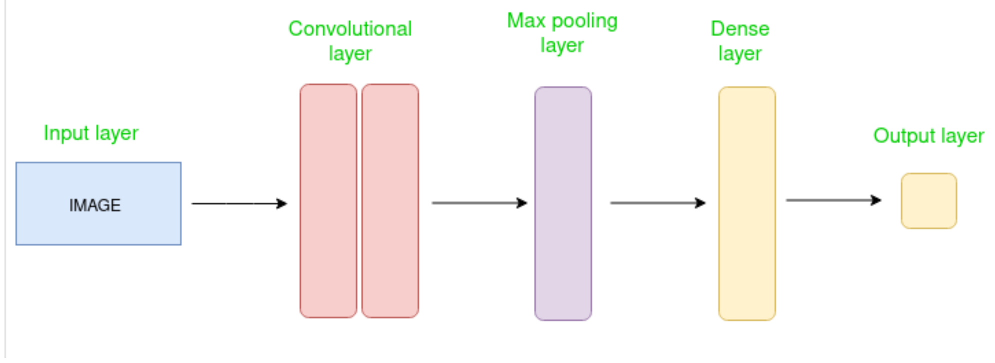
Unit 5:

A **Convolutional Neural Network (CNN)** is a type of Deep Learning neural network architecture commonly used in Computer Vision. Computer vision is a field of Artificial Intelligence that enables a computer to understand and interpret the image or visual data.
When it comes to Machine Learning, Artificial Neural Networks perform really well. Neural Networks are used in various datasets like images, audio, and text. Different types of Neural Networks are used for different purposes, for example for predicting the sequence of words we use **Recurrent Neural Networks** more precisely an LSTM, similarly for image classification we use Convolution Neural networks.

CNN Architecture

Convolutional Neural Network consists of multiple layers like the input layer, Convolutional layer, Pooling layer, and fully connected layers.

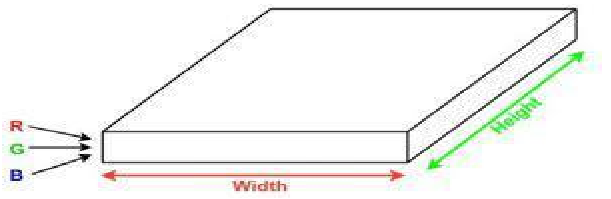


Simple CNN architecture

The Convolutional layer applies filters to the input image to extract features, the Pooling layer downsamples the image to reduce computation, and the fully connected layer makes the final prediction. The network learns the optimal filters through backpropagation and gradient descent.

How Convolutional Layers Works?

Convolution Neural Networks or covnets are neural networks that share their parameters. Imagine you have an image. It can be represented as a cuboid having its length, width (dimension of the image), and height (i.e the channel as images generally have red, green, and blue channels).



Now imagine taking a small patch of this image and running a small neural network, called a filter or kernel on it, with say, K outputs and representing them vertically. Now slide that neural network across the whole image, as a result, we will get another image with different widths, heights, and depths. Instead of just R, G, and B channels now we have more channels but lesser width and height. This operation is called **Convolution**. If the patch size is the same as that of the image it will be a regular neural network. Because of this small patch, we have fewer weights.

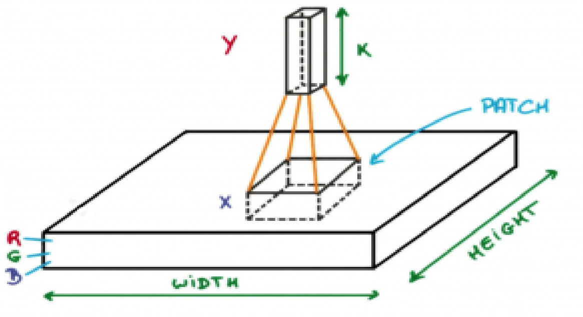


Image source: Deep Learning Udacity

Mathematical Overview of Convolution

Now let's talk about a bit of mathematics that is involved in the whole convolution process.

- Convolution layers consist of a set of learnable filters (or kernels) having small widths and heights and the same depth as that of input volume (3 if the input layer is image input).
- For example, if we have to run convolution on an image with dimensions 34x34x3. The possible size of filters can be axax3, where 'a' can be anything like 3, 5, or 7 but smaller as compared to the image dimension.
- During the forward pass, we slide each filter across the whole input volume step by step where each step is called **stride** (which can have a value of 2, 3, or even 4 for high-dimensional images) and compute the dot product between the kernel weights and patch from input volume.
- As we slide our filters we'll get a 2-D output for each filter and we'll stack them together as a result, we'll get output volume having a depth equal to the number of filters. The network will learn all the filters.

Layers Used to Build ConvNets

A complete Convolution Neural Networks architecture is also known as covnets. A covnets is a sequence of layers, and every layer transforms one volume to another through a differentiable function.

Types of layers: datasets

Let's take an example by running a covnets on of image of dimension 32 x 32 x 3.

- **Input Layers:** It's the layer in which we give input to our model. In CNN, Generally, the input will be an image or a sequence of images. This layer holds the raw input of the image with width 32, height 32, and depth 3.
- **Convolutional Layers:** This is the layer, which is used to extract the feature from the input dataset. It applies a set of learnable filters known as the kernels to the input images. The filters/kernels are smaller matrices usually 2×2, 3×3, or 5×5 shape. it slides over the input image data and computes the dot product between kernel weight and the corresponding input image patch. The output of this layer is referred as feature maps. Suppose we use a total of 12 filters for this layer we'll get an output volume of dimension 32 x 32 x 12. channel for each filter
- **Activation Layer:** By adding an activation function to the output of the preceding layer, activation layers add nonlinearity to the network. it will apply an element-wise activation function to the output of the convolution layer. Some common activation functions

are **RELU**: max(0, x), **Tanh**, **Leaky RELU**, etc. The volume remains unchanged hence output volume will have dimensions 32 x 32 x 12.

- **Pooling layer:** This layer is periodically inserted in the covnets and its main function is to reduce the size of volume which makes the computation fast reduces memory and also prevents overfitting. Two common types of pooling layers are **max pooling** and **average pooling**. If we use a max pool with 2 x 2 filters and stride 2, the resultant volume will be of dimension 16x16x12.

