Problem Statement

Let's learn how a self-driving car can "see" its surroundings using semantic image segmentation.

We'll cover the following

Introduction

The definition of a self-driving car is a vehicle that drives itself, with little or no human intervention. Its system uses several sensory receptors to perceive the environment. For instance, it identifies the drivable area, weather conditions, obstacles ahead and plans the next move for the vehicle accordingly.

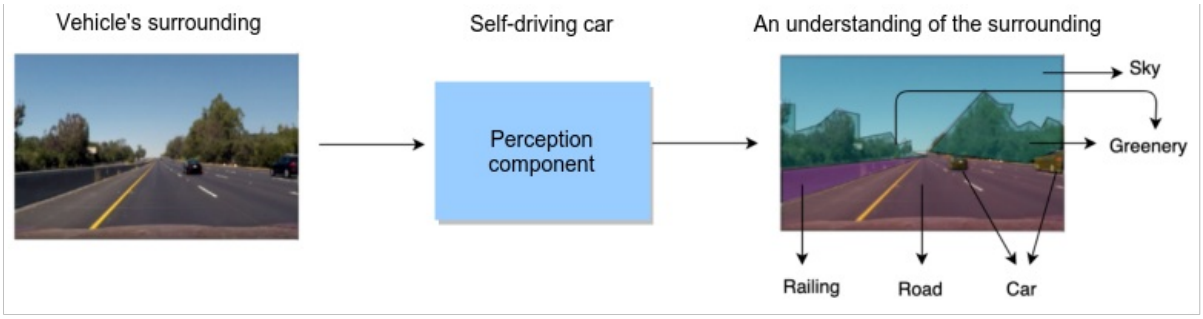
There are different levels of autonomy in such vehicles. Tesla currently implements assisted driving, where the driving is autonomous, but someone is behind the wheel. Waymo (Google's self-driving car), in contrast, is aiming for complete autonomy under all driving conditions (no driver required). Beyond human transportation, self-driving vehicles can also be utilized as a service for various purposes, e.g., Nuro is building self-driving vehicles for local goods transportation.

> Self-driving vehicles are perfect real-world systems for handling multi-sensory inputs that focus primarily on computer vision-based problems (e.g., object classification/ detection/ segmentation), using machine learning.

Now that you have some context, let's look at the problem statement.

Problem statement

The interviewer has asked you to design a self-driving car system focusing on its perception component (*semantic image segmentation in particular*). This component will allow the vehicle to perceive its environment and make informed driving decisions. Don't worry we will describe semantic image segmentation shortly.



Perform semantic image segmentation on the driving scene

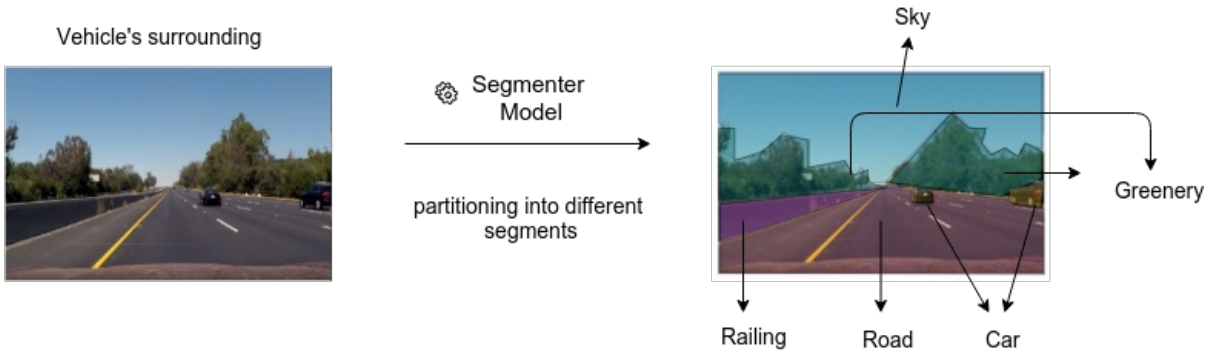
Interviewer's questions

The interviewer might ask the following questions about this problem, narrowing the scope of the question each time.

1. How would you approach a computer vision-based problem in terms of the self-driving car?
2. How would you train a semantic image segmentation model for autonomous driving?
3. How will the segmentation model fit in the overall autonomous driving system architecture?
4. How would you deal with data scarcity in imaging dataset?
5. How would you best apply data augmentation on images?
6. What are the best model architectures for image segmentation tasks?
7. Your optimized deep learning model gives a high performance on the validation set, but it fails when you take the self-driving car on the road. Why? How would you solve this issue?

> Answers
> `Q1` will be answered shortly when we talk about *subtasks*. `Q2` and `Q3` will be answered in the *architectural components* lesson. `Q4`, `Q5` and `Q7` will be answered in the *training data generation* lesson. `Q6` will be answered in the *modelling* lesson.
>
> Any other similar (multi-sensory) computer vision problem asked during the interview can be solved in a similar manner to the self-driving car problem.

Let's have a quick look at some of the major software and hardware components of the self-driving vehicle system before you attempt to answer the first question.



Semantic image segmentation of vehicle surroundings

Hardware support

Let's look at the *sensory receptors* that allow the vehicle to "see" and "hear" its environment and plan its course of action accordingly.

Camera

The camera provides the system with high-resolution visual information of its surrounding. The visual input from a frame-based camera can also be used to get an estimate of the depth/distance of objects from the self-driving car. However, the usefulness of the camera depends heavily on the lighting conditions and may not work optimally in the night scenarios.

Radar

The radar uses radio waves' reflections from solid objects to detect their presence. It adds depth on top of the visual information so that the system can accurately sense the distance between various objects. It works well in varied conditions, like low light, dirt and cloudy weather, as well as long operating distances.

Lidar

Lidar, in contrast, uses laser light reflecting off objects to create a *precise* 3D image of the environment, while measuring the distance at which objects are positioned. They are good at even detecting *small objects*. However, they do not work well during *rainy/foggy weather* and are quite *expensive*.

> One approach that exists in the industry is to get input for the system using a fusion of these sources to overcome their shortcomings.

The following diagram provides a good comparison of each type of hardware support.

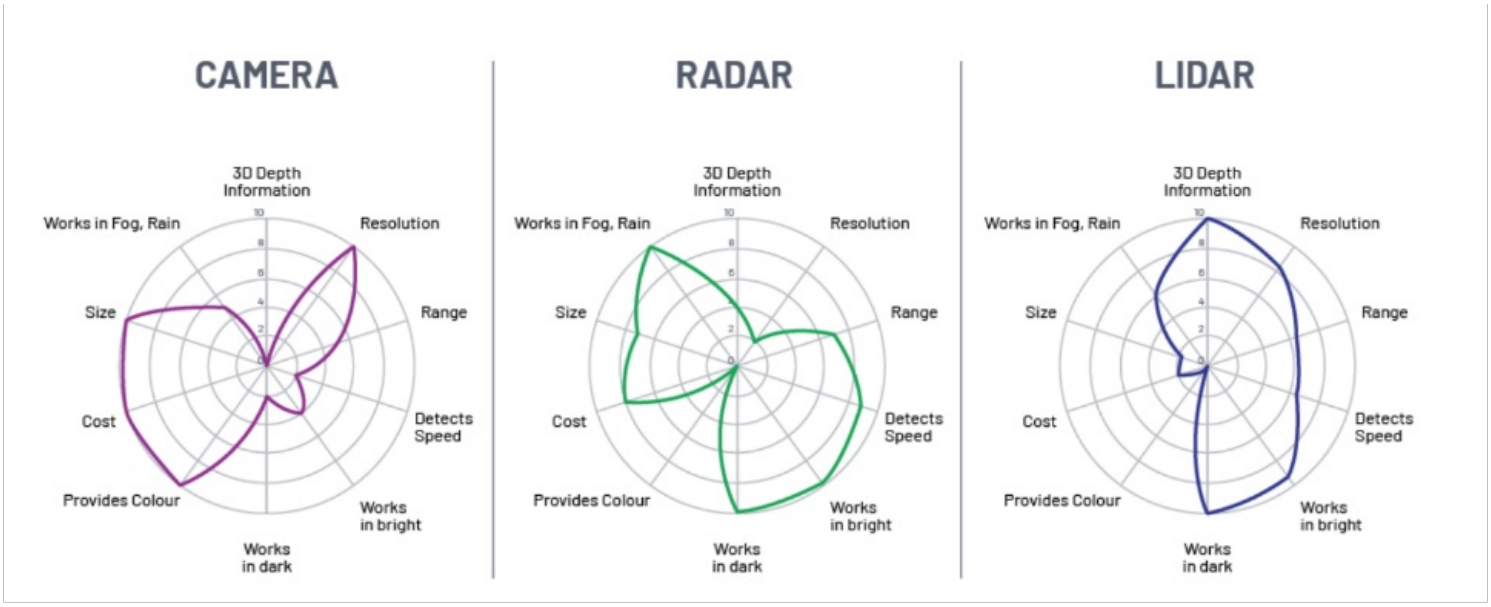


Image source: https://www.outsight.tech/technology/fused-sensing-vs.-sensor-fusion

Microphones

You have seen the devices that serve as the eyes for the self-driving system. Now, let's look at the ears of the system.

When a person hears an ambulance siren, they can detect the source of the sound, as well as the speed and direction at which the source is moving. Similarly, the self-driving vehicle uses microphones to gather audio information from the surroundings.

> Auditory information can also help the system learn about weather conditions. For instance, whether the road is wet or not (rainy weather) can be judged by the sound of the vehicle on the road. As a result, the system may decide to decrease the speed of the vehicle.

Now, you have seen the tools that provide input data to the self-driving vehicle so that it can perceive its environment.

Subtasks

The pipeline from environment perception to planning the movement of the vehicle can be split into several machine learning *subtasks* which we will describe shortly.

In the field of computer vision, the first three subtasks represent approaches that can be adapted to deal with imaging datasets (*driving images in your case*), e.g., classification, localization, segmentation, etc. Using an example, let's break this down to understand how to approach a problem statement of this scale.

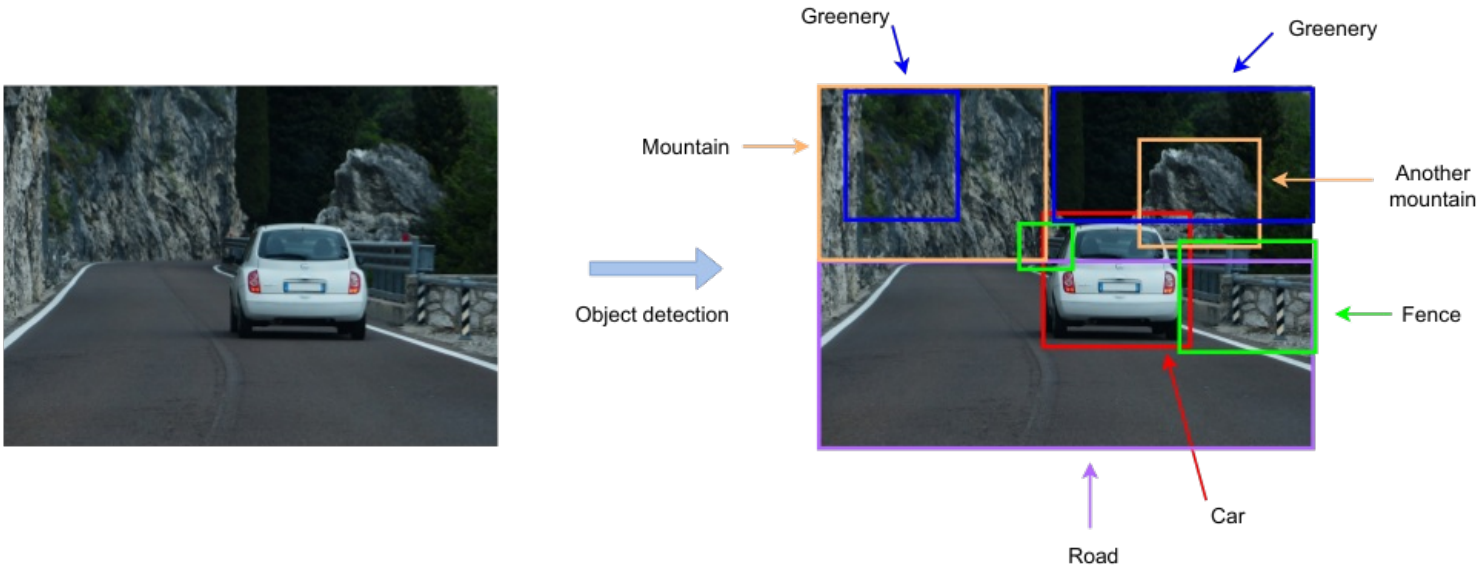
You start in the morning, and your job is to take the self-driving vehicle from Boston to New York City. Imagine a self-driving vehicle on a highway in daylight with a clear sky and less traffic. If you train an *image classier* on the labeled data, then feeding the input camera frames in real-time would probably generate the average classification scores like this: {"car":0.1, "road":0.5, "sky":0.3, "trees":0.05, "misc":0.05}.

This score provides a decent estimate of the overall categories of objects present in the surroundings. However, it does not help locate these objects. This can be solved through an *image localizer*, which will generate bounding boxes, i.e., locations on top of the predicted objects. This leads you to your first subtask, which is *object detection*.

---

1. Object detection

   In object detection, we detect instances of different class objects (e.g., greenery, fences, cars, roads, and mountains) in the surroundings and *localize* them by drawing bounding boxes.



Object detection: Different objects are identified and localized with the help of bounding boxes, as indicated in different colors

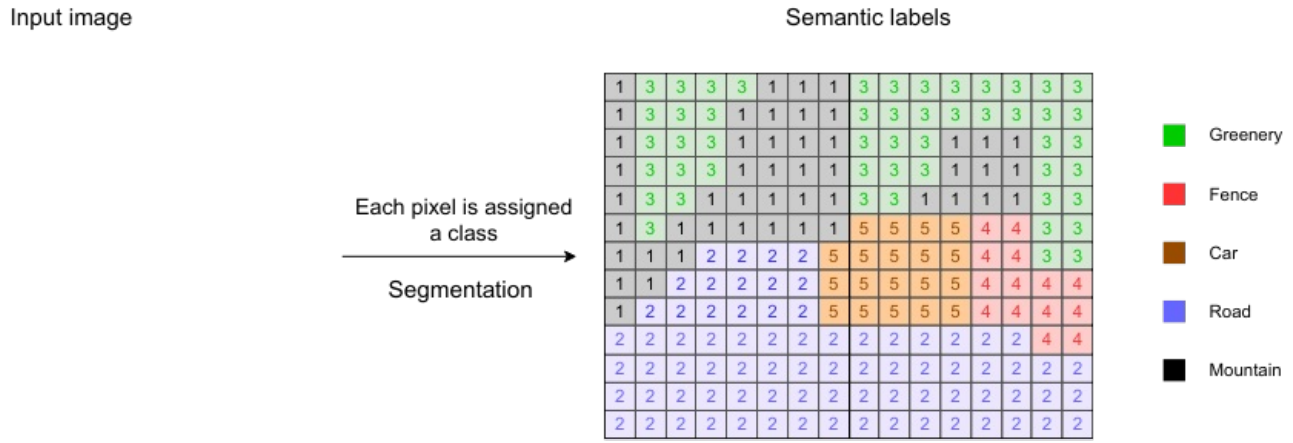
---

Continuing with the example, let's complicate the problem. You enter New York City around noon and encounter medium traffic. Your image localizer will generate a lot of overlapping bounding boxes. To resolve this, you need to upgrade the system to segment the object categories to find the optimal available path for the vehicle. You build an *image segmenter* that can generate the predicted binary mask/category on top of bounding boxes and classifications through semantic segmentation. This leads to the second subtask, i.e., semantic segmentation.

---

2. Semantic segmentation

   Semantic segmentation can be viewed as a pixel-wise classification of an image. Objects of the same class are *assigned the same label* as shown in the following diagram.



