VIP-IPA: Lane Detection Team

Final Report

Spring 2024

William Stevens, Vishal Urs, Karthik Selvaraj,

Gabriel Torres, and Gaurish Lakhanpal

CONTENTS

1. Abstract

With the prevalence of autonomous vehicles, the computer vision algorithms utilized for autonomous driving must be robust and accurate to assess road features through images captured in real time. In our previous works, we have implemented multi-task neural network models according to various architectures, such as YOLOv4 and DETR. These models, trained on the BDD100k dataset, produced detections and classifications of objects, as well as segmentations and classifications of lane lines, in an image. While these models were successful, they only highlighted relevant features on the images, which is not useful information to driving software. Thus, this semester we explored a new approach to lane segmentation that would produce a more useful output: an aerial representation of the road structure when given ground-level images. The design we have explored involves using an optimized neural network structure containing multiple transformers, inspired by the LaneSegNet and BEVFormer architectures. This process has involved the research, development, integration, and testing of new concepts, including Deformable Attention and Perspective Feature Transformation, in order to produce a model that can accurately and efficiently translate ground-level captures into an aerial view of the road features. This will provide autonomous driving software with more useful information and context, such as the position and direction of road lanes relative to the vehicle. This new structure will allow our model to accurately perform the important vision tasks for autonomous driving at a low computation cost, while producing information relevant to the autonomous driving pipelines used in the real world.

2. Introduction

The goal of this semester was to expand our computer vision algorithm output to the next level. The Fall 2023 implementation was successful in its development of a DETR architecture that was able to perform object detection with promising accuracy. However, this output is not of much use to autonomous driving software, which must make real-time driving decisions based on a complete semantic understanding of the road surroundings. Therefore, the main objective of the team this semester was to research and implement a new framework that would take dashboard images and transform them into useful semantic information.

3. DATASET

We are using the OpenLane-V2 dataset [1] for our project this semester because it contains 2,000 road scenarios in the form of 15-second videos, with seven camera angles in a 360^o span around the car (See Fig. 1). For each of these videos, OpenLane-V2 contains 36 frames, each with a groundtruth annotation representative of what a satellite view of the road scenario looks like. This Bird's-Eye-View (BEV) perspective will be used as a measurement for how accurate the model's output is, as it will be what the model is learning to replicate. The videos in OpenLane-V2 are taken from around the world, with multiple weather and lighting conditions. The diversity of geography, environment, and weather allows models trained on the dataset to be robust and able to generalize its predictions when encountering new environments or changes in seasons. In the annotations themselves, the groundtruth BEV contain three classes: centerlines, lane boundaries, and crosswalks. See Fig. 2 for an example. The orange centerlines represent the center of a given lane path, with arrow markings indicating the legal direction cars must take. The blue lane boundaries represent the delimiters separating lanes from each other or the curb, with dashed or dotted patterns to demonstrate the type of lane line. The yellow crosswalk lines represent the outer borders of crosswalks on the road. Being able to determine such information about the important boundaries and features of lanes in a road scenario is very valuable to autonomous driving software, which is why the annotations in OpenLane-V2 are useful to the purpose of our project this semester. The dataset's large size, wide variety, and highly detailed annotations make it a great fit for the success of our training purposes.

Fig. 1: OpenLane-V2 Surrounding View Images Example. [1]

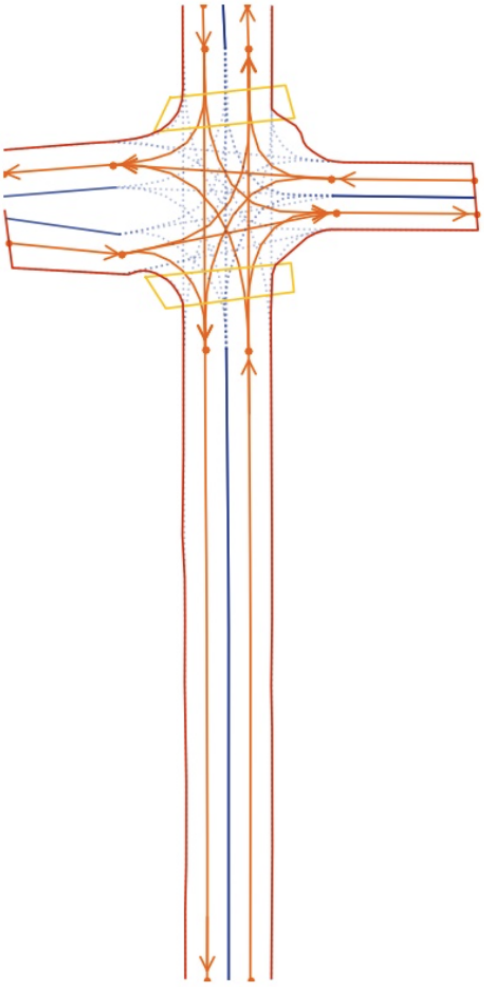

Fig. 2: BEV Annotations Example. [1]

4. NEURAL NETWORK FUNDAMENTALS

A. *Introduction*

Neural networks are the core foundation of most machine learning techniques. The concept of a neural network takes inspiration from the human brain's means of processing information through biological neurons. A neural network is a digital attempt at replicating a human brain — it is a framework containing hundreds, thousands, or even millions of nodes that are densely interconnected. This network of interconnected nodes is fed an input which is processed through all nodes of the network to provide an output.

Each individual node can be perceived as a linear regression expression, composing of a weight and a bias:

$$z = ax + b \tag{1}$$

where x is the input value, a is the node's weight, b is the node's bias, and z is the output of the linear regression. The combination of numerous such nodes defines an individual layer in a neural network. The output from each node is added towards a combined sum that serves as the input to the next layer. The process of using the output of each layer as the input for the next layer is termed forward propagation. A single-layered neural network with a vector of weights and biases can be visualized through Fig. 3.

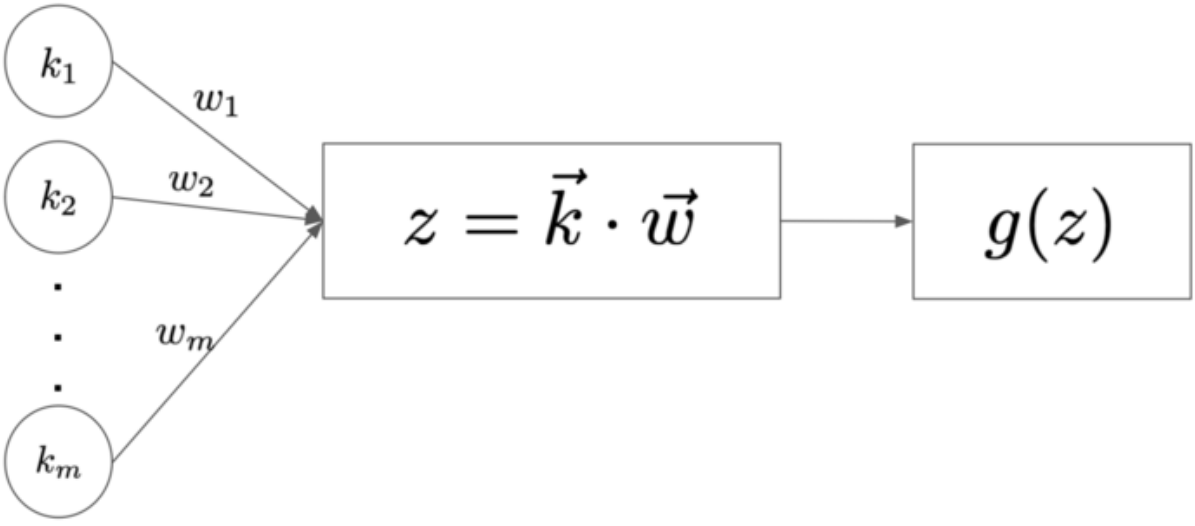

Fig. 3: Single Layer Perceptron

where $\vec{k}$ is the input vector, $\vec{w}$ is the vector of weights for m nodes. After the weights and bias have been applied on the input $\vec{k}$, the sum of the outputs z is passed through an activation function, depicted as $g(z)$ in Fig. 3. The activation function is an essential part of neural network that introduces non-linearity to the model. Without an activation function, a neural network would simply be a linear regression model. There are several activation functions, each suited better for a particular purpose. The application of weights, biases, and activation to the input defines a singular layer of a neural network. This process is performed in every layer, continually adding complexity to a multi-layered network.

B. The Activation Function

As mentioned previously, an activation function is a function that is applied to the output of a perceptron. Typically, nonlinear activation functions are used in multi-layered networks; the purpose of doing this is to introduce non-linearity to the output of each perceptron. This allows the network to approximate functions that may not be linear. This is important because most real-world data is rarely linear. Below, we have described several widely used activation functions:

Sigmoid:

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \tag{2}$$

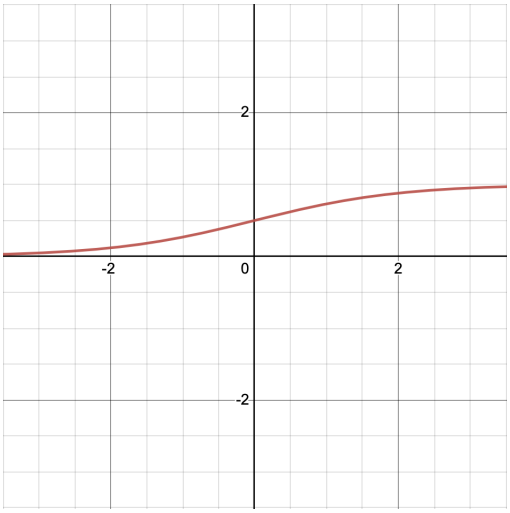

Fig. 4: Sigmoid Function

The Sigmoid function is a popular nonlinear activation function. As depicted in Fig. 4, the range of values produced by this function is between 0 and 1, which can prove to be useful if we would like our network to output a probability. However, the Sigmoid function is prone to producing gradients close to 0. This is because the function is bounded above and below, causing the gradient to approach 0 when the function's input moves further away from 0. As a result, this can have adverse effects on the training process and the model performance.

Rectified Linear Unit (ReLU):

$$ReLU(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases} \tag{3}$$

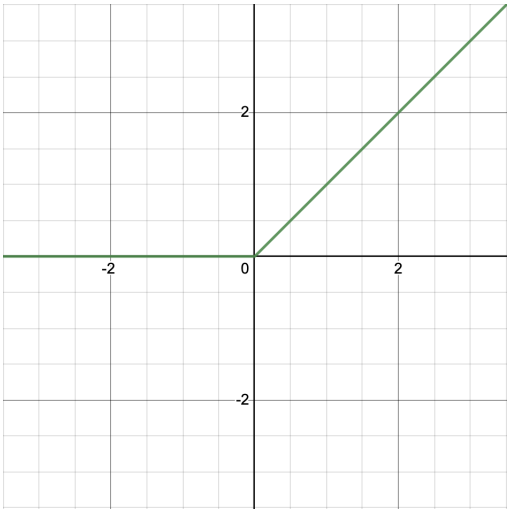

Fig. 5: ReLU Function

Illustrated in Fig. 5, the ReLU function is an activation function that outputs 0 when given a negative input and returns the input otherwise. The ReLU activation function is used due to its effectiveness in networks performing computer vision tasks. An issue with the ReLU function is that it is not differentiable at the origin. This means the calculation of the gradient of ReLU requires computationally expensive approximations. Additionally, since ReLU outputs 0 for any negative inputs, this will result in gradients of 0, as well. This results in what is known as the

dying ReLU problem. This is when the input of the activation function is negative, meaning the output produced by the perceptron is 0 with a gradient of 0; as a result, the perceptron will always produce 0 and will not change due to the gradient being 0.

Leaky Rectified Linear Unit (Leaky ReLU):

$$LeakyReLU(x) = \begin{cases} x, & x > 0 \\ \alpha x, & x \leq 0, \text{ where } \alpha < 1 \end{cases} \tag{4}$$

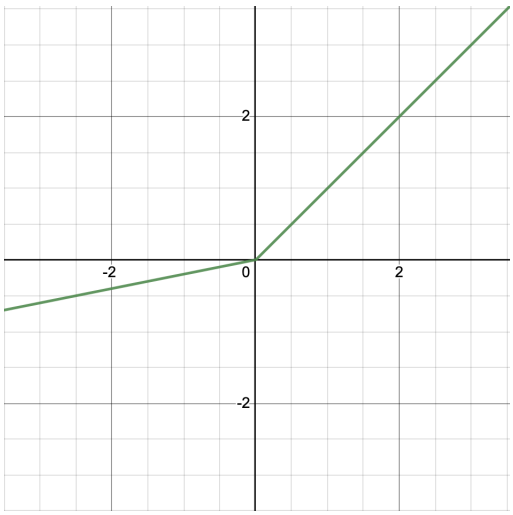

Fig. 6: Leaky ReLU Function

The Leaky ReLU function, shown in Fig. 6, is a variation of ReLU and is also a popular activation function used in computer vision tasks. Similar to ReLU, the gradient calculation of Leaky ReLU requires computationally expensive approximations since it is also not differentiable at the origin. However, Leaky ReLU solves the dying ReLU problem by allowing nonzero outputs for negative inputs. Therefore, Leaky ReLU can prove to be a more appropriate activation function when inputs are negative.

C. Loss

A loss function is a function that is used to quantify the error between the predicted output of a neural network and the ground truth. This loss value is used to train a neural network

to artificially "learn" from its mistakes. Thus, it is important to minimize the loss function to improve model accuracy and produce the best possible results. In other words, the optimal value of θ should satisfy the following:

$$\hat{\theta} = \arg\min_{\theta} L_{\mathbf{x},\,\mathbf{y}}(\theta) \tag{5}$$

Illustrated in Eq. 6 is an example of the loss function mean-absolute-error (MAE):

$$L_{\mathbf{x},\,\mathbf{y}}(\theta) = \frac{1}{N}\sum_{n=0}^{N} |y^n - f_\theta(x^n)| \tag{6}$$

Where θ is the set of tunable parameters, N is the number of output nodes (or the dimensions of the output vector), $f_\theta(x^n)$ is the output produced by the neural network given the input x^n, and y^n is the groundtruth output. MAE is a basic loss function that can be used to calculate the loss. However, since this function is not differentiable at the origin, this means additional expensive computations are required to approximate the derivative at the origin.

Another example of a loss function is the mean-squared-error (MSE) loss shown in Eq. 7:

$$L_{\mathbf{x},\,\mathbf{y}}(\theta) = \frac{1}{N}\sum_{n=0}^{N} |y^n - f_\theta(x^n)|^2 \tag{7}$$

MSE is another basic loss function but, unlike MAE, is continuously differentiable. Despite this, some issues arise with the MSE loss function. Due to the quadratic nature of MSE, the loss function produces higher gradients for losses further from 0. However, MSE also produces lower gradients for losses closer to 0. Therefore, this can result in difficulties in fine-tuning the weights of the neural network.

D. Gradient Descent

Gradient descent is an algorithm used to minimize loss. When we construct a neural network, our primary goal is to achieve an accurate output. In order to improve accuracy, our calculated loss should be minimal. As described in the previous section, the loss function uses the weights, biases, and training images to compute a particular cost or loss. However, the computed cost can be reduced through the process of gradient descent. The term gradient defines the steepness

or slope of a curve; the direction of ascent or descent of a curve. The term descent represents the action of moving downwards. Thereby, we can summarize gradient descent as the process of "moving down along a curve."