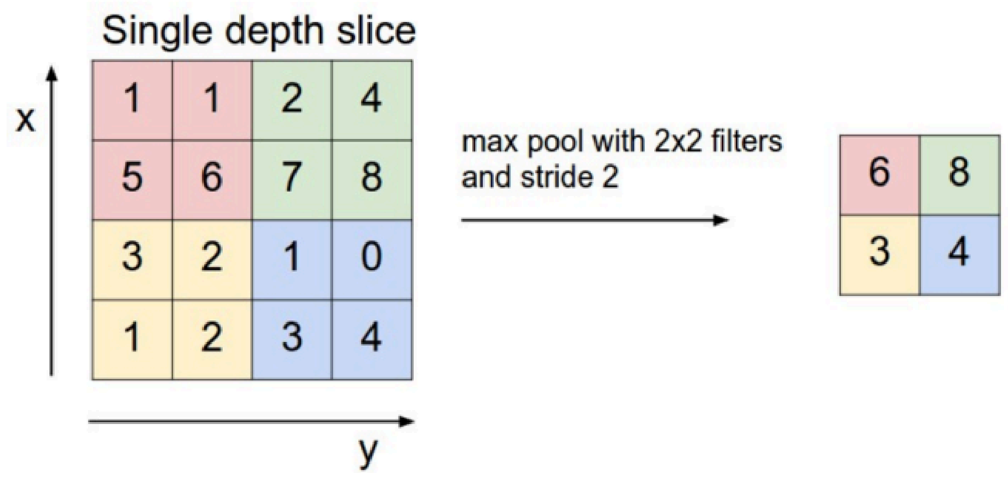


Image source: cs231n.stanford.edu

- **Flattening:** The resulting feature maps are flattened into a one-dimensional vector after the convolution and pooling layers so they can be passed into a completely linked layer for categorization or regression.
- **Fully Connected Layers:** It takes the input from the previous layer and computes the final classification or regression task.

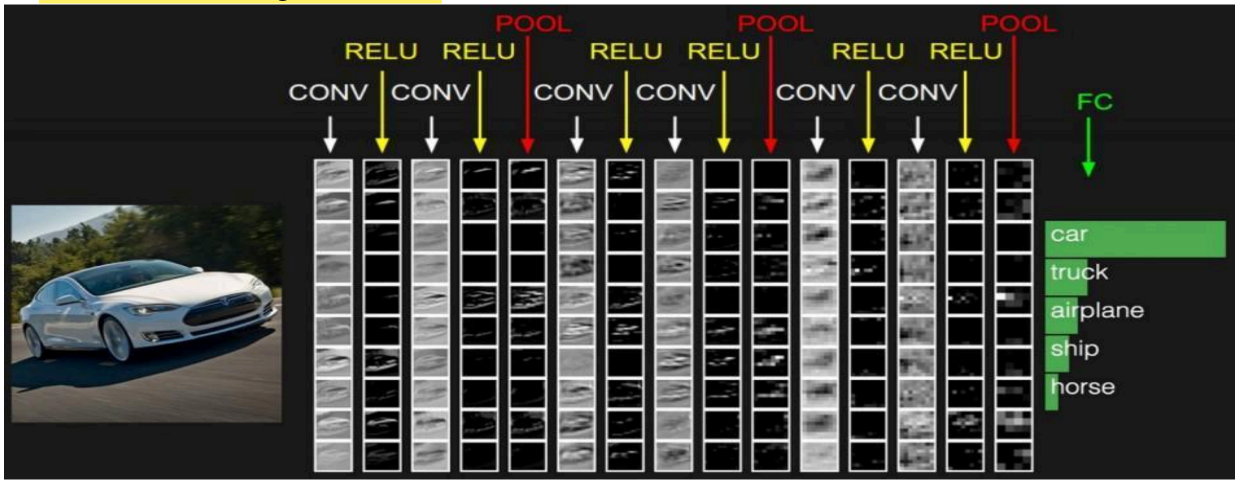


Image source: cs231n.stanford.edu

- **Output Layer:** The output from the fully connected layers is then fed into a logistic function for classification tasks like sigmoid or softmax which converts the output of each class into the probability score of each class.

Convolutional Neural Networks (CNNs) and different CNN architectures: Building blocks of CNN, Transfer Learning, LeNet, AlexNet, ZF-Net, VGGNet, GoogLeNet, ResNet, Visualizing CNNs, Guided Backpropagation.

can be asked in ppr:-

You need to build a CNN for an image classification task. Describe how you would preprocess the data, design the model, and evaluate its performance.

To build a Convolutional Neural Network (CNN) for an image classification task, here's a comprehensive outline that includes data preprocessing, model design, and performance evaluation.

1. Data Preprocessing

Data preprocessing is crucial to prepare the dataset for effective training. Here are the key steps:

- **Data Collection**: Gather a labeled dataset (e.g., CIFAR-10, MNIST, or your custom dataset).
- **Image Resizing**: Resize images to a uniform size that matches the input dimensions expected by the CNN (e.g., 224x224 pixels).
- **Normalization**: Scale pixel values to a range of [0, 1] or standardize to have a mean of 0 and a standard deviation of 1. This helps improve convergence during training.

```
python
Copy code
images = images / 255.0  # For scaling
```

- **Data Augmentation**: Apply techniques such as rotation, flipping, zooming, and brightness adjustments to artificially expand the dataset and improve model robustness.

```
python
Copy code
from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)
```

- **Train-Validation-Test Split**: Split the dataset into training, validation, and test sets (commonly 70%-20%-10% or similar ratios) to evaluate model performance accurately.