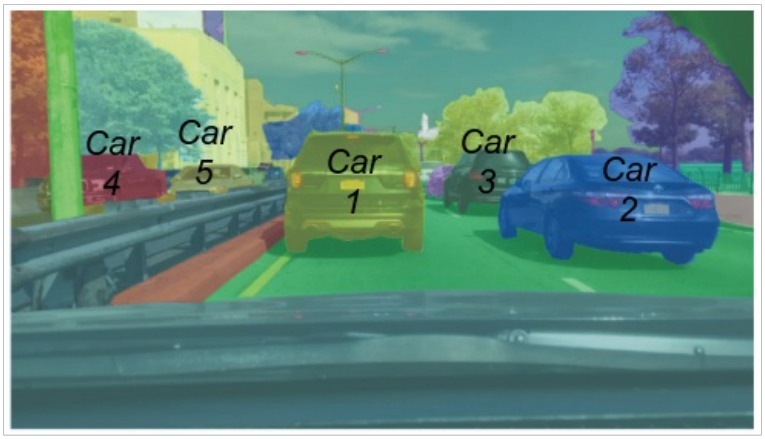
Semantic segmentation

Let's further complicate the problem. Now, you need to navigate a self-driving vehicle in a very busy street (e.g., Times Square) in Manhattan. In this case, you need to segment individual objects in real-time with high precision: You upgrade the system from semantic to instance-based segmentation, which will generate a predictive binary mask/object on top of the bounding boxes and classifications. This leads us to the third subtask, i.e., *instance segmentation*.

---

3. Instance segmentation

   Semantic segmentation does not differentiate between different instances of the same class. Instance segmentation, however, combines object detection and segmentation to classify the pixels of each instance of an object. It first detects an object (a particular instance) and then classifies its pixels. Its output is as follows:



Instance segmentation: Makes a distinction between the boundary of different instances of the same class

4. Scene understanding

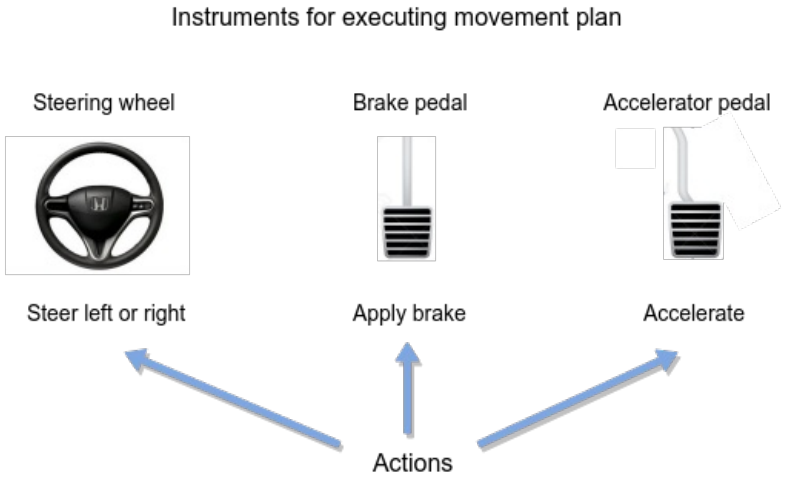
   Here, you try to understand what is happening in the surroundings. For instance, you may find out that a person is walking towards us, and you need to apply brakes. Or, as in the following diagram, we see that a car is going ahead of us on the highway.

Car ahead on the highway

Scene understanding: given an input image, the output is a text-based command: "Car ahead on the highway"

5. Movement plan

   After you have identified the objects, segmented unique objects in the image and understood the scenario, it's time to decide the movement plan for the self-driving vehicle. For example, you might decide to slow down (apply brakes) due to a car ahead.



Instruments for executing movement plan for the self-driving car

In this chapter, the focus will be the first two subtasks, i.e., the following machine learning problem:

"Perform semantic segmentation of the self-driving vehicle's surrounding environment."

← Back

Ranking

Next →

Metrics

Mark as Completed

Metrics

Let's explore some metrics that will help evaluate the performance of the vision-based self-driving car system.

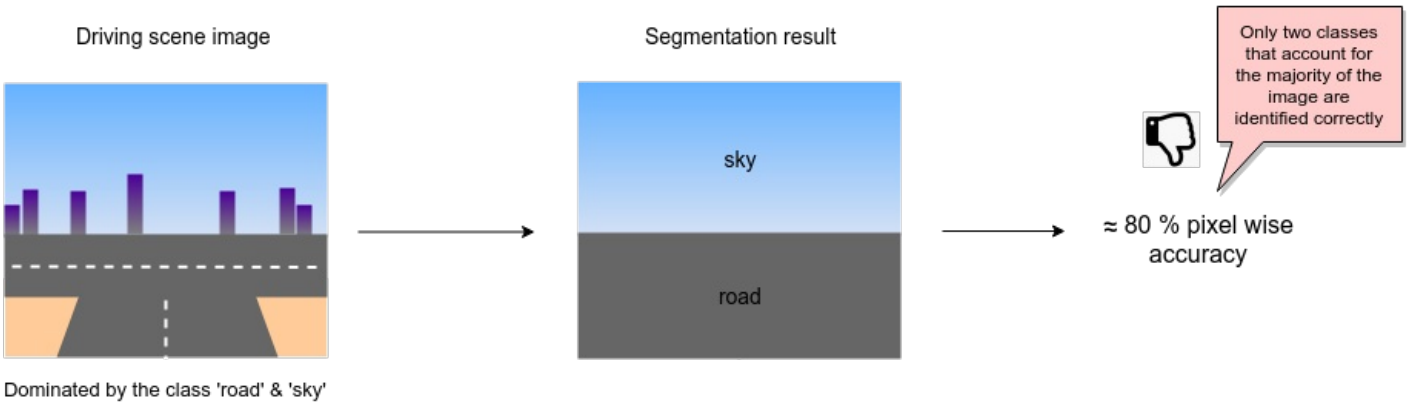
We'll cover the following

Component level metric

The component of the self-driving car system under discussion here is the semantic segmentation of objects in the input image. In order to look for a suitable metric to measure the performance of an image segmenter, the first notion that comes to mind is the *pixel-wise accuracy*. Using this metric, you can simply compare the ground truth segmentation with the model's predictive segmentation at a pixel level. However, this might not be the best idea, e.g., consider a scenario where the driving scene image has a major class imbalance, i.e., it mostly consists of sky and road.

> For this example, assume one-hundred pixels (ground truth) in the driving scene input image and the annotated distribution of these pixels by a human expert is as follows: sky=45, road=35, building=10, roadside=10.

If your model correctly classifies all the pixels of only sky and road (i.e., sky=50, road=50), it will result in high pixel-wise accuracy. However, this is not really indicative of good performance since the segmenter completely misses other classes such as building and roadside!



High pixel-wise accuracy doesn't always translate to a great performance by the segmentation model

In general, you need a higher pixel-wise accuracy for objects belonging to each class as an output from the segmenter. The following metric caters to this requirement nicely.

IoU

Intersection over Union (IoU) divides the overlapping area between the predicted segmentation and the ground truth in perspective, by the area of union between the predicted segmentation and the ground truth.

$IoU = \frac{area\ of\ overlap}{area\ of\ union}$ or $IoU = \frac{|P_{pred} \cap P_{gt}|}{|P_{pred} \cup P_{gt}|}$

This metric ranges from 0–1 (or 0%–100%). '0' indicates no overlap while '1' indicates perfectly overlapping segmentation.

The driving images contain objects of multiple classes, as shown above (e.g., building, roadside, sky, road, etc.). So, you will be performing multi-class segmentation, for which the **mean IoU** is calculated by taking the *average of the IoU for each class*.

Now, let's calculate the mean IoU and see how it differs from pixel accuracy. You will be considering the same driving scene image and its predicted segmentation, as shown above.

You will begin by calculating the IoU for each class. Here, the "area of overlap" means the number of pixels that belong to the particular class in both the prediction and ground-truth. Whereas, the "area of union" refers to the number of pixels that belong to the particular class in the prediction and in ground-truth, but not in both (the overlap is subtracted).

> Let's apply the calculations: ground truth: [sky=45, road=35, building=10, roadside=10], segmenter predictions: [sky=50, road=50, building=0, roadside=0].

$IoU_C = \frac{P_{pred} \cap P_{gt}}{(P_{pred}+P_{gt})-(P_{pred} \cap P_{gt})}$

$where\ P_{pred} = number\ of\ pixels\ classified\ as\ class\ C\ in\ prediction\ and\ P_{gt} = number\ of\ pixels\ classified\ as\ class C\ in\ ground\ truth$

$IoU_{sky} = \frac{45}{(50+45)-45} = 90\%$

$IoU_{road} = \frac{35}{(50+35)-35} = 70\%$

$IoU_{building} = \frac{0}{(0+10)-0} = 0\%$

$IoU_{road\ side} = \frac{0}{(0+10)-0} = 0\%$

$Mean\ IoU = \frac{IoU_{road}+IoU_{sky}+IoU_{road\ side}+IoU_{building}}{4} = \frac{90+70+0+0}{4} = 40\%$

You can see that Mean IoU (40%) is significantly lower than the pixel-wise accuracy (80%) and provides an accurate picture of how well your segmentation is performing.

> You will be using IoU as an offline metric to test the performance of the segmentation model.

End-to-end metric

You also require an online, end-to-end metric to test the overall performance of the self-driving car system as you plug in your new image segmenter to see its effect.

Manual intervention

Ideally, you want the system to be as close to self-driving as possible, where the person never has to intervene and take control of the driving. So, you can use manual intervention as a metric to judge the success of the overall system. If a person rarely has to intervene, it means that your system is performing well.

> This is a good metric for the early testing phase of the self-driving car system. During this time you have a person ready to take over in case the self-driving system makes a poor decision.

Simulation errors

Another approach is to use historical data, such as driving scene recording, where an expert driver was driving the car. You will give the historical data as input to your self-driving car system with the new segmentation model and see how its decisions align with the decisions made by an expert driver.

You will assume that the decisions made by the professional driver in that actual scenario are your ground truths. The overall objective will be to minimize the movement and planning errors with these ground truths.

As you replay the data with new segmenter, you will experiment to measure whether your segmentation is resulting in a reduction of your simulation-based errors or not. This will be a good end to end metric to track.

← Back
Problem Statement

Next →
Architectural Components

Mark as Completed

Report an Issue
Ask a Question
(https://discuss.educative.io/tag/metrics__self-driving-car-image-segmentation__grokking-the-machine-learning-interview)

Architectural Components

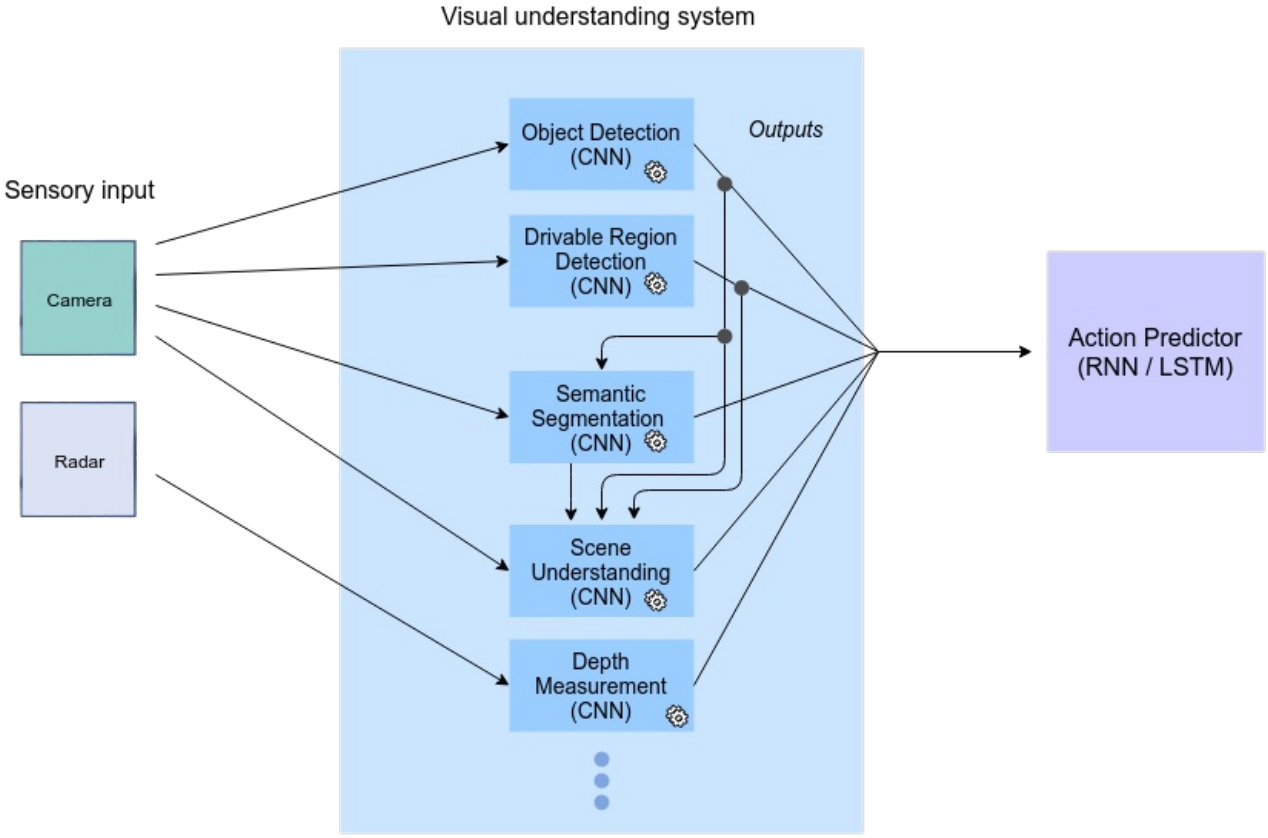
Let's take a look at the architectural components of the self-driving car system.

We'll cover the following

- Overall architecture for self-driving vehicle
- System architecture for semantic image segmentation

Overall architecture for self-driving vehicle

Let's discuss a simplified, high-level architecture for building a self-driving car. Our discussion entails the important learning problems to be solved and how different learning models can fit together, as shown below.



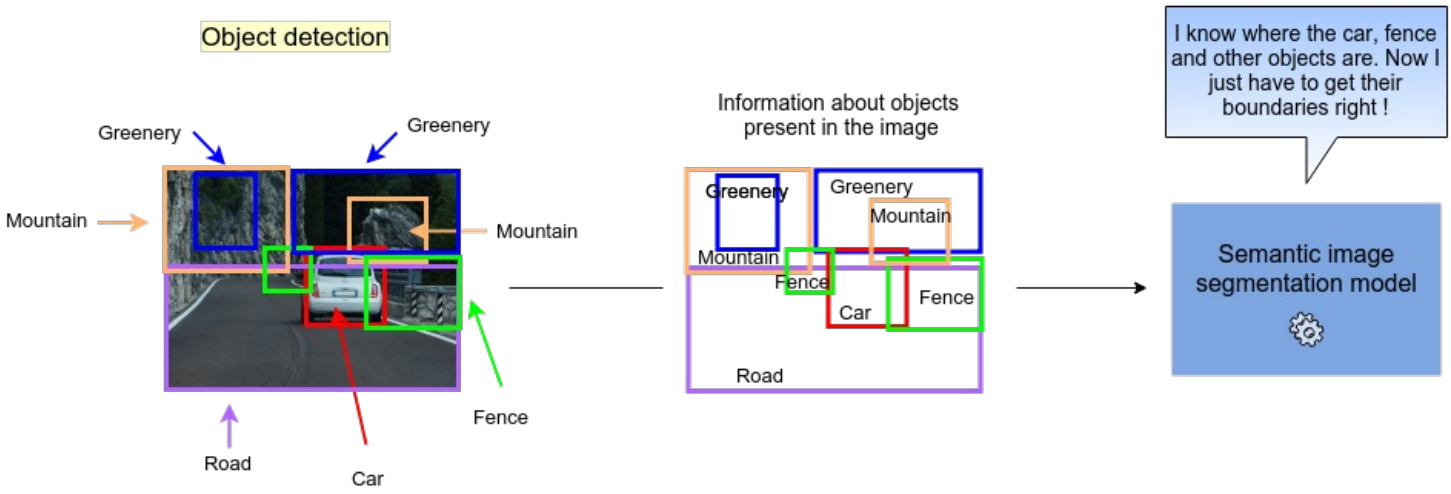
High-level architecture (CNN: Convolutional neural network, RNN: Recurrent neural network, LSTM: Long short-term memory)

The system is designed to receive sensory inputs via cameras and radars, which are fed to the *visual understanding system* consisting of different convolutional neural networks (CNN), each for a specific subtask. The output of the *visual understanding system* is used by the *action predictor RNN or LSTM*. Based on the visual understanding of the environment, this component will plan the next move of the vehicle. The next move will be a combination of outcomes, i.e., applying brakes, accelerating, and/or steering the vehicle.