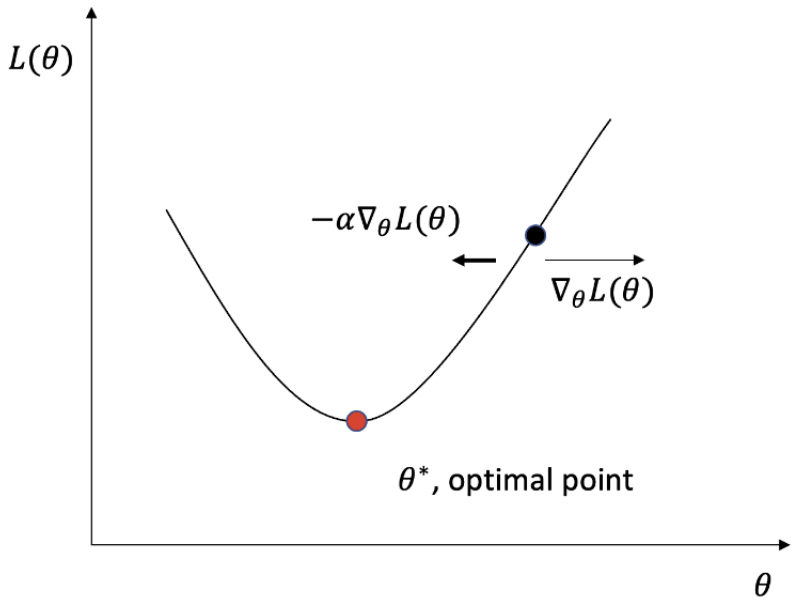

Fig. 7: Gradient Descent Visualized

Fig. 7 illustrates the loss function with respect to the learnable parameter θ (assume we only have one parameter for this 2D example). Let the starting value of θ correspond to the black point on the graph. The goal is to reach the minimum point of the loss function: the red point. When at the initial point, we do not know the global structure of the function. However, we can compute the local parameters, such as the gradient, to try and reach the minimum point. This can be achieved because the gradient can be used to find the direction of steepest descent, which will eventually lead towards the minimum point. Making small steps towards the minimum of the curve will eventually reach the optimum point. Formally, the gradient descent algorithm can be summarized as:

1) Find gradient $\nabla_\theta L(\theta)$
2) Assign $\theta := \theta - \gamma\nabla_\theta L(\theta)$

3) Repeat until convergence

Fig. 7 demonstrates the concept in a 2D setting. However, practical applications often involve multi-dimensional loss functions. Using the algorithm in the form represented above would not provide effective results. Optimized versions of gradient descent such as Momentum, Adaptive Momentum (ADAM), and Adaptive Gradient (AdaGrad) can be used to obtain better results.

E. Backpropagation

In order to perform gradient descent and update weights, we will need to obtain the gradient of each weight in the network, which will depend on all of its subsequent connected weights. Thus, computing the partial derivative of each weight is very computationally expensive due to complexity of neural networks, especially deep neural networks. This computation is called backpropagation, due to the backward nature of the chain rule computation when calculating partial derivatives. For future reference, Fig. 8 displays an example of a deep neural network.

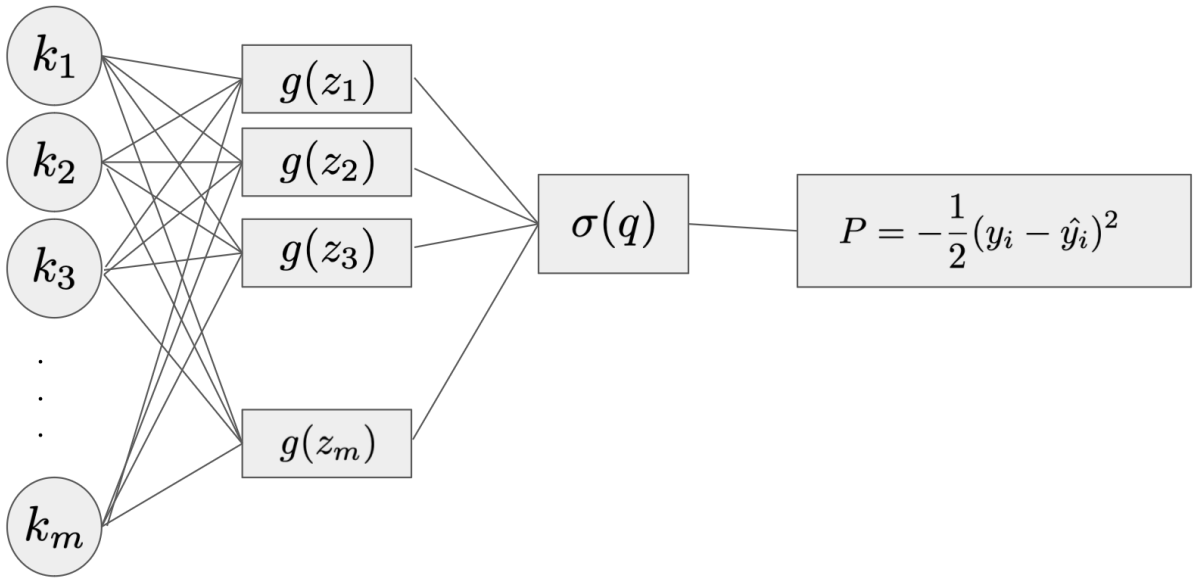

Fig. 8: Deep Neural Network.

To lay out the structure of this deep neural network let us define the overall architecture. The first layer can be mathematically represented as such:

$$\vec{z} = W\vec{k} \tag{8}$$

where W is the weight matrix, $\vec{k}$ is the inputs, and $\vec{z}$ is the corresponding resultant vector. Subsequently, $\vec{z}$ is passed through the activation function, $g(z_i)$, element by element $(z_1..z_m)$.

The second layer is represented as such:

$$\begin{aligned} \vec{b} &= g(\vec{z}) \\ q &= \vec{a} \cdot \vec{b} \end{aligned} \tag{9}$$

where $\vec{a}$ is a second set of weights. However in this case the weights are represented as a vector instead of a matrix to get a scalar output, q. With the output, q is then run through another activation function,(σ), where the result will be evaluated by the performance function.

$$\begin{aligned} \hat{y}_i &= \sigma(q) \\ P &= -\frac{1}{2}(y_i - \hat{y}_i)^2 \end{aligned} \tag{10}$$

With the architecture defined, the next step is to find the partial derivative with respect to each weight in the neural network. In order to find the partial derivatuve, the chain rule is used. For example, let us find the partial derivative with respect to a_i for some weight in $\vec{a}$:

$$\frac{\partial P}{\partial a_i} = \frac{\partial P}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial q} \frac{\partial q}{\partial a_i} \tag{11}$$

If the partial derivatives are taken individually:

$$\begin{aligned} \frac{\partial P}{\partial \hat{y}_i} &= (y_i - \hat{y}_i) \\ \frac{\partial \hat{y}_i}{\partial q} &= \sigma^{'}(q) \end{aligned} \tag{12}$$

Since q is defined as a dot product the $\frac{\partial q}{\partial a_i}$ is as follows:

$$\begin{aligned} q = \vec{a} \cdot \vec{b} &= a_1 b_1 + a_2 b_2 + ...a_i b_i ... + a_m b_m \\ \frac{\partial}{\partial a_i}(q = \vec{a} \cdot \vec{b} &= a_1 b_1 + a_2 b_2 + ...a_i b_i ... + a_m b_m) \end{aligned} \tag{13}$$

$$\frac{\partial q}{\partial a_i} = b_i$$

Putting it all together:

$$\frac{\partial P}{\partial a_i} = (y_i - \hat{y_i})\sigma^{'}(q)b_i \tag{14}$$

With the partial derivative of the second set of weights known, let us move onto the first set of weights. Assume that we want to find the partial derivative with respect to w_{ij} for some weight in matrix W. The partial derivative would be:

$$\frac{\partial P}{\partial w_{ij}} = \frac{\partial P}{\partial \hat{y_i}} \frac{\partial \hat{y_i}}{\partial q} \frac{\partial q}{\partial \vec{b}} \frac{\partial \vec{b}}{\partial \vec{z}} \frac{\partial \vec{z}}{\partial w_{ij}} \tag{15}$$

Taking the partials individually:

$$\begin{aligned} \frac{\partial P}{\partial \hat{y_i}} &= (y_i - \hat{y_i}) \\ \frac{\partial \hat{y_i}}{\partial q} &= \sigma^{'}(q) \end{aligned} \tag{16}$$

However, we run into an issue taking the partial derivative, $\frac{\partial q}{\partial \vec{b}}$. After all, what is the partial derivative of something with respect to a vector? So we have to change the partial derivative to a scalar value. Since we know that the w_{ij} is in the ith row of the matrix we take the partial derivative of the element in the ith row of $\vec{b}$, b_i.

$$\begin{aligned} q = \vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2 + ...a_i b_i ... + a_m b_m \\ \frac{\partial}{\partial b_i}(q = \vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2 + ...a_i b_i ... + a_m b_m) \\ \frac{\partial q}{\partial b_i} = a_i \end{aligned} \tag{17}$$

Since we defined each element in $\vec{b}$ as the result of applying the activation function (g) on each element in $\vec{z}$ then:

$$\begin{aligned} b_i &= g(z_i) \\ \frac{\partial b_i}{\partial z_i} &= g^{'}(z_i) \end{aligned} \tag{18}$$

For the last partial, since the we defined $\vec{z}$ as the product of matrix multiplication then:

$$\vec{z} = W\vec{k}$$

$$\vec{z} = \begin{bmatrix} — & \vec{w_1} & — \\ — & \vec{w_2} & — \\ & \vdots & \\ — & \vec{w_i} & — \\ & \vdots & \\ — & \vec{w_m} & — \end{bmatrix} \begin{bmatrix} | \\ \vec{k} \\ | \end{bmatrix} = \begin{bmatrix} \vec{w_1} \cdot \vec{k} \\ \vec{w_2} \cdot \vec{k} \\ \vdots \\ \vec{w_i} \cdot \vec{k} \\ \vdots \\ \vec{w_m} \cdot \vec{k} \end{bmatrix} \tag{19}$$

$$z_i = \vec{w_i} \cdot \vec{k} = w_{i1}k_1 + w_{i2}k_2 + ...w_{ij}k_j... + w_{im}k_m$$

$$\frac{\partial}{\partial w_{ij}}(z_i = \vec{w_i} \cdot \vec{k} = w_{i1}k_1 + w_{i2}k_2 + ...w_{ij}k_j... + w_{im}k_m)$$

$$\frac{\partial z_i}{\partial w_{ij}} = k_j$$

The updated partial derivative is:

$$\begin{aligned} \frac{\partial P}{\partial w_{ij}} &= \frac{\partial P}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial q}\frac{\partial q}{\partial b_i}\frac{\partial b_i}{\partial z_i}\frac{\partial z_i}{\partial w_{ij}} \\ \frac{\partial P}{\partial w_{ij}} &= (y_i - \hat{y}_i) * \sigma^{'}(q) * a_i * g^{'} * (z_i) * k_j \end{aligned} \tag{20}$$

Finding the partial derivative of this deep neural network was not too difficult. However, in a neural network that has several more layers the chain rule would become inevitably long. So the computation time would take much longer. However, if you notice that since we computed the second set weights first, we can reuse the partial derivatives, $\frac{\partial P}{\partial \hat{y}_i}$ and $\frac{\hat{y}_i}{\partial q}$. Although this would only work if we start differentiating at the output layer, or the back of the network, hence why it is called backpropagation. So in a neural network with several more layers, we can reuse portions of the chain rule we calculated in the layer before to significantly reduce the computational time it takes to train the architecture.

5. Convolutional Neural Networks and Residual Connections

A. *Introduction*

A Convolutional Neural Network (CNN) is a specific type of neural network that is often used in the field of image processing. In a CNN, a particular filter of dimension nxn is slid over an input image while an element-wise multiplication is performed in each stride of the filter. Ironically, this process of simply summing the outputs of an element-wise multiplication between the input image and the filter is not convolution; the mathematically accurate term to describe such an operation is cross-correlation. The filter would need to be rotated 180 degrees for the process to be considered a convolution. Although the layers of a CNN are called convolutional layers, the actual operation being performed is cross-correlation. This operation results in a feature map that highlights particular characteristics of the input image. Fig. 9 provides a high-level illustration of a CNN.

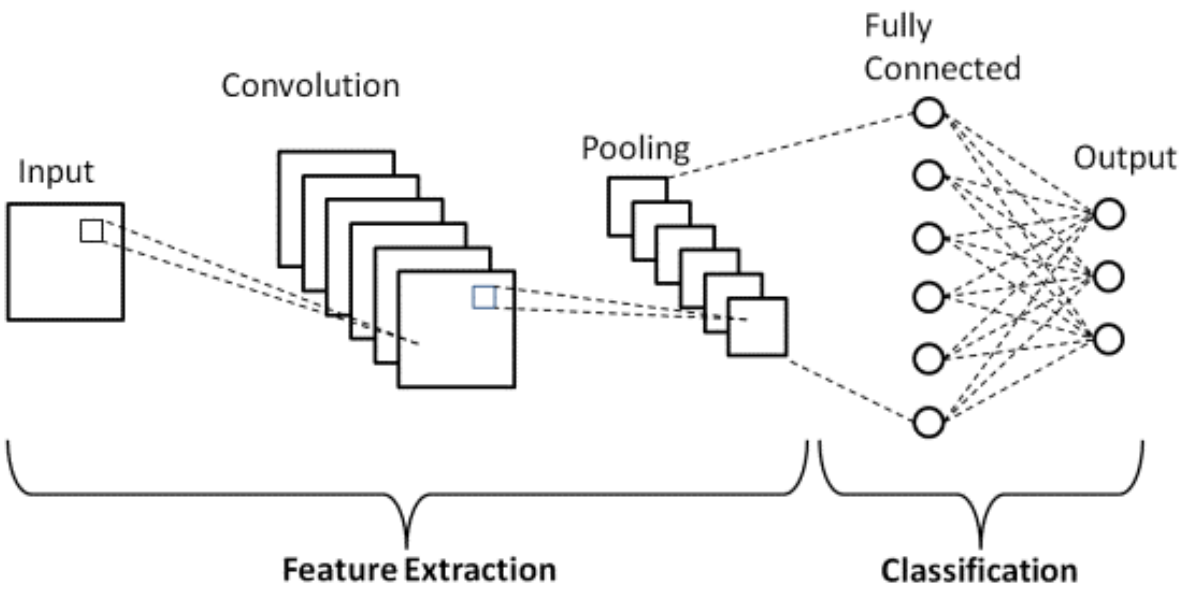

Fig. 9: Convolutional Neural Network (CNN)

The basic building blocks of a CNN include convolutional layers, pooling layers, a fully connected layer, and an output layer. The convolutional layers contain the activations and the

output layer contains the loss as discussed in the Neural Network section earlier. Convolutional layers perform the cross-correlation operation on the input image using a set of filters. The output of a convolutional layer is a set of feature maps; each filter outputs a corresponding feature map. Pooling layers are used to resize the feature maps, to reduce their dimensions. Pooling can be done in different ways; Max Pooling is a commonly used pooling technique that acquires the maximum of a local region in the feature map. Sliding a window across the feature map and extracting the maximum value of every window results in a smaller feature map. This process is done to reduce the computation in the CNN. Finally, fully connected layers perform a linear transformation on the flattened output of the previous layer, followed by a non-linear activation function. One of the key advantages of CNNs is their ability to learn hierarchical representations of an image. Lower-level filters in the network learn to detect simple features such as edges and corners, while higher-level filters learn to detect more complex features such as shapes and patterns. This allows the network to effectively capture the structure of the image and classify it correctly.