2. Model Design

Designing the CNN involves choosing the architecture and specifying layers. Here's a simple yet effective architecture:

- **Input Layer**: Specify the input shape based on preprocessed image dimensions (e.g., `(224, 224, 3)` for RGB images).
- **Convolutional Layers**: Use several convolutional layers with filters (e.g., 32, 64, 128) and kernel sizes (e.g., 3x3) followed by activation functions (e.g., ReLU).

```
python
Copy code
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(224, 224,
3)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
```

- **Flatten Layer**: Flatten the output from the convolutional layers to feed into fully connected layers.

```
python
Copy code
model.add(Flatten())
```

- **Dense Layers**: Add one or more dense layers with a suitable number of neurons, followed by activation functions (e.g., ReLU).

```
python
Copy code
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))  # Dropout for regularization
```

- **Output Layer**: Use a dense layer with a number of units equal to the number of classes and a softmax activation function for multi-class classification.

```
python
Copy code
model.add(Dense(num_classes, activation='softmax'))
```

- **Compile the Model**: Specify the optimizer (e.g., Adam), loss function (e.g., categorical cross-entropy for multi-class), and evaluation metrics (e.g., accuracy).

```
python
Copy code
```

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

3. Model Training

- **Fit the Model**: Train the model on the training set, validating on the validation set. Use callbacks like ModelCheckpoint and EarlyStopping to optimize training.

```
python
Copy code
history = model.fit(train_data, train_labels, validation_data=(val_data,
val_labels), epochs=50, batch_size=32, callbacks=[...])
```

4. Performance Evaluation

- **Evaluate on Test Set**: After training, assess the model's performance on the unseen test data.

```
python
Copy code
test_loss, test_accuracy = model.evaluate(test_data, test_labels)
print(f'Test Accuracy: {test_accuracy}')
```

- **Confusion Matrix**: Generate a confusion matrix to analyze model performance across different classes.

```
python
Copy code
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

predictions = model.predict(test_data)
predicted_classes = np.argmax(predictions, axis=1)
cm = confusion_matrix(test_labels, predicted_classes)
sns.heatmap(cm, annot=True)
plt.show()
```

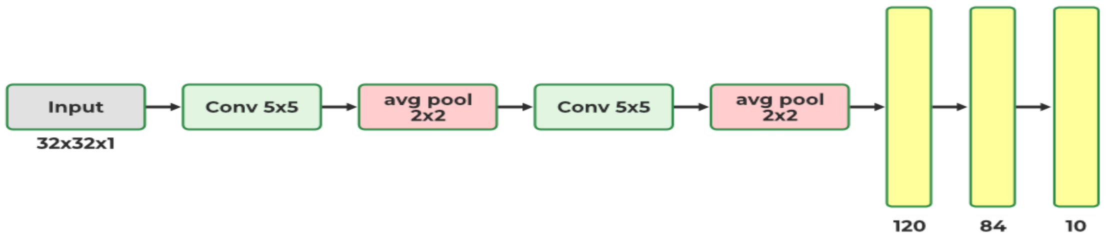
- **Classification Report**: Use metrics like precision, recall, and F1-score for a comprehensive evaluation.

```
python
Copy code
from sklearn.metrics import classification_report

print(classification_report(test_labels, predicted_classes))
```

Convolutional Neural Network(CNN) is a neural network architecture in Deep Learning, used to recognize the pattern from structured arrays. However, over many years, **CNN architectures have evolved**. Many variants of the fundamental CNN Architecture This been developed, leading to amazing advances in the growing deep-learning field.
Let's discuss, How CNN architecture developed and grow over time.

1. LeNet-5

- The First LeNet-5 architecture is the most widely known CNN architecture. It was introduced in 1998 and is widely used for handwritten method digit recognition.
- LeNet-5 has 2 convolutional and 3 full layers.
- This LeNet-5 architecture has 60,000 parameters.



LeNet-5

- The LeNet-5 has the ability to process higher one-resolution images that require larger and more CNN convolutional layers.
- The leNet-5 technique is measured by the availability of all computing resources

Example Model of LeNet-5

- Python3

```
import torch

from torchsummary import summary

import torch.nn as nn

import torch.nn.functional as F
```

```
class LeNet5(nn.Module):
    def __init__(self):
        # Call the parent class's init method
        super(LeNet5, self).__init__()
        # First Convolutional Layer
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, stride=1)
        # Max Pooling Layer
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        # Second Convolutional Layer
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, stride=1)
        # First Fully Connected Layer
        self.fc1 = nn.Linear(in_features=16 * 5 * 5, out_features=120)
        # Second Fully Connected Layer
        self.fc2 = nn.Linear(in_features=120, out_features=84)
        # Output Layer
        self.fc3 = nn.Linear(in_features=84, out_features=10)
    def forward(self, x):
        # Pass the input through the first convolutional layer and activation function
        x = self.pool(F.relu(self.conv1(x)))

        # Pass the output of the first layer through

        # the second convolutional layer and activation function

        x = self.pool(F.relu(self.conv2(x)))

        # Reshape the output to be passed through the fully connected layers

        x = x.view(-1, 16 * 5 * 5)

        # Pass the output through the first fully connected layer and activation function

        x = F.relu(self.fc1(x))

        # Pass the output of the first fully connected layer through

        # the second fully connected layer and activation function

        x = F.relu(self.fc2(x))

        # Pass the output of the second fully connected layer through the output layer

        x = self.fc3(x)

        # Return the final output

        return x



lenet5 = LeNet5()

print(lenet5)
```

Output:

```
LeNet5(
 (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))

 (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)

 (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))

 (fc1): Linear(in_features=400, out_features=120, bias=True)

 (fc2): Linear(in_features=120, out_features=84, bias=True)

 (fc3): Linear(in_features=84, out_features=10, bias=True)

)
```

Model Summary :

Print the summary of the lenet5 to check the params

- Python3

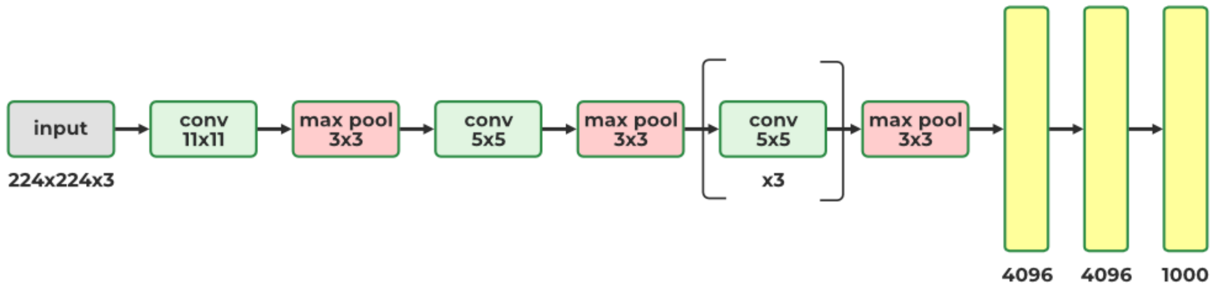
```
# add the cuda to the mode

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

lenet5.to(device)

#Print the summary of the model

summary(lenet5, (1, 32, 32))
```