> We won't be discussing input through Lidar here. However, it can also be used for scene analysis similar to a camera, especially for reconstructing a 3D view of an environment.

Now, let's have a look inside the visual understanding system. The object detection CNN detects and localizes all the obstacles and entities (e.g., humans and other vehicles) in the vehicle's environment. This is essential information because the action predictor RNN may predict to slow down the vehicle due to the impending obstacle (e.g., a person crossing the road). However, the most crucial information for the action predictor RNN is information that allows it to extract a drivable path for the vehicle. Therefore, due to the significance of this task, we will train a separate model for this purpose: the drivable region detection CNN. It will be trained to detect the road lanes to help the system decide whether the pathway for the vehicle is clear or not.

Moreover, as the object detection CNN identifies the key objects in the image (predicts bounding boxes), it is further required to share its output along with the raw pixel data for *semantic image segmentation* (i.e., draw pixel-wise boundaries around the objects). These boundaries will help in navigating the autonomous vehicle in a complex environment where overlapping objects/obstacles are presented.



Object detection model's output is fed into the segmentation model

Machine learning models used in the components

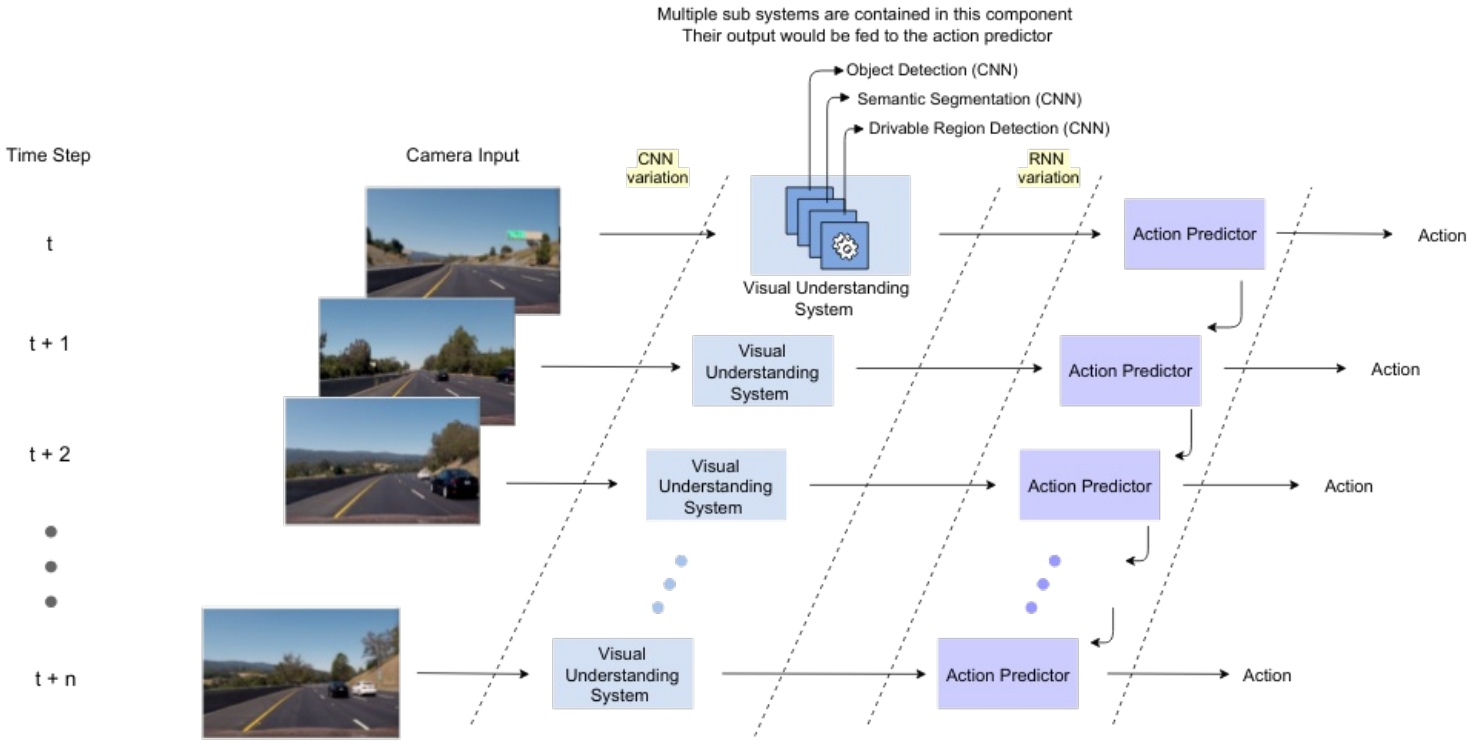
Many of the subtasks in the visual understanding component are carried out via specialized CNN models that are suited for that particular subtask.

The action predictor component, on the other hand, needs to make a movement decision based on:

1. Outputs of all the visual understanding sub-tasks
2. Track/record of the vehicle's movements based on previous scene understanding

This can be best learned through a recurrent neural network (RNN) or long short-term memory (LSTM) that can utilize the temporal features of the data, i.e., previous and current predictions from the scene segmentation as inputs.

Let's dive a little deeper into how this is happens, focusing on the input from the camera.



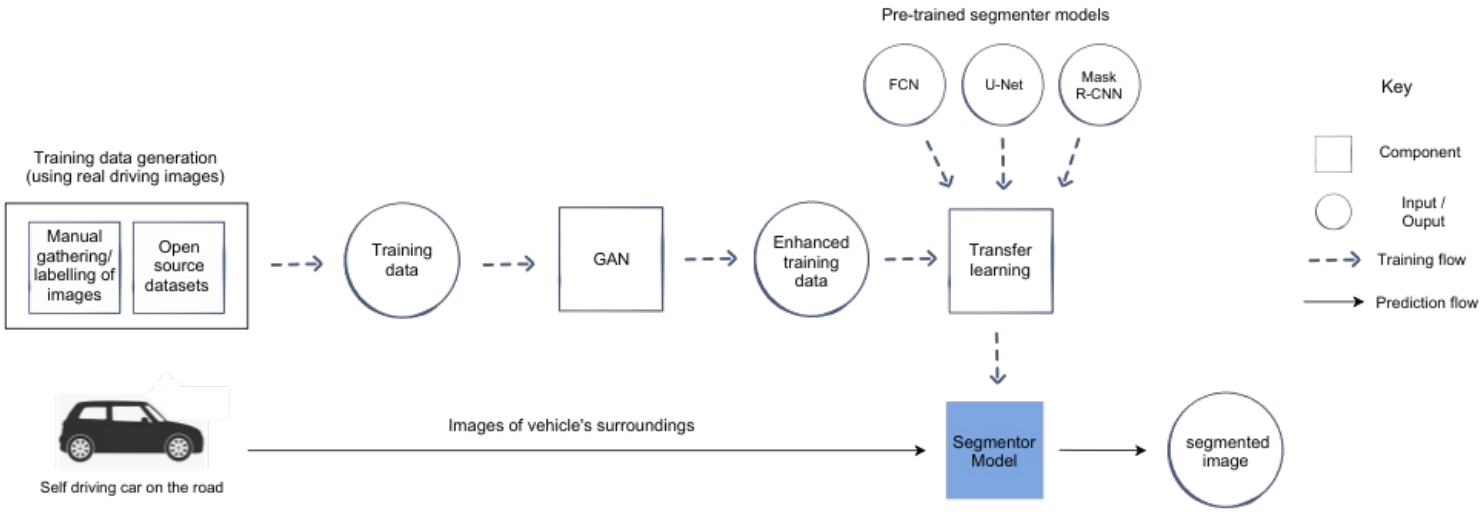
How the self-driving vehicle architecture works in a block diagram

You will receive the video frames covering the vehicle's surroundings from the camera as input. This video is nothing more than a sequence of images (frames per second). The image at each time step (t, t+1,...) will be fed to the visual understanding system. The multiple outputs of this component will form the input to the self-driving vehicle's action predictor. The action predictor will also use the previous time step information (t-1) while predicting the action for the current time step (t).

> RNN will utilize the past information to make "informed" predictions in real-time.

System architecture for semantic image segmentation

You just saw how the semantic image segmentation task fits in the overall architecture for the self-driving vehicle system architecture. Now, let's zoom in on the architecture for the semantic image segmentation task.



Architectural diagram for semantic image segmentation

The above diagram shows the training flow and prediction flow for the semantic image segmentation system architecture.

The training flow begins with training data generation, which makes use of two techniques. In the first technique, real-time driving images are captured with the help of a camera and are manually given pixel-wise labels by human annotators. The latter technique simply uses open-source datasets of self-driving vehicle images. This training data is then enhanced or augmented with the help of generative adversarial networks (GANs). Collectively all the training data is then used to train the segmenter model. Transfer learning is applied with the segmenter model to utilise powerful feature detectors from pre-trained models (FCN, U-Net, Mask R-CNN). The pre-trained models are optimized on your datasets to get the final model.

In the prediction flow, your self-driving vehicle would be on the road. You would receive real-time images of its surroundings, which would be given to the segmenter model for semantic image segmentation.

← Back

Metrics

Next →

Training Data Generation

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/architectural-components__self-driving-car-image-segmentation__grokking-the-machine-learning-interview)

Training Data Generation

Let's generate training data for the semantic image segmentation model.

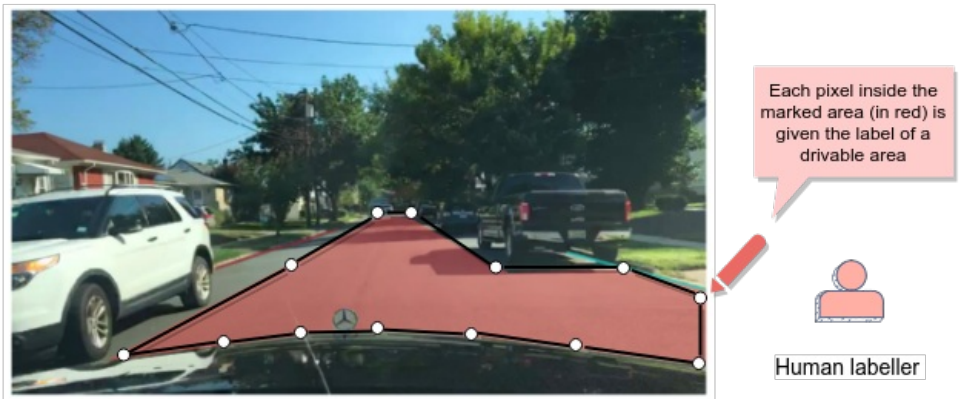
We'll cover the following

Generating training examples

Autonomous driving systems have to be extremely robust with no margin of error. This requires training each component model with all the possible scenarios that can happen in real life. Let's see how to generate such training data for performing semantic segmentation.

Human-labeled data

First, you will have to gather driving images and hire people to label the images in a pixel-wise manner. There are many tools available such as Label Box (https://labelbox.com/product/image-segmentation) that can facilitate the pixel-wise annotation of images.



Manual labelling of driving images to generate training data for semantic segmentation

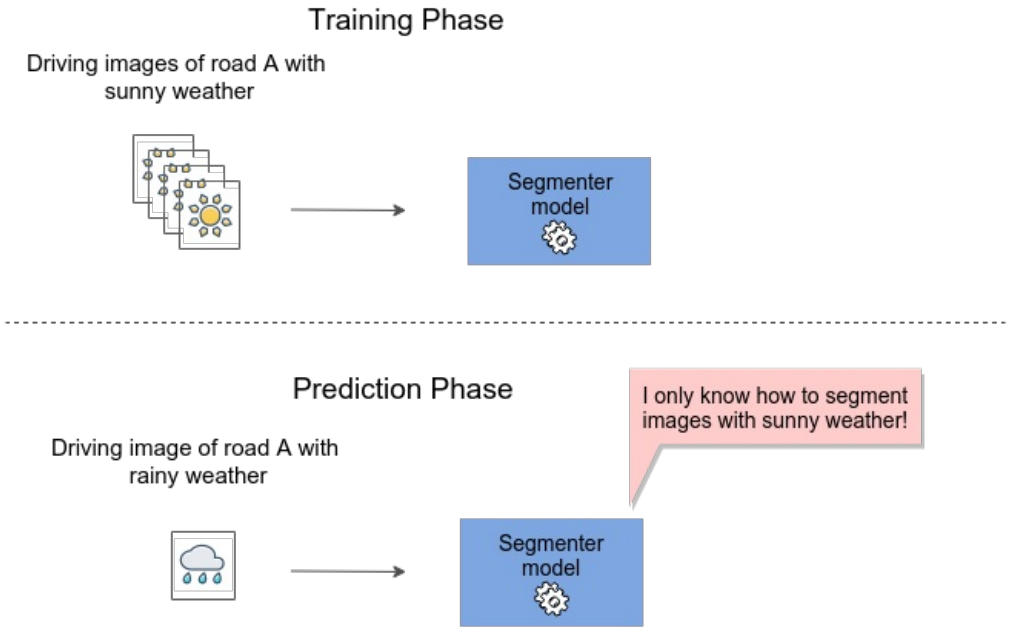
All the other parts of the image are also labelled similarly.

Open source datasets

There are quite a few open-source datasets available with segmented driving videos/images. One such example is the "BDD100K: A Large-scale Diverse Driving Video Database" (https://bair.berkeley.edu/blog/2018/05/30/bdd/). It contains segmented data for full frames, as shown with an example below.

BDD100K: full frame segmentation

These two methods are effective in generating manually labeled data. In most cases, your training data distribution will match with what you observe in the *real-world scene* images. However, you may not have enough data for al the **conditions** that you would like your model to make predictions for such as snow, rain, and night. For a self-autonomous vehicle to be perfect, your segmenter should work well for all the possible weather conditions, as well as cover a variety of obstacle images in the road.



A segmenter trained on a certain type of images won't be able to handle unseen conditions reliably

The sunny vs rainy condition is just one example; there can be many such situations.

It is an important area to think about how you can give your model *training data* for all conditions. One option is to manually label a lot of examples for each scenario. The second option is to use powerful data augmentation techniques to generate new training examples given your human-labeled data, especially for conditions that are missing in your training set. Let's discuss the second approach, using generative adversarial networks (GANs).

Training data enhancement through GANs

In the big picture, your self-driving system should compete with human intelligence when it comes to making decisions and planning movements. The segmenter can play its role in creating a reliable system by being able to accurately segment the driving scene in any condition that the vehicle experiences.

Achieving this target requires a diverse set of training data that covers all the possible permutations of the driving scene. For instance, assume that your dataset only contains ten-thousand driving images in the snowy Montreal conditions and fifty-thousand images in the sunny California conditions. You need to devise a way to *enhance your training data* by converting snowy conditions of the ten-thousand Montreal images to sunny conditions and vice versa.

The target includes two parts:

1. Generating new training images
2. Ensuring generated images have different conditions (e.g., weather and lighting conditions).

