Image processing

Topics Covered

Image Scaling for ANN

Building an effective neural network model requires careful consideration of the network architecture as well as the input data format. This article deals with the latter.

The most common image data input parameters are the number of images, image height, image width, number of channels, and the number of levels per pixel.

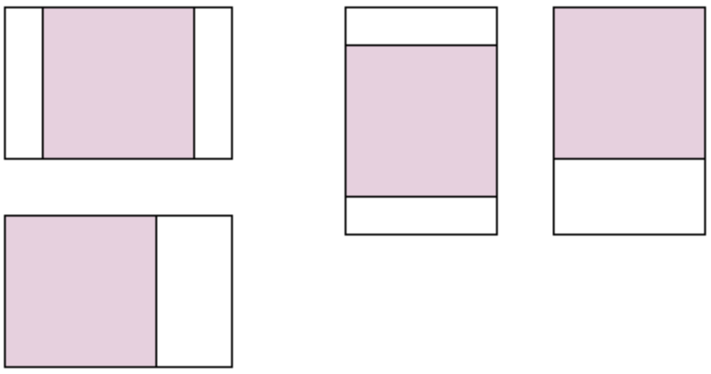
Typically, we have 3 channels of data corresponding to the colors Red, Green, Blue (RGB) Pixel levels are usually [0,255]. For this exercise let’s choose the following values

- number of images = 100
- image width, image height =100
- 3 channels, pixel levels in the range [0–255]

Let's look at one popular data-set. Labeled Faces in the Wild is a database of facial images, originally designed for studying the problem of face recognition. The data-set contains more than 13,000 images of faces collected from the web, and each face has been labeled with the name of the person pictured. Data-set images need to be converted into the described format. After downloading the image data, notice that the images are arranged in separate sub-folders, by name of the person. We'll need to get all the photos into a common directory for this exercise. Let's take the first 100 images and copy them into a working directory. The data contains faces of people 'in the wild', taken with different light settings and rotation. They appear to have been centered in this data set, though this need not be the case.

There are a number of pre-processing steps we might wish to carry out before using this in any Deep Learning project. The paragraphs below list some of the most common.

Uniform aspect ratio: One of the first steps is to ensure that the images have the same size and aspect ratio. Most of the neural network models assume a square shape input image, which means that each image needs to be checked if it is a square or not, and cropped appropriately. Cropping can be done to select a square part of the image, as shown. While cropping, we usually care about the part in the center.

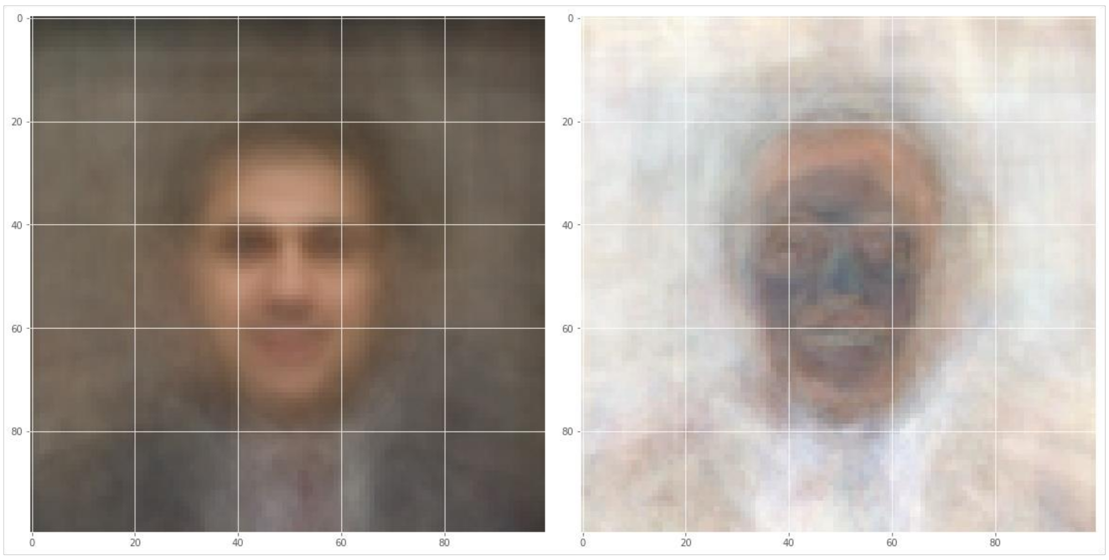


Back to our data-set. Sampling a few pictures randomly we see each image in the data-set appears to have dimensions of 250 by 250 pixels, which does simplify things.

Example image: Square 250 x 250

Image Scaling: Once we've ensured that all images are square (or have some predetermined aspect ratio), it's time to scale each image appropriately. We've decided to have images with width and height of 100 pixels. We'll need to scale the width and height of each image by a factor of 0.4 (100/250). There are a wide variety of up-scaling and down-scaling techniques and we usually use a library function to do this for us.

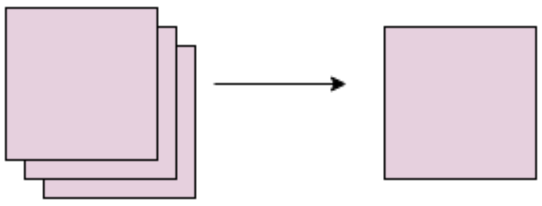
Mean, Standard Deviation of input data: Sometimes it's useful to look at the 'mean image' obtained by taking the mean values for each pixel across all training examples. Observing this could give us insight into some underlying structure in the images. For example, the mean image from the first 100 images of our data-set is shown below to the left. Clearly, this has the loose impression of a human face and lets us conclude that the faces are somewhat aligned to the center and are of comparable size. We may choose to augment our data with **perturbed** images if we don't want our input data to have this innate structure. The standard deviation of all images is shown to the right. Higher variance values show up whiter, so we see that the pictures vary a lot at the boundaries compared to the center.



