CS 512: Title: Towards End-to-End Lane Detection: An Instance Segmentation Approach

Pooja Jaiswal
Department of Computer Science Illinois Institute of Technology

ABSTRACT

This report is built upon a branched ENet architecture with a shared encoder and different decoder branches. We propose an instance segmentation algorithm by learning a deconvolution network. We learn the network on top of the convolutional layers adopted from VGG 16- layer net. The deconvolution network is composed of deconvolution and unpooling layers, which identify pixel-wise class labels and predict segmentation masks. We apply the trained network to each proposal in an input image and construct the final binary segmentation map by combining the results from all proposals in a simple manner. The proposed algorithm mitigates the limitations of the existing methods based on fully convolutional networks by integrating deep deconvolution network and proposal-wise prediction; our segmentation method typically identifies detailed structures and handles objects in multiple scales naturally. Our network demonstrates outstanding performance in tuSimple dataset figure1, and we achieve the best accuracy (97.5% approx.) among the methods trained with no external data through ensemble with the fully convolutional network.

INTRODUCTION

In modern time Camera-based lane detection is a vital step towards the fully autonomous self-driving vehicle in such environmental perception as it allows to properly position itself within the road lanes. It is also crucial for any subsequent lane departure or trajectory planning decision instantly. Traditional lane detection methods rely on a combination of highly specialized handcrafted features and heuristics to distinguish lanes segments; however, those techniques were certain limitations such as outlining lane it includes roadside trees, fences, or intersections which gives false identifications. Apart from the ability of the networks to segment out lane markings better, their big receptive field allows them to also estimate lanes even in cases when no markings are present in the image. However, the generated binary lane segmentations still need to be disentangled into the different lane instances. To handle this issue, a few methodologies have applied post-processing techniques that rely again on heuristics, usually guided by geometric properties in an instance within the lane class. Inspired by the success of dense prediction networks in binary segmentation and instance segmentation tasks we design a branched, multi-task network for lane instance segmentation, consisting of a lane segmentation branch and a lane embedding branch that can be trained end-to-end.

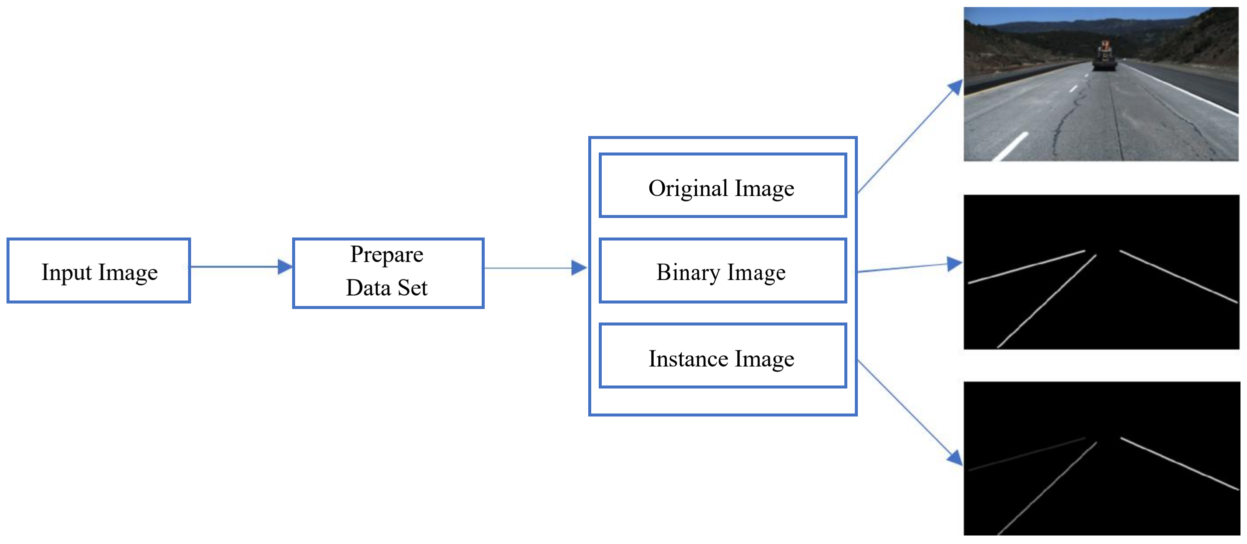


Figure 1: Dataset consist of three different types of Images.

The lane segmentation branch has two output classes, background or lane, while the lane embedding branch further disentangles the segmented lane pixels into various lane instances. By splitting the lane detection problem into the two tasks, we can fully utilize the power of the lane segmentation branch without it having to assign different classes to different lanes. Instead, the lane embedding branch, which is trained using a clustering loss function, assigns a lane id to each pixel from the lane segmentation branch while ignoring the background pixels. By doing so, we alleviate the problem of lane changes and we can handle a variable number of lanes. An overview of our full pipeline can be seen in figure 3. Our contributions can be summarized to the following: (1) A branched, multi-task architecture to cast the lane detection problem as an instance segmentation task, that handles lane changes and allows the inference of an arbitrary number of lanes. The lane segmentation branch outputs dense, per-pixel lane segments, while the lane embedding branch further disentangles the segmented lane pixels into different lane instances. (2) A network architecture that works based on a VGG16 architecture, which is a very large model designed for multi-class classification. These references propose networks with huge numbers of parameters, and long inference times which require processing images at rates higher than 10 fps.

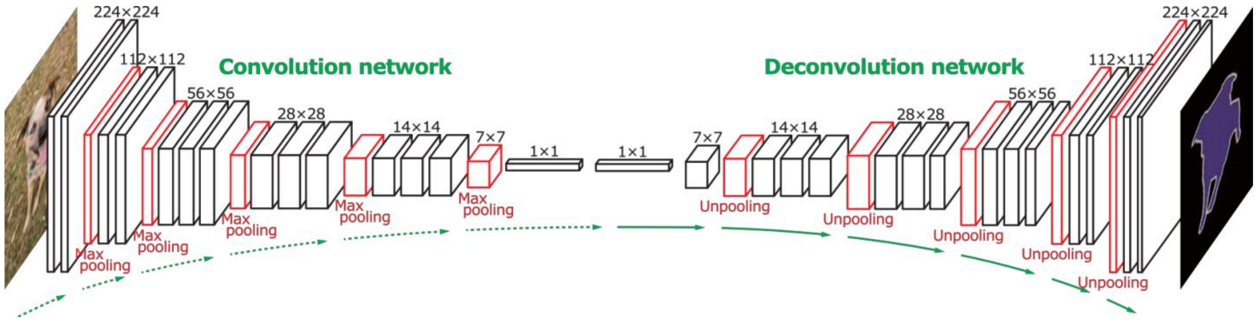


Figure 2: A network architectures that works based on a VGG16 (Encoder – Decoder) architecture

PROPOSED METHOD

To predict depth, binary and instance segmentation in real-time, we modify the ENet architecture into a multi-branched network, the network, which we will refer to as LaneNet consists of an encoding step that has three stages (stage 1, 2, 3) and a decoding step that has two stages (stage 4, 5). Since the ENet decoding step is merely for upscaling and finetuning the output of the encoding step, sharing the full encoder (stages 1, 2, 3) between all branches would lead to poor results. Instead, our multi-branch network is constructed as follows: our shared" encoder" consists of stages 1 and 2 of the original Enet network, before continuing to each branch that combines stage 3 of the original ENet encoder with stages 4 and 5 of the original ENet decoder.

