Introduction to Computer Vision

A Practical Approach

By:
Shreyas Om

Who Am I ?

Shreyas Om

Upcoming Data Engineering Analyst @Tredence Analytics
CXM Intern @Adobe
Data Scientist Intern @ Cornerstone Solutions
Machine Learning Intern @ Feynn Labs

Co Authored Research Papers:

1) An Improved Deep Learning Model Implementation for Pest Species Detection.
2) Unraveling the Potential of Knowledge Graphs and Graph Neural Networks
3) Revolutionary Dehazing Advances: A Comparative Study.
4) A novel deep learning method for plant disease identification from Leaf images.

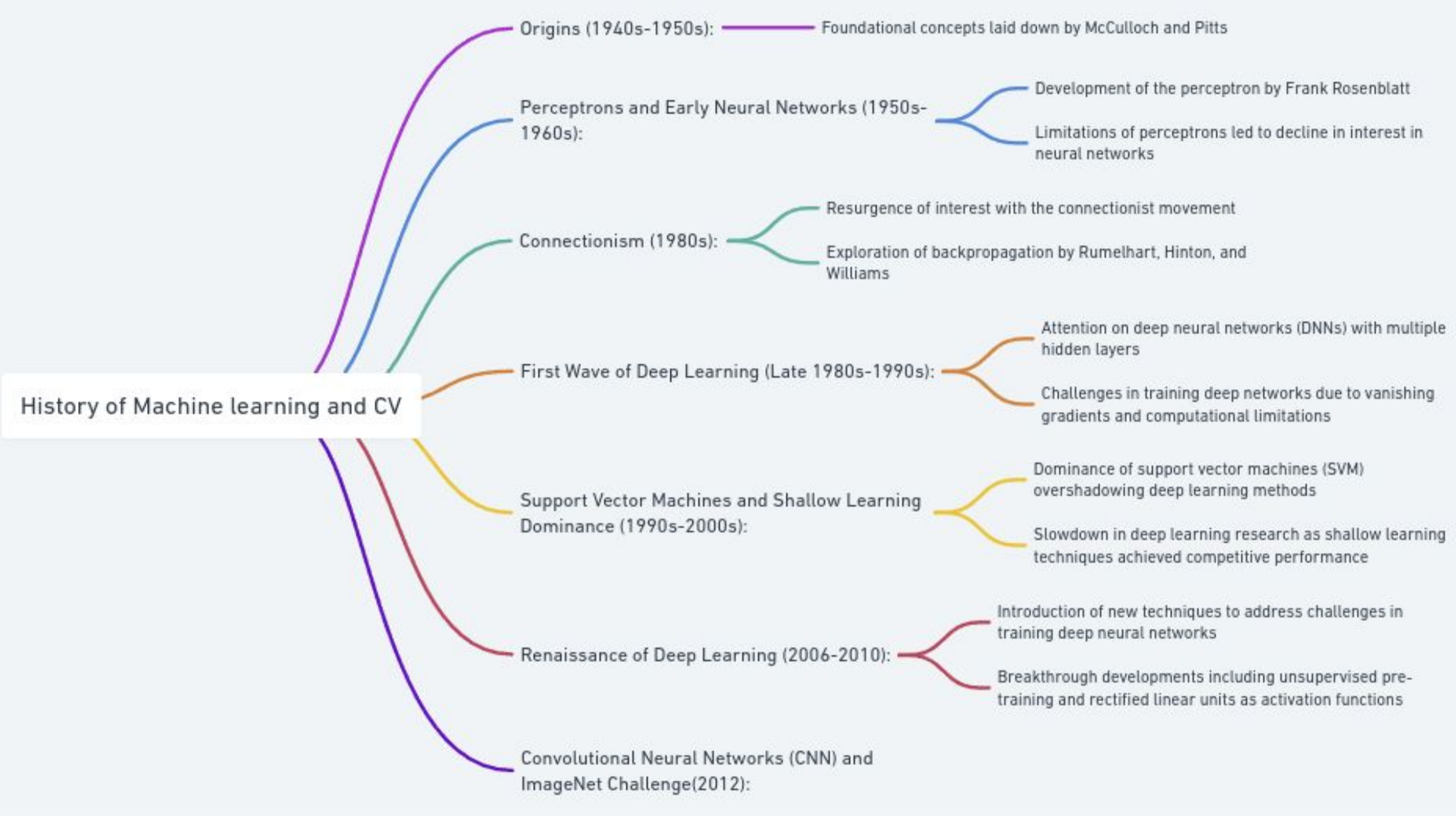
Contact me:
linkedin.com/in/shreyas-om
github.com/om-shreyas

What to Expect

Computer vision is a vast topic with numerous advancements being made continuously. Along with that it has arduous maths and an in depth theory behind it… Which we will skip.

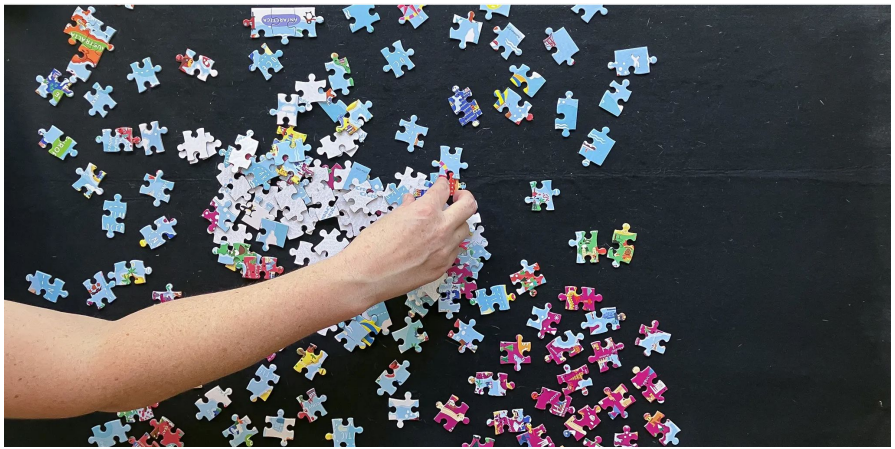
Today I'll show you how you can practically approach a computer vision problem and what and all you need to be knowing make your own computer vision model.