Output:

```
----------------------------------------------------------------

        Layer (type)               Output Shape         Param #

================================================================

            Conv2d-1            [-1, 6, 28, 28]             156

         MaxPool2d-2            [-1, 6, 14, 14]               0

            Conv2d-3           [-1, 16, 10, 10]           2,416

         MaxPool2d-4             [-1, 16, 5, 5]               0

            Linear-5                  [-1, 120]          48,120
        Linear-6                [-1, 84]         10,164
        Linear-7                [-1, 10]            850
================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.06
Params size (MB): 0.24
Estimated Total Size (MB): 0.30
----------------------------------------------------------------
```

2. AlexNNet

- The AlexNet CNN architecture won the 2012 ImageNet ILSVRC challenges of deep learning algorithm by a large variance by achieving 17% with top-5 error rate as the second best achieved 26%!
- It was introduced by Alex Krizhevsky (name of founder), The Ilya Sutskever and Geoffrey Hinton are quite similar to LeNet-5, only much bigger and deeper and it was introduced first to stack convolutional layers directly on top of each other models, instead of stacking a pooling layer top of each on CN network convolutional layer.
- AlexNNet has 60 million parameters as AlexNet has total 8 layers, 5 convolutional and 3 fully connected layers.
- AlexNNet is first to execute (ReLUs) Rectified Linear Units as activation functions
- it was the first CNN architecture that uses GPU to improve the performance.



ALexNNet

Example Model of AlexNNet

- Python3

```
import torch

from torchsummary import summary

import torch.nn as nn

import torch.nn.functional as F

class AlexNet(nn.Module):

    def __init__(self, num_classes=1000):

        # Call the parent class's init method to initialize the base class

        super(AlexNet, self).__init__()

        # First Convolutional Layer with 11x11 filters, stride of 4, and 2 padding

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=96, kernel_size=11, stride=4,
padding=2)



        # Max Pooling Layer with a kernel size of 3 and stride of 2

        self.pool = nn.MaxPool2d(kernel_size=3, stride=2)

        # Second Convolutional Layer with 5x5 filters and 2 padding

        self.conv2 = nn.Conv2d(in_channels=96, out_channels=256, kernel_size=5, padding=2)


        # Third Convolutional Layer with 3x3 filters and 1 padding
        self.conv3 = nn.Conv2d(in_channels=256, out_channels=384, kernel_size=3,
padding=1)

        # Fourth Convolutional Layer with 3x3 filters and 1 padding

        self.conv4 = nn.Conv2d(in_channels=384, out_channels=384, kernel_size=3,
padding=1)

        # Fifth Convolutional Layer with 3x3 filters and 1 padding

        self.conv5 = nn.Conv2d(in_channels=384, out_channels=256, kernel_size=3,
padding=1)

        # First Fully Connected Layer with 4096 output features

        self.fc1 = nn.Linear(in_features=256 * 6 * 6, out_features=4096)

        # Second Fully Connected Layer with 4096 output features

        self.fc2 = nn.Linear(in_features=4096, out_features=4096)

        # Output Layer with `num_classes` output features

        self.fc3 = nn.Linear(in_features=4096, out_features=num_classes)

    def forward(self, x):

        # Pass the input through the first convolutional layer and ReLU activation function

        x = self.pool(F.relu(self.conv1(x)))

        # Pass the output of the first layer through

        # the second convolutional layer and ReLU activation function

        x = self.pool(F.relu(self.conv2(x)))
```

```
# Pass the output of the second layer through

# the third convolutional layer and ReLU activation function

x = F.relu(self.conv3(x))

# Pass the output of the third layer through

# the fourth convolutional layer and ReLU activation function

x = F.relu(self.conv4(x))

# Pass the output of the fourth layer through

# the fifth convolutional layer and ReLU activation function

x = self.pool(F.relu(self.conv5(x)))

# Reshape the output to be passed through the fully connected layers

x = x.view(-1, 256 * 6 * 6)

# Pass the output through the first fully connected layer and activation function

x = F.relu(self.fc1(x))

x = F.dropout(x, 0.5)

# Pass the output of the first fully connected layer through

# the second fully connected layer and activation function

x = F.relu(self.fc2(x))

# Pass the output of the second fully connected layer through the output layer
```

```
        x = self.fc3(x)

        # Return the final output

        return x

alexnet = AlexNet()

print(alexnet)
```

Output:

```
AlexNet(
  (conv1): Conv2d(3, 96, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
  (pool): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(96, 256, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (conv3): Conv2d(256, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(384, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv5): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=9216, out_features=4096, bias=True)
  (fc2): Linear(in_features=4096, out_features=4096, bias=True)
  (fc3): Linear(in_features=4096, out_features=1000, bias=True)
)
```

Model Summary :
Print the summary of the alexnet to check the params

- Python3

```
# add the cuda to the mode
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

alexnet.to(device)

#Print the summary of the model

summary(alexnet, (3, 224, 224))
```

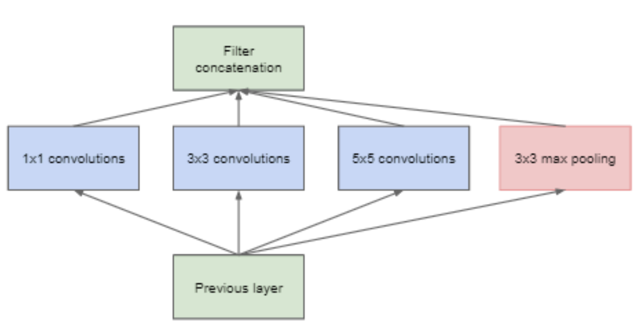
Output:

```
----------------------------------------------------------------

        Layer (type)               Output Shape         Param #

================================================================

            Conv2d-1           [-1, 96, 55, 55]          34,944

         MaxPool2d-2           [-1, 96, 27, 27]               0

            Conv2d-3          [-1, 256, 27, 27]         614,656

         MaxPool2d-4          [-1, 256, 13, 13]               0

            Conv2d-5          [-1, 384, 13, 13]         885,120

            Conv2d-6          [-1, 384, 13, 13]       1,327,488

            Conv2d-7          [-1, 256, 13, 13]         884,992

         MaxPool2d-8            [-1, 256, 6, 6]               0

            Linear-9                 [-1, 4096]      37,752,832

           Linear-10                 [-1, 4096]      16,781,312

           Linear-11                 [-1, 1000]       4,097,000

================================================================

Total params: 62,378,344

Trainable params: 62,378,344

Non-trainable params: 0

----------------------------------------------------------------

Input size (MB): 0.57

Forward/backward pass size (MB): 5.96
```