Binary Segmentation:

The segmentation branch of LaneNet has trained to output a binary segmentation map, indicating which pixels belong to a lane and which not. To construct the ground-truth segmentation map, we connect all ground-truth lane together, forming a connected line per lane including objects like occluding cars, or also in the absence of explicit visual lane segments, like dashed or faded lanes. This way, the network will learn to predict lane location even when they are occluded or in adverse circumstances. The segmentation network is trained with the standard cross-entropy loss function.

Instance Segmentation:

The instance segmentation using a typical feed-forward network without having to resort to slower detect-and-segment approaches, we use a recently introduced discriminative loss function suited for real-time instance segmentation that can be plugged into an off-the-shelf network. The intuition behind the proposed loss function is that pixel embeddings (i.e. the network's output for each pixel) with the same instance should end up close together, while embeddings with a different instance should end up far apart.

REQUIREMENT

Hardware Requirement:

The required space of 0.7MB for ENet parameters so that it possible to fit the whole network in an extremely fast on-chip memory in embedded processors. Also, this alleviates the need for model compression making it possible to use general -purpose neural network libraries. However, if one needs to operate under incredibly strict memory constraints, these techniques can still be applied to ENet as well.

Software Requirement:

The tuSimple lane dataset for testing deep learning methods on the lane detection task which consists 3626 training and 2782 testing images on different circumstances like disturbances, various illumination conditions, including daytime, dusk, a night with or without lighting and different road types with 2-lane/3-lane/4-lane or more highway road.

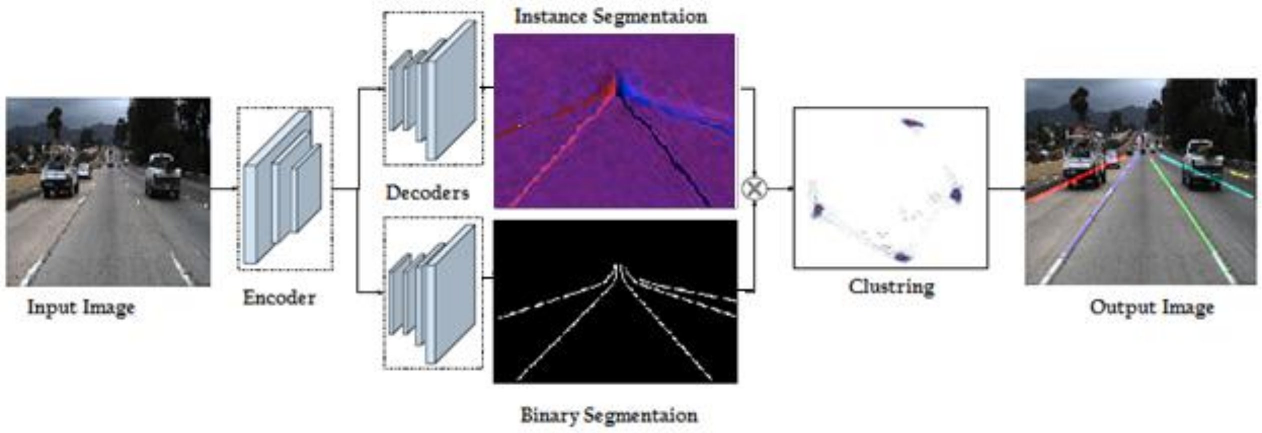


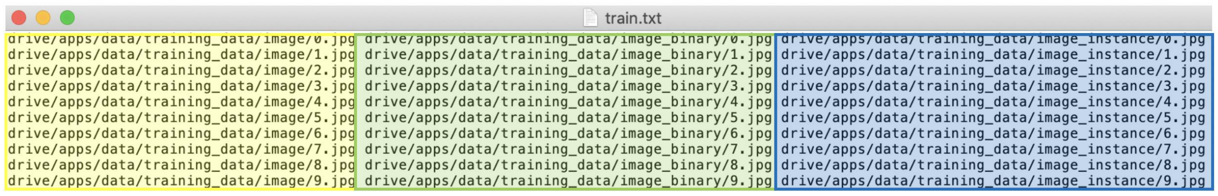
Figure 3: LaneNet Model Design

IMPLEMENTATION

To achieve "End-to-End Lane Detection through Instance Segmentation", CNN (Encoder – Decoder) model is used. To make this possible, a model is train on the "TuSimple" data set and further the trained model is used for prediction of lanes. Below steps are involved for achieving lane detection on the road image.

1. Preparation of training & testing data set :-

For training, we have used the TuSimple dataset which contains 3626 training image with ground truth JSON file and 2782 testing images with its ground truth JSON file. As for lane detection, we are using instance segmentation approach, which requires binary and instance images. We have implemented the process of generating binary and instance images and once the images are generated, we will now generate the text files for training and testing of each dataset. One is "train.txt" and other is "val.txt". "train.txt" contains the list of URL path for the training images and "val.txt" contains the list of URL path for testing images. The structure of the file looks like:-



```
drive/apps/data/training_data/image/0.jpg drive/apps/data/training_data/image_binary/0.jpg drive/apps/data/training_data/image_instance/0.jpg
drive/apps/data/training_data/image/1.jpg drive/apps/data/training_data/image_binary/1.jpg drive/apps/data/training_data/image_instance/1.jpg
drive/apps/data/training_data/image/2.jpg drive/apps/data/training_data/image_binary/2.jpg drive/apps/data/training_data/image_instance/2.jpg
drive/apps/data/training_data/image/3.jpg drive/apps/data/training_data/image_binary/3.jpg drive/apps/data/training_data/image_instance/3.jpg
drive/apps/data/training_data/image/4.jpg drive/apps/data/training_data/image_binary/4.jpg drive/apps/data/training_data/image_instance/4.jpg
drive/apps/data/training_data/image/5.jpg drive/apps/data/training_data/image_binary/5.jpg drive/apps/data/training_data/image_instance/5.jpg
drive/apps/data/training_data/image/6.jpg drive/apps/data/training_data/image_binary/6.jpg drive/apps/data/training_data/image_instance/6.jpg
drive/apps/data/training_data/image/7.jpg drive/apps/data/training_data/image_binary/7.jpg drive/apps/data/training_data/image_instance/7.jpg
drive/apps/data/training_data/image/8.jpg drive/apps/data/training_data/image_binary/8.jpg drive/apps/data/training_data/image_instance/8.jpg
drive/apps/data/training_data/image/9.jpg drive/apps/data/training_data/image_binary/9.jpg drive/apps/data/training_data/image_instance/9.jpg
```

The text files will be used by the training module to fetch the list of images into the program using the paths mentioned in the text file. Once the data set is prepared, it will be feed into the CNN for training.