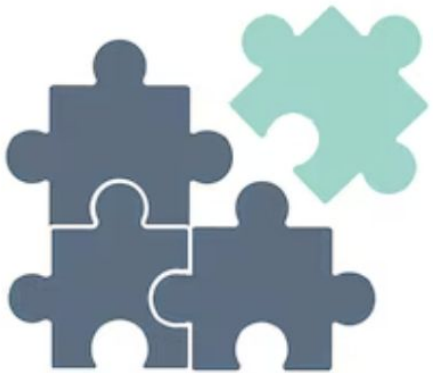
Finally we will learn how to use Azure computer vision API to perform our own computer vision tasks


History of Machine learning and CV
Origins (1940s-1950s):
Foundational concepts laid down by McCulloch and Pitts
Perceptrons and Early Neural Networks (1950s-1960s):
Development of the perceptron by Frank Rosenblatt
Limitations of perceptrons led to decline in interest in neural networks
Connectionism (1980s):
Resurgence of interest with the connectionist movement
Exploration of backpropagation by Rumelhart, Hinton, and Williams
First Wave of Deep Learning (Late 1980s-1990s):
Attention on deep neural networks (DNNs) with multiple hidden layers
Challenges in training deep networks due to vanishing gradients and computational limitations
Support Vector Machines and Shallow Learning Dominance (1990s-2000s):
Dominance of support vector machines (SVM) overshadowing deep learning methods
Slowdown in deep learning research as shallow learning techniques achieved competitive performance
Renaissance of Deep Learning (2006-2010):
Introduction of new techniques to address challenges in training deep neural networks
Breakthrough developments including unsupervised pre-training and rectified linear units as activation functions
Convolutional Neural Networks (CNN) and ImageNet Challenge(2012):

CNN (Convolutional Neural Networks)

- Convolutional: A mathematical operation that combines two signals to form a third signal. This is a key technique in digital signal processing.

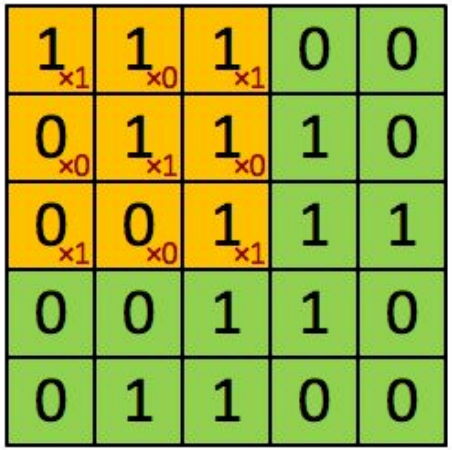


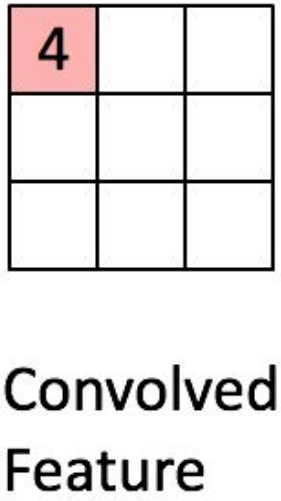


Continued.....

- Key operation: Convolution involves sliding a small filter (kernel) over the input image.
- Purpose: Extracts local patterns and features from the input.
- Layers involved: Convolutional, pooling, and activation functions.
- Progression: Features are passed through multiple layers to learn increasingly abstract representations.
- Effectiveness: CNNs excel in tasks like image classification, object detection, and semantic segmentation.
- Ability: Automatically learns spatial hierarchies of features from raw pixel data.