For the first part, you can utilize GANs. They are deep learning models used for *generation of new content*. However, for the second part, i.e., to generate a different variation of the original segmented image, a simple GAN would not work. You would have to use **conditional generative adversarial networks** (cGANs) instead. Let's see why.

Minimize

GANs

Generative adversarial networks belong to the set of *generative models* in contrast to discriminative models, i.e., they focus on learning the unknown probability distribution of the input data to **generate** similar inputs rather than just classifying inputs as belonging to a certain class.

> For instance, given panda images, a generative model will try to understand how pandas look like (learn the probability distribution) rather than trying to accurately predict if an image is of a panda or another animal. This allows you to *generate new panda images*.

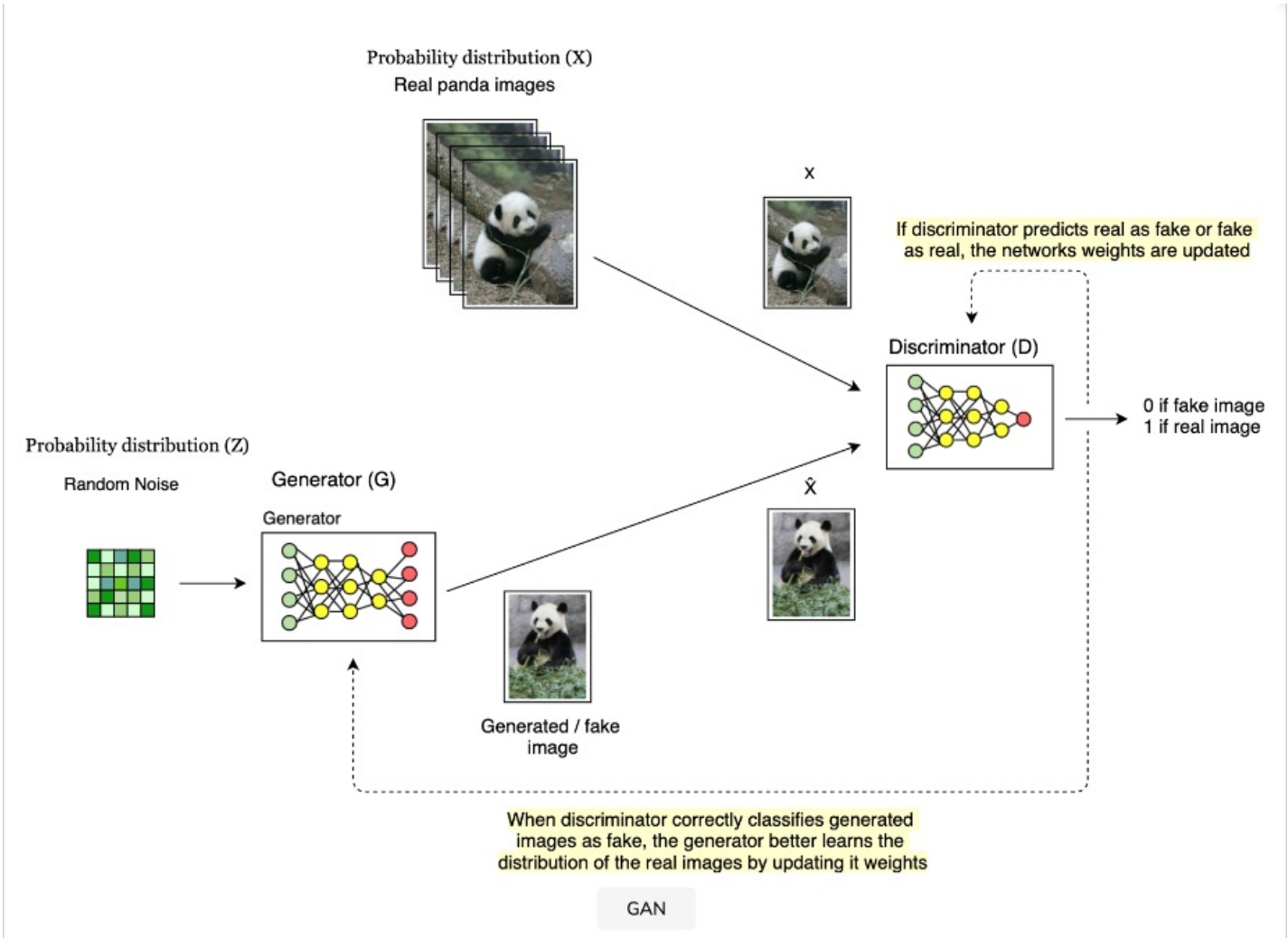
GANs are made of two deep learning models that compete against each other (hence, the term adversarial):

1. Generator (Forger)

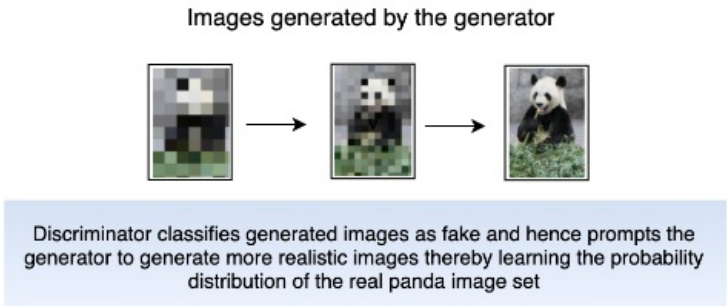
   This neural network receives a sample from a random distribution Z (noise). It transforms it into something (a panda) that looks like it came from the target/true distribution(panda images). This is done by learning a mapping between the input samples and the images in the true distribution, during the training phase.

2. Discriminator (Detective)

   This neural network receives images from the true distribution X as well as the new/fake images $\hat{X}$ generated by the generator. The goal is to discriminate between real and generated images. The network is trained to output 0 for a generated image and 1 for a real image.



The generator's tries to fool the discriminator into thinking that its generated images belong to the true distribution. On the other hand, the discriminator tries to catch all the fake/generated images. This encourages the generator to learn the true probability distribution more closely and generate such real looking images that the discriminator's output converges to 0.5, i.e., it is not able to distinguish fake from real.



The generator's output improves throughout the training iterations.

Supervised or unsupervised learning?

You saw that the generator model of the GAN learns the target probability distribution (looks for patterns in the input data) to generate new content; this is known as *generative modeling*.

Generative modeling is an unsupervised task where the model is not told what kind of patterns to look for in the data and there is no error metric to improve the model. However, the *training process of the GAN is posed as a supervised learning problem* with the help of a discriminator. The discriminator is a supervised

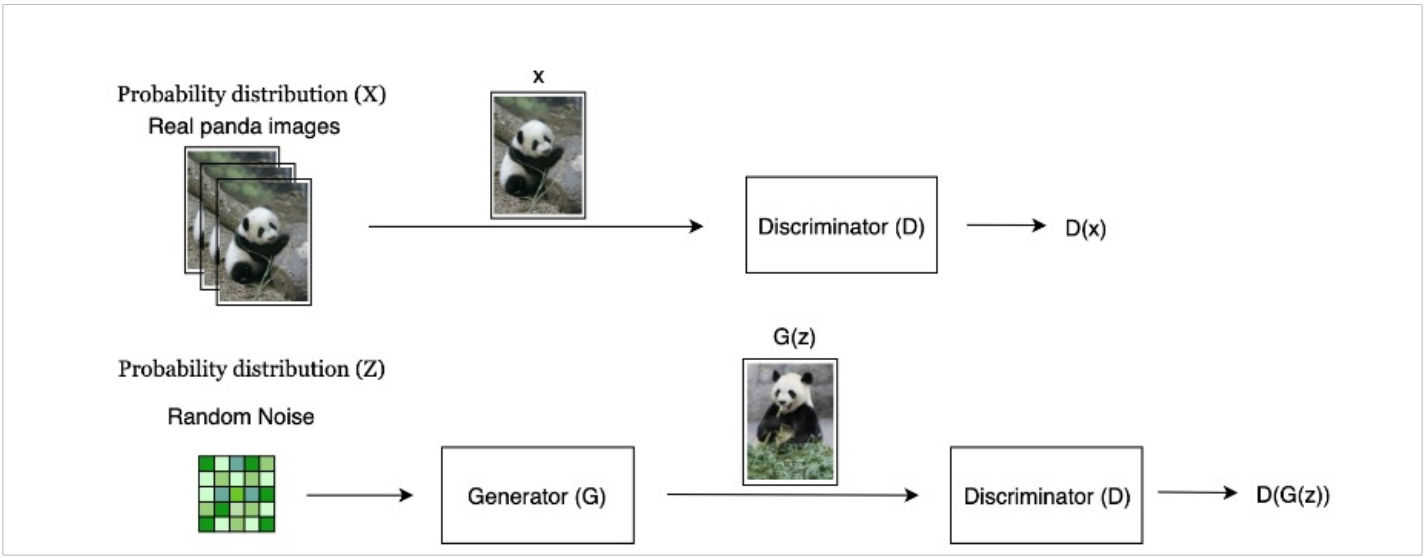
classifier that distinguishes between real and generated images. This allows us to devise a **supervised loss function for the GAN**.

Loss Function

The *binary cross entropy loss function* for the GAN is as follows:

$$min_G\ max_D\ \{E_{x\sim p_{data}(x)}[log\ D(x)] + E_{z\sim p_z(z)}[log(1 - D(G(z)))]\}$$

Its made up of two terms. Let's see what each part means (This is the combined loss for both discriminator and generator network).



- x is an image from the true distribution
- z is the random input to the generator
- $p_{data}(x)$ is the probability distribution of the real images.
- $p_z(z)$ is the probability distribution of the generated images.
- G(z) is the generator's output when given noise z.
- D(x) is the discriminator's probability (judgement) that the *real data instance x* is real.
- D(G(z)) is the discriminator's probability (judgement) that the *generated instance x* is real.
- E_x is the expected value over all real data instances.
- E_z is the expected value over random inputs to the generator.

> The expected value of a variable X is the average of all values of X weighted by their occurrence probability.

The GAN's objective is to replicate the p_{data} with p_z. Binary cross entropy helps to measure the distance between these two distributions. Let's see how this works.

The term *D(x)* is the probability that a data point belongs to *class real*, whereas the term *1-D(G(z))* is the probability of belonging to *class fake*. The loss function works by measuring how far away from the actual value (real 1 or fake 0) the prediction is for each class and then averages the errors (Expectation) to obtain the final loss.

Discriminator's aim

The discriminator wants to distinguish between real and fake. Earlier, you mentioned the final form of the loss. Let's see how the discriminator's loss function is derived from the original form of the binary cross-entropy loss as follows:

$$Loss(\hat{y}, y) = [ylog\hat{y} + (1-y)log(1-\hat{y})]$$

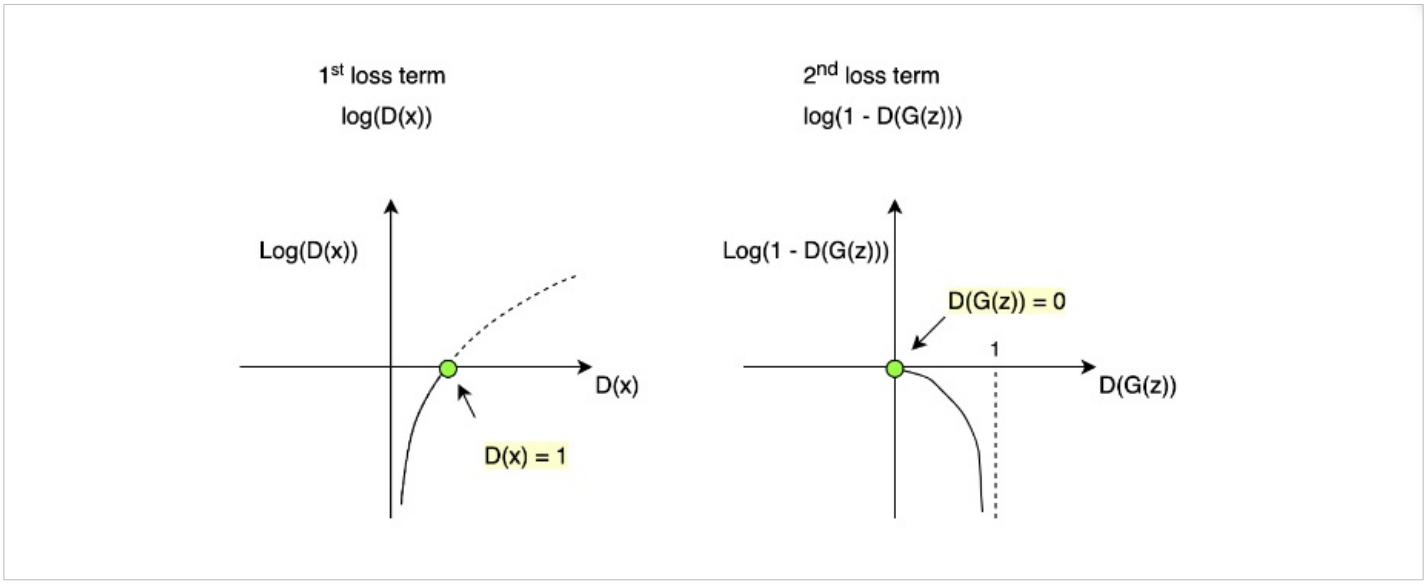
When you have the image coming from the true distribution, the function takes the form:

$$\begin{aligned} Loss(D(x), 1) &= [1 * log(D(x)) + (1-1) * log(1-D(x)] \\ &= log(D(x)) - (1) \end{aligned}$$

When you have the image coming from the generator, the function takes the form:

$$\begin{aligned} Loss(D(G(z)), 0) &= [0 * log(D(G(z)) + (1-0) * log(1-D(G(z))] \\ &= log(1-D(G(z)) - (2) \end{aligned}$$

The terms (1) and (2) are added to achieve the final loss. To see why the discriminator maximises this loss (max_D part), you need to plot both terms.



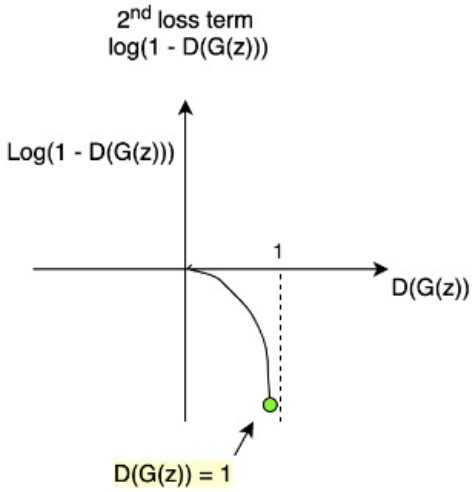
You can observe that the discriminator needs to maximize the 1^{st} term in order to correctly predict "1" for the real input image. Also, it needs to maximize the second term to correctly predict "0" for the generated input image.

Generator's Aim

The generator aims to generate such an image that the discriminator classifies it as real or 1. The generator's loss function is the same as when the discriminator receives generated image, i.e.,

$$Loss(D(G(z)), 0) = log(1-D(G(z))$$

However, unlike the discriminator, the generator minimizes this term . Let's look at the plot again.



As you can see in the log plot of the second term, in order to fool the discriminator (D(G(z))=1), you need to minimize the term.

Training the GAN

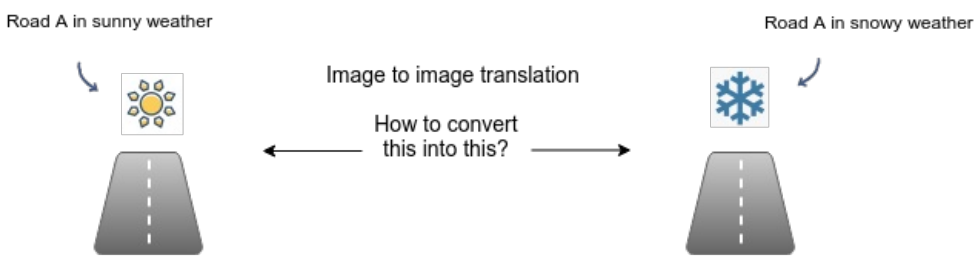
The generator and discriminator are trained alternatively.