Images depicting mean (left) and standard deviation (right) of the set of data inputs

Normalizing image inputs: Data normalization is an important step which ensures that each input parameter (pixel, in this case) has a similar data distribution. This makes convergence faster while training the network. Data normalization is done by subtracting the mean from each pixel and then dividing the result by the standard deviation. The distribution of such data would resemble a Gaussian curve centered at zero. For image inputs we need the pixel numbers to be positive, so we might choose to scale the normalized data in the range [0,1] or [0, 255]. For our data-set example, the following montage represents the normalized data.

Dimensionality reduction: We could choose to collapse the RGB channels into a single gray-scale channel. There are often considerations to reduce other dimensions, when the neural network performance is allowed to be invariant to that dimension, or to make the training problem more tractable.



Data augmentation: Another common pre-processing technique involves augmenting the existing data-set with perturbed versions of the existing images. Scaling, rotations and other affine transformations are typical. This is done to expose the neural network to a wide variety of variations. This makes it less likely that the neural network recognizes unwanted characteristics in the data-set.

TensorFlow Image Data generator - Usage and Benefits

Deep Learning has helped us achieve state-of-the-art performance on computer vision tasks. Nevertheless, they are susceptible to overfitting. Let’s go through a simple example

Say you want to build a neural network to identify Toyota Corollas. You have collected training data, and your images are very similar to the ones shown below.

Toyota Corolla. Source (left): Motortrend, Source (right): Car And Driver

You want to test your model in the wild. Say you come across the following image.

Toyota Corolla. Source: edmunds

Notice the car is facing the opposite direction compared to your training images. Given such an image, your model may fail to identify the car is a Toyota Corolla. When you have a training dataset with images that look very similar to each other, your model will fail to learn relevant features and will be unable to generalize to images it sees in the real-world. To avoid this problem, we can apply data augmentation techniques to alleviate over-fitting and enhance the model's generalizability.

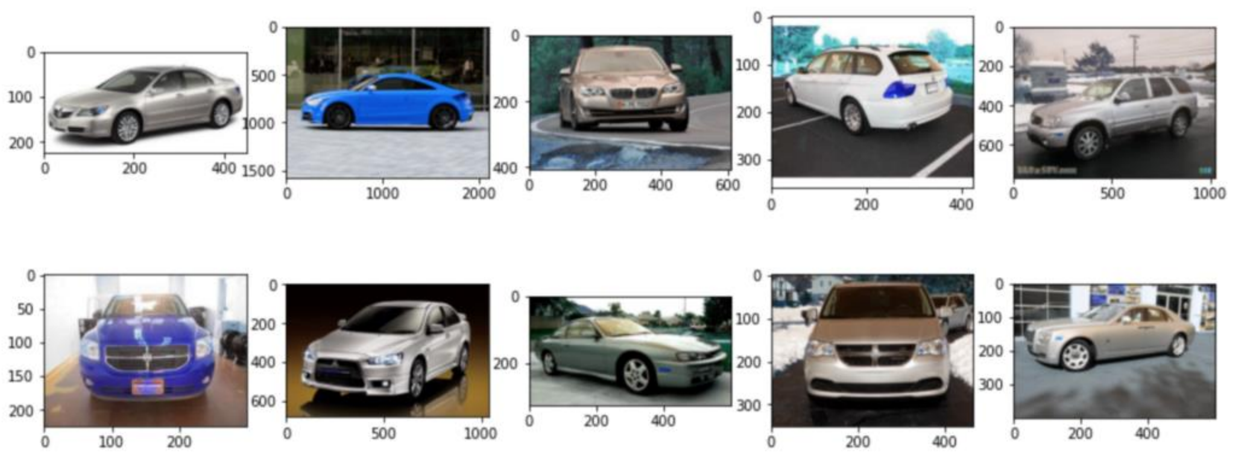
It is essential not to get carried away with data augmentation and apply transformations that are outside the context of the real world.

Toyota Corolla (flipped vertically). Source: edmunds

Apply transformations that are within the context of the problem you are attempting to solve.

Data augmentation using ImageDataGenerator

We will use the Stanford Car Dataset for this tutorial.
The dataset is large. For this tutorial, we will sample a few images to understand data augmentation.



Sampled images

Next, we will define the parameters of the image generator.

```
DATA_AUG_BATCH_SIZE = 2  # batch size for data augmentation
img_size = (224, 224)  # input image size to model
# Number of steps to perform data augmentation
n_steps_data_aug = np.ceil(df_sample.shape[0]/DATA_AUG_BATCH_SIZE).astype(int)
```

- We define a batch size for the image generator. This is analogous to batches in model training.
- img_size is the image size required by the model. If you are using transfer learning specific models have their own specifications for the input size of the images to the model.
- n_steps_data_aug is analogous to epochs in model training. In this tutorial, we generate a new image corresponding to every sampled image. Iterating the generator n_steps_data_aug times generates the required number of images.

