax layers and help with the training of earlier layers by providing additional supervision signals.

- **1x1 Convolutions:**

1x1 convolutions are used to reduce the number of input channels and control the depth of the network. They are computationally efficient and help with feature extraction.

- **ReLU Activation:**

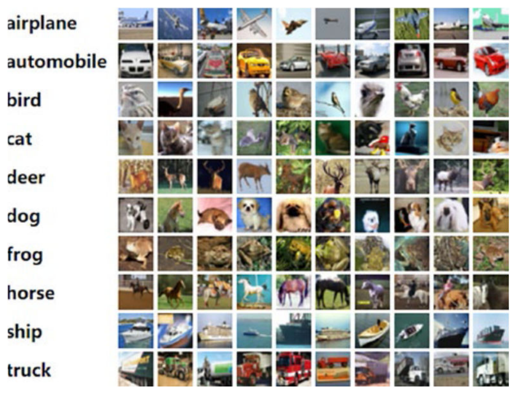
Rectified Linear Unit (ReLU) is used as the activation function throughout the network.

Architecture Summary:
Input: Variable size (commonly 224x224x3 for ImageNet)
Inception Modules: Multiple, with varying filter sizes
Global Average Pooling: Instead of fully connected layers
Auxiliary Classifiers: Included during training
Output: 1,000 classes (ImageNet)
Usage:
GoogleNet was designed for image classification tasks and participated in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC).
Impact:

GoogleNet demonstrated that deep networks with a large number of parameters could be trained efficiently.
The inception module idea has inspired subsequent architectures, including later versions of Inception (V2, V3, and V4).

HANDSON CIFAR10 &VGG13:
CIFAR-10 stands for the Canadian Institute for Advanced Research (CIFAR) 10 dataset. It is a well-known and widely used benchmark dataset in the field of computer vision and machine learning.
CIFAR-10 consists of a diverse set of 60,000 color images, each of size 32x32 pixels, belonging to 10 different classes. The dataset is split into 50,000 training images and 10,000 testing images.

The 10 classes in CIFAR-10 are:



Airplane
Automobile
Bird
Cat
Deer
Dog
Frog
Horse
Ship
Truck
Each class contains 6,000 images in total (5,000 in the training set and 1,000 in the test set). The dataset is designed to be challenging due to the small size of the images, low resolution, and the diversity of object classes.

CODE:
```
# Load CIFAR10 data set
(x,y), (x_test, y_test) = datasets.cifar10.load_data()
# Delete one dimension of y, [b,1] => [b]
y = tf.squeeze(y, axis=1)
y_test = tf.squeeze(y_test, axis=1)
# Print the shape of training and testing sets
print(x.shape, y.shape, x_test.shape, y_test.shape)
# Create training set and preprocess
train_db = tf.data.Dataset.from_tensor_slices((x,y))
train_db = train_db.shuffle(1000).map(preprocess).batch(128)
# Create testing set and preprocess
test_db = tf.data.Dataset.from_tensor_slices((x_test,y_test))
test_db = test_db.map(preprocess).batch(128)
# Select a Batch
sample = next(iter(train_db))
print('sample:', sample[0].shape, sample[1].shape,tf.reduce_min(sample[0]),
tf.reduce_max(sample[0]))
```

we will modify the VGG13 network structure according to the characteristics of our data set to complete CIFAR10 image recognition as follows:
• Adjust the network input to 32 × 32. The original network input is 224 × 224, resulting in too large input feature dimensions and too large network parameters.
• The dimensions of the three fully connected layers are [256,64,10] for the setting of ten classification tasks.