B. Convolution and Cross-Correlation

As mentioned previously, a CNN introduces the concept of sliding a filter with dimensions $n \times n$ over an input image, performing an element-wise multiplication, and summing the result. This process was given the name convolution in the field of computer vision, hence the name convolutional neural network. However, this process is not convolution but actually cross-correlation. Here is a diagram of how the cross-correlation operation is performed:

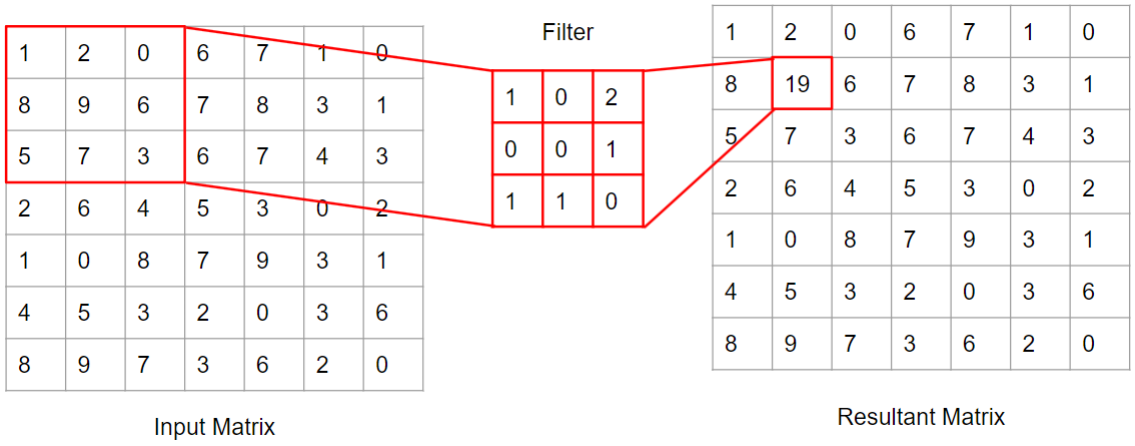

Fig. 10: Cross-Correlation Operation

This cross-correlation operation is what is performed in CNNs. The convolution operation is very similar, but the key difference is that the filter is flipped before the element-wise multiplication is performed. In other words, the filter is turned 180°, and then the cross-correlation operation is performed with this flipped filter. Below is a diagram of this process:

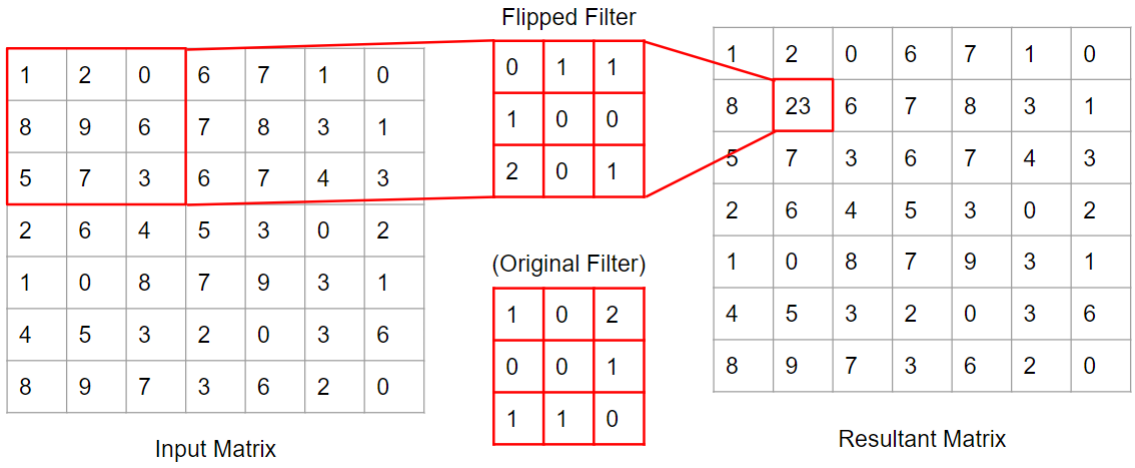

Fig. 11: Convolution Operation

Therefore, the use of the word convolution in CNNs is a misnomer as a cross-correlation operation is what is really being used. The reason cross-correlation is used in favor of convolution is because convolution requires extra computation to flip the filter. This can lead to longer training time and slower model performance.

C. Residual Connections and ResNet

In theory, convolutional neural networks should be more accurate as they get deeper, as their increased complexity should allow the capture and understanding of more complete concepts. However, it was found that models actually began to lose performance accuracy with depth. The concept of residual connections, first introduced in 2015 by "Deep Residual Learning for Image Recognition" [2], aimed to combat this issue by introducing skip connections in CNNs. The authors believed that the earlier layers in deep CNNs were converging on the correct signal, but that later layers were for some reason unable to learn the identity function and pass the signal forward, thus causing the performance issue. As a result, shallower networks had relatively higher performance in comparison to deeper models, because the later layers could not contribute anything meaningful to the network, and often impacted it negatively.

The core hypothesis behind the ResNet paper was that a deep neural network should perform better, otherwise the same, when compared to a shallower neural network [2]. Thus, the skip connections proposed in ResNet function by saving the input to a series of layers, passing it through those layers, and then taking the sum of the original input with the layer output, as shown in Fig. 12 In traditional models without skip connections, the layers learn the transformation function $f(x)$ for the input x to map to the desired output $h(x)$. However, with the introduction of the skip connection, the layers learn the relationship $f(x) + x = h(x)$. In other words, the difference between the input and desired output, $h(x) - x$.

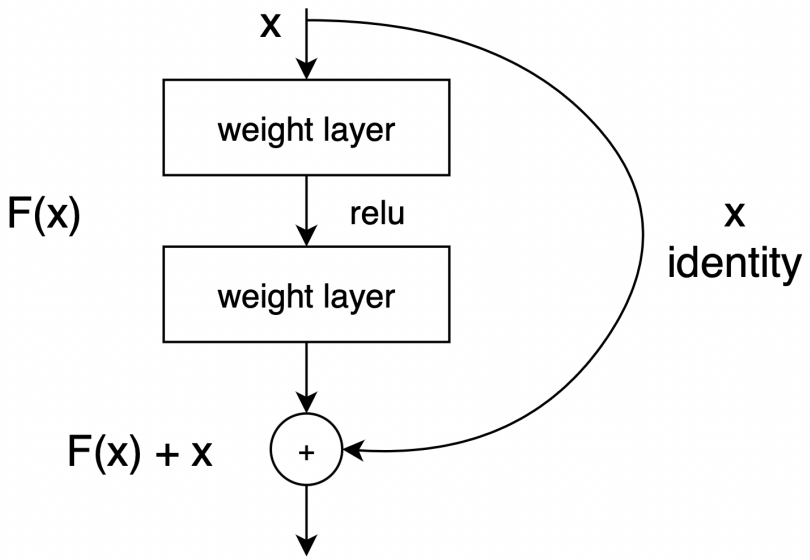

Fig. 12: Residual Block. [2]

The use of skip connections within a deep neural network maintains the accuracy increase achieved from the large layer depth, while avoiding the high training error caused by accuracy degradation [2]. This achievement is what gives ResNet the ability to train on deeper networks without accuracy degradation. In the YOLOv3 model we developed, the concept of residual connections is utilized to combat the accuracy degradation issue that came from the increase in network depth.

6. Detection Transformers

A. Introduction

In the Fall 2023 semester, our team goal researched and implemented a new framework for performing object detection on the BDD100k dataset [3] - which is a similar dataset to OpenLane-V2, but with just lane segmentation and bounding box annotations. We investigated the concept of transformers, which were novel machine learning technique proposed in the 2017 paper "Attention Is All You Need" [4], and have since been instrumental in the development of revolutionary text processing and generation tools like ChatGPT. While such tools were not relevant to the vision task of object detection, papers such as "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale (ViT)" [5] and "End-to-End Object Detection with Transformers (DETR)" [6] have demonstrated that the transformer concept can be applied to object detection tasks with similar success. So, our team researched both of these concepts. The ViT network was able to match state-of-the-art approaches such as YOLO on image classification and other tasks. The idea behind ViT was to split the image into identical patches as shown in Fig. 13, flatten them, and then apply a transformer network described in Section 6.2. While this approach worked well for classification tasks, it was unclear how we could extract features of different sizes for detection. Another problem with ViT was that it did not specify the loss function appropriate for detection. Fortunately, the DETR architecture was able to solve all of these issues. The paper introduced a different way of feature extraction as well as a new loss function appropriate for transformer architectures. So, the Fall 2023 team ended up adapting the DETR implementation to fit the vision task and dataset, for the purpose of performing unified object detection and classification. This concept is relevant because a later improvement of DETR, Deformable-DETR, is what influenced the attention mechanisms used in LaneSegNet, and that we implemented this semester.

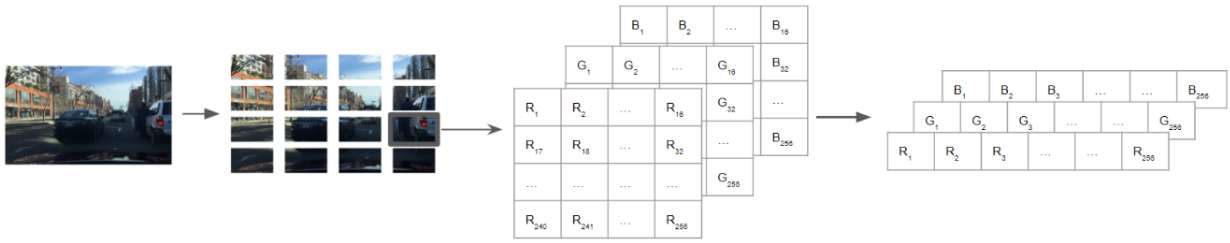

Fig. 13: Patch formation for Vision Transformer

B. Transformer Concept

The original transformer proposal introduced in "Attention Is All You Need" [4] can attribute its novelty to its concept of utilizing attention mechanisms to quantify the relationships between tokens in a sequence. In natural language processing, these tokens are represented by words in a sentence. The transformer concept disregards any usage of recurrent neural networks in lieu of an extensive attention mechanism in order to determine the dependencies between tokens.

The attention mechanism used in the DETR architecture requires a unique input. The original sequence of tokens is transformed into a vector of data points where each point contains information about the token, such as the word length and letters used for example. The attention mechanism begins by adapting the input vector by performing a linear transformation to output three matrices. These matrices — named the queries, keys, and values — signify different representations of the patterns and relationships between tokens in the sequence.

The queries represent what parts of the input the mechanism should focus on. Each query vector in the matrix is compared against the keys to determine the relevance or similarity between the query and different parts of the input sequence. The keys represent the input sequence that the model uses to determine the importance of different elements in the input sequence with respect to the queries. The similarity between the queries and keys helps the model to ascertain which parts of the input are most relevant to the current step in the decoding process. The values are the actual representations of the input sequence that are used to produce the output. The values are weighted by the attention scores obtained from comparing the queries and keys, which determines how much focus or importance each part of the input sequence should have

in influencing the output. Once these matrices are obtained from the input sequence, we can perform the Scaled Dot-Product Attention in Eq. 21 below as described in the paper [4].

$$\text{Attention}(Q, K, V) = \text{SoftMax}(\frac{QK^T}{\sqrt{d_k}})V \tag{21}$$

Where Q is the matrix of queries $(d_x \times d_k)$, K is the matrix of keys $(d_x \times d_k)$, and V is the matrix of values $(d_v \times d_v)$, and d_x is the length of the input sequence.

In the Scaled Dot-Product attention mechanism above, we first compute the dot product of Q and K^T to obtain a dot product matrix. This matrix measures the compatibility, or similarity, between the query matrix and the key matrix. This dot product matrix is then divided by the square root of the shared dimensionality of these matrices (d_k), to scale down the dot products and improve training stability. The SoftMax function, defined in Eq. 22 is then applied to this dot product matrix to compute a probability distribution of the dot products. The dot product of this distribution and the values matrix is then computed to obtain an attention matrix where each row represents a token's attention towards all other tokens in the sequence.

$$\text{SoftMax}([x_1, ..., x_n]) = [p_1, ..., p_n] \text{ where } p_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{22}$$

Where $X = [x_1, ..., x_n]$ is the input sequence vector and $P = [p_1, ..., p_n]$ is the probability distribution output.

The next component of the transformer's attention mechanism is Multi-Head Self Attention (MSA) [4]. This component, as shown in Fig. 14, takes h sequences of the three aforementioned matrices (queries, keys, values) and inputs each sequence into a Scaled Dot-Product attention function in parallel. The purpose of this component is to map each set of key-value pairs to a single output, representative of a weighted sum of the given values. After each sequence is sent through the attention function, the h output matrices are concatenated, or stacked along a dimension, to output a singular attention matrix. The parallel nature of MSA allows the network to focus on different sections of the input data and capture various dependencies and relationships

between the tokens simultaneously.

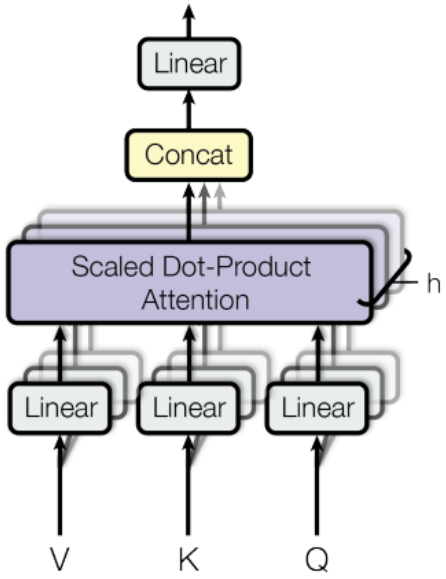

Fig. 14: Multi-Head Self Attention. [4]

C. Transformer Architecture

With the concept of Multi-Head Self Attention, we can use it to build the transformer network. The design was initially proposed in Attention is All You Need [4], and the original detection transformer paper [6] closely follows and is influenced by it. The structure of the original transformer is presented in Fig. 15.

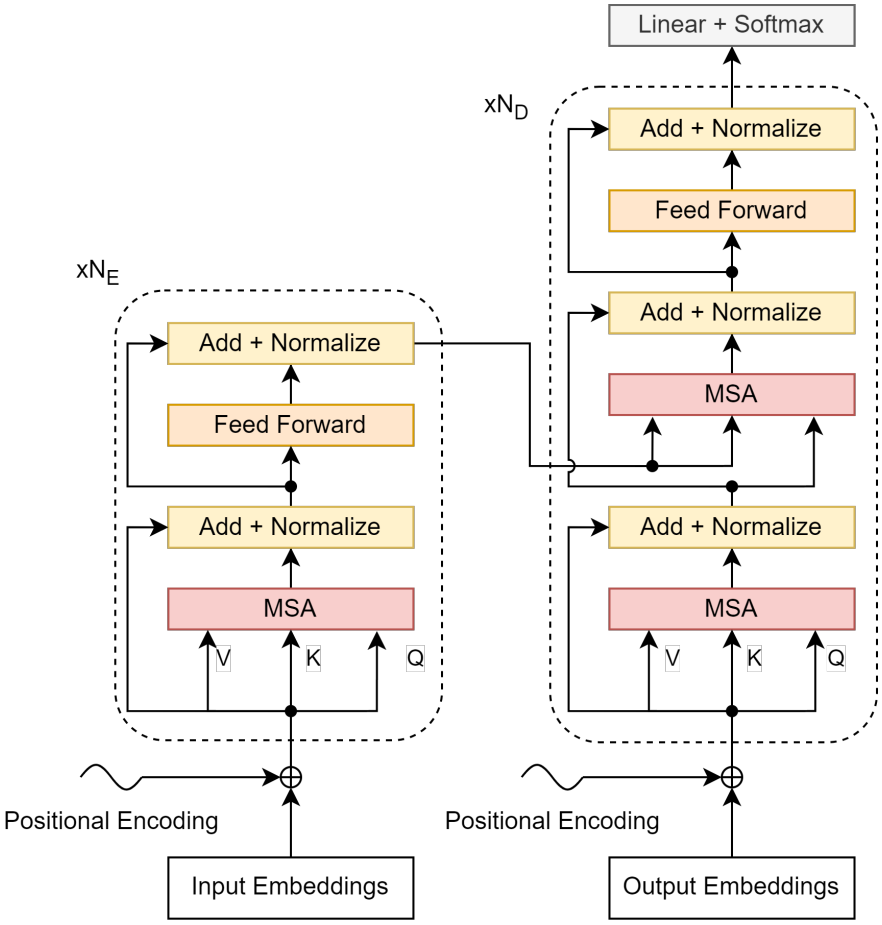

Fig. 15: Original Transformer architecture

The transformer follows a well-established encoder-decoder architecture. In the original implementation, there are $N_E = 6$ encoder blocks as well as $N_D = 6$ decoder blocks, as can be seen on the diagram. The encoder block starts with input embeddings which are an efficient way of assigning vectors to non-uniform data, such as words. Before going into the encoder block, we add positional encoding to these vectors. This is needed to make sure that the network has a way to infer the order of input features.