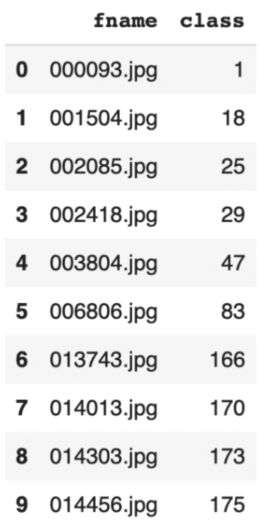
```
# Image data generator. Transformations to be applied
datagen = ImageDataGenerator(rescale=1./255,
rotation_range=40,
width_shift_range=0.2,
height_shift_range=0.2,
zoom_range=0.2,
horizontal_flip=True,
brightness_range=[0.4, 1.0],
fill_mode='nearest')
```

- rescale multiplies each pixel value with the rescale factor. It helps with faster convergence.
- rotation_range rotates the image randomly with maximum rotation angle, 40.
- width_shift shifts the image by the fraction of the total image width, if float provided.
- height_shift shifts the image by the fraction of the total image height, if float provided.
- zoom_range zooms into the image. A random number is chosen in the range, [1-zoom_range, 1+zoom_range].
- horizontal_flip randomly flips images, horizontally.
- brightness_range alters the brightness of the image. A random number is picked in the range provided.
- fill_mode fills the empty space in an image using various techniques selected by the user. Empty spaces occur when images are shifted along their width or height.

There are a few ways to pass the images to the data generator.

- flow: You pass image data and label data as arrays.
- flow_from_dataframe: You pass a dataframe and a path to the directory that contains the images. The dataframe has two columns, one column contains the file path relative to the directory path, and the second column contains the labels.
- flow_from_directory: You pass the directory path that contains the images categorized by subdirectories. Each subdirectory corresponds to a label — an example of a file path structure, imgs/dogs/img01.png, imgs/cats/img01.png. There are two classes, dogs and cats.

We will use flow_from_dataframe to feed the images to the data generator. Below you can see how the dataframe is structured. In the Google Colab notebook, you can look at the code to see how I create this dataframe.



	fname	class
0	000093.jpg	1
1	001504.jpg	18
2	002085.jpg	25
3	002418.jpg	29
4	003804.jpg	47
5	006806.jpg	83
6	013743.jpg	166
7	014013.jpg	170
8	014303.jpg	173
9	014456.jpg	175

Dataframe with image file names and corresponding labels. Source: Image by author

```
# Feed images to the data generator
aug_gen = datagen.flow_from_dataframe(dataframe=df_sample, directory=img_path,
save_to_dir=aug_img_path, save_prefix='aug', save_format='jpeg', x_col="fname",
y_col="class", batch_size=DATA_AUG_BATCH_SIZE, seed=SEED, shuffle=False,
class_mode="categorical", target_size=img_size)
Output:
Found 10 validated image filenames belonging to 10 classes.
```

The above code feeds the images to the data generator. df_sample is the dataframe I have shown above. If you would like to save the images to a directory, provide a path to save_to_dir. ImageDataGenerator saves the generated images with file names defined by a pattern:prefix_idx_randn

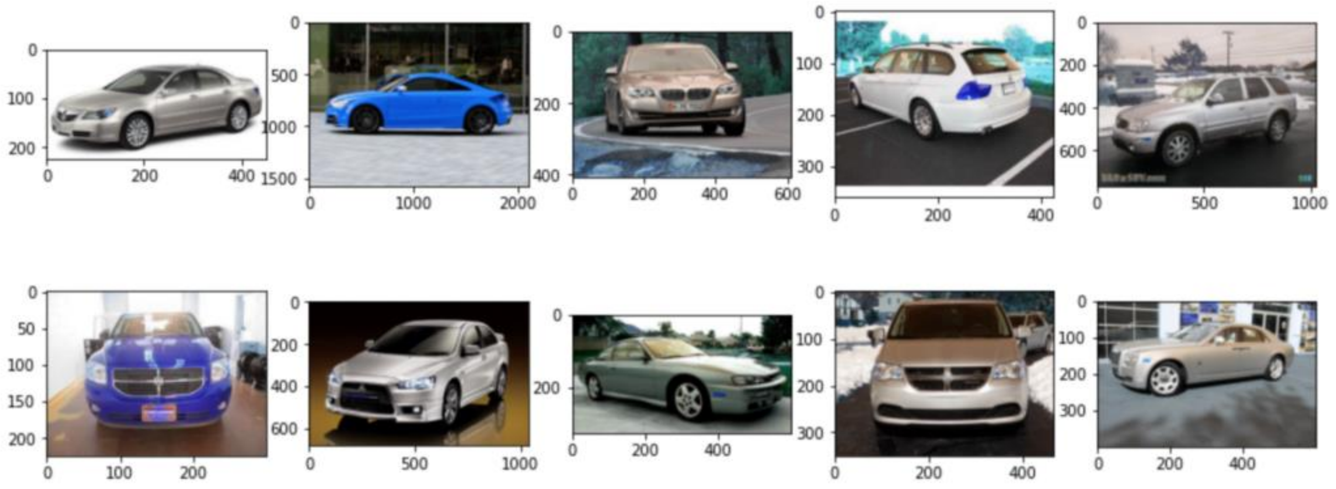
- **prefix:** prefix provided by user
- **idx:** index of data point in the dataframe fed to generator
- **randn:** random number

```
# Generated file names
['aug_1_5664789.jpeg',  'aug_7_631054.jpeg',  'aug_5_6028627.jpeg',
'aug_4_8351445.jpeg',  'aug_0_9784633.jpeg',  'aug_6_5123896.jpeg',
'aug_2_3921660.jpeg',  'aug_3_7757856.jpeg',  'aug_8_8252162.jpeg',
'aug_9_5698539.jpeg']
```

