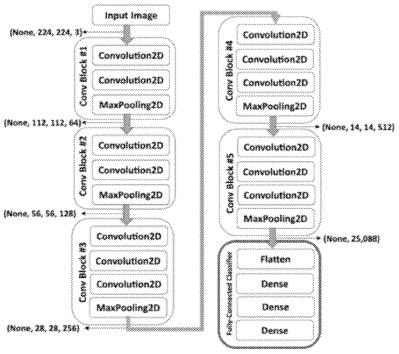
First - Refer images

VGG16

* VGG16 (2014): 16 layers

16 trainable layers



VGG16

* VGG 19

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					

Table 2: **Number of parameters** (in millions).

Network	A, A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

VGG16

- Most parameters are in the FC players (due to not having way sharing there). Hence, motivation for fully convolutional networks (no FC layers)
- Most memory in early convolution layers (due to high spatial resolution). Memory cannot be released before completing the backward pass

The convolutional layers share weights but the last fully connected layers do noy share--> so most parameters are in the Fully connected ones.

so, we would want to have all convolutional layers and remove the fully connected layers

The fully connected layers become a fixed size once we have trained the model and so the number of parameters cannot be changed. That is not the case for convolutional layers

VGG16

using pretrained VGG16 in Keras:

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing import image

# Load pre-trained VGG16 model with imagenet weights
model = keras.applications.VGG16(weights='imagenet')

# Load an example image of a dog and preprocess it for input to the model
img_path = 'dog.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = keras.applications.vgg16.preprocess_input(x)

# Use the model to predict the top 5 most likely classes for the image
preds = model.predict(x)
decoded_preds = keras.applications.vgg16.decode_predictions(preds, top=5)[0]
for pred in decoded_preds:
    print(pred[1], ':', pred[2])
```

the n/w expects a batch, so we give the batch size to be 1

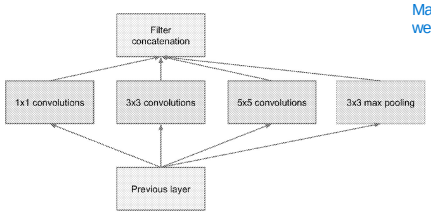
preprocess–>normalize, scale/resize to what the n/w expects

Inception

instead of having pyramid observation at different level, here we have multiple view levels at the same layer itself

- GoogleNet / Inception (2014)
- 22 layers with fewer parameters (5M)
- In the inception block, multiple receptive fields cause dimensionality increase

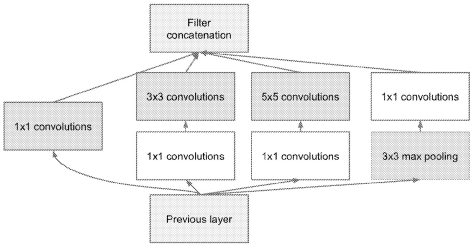
Make sure the dimensions of the output from these conv are same so that we can concatenate the results to the next layer. we also need to make sure that we control the number of outputs



(a) Inception module, naïve version

Inception

- A modified inception block that reduces the number of channels using 1x1 convolutions



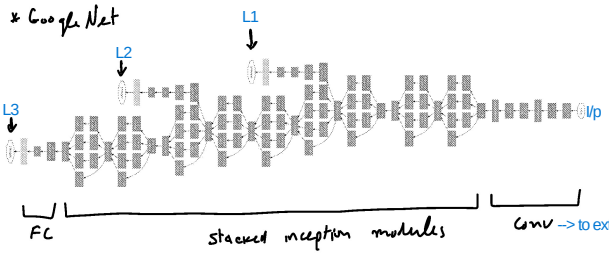
(b) Inception module with dimensionality reduction

Inception

```
from keras.layers import Conv2D, MaxPooling2D, concatenate

def inception_block(x, filters):
    # First branch with 1x1 convolution
    branch1x1 = Conv2D(filters[0], (1,1), padding='same', activation='relu')(x)

    # Second branch with 1x1 and 3x3 convolution
    branch3x3 = Conv2D(filters[1], (1,1), padding='same', activation='relu')(x)
    branch3x3 = Conv2D(filters[2], (3,3), padding='same', activation='relu')
(branch3x3)