For training the discriminator, the weights of the generator model are frozen. Two mini-batches are taken. One is from the real images data. The other consists of generated images obtained by feeding random noise samples to the generator. There are no labels associated with this data. However, the discriminator still learns due to the formulation of the loss function, which that we discussed earlier. The discriminator updates its weights to *maximize* the terms leading to fake images being classified as 0 and real as 1.

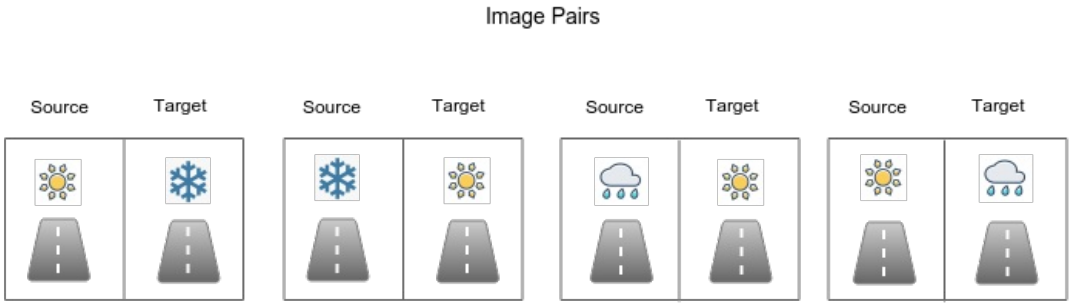
For training the generator, the weights of the discriminator model are frozen. A mini-batch is taken from the noise. Once again there are no labels, but the model updates its weights to *minimize* the loss term, leading to fake images being classified as 1.

If you train a GAN on a set of driving images, you do not have any control over the generator's output, i.e., it would just generate driving images similar to the images present in the training dataset. However, that is not particularly useful in your case.

Therefore, we want to perform **image-to-image translation** (a kind of data augmentation) to enhance our training data, i.e., you want to map features from an input image to an output image. For instance, you want to learn to map driving images with one weather condition to another weather condition, or you may want to convert day time driving images to night time images and vice versa. This can be done with a cGAN which is a deep neural network that extends the concept of GAN.



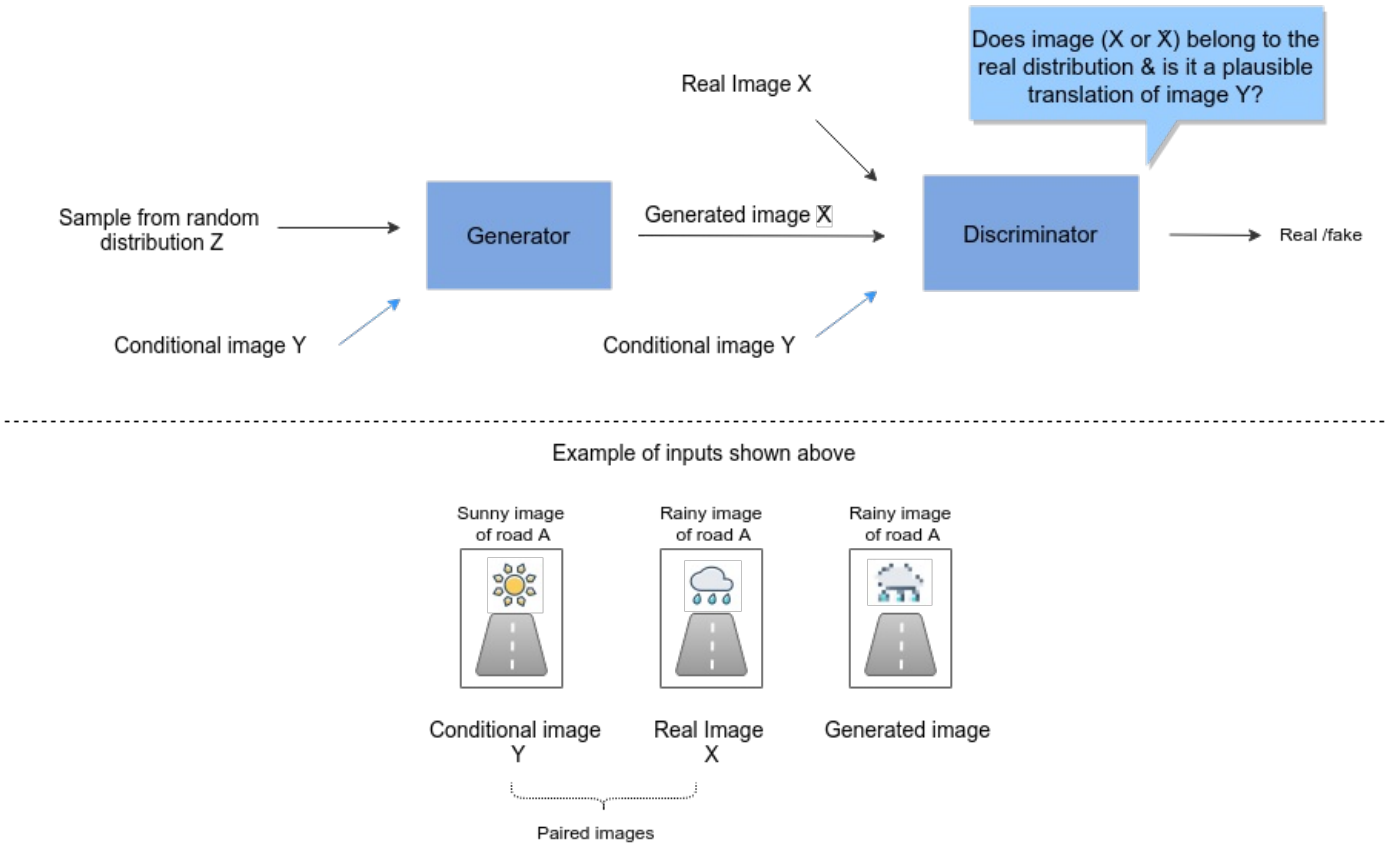
For image translation, your training data should consist of paired images. For example, image pairs could be of the same road during day and night. They could also be of the same road during winter and summer.



Training data for cGAN

In cGAN, you give the source image that you want to translate to the generator, along with an additional input (target image as a condition). This will guide the generator's output image to be of the same place but with a different weather/light condition.

You will also feed the source image to the discriminator so that apart from its usual task of discriminating between real and generated images, it will also see if the given image is a plausible translation of the source image.



cGAN: both generator and discriminator are conditioned on y (image to be translated)

The loss function for cGAN is as follows:

$$min_G\ max_D\ \{E_{x \sim p_{data}(x)}[log\ D(x|y)] + E_{z \sim p_z(z)}[log(1 - D(G(z|y)))]\}$$

To understand this loss function you can refer to the expandable section on GANs above. You see that the cGAN loss function differs from the GAN loss function as the discriminator's probability D(x) or D(G(z)) is now conditioned on y.

Once you have trained the cGAN for the image translation task, you can now feed it the driving images you manually collected through the generator model to enhance the training data.

Using a similar approach, you can build multiple cGAN models for different conditions, including generating night images, cloudy images, snow images etc., and enhance our training data set with examples of all conditions.

> This chapter does not have a feature engineering lesson, because the convolutional layers (in the CNN) extract features themselves.

Targeted data gathering

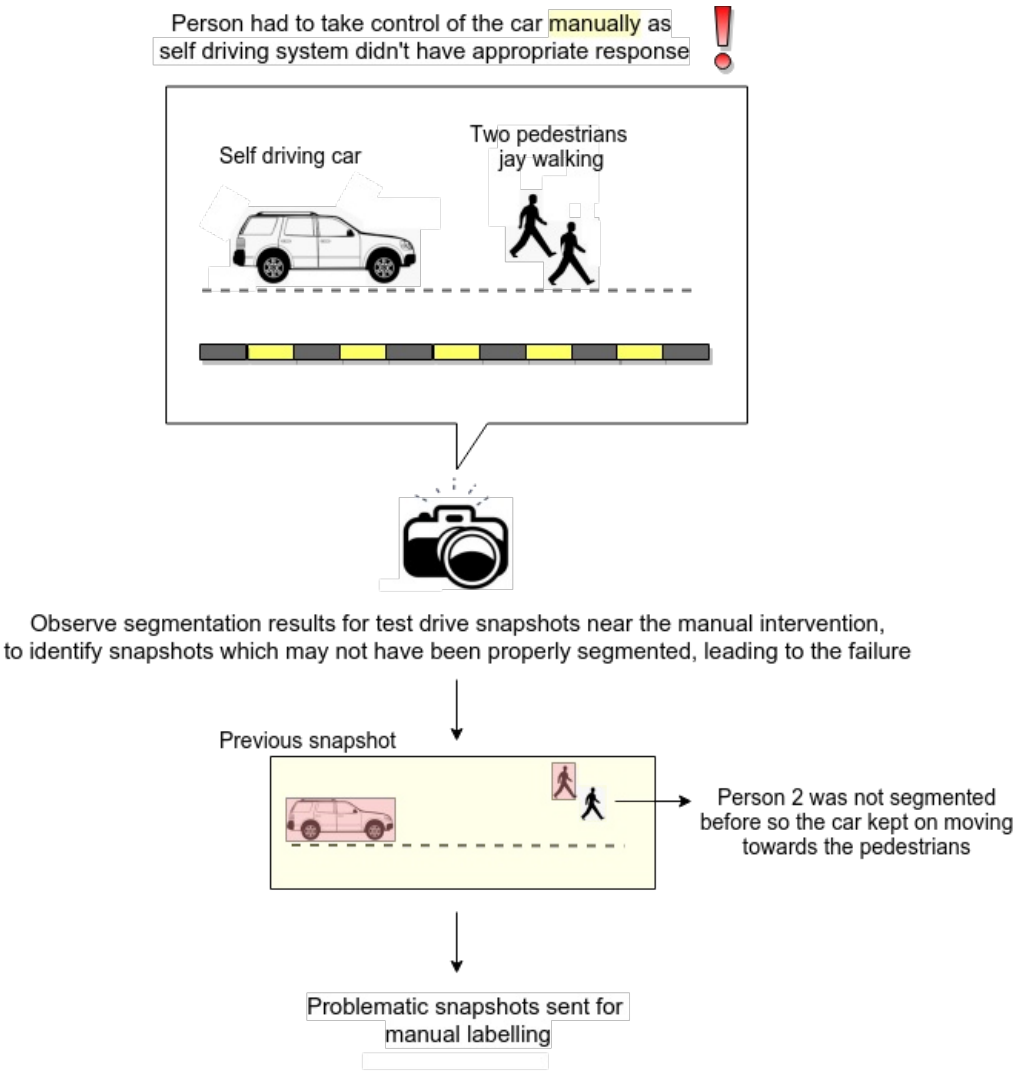
Earlier in the chapter, we discussed manual intervention as a metric for our end-to-end self-driving car system. You can use this metric to identify scenarios where your segmentation model is not performing well and learn from them.

> Here, you aim to gather training data specifically on the areas where you fail while testing the system. It will help to further enhance the training data.

You will test the self-driving car, with a person at the wheel. The vehicle has the capability to record the video of this test drive. After the test drive is over, you will observe the system and look for instances where it did not perform well and the person had to intervene.

Consider a scenario where you observe that the person had to manually intervene when two pedestrians were jaywalking. You will check the performance of the segmentation model on, let's say, the last twenty frames before the manual intervention took place. You might see that segmentation failed to predict one of the two pedestrians. You will select snapshots of those instances from the recording and ask the manual labelers to labels those snapshots since it means that the training data lacks such examples.

You will then focus on collecting such examples and ask the manual labelers to label them.



Why you need targeted data gathering

After this, you can also apply data augmentation on the newly gathered training examples to further improve your model's performance.

Back

Architectural Components

Next

Modeling

Mark as Completed

Report an Issue

Ask a Question (https://discuss.educative.io/tag/training-data-generation__self-driving-car-image-segmentation__grokking-the-machine-learning-interview)

Modeling

In modeling discussion, we will discuss two key aspects.
1. SOTA Segmentation Models: What are the state-of-the-art segmentation models and their architectures?
2. Transfer Learning: How can you use these models to train a better segmenter for the self-driving car data?

We'll cover the following

SOTA segmentation models

Machine learning in general and deep learning, in particular, have progressed a lot in the domain of computer vision-based applications during the last decade. The models enlisted in this section are the most commonly used deep neural networks that provide state-of-the-art (SOTA) results for *object detection and segmentation tasks*. These tasks form the basis for the self-driving car use case.

FCN

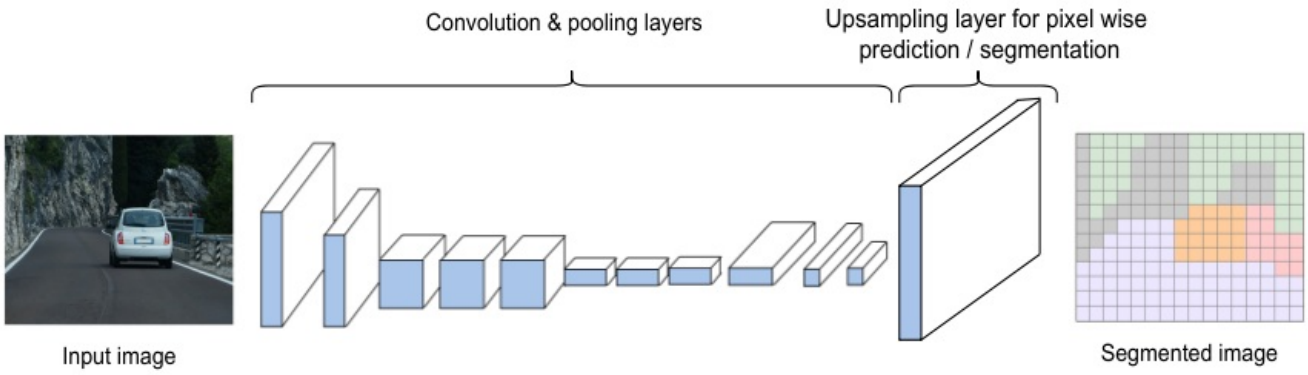
Fully convolutional networks (FCNs) are one of the top-performing networks for semantic segmentation tasks.

> Segmentation is a dense prediction task of pixel-wise classification.

A typical FCN operates by fine-tuning an image classification CNN and applying pixel-wise training. It first compresses the information using multiple layers of convolutions and pooling. Then, it up-samples these feature maps to predict each pixel's class from this compressed information.