An Overview of CNN code



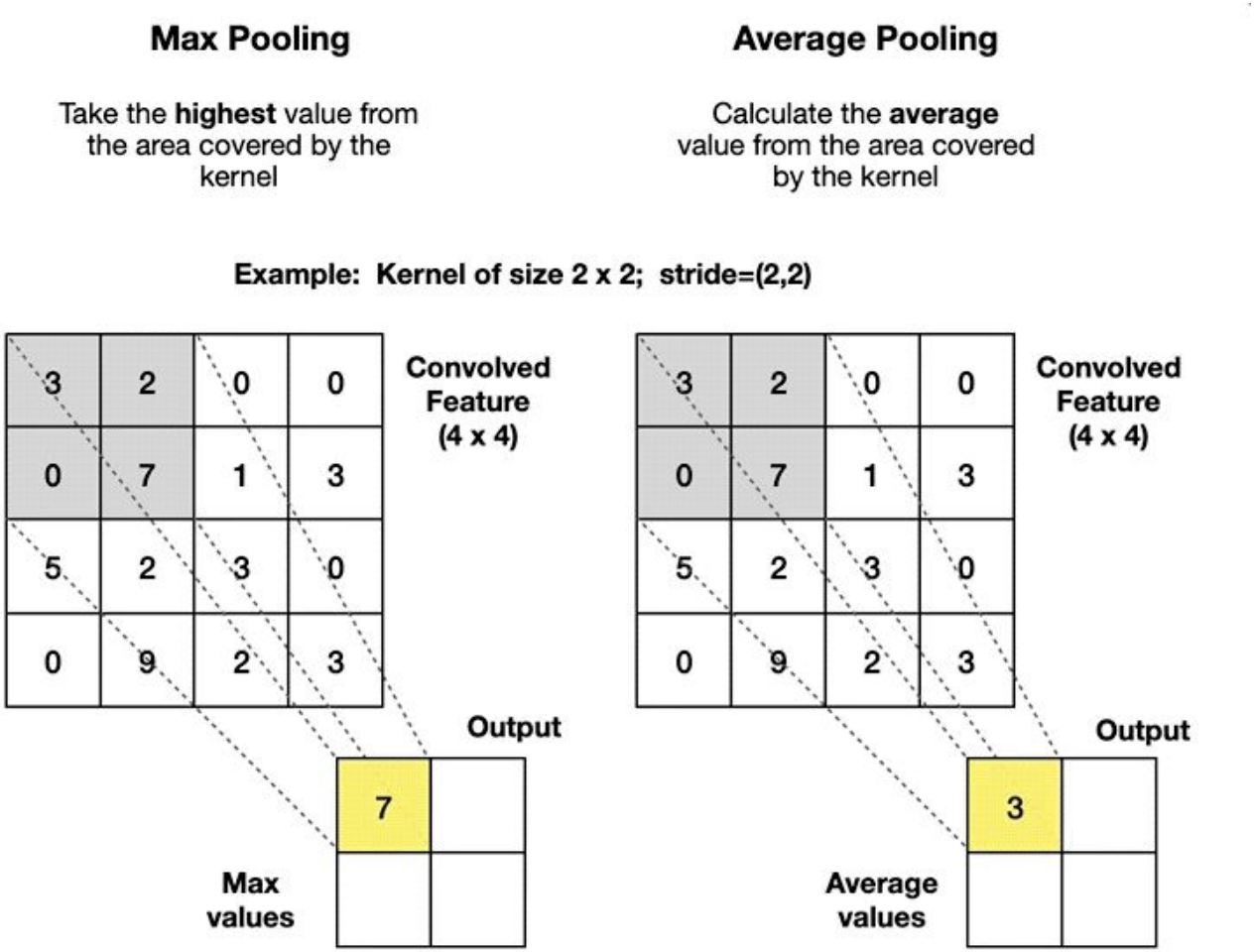


- tf.keras.layers.Conv2D(filters=32, kernel_size=(3, 3), strides=(1, 1), padding='valid', activation='relu', input_shape=(28, 28, 1))
- filters: The number of filters or kernels to apply to the input data.
- kernel_size: The size of the convolutional filters.
- strides: The stride of the convolution operation.
- padding: The padding method (valid or same) to apply during convolution.

Pooling Layers

The working of pooling layers in Convolutional Neural Networks (CNNs) involves downsampling the feature maps generated by the convolutional layers. Pooling layers help reduce the spatial dimensions of the feature maps

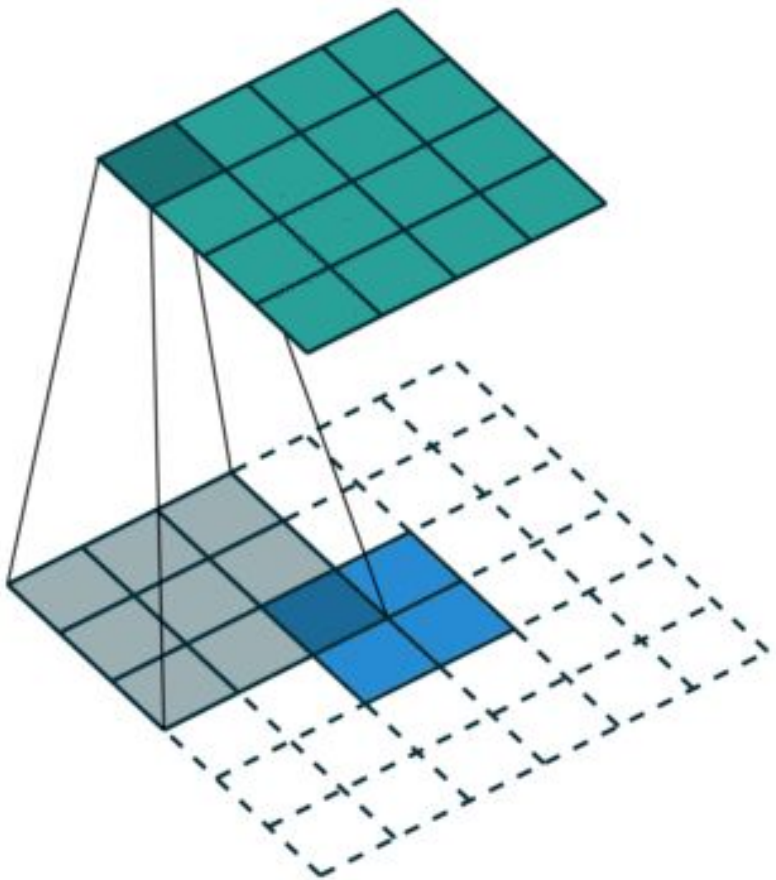
- Sliding Window: Utilizes a small window moving across the input feature map.
- Aggregation Operation: Within each window, a pooling operation is performed, such as max pooling (retaining the maximum value), average pooling (computing the average), or min pooling (retaining the minimum).
- Downsampling: After pooling, the window moves to the next location by a predefined stride, reducing the spatial dimensions of the feature maps.
- Dimensionality Reduction: Reduces the size of feature maps while retaining important information, aiding in computational efficiency and controlling overfitting.
- Example: In max pooling, for instance, a 2x2 window with a stride of 2 retains the maximum value within each window, halving the spatial dimensions of the feature maps.