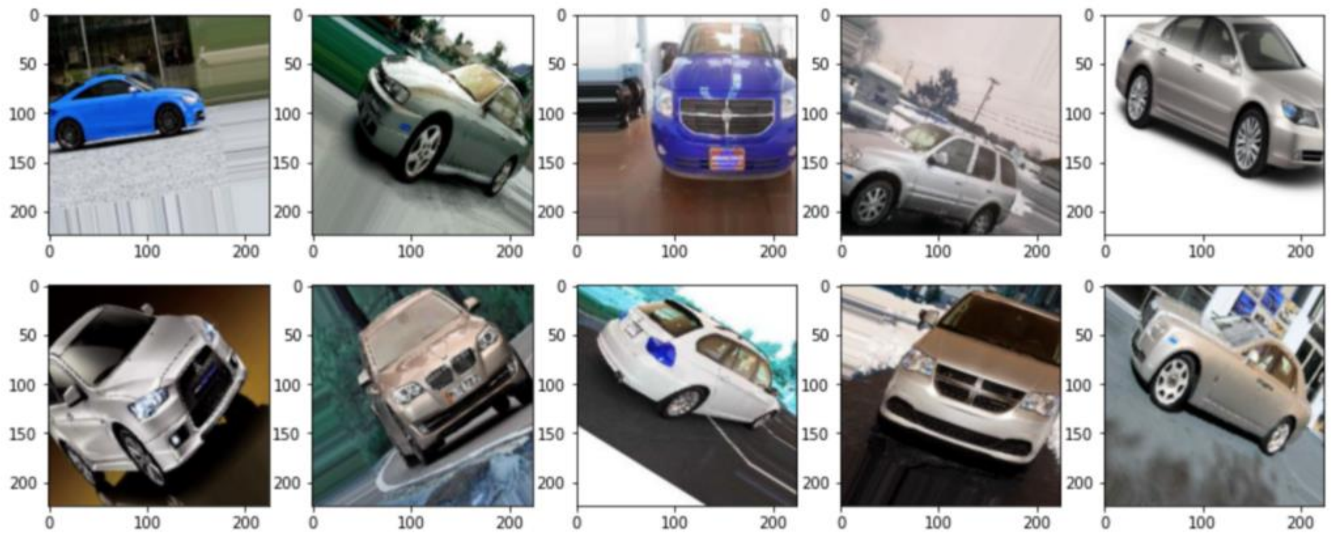
Generate new images.

```
# Run the data generator
for i in range(n_steps_data_aug):
    next(aug_gen)
# Number of augmented images created
aug_file_nm = os.listdir(aug_img_path)
number_files = len(aug_file_nm)
print("Number of new images generated: {}".format(number_files))
Output:
Number of new images generated: 10
```

We use n_steps_data_aug to control how many times to run the data generator. Earlier in the article, I mentioned that we want to generate one new image for every sampled image.

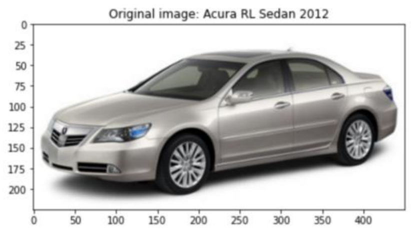


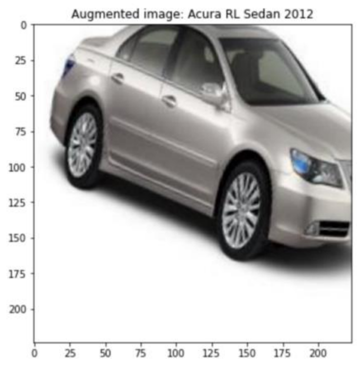
Original images


Transformed images.

Some of the images may not resemble real-world data, so be very careful when applying transformations to images.

Lastly, if you would like to associate the appropriate classes to the generated images, you can split the generated file name, extract the index and determine the label.





Using the index you can determine the label associated to the generated image.

Benefits

- **ImageDataGenerator** class provides a quick and easy way to augment your images. It provides a host of different augmentation techniques like standardization, rotation, shifts, flips, brightness change, and many more.
- However, the main benefit of using the Keras ImageDataGenerator class is that it is designed to provide real-time data augmentation. Meaning it is generating augmented images on the fly while your model is still in the training stage. How cool is that!
- ImageDataGenerator class ensures that the model receives new variations of the images at each epoch. But it only returns the transformed images and does not add it to the original corpus of images. If it was, in fact, the case, then the model would be seeing the original images multiple times which would definitely overfit our model.
- Another advantage of ImageDataGenerator is that it requires lower memory usage. This is so because without using this class, we load all the images at once. But on using it, we are loading the images in batches which saves a lot of memory.