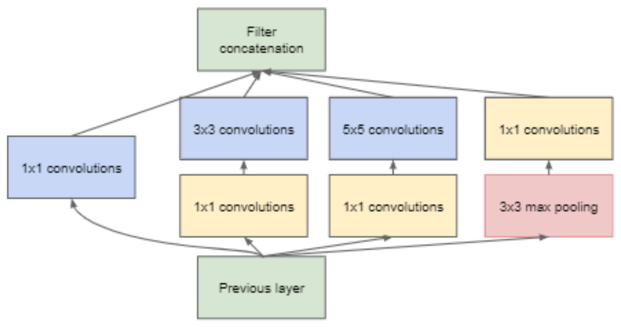
```
Params size (MB): 237.95

Estimated Total Size (MB): 244.49

----------------------------------------------------------------
```

3. GoogleNet (Inception vl)

- The GoogleNet architecture was created by Christian Szegedy from Google Research and achieved a breakthrough result by lowering the top-5 error rate to below 7% in the ILSVRC 2014 challenge. This success was largely attributed to its deeper architecture than other CNNs, enabled by its inception modules which enabled more efficient use of parameters than preceding architectures
- GoogleNet has fewer parameters than AlexNet, with a ratio of 10:1 (roughly 6 million instead of 60 million)
- The architecture of the inception module looks as shown in Fig.



(a) Inception module, naïve version



(b) Inception module with dimension reductions

GoogleNet (Inception Module)

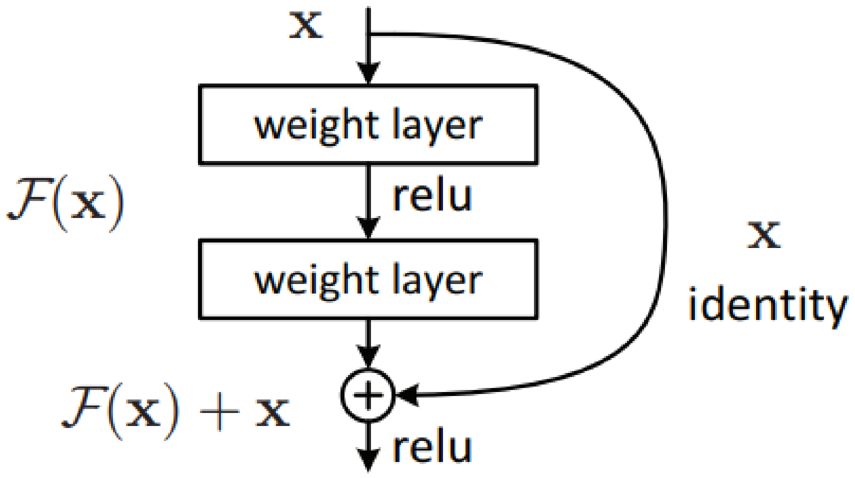
- The notation "3 x 3 + 2(5)" means that the layer uses a 3 x 3 kernel, a stride of 2, and SAME padding. The input signal is then fed to four different layers, each with a RelU activation function and a stride of 1. These convolutional layers have varying kernel sizes (1 x 1, 3 x 3, and 5 x 5) to capture patterns at different scales. Additionally, each layer uses SAME padding, so all outputs have the same height and width as their inputs. This allows for the feature maps from all four top convolutional layers to be concatenated along the depth dimension in the final depth concat layer.
- The overall GoogleNet architecture has 22 larger deep CNN layers.

4. ResNet (Residual Network)

- Residual Network (ResNet), the winner of the ILSVRC 2015 challenge, was developed by Kaiming He and delivered an impressive top-5 error rate of 3.6% with an extremely deep CNN composed of 152 layers. An essential factor enabling the training of such a deep network is the use of skip connections (also known as shortcut connections). The signal that enters a layer is added to the output of a layer located higher up in the stack. Let's explore why this is beneficial.

- When training a neural network, the goal is to make it replicate a target function h(x). By adding the input x to the output of the network (a skip connection), the network is made to model f(x) = h(x) – x, a technique known as residual learning.

F(x) = H(x) - x which gives *H(x) := F(x) + x*.


x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

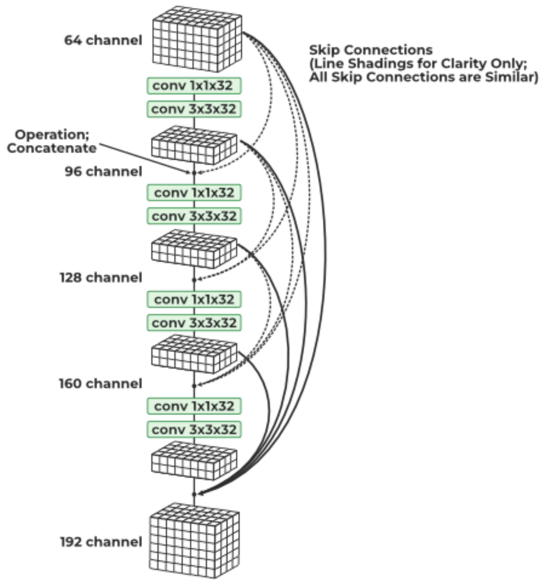
Skip (Shortcut) connection

- When initializing a regular neural network, its weights are near zero, resulting in the network outputting values close to zero. With the addition of skip connections, the resulting network outputs a copy of its inputs, effectively modeling the identity function. This can be beneficial if the target function is similar to the identity function, as it will accelerate training. Furthermore, if multiple skip connections are added, the network can begin to make progress even if several layers have not yet begun learning.
- the target function is fairly close to the identity function (which is often the case), this will speed up training considerably. Moreover, if you add many skin connections, the network can start making progress even if several
- The deep residual network can be viewed as a series of residual units, each of which is a small neural network with a skip connection

5. DenseNet

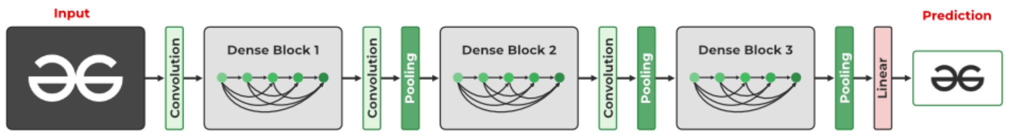
- The DenseNet model introduced the concept of a densely connected convolutional network, where the output of each layer is connected to the input of every subsequent layer. This design principle was developed to address the issue of accuracy decline caused by the vanishing and exploding gradients in high-level neural networks.
- In simpler terms, due to the long distance between the input and output layer, the data is lost before it reaches its destination.
- The DenseNet model introduced the concept of a densely connected convolutional network, where the output of each layer is connected to the input of every subsequent layer. This

design principle was developed to address the issue of accuracy decline caused by the vanishing and exploding gradients in high-level neural networks.