For generation of binary and segmentation image, we first parse the JSON truth file sequentially. The structure of the JSON file is mentioned below:-

```
{
  'raw_file': 20th frame file path in a clip.
  'lanes': A list of lanes. For each list of one lane, the elements are width values on image.
  'h_samples': A list of height values corresponding to 'lanes', which means len(h_samples) == len(lanes[i])
}
```

On parsing the JSON file, we create a point pair corresponding to a lane point. Once the list of lane points are obtained, the lane points are drawn using **polylines** method in OpenCV onto a blank image with a black background with the same size as of original image. The image generated is saved as grayscale without channel information. For a binary image the polylines use white color to plot lines and for instance image, it uses different shades of gray. Figure 1 shows the sample binary and instance image. File "**plot_json.py**" contains the code for the implementation of image set generation for training and testing set.

2. Training Module :

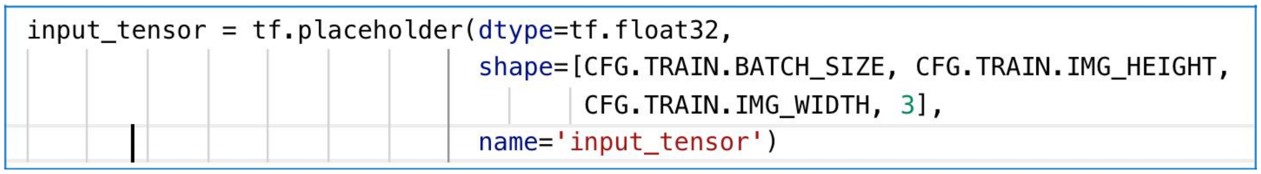
Once the dataset is prepared, we move forward to building the training module. As per the paper, the architecture of the training module depends on the encoder and decoder. The paper referenced another paper for the network design i.e. "A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation" where encoder module uses the VGG16 encoder module and decoder module is fully convolutional Network decoder. "**train_model.py**" file implements a complete training module.

Steps involved in preparing the training module for training and validation is mentioned below:-

a) A global file is declared which contains all the important parameters that are being used in the program. Some are batch_size, learning_rate, no_of_epochs,learningrate_decay_rate, etc.

b) Initially from the provided "train.txt" and "val.txt", we prepare the training and validation dataset which contains the list of images path for original, binary and instance image in a random fashion to cover all the possible scenarios. The randomization logic and dataset creation are done in "**data_processor.py**". The class also contains method "**next_batch**" which is called while training and returns the set of image array depending on the passed batch size. The returned batch of data is further used for training.

c) Next the three input tensor is prepared for image as "**input_tensor**", binary as "**binary_label_tensor**" and instance as "**instance_label_tensor**" images. The shape of the tensor used for example [8, 256, 512, 1] which is [batch_size, image_height, image_width, image_channel]. The tensor will be created as mentioned below:-



```
input_tensor = tf.placeholder(dtype=tf.float32,
                              shape=[CFG.TRAIN.BATCH_SIZE, CFG.TRAIN.IMG_HEIGHT,
                                     CFG.TRAIN.IMG_WIDTH, 3],
                              name='input_tensor')
```

d) Now the loss is computed for the CNN model. Here CNN uses encoder and decoder model for training model. The "compute_loss" function is explained later in the report. This loss method returns array of losses containing "total_loss", "binary_seg_logits", "instance_seg_logits", "binary_seg_loss" and "discriminative_loss". These values will be further used in the tensor. Code Snippet for the calculating loss is mentioned in "**merge_model.py**".

e) Accuracy is calculated on the "binary_seg_logits" using the below code :-

```
# it just normalizes the values. The outputs of softmax can be interpreted as
probabilities.
```

```
out_logits = tf.nn.softmax(logits=out_logits)
# Returns the index with the largest value across axes of a tensor
out_logits_out = tf.argmax(out_logits, axis=-1)
out = tf.argmax(out_logits, axis=-1)
# Inserts a dimension of 1 at the dimension index axis of input's shape
out = tf.expand_dims(out, axis=-1)

# Returns the truth value of (binary_label_tensor == 1) element-wise. and
# returns the coordinates of true elements of condition
idx = tf.where(tf.equal(binary_label_tensor, 1))
# Gather slices from out into a Tensor with shape specified by indices
pix_cls_ret = tf.gather_nd(out, idx)
# Computes number of nonzero elements across dimensions of pix_cls_ret tensor
accuracy = tf.count_nonzero(pix_cls_ret)
accuracy = tf.divide(accuracy, tf.cast(tf.shape(pix_cls_ret)[0], tf.int64))
```

f) **Momentum optimizer** is used for minimizing the loss. Here the learning rate is initially taken as 0.0001. Moreover we have also implemented learning rate decay for the optimizer in the process of training on every 500 epochs. Initially the learning rate take bigger steps but as learning approaches convergence, having a lower learning rate allows us to take smaller steps which improves learning of the model. Hence the learning rate decay is being used.

```
learning_rate = tf.train.exponential_decay(CFG.TRAIN.LEARNING_RATE,global_step,
        CFG.TRAIN.LR_DECAY_STEPS, CFG.TRAIN.LR_DECAY_RATE, staircase=True)

optimizer = tf.train.MomentumOptimizer(
        learning_rate=learning_rate, momentum=0.9).minimize(loss=total_loss,
        var_list=tf.trainable_variables(),global_step=global_step)
```

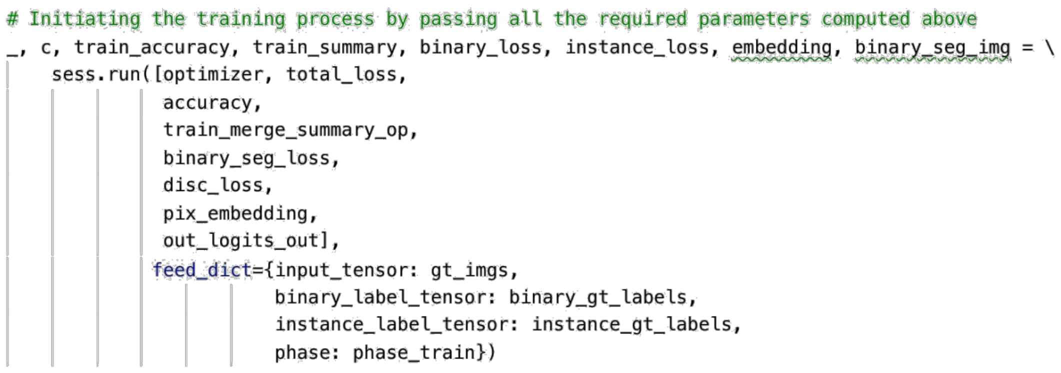
g) Now, as we are training a model, that model needs to be saved. After training process, the saved model will be used for predictions. “tf.train.Saver ( )” is used for saving model.