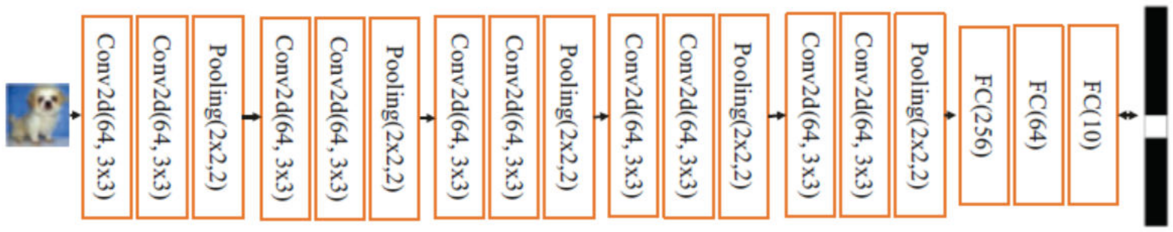


***Figure 10-50.** Adjusted VGG13 model structure*

```
conv_layers = [
# Conv-Conv-Pooling unit 1
# 64 3x3 convolutional kernels with same input and
output size
layers.Conv2D(64, kernel_size=[3, 3], padding="same",
activation=tf.nn.relu), layers.Conv2D(64, kernel_size=[3, 3], padding="same",
activation=tf.nn.relu),
# Reduce the width and height size to half of its original
layers.MaxPool2D(pool_size=[2, 2], strides=2,
padding='same'),
# Conv-Conv-Pooling unit 2, output channel increases to
128, half width and height
layers.Conv2D(128, kernel_size=[3, 3], padding="same",
activation=tf.nn.relu),
layers.Conv2D(128, kernel_size=[3, 3], padding="same",
activation=
tf.nn.relu),
layers.MaxPool2D(pool_size=[2, 2], strides=2,
padding='same'),
# Conv-Conv-Pooling unit 3, output channel increases to
256, half width and height
layers.Conv2D(256, kernel_size=[3, 3], padding="same",
activation=tf.nn.relu),
layers.Conv2D(256, kernel_size=[3, 3], padding="same",
activation=tf.nn.relu),
layers.MaxPool2D(pool_size=[2, 2], strides=2,
padding='same'),
# Conv-Conv-Pooling unit 4, output channel increases to
512, half width and height
layers.Conv2D(512, kernel_size=[3, 3], padding="same",
activation=tf.nn.relu),
layers.Conv2D(512, kernel_size=[3, 3], padding="same",
activation=tf.nn.relu),
layers.MaxPool2D(pool_size=[2, 2], strides=2,
padding='same'), # Conv-Conv-Pooling unit 5, output channel increases to
512, half width and height
layers.Conv2D(512, kernel_size=[3, 3], padding="same",
activation=tf.nn.relu),
layers.Conv2D(512, kernel_size=[3, 3], padding="same",
activation=tf.nn.relu),
layers.MaxPool2D(pool_size=[2, 2], strides=2,
padding='same')
]
conv_net = Sequential(conv_layers)
```

```
# Create 3 fully connected layer sub-network
fc_net = Sequential([
layers.Dense(256, activation=tf.nn.relu),
layers.Dense(128, activation=tf.nn.relu),
layers.Dense(10, activation=None),
])
# build network and print parameter info
conv_net.build(input_shape=[4, 32, 32, 3])
fc_net.build(input_shape=[4, 512])
conv_net.summary()
fc_net.summary()
# merge parameters of two sub-networks
variables = conv_net.trainable_variables + fc_net.trainable_
variables
# calculate gradient for all parameters
grads = tape.gradient(loss, variables)
# update gradients
optimizer.apply_gradients(zip(grads, variables))
```

CONVOLUTIONAL LAYER VARIANTS:

Standard Convolutional Layer:

The standard convolutional layer applies a set of filters to the input data. Each filter is convolved with the input, producing feature maps. The operation involves sliding the filter over the input and computing dot products at each location. The result is passed through an activation function (commonly ReLU).

Dilated (Atrous) Convolution:

Dilated convolution introduces gaps between the elements of the filter. The dilation rate determines the spacing between the values in the filter. This allows the convolutional layer to have a larger receptive field without increasing the number of parameters, making it effective for capturing long-range dependencies in the data.

Depthwise Separable Convolution:

Depthwise separable convolution decomposes the standard convolution into two steps:
Depthwise Convolution: Applies a separate filter to each input channel independently.
Pointwise Convolution: Applies a 1x1 convolution to combine the outputs from the depthwise convolution. This reduces computational cost and parameter count, making it efficient for mobile and edge devices.