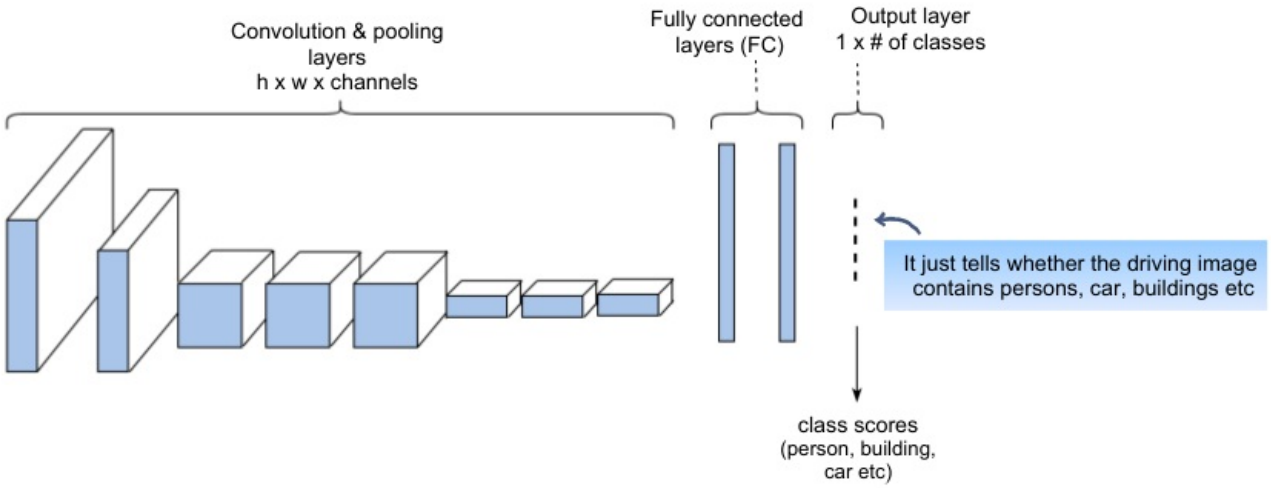
FCN architecture

Let's see how this FCN is made from a CNN trained for image classification



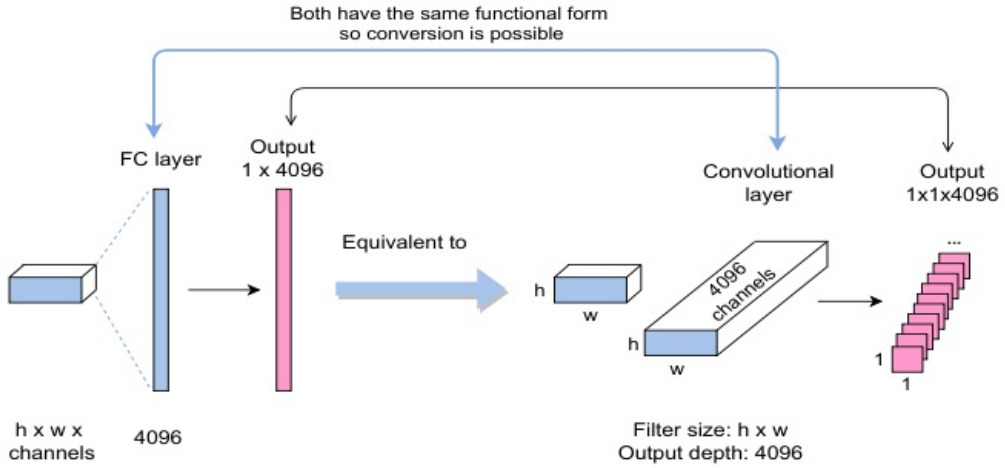
FCN Architecture

CNN trained for image classification



A typical architecture of a CNN-based object classifier

Converting Fully connected (FC) layers of CNN to Convolutional layers

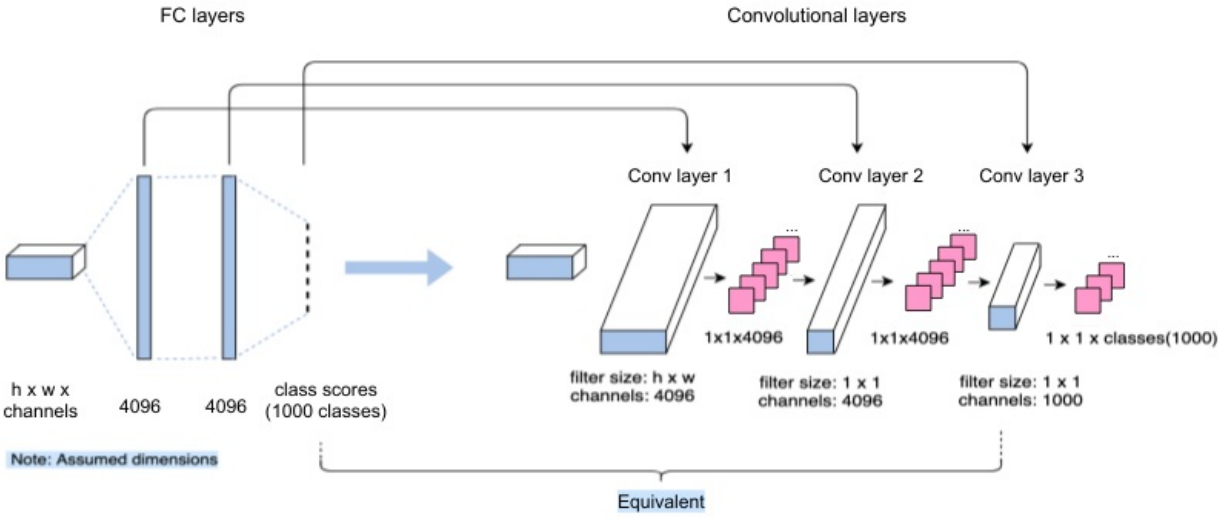


Note: Assumed dimensions

FC layers in CNN can be viewed as convolutions with kernels that cover their entire input regions

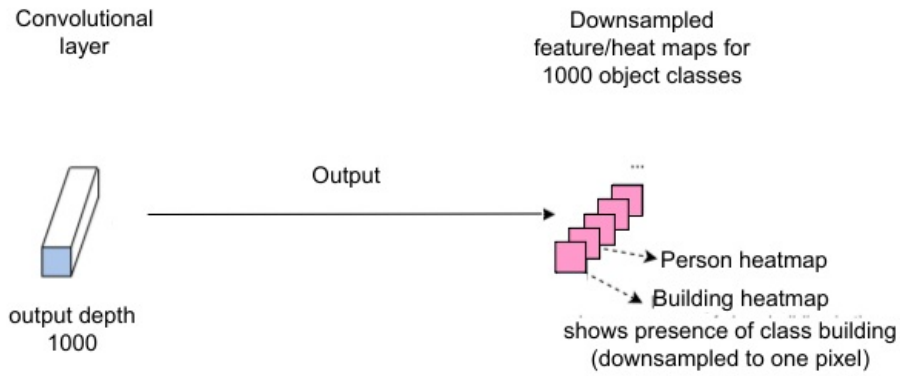
FC layers are equivalent to convolutions with kernels that cover their entire input regions

Converting FC layers to convolutional layers form the
basis of the dense (pixel-wise) prediction aka segmentation



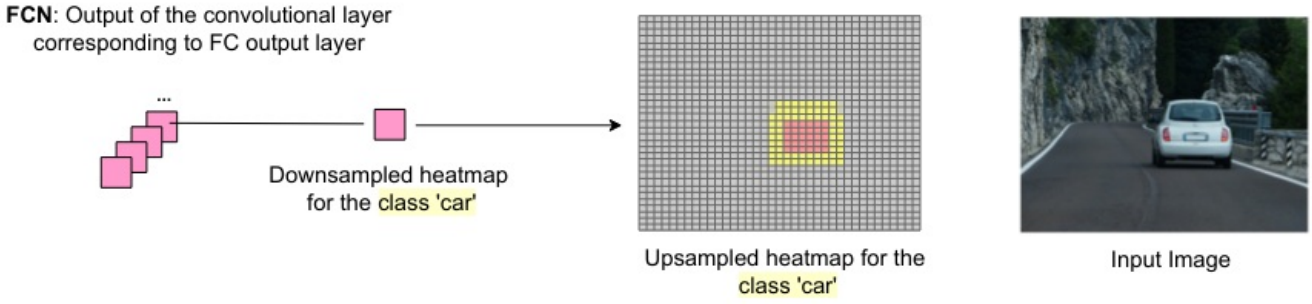
FC layers replaced with equivalent convolutional layers that cover entire input regions

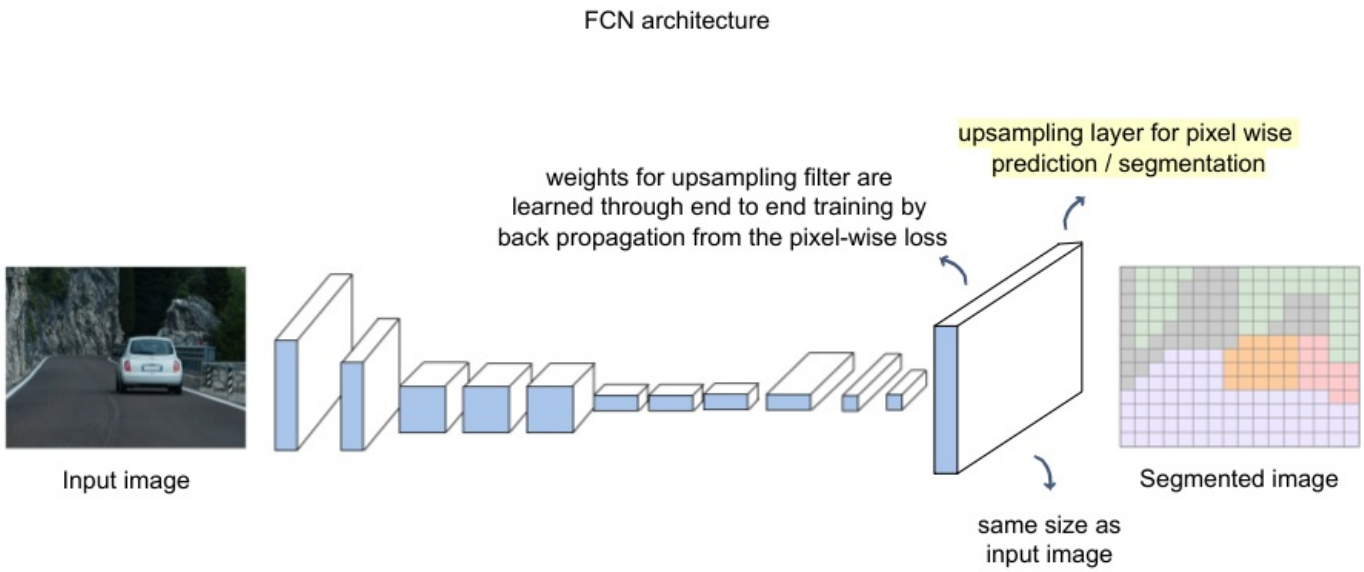
We need to upsample the feature maps to obtain pixel wise segmentation equal to the size of input image



Equivalent of Convolutional layer to downsampled feature maps

We need to upsample the heatmaps to obtain pixel wise segmentation same as the size of the input image

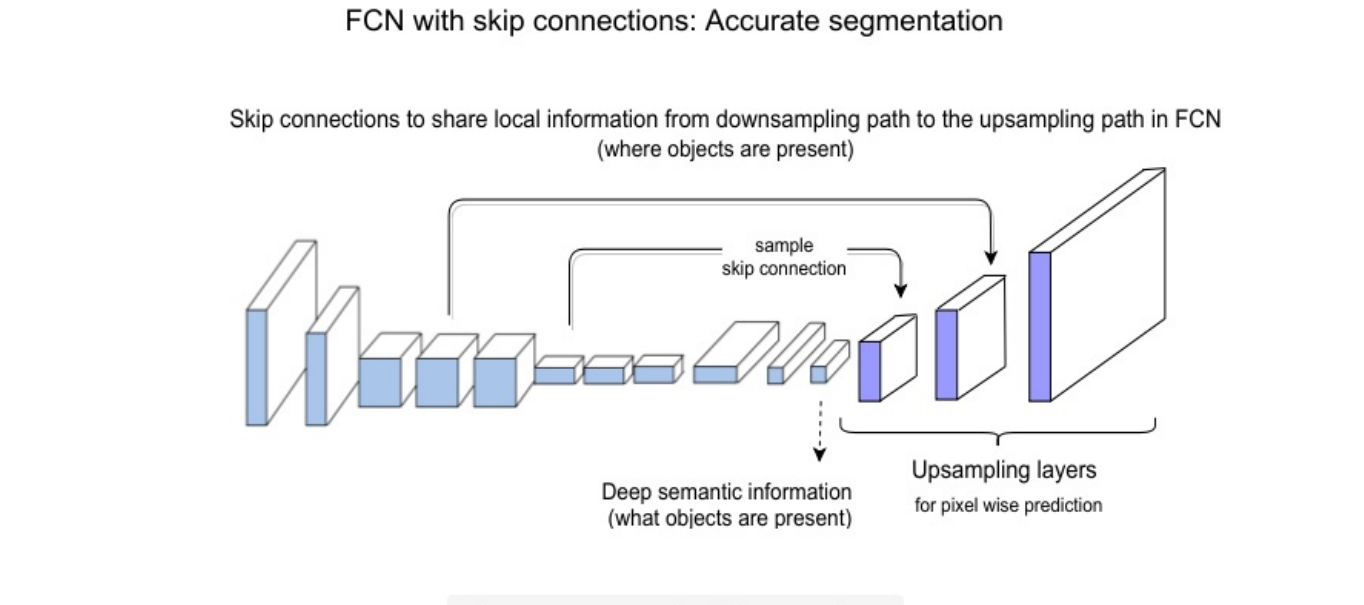




Upsampling layer is added at the end of the convolution and pooling process to obtain pixel wise segmentation from downsampled feature/heat maps

7 of 7

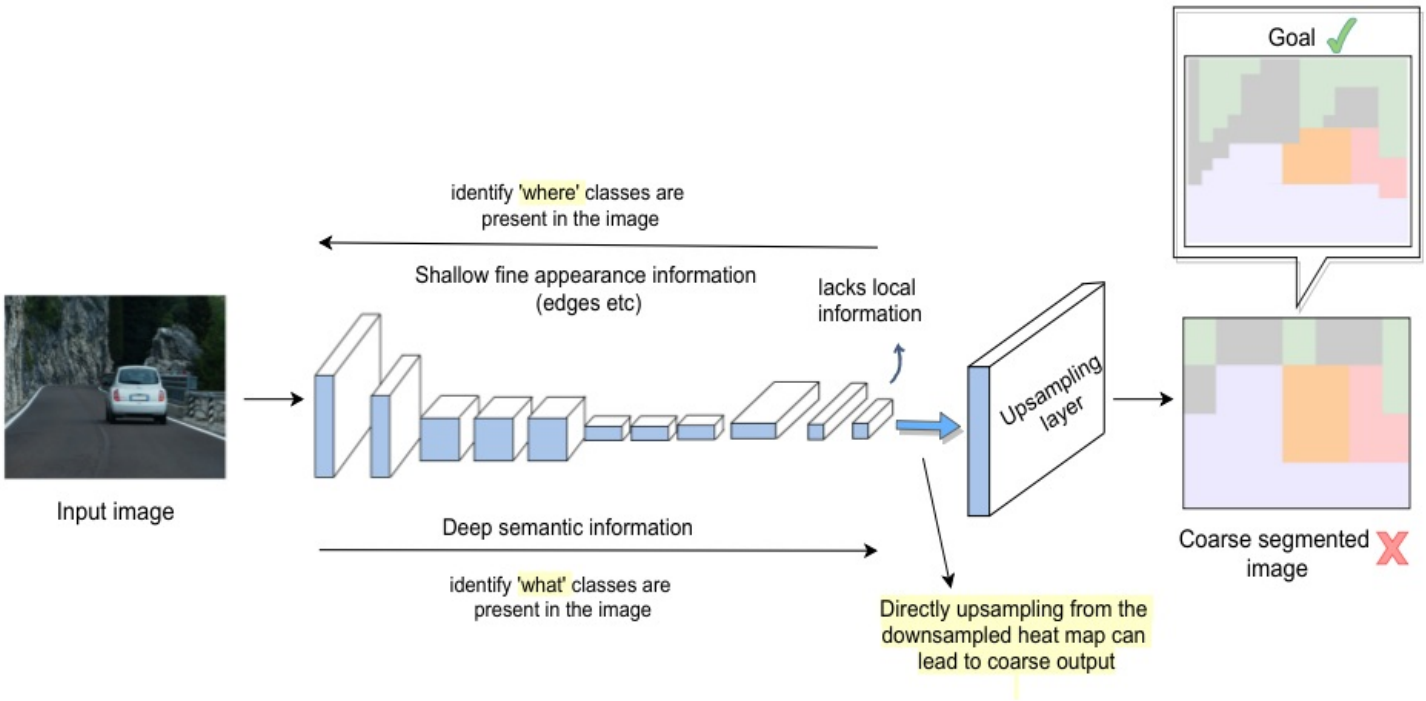
The convolutional layers at the end (instead of the fully connected layers) also allow for dynamic input size. The output pixel-wise classification tends to be coarse, so you make use of skip connections to achieve good edges. The initial layers capture the edge information through the deepness of the network. You use that information during our upsampling to get more refined segmentation.