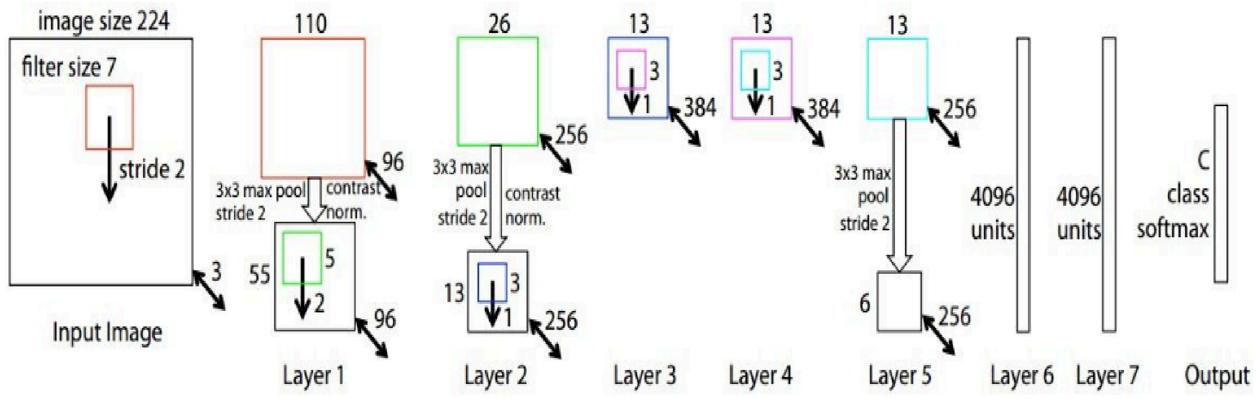
DenseNet

- All convolutions in a dense block are ReLU-activated and use batch normalization. Channel-wise concatenation is only possible if the height and width dimensions of the data remain unchanged, so convolutions in a dense block are all of stride 1. Pooling layers are inserted between dense blocks for further dimensionality reduction.
- Intuitively, one might think that by concatenating all previously seen outputs, the number of channels and parameters would exponentially increase. However, DenseNet is surprisingly economical in terms of learnable parameters. This is because each concatenated block, which may have a relatively large number of channels, is first fed through a 1×1 convolution, reducing it to a small number of channels. Additionally, 1×1 convolutions are economical in terms of parameters. Then, a 3×3 convolution with the same number of channels is applied.
- The resulting channels from each step of the DenseNet are concatenated to the collection of all previously generated outputs. Each step, which utilizes a pair of 1×1 and 3×3 convolutions, adds K channels to the data. Consequently, the number of channels increases linearly with the number of convolutional steps in the dense block. The growth rate remains constant throughout the network, and DenseNet has demonstrated good performance with K values between 12 and 40.
- Dense blocks and pooling layers are combined to form a Tu DenseNet network. The DenseNet21 has 121 layers, however, the structure is adjustable and can readily be extended to more than 200 layers



ZFNet
ZFNet is a modified version of AlexNet which gives a better accuracy.

One major difference in the approaches was that ZF Net used 7x7 sized filters whereas AlexNet used 11x11 filters. The intuition behind this is that by using bigger filters we were losing a lot of pixel information, which we can retain by having smaller filter sizes in the earlier conv layers. The number of filters increase as we go deeper. This network also used ReLUs for their activation and trained using batch stochastic gradient descent.



ZFNet (Zeiler and Fergus Net) is a convolutional neural network architecture that emerged as a significant improvement over its predecessor, AlexNet, in the field of deep learning and computer vision. Introduced by Matthew Zeiler and Rob Fergus in their 2014 paper, ZFNet achieved state-of-the-art performance in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) of 2013.

Key Features of ZFNet

1. **Architecture Improvements**:
   - **Deeper and Wider**: ZFNet has a deeper architecture than AlexNet, featuring more layers and a larger number of filters in the convolutional layers.
   - **Smaller Convolutional Filters**: It employs smaller filters (3x3) in the convolutional layers instead of larger filters (11x11 or 5x5). This allows for more non-linearities and a better learning capacity.

    - **Increased Depth**: The network consists of 8 layers with learnable parameters (5 convolutional layers and 3 fully connected layers).
2. **Receptive Field Visualization**:
    - One of the contributions of ZFNet was the visualization of the receptive fields of the network. The authors demonstrated how the model learns to recognize complex patterns by visualizing intermediate layers.
3. **Use of Deconvolutional Networks**:
    - ZFNet employs deconvolutional layers to visualize the features learned by the convolutional layers. This allows researchers to understand what features the network is focusing on at various stages.
4. **Overlapping Pooling**:
    - ZFNet utilizes overlapping pooling layers, which help reduce spatial dimensions while retaining important features, contributing to improved performance.

Architecture Overview

- **Layer Structure**:
    - **Convolutional Layers**: ZFNet has several convolutional layers that extract features from the input images.
    - **Activation Functions**: It commonly uses ReLU (Rectified Linear Unit) as the activation function to introduce non-linearity.
    - **Pooling Layers**: Max pooling layers reduce the dimensionality of feature maps while retaining critical information.
    - **Fully Connected Layers**: The final layers are fully connected, which map the learned features to the output classes.

Performance

ZFNet demonstrated significant improvements over AlexNet, especially in terms of classification accuracy on the ImageNet dataset. Its design choices paved the way for further developments in deep learning architectures.