    # Third branch with 1x1 and 5x5 convolution
    branch5x5 = Conv2D(filters[3], (1,1), padding='same', activation='relu')(x)
    branch5x5 = Conv2D(filters[4], (5,5), padding='same', activation='relu')
(branch5x5)

    # Fourth branch with max pooling and 1x1 convolution
    branch_pool = MaxPooling2D((3,3), strides=(1,1), padding='same')(x)
    branch_pool = Conv2D(filters[5], (1,1), padding='same', activation='relu')
(branch_pool)

    # Concatenate the outputs of the four branches
    output = concatenate([branch1x1, branch3x3, branch5x5, branch_pool], axis=3)

    return output
```

Inception

* GoogLeNet



- A single FC layer leads to fewer parameters
- Convergence may be difficult. Use auxiliary classifiers to help with vanishing ingredients (not needed with batch normalization)

Inception

* Example of using pre-trained inception network:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.inception_v1 import InceptionV1
from tensorflow.keras.applications.inception_v1 import preprocess_input,
decode_predictions
import numpy as np

# Load the pre-trained InceptionV1 model
model = InceptionV1(weights='imagenet')

# Load an example image of a dog
img_path = 'path/to/dog/image.jpg'
img = image.load_img(img_path, target_size=(224, 224))

# Convert the image to a numpy array and preprocess it
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

# Use the pre-trained model to predict the class of the image
preds = model.predict(x)

# Decode the predictions and print the top 3 results
decoded_preds = decode_predictions(preds, top=3)[0]
for pred in decoded_preds:
    print(pred[1], pred[2])
```

vanishing gradients in Deeper n/ws---->when we train we push the gradients forward and as feedback we push them backward again. when the n/w is deeper, during the back propagation the gradients becomes smaller and smaller therefore vanishes and finally doesnt train the inital part of the n/w while coming from last layer the backward direction

We send feedback gradients based on the loss function at the end of the n/w generally. To solve the problem of vanishing gradients in inception, the loss is calculated at appropriate intervals and the gradients are pushed back without being vanished and trains every layer.

That is the L1,L2, L3 layers-->where the losses are calculated

classification-->categorical loss entropy-->the loss of all 3 L1,l2,l3 are added and compute gradient to answer the classification

--> to extract basic features before sending through the inception blocks

If we wanted to use the weights of a pretrained model, for better results we should do the preprocessing for our i/p also the same way it was done for that trained model

View images for other inception types

ResNet

- ResNet (2015):
- 152 layers (Microsoft). Optimization is more difficult with more layers (more DOF)

the sudden drop in both 56 and 20 may be due to the change in the learning rate

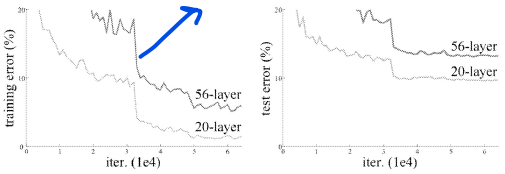


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

Motivation - Deeper layers needed to extract features and solve complex problems but the deeper the n/w, the more difficult it is to train the n/w.

ResNet

- Residual blocks:
- It is easier to learn the ~~F(x)~~ H(x) residual compared with ~~H(x)~~ F(x) because it is easier to learn deviation from identity instead of a function
- Skip connections help with vanishing ingredients

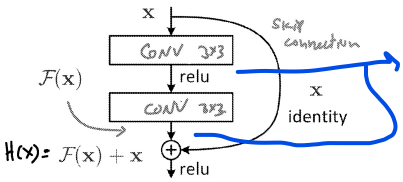


Figure 2. Residual learning: a building block.

batch normalization also done in b/w

Here, F(x)-->tells what needs to be changed from x ...h(x) = F(x) +x will be passed to the next layer. if there is nothign to learn/change , f(x)=0 and the input info is passed along to the next layer as h(x)=x...useful expescially in the intial stage of training where we intialize weights close to zero

Advantage -
1. If we do not learn anything also through the n/w, we still pass along the i/p threby not destroying the o/p
2. skip connection - when we push gradients backwards for learning , when it goes thru multiple layers, the gradient may vanish but fo the skip /direct connection, there wont be loss of gradients