- tf.keras.layers.MaxPooling2D(pool_size=(2, 2), strides=(2, 2), padding='valid'),
- pool_size: The size of the pooling window.
- strides: The stride of the pooling operation.
- padding: The padding method to apply during pooling.

De Convolution

- Deconvolution increases spatial resolution in neural networks.
- Unlike convolution, which downsamples input, deconvolution upsamples feature maps.
- It reverses convolution operations by padding input and applying filters to expand spatial dimensions.
- Deconvolution layers aid tasks like image segmentation, crucial for detailed spatial information.
- They recover spatial details lost during downsampling, facilitating high-resolution image or feature map reconstruction.
- This technique is essential for tasks like image generation, super-resolution, and semantic segmentation in computer vision.



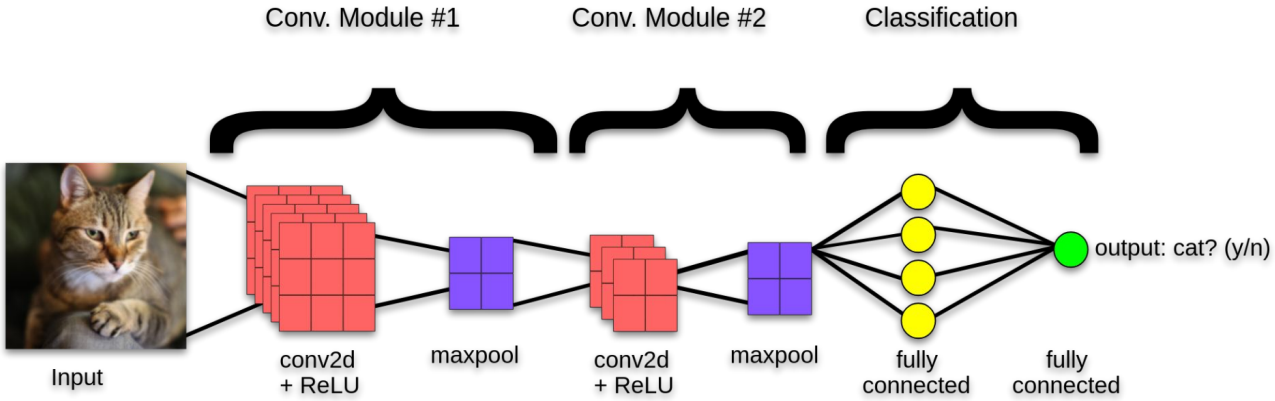
- tf.keras.layers.Conv2DTranspose(filters=32, kernel_size=(3, 3), strides=(2, 2), padding='same', activation='relu'),
- filters=32: This parameter determines the number of filters (or channels) that the layer will learn. In this case, it's set to 32, meaning the layer will output 32 feature maps.
- kernel_size=(3, 3): This parameter defines the size of the convolutional filter or kernel. It's a tuple (3, 3), indicating a 3x3 filter will be used.
- strides=(2, 2): This parameter specifies the stride of the convolution operation. Here, it's set to (2, 2), meaning the filter will move 2 pixels at a time along both the height and width dimensions of the input feature map.
- padding='same': This parameter determines the padding strategy to be applied to the input feature map. 'same' padding means the output feature map will have the same spatial dimensions as the input, achieved by padding the input as necessary.
- activation='relu': This parameter specifies the activation function applied to the output of the convolution operation. 'relu'

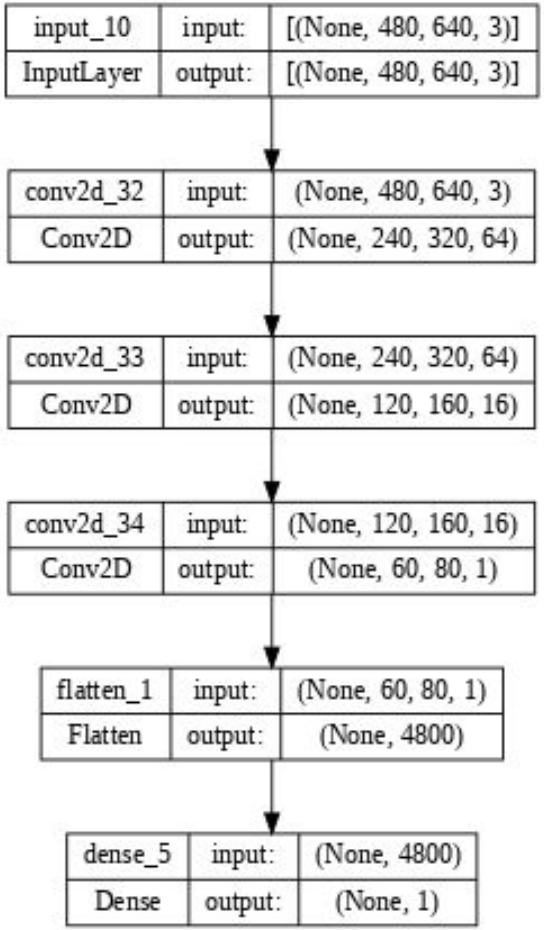
Classification

- Classification problems in CV relates to when we are classifying an image in different categories. In real life image classification are used as follows:
- Medical Diagnosis: Diagnosing diseases from medical imaging like X-rays and MRI scans.
- Quality Control: Detecting defects and ensuring consistency in manufacturing and agriculture.
- Biometric Identification: Identifying individuals using facial and fingerprint recognition.
- Environmental Monitoring: Analyzing satellite imagery for land cover, deforestation, and wildlife habitats.
- Retail and E-commerce: Categorizing products and enhancing shopping experiences.
- Security and Surveillance: Detecting suspicious activities and monitoring public spaces.
- Social Media Moderation: Filtering inappropriate content on social media platforms.
- Natural Resource Management: Monitoring land use, vegetation cover, and water bodies.
- Augmented Reality and Gaming: Enhancing user experiences through object recognition and scene understanding.