The first part of the encoder network is the multi-headed self-attention (labeled as MSA). This mechanisms splits the input into keys, queries, and values and looks for dependencies between the features. After the MSA block, Layer Normalization is utilized before a residual skip connection, which both help with gradient flow. The last two stages of the encoder block are a simple feed-forward network with one or two layers followed by the same layer normalization

and residual skip connection. The output of the encoder block feeds directly into the next encoder block N_E times before entering the decoder network.

The features generated by the encoder component of the transformer need to be decoded into the desired output. The decoder network is very similar to the encoder in terms of building blocks. We start with learned output embeddings, add the positional encoding, and proceed toward the decoder blocks. In a decoder block, we have two attention layers. The first one uses the self-attention mechanism from Section 6.2, which splits the input into keys, queries, and values. In the next layer, cross-attention is used, which takes as input the keys and values from the output of the encoder, along with the queries from the current features running through the decoder. By doing this, the decoder gradually incorporates information from the encoder into its final output. Similar to the encoder block, the decoder blocks finish by applying a feed-forward network. All of the three decoder blocks (self-attention, cross-attention, and feed-forward) use layer normalization and residual skip connections. After N_D decoder blocks, the output is converted into the desired format. The original DETR paper used a linear layer as well as softmax.

D. Transformers for Vision and DETR Architecture

The detailed architecture of DETR is presented in Fig. 16. The general architecture is very similar to that described in the previous section, with some differences. Firstly, the transformer network does not take the raw image directly as input. Instead, a feature extractor generates features representative of patterns in the image, with such features being inputted to the transformer. One example of such a feature extractors would be a convolutional neural network. In our implementation, we used a ResNet-50 network pre-trained on the COCO dataset. This ensures that the transformer receives features containing detailed semantic information about the image.

An additional difference is how the positional encodings are used. Instead of only one encoding at the beginning of each stage (encoder and decoder), positional encodings are added only to the keys and queries of encoder in self-attention as well as the keys of the decoder in cross-attention. Finally, the output of the decoder stage produces N_Q queries, where each query representing a potential object detection. With this architecture, the network outputs a set of predicted objects with no particular structure. This often leads to problems when it comes to evaluating the loss,

as the model does not know which groundtruth bounding box to compare each output with. This problem is solved through the use of bipartite matching loss, described in Section 6.5.

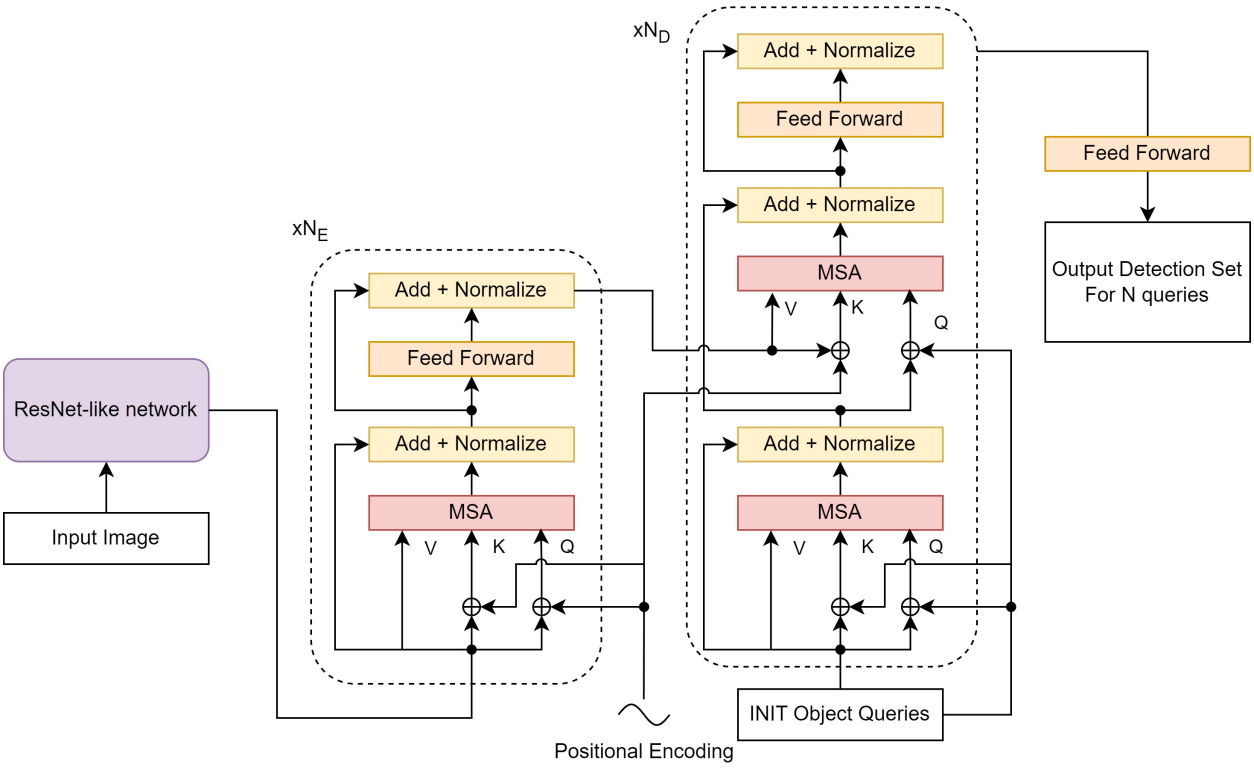

Fig. 16: Detection Transformer Full Architecture

1) Attention masking: The detection transformer, unlike most CNN models described earlier, is capable of working with images of different sizes. To accomplish this, it must utilize a canvas size of (W_c, H_c) where $W_c = \max_{i \in \text{images}} W_i$, W_i is the width of image i, and the same for the height H_c. Then, the image is padded around its borders with zeros. To make sure that these values do not propagate through the network, attention masking is used. When propagating a value through the network, a pair $(\textbf{values}, \textbf{mask})$ is propagated, where **mask** will have binary indicators to signify whenever the pixel value should be irrelevant. To see how attention masking is applied in the attention mechanism, one must understand the object shapes used in attention. The equation for attention (21) is based on matrices Q an K of sizes $(d_x \times d_k)$ and $(d_x \times d_k)$ respectively. Since QK^T is just a matrix of dot-products of size $(d_x \times d_x)$, each row corresponds to a query and each column to a key. These are the corresponding attention values as represented by query-key correspondence. Thus, if we make certain elements of this matrix very small, this

will ensure that the entries from the values matrix corresponding to the irrelevant entries will not be included in the resulting attention. This boils down to setting the columns to $-\inf$ for the features where the attention mask value is 1.

2) Positional encoding: Positional coding is important to transformers as well, as they introduce sequence context to the data. In the attention mechanism, the procedure is invariant to the permutations of input elements. If two features are swapped, all the corresponding rows or columns in Q, K, and V will be also swapped, thus obtaining a fully swapped result. This is not desirable since the model should understand when some elements are closer in the sequence or further apart. While this is obvious in the domain of natural language processing, this ordering is just as important for image features. After the feature extractor, a set of features of dimension (C, H_f, W_f) is obtained where C is the number of features and (H_f, W_f) is the size of each feature. Every 2D feature map along the first dimension corresponds to different aspects of the input image, which may be related in different ways. Therefore, having an understanding of the ordering is quite useful.

Positional encodings P are the same size as the feature set X that they encode. Assume the input X is a matrix of shape $(d_x \times d_{model})$. The positions of the features should be encoded along the first dimension. There are two common approaches - using manually constructed sine and cosine waves or having the positional encodings be learned parameters. Using learned parameters is relatively straightforward, as this simply involves introducing a new learnable weight matrix. The sine and cosine method, used in the original implementation of transformers in [4] works as follows.

The goal is to obtain a set of unique sine and cosine sequences for each position in X. If t is the position along the first dimension, define:

$$\omega_k = \frac{1}{T^{\frac{2k}{d_{model}}}}$$

where T is a balancing parameter, often set to $10,000$ in practice. At position t, define the

following d_{model}-dimensional vector as:

$$p_{t,i} = \begin{cases} \sin \omega_{\lfloor \frac{i}{2} \rfloor} & i \vdots 2 \\ \cos \omega_{\lfloor \frac{i}{2} \rfloor} & i \not\vdots 2 \end{cases}$$

Whose values can be combined into a row vector:

$$\underset{1 \times d_{model}}{p_t} = \begin{bmatrix} \sin(\omega_1 t) & \cos(\omega_1 t) & \cos(\omega_2 t) & \cos(\omega_2 t) & \dots \end{bmatrix}$$

which finally will provide the positional encoding matrix:

$$\underset{d_x \times 1}{P} = \begin{bmatrix} — & p_1 & — \\ — & p_2 & — \\ & \vdots & \\ — & p_{d_x} & — \end{bmatrix}$$

Finally, this matrix is added to the original input: $X \rightarrow X + P$. Note that in the case of images the input is not of size $(d_x \times d_{model})$ but generally of dimension $(d_{features} \times W \times H)$. In this case, the positional encodings are applied with positional index t being along the first dimension, the features. The main difference here is that the sine and cosine functions will be calculated independently along both the W and H dimensions, which will lead to multiple unique sequences.

E. Bipartite Matching Loss

To quantify the networks's object detection accuracy after each iteration, the DETR architecture employs bipartite matching loss [6].

This loss function requires a hyperparameter N, representative of the fixed number of bounding boxes the network will detect within a singular image. As the DETR paper suggests, the value for N should be quite large [6], since it should be at least as large as the maximum number of groundtruth objects in any image from our dataset. In practice, images will have varying numbers of groundtruth objects, so images with less than N groundtruth objects will result in

an excessive amount of detections. Fortunately, bipartite matching loss is designed to account for such situations.

Once the N detections are obtained for a given image, bipartite matching loss will supplement the groundtruths with M "empty object" elements, where M is the remainder amount between the number of groundtruths and the number of detections. The bipartite matching function will aim to match the M false detections to these empty object elements, thus removing them from the loss computation. The Bipartite matching loss function will obtain an optimal set $\hat{\sigma}$ (defined in Eq. 23), representative of the set of matches from all match permutations which minimizes the total Hungarian loss between each ground truth box and its corresponding detection match. A visual representation of how the network obtains the optimal matching set is shown in Fig. 17.

$$\hat{\sigma} = \underset{\sigma \in \sigma_N}{\arg\min} \sum_{i}^{N} \mathscr{L}_{Hungarian}(y_i, \hat{y}_{\sigma(i)}) \tag{23}$$

Where $\hat{\sigma}$ is the permutation of detection matches which minimizes the bipartite loss, N is the hyperparameter representative of the number of objects to detect, σ_N is all permutations of detection matches, $\mathscr{L}_{Hungarian}$ is the matching cost between a given detection and groundtruth, y_i is the i^{th} ground-truth bounding box, $\hat{y}_{\sigma(i)}$ is the corresponding bounding box detection match from the i^{th} match set, c_i is the groundtruth class of the i^{th} object, and $\hat{c}_i$ is the predicted class for the i^{th} detected box.

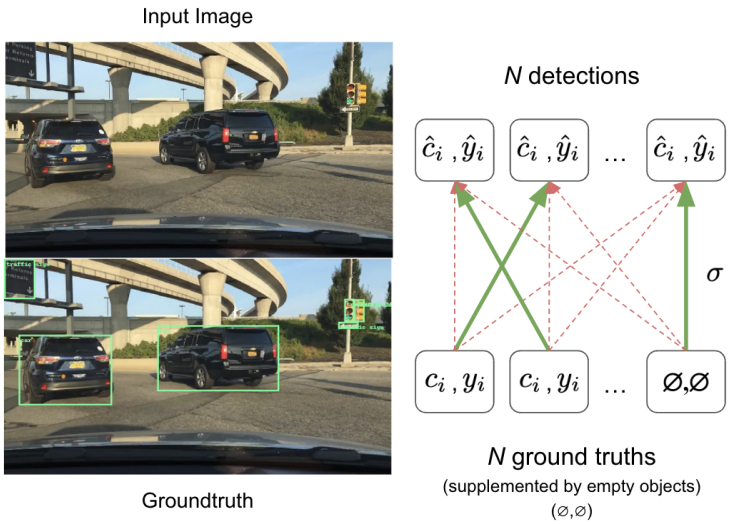

Fig. 17: Bipartite Matching Loss. [6]

The Hungarian Loss function that the Bipartite matching minimizes is defined in the DETR paper [6] to measure two penalty terms. One of these terms handles bounding box detection accuracy, and the other handles bounding box classification accuracy. Hungarian loss, as shown in Eq. 24, calculates the negative log-likelihood of the probability that a given detection class matches the ground truth class, and adds the bounding box penalty term (ignored for empty objects), which is defined as Complete-IOU loss.

$$\mathscr{L}_{Hungarian}(y_i, \hat{y}_i) = \sum_{i=1}^{N} \left(-log(\hat{p}_{\hat{\sigma}(i)}(c_i)) + \mathbb{1}_{\{c_i \neq \varnothing\}} C_{IOU}\right) \tag{24}$$

Where $\hat{\sigma}$ is the permutation of detection matches which minimizes the bipartite loss, N is the hyperparameter representative of the number of objects to detect, y_i is the i^{th} ground-truth bounding box, c_i is the groundtruth class of the i^{th} object, $\hat{p}_{\hat{\sigma}(i)}(c_i)$ is the i^{th} detection's probability of matching the groundtruth class, $\mathbb{1}_{\{c_i \neq \varnothing\}}$ is the coefficient to ignore empty object bounding boxes, and C_{IOU} is the bounding box loss function, as defined in Section **??**.

F. Evaluation Metrics

Mean Average Precision (mAP):

For evaluating the accuracy of the Fall 2023 model and its detections, the Mean Average Precision (mAP) metric was used. To calculate mAP, we need to use both the Precision and

Recall metrics. Precision is the total number of true positive predictions made by the model divided by the number of true positive predictions plus the false positive predictions. Similarly, recall is the total number of true positive predictions made by the model divided by the number of true positive predictions plus the false negative predictions. mAP is calculated using average precision (AP), which is the area under the precision-recall curve in Eq. 43.

$$Precision = \frac{TP}{TP + FP} \tag{25}$$

$$Recall = \frac{TP}{TP + FN} \tag{26}$$

$$AP = \int_0^1 Precision(Recall) d(Recall) \tag{27}$$

$$mAP = \frac{1}{C} \Sigma_{i=1}^{C} AP_i \tag{28}$$

Where AP_i is the average precision of class i, and C is the number of classes. This is an important metric used to understand how well a given model detects objects.