Transposed Convolution (Deconvolution):

Transposed convolution is used for upsampling. It helps increase the spatial resolution of the feature maps. In this operation, the filter is applied to the input with gaps, effectively "spreading" the information. It is often used in generative models, image segmentation, and other tasks where upsampling is required.

Grouped Convolution:
Grouped convolution divides the input channels into groups, and each group is convolved with a subset of filters. This helps reduce the computational complexity and encourages feature diversity. Grouped convolution is commonly used in models like AlexNet.

Spatial Separable Convolution:

Spatial separable convolution factorizes the standard 2D convolution into two 1D convolutions. It performs a row-wise convolution followed by a column-wise convolution. This reduces the number of parameters and computational cost while capturing spatial dependencies effectively.

Octave Convolution:

Octave convolution processes input data at different spatial resolutions simultaneously. It involves operations at both high and low-frequency bands. This allows the network to capture features at different scales and is particularly useful for handling objects of varying sizes within the input data.

Attention-based Convolution:

Attention mechanisms can be incorporated into convolutional layers to dynamically adjust the weights of different spatial locations based on their importance. This enhances the network's ability to focus on relevant regions, particularly useful in tasks where certain parts of the input are more informative than others.

Mixed Convolutional Layers:

Some architectures, like the Inception module, use a mix of convolutional layers with different filter sizes (1x1, 3x3, 5x5). This allows the network to capture features at various spatial scales simultaneously, improving its ability to learn diverse patterns.

These convolutional layer variants are designed to address specific challenges and constraints in different applications, offering flexibility and efficiency improvements for various tasks. Choosing the appropriate variant depends on the specific requirements of the problem at hand and the available computational resources.

DEEP RESIDUAL NETWORK:

A Deep Residual Network, commonly known as ResNet, is a type of neural network architecture designed to overcome the challenges of training very deep neural networks. It was introduced by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun in their paper titled "Deep Residual Learning for Image Recognition," presented at the 2016 Conference on Computer Vision and Pattern Recognition (CVPR).

Motivation for ResNet:

The primary motivation for ResNet arises from the observation that as the depth of a neural network increases, the training accuracy often saturates and then degrades rapidly. This phenomenon is known as the vanishing gradient problem, where the gradients become extremely small during backpropagation, making it challenging for the network to learn meaningful representations.

Key Features of ResNet:

Residual Blocks:

The core innovation in ResNet is the use of residual blocks. A residual block contains a shortcut connection, or a skip connection, that bypasses one or more layers. The idea is to learn the residual, or the difference between the input and the output of the block. This allows the network to focus on learning the residual information, making it easier to train deep networks.

Identity Shortcut Connections:

The most common form of the shortcut connection in ResNet is an identity mapping. The input to a residual block is added directly to the output of the block. Mathematically, if x is the input and F(x) is the output of the block, the residual block computes F(x)+x. This allows the gradients to flow directly through the identity mapping during backpropagation.

Bottleneck Architecture:

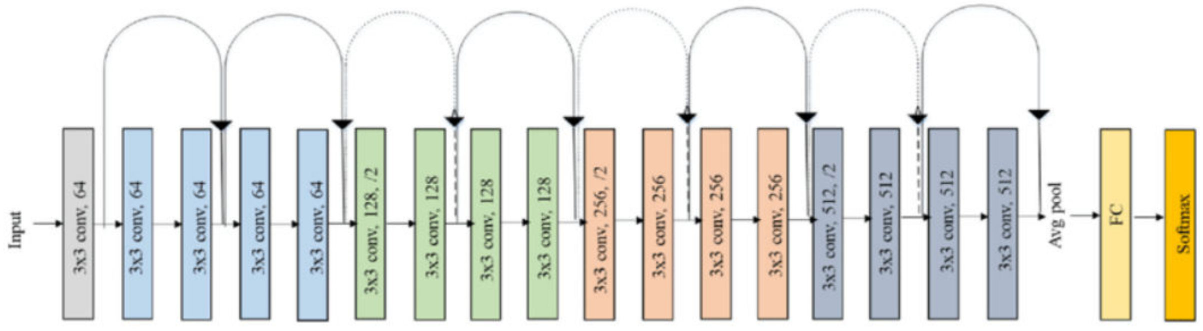
ResNet introduces a bottleneck architecture in which each residual block consists of three convolutional layers: 1x1, 3x3, and 1x1. The 1x1 convolutions are used to reduce and then increase the dimensionality of the data, making the network more computationally efficient.