A general gist on how to approach this:

- The way we handle image recognition is basically to look at every pixel/set of pixels and finally reach a dense layer representing the number of classes that were to classified.
- We continuously step down the image and kernel size till we reach a handalable size of neurons and then flatten them to reach the final layer of dense neurons.

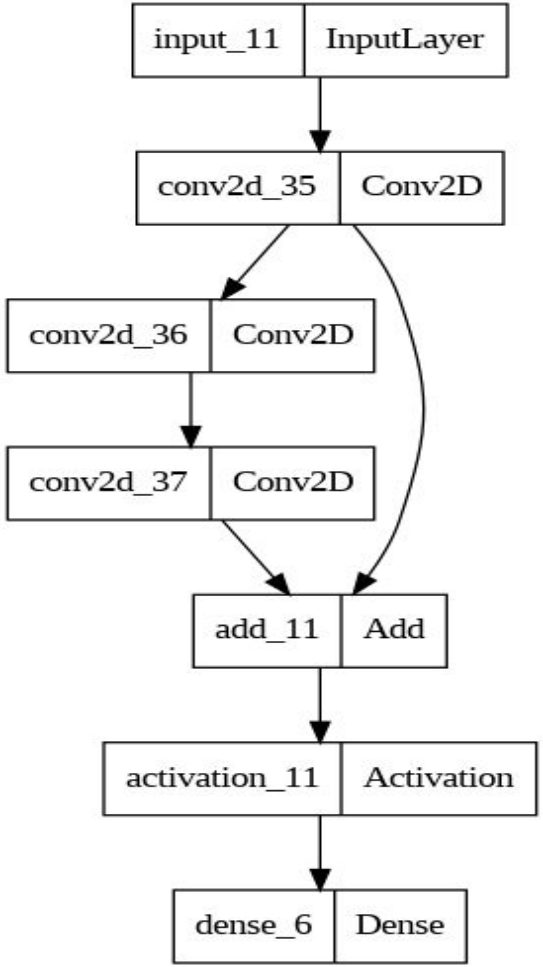




An example of this

```
def discriminator_network(img_shape):
    model = Sequential()
    model.add(Input(img_shape))
    model.add(Conv2D(64, (64, 64), strides=(2, 2) ,padding='same', activation='relu'))
    model.add(Conv2D(16, (16, 16), strides=(2, 2) ,padding='same', activation='relu'))
    model.add(Conv2D(1, (1, 1), strides=(2, 2) ,padding='same', activation='relu'))
    model.add(Flatten())
    model.add(Dense(1,activation='sigmoid'))
    model.compile(loss='categorical_crossentropy', optimizer=”Adam”)
    model.compile('adam', loss=tf.keras.losses.BinaryCrossentropy(),
    metrics=['accuracy'])

    return(model)
```

- This is a pretty straightforward example of a classifier structure. But there are a lot of issues with it.



Resnet an Optimal Classifier

- Moving in a sequential manner, leads to omitting the actual data.
- Smaller problem in simpler networks with one or two hidden layers, unlike actual high level problem with dozens of layers.
- For example if we had a kernel which so overpowered the data that it caused the data to change so drastically that the coming layer lost the details of the previous layer.
- This was solved using the Resnet model. It is also a pre existing model with various version such as Resnet 50 etc, but we will focus on its structure.
- It basically focuses on skip connection that is for every layer, we will also have a interconnected layer to the output of the layer prior to it.
- That is along with being connected to the next layer, every layer is also connected to a layer after it. This enables us to main consistency of data.

Code For this

```
def residual_block(x, filters, kernel_size):
    y = layers.Conv2D(filters, kernel_size, padding='same', activation='relu')(x)
    y = layers.Conv2D(filters, kernel_size, padding='same')(y)
    y = layers.add([x, y])
    return layers.Activation('relu')(y)

def ResNet(input_shape, num_classes):
    inputs = layers.Input(shape=input_shape)
    x = layers.Conv2D(64, 7, strides=2, padding='same', activation='relu')(inputs)
    # Stack residual blocks
    for _ in range(1):
        x = residual_block(x, 64, 3)
    outputs = layers.Dense(num_classes, activation='softmax')(x)
    model = Model(inputs, outputs)
    return model

# Example usage
input_shape = (224, 224, 3)
num_classes = 1000
resnet_model = ResNet(input_shape, num_classes)
resnet_model.summary()
```