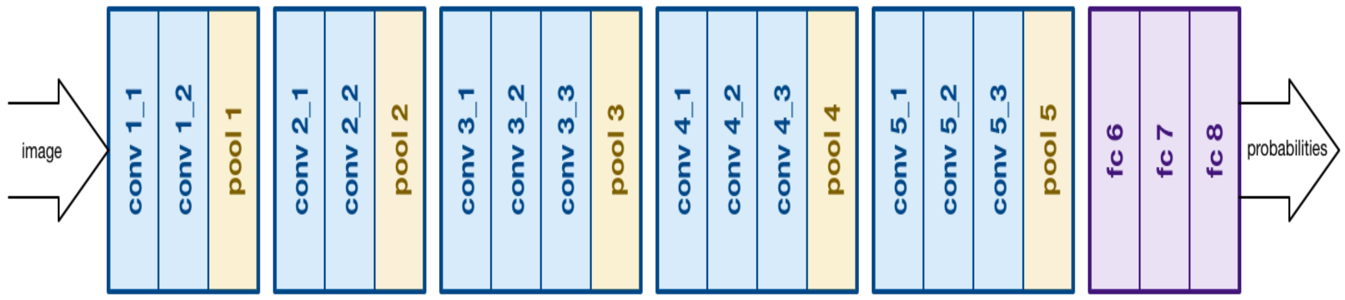
Legacy

ZFNet is often viewed as a stepping stone in the evolution of convolutional neural networks, influencing subsequent architectures such as GoogLeNet and ResNet. Its insights into visualization and feature extraction have been crucial for understanding how neural networks learn.

VGG Net:

The idea of VGG was submitted in 2013 and it became a runner up in the ImageNet contest in 2014. It is widely used as a simple architecture compared to AlexNet and ZFNet.

VGG Net used 3x3 filters compared to 11x11 filters in AlexNet and 7x7 in ZFNet. The authors give the intuition behind this that having two consecutive 2 consecutive 3x3 filters gives an effective receptive field of 5x5, and 3 – 3x3 filters give a receptive field of 7x7 filters, but using this we can use a far less number of hyper-parameters to be trained in the network.



VGGNet is a convolutional neural network architecture developed by the Visual Geometry Group (VGG) at the University of Oxford. It gained significant attention for its depth and performance in image classification tasks, particularly in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2014.

Key Features of VGGNet

1. **Architecture Depth**:
   - o VGGNet is characterized by its depth, with configurations that include up to 19 layers (VGG16 and VGG19 are the most commonly referenced versions). The architecture consists of a series of convolutional layers followed by fully connected layers.
2. **Uniform Architecture**:
   - o The network employs a simple and uniform architecture using small convolutional filters (3x3) throughout the network, which helps capture fine details while maintaining a manageable number of parameters.
   - o The model typically uses 2x2 max pooling layers to reduce dimensionality after a series of convolutional layers.
3. **Stacked Convolutions**:
   - o VGGNet uses multiple stacked convolutional layers with the same spatial dimensions before downsampling. For example, instead of using a single 5x5 filter, it uses two 3x3 filters stacked on top of each other.
4. **ReLU Activation**:
   - o The network employs the ReLU (Rectified Linear Unit) activation function, which introduces non-linearity and helps mitigate the vanishing gradient problem.
5. **Fully Connected Layers**:

   - After the convolutional and pooling layers, VGGNet typically includes several fully connected layers, with the final layer using a softmax activation function for classification tasks.

Architecture Overview

- **VGG16**:
   - 16 layers with learnable parameters: 13 convolutional layers and 3 fully connected layers.
- **VGG19**:
   - 19 layers with learnable parameters: 16 convolutional layers and 3 fully connected layers.

Performance

VGGNet achieved outstanding results in image classification tasks and is known for its high accuracy. Its design principles have influenced many subsequent architectures and have been widely adopted in various applications, including transfer learning.

Legacy and Applications

1. **Transfer Learning**: VGGNet is often used as a base model for transfer learning due to its well-defined architecture and good performance on diverse datasets.
2. **Feature Extraction**: The convolutional layers can be utilized to extract features from images for tasks such as object detection and image segmentation.
3. **Influence on Subsequent Architectures**: The design principles of VGGNet (e.g., small filters and deep networks) have influenced many later architectures, including ResNet and Inception.

Limitations

- **Computationally Intensive**: VGGNet is known for its large number of parameters, making it computationally expensive and slower to train compared to more recent architectures.
- **Memory Usage**: The deep structure requires significant memory, which can be a limitation in resource-constrained environments.

Guided Propagation

Vanilla backpropagation at ReLUs and DeconvNets are combined to create guided backpropagation. The negative neurons are deactivated by ReLU, an activation function. The deconvolution and unpooling layers make up DeconvNets. Just the features that the neuron recognizes in an image are of relevance to us. We, therefore, set all negative gradients to 0 when propagating the gradient. We don't care if a pixel someplace near our neuron "suppresses" (has a negative value) another neuron. The pixel importance is represented by a value in the filter map that is larger than zero. This value overlaps with the input image to show which pixel from the input image contributed the most.

can be asked in ppr:-

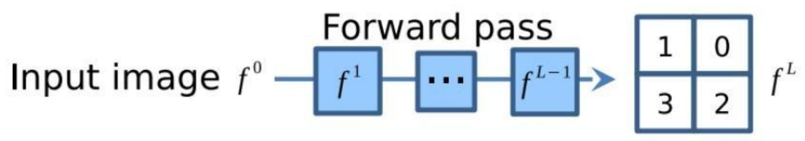
Working on Guided Propagation

Given below is an example of how guided backpropagation works:

Similar to Grad-CAM, guided backpropagation is a gradient-based visualization technique.

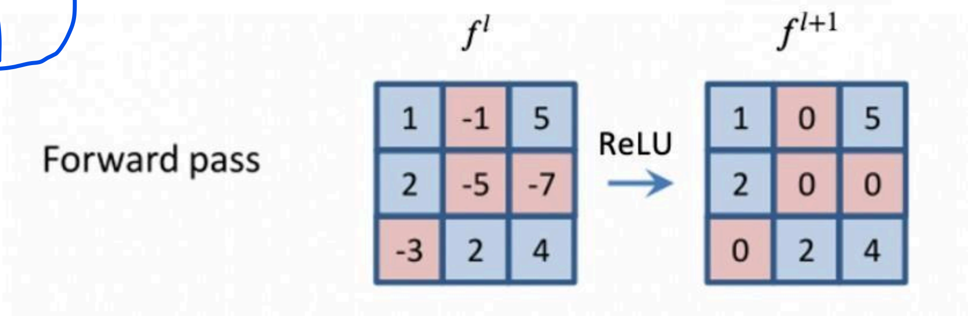
Given an input image and a pre-trained network, guided backpropagation can tell us which pixels in the input image matter for the correct classification. And it is called **guided** backpropagation because we choose('guide') which neurons are to be activated for backpropagation.