h) Also, to view the status of the training and visualize the training process, Tensorboard is used. So, next we configure the tensor board which will display the graphs of how the training is approaching. The tensor board will display “train_cost”, “val_cost”, “train_accuracy”, “val_accuracy”, “train_binary_seg_loss”, “val_binary_seg_loss”, “train_instance_seg_loss”, “val_instance_seg_loss” and “learning_rate”.

i) Before training starts, we check for any weights that have been provided through CLI. If provided, we begin training from initializing the provided weight file. By default when there is no weight file, training uses pretrained “VGG16” weight file.

j) In the training process

- For each epoch, we first fetch the label files i.e. the images for original, binary and instance image in batch.
- All the labels are resized for the tensor to accept.
- Now the training process begins. Below is the snippet to initiate training process.



```
# Initiating the training process by passing all the required parameters computed above
_, c, train_accuracy, train_summary, binary_loss, instance_loss, embedding, binary_seg_img = \
    sess.run([optimizer, total_loss,
              accuracy,
              train_merge_summary_op,
              binary_seg_loss,
              disc_loss,
              pix_embedding,
              out_logits_out],
             feed_dict={input_tensor: gt_imgs,
                        binary_label_tensor: binary_gt_labels,
                        instance_label_tensor: instance_gt_labels,
                        phase: phase_train})
```

k) Similarly the validation process is done.

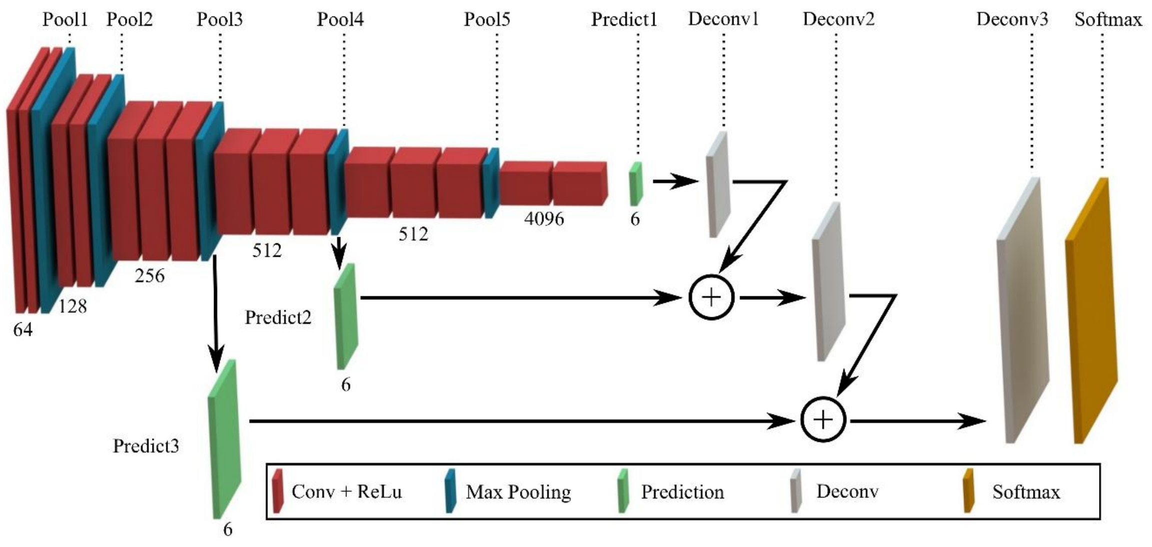
l) In the training process, all the values are printed to know the status of training process like, accuracy, loss, etc.

m) Lastly for every 2000 epochs, a fresh model file is written for testing purpose i.e. how trained model behaves.

CNN Model Used:

According to the paper, LenenNet uses ENet i.e. encoder decoder network. ENet originally have five stages in which LaneNet uses only the three stages for encoder. The encoder actually works on VGG16 encoder which has multiple layers. The encoder network removed the fully connected layer and placed the decoder layer that is used for binary label and instance label.

Detailed explanation of encoding and decoding layer is mentioned below:-



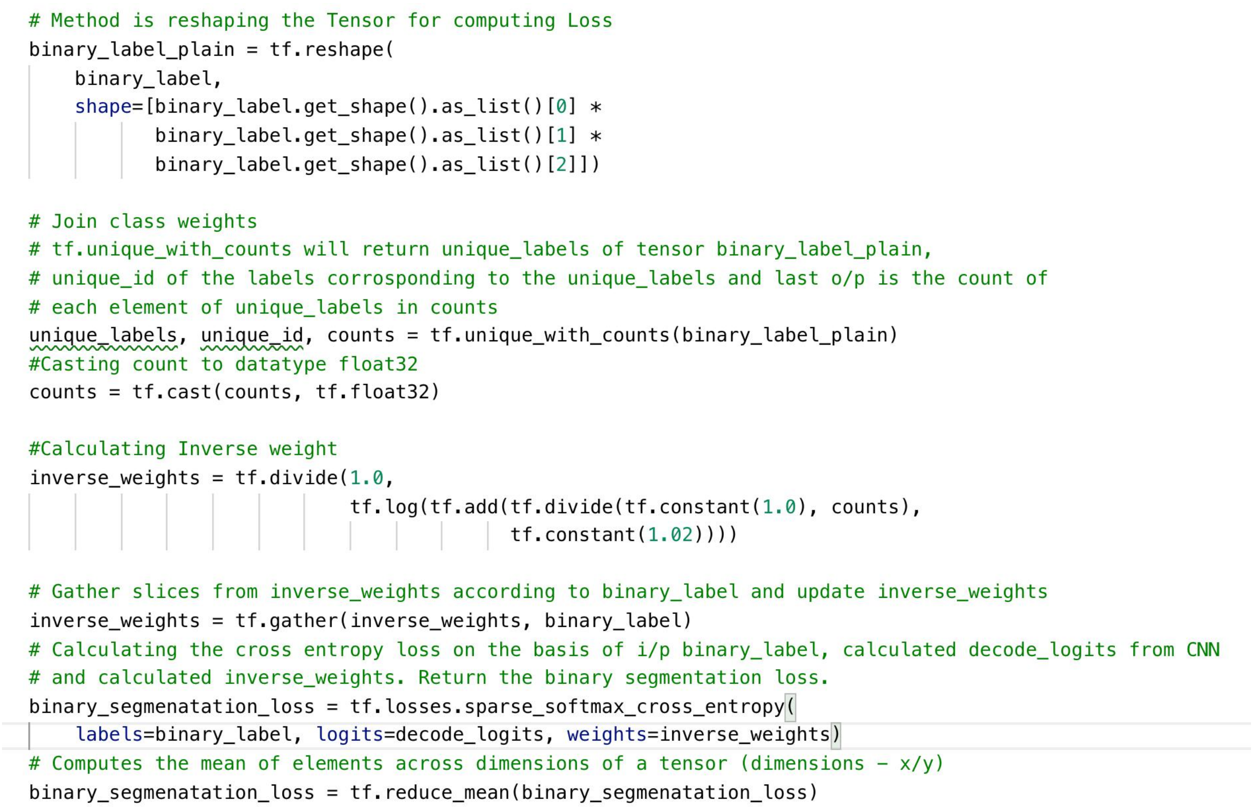
The main purpose of encoding is basically to extract features from the image by applying multiple convolutional layers. The encoding is then followed by a decoding process using a series of upsampling layers. Upsampling is performed using transposed convolution called as "**Deconvolution**". This is an operation that goes in the opposite direction to a convolution and allows us to translate the activations into something meaningful related to the image size by scaling up the activation size to the same image size. As in the process, we lose some resolution because the activations were downscaled and therefore to add back some resolution by adding activations from the previous layer called as skip connections.