Accurate segmentation: Skip connections

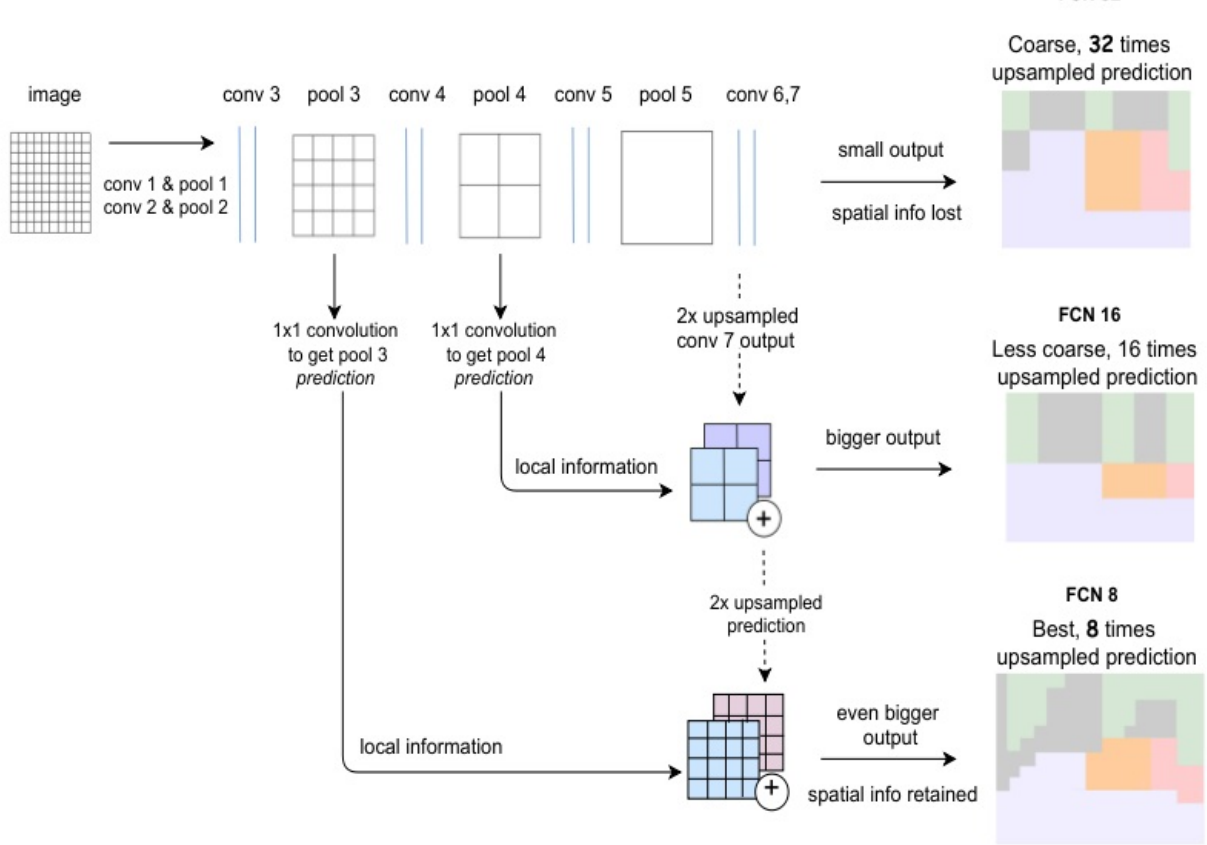
1 of 3

FCN without skip connections: Coarse segmentation



Coarse segmentation: No skip connection

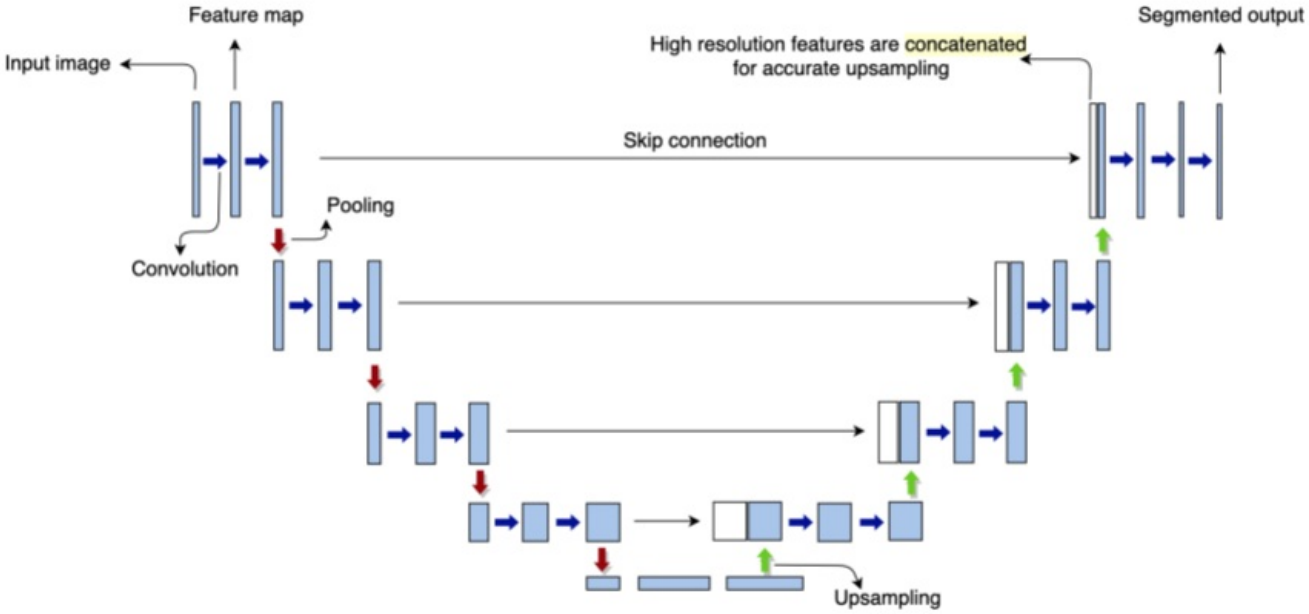
How skip connections work



How skip connections work

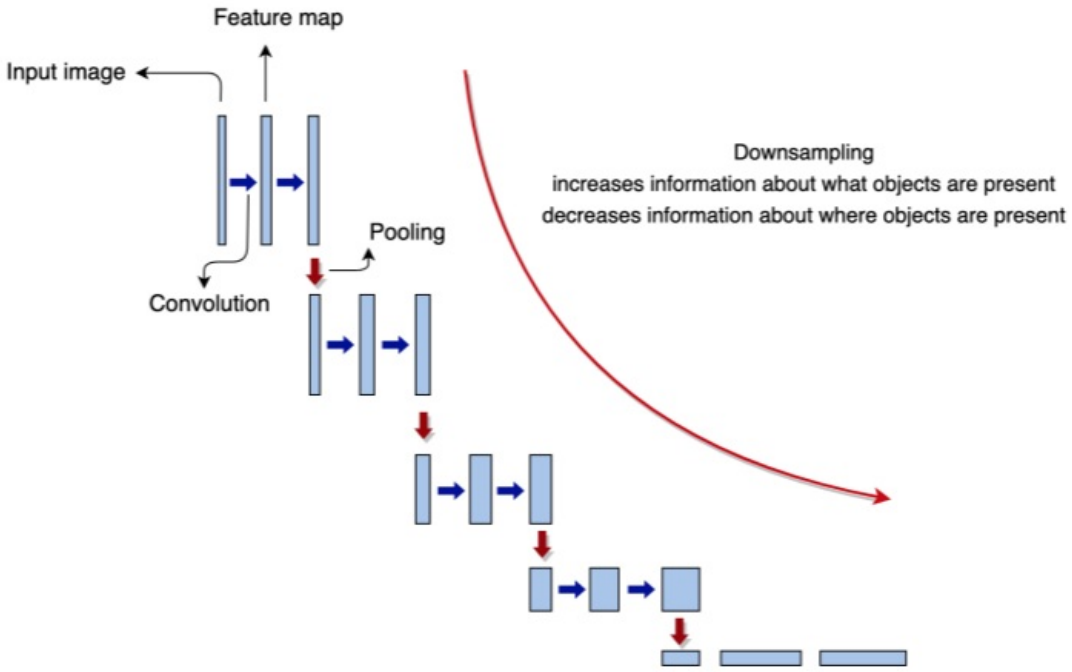
U-Net

U-Net is commonly used for semantic segmentation-based vision applications. It is built upon the FCN architecture with some modifications. The architectural changes add a powerful feature in the network to require less training examples. The overall architecture can be divided into two halves. The first half down-samples the convolutional features through the pooling operation. The second half up-samples the feature maps to generate the output segmentation maps. The upsampling process makes use of skip connections to concatenate high-resolution features from the downsampling portion. This allows for accurate upsampling.



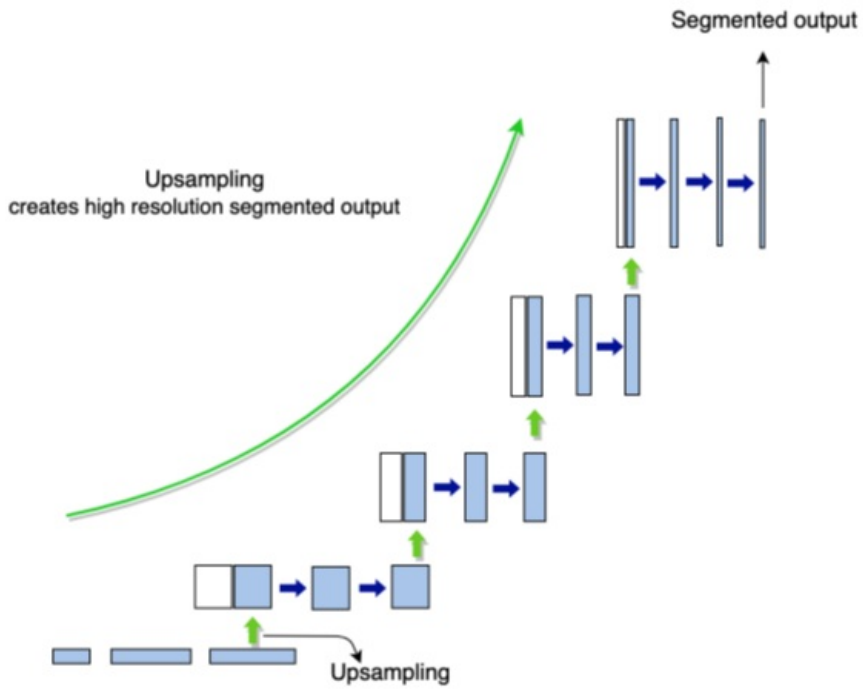
U-Net architecture: like an FCN it does not have FC layers

1 of 4



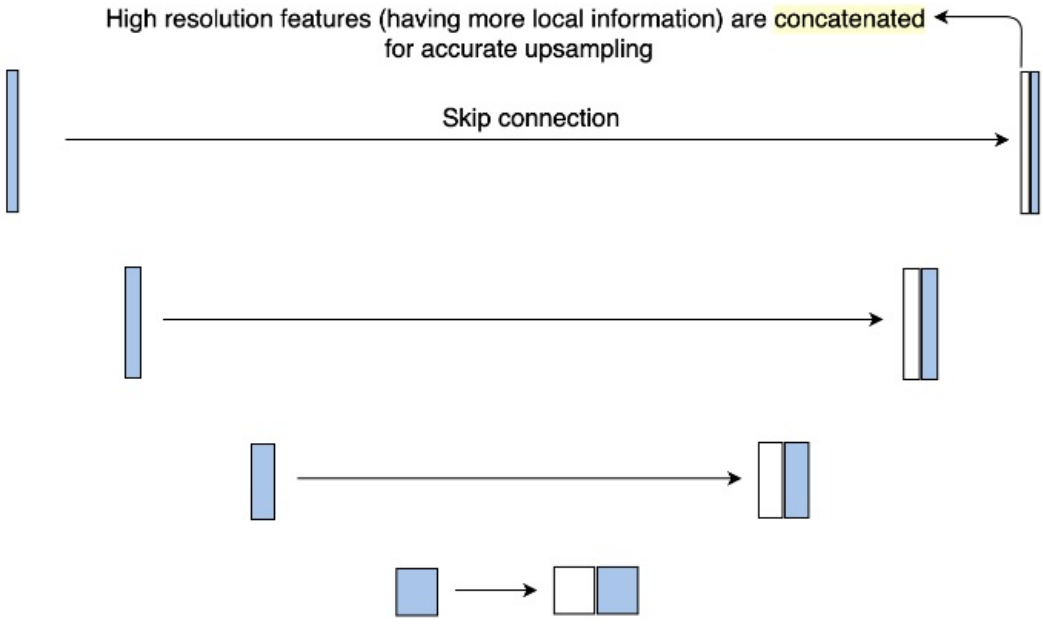
Downsampling

2 of 4



Upsampling

3 of 4



Using skip connections during upsampling allows strong semantic features at each resolution scale

4 of 4

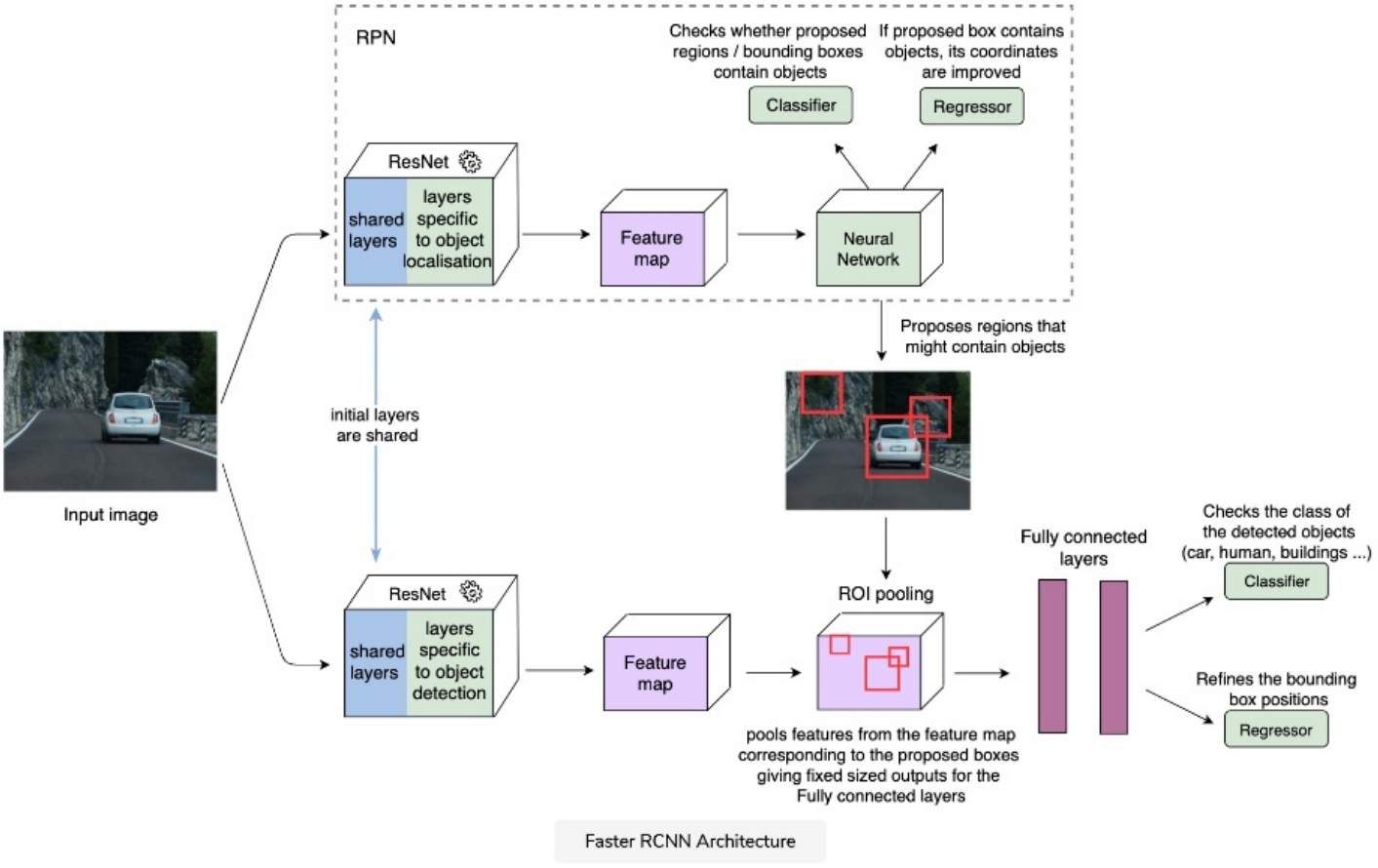
Mask R-CNN

Mask R-CNN is a powerful deep neural network that is widely used for many real-world instance segmentation applications. It combines the best of both the worlds: Faster R-CNN for object detection and localization and FCN for pixel-wise instance segmentation of objects.