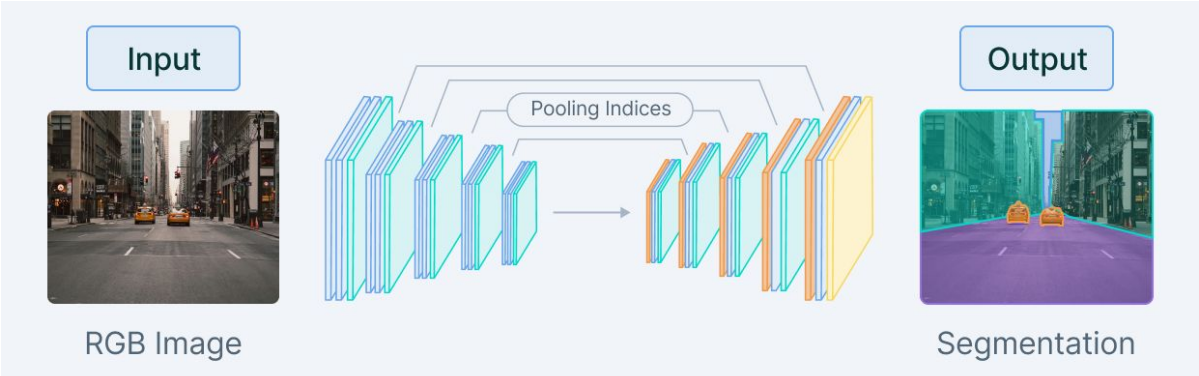
Image Transformation

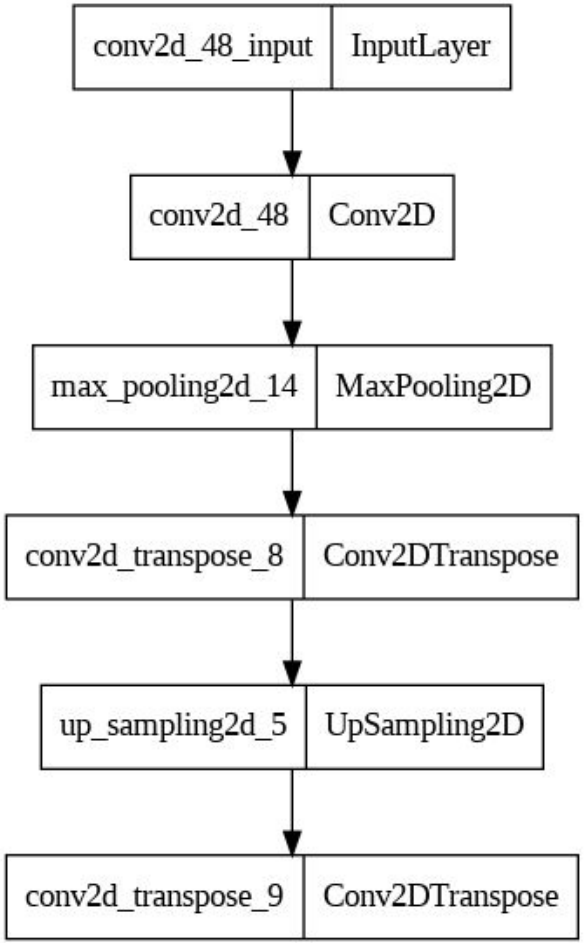
Image transformation is the process of changing existing images to match initial output. This can be of various kind:

- Medical Imaging: Improves diagnostic accuracy by preprocessing images with noise reduction and contrast enhancement.
- Image Compression: Reduces storage and bandwidth requirements for digital images while preserving visual quality.
- Image Restoration: Repairs degraded images through methods like inpainting and deblurring.
- Style Transfer: Transforms images by applying artistic styles, enabling creative expression.
- Image Registration: Aligns images from different sources for analysis in remote sensing and GIS.

A General Gist Of this

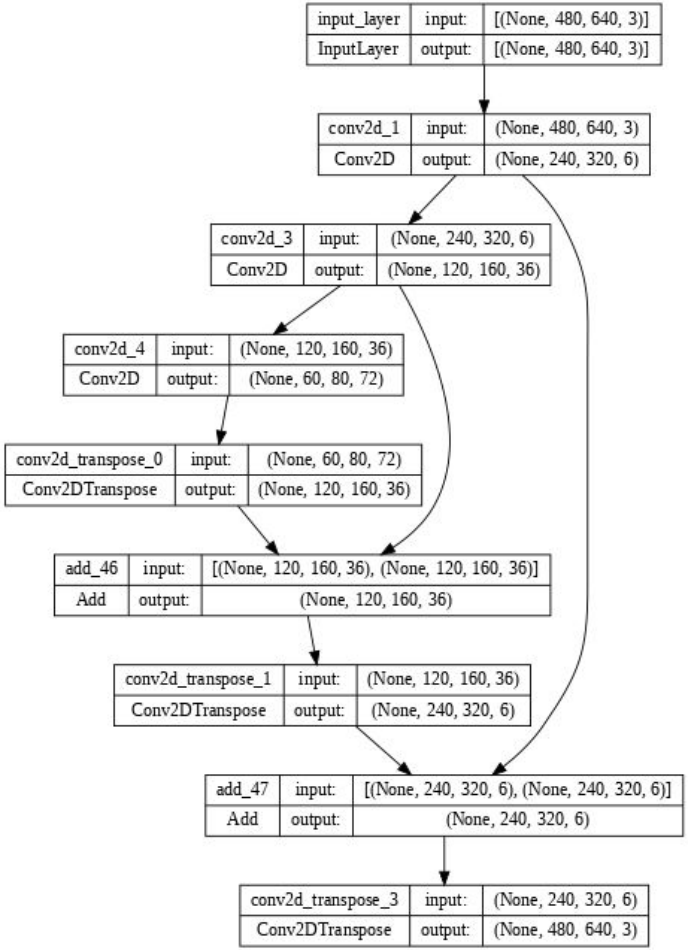
- The way we approach this is that we downsample and upsample the images and this changes the structure of images
- This process is called Encoding of images.
- Based on your requirements you can change the final layer that is increasing the resolution or decreasing the resolution.