Global Average Pooling (GAP):

ResNet typically uses global average pooling instead of fully connected layers at the end of the network. GAP computes the average of each feature map, resulting in a fixed-size vector regardless of the spatial dimensions of the input. This reduces the number of parameters and helps prevent overfitting.

Batch Normalization:

Batch normalization is commonly used in ResNet to normalize the activations and stabilize training. It helps in mitigating issues like internal covariate shift and allows for more straightforward training of very deep networks.



Original ResNet-18 Architecture

ResNet Variants:
ResNet has several variants, including ResNet-18, ResNet-34, ResNet-50, ResNet-101, and ResNet-152, which differ in terms of the number of layers. The larger variants often achieve higher accuracy but require more computational resources.
Impact:
ResNet has had a significant impact on the field of computer vision and deep learning, enabling the training of very deep networks that were previously challenging to optimize. It has become a common architecture for image classification, object detection, and other computer vision tasks.

```
import tensorflow as tf
from tensorflow.keras import layers, Model

def residual_block(x, filters, kernel_size=3, stride=1):
    y = layers.Conv2D(filters, kernel_size=kernel_size, strides=stride, padding='same')(x)
    y = layers.BatchNormalization()(y)
    y = layers.Activation('relu')(y)

    y = layers.Conv2D(filters, kernel_size=kernel_size, strides=1, padding='same')(y)
    y = layers.BatchNormalization()(y)

    if stride != 1 or x.shape[-1] != filters:
        x = layers.Conv2D(filters, kernel_size=1, strides=stride, padding='same')(x)
        x = layers.BatchNormalization()(x)

    y = layers.add([x, y])
    y = layers.Activation('relu')(y)
    return y

def build_resnet18(input_shape=(224, 224, 3), num_classes=1000):
    input_tensor = layers.Input(shape=input_shape)

    x = layers.Conv2D(64, kernel_size=7, strides=2, padding='same')(input_tensor)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
    x = layers.MaxPooling2D(pool_size=3, strides=2, padding='same')(x)

    x = residual_block(x, filters=64)
    x = residual_block(x, filters=64)
    x = residual_block(x, filters=128, stride=2)
    x = residual_block(x, filters=128)
    x = residual_block(x, filters=256, stride=2)
    x = residual_block(x, filters=256)
    x = residual_block(x, filters=512, stride=2)
    x = residual_block(x, filters=512)

    x = layers.GlobalAveragePooling2D()(x)
    output_tensor = layers.Dense(num_classes, activation='softmax')(x)

    model = Model(inputs=input_tensor, outputs=output_tensor)
    return model

# Instantiate the model
resnet18 = build_resnet18()

# Display the model summary
resnet18.summary()
```

DENSENET:
DenseNet, short for Densely Connected Convolutional Networks, is a neural network architecture introduced by Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger in their paper "Densely Connected Convolutional Networks," presented at CVPR 2017. DenseNet is known for its dense connectivity pattern, where each layer receives input from all preceding layers, creating dense connections within the network.

Key Features of DenseNet:

Dense Connectivity:
In DenseNet, each layer is connected to every other layer in a feed-forward fashion. This dense connectivity pattern facilitates the flow of information and gradients throughout the network. Each layer receives the feature maps from all preceding layers as input, including its own feature map.

Dense Blocks:
Dense connectivity is achieved through the use of dense blocks. A dense block consists of multiple layers, and each layer produces a set of feature maps. These feature maps are concatenated and serve as input for subsequent layers within the same dense block. The dense block structure enables feature reuse and enhances the learning capability of the network.