G. Results

The DETR network developed in Fall of 2023 semester was trained for 100 epochs with evaluation metric results shown in Table I. As described, the mAP metric was used to compare against the Facebook implementation. At an IOU threshold of 0.5, the network achieved a mAP of 33.4, slightly lower than Facebook's DETR which achieved 40.6. When investigating the breakdown of accuracies specific to object sizes, it was found that large objects had a mAP of 40.2, medium objects had 21.0, and small objects had 4.0. Although the accuracies were lower than Facebook's implementation across the board, the results were promising considering that Facebook's DETR was trained for 300 epochs compared to our 100. The results of the model can be further contextualized through a visualization of the model outputs shown in Fig. 18 - 23. Note that Fig. 18 - 21 are images from the BDD100k dataset, and Fig. 22 - 23 are images from around Purdue, completely external to BDD100k, for the purpose of testing generalizability.

Metric	Our DETR (100 epochs)	Facebook DETR (300 epochs)
mAP@0.5	33.4	40.6
mAP@0.5 (Large)	40.2	60.2
mAP@0.5 (Medium)	21.0	44.3
mAP@0.5 (Small)	4.0	19.9

TABLE I: Metric Comparison of DETR Implementation.

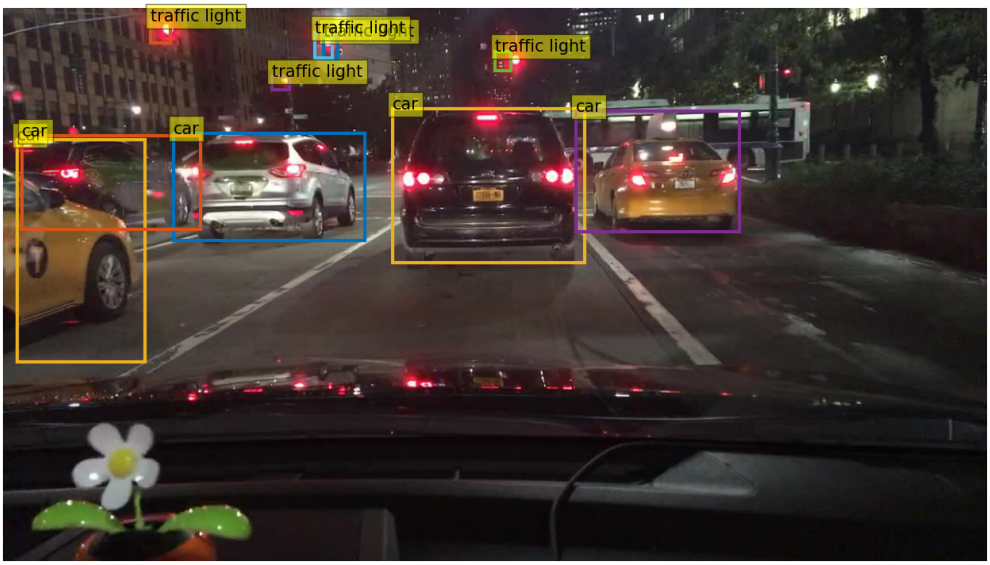

Fig. 18: Results of DETR model (BDD100k)

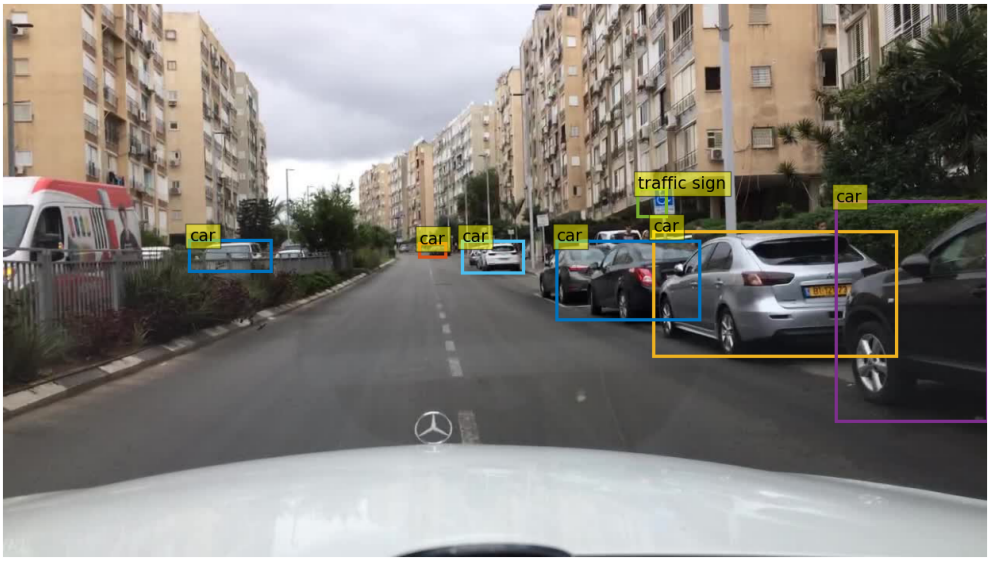

Fig. 19: Results of DETR model (BDD100k)

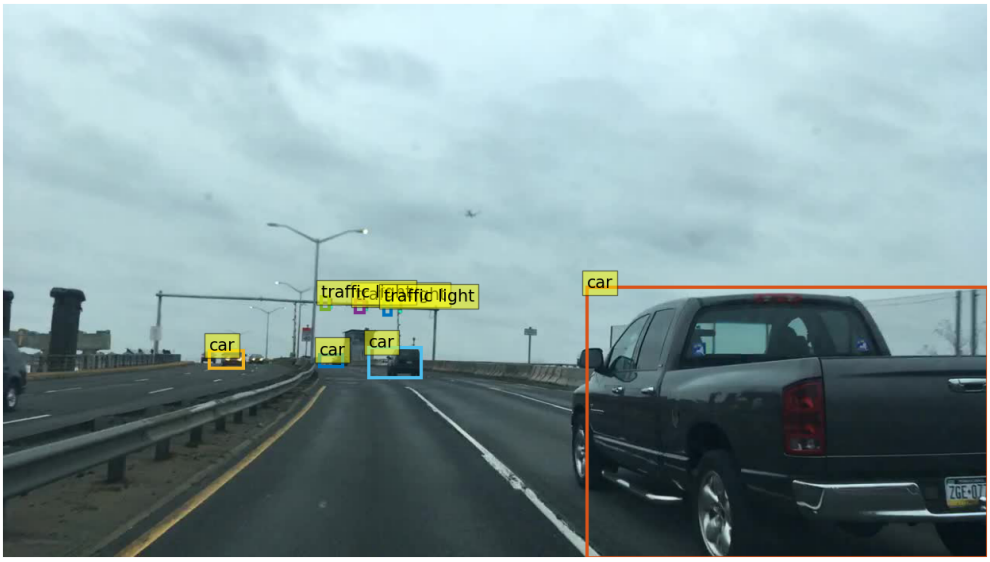

Fig. 20: Results of DETR model (BDD100k)

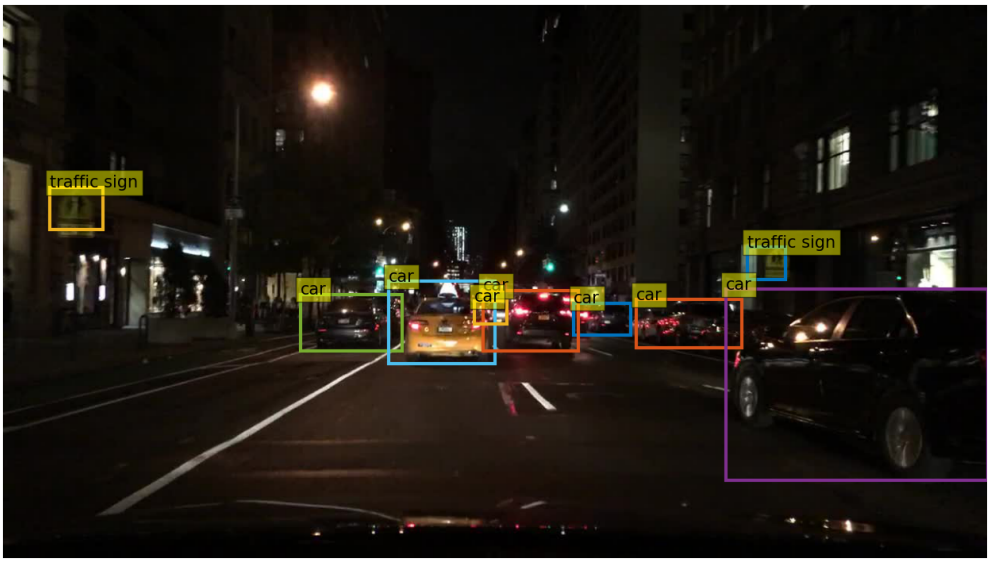

Fig. 21: Results of DETR model (BDD100k)

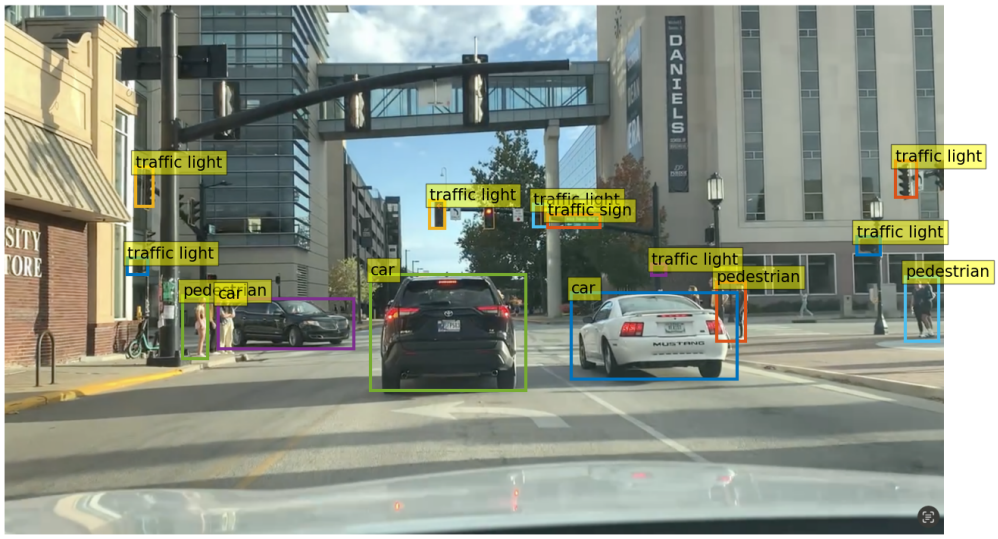

Fig. 22: Results of DETR model (Purdue)

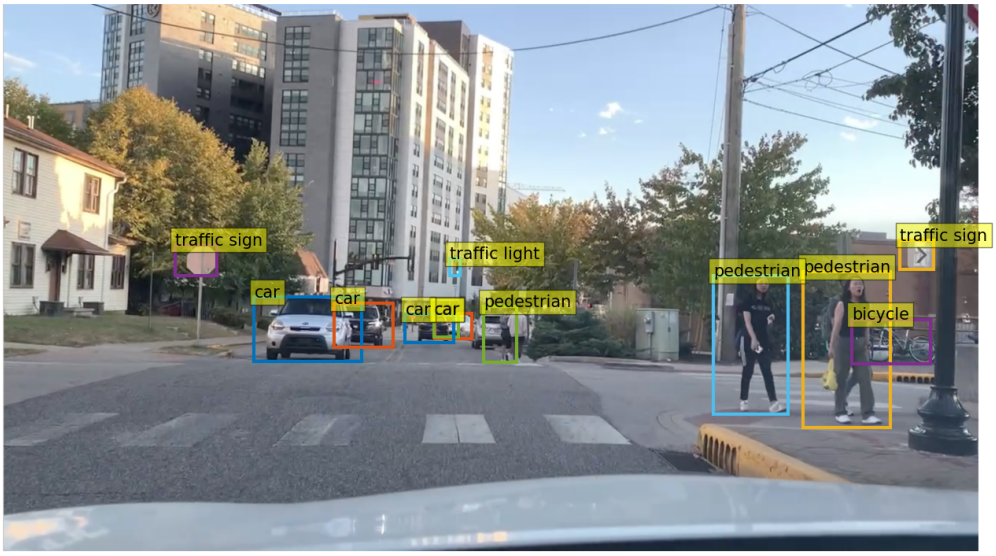

Fig. 23: Results of DETR model (Purdue)

7. LaneSegNet

A. Introduction

To build off of the DETR results from last semester, and explore a new approach to lane segmentation that would produce a more useful output to autonomous driving software than object detections and segmentations, we investigated the LaneSegNet [7] architecture. LaneSegNet was developed in December 2023, and leverages the OpenLane-V2 [1] dataset to build a lane topology prediction model. LaneSegNet differs from classic lane-line detection architectures due to its ability of combining topological information with lane-line information to produce a more defined and contextual understanding of the geographic environment. Most models in the field of autonomous driving are able to produce accurate lane lines but do not take into account the topological information that can enable the algorithm to develop more informed decisions relative to its environment. LaneSegNet employs a Lane Attention module that enables it to acquire crucial details like the relationship between lanes and four-way intersections, lane exits, and pedestrian crossings to develop a robust mapping.

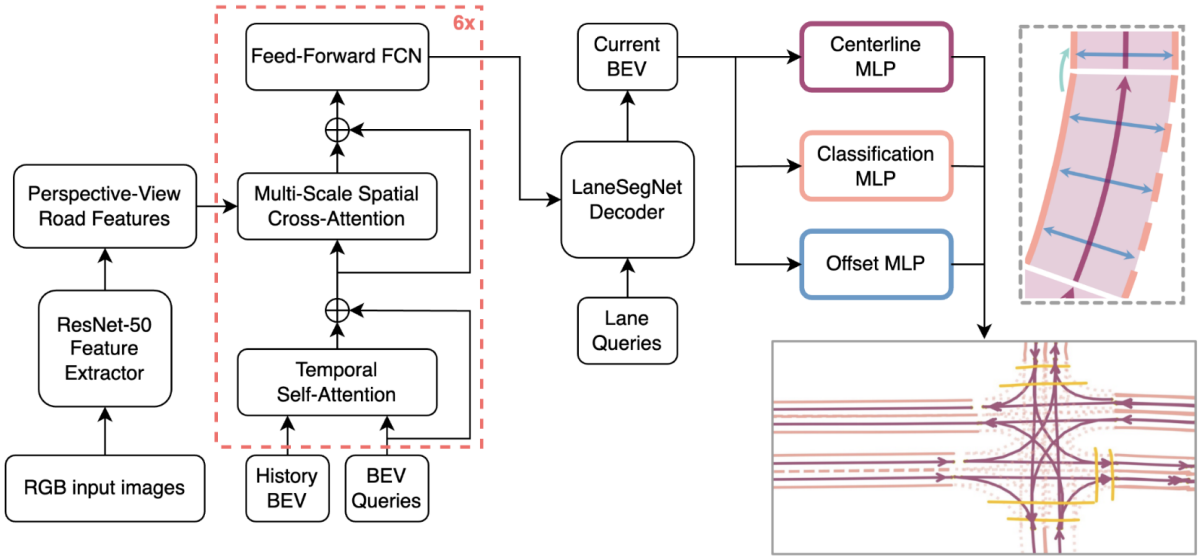

Fig. 24: LaneSegNet Architecture

Fig. 24 illustrates the LaneSegNet architecture, which incorporates components from other networks like ResNet-50 [2], BEVformer [8], and the attention mechanism. The attention mech-

anisms applied in the LaneSegNet architecture are different from the classic attention mechanism proposed in "Attention is All You need" [4]. LaneSegNet uses variations of Deformable Attention (Section 7.2), which is a modified version of the classic mechanism that allows for more efficient contextual information extraction from images. The attention mechanisms used in the LaneSegNet encoder, Temporal Self Attention (Section 7.3) and Multi-Scale Spatial Cross-Attention (Section 7.4), are all based on the Deformable Attention mechanism. The decoder contains the Lane Attention (Section 7.6) module, which is able to extract long-range information from the inputs - a limitation of Deformable Attention. The decoded lane queries are processed with the BEV features to the prediction heads. The centerline, classification, and offset heads employ parallel MLPs to develop the mapping with defined lane segments.