Common Problems in Computer Vision

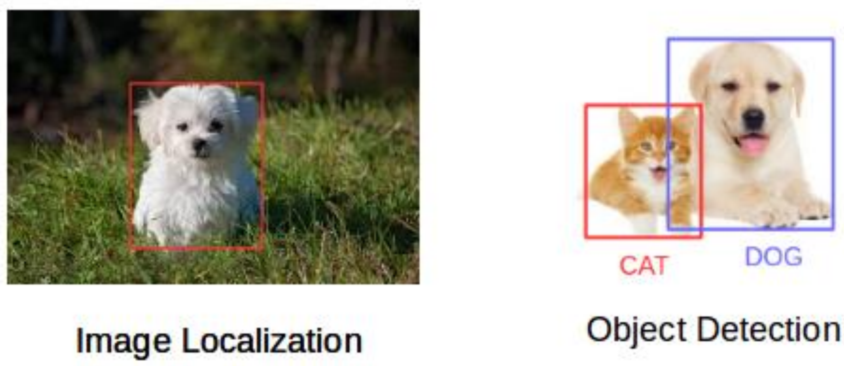
In the computer vision field, one of the most common doubt which most of us have is what is the difference between image classification, object detection and image segmentation. So lets break down these terminologies which will help you to understand the difference between each of them. Let’s start with understanding what is image classification:
Consider the below image:

You will have instantly recognized it. It’s a dog. Take a step back and analyze how you came to this conclusion. You were shown an image and you classified the class it belonged to (a dog, in this instance). And that, in a nutshell, is what **Image Classification** is all about.

As you saw, there’s only one object here: a dog. We can easily use image classification model and predict that there’s a dog in the given image. But what if we have both a cat and a dog in a single image?

We can train a multi-label classifier, in that instance. Now, there’s another caveat - we won’t know the location of either animal/object in the image.

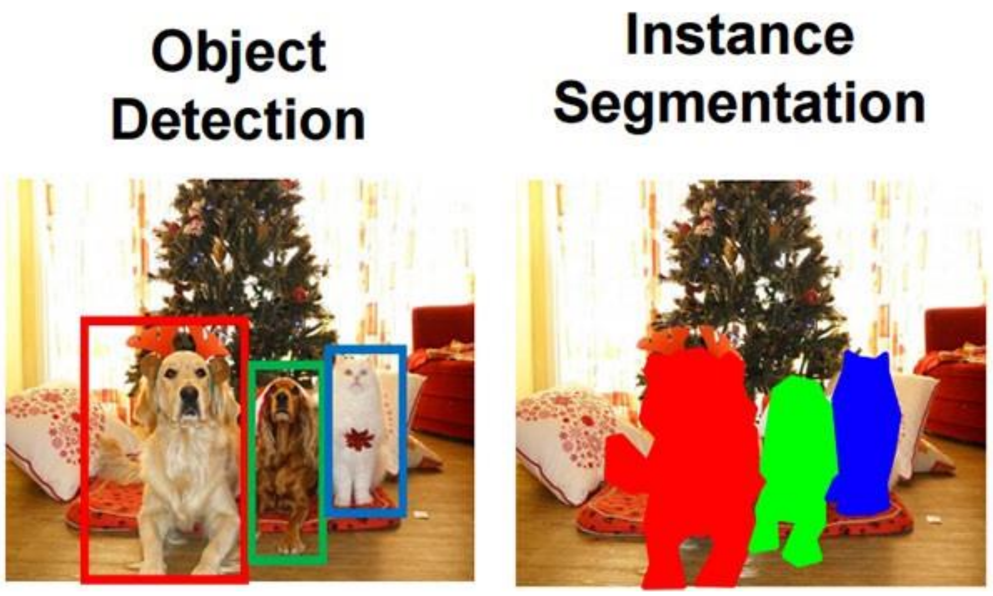
That’s where **Image Localization** comes into the picture. It helps us to identify the location of a single object in the given image. In case we have multiple objects present, we then rely on the concept of **Object Detection**. We can predict the location along with the class for each object using OD.



Before detecting the objects and even before classifying the image, we need to understand what the image consists of. This is where **Image Segmentation** is helpful.

We can divide or partition the image into various parts called segments. It’s not a great idea to process the entire image at the same time as there will be regions in the image which do not contain any information. By dividing the image into segments, we can make use of the important segments for processing the image. That, in a nutshell, is how **Image Segmentation** works.

An image, as you must have known, is a collection or set of different pixels. We group together the pixels that have similar attributes using image segmentation:



By applying Object Detection models, we will only be able to build a bounding box corresponding to each class in the image. But it will not tell anything about the shape of the object as the bounding boxes are either rectangular or square in shape.

Image Segmentation models on the other hand will create a pixel-wise mask for each object in the image. This technique gives us a far more granular understanding of the object(s) in the image.

We hope you now have a clear understanding of what is Image Classification, Image Localization, Object Detection and Image Segmentation. To quickly summarize:

Image Classification helps us to classify what is contained in an image. Image Localization will specify the location of single object in an image whereas Object Detection specifies the location of multiple objects in the image. Finally, Image Segmentation will create a pixel wise mask of each object in the images. We will be able to identify the shapes of different objects in the image using Image Segmentation.

So Now Lets Understand Other Common Problems Faced In Industry For Implementation Of Computer Vision

1. Suboptimal Hardware Implementation

Computer vision applications are a double-pronged setup, featuring both software algorithms and hardware systems (cameras and often IoT sensors). Failure to properly configure the latter leaves you with significant blind spots.