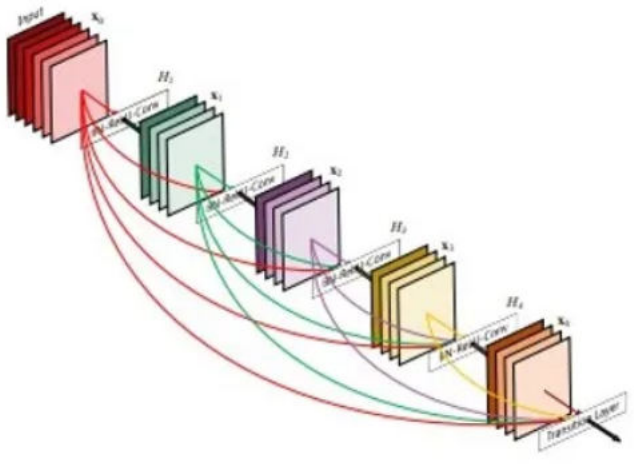
Bottleneck Layers:
DenseNet often includes bottleneck layers within dense blocks. A bottleneck layer is a composite layer with a combination of 1x1 and 3x3 convolutions. It helps reduce the dimensionality of the feature maps, making the network more computationally efficient.

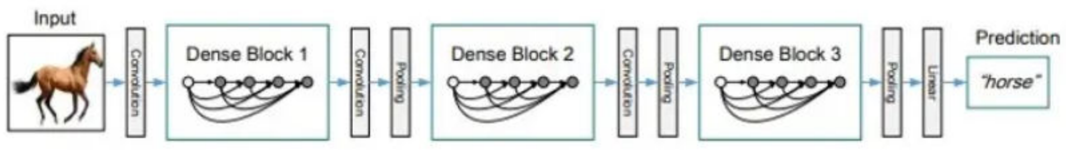
Transition Layers:
Between dense blocks, transition layers are used to downsample the spatial dimensions of the feature maps. Transition layers typically consist of a 1x1 convolutional layer followed by average pooling. They help control the number of parameters and computational cost.

Global Average Pooling (GAP):
Similar to ResNet, DenseNet often uses global average pooling instead of fully connected layers at the end of the network. GAP computes the average of each feature map, resulting in a fixed-size vector regardless of the spatial dimensions. This further reduces the number of parameters and helps prevent overfitting.

Batch Normalization and ReLU Activation:
Batch normalization and ReLU activation functions are commonly used throughout the network to stabilize training and introduce non-linearity.

DenseNet Architecture:
The overall architecture of DenseNet consists of multiple dense blocks separated by transition layers. The final layer is a global average pooling layer followed by a dense layer with a softmax activation for classification.





```
import tensorflow as tf
from tensorflow.keras import layers, Model

def dense_block(x, blocks, growth_rate):
    for _ in range(blocks):
        y = layers.BatchNormalization()(x)
        y = layers.Activation('relu')(y)
        y = layers.Conv2D(4 * growth_rate, 1, padding='same')(y)
        y = layers.BatchNormalization()(y)
        y = layers.Activation('relu')(y)
        y = layers.Conv2D(growth_rate, 3, padding='same')(y)

        x = layers.concatenate([x, y], axis=-1)

    return x

def transition_layer(x, reduction):
    filters = int(x.shape[-1] * reduction)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
    x = layers.Conv2D(filters, 1, padding='same')(x)
    x = layers.AveragePooling2D(2, strides=2)(x)
    return x

def build_densenet(input_shape=(224, 224, 3), num_classes=1000, blocks=[6, 12, 24, 16],
growth_rate=32, reduction=0.5):
    input_tensor = layers.Input(shape=input_shape)

    x = layers.Conv2D(64, 7, strides=2, padding='same')(input_tensor)
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
    x = layers.MaxPooling2D(3, strides=2, padding='same')(x)

    for block in blocks:
        x = dense_block(x, block, growth_rate)
        x = transition_layer(x, reduction)

    x = layers.GlobalAveragePooling2D()(x)
    output_tensor = layers.Dense(num_classes, activation='softmax')(x)

    model = Model(inputs=input_tensor, outputs=output_tensor)
    return model

# Instantiate the model
densenet = build_densenet()

# Display the model summary
densenet.summary()
```