B. Deformable Attention

Deformable Attention is a modified form of the vanilla Attention [4] mechanism. The vanilla Attention mechanism obtains attention relationships between all patches in the image. Deformable attention optimizes this process by using a reference point and learned offsets to evaluate a smaller set of key points surrounding the reference point. The concept was developed in the Deformable-DETR [9] paper, which was influenced by the intutition behind Deformable Convolution.

$$DeformAttn(z_q, p_q, x) = \sum_{m=1}^{M} W_m \sum_{k=1}^{K} A_{mqk} W'_m x(p_q + \Delta p_{mqk}) \tag{29}$$

We consider x as the input feature map such that $x \in \mathbb{R}^{C \times H \times W}$. W_m denotes the attention head. Let q index a query element with content feature z_q. A 2-D reference point p_q is used along which the learned offsets p_{mqk} are applied to obtain the Deformable attention boundaries. The attention head is indexed by m and the sampled keys are indexed by k where K is the total number of samples. A_{mqk} denotes the scalar attention weight lies in the range [0,1], normalized by $\sum_{k=1}^{K} A_{mqk} = 1$. The offsets are applied to the reference points in ($p_q + \Delta p_{mqk}$) and bi-linear interpolation is applied in computing $x(p_q + \Delta p_{mqk})$. The query feature z_q is fed to a linear projection operator of 3MK channels, where the first 2MK channels encode the sampling offsets

(Δp_{mqk}) and the remaining MK channels are fed to a Softmax operator to obtain the attention weights A_{mqk}. This mechanism is leveraged to greater scale to process multiple input feature scales simultaneously. This is applied for Multi-Scale Deformable Attention which is based on the Deformable Attention mechanism discussed above.

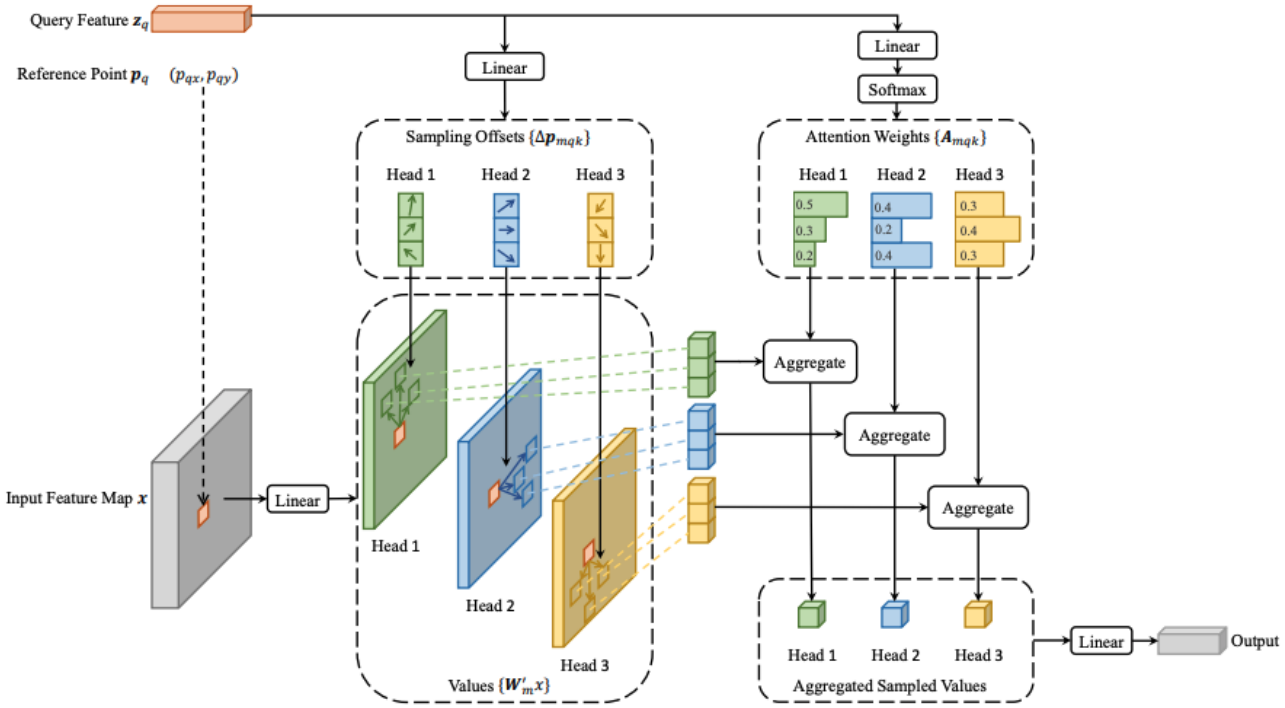

Fig. 25: Deformable Attention. [9]

The above figure, derived from [9], illustrates the mathematical relationship described earlier. The query feature is processed with the learned sampling offsets which are applied to the attention heads relative to the reference point from the input feature map. These are extracted and processed with the attention weights to acquire the aggregated sampled values. These are passed through a linear layer to produce the corresponding output.

C. Temporal Self-Attention

Temporal Self-Attention (TSA) employs the Deformable Attention mechanism to understand the surrounding environment. Temporal clues enable the model to understand contextual infor-

mation such as the velocity of moving objects. This mechanism incorporates history Bird's Eye View (BEV) features.

$$TSA(q, \{Q, B'_{t-1}\}) = \sum_{V \in \{Q, B'_{t-1}\}} DeformAttn(p_q, V) \tag{30}$$

where Q denotes the BEV Queries, p denotes the point (x, y) in the BEV plane, Q_p is the BEV query associated with point p, B_{t-1} denotes the BEV features at time-step $t-1$ (history BEV), and B'_{t-1} is the history BEV realigned to match real-world location with Q.

The history BEV features from the previous time stamp B_{t-1} and the BEV queries Q from the current timestamp interact with each other to acquire and process temporal information.

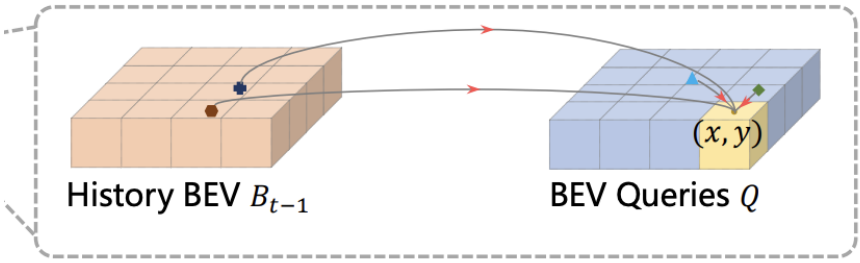

Fig. 26: History BEV and Query Alignment

The (x, y) point on the BEV plane is acquired in Q and processed relative to B_{t-1}. Using ego-motion, the features are aligned with the queries forming B'_{t-1}. This process is executed through a particular amount of time to incorporate all the feature and query maps. The element that differentiates this mechanism from vanilla Deformable attention is that the offsets in TSA are predicted through the concatenation of Q and B'_{t-1}. The first sample of each sequence will degenerate into a self-attention mechanism since the first time step has no history. In the first sample, we replace the BEV features $\{Q, B'_{t-1}\}$ with duplicate BEV queries $\{Q, Q\}$.

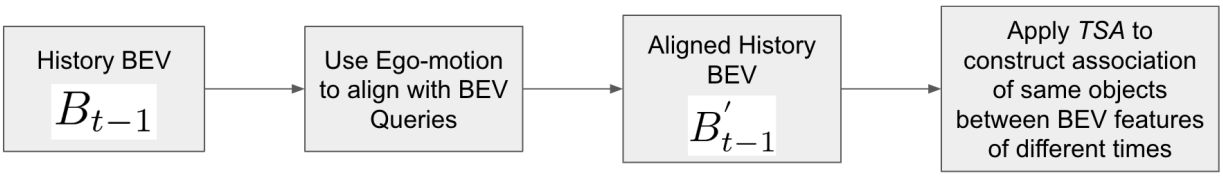

Fig. 27: Temporal Self Attention

D. Spatial Cross-Attention

Spatial Cross-Attention (SCA) is another attention mechanism used by the LaneSegNet architecture. Similar to Temporal Self-Attention, SCA is based on the deformable attention mechanism; however, SCA inquires about the spatial information from the multi-camera feature maps rather than temporal information of feature maps at different timesteps.

As mentioned previously, the BEV queries $Q \in \mathbb{R}^{H \times W \times C}$ is a group of grid-shaped learnable parameters, such that the query $Q_p \in \mathbb{R}^{1 \times C}$ located at $p = (x, y)$ of Q is responsible for that grid cell region in the BEV plane of size $H \times W$. SCA is performed by lifting each query of the BEV plane Q_p and sampling N_{ref} 3D reference points. These 3D reference points then go through a projection function $P(p, i, j)$ to find the 2D point on the i^{th} camera view that corresponds to each specific reference point. Some of these 3D reference points will be projected onto multiple camera views; that is why the projection function accepts the camera view as an argument. The set of these used or "hit" camera views is V_{hit}. The projection function adjusts the point p so that the car taking the videos is in the middle of the coordinate system. Since objects located at this point will appear at different heights, a set of predefined anchor heights $\{z'_j\}_{j=1}^{N_{\text{ref}}}$ is used for the reference points at this point p. This pillar of 3D reference points $(x', y', z'_j)_{j=1}^{N_{\text{ref}}}$ is then projected into 2-dimensional space to locate the reference points on the camera view feature maps. The following equation represents how the projection function works:

$$
\begin{aligned}
P(p, i, j) &= (x_{ij}, y_{ij}) \\
\text{where } z_{ij} \begin{bmatrix} x_{ij} & y_{ij} & 1 \end{bmatrix}^T &= T_i \begin{bmatrix} x' & y' & z'_j & 1 \end{bmatrix}^T, \\
x' = s(x - \frac{W}{2}), &\qquad y' = s(y - \frac{H}{2})
\end{aligned} \tag{31}
$$

where $p = (x, y)$, i iterates through the camera views, j iterates through the 3D reference points, W and H are the spatial dimensions of the BEV plane, s is the real-world size in meters that each BEV plane grid cell corresponds to, (x', y') is the point p adjusted to put the origin of the plane at the center of the camera views, z'_j is the height of the j^{th} reference point for Q_p, T_i is the known matrix used to project 3D points to 2D points on the i^{th} camera view, and (x_{ij}, y_{ij}) is the j^{th} reference point's projection onto the i^{th} camera view.

The SCA function goes through each projected reference point on each hit camera view and sums the output of deformable attention on the BEV queries, j^{th} reference point, and feature maps of the i^{th} camera view. The sum is then divided by the number of camera views hit by all reference points for Q_p. The equation that represents the entire Spatial Cross-Attention mechanism is as follows:

$$SCA(Q_p, F_t) = \frac{1}{|V_{\text{hit}}|} \sum_{i \in V_{\text{hit}}} \sum_{j=1}^{N_{\text{ref}}} \text{DeformAttn}(Q_p, P(p, i, j), F_t^i) \tag{32}$$

where V_{hit} is the set of camera views that the projected reference points are on, i iterates through these hit camera views, N_{ref} is the number of reference points chosen for each BEV query, j iterates through the reference points, Q_p is the BEV query at the 2D point $p = (x, y)$, F_t^i is the set of feature maps of the i^{th} camera view at time t of the video, and $P(p, i, j)$ is the projection function used to project the j^{th} 3D reference point onto the 2D i^{th} camera view. The output of SCA is the BEV queries Q updated with the spatial information and features at time t.

E. LaneSegNet Encoder: BEVFormer

The LaneSegNet [7] architecture follows a transformer structure containing three main components: an encoder, a decoder, and a predictor. This section explains how the encoder module of the LaneSegNet architecture works. The LaneSegNet encoder follows the BEVFormer [8] architecture, a spatiotemporal transformer that utilizes the previously mentioned Temporal Self-Attention and Spatial Cross-Attention mechanisms. The BEVFormer architecture accepts perspective-view videos from multiple camera angles and produces a set of Bird's-eye-view (BEV) feature maps. These BEV feature maps are then passed to the LaneSegNet decoder to be further utilized. Below is a portion of Fig. 24 that shows how the BEVFormer module functions:

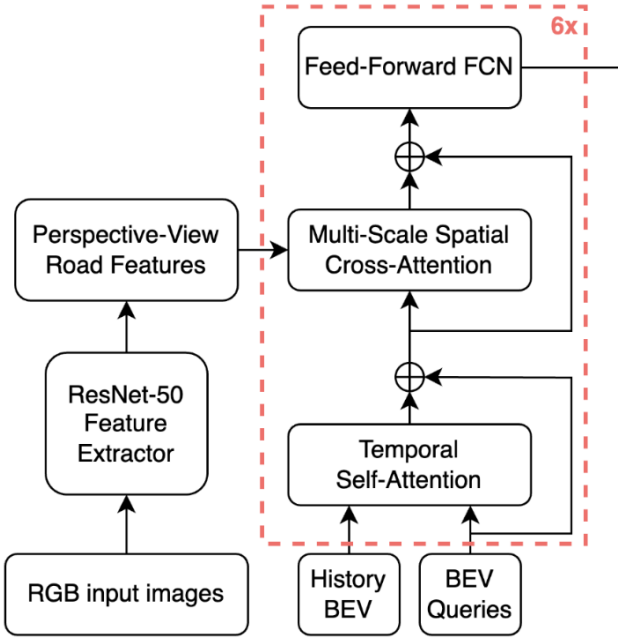

Fig. 28: LaneSegNet Encoder (BEVFormer) Architecture

Before anything is done to the data, input images are sampled at the same rate and times from the videos of each camera angle. These images at different times and angles are then passed through a feature extractor that produces a set of feature maps for each camera and timestep. The BEVFormer architecture, specifically, uses the ResNet-50 [2] feature extractor to produce these features. These feature maps are then passed to the encoder stack of BEVFormer to be used later. The other two inputs accepted by the BEVFormer encoder stack are the History BEV B_{t-1} and the BEV Queries Q. These two inputs are passed to the Temporal Self-Attention mechanism with a residual connection to obtain temporal information regarding the car's environment. After the residual connection, the BEV Queries are normalized and passed to the Spatial Cross-Attention mechanism with another residual connection. The Spatial Cross-Attention module accepts the perspective-view feature maps generated by the ResNet-50 backbone and performs cross-attention between these images and the BEV Queries Q to obtain spatial information regarding features in the car's environment. The residual connection is added to the output of this mechanism, and this output is normalized again before being passed into a feed-forward network. After one last step of adding the residual connection and normalizing the output of the feed-forward network,

the refined BEV features are generated from time t (B_t). The BEV features are then passed into the next encoder layer until 6 iterations of the encoder have been performed. The output of the last encoder layer will be the final BEV features at time t that will be used by the LaneSegNet decoder. This process is then repeated for each time step to generate BEV features for the entire 15-second-long video.

F. Lane Attention

The Deformable Attention mechanism is used in a wide variety of object detection tasks due to its optimizations and effectiveness compared to Multi-Head Attention. When it comes to map learning tasks, however, the Deformable Attention falls short. As explained previously, deformable attention uses reference points placed at the centroid of predicted objects and refines the sampling locations used by adjusting the sampling offsets through iterations. In the first iteration of the decoder layer, the sampling offsets are initialized using information about the geometry of the predicted 2D objects.

In map learning, Deformable Attention struggles to capture the elongated shape of lane lines as it is not well-equipped to identify and incorporate long-range information pertinent to attention. Therefore, an attention mechanism that would be more effective at the task of map learning would need to incorporate long-range information while extracting local details. The LaneSegNet [7] architecture introduces a new attention mechanism to remedy these issues: Lane Attention.