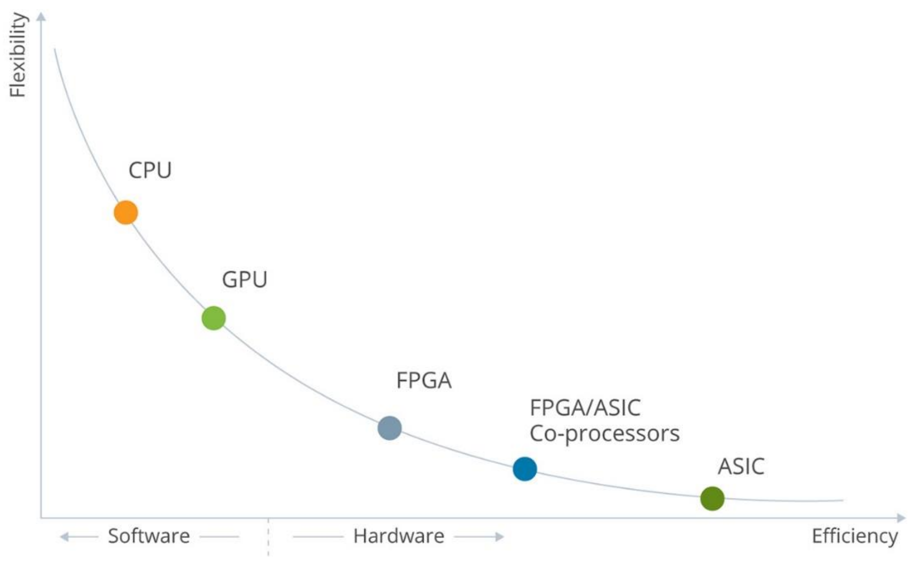
Hence, you need to first ensure that you have a camera, capable of capturing high-definition video streams at the required rate per second. Then you need to ensure a proper angle for capturing the object's position. When you test the setup, make sure that the cameras:

- Cover all areas of surveillance.
- Are properly positioned. The frame captures the object from the correct angle.
- All the necessary configurations are incheck.

Apart from cameras, you will also need to ensure proper hardware acceleration for computer vision software. Machine learning and deep learning algorithms are computationally demanding and require optimized memory architecture for faster memory access. Respectively, stack up on CPUs and GPUs.

CPUs, in particular, are the best fit for computer vision tasks (especially those involving deep learning and neural networks) as they support effective serial computations with complex task scheduling. The chart below compares the efficacy rates of different hardware acceleration solutions for computer vision:

Computer Vision Solution Flexibility and Efficiency for Different Hardware Implementations



2. Underestimating the Volume and Costs of Required Computing Resources

The boom in popularity of AI technologies (ML, DL, NLP, and computer vision among others) have largely commoditized access to best-in-class computer vision libraries and deep learning frameworks, distributed as open-source solutions.

For skilled data scientists, coming up with an algorithmic solution to computer vision problems is no longer an issue (even if we are talking about advanced computer vision scenarios such as autonomous driving). The modelling know-how is already available. It is computing power that often constrains model scaling and large-scale deployments.

We noticed that when going forward with computer vision technology, many business leaders underestimate:

- Project hardware needs
- Cloud computing costs

As a result, some invest in advanced algorithm development, but subsequently, fail to properly train and test the models because the available hardware doesn't meet the project technical requirements.

To better understand the costs of launching complex machine learning projects, let us take a look at the current state-of-the-art image recognition algorithm. A semi-supervised learning approach Noisy Student (developed by Google) relies on convolutional neural networks (CNN) architecture and over 480M parameters for processing images. Understandably, such operations require immense computing power. Researchers from AI21 Labs estimated the average cost of training similar deep learning networks in the cloud (using AWS, or Azure or GCP):

- With 110 million parameters: $2.5k – $50k
- With 340 million parameters: $10k – $200k
- With 1.5 billion parameters: $80k – $1.6m

Please be aware that the above only accounts for model training costs (without retraining), but the ballpark estimates include the costs of performing hyper-parameter tuning and doing multiple runs per set.

The best computer vision software is power-hungry. Hence, make sure that you factor in all the computing costs, invest in proper hardware, and proactively optimize the costs of cloud computing resources, allocated to computer vision.

3. Short Project Timelines

When estimating the time-to-market for computer vision applications, some leaders overly focus on the model development timelines and forget to factor in the extra time needed for:

- Camera setup, configuration, and calibration
- Data collection, cleansing, and validation
- Model training, testing, and deployment