To perform guided backpropagation, firstly we do a forward pass to the layer that we are interested in:



This means that if the input values in the previous layer are negative, the output after the ReLU operation is set to 0, e.g.

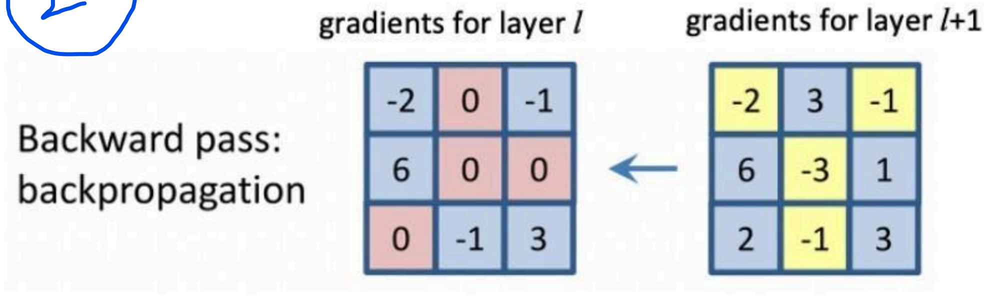
1



Under this circumstance, we can say that the neurons with negative values are **dead**.

For the ReLU, when we do a **traditional** backpropagation, it passes gradient to a previous layer only if the **original input** was positive(the neurons are not dead), e.g.

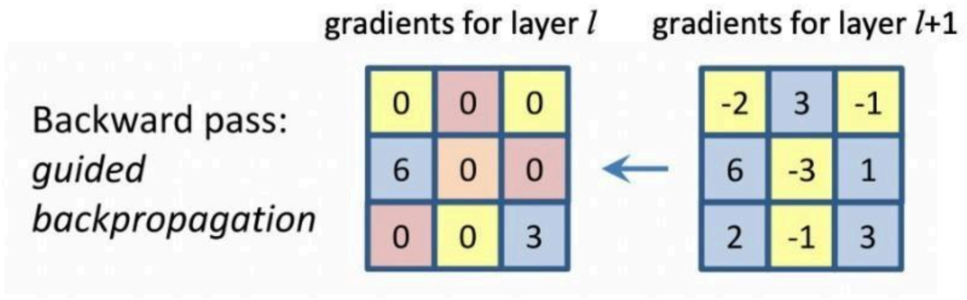
2



The neurons marked red are dead, and the gradients can not be backpropagated.

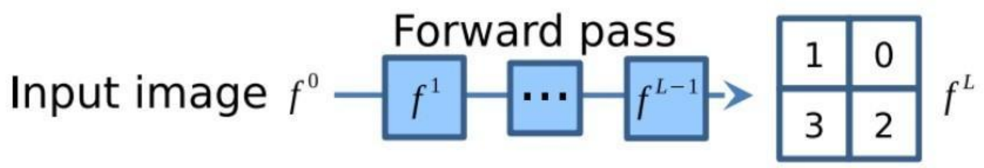
As shown, the gradients for the dead neurons(marked red) are set to 0.

In the case of **guided** backpropagation, we also do not backpass the negative gradients(the neurons marked yellow in the above picture) to the previous layer, that is:

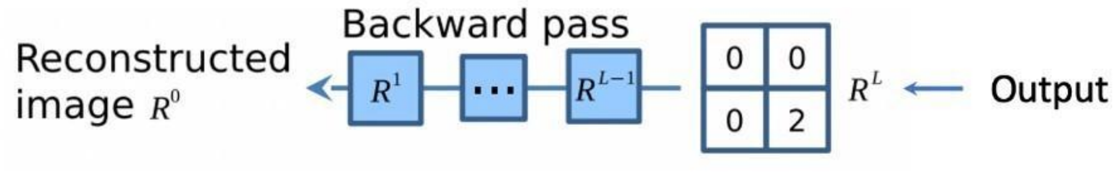

gradients for layer l
gradients for layer l+1
Backward pass: *guided backpropagation*
0 0 0
6 0 0
0 0 3
-2 3 -1
6 -3 1
2 -1 3

The gradients for both the red and yellow neurons are not backpropagated.

Recap our forward pass, which is:



Suppose we are only interested in the neuron at the right bottom corner, which has the gradient value 2. We can retain this neuron and set all other neurons to 0. Then the gradient for the chosen neuron is backpropagated all the way to the inputs to get a reconstruction:



We have talked about how to choose neurons for guided backpropagation above(i.e. choose the neurons that are not marked red or yellow). That is, in the guided backpropagation, the backpropagation logic is:

$$R_i^l = (f_i^l > 0) \cdot (R_i^{l+1} > 0) \cdot R_i^{l+1}$$

where $R_i^{l+1} = \frac{\partial out}{\partial f_i^{l+1}}$.

Steps to Implement

Here is an implementation of a convolution neural network where we have used guided backward propagation that helps us to visualize fine-grained details in an image.

Here is an implementation of a convolution neural network where we have used guided backward propagation that helps us to visualize fine-grained details in an image.

1. First, we will import all the necessary libraries that we are going to import to make a model.

2. Now we will build AlexNet Model.

3. Load the image.

4. Let us preprocess the image by using AlexNet's preprocess function.

5. Importing AlexNet's transferred learning model on the imagenet dataset.

6. We are creating a model until the last convolution layer from the imported AlexNet's transferred learning model. When we use the fully connected layer in the deep learning CNN model, we lose the spatial information which is retained by convolution layers.

7. Gradients of ReLU are overridden by applying the RELU function that allows the fine-grained control over the gradients for backpropagating non-negative gradients to have a more efficient or numerically stable gradient.

8. Applying the guided ReLU function to all the convolution layers wherever the activation function was ReLU.

9. Finally, visualize the original image and guided backpropagation image.