ResNet

- Residual block properties:
  - Zero weights in the block produce identity instead of destroying the signal. The network can learn to zero blocks to eliminate unneeded layers
  - In a normal network zero weights shut down information whereas here zero weights cause the information to pass through units without change
  - Passing gradients directly through skip connections leads to quicker training

ResNet

```
from tensorflow.keras.layers import Conv2D, BatchNormalization, Add, Activation

def resnet_block(inputs, num_filters, kernel_size, strides):
    x = Conv2D(num_filters, kernel_size=kernel_size, strides=strides,
padding='same')(inputs)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    x = Conv2D(num_filters, kernel_size=kernel_size, strides=1, padding='same')
(x)
    x = BatchNormalization()(x)

    # Add the input to the output of the second convolution layer
    res = Add()([x, inputs])
    out = Activation('relu')(res)

    return out
```

ResNet

- Complete ResNet architecture:
  - Start with regular convolution layers
  - Stack residual blocks
  - Periodically double channels and pool spatially
  - Single FC layer with softmax for output
  - Batch normalization after every convolution a layer
  - No dropout

ResNet

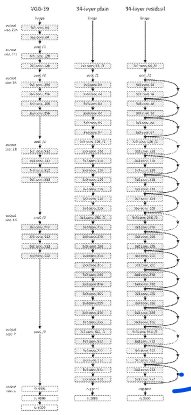
floating point operations

VGG 19: 19.6 B FLOPs

Plain 34: 3.6 B FLOPs

ResNet 34: 3.6 B FLOPs

just doesnt have additions of skip connections but not much difference



Resnet without skip connection for comparison of performance

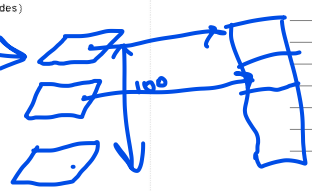
1000 classes in imagenet

Figure 3. Example network architectures for ImageNet. **Left**: the VGG-19 model [41] (19.6 billion FLOPs) as a reference. **Middle**: a plain network with 34 parameter layers (3.6 billion FLOPs). **Right**: a residual network with 34 parameter layers (3.6 billion FLOPs). The dotted shortcuts increase dimensions. **Table 1** shows more details and other variants.

ResNet

```
def resnet(input_shape, num_classes, num_filters, kernel_size, strides,
num_blocks):
    # Input layer
    inputs = Input(shape=input_shape)

    # Convolutional layer with batch normalization and ReLU activation
    x = Conv2D(num_filters, kernel_size=kernel_size, strides=strides,
padding='same')(inputs)
    x = BatchNormalization()(x)
    x = Activation('relu')(x)

    # Residual blocks
    for i in range(num_blocks):
        x = resnet_block(x, num_filters, kernel_size, strides)

    # Global average pooling layer
    x = GlobalAveragePooling2D()(x)

    # Fully connected output layer
    x = Dense(num_classes, activation='softmax')(x)

    # Create model
    model = Model(inputs=inputs, outputs=x, name='ResNet')

    return model
```

making the skip connection



Average pooling reducing the number of parameters before reaching the flatten layer

View images for graphs

ResNet

x variations:

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112 × 112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Table 1. Architectures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of blocks stacked. Downsampling is performed by conv3_1, conv4_1, and conv5_1 with a stride of 2.

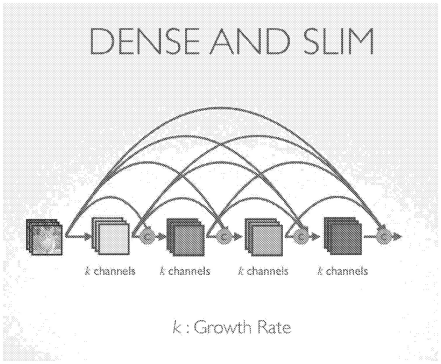
Refer image

ResNet

```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import load_img, img_to_array
from tensorflow.keras.applications.resnet50 import ResNet50, preprocess_input,
decode_predictions
import numpy as np

# load the pre-trained ResNet50 model
model = ResNet50(weights='imagenet')

# load the image to classify
img = load_img('cat.jpg', target_size=(224, 224))

# convert the image to a numpy array
img_array = img_to_array(img)

# expand the dimensions of the image to match the input shape of the model
img_array = np.expand_dims(img_array, axis=0)

# preprocess the input image
img_array = preprocess_input(img_array)

# make a prediction on the image
prediction = model.predict(img_array)

# decode the predictions
labels = decode_predictions(prediction, top=3)[0]

# print the labels
for label in labels:
    print(label[1], label[2])
```