Loss Calculation:

Loss is basically used for measuring the inconsistencies between the actual value and the predicted values. In the overall training, the main aim is to minimize the loss and increase the accuracy. In this scenario we are calculating two losses mentioned below:-

a) Binary Segmentation Loss :-

As per the paper, the binary segmentation network is trained with the standard cross-entropy loss function. Since the two classes (lane/background) are highly unbalanced, we apply bounded inverse class weighting. Code Snippet mentioned below:-



```
# Method is reshaping the Tensor for computing Loss
binary_label_plain = tf.reshape(
    binary_label,
    shape=[binary_label.get_shape().as_list()[0] *
           binary_label.get_shape().as_list()[1] *
           binary_label.get_shape().as_list()[2]])

# Join class weights
# tf.unique_with_counts will return unique_labels of tensor binary_label_plain,
# unique_id of the labels corrosponding to the unique_labels and last o/p is the count of
# each element of unique_labels in counts
unique_labels, unique_id, counts = tf.unique_with_counts(binary_label_plain)
#Casting count to datatype float32
counts = tf.cast(counts, tf.float32)

#Calculating Inverse weight
inverse_weights = tf.divide(1.0,
                            tf.log(tf.add(tf.divide(tf.constant(1.0), counts),
                                          tf.constant(1.02))))

# Gather slices from inverse_weights according to binary_label and update inverse_weights
inverse_weights = tf.gather(inverse_weights, binary_label)
# Calculating the cross entropy loss on the basis of i/p binary_label, calculated decode_logits from CNN
# and calculated inverse_weights. Return the binary segmentation loss.
binary_segmenatation_loss = tf.losses.sparse_softmax_cross_entropy(
    labels=binary_label, logits=decode_logits, weights=inverse_weights)
# Computes the mean of elements across dimensions of a tensor (dimensions - x/y)
binary_segmenatation_loss = tf.reduce_mean(binary_segmenatation_loss)
```

b) Instance Segmentation Loss (Discriminative Loss) :-

As per the paper, instance segmentation loss uses discriminative loss method referenced from paper “Semantic Instance Segmentation with a Discriminative Loss Function”. The code snippet for the discriminative loss function is written in “discriminative_loss.py”.

In the discriminative loss function, **C** is the number of clusters in the ground truth, **Nc** is the number of elements in cluster c, xi is an embedding, **μc** is the mean embedding of cluster c (the cluster center), ‖·‖ is the L1 or L2 distance, and [x]+ = max(0, x) denotes the hinge. **δv** and **δd** are respectively the margins for the variance and distance loss. The loss can then be written as follows:

$$L_{var} = \frac{1}{C}\sum_{c=1}^{C}\frac{1}{N_c}\sum_{i=1}^{N_c}[\|\mu_c - x_i\| - \delta_{\mathrm{v}}]_+^2$$

$$L_{dist} = \frac{1}{C(C-1)}\sum_{\substack{c_A=1\\c_A\neq c_B}}^{C}\sum_{c_B=1}^{C}[2\delta_{\mathrm{d}} - \|\mu_{c_A} - \mu_{c_B}\|]_+^2$$

$$L_{reg} = \frac{1}{C}\sum_{c=1}^{C}\|\mu_c\|$$

$$L = \alpha \cdot L_{var} + \beta \cdot L_{dist} + \gamma \cdot L_{reg}$$

Code Snippet mentioned below:-

```
#### Calculate the Instance partition loss using discriminative loss function ####
# Calculate the discriminative loss function
# Fetching the instance image results from CNN result array
decode_deconv = inference_ret['deconv']
# Pixel embedding
#Applying Convolution layer  followed by activation layer ReLU
pix_embedding = self.conv2d(inputdata=decode_deconv, out_channel=4, kernel_size=1,
                            use_bias=False, name='pix_embedding_conv')
pix_embedding = self.relu(inputdata=pix_embedding, name='pix_embedding_relu')
# Calculate discriminative loss
# Preparing Image Shape after applying conv and relu activation layer.
image_shape = (pix_embedding.get_shape().as_list()[1], pix_embedding.get_shape().as_list()[2])
#Invoking discriminative loss function which returns variance term, distance
# term and regularization term
disc_loss, l_var, l_dist, l_reg = \
    discriminative_loss.discriminative_loss(
        pix_embedding, instance_label, 4, image_shape, 0.5, 3.0, 1.0, 1.0, 0.001)
```

The overall discriminative loss function implementation was done with the help of below URL:-
https://github.com/hq-jiang/instance-segmentation-with-discriminative-loss-tensorflow/blob/master/loss.py

c) Calculating Total Loss :

Lastly the total loss is calculated that is basically used by the Momentum optimizer in our case to optimize the learning by minimizing the overall loss. Below Logic is used:-

```
# Consolidation loss
# Calculating Total Loss  i.e binary seg. loss + disciminative loss and
# regularization loss
l2_reg_loss = tf.constant(0.0, tf.float32)
for vv in tf.trainable_variables():
    if 'bn' in vv.name:
        continue
    else:
        l2_reg_loss = tf.add(l2_reg_loss, tf.nn.l2_loss(vv))
l2_reg_loss *= 0.001
total_loss = 0.5 * binary_segmenatation_loss + 0.5 * disc_loss + l2_reg_loss
```

3. Testing Trained Model:

Once the training is successfully done for the model, the model needs to be tested on some testing data. So, the main purpose of the testing module is to test the trained model and predict lanes on what it has been trained using CNN. For Testing, the program has two options, one is for testing individual image and the other one is for testing batch of images. "**test_model.py**" is used to perform testing of the model.