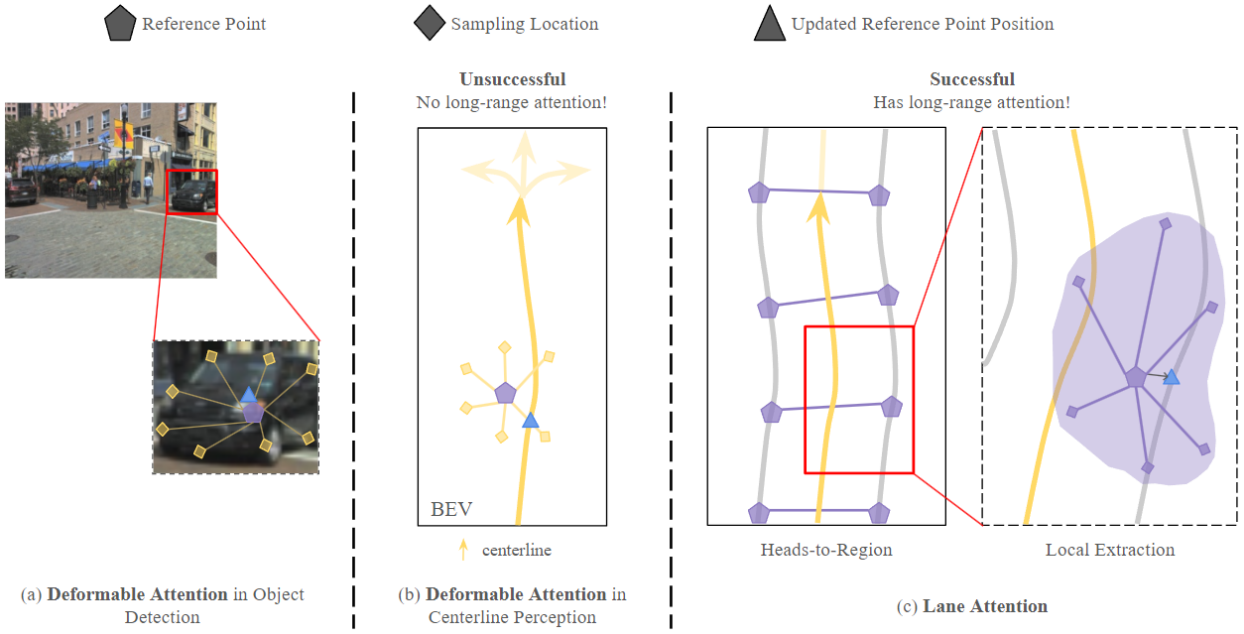

Fig. 29: Comparing Lane Attention and Deformable Attention

Lane Attention is a *heads-to-region* multi-head attention mechanism. This means the attention module creates multiple local regions with reference points to sample from; each of these regions has an attention head dedicated to it. Lane Attention also utilizes a multi-branch mechanism that dedicates sub-heads of the attention mechanism to a set of sampling points within a local region. In figure 29, the advantages of this approach compared to Deformable Attention is demonstrated as it has the advantage of retaining long-range information and connections. The following equation is the mathematical representation of the Lane Attention mechanism:

$$\text{LaneAttn}(q_i, p_i, B) = \sum_{m=1}^{M} W_m \left[\sum_{k=1}^{K} a_{i,m,k} \cdot W'_m \text{Bi-linear}(B, p_{i,m} + \Delta p_{i,m,k}) \right] \tag{33}$$

where $B \in \mathbb{R}^{H \times W \times C}$ is the set of BEV features generated by the LaneSegNet [7] encoder (BEVFormer [8]), H and W are the spatial dimensions of the BEV plane, C is the dimension of the embedding used in BEVFormer, $q_i \in \mathbb{R}^C$ is the i^{th} lane segment query, $p_i \in \mathbb{R}^{M \times 2}$ is the set of reference points for the i^{th} lane segment query, M is the number of attention heads (or the number of local regions/reference points), K is the number of sampling locations per head/region, $C_v = C/M$ is used as a dimension for a split channel to retain the results of each attention head, $W_m \in \mathbb{R}^{C \times C_v}$ and $W'_m \in \mathbb{R}^{C_v \times C}$ are learnable weights that project the results to

the split channels, $a_{i,m,k}$ are the attention weights, $\Delta p_{i,m,k} \in \mathbb{R}^2$ is the learned sampling offset for the k^{th} sampling point of the m^{th} reference point, and the Bi-linear function performs a bi-linear interpolation on each sampling points in the BEV plane.

Equation 33 shows that the Lane Attention mechanism retains a query of dimension C for each point on the output lane map. An attention *head* is then assigned to each of the M reference points, and a set of K sampling points is obtained for each reference point to create a *region*. The sampling offsets are learnable and can either attend to long-range or local features. The module then iterates through each sampling point for each attention head and locates this point on the 2D BEV plane using bi-linear interpolation. The BEV features at this point follow the same process as Deformable Attention as they are multiplied by a weight matrix that projects these features to the split channel C_v and the attention weights are applied. After all sampling locations go through this process, another set of weights is applied to the sum of their attention outputs. This weight matrix also projects these attention values back into the channel dimension C. This output represents the lane attention output for the m^{th} attention head. The results of all attention heads are summed to generate the updated lane attention query.

G. LaneSegNet Decoder

Once the BEV features are generated by the LaneSegNet Encoder [8], the feature maps are passed through the LaneSegNet Decoder. The decoder of this architecture utilizes the aforementioned lane attention mechanism and has 6 total layers. The input to each layer is the lane segment queries q. These queries go through an iteration of self-attention and are then passed through the lane attention module. After the query embedding is updated with the lane attention mechanism, they are passed through a feed-forward neural network to update the positions of the reference points to be where the predicted lane lines are. This feed-forward network predicts the lane centerlines and the offset (distance from centerline to edge), and this is used to adjust the reference points used for each lane segment query. These reference points are then used in the next decoder layer as they are contained in the lane segment query embedding.

For the first iteration of the decoder, the sampling offsets are initialized using geometric information around the reference point; this is done by using a grid of the unit circle with an

increasing radius around the reference point in the lane segment query. Geometric information within the grid is used to obtain sampling points at various distances. The reference points are initialized identically for each attention query based on the position embedding. This means reference points for each attention head are initialized in the same positions with respect to the location of the lane segment query on the BEV plane. Instead of using geometric information for initialization similar to the sampling offsets, reference points are identically initialized to avoid the additional task of initially learning the intricate geometries of the features. These reference points are then adjusted for each lane segment query. The final output of the last layer of the decoder is then passed to the LaneSegNet Predictor.

H. LaneSegNet Predictor

After the current BEV feature has been determined by the model, it must be transformed through post-processing to draw all predicted and detected lane line features on the BEV plane. This is achieved through a sequence of Multi-Layer Perceptron (MLP) heads, each responsible for processing a separate feature. The three heads, as shown in the prediction head of LaneSegNet architecture in Fig. 30, handle the centerline, lane classification, and offset. The centerline head, highlighted in purple, represents the central path of the lane, and is responsible for representing the curvature, location, and direction of the lane. The classification head, highlighted in orange, represents the right and left boundaries of the lane, as well as which type of lane line these boundaries are (solid, dashed, etc.). The offset head, highlighted in blue, represents the offset distance between the centerline and the lane line boundaries, making it responsible for determining the width of the lane. When all these feature heads are aggregated together, they produce the desired BEV output, containing the predicted satellite view of how the road scenario looks.

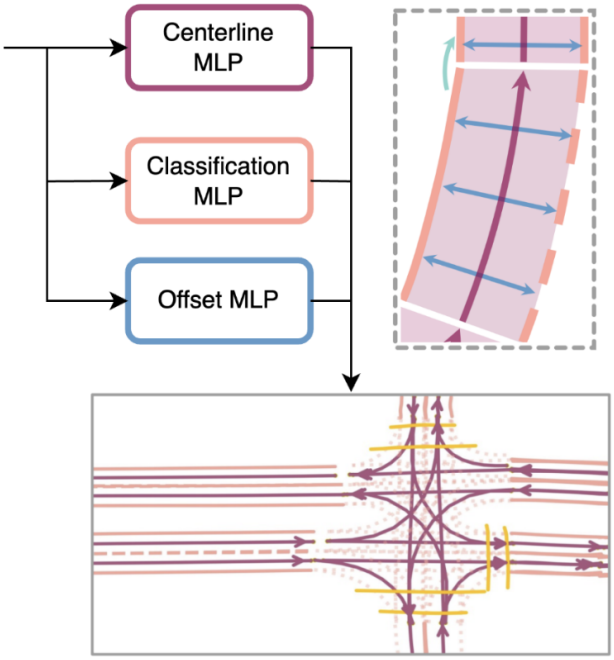

Fig. 30: LaneSegNet Predictor Component.

I. LaneSegNet Loss

The way that the loss is calculated in LaneSegNet is predicated on the Detection Transformer (DETR) loss. Both architectures utilize the bipartite matching to find the lowest cost however, the architectures differ in the components that make up their respective loss functions. The LaneSegNet loss function is defined as follows:

$$\mathcal{L} = \lambda_{vec}\mathcal{L}_{vec} + \lambda_{seg}\mathcal{L}_{seg} + \lambda_{cls}\mathcal{L}_{cls} \tag{34}$$

The first component, $\lambda_{vec}\mathcal{L}_{vec}$, in Eq. 34 evaluates the l_1 loss between the vectorized candidate lane $\vec{v}$ and the ground truth $\hat{v}$ lane. For training, the LaneSegNet paper [7] set the hyperparameter λ_{vec} to 0.025. The next component in the LaneSegNet loss function is the lane segmentation loss. The lane segmentation loss is calculated by using dice loss and cross-entropy loss, as shown in Eq. 35.

$$\mathcal{L}_{seg} = \lambda_{dice}\mathcal{L}_{dice} + \lambda_{ce}\mathcal{L}_{ce} \tag{35}$$

Essentially, dice loss evaluates how well the segmentation of the architecture overlays on the ground truth segmentation. The proper mathematical definition is defined in Eq. 36.

$$\mathcal{L}_{dice} = 1 - 2\left(\frac{|C \cap G|}{|C| + |G|}\right) \tag{36}$$

The numerator, $|C \cap G|$, represents the area of intersection between the candidate segmentation, $|C|$, and the ground truth segmentation, $|G|$. The denominator, $|C| + |G|$, is the total area of the ground truth segmentation and candidate segmentation. Hence why there is a multiple of two in front of the term because in a perfect case where the candidate segmentation equals to the ground truth segmentation ($|G| = |C|$) the term $\left(\frac{|C \cap G|}{|C| + |G|}\right)$ would equal to $\frac{1}{2}$ which would make the overall dice loss equal to zero. As for the hyperparameter, λ_{dice}, the LaneSegNet paper [7] set it to 1.0. The next component of the lane segmentation loss uses cross-entropy for semantic segmentation. The cross-entropy loss is computed as shown by Eq. 37.

$$\mathcal{L}_{ce} = -w_{yn} \log\left(\frac{e^{x_{n,yn}}}{\sum_{c=1}^{C} e^{x_{n,c}}}\right) \tag{37}$$

The cross entropy loss will compare each pixel individually to the accepted ground truth. The w_{yn} is a weighting coefficient that corresponds to each ground truth value. This is so that each individual pixel will be weighted the same and therefore have equal learning in the loss function. In the LaneSegNet paper [7], the λ_{ce} is set to 1.0. However, the scaling factor for the entire segmentation loss, λ_{seg}, is set to 3.0. The next component of the loss function determines classification. LaneSegNet uses focal loss, as defined in Eq. 38, to determine the loss between the candidate class label, $\tilde{C}_{\hat{\sigma_i}}$ and the predicted class label, C_i.

$$\mathcal{L}_{cls} = \sum_{i=0}^{N-1} \mathcal{L}_{focal}(\tilde{C}_{\hat{\sigma_i}}, C_i) \tag{38}$$

With the loss function defined, computing the lowest cost follows that of the DETR architecture. LaneSegNet has a set number of output candidates N. Each of these candidates, N_i, need to be matched to their corresponding ground truth values. The easiest form of matching is pair wise matching where the order in which each element comes in the set of candidates directly corresponds to the order in which each element comes in the ground truth set, G. However, this

may not always yield the lowest cost. By using bipartite matching (Eq. 23) each element in the set of N will one-to-one match with a element in the ground truth set G for the lowest possible cost. However since the set G will vary depending on the amount of lanes in the scene there will be cases were the fixed set N will have more elements than the ground truth set G. In that case, the excess elements in the set N that are not matched to an element in G will be assigned a no object class. This no object class will result in no lane being drawn in the output of the architecture.

J. Evaluation Metrics

We used the mean average precision (mAP) metric to evaluate our model. mAP is used to pair the Precision and Recall metrics, which are different ways of measuring the accuracy of predictors. Precision represents the total number of true positive predictions made by the model divided by the number of true positive predictions plus the false positive predictions. Similarly, Recall represents the total number of true positive predictions made by the model divided by the number of true positive predictions plus the false negative predictions. Since our model predicts the location and shape of lane features in the BEV plane, determining the success of the predictions when compared to the groundtruth annotation is slightly more challenging, and the classic method of using intersection over union (IOU) is not scalable to our purposes, because lane lines are naturally thin and long features. For example, a lane line prediction that perfectly matches the shape and curvature of the groundtruth line, but is predicted five pixels to the left, will be assigned 0% accuracy due to there being no overlap.

As suggested by the OpenLane-V2 dataset, we employed a new method for determining geometric similarity of lane lines: Fréchet distance [1]. Fréchet distance, defined in Eq. 39, measures the similarity between two curves by computing the minimally sufficient distance between all sequences of point pairs between the curves.

$$F(G, D) = \min_{L \in C(G,D)} ||L|| \tag{39}$$

Where $G = [g_1, ..., g_n]$ is a vector of points from the groundtruth lane line, $D = [d_1, ..., d_n]$ is a

vector of points from the detected lane line, $C(G, D)$ is the set of all pairing sequences between points in G and D, $L = [(g_1, d_1), (g_2, d_2)...]$ is a possible pairing sequence from $C(G, D)$, and $||L||$ is the largest distance between a pair of points in a possible sequence L. The Fréchet distance computation iterates over all possible point pairing sequences between G and D, and find the sequence which results in the minimally sufficient distance between all point pair in that sequence. Essentially, it will find a scalar value representative of the geometric distance between the detected and groundtruth lane lines, determining their geometric similarity. Naturally, a low Fréchet distance means the lane lines are very similar in location and curvature. This geometric similarity metric is used to determine whether or not a lane line prediction should be a true positive, or false positive. To determine these values, and their negative counterparts, the average across three thresholds $\{1.0, 2.0, 3.0\}$ for the Fréchet distance is computed.

Once the necessary counts have been determined, mAP is calculated using average precision (AP), which is the area under the precision-recall curve in Eq. 43.

$$Precision = \frac{TP}{TP + FP} \tag{40}$$

$$Recall = \frac{TP}{TP + FN} \tag{41}$$

$$AP = \int_0^1 Precision(Recall) d(Recall) \tag{42}$$

$$mAP = \frac{1}{C} \sum_{i=1}^{C} AP_i \tag{43}$$

Where AP_i is the average precision of class i, and C is the number of classes.

K. Results

This semester we were able to successfully implement a replication of the LaneSegNet network this semester, inspired by the official implementation in the paper [7]. Our model was trained for 30 epochs and has been able to produce quite promising numerical and visual results. Although the official implementation was only trained for 24 epochs, our model was not quite able to surpass the accuracy of the official implementation. However, the numerical metrics and visual outputs are not too far behind. The numerical metric we used for evaluation was mAP, as described in Section 7.10, in order to compare our output accuracy against that of the official implementation, which uses the same metric. After 30 epochs, our model was able to achieve a mAP of 23.5, which is lower than the official implementation at 32.6. Fig. II further contextualizes our metric results, with checkpoints at every multiple of 10 epochs.