Minimize

Faster R-CNN

It is a state-of-the-art deep neural network that proposes regions in the image which contain different objects by drawing bounding boxes around them. The objects are detected by a fully-convolutional **Region Proposal Network (RPN)** that generates predictions on image regions through a sliding window fashion. RPN receives image features from a backbone CNN-based image classifier, e.g., ResNet. The following figure shows the block diagram architecture of faster R-CNN.



Faster RCNN Architecture

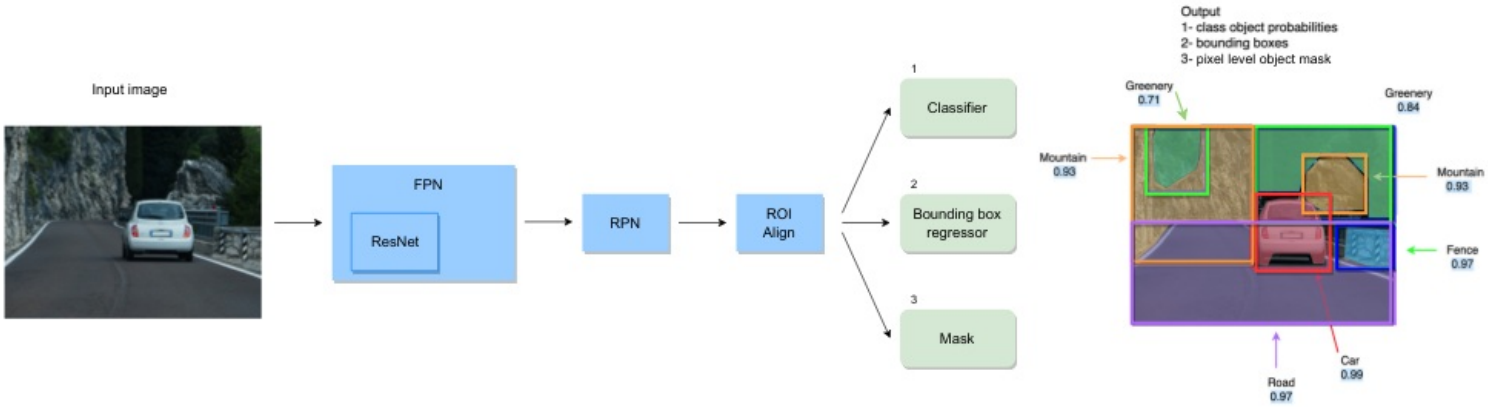
Mask R-CNN architecture is constructed by using a convolutional backbone of a powerful CNN classifier (e.g. ResNet) followed by a Feature Pyramid Network (FPN). FPN extracts feature maps from the input image at different scales, which are fed to the Region Proposal Network (RPN). RPN applies a convolutional neural network over the feature maps in a sliding-window fashion to predict region proposals as bounding boxes that will contain class objects. These proposals are fed to the RoI Align layer that extracts the corresponding ROIs (regions of interest) from the feature maps to align them with the input image properly. The ROI pooled outputs are fixed-size feature maps that are fed to parallel heads of the Mask R-CNN.

To reduce the computational cost, Mask R-CNN has three parallel heads to perform

1. Classification

2. Localization
3. Segmentation

The output layer of the classifier returns a discrete probability distribution of each object class. Localisation is performed by the regressor whose output layer generates the four bounding-box coordinates. The third arm consists of an FCN that generates the binary masks of the predicted objects. The following is a block diagram of the Mask R-CNN.



Mask RCNN

> It is worth mentioning that most of the deep vision architectures are not inherently designed to preserve sensitivity for different scales, sizes and rotations of the objects/images. The pooling operation in these networks is only able to capture the translational variance of the labeled objects/images. Hence, it is critical to cover the variation in scale and rotation of your training images and keep the testing image size similar to the image size in your training dataset.

Transfer learning

Let's answer the question of how to utilize the discussed SOTA segmentation models to build one for the self-driving car.

Transfer learning is a widely used technique in the field of deep learning: utilizing pre-trained powerful deep neural networks (DNNs) for weak or small datasets. When you kickstart a deep learning project with a small ground-truth (human-annotated) training data, it is often challenging to scale the performance of your model on a standard-sized large training dataset. Since increasing the size of human-annotated data can be time-consuming, transfer learning helps take advantage of a powerful deep neural network that is pre-trained on a different but large human-annotated dataset that is similar in nature to the dataset in hand.

> We use transfer learning to build upon learned knowledge from one dataset to improve learning in the dataset.
>
> The trick of transfer learning can be explained through the functionality of the human vision system, which is the basis of convolutional and pooling layers in deep networks. There is a striking similarity between the low and high-level feature detectors (simple and complex neurons) of the primate vision

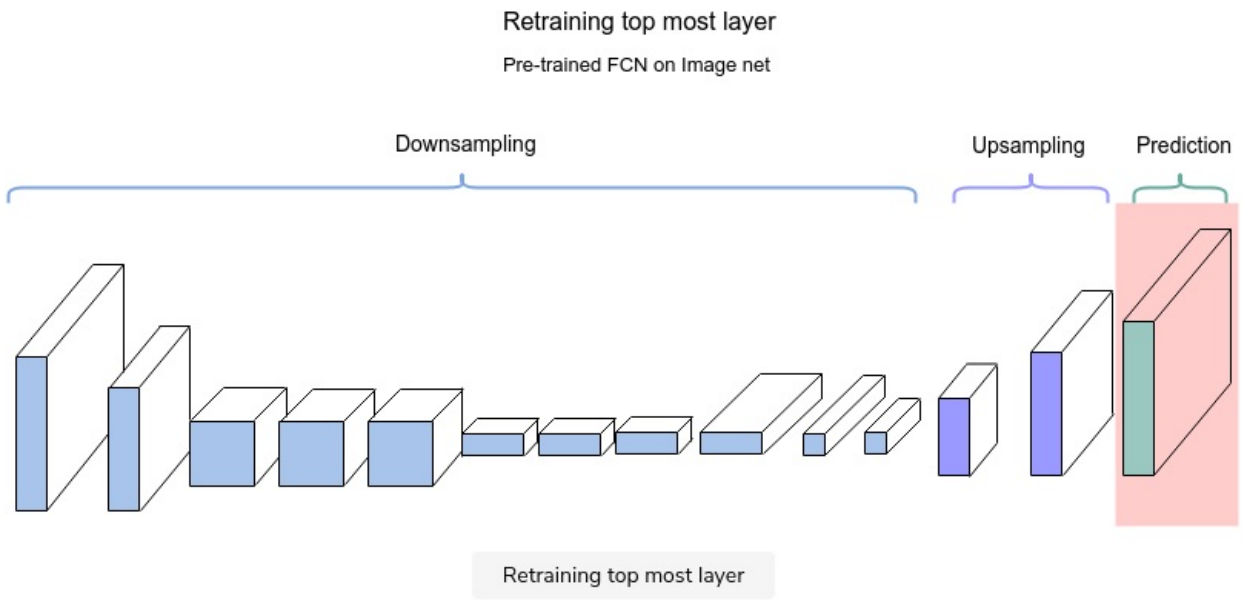
> system and convolutional and pooling operations in deep networks.

In a trained model, the low-level feature detectors are fine-tuned to filter high-frequency visual information (edges), which are mostly common in natural images. For example, you can transfer the learned edge detectors of a model trained to detect cats to detect dogs. For more similar objects/images, this similarity can also be correlated in high-level feature detectors. In this regard, freezing the layers of a trained deep neural network in transfer learning helps take advantage of the learned feature detectors from a large and rather similar natural imaging dataset.

Assume that, from the above-mentioned SOTA models, you want to utilise a pre-trained FCN (on ImageNet dataset) for performing segmentation for your self-driving vehicle. Let's see three different approaches where you can apply transfer learning.

Retraining topmost layer

You can take advantage of the large features' bank in the ImageNet FCN and re-train it for your smaller driving images dataset. In this case, you need to update the final pixel-wise prediction layer in the pre-trained FCN (i.e., replace the classes in ImageNet trained model with classes in driving image set) and retrain the final layer through an optimizer while freezing the remaining layers in the FCN.



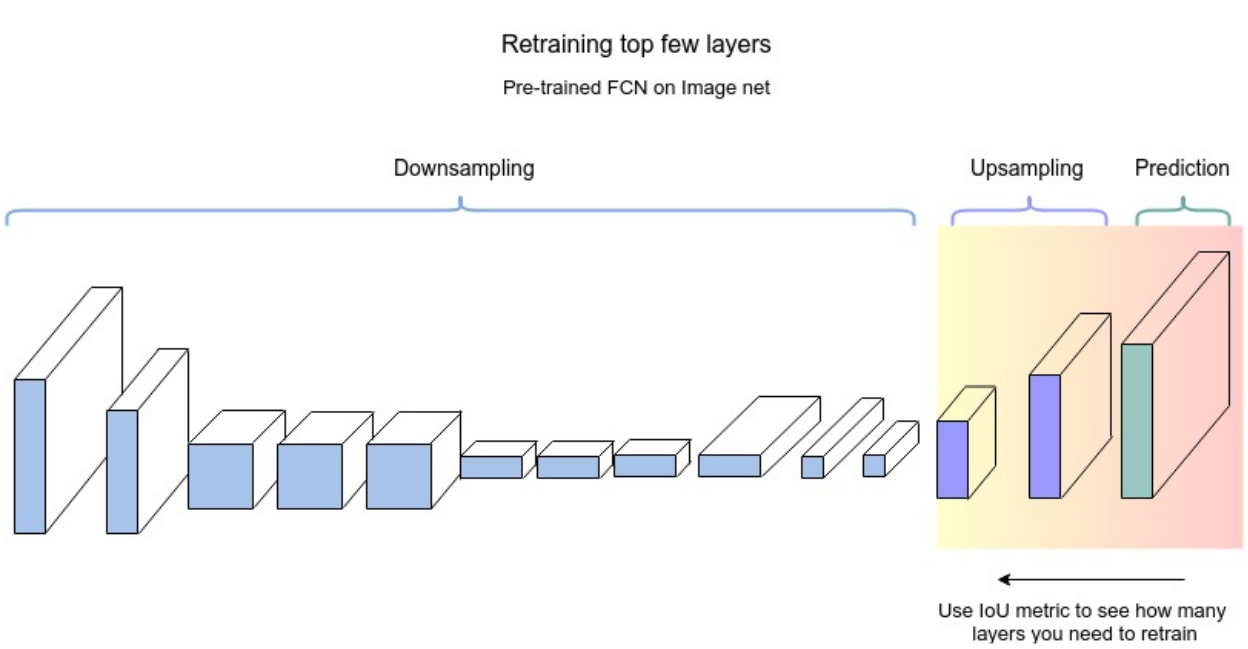
Retraining top most layer

This approach makes the most sense when the driving images data is limited and/or you believe that the current learned layers capture the information that you need for making a prediction.

Retraining top few layers

Similarly, in cases where you have a medium-sized driving images dataset available, you can also try retraining some deeper upsampling layers to improve the accuracy of your segmenter.

You can *experiment with how many upsampling layers you need to retrain* by looking at the *IoU metric* to see if retraining any further improves the performance of the segmentation model.



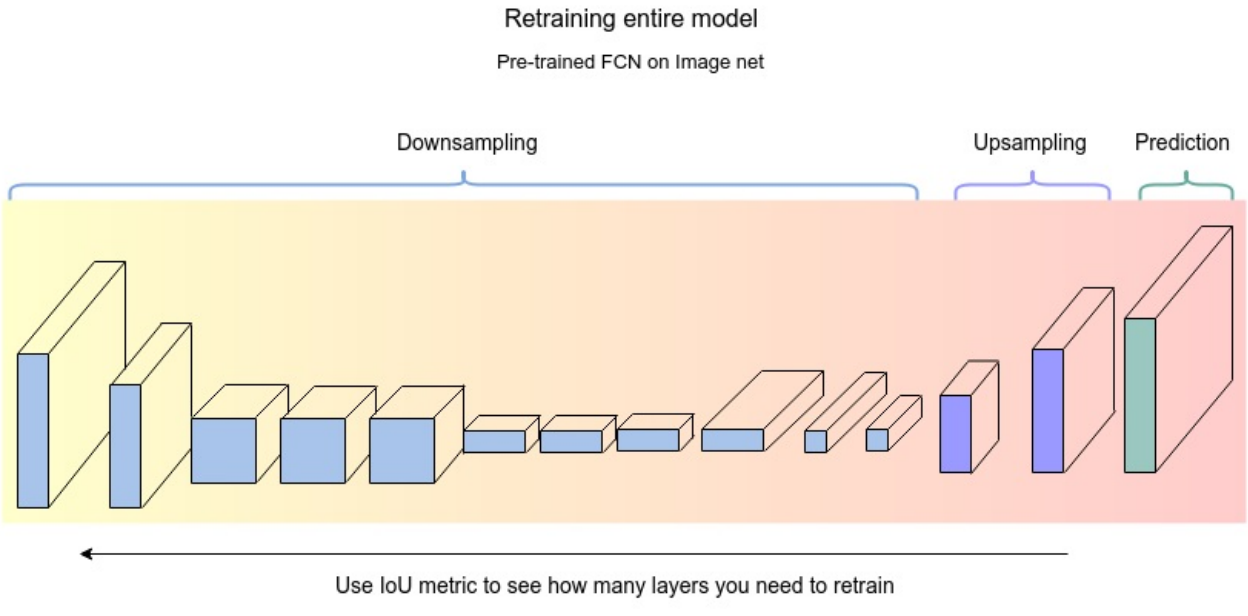
Retraining top few layers

This approach makes the most sense when you have a decent amount of driving images data. Also, shallow layers generally don't need training because they are capturing the basic image features, e.g., edges, which don't need retraining.

Retraining entire model

As the size of your dataset increase, you can consider retraining even more layers or retraining the entire model. Once again, you can use the IoU metric to evaluate the optimal number of layers whose retraining can increase model performance.

Generally, retraining the entire network is laborious and time-consuming. It is usually done only if the dataset under consideration has completely different characteristics from the one on which the network was pre-trained.



Retraining entire model

On top of choosing or designing the optimal DNN architecture, one can also be creative in adding some extra

features in the network to handle the data bias and variance issues. For instance, adding a regularizer (e.g., L1/L2) after a DNN architecture, making a hyper-parameter sweep for fine-tuning and experimenting with different optimizers (e.g., Adam and SGD) during training can further enhance the performance of the model.

← Back
Training Data Generation

Next →
Problem Statement

Mark as Completed

Report an Issue Ask a Question