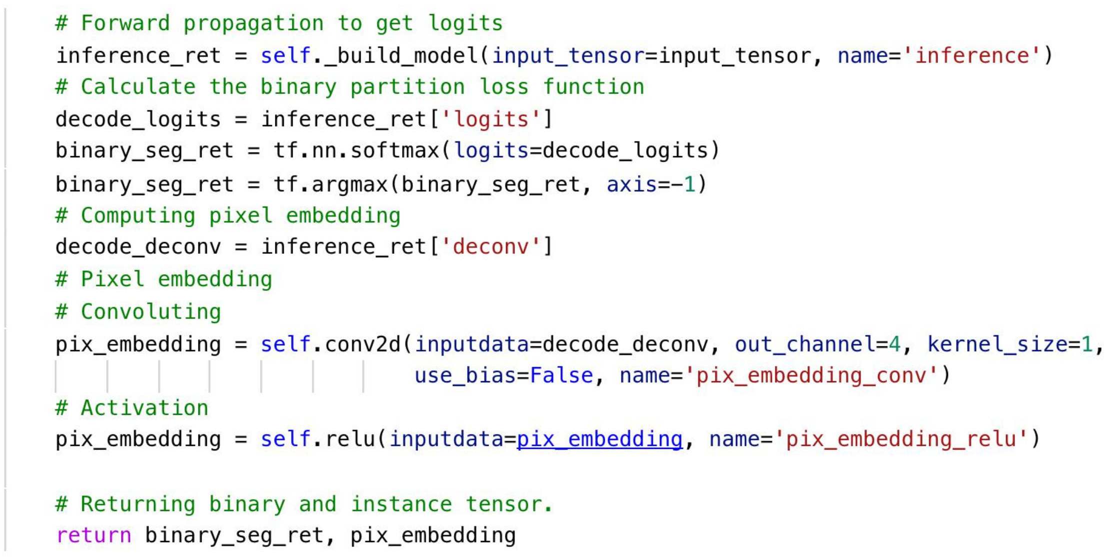
Steps involved in testing the trained module are mentioned below for single image: -

a) Initially we read the image that was passed through the CLI and process so that it can be passed in the tensor.

b) Next, we create input tensor which is further used for storing model information.

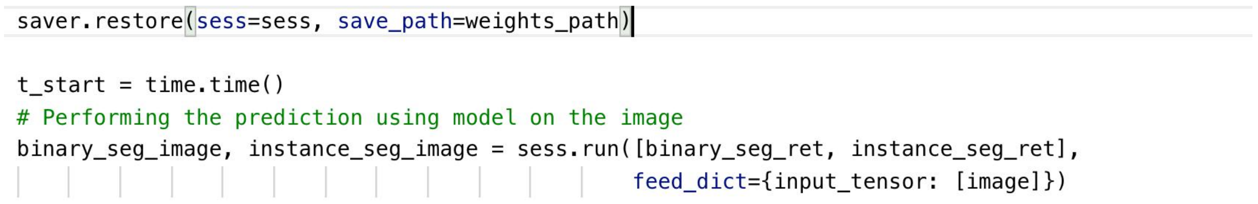
```
input_tensor = tf.placeholder(dtype=tf.float32, shape=[1, 256, 512, 3],
name='input_tensor')
```

c) Further, we will fetch binary and instance model that will be used for prediction by providing it to tensor. The code snippet mentioned below is available in "**merge_model.py**".



```
    # Forward propagation to get logits
    inference_ret = self._build_model(input_tensor=input_tensor, name='inference')
    # Calculate the binary partition loss function
    decode_logits = inference_ret['logits']
    binary_seg_ret = tf.nn.softmax(logits=decode_logits)
    binary_seg_ret = tf.argmax(binary_seg_ret, axis=-1)
    # Computing pixel embedding
    decode_deconv = inference_ret['deconv']
    # Pixel embedding
    # Convoluting
    pix_embedding = self.conv2d(inputdata=decode_deconv, out_channel=4, kernel_size=1,
                                use_bias=False, name='pix_embedding_conv')
    # Activation
    pix_embedding = self.relu(inputdata=pix_embedding, name='pix_embedding_relu')

    # Returning binary and instance tensor.
    return binary_seg_ret, pix_embedding
```

d) Next the model will be restored using "saver.restore( )" and further prediction will be done by passing the model tensor and image that needs to be predicted.



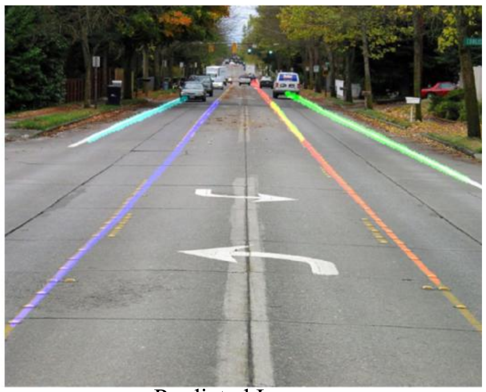
```
saver.restore(sess=sess, save_path=weights_path)

t_start = time.time()
# Performing the prediction using model on the image
binary_seg_image, instance_seg_image = sess.run([binary_seg_ret, instance_seg_ret],
                                                feed_dict={input_tensor: [image]})
```

e) The prediction returns the “Binary Segmentation Image” and “Instance Segmentation Image”. These images are passed to another method where noises are removed which is implemented in “**postprocess.py**”.

f) Next the processed images are passed to the clustering method which is implemented using mean shift clustering. The clustering method used to draw lanes on the coordinates that were obtained after prediction. The line is drawn using polylines.

g) Lastly the lanes returned from lane fitting are plotted over the original image and displayed.

Original Image


Predicted Image

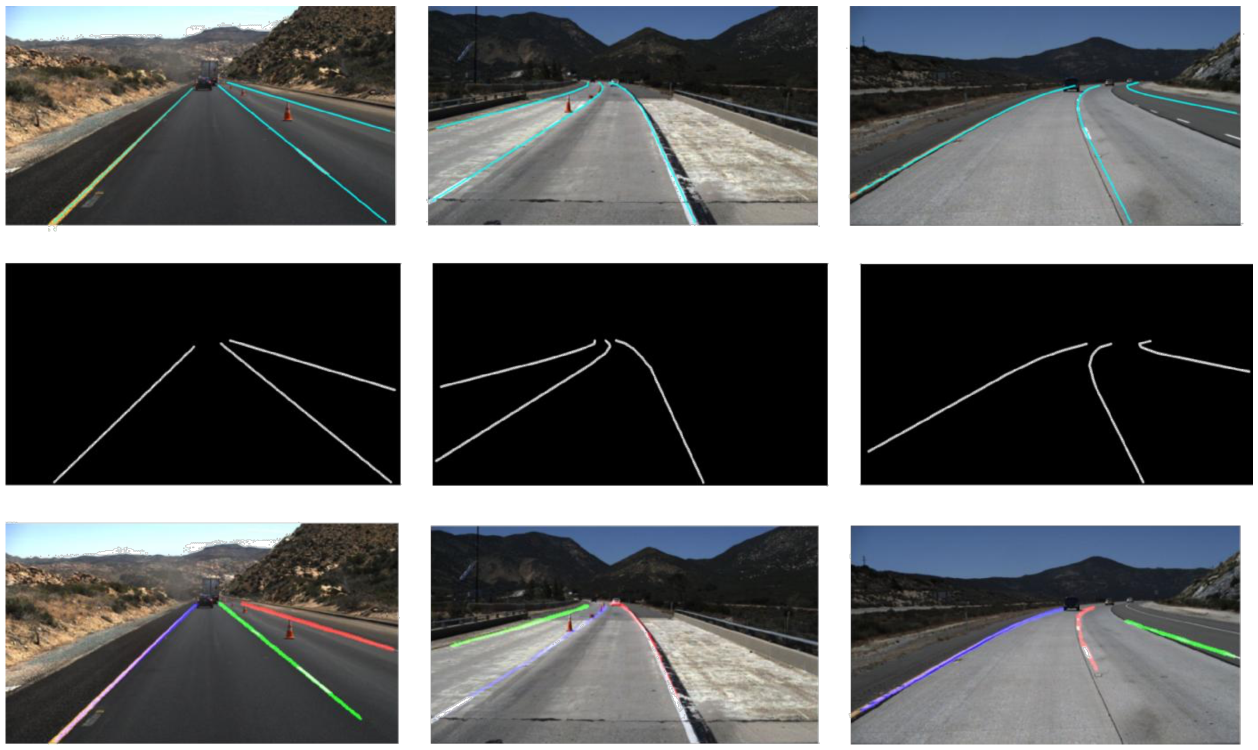
h) In case when the prediction is performed in the batch mode, the program calculates the no of files in the directory and the bath size that have been provided through CLI. It calculates the no of epochs that it needs to perform to process all image using (No of Images / batch size) and generate the processed image in the directory passed through CLI as “saved_dir”.