An example of this

```
ef SequentialImageTransformer(input_shape):
 model = tf.keras.Sequential([

    # Encoder
    layers.Conv2D(64, (3, 3), activation='relu', padding='same', input_shape=input_shape),
    layers.MaxPooling2D((2, 2), padding='same'),

    # Decoder
    layers.Conv2DTranspose(64, (3, 3), activation='relu', padding='same'),
    layers.UpSampling2D((2, 2)),
    layers.Conv2DTranspose(3, (3, 3), activation='sigmoid', padding='same')
])
 return model

put_shape = (256, 256, 3)
nage_transformer = SequentialImageTransformer(input_shape)
nage_transformer.summary()
```



Auto Encoder An Optimal Transformer

- Same as the problem with the classifier, transformers face similar issue that they lose the gist of there original data and hene cause the model to “hallucinate”.
- To cover this issue we again use skip connection but rather we connect the initial layers to the final layers in a step-wise manner hence reminding us of what the initial data was and not to stray too far from it.
- The downsampling and upsampling of the data is interconnected and hence maintains the “Saintity” of the data.

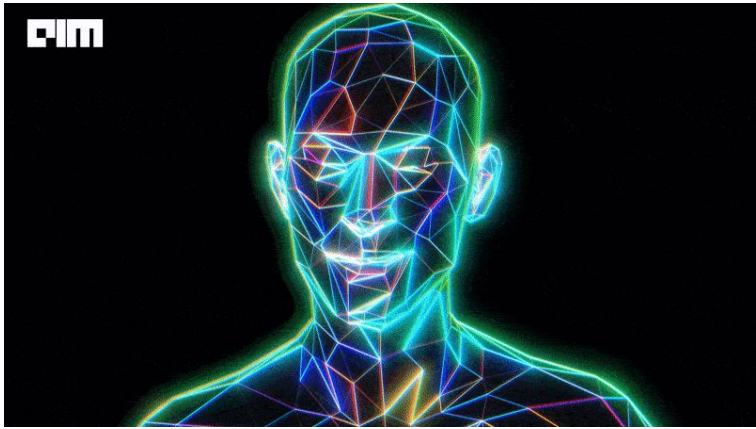
Code For this

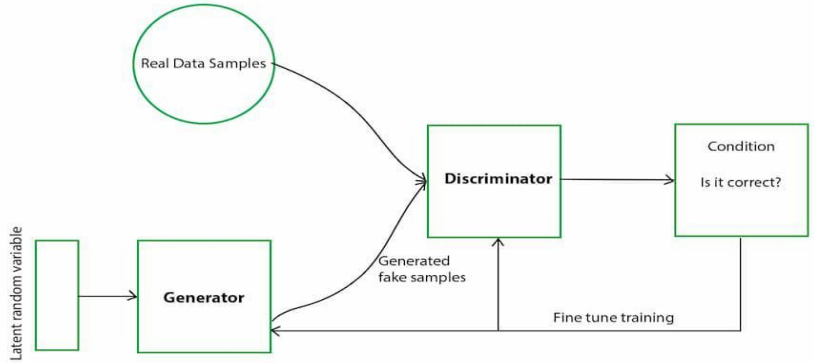
```
def genreator_network(input_dimension):
  input_layer = Input(input_dimension, name='input_layer')

  layer1 = Conv2D(6, (3, 3), strides=(2, 2) ,padding='same', activation='relu', name='conv2d_1')
  layer3 = Conv2D(36, (7, 7), strides=(2, 2), padding='same', activation='relu', name='conv2d_3')
  layer4 = Conv2D(72, (9, 9), strides=(2, 2), padding='same', activation='relu', name='conv2d_4')

  x = input_layer
  x1 = layer1(x)
  x5 = layer3(x1)
  x6 = layer4(x5)

  layer0_0 = Conv2DTranspose(36, (9, 9), strides=(2, 2), padding='same', activation='relu', name='conv2d_transpose_0')
  layer1_1 = Conv2DTranspose(6, (7, 7), strides=(2, 2), padding='same', activation='relu', name='conv2d_transpose_1')
  layer3_3 = Conv2DTranspose(3, (3, 3), strides=(2, 2), padding='same', activation='relu', name='conv2d_transpose_3')

  x0_0 = layer0_0(x6)
  x1_1 = layer1_1(Add()([x0_0, x5]))
  x5_5 = layer3_3(Add()([x1_1, x1]))

  model = tf.keras.models.Model(input_layer, x5_5)

  model.compile('adam', loss='mean_squared_error', metrics=['accuracy'])
  model.summary()

  return model
```

Generative Network

GAN networks or generative adversarial networks utilize classifier and transformer into one combined model.

- Image Generation: realistic images for various purposes like artwork and data augmentation.
- Image-to-Image Translation: Transform images between domains, e.g., black-and-white to color or sketches to photos.
- Super-Resolution: Enhance image quality
- Style Transfer: Apply artistic styles to images, popular in photography and digital art.
- Face Aging: Simulate aging or rejuvenation in human faces for entertainment and cosmetic simulations.
- Virtual Try-On: Visualize clothing and accessories on virtual models in e-commerce.
- Data Augmentation: Generate synthetic data to augment training datasets for machine learning models.
- Anomaly Detection: Identify abnormalities by learning normal data distributions, used in cybersecurity and quality control.
- Text-to-Image Synthesis: Generate images from textual descriptions, aiding storytelling and design.