	Epochs	mAP
Official LaneSegNet	24	32.6
Our LaneSegNet	10	20.2
Our LaneSegNet	20	22.4
Our LaneSegNet	30	23.5

TABLE II: Metric Comparison of LaneSegNet Implementation.

Our accuracy is lower than the official implementation, and we believe this is due to the fact that computation constraints prevented us from running more than 10 epochs at a time. So, in order to reach 30 epochs, we saved the weights file after each checkpoints and attempted to resume training from those weights. However, looking at the loss curve in Fig. 31, it appears that resuming training at these checkpoints causes some hiccups to occur where the loss jumps up before resuming its decline. It is possible that resuming our training at various checkpoints is periodically stagnating the learning, and a that continuous training session may result in a smoother loss curve and higher mAP.

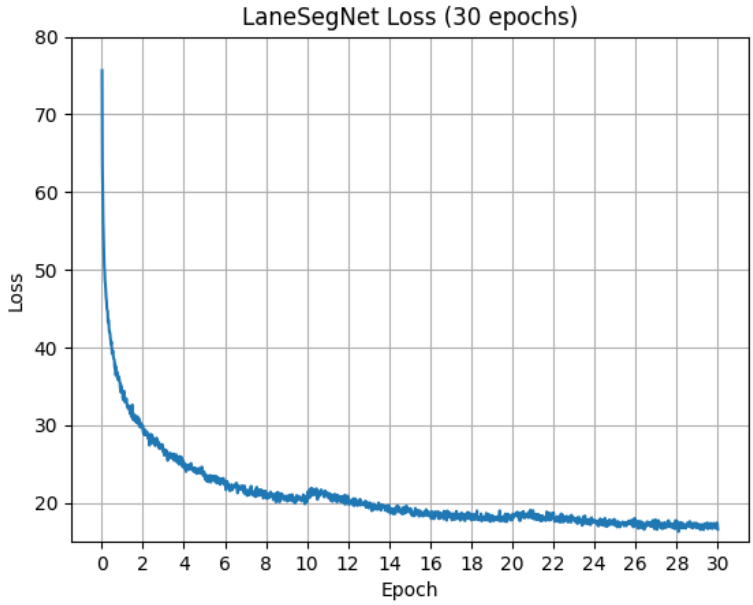

Fig. 31: Our Network's Loss after 30 Epochs.

The results of the model can be further contextualized through a visualization of the model outputs shown in Fig. 32 - 34. All these images are taken from the isolated test subset of the OpenLane-V2 dataset, for the purpose of measuring generalizability. As the visual results show, our model is able to capture the general structure of the road, and can accurately determine the locations of intersections. However, the model struggles to accurately predict the number and shapes of any crossing lanes in the perpendicular road. As the model begins its 15-second drive from the bottom of the BEV plane, it has more information about the road it is traversing as opposed to the road it is simply crossing.

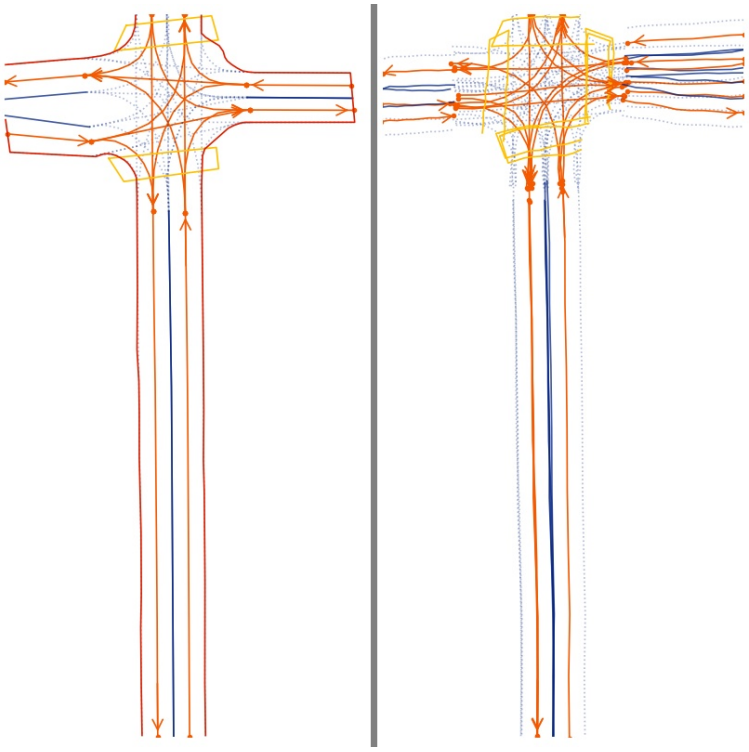

Fig. 32: Groundtruth (Left) and Model Output (Right)

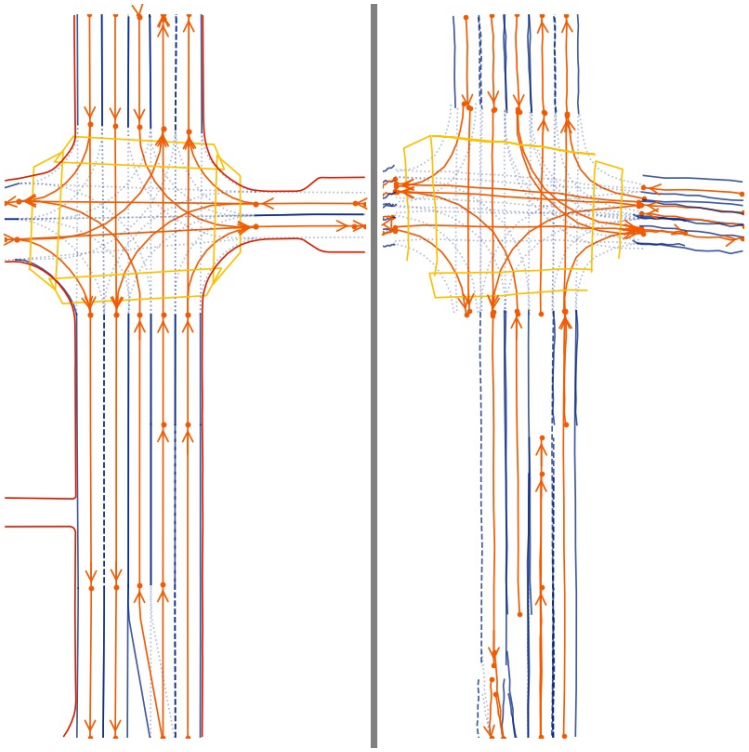

Fig. 33: Groundtruth and Model Output

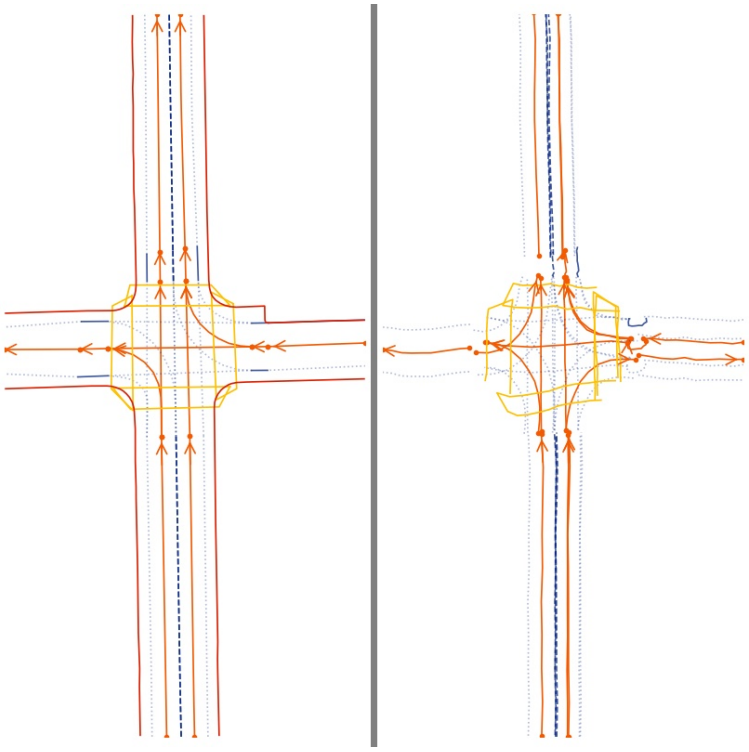

Fig. 34: Groundtruth and Model Output

8. Future Work

One of the biggest problems with the LaneSegNet architecture we followed this semester is the sheer size of the network, which causes training time to be quite long. Some ideas for future work on this project could involve experimenting with various optimization techniques to try and reduce the network size. One component of the network we have investigated optimizing is the ResNet-50 feature extractor, which can be reduced by employing a shallower network with less convolutional layers to reduce computation time. Additionally, some papers describing a technique called "channel pruning" could achieve a similar optimization by dropping out the less relevant channels in deeper convolutional layers. Another optimization approach would involve the transformer components of LaneSegNet, specifically the encoder and/or the decoder. These control flows are repeated multiple times, so it is possible a reduction in repeats would reduce computation cost without sacrificing too much accuracy.

9. Conclusion

The results obtained this semester are promising, as our implementation of the LaneSegNet network is able to perform quite effective detection, prediction and classification of lane lines in the BEV plane. Despite the lower accuracy our model achieved when compared to the official implementation, we believe such a difference would be alleviated without our training constraints. The numerical and visual results are both quite promising. Our model's mAP got close to the official implementation of 32.6, and visually, it captures the general structure of the road along with mostly accurate semantic and structural information about the lanes. Despite our model's difficulty obtaining straight lines and correctly identifying the number of lanes in a given road, we feel confident that our work this semester provides a very strong foundation for future work in lane topology generation, that has potential to be further improved by computation and accuracy optimizations. In summary, our final LaneSegNet model was able to achieve the desired task, with some potential for improvement in future works. These results are satisfactory to us, and we believe the work will be very useful for the team to continue to work on in the future.

REFERENCES

[1] H. Wang, T. Li, Y. Li, L. Chen, C. Sima, Z. Liu, B. Wang, P. Jia, Y. Wang, S. Jiang, F. Wen, H. Xu, P. Luo, J. Yan, W. Zhang, and H. Li, "Openlane-v2: A topology reasoning benchmark for unified 3d hd mapping," 2023.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," Dec 2015. [Online]. Available: https://arxiv.org/abs/1512.03385

[3] F. Yu, H. Chen, X. Wang, W. Xian, Y. Chen, F. Liu, V. Madhaven, and T. Darrell, "Bdd100k: A diverse driving dataset for heterogeneous multitask learning," 2018.

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," Jun 2017. [Online]. Available: https://arxiv.org/abs/1706.03762

[5] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," Oct 2020. [Online]. Available: https://arxiv.org/abs/2010.11929

[6] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, "End-to-end object detection with transformers," May 2020. [Online]. Available: https://arxiv.org/abs/2005.12872

[7] T. Li, P. Jia, B. Wang, L. Chen, K. Jiang, J. Yan, and H. Li, "Lanesegnet: Map learning with lane segment perception for autonomous driving," 2023. [Online]. Available: https://arxiv.org/abs/2103.14030

[8] Z. Li, W. Wang, H. Li, E. Xie, C. Sima, T. Lu, Q. Yu, and J. Dai, "Bevformer: Learning bird's-eye-view representation from multi-camera images via spatiotemporal transformers," 2022. [Online]. Available: https://arxiv.org/pdf/2203.17270.pdf

[9] X. Zhu, W. Su, L. Lu, B. Li, X. Wang, and J. Dai, "Deformable detr: Deformable transformers for end-to-end object detection," 2021. [Online]. Available: https://openreview.net/pdf?id=gZ9hCDWe6ke

10. Appendix

A. *William Stevens*

This was my fifth semester in VIP-IPA, and my fourth semester on the Lane Detection team. We had a very strong and knowledgeable team this semester, so we took on a very challenging project this semester. With me having the most experience on the team, I took responsibility for planning our team's goal and directing us towards achieving it. This semester, I spent most of my time researching the various concepts introduced in the LaneSegNet paper [7], building off of my experience with basic transformers from last semester. These concepts, all used together in the LaneSegNet architecture, include attention mechanisms like Deformable Attention and its descendants Temporal Self-Attention, Spatial Cross-Attention, and Lane Attention. Understanding these attention mechanisms and how they fit into the LaneSegNet architecture also required some supplementary research into Deformable-DETR [9], which is what first inspired BEVFormer and LaneSegNet. Fortunately, most of our team was familiar with DETR due to our work last semester, so we were able to collaborate and help each other work through these concepts. Once we were comfortable with all these concepts, we worked on implementing them in code, following the general network structure of the official LaneSegNet implementation. I contributed to this code development, along with running and monitoring much of the training sessions with Gaurish's help from his strong experience in training machine learning models.

On a more weekly basis, I directed the work of the team during our virtual meetings outside of normal class time, as well as planning and contributing to the weekly presentations. In addition, I designed and created our project poster for participation and presentation in the Undergraduate Research Conference. Within this report, I was responsible for writing the abstract, introduction, dataset, LaneSegNet predictor/evaluation/results, future work and the conclusion. I also adapted most of the Neural Network Fundamentals, CNN, and Detection Transformer sections from last semester's report to fit into the context of this semester.

B. Vishal Urs

This semester was my second and final term in the VIP team. Since most of the team was already familiar with each other, it was not too challenging to continue building from where we left off in the previous semester. We concluded the previous semester with our Detection Transformer (DETR) architecture and aimed to work on a more complex and defined architecture called LaneSegNet this semester. We revised our work on CNNs and DETR. I had not worked with transformers to a significant extent in the previous semester, but this semester's research was strongly focused on transformers and the attention mechanism. I was able to grasp the essential concept of self-attention which has been elemental in helping me break down modifications to the attention mechanism. I spent time developing my own implementation of a Vision Transformer from scratch before indulging into developing the LaneSegNet model. LaneSegNet built on ideas from a combination of other papers and we went through numerous other architectures like ResNet, DeformableDETR, and BEVformer to build the basis of knowledge needed to break down LaneSegNet. Through this experience, I have been able to learn a lot and connect different ideas together to develop an understanding of how the lane segmentation is done. We implemented a version of LaneSegNet and trained it on the OpenLaneV2 dataset. Since this dataset comprised of video data, it was very large in memory and took several hours to train. Our results are not perfect; there are some noticeable limitations in our results. However, considering that the original paper used eight GPUs and we only had one, our results are very promising. Our mAP scores are not as high as the original paper, but they are higher than we expected. On the weekly basis, I made consistent code submissions relative to my work and contributed to the slide presentations as well. For this report, I worked on the LaneSegNet introduction, Deformable Attention, and Temporal Self-Attetion sections. Through this team, I've had the opportunity to help my team-mates when needed and learn from them to progress my growth in the field of image processing and machine learning.

C. Karthik Selvaraj

This was my second semester participating in the VIP Image Processing and Analysis team for senior design. Last semester, the team was focused on building the Detection Transformer (DETR) architecture to perform object detection tasks on dashcam images. This is when I began to learn more about transformers and why they are used in computer vision applications. The team started the semester by searching for architectures to perform other lane detection tasks; this is when the team found the LaneSegNet architecture and the task of lane mapping. The LaneSegNet research paper utilizes many different architectures, so the work of understanding these different parts of the whole was distributed. The encoder module of the LaneSegNet architecture stems from the BEVFormer architecture. I spent a lot of time understanding the two attention mechanisms used in BEVFormer: Temporal Self-Attention and Spatial Cross-Attention. Additionally, I was also responsible for looking into the LaneSegNet decoder architecture and understanding