RESULTS

Training was performed on TuSimple dataset with 3626 training image and 2782 testing images. Training was done for 10,000 epochs which took around 24 hours approx. to complete the full training. Initially, in the beginning, the training accuracy dropped to 500 epochs but after that, it began to increase, and we got an accuracy of 99 % approx. The batch size for training and validation set is set to 8. On taking a higher batch size, the Google Colab throws an error because of memory constraint. As the original image size for the data set is 1280 x 720, so it takes a lot of time for processing and generally throws memory error.

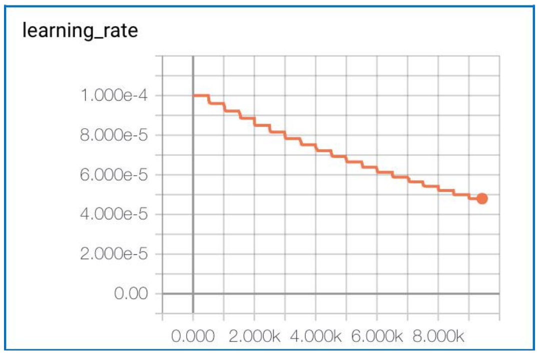
```
---------------FINAL RESULTS-----------------
Results after last Epochs
Train binary loss:      0.03223518
Train instance loss:    0.10611932
Train accuracy:         0.9952417771962172
Total training loss     1.1915557
Test binary loss:       22.687647
Test instance loss:     2.4224758
Test accuracy:          0.5073552263545011
Total test loss         13.67744

---------------------------------------------
```

On performing the testing on the model trained for 10000 epochs, we get the results shown below. The first row of the figure shows the ground truth plotted on the image, the second row shows the binary image and the third row shows the actual predicted results.

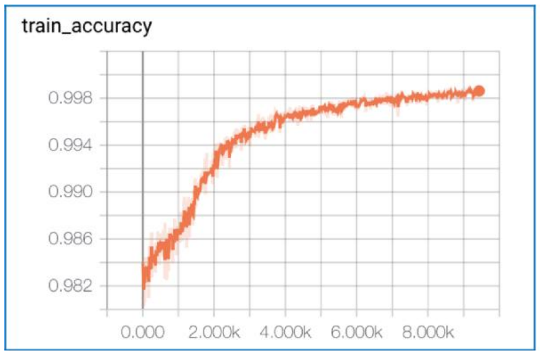


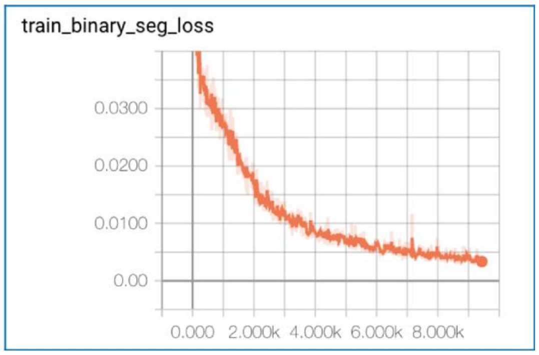
Graphical representation of the learning rate decay, accuracy, loss for the binary and instance branch explained below: -

The graph on the right side shows the learning rate decay for 10000 epochs with decay of 0.1 on every 500th epochs. The training starts from learning rate 0.0001 and start decaying over the training period.

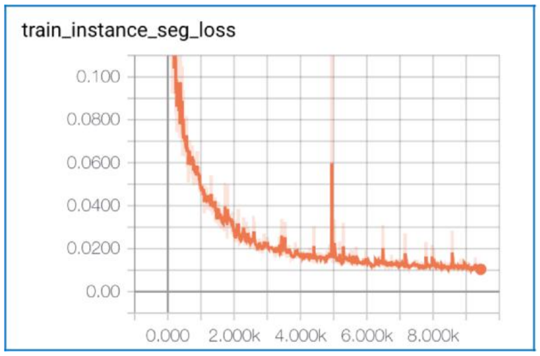


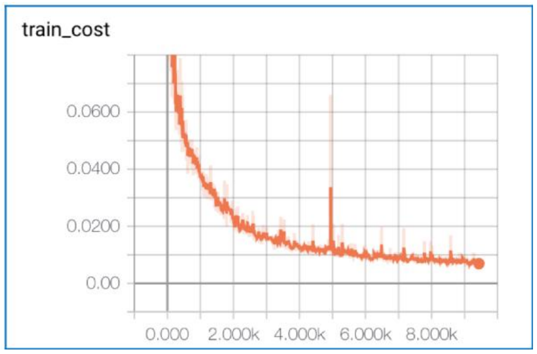
The below graph shows the training accuracy and tarining binary segmentation loss.



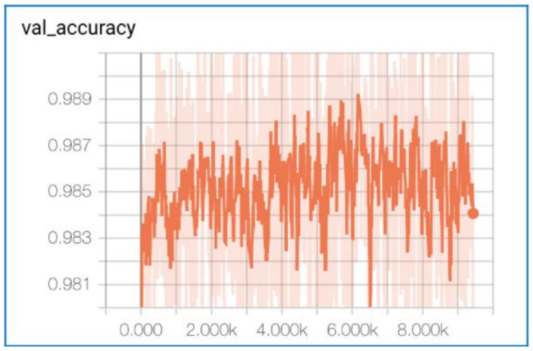


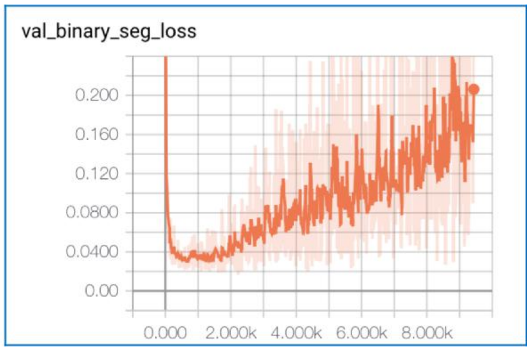
Below graph shows the training instance segmentation loss and training cost (totat training loss).