Working of GAN network

- GANs (Generative Adversarial Networks) consist of two neural networks: a generator and a discriminator.
- Generator: It creates fake data (e.g., images) from random noise. The goal is to generate data that is indistinguishable from real data.
- Discriminator: It tries to distinguish between real and fake data. It learns to classify whether the input data is real (from the dataset) or fake (generated by the generator).
- During training, the generator aims to fool the discriminator by generating realistic data, while the discriminator improves its ability to distinguish real from fake data. This adversarial process continues until the generator produces data that is convincing enough to fool the discriminator, resulting in high-quality generated data.
- There is no actual structure of the GAN network rather it depends on which generator and discriminator are we using and what's the training process

An example of GAN networks

```
def gan_net(generator, discriminator, img_shape):

    discriminator.trainable=False
    inp=Input(img_shape)
    X=generator(inp)
    out=discriminator(X)
    gan = Model(inp,out)

gan.compile(loss='binary_crossentropy',optimizer='adam')
    print(gan.summary())
    return(gan)
```

```
for e in range(epoch):
    generated_images = generator.predict(np.array(hazy_images)/(255.0))

        real_labels = np.ones((size, 1))
        fake_labels = np.zeros((size, 1))
        d_loss_real =
discriminator.train_on_batch(np.array(target_images)/(255.0), real_labels)
        print("Real Train complete")
        d_loss_fake =
discriminator.train_on_batch(np.array(generated_images)/(255.0),
fake_labels)
        print("Fake Train complete")
        d_loss = np.add(d_loss_real, d_loss_fake)/3

        valid_labels = np.ones(((len(hazy_images), 1)))
        g_loss = gan.train_on_batch(np.array(generated_images)/(255.0),
valid_labels)
        print("GAN Train complete")

        print(f"Epoch {e,i}, D Loss: {d_loss[0]}, G Loss: {g_loss}")
```

Object Detection

Object Detection is one of the most complicated and diverse field of computer vision. Like we have been playing with legos uptil now, switching up or pre existing block design, Object detection throws it all out of the window and has its own set of workings. It requires a different kind of data preparation and also there is not set direction of computer vision. Some rely on image transformer so some use classifiers. So for the sake of being an introduction I will not be covering that. But here are some resources and walkthroughs if you are interested in that.

https://www.datacamp.com/blog/yolo-object-detection-explained

https://www.youtube.com/watch?v=yqkISICHH-U

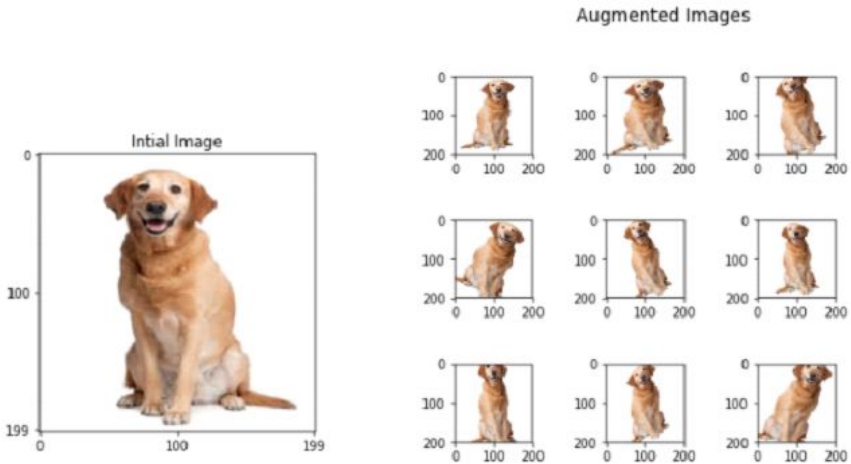
Data pre processing

Images stored on our devices are not viable to be trained on a ml model directly for that we need to change them into numerical arrays of specific sizes and that is what makes it possible for neural networks to train on them.

```
image_path = 'sample_image.jpg'

image = tf.keras.preprocessing.image.load_img(image_path)

image_array = tf.keras.preprocessing.image.img_to_array(image)
```

Data Augmentation

In case of computer vision more often than not, we face a very common issue that we don't have enough data or images. As collecting quality images is a long manual and tedious task.
For this we apply the concept of data augmentation. What this does is to tweak the existing images so as to the model can have a bit more variance. It a lot of times increase the amount of images by 2-5x times.




```
from tensorflow.keras.layers import
RandomFlip, RandomRotation,
Dropout, GaussianNoise,
RandomZoom

def build_model(input_shape=(256,
256, 3), num_classes=2):
  input_layer =
layers.Input(shape=input_shape)

  x =
RandomFlip("horizontal_and_vertical"
)(input_layer)

  x = RandomRotation(0.2)(x)

  x = GaussianNoise(0.1)(x)

  x = RandomZoom(0.2)(x)
```

Handling Overfitting

Overfitting is a pretty common problem that any ml engineer faces when working with models as it is very easy to over do the number of parameters/layers/kernels and that will cause the model to start to remember the patterns rather than understanding them.

To solve this we apply batch normalization which basically changes the signal to be in the range of a normalized distribution and finally add dropout layers which randomly switches off few of the neurons hence removing and dependency on any specific part of the network

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

def SequentialImageTransformer(input_shape):
  model = tf.keras.Sequential([
    # Encoder
    layers.Conv2D(64, (3, 3), activation='relu', padding='same', input_shape=input_shape),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2), padding='same'),
    layers.Dropout(0.2),  # Add dropout

    # Decoder
    layers.Conv2DTranspose(64, (3, 3), activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.UpSampling2D((2, 2)),
    layers.Conv2DTranspose(3, (3, 3), activation='sigmoid', padding='same')
  ])
  return model
```

Q/A

Free to ask Any Questions

THANK YOU