# Deep Learning Assignment

## Introduction

Lane prediction is an integral aspect of autonomous driving. Not only lanes act as a guide to steer vehicles but also helps in safe driving. Lane detection is a proven use case of semantic image segmentation. Semantic image segmentation involves classifying each pixel of a given image. This not only helps in understanding what objects are available in the image, but also gives a wider understanding of the entire landscape given in the image.

With recent advancements in deep learning, there are several model architectures we can use to solve the given problem. Architectures like U-Net, Mask RCNN, SegNet, PSPNet, RefineNet, DeepLab v3 are widely used in solve semantic image segmentation.

## Given Problem:

We are given a sample of 501 images of street view along with their labels. We need to detect the lanes from the given images.

## Solution:

The given problem is a possible use case of semantic image segmentation. We are going to build a quick and simple solution using U-Net architecture.

## What is U-Net ?

In simple words, a U-Net architecture can be thought of as an encoder and decoder network.
The basic idea behind encoder and decoder is facilitate faster training while preserving pixel level information in the output predictions. In short, the encoder reduces the image size by downsampling the images and decoder gradually upsamples the image to the original input size. We can say, the encoder reduces the spatial dimension and decoder gradually recovers the object details while maintaining precise localization combined with contextual information.

U-Net’s architecture has a U like shape as seen here.

## How to build the model ?

I built the model on Google Colab 12 GB GPU instance using Python.

Following is my approach to build the model:

1. The given image size is 720 X 1280 px. We resize the images to 256 x 256. We resize it because the machine ran out of memory on any size bigger than this.
2. We read and convert the input images (train data) into a three dimensional matrix (256 x 256 x 3).
3. We read and convert the input labels (masks) into one dimensional matrix since they are grey-scale images (256 x 256 x 1).
4. We split the input images into train and validation set. 20% of the train data is used as validation set.
5. We feed the matrix of input images to a U-Net architecture. We train the model for 30 epochs with a batch size of 16.
6. The model predicts probability of each pixel contributing to being a lane or not.

**How to measure performance of the model ?**

We used **Mean average precision at different (IoU) thresholds** metric to evaluate model performance. The model's score was 0.56 on validation data. Basically, IOU  measures the percent overlap between actual and predicted labels (or masks). It is calculate using: $IoU(A,B) = (A \cap B) / (A \cup B)$ where A, B refers the the actual and predicted image vectors. IOU is also known as Jaccard Index.

The final metric is calculated as:

1. For each image, for each value of different threshold ranging between  0.5 to 0.95, we calculate the precision and get the mean precision.
2. For all the images, we divide the average precision for each image by number of images to get average precision score for all images in the evaluation data.

We used dice coefficient as the loss function. It is basically a measure of overlap between actual and predicted values. It ranges between 0 and 1 where 1 denotes a perfect match.

**How to automate this solution?**

We can create a API with an endpoint which receives an image, makes predictions and returns a matrixes of pixel values of the same dimension as input image.
Following the stack we can use to make a quick POC:

1. Programming Language: Python
2. Libraries: Keras, Numpy, Matplotlib, CV2, Scikit-learn, Tensorflow.
3. Framework: Flask or aiohttp (asynchronous and faster)
4. Datapipeline: Once the POC is tested, we will need to create a data pipeline to schedule the model training automatically. We can use Airflow for this purpose.
5. Storage: We can use AWS S3 buckets for storing the models.
6. Supporting tools: We can use Docker for API deployment on Openshift (Kubernetes) clusters.
7. Training instance: GPU / CPU enabled instance

*Additional:* If the dataset is huge and data pre-processing takes long time, we could spin up AWS EMR clusters and use spark to do parallel processing of image data.

**Things I could have done:**

1. Image augmentation: Using techniques like horizontal, vertical flips, zoom, rotation etc we could have increased the sample size.
2. Using pre-trained encoder model: We could use some pre-trained model like VGG16 as the base models in the U-net architechture
3. Using other architectures: Other architectures like DeepLab, Segnet could have been explored.
4. Training model for more epochs on full size images: Instead of resizing, we could have trained the model on full image size thus maintaining better pixel level information.
5. Ensemble Models: Training multiple models and averaging pixel level predictions.