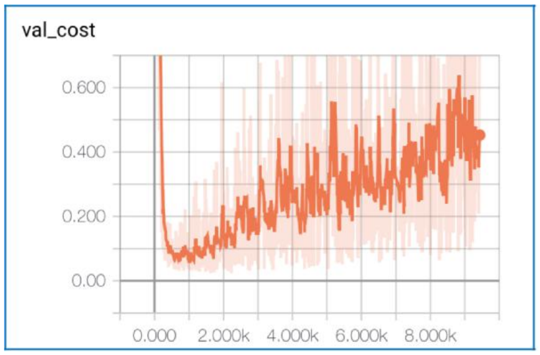


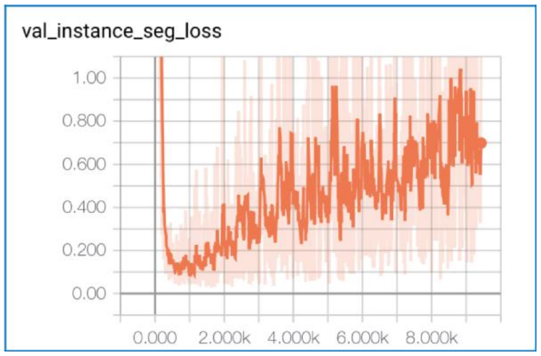
Below graph shows the value accuracy and the value binary segmentation loss over the validation process.





Below graph shows the value cost (total value loss) and the value instance segmentation loss over the validation process.

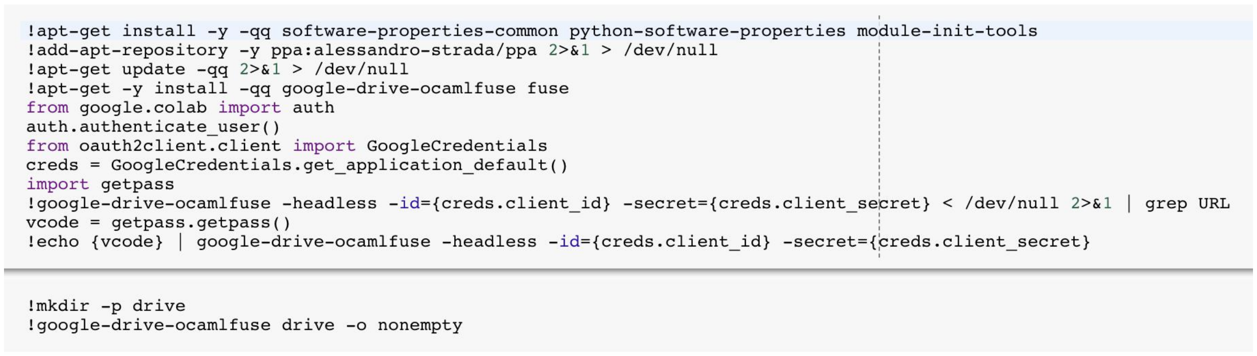




EXECUTION PROCESS

As the project is all about training the convolutional neural network, we were not having machines with GPU and thus we used Google Colab with google drive for the training the model. To execute the program that have been uploaded with the report, below steps needs to be followed:-

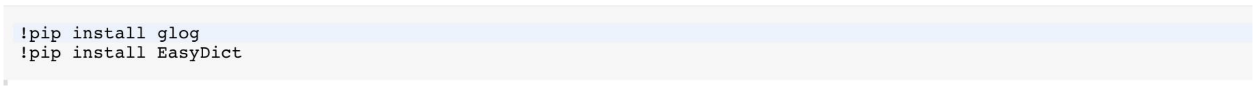
- Initially we enabled tensorboard to view all the learning progress.
- Next, we will create an apps folder in the google drive root directory.
- Connect the google drive apps folder with the Google Colab.
- Copy all the files from Google drive to the Google Colab temp folder.



```
!apt-get install -y -qq software-properties-common python-software-properties module-init-tools
!add-apt-repository -y ppa:alessandro-strada/ppa 2>&1 > /dev/null
!apt-get update -qq 2>&1 > /dev/null
!apt-get -y install -qq google-drive-ocamlfuse fuse
from google.colab import auth
auth.authenticate_user()
from oauth2client.client import GoogleCredentials
creds = GoogleCredentials.get_application_default()
import getpass
!google-drive-ocamlfuse -headless -id={creds.client_id} -secret={creds.client_secret} < /dev/null 2>&1 | grep URL
vcode = getpass.getpass()
!echo {vcode} | google-drive-ocamlfuse -headless -id={creds.client_id} -secret={creds.client_secret}
```

```
!mkdir -p drive
!google-drive-ocamlfuse drive -o nonempty
```

- Install below dependencies



```
!pip install glog
!pip install EasyDict
```

- For training module, we use below command when some pretrained weights are available :-

  !python drive/apps/train_model.py --net vgg --dataset_dir drive/apps/data/training_data/ --weights_path drive/apps/model/10000/lanenet_vgg_2018-11-18-19-57-18.ckpt-0

  When no pretrained weight file is available, just use below commant to train the model from

  scratch, !python drive/apps/train_model.py --net vgg --dataset_dir drive/apps/data/training_data/

- Once the model is trained and saved, the model can be tested for prediction on local machine. The testing canbe done in two modes. First one is single image mode using below command,

  python test_model.py --is_batch False --batch_size 1 --weights_path model/10000/lanenet_vgg_2018-11-18-19-57-18.ckpt-0 --image_path drive/apps/data/testing_data/test_images/lane.jpg

  When the testing needs to be done in batches, below command needs to be used,

  python test_model.py --is_batch True --batch_size 2 --save_dir drive/apps/data/testing_data/test_images/ret --weights_path model/10000/lanenet_vgg_2018-11-18-19-57-18.ckpt-0 --image_path drive/apps/data/testing_data/test_images/

  After execution of the command, the output will be saved to --save_dir path provided.

  Project Structure:

drive/apps/data/ - Contains all the training and testing data sets. We have uploaded only 20 images sets on which both testing and training can be performed because of the size issue.

model/10000/ - Contains trained model file on which prediction can be done. If one wants to perform training, the training files will be generated in the model folder directly.

All the other project python code files are available in the root directory.

LIMITATIONS

One of the most important techniques that have allowed us to reach these levels of performance is convolutional layer factorization in ENet. However, we have found one surprising drawback. Although applying this method allowed us to greatly reduce the number of floating-point operations and parameters, it also increased the number of individual kernels calls, making each of them smaller.

CONCLUSION

Overall, our system is fast but lags the state-of-art in terms of segmentation accuracy. In our project, we have presented a method for end-to-end lane detection at 50 fps. Inspired by ongoing instance segmentation techniques, our method can detect a variable number of lanes and can adapt to lane change moves, in contrast to other related deep learning approaches. To parameterize the segmented lanes using low order polynomials, we have trained a network to generate the parameters of a perspective transformation, conditioned on the image, in which lane fitting is optimal. This network is trained using a custom loss function for lane fitting. Our method is robust against ground plane's slope changes, by adapting the parameters for the transformation accordingly. Furthermore, we observe that jointly training tasks can potentially lead to increased performance.