First - Refer images

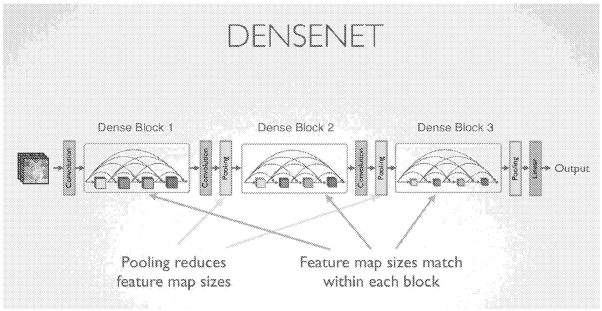
DenseNet



skip connection to all the subsequent outputs

Here,we limit ourselves to k channels

DenseNet



DenseNet

* DenseNet implementation:

The main features of the DenseNet architecture include:

1. Dense Connections: Each layer in the network is connected to all the preceding layers, allowing for feature reuse and reducing the number of parameters.
2. Bottleneck Layers: The architecture uses bottleneck layers to reduce the number of feature maps before each dense block, further reducing the number of parameters. use 1x1 con
3. Transition Layers: Between each dense block, there is a transition layer that reduces the number of feature maps and spatial dimensions.
4. Global Average Pooling: Instead of using a fully connected layer at the end of the network, DenseNet uses global average pooling to reduce the number of parameters and improve regularization.

DenseNet

```
from keras.layers import Input, Conv2D, MaxPooling2D, Dense,
GlobalAveragePooling2D, BatchNormalization, Dropout, concatenate
from keras.models import Model

def dense_block(x, blocks, growth_rate):
    for i in range(blocks):
        conv = Conv2D(growth_rate, (3,3), padding='same', activation='relu')(x)
        x = concatenate([x, conv]) skip connectio
    return x

def transition_layer(x, compression):
    num_filters = int(x.shape.as_list()[-1] * compression)
    x = Conv2D(num_filters, (1,1), padding='same', activation='relu')(x)
    x = MaxPooling2D((2,2))(x)
    return x
```

DenseNet

```
def DenseNet(input_shape, depth, num_classes, growth_rate=12, blocks_per_layer=4,
compression=0.5):
    # Input layer
    inputs = Input(shape=input_shape)

    # Initial convolution layer
    x = Conv2D(2*growth_rate, (3,3), padding='same', activation='relu')(inputs)

    # Dense blocks and transition layers
    layers = []
    for i in range(depth):
        x = dense_block(x, blocks_per_layer, growth_rate)
        layers.append(x)
        if i < depth-1:
            x = transition_layer(x, compression)

    # Global average pooling and output layer
    x = BatchNormalization()(x)
    x = GlobalAveragePooling2D()(x)
    x = Dense(num_classes, activation='softmax')(x)

    # Build model
    model = Model(inputs, x, name='DenseNet')

    return model
```

DenseNet

To create a DenseNet with 121 layers, 32 blocks per layer, and a growth rate of 32, we can call the `DenseNet` function as follows:

```
model = DenseNet(input_shape=(224, 224, 3), depth=121, num_classes=1000,
growth_rate=32, blocks_per_layer=32, compression=0.5)
```

Review images

DenseNet

```
import tensorflow as tf
from tensorflow.keras.applications.densenet import DenseNet121, preprocess_input
from tensorflow.keras.preprocessing.image import load_img, img_to_array
import numpy as np

# Load the pre-trained model
model = DenseNet121(weights='imagenet')

# Load and preprocess the image
image_path = 'cat.jpg'
image = load_img(image_path, target_size=(224, 224))
image_array = img_to_array(image)
image_array = preprocess_input(image_array)

# Make predictions on the image
predictions = model.predict(np.array([image_array]))
predicted_classes =
tf.keras.applications.densenet.decode_predictions(predictions, top=1)[0]
class_name = predicted_classes[0][1]
class_prob = predicted_classes[0][2]

# Print the predicted class and probability
print(f"Predicted class: {class_name}")
print(f"Probability: {class_prob}")
```