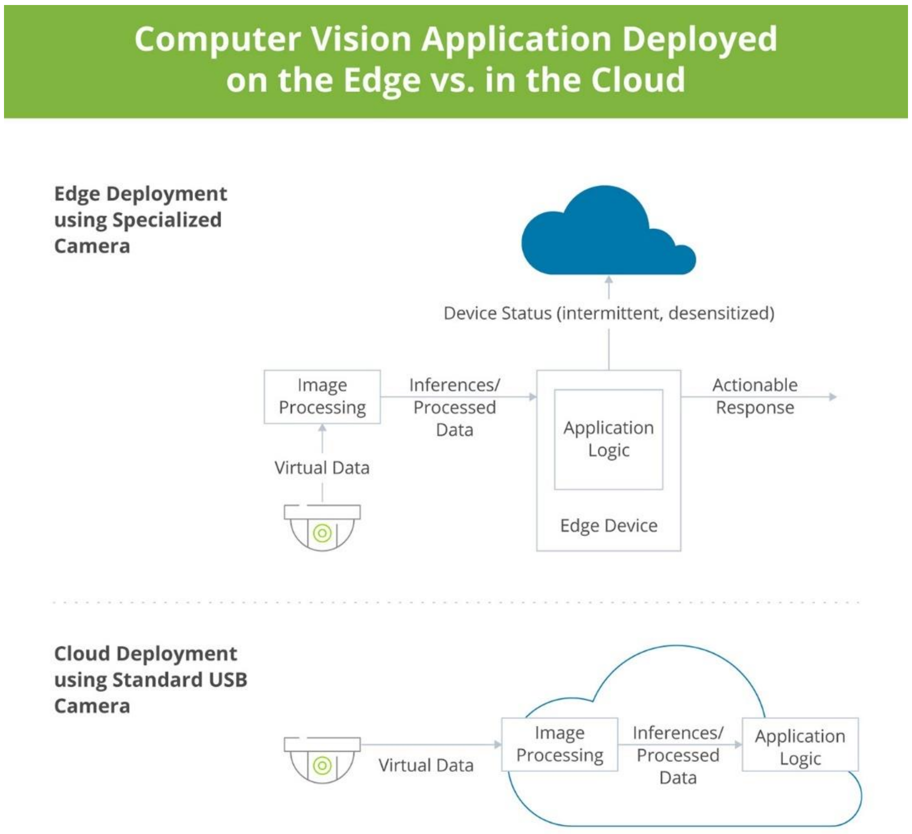
Combined, these factors can significantly shift project timelines. How can you make more accurate estimates?

A 2019 study by Algorithmia brought some sobering insights:

- In 2019, 22% of enterprises successfully productized their AI models.
- Among this cohort, 22% needed one to three months to bring one ML model into production (given that they have a mature MLOps process).
- Another 18% required three or more months for deployments.

The deployment stage of computer vision technologies, especially the complex systems running in hybrid cloud environments, requires careful planning and preparation. In particular, consider the pros and cons of different target deployment environments such as:

- **Cloud platforms:** Deploying computer vision algorithms to the cloud makes sense if you need the ability to rapidly scale the model performance, maximize uptime, and maintain close proximity to the data lakes the model relies on. On the other hand, the cloud may not always be suitable for processing sensitive data. Plus, the computing costs can rise sharply without constant optimization.
- **Local hardware:** It is possible to run computer vision solutions on-premises too. The option works best for businesses with idle server resources who would like to re-deploy them for testing pilot deployments and small-scale projects. The con, however, is expensive and/or constricted scalability.
- **Edge:** Running computer vision models on the edge devices, connected or embedded into the main product (e.g., the camera system, manufacturing equipment, or drone) can significantly reduce latency for transferring high-definition visuals (video in particular). However, edge deployments also require more complex system architecture and advanced cybersecurity measures in place to protect data transfers. One of the solutions to these issues is to use peripheral cameras that do not have internet connectivity but can be meshed with edge devices to expand the application functionality.


Computer Vision Application Deployed on the Edge vs. in the Cloud
Edge Deployment using Specialized Camera
Device Status (intermittent, desensitized)
Image Processing
Inferences/ Processed Data
Application Logic
Actionable Response
Virtual Data
Edge Device
Cloud Deployment using Standard USB Camera
Virtual Data
Image Processing
Inferences/ Processed Data
Application Logic

4. Lack of Quality Data

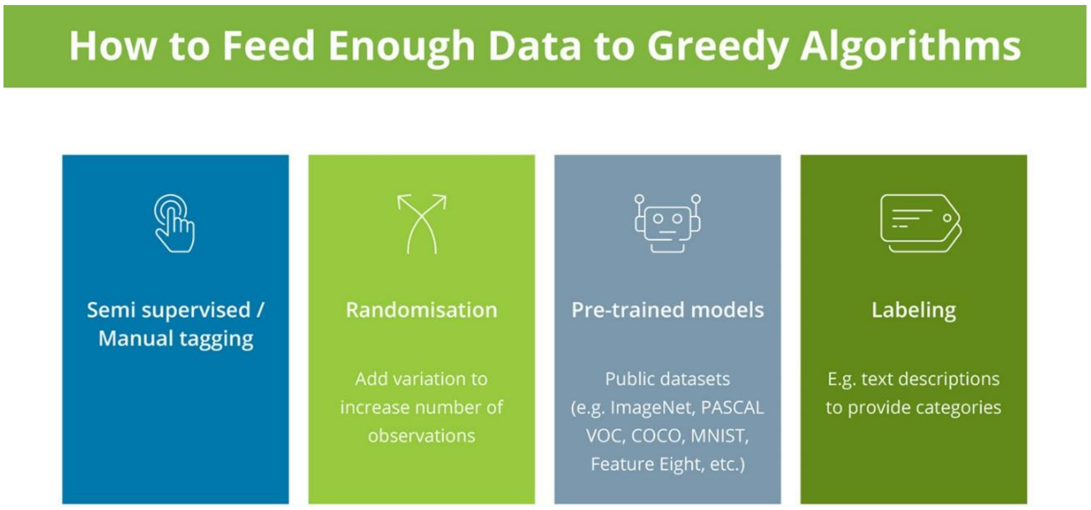
Labeled and annotated datasets are the cornerstone to successful CV model training and deployment. General-purpose public datasets for computer vision are relatively easy to come by. However, companies in some industries (for example, healthcare) often struggle to obtain high-quality visuals for privacy reasons (for example, CT scans or X-ray images). Other times, real-life video and images are hard to come by (for example, footage from road accidents and collisions) in sufficient quantity.

Another common constraint is the lack of a mature data management process within organizations which makes it hard to obtain proprietary data from siloed systems to enhance the public datasets and collect extra data for re-training.

In essence, businesses face two main issues when it comes to data quality for computer vision projects:

- Insufficient public and proprietary data available for training and re-training
- The obtainable data is of poor quality and/or requires further costly manipulations (e.g., annotation).

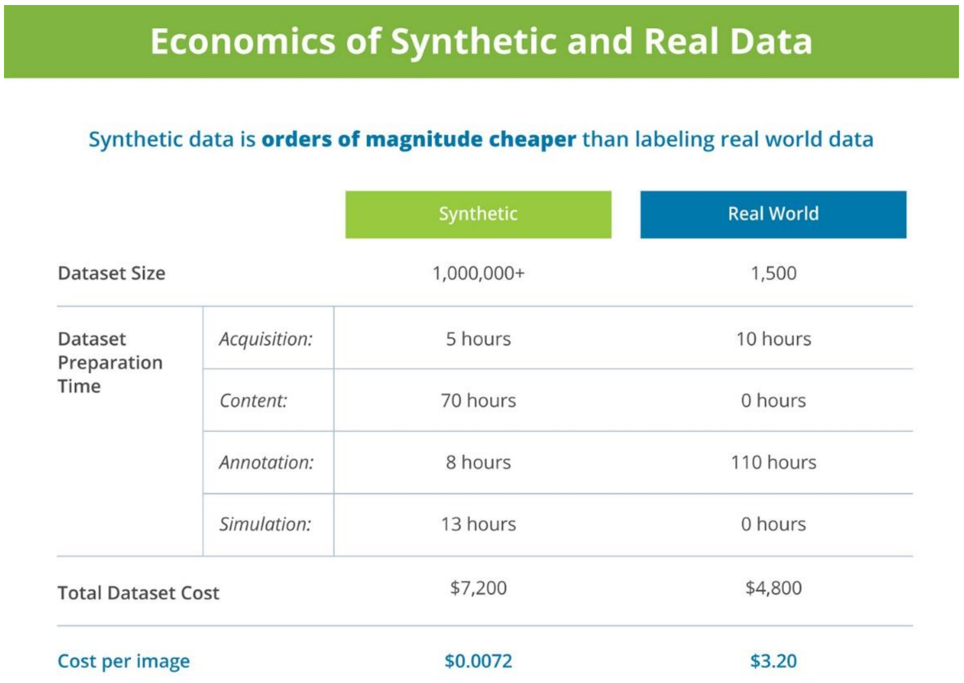
The better news is that you can improve your datasets programmatically to enhance the model performance:



How to Feed Enough Data to Greedy Algorithms

Semi supervised / Manual tagging	Randomisation	Pre-trained models	Labeling
	Add variation to increase number of observations	Public datasets (e.g. ImageNet, PASCAL VOC, COCO, MNIST, Feature Eight, etc.)	E.g. text descriptions to provide categories

Synthetic data is another novel approach to improving data quality for computer vision projects. Synthetic datasets are programmatically generated visuals that can be used for more effective model training and retraining as the chart below illustrates:



Economics of Synthetic and Real Data

Synthetic data is **orders of magnitude cheaper** than labeling real world data

		Synthetic	Real World
Dataset Size		1,000,000+	1,500
Dataset Preparation Time	*Acquisition:*	5 hours	10 hours
	Content:	70 hours	0 hours
	Annotation:	8 hours	110 hours
	Simulation:	13 hours	0 hours
Total Dataset Cost		$7,200	$4,800
Cost per image		$0.0072	$3.20

Synthetic data is more affordable to produce, but how does its usage affect model accuracy? Several use cases illustrate that models trained on mixed datasets (featuring both real-world and synthetic data) often perform better than those using real-world data only.

- A robot arm trained for grabbing accuracy using synthetic data only can achieve accurate performance results with real-life objects.
- Google's Waymo is said to be validated and cross-trained with synthetic data.
- Another subdivision at Google trained a computer vision model for detecting items on the supermarket shelves using synthetic data. The algorithm did better than one trained on real-life data.

5. Inadequate Model Architecture Selection

Let us be honest: most companies cannot produce sufficient training data and/or don't have high MLOps maturity for churning out advanced computer vision models, performing in line with the state of the art benchmarks. Respectively, when it comes to project requirements gathering, the line of business leaders often set overly ambitious targets for the data science teams without assessing the feasibility of reaching such targets.

As a result, when it comes down to the later development stages, some leaders may realize that the developed model:

- Does not meet the stated business objectives (due to earlier overlooked requirements in terms of hardware, data quality/volume, computing resources)
- Demands too much computing power, which makes scalability cost-inhibitive.
- Delivers insufficient results in terms of accuracy/performance and cannot be used in business settings.
- Relies on a custom architecture that is too complex (and expensive) to maintain or scale in production.

Summary

- Deep learning uses neural nets with a lot of hidden layers (dozens in today’s state of the art) and requires large amounts of training data.
- Data normalization is an important step which ensures that each input parameter (pixel, in this case) has a similar data distribution. This makes convergence faster while training the network.
- Computer vision applications are a double-pronged setup, featuring both software algorithms and hardware systems (cameras and often IoT sensors). Failure to properly configure the latter leaves you with significant blind spots.
- The cloud may not always be suitable for processing sensitive data. Plus, the computing costs can rise sharply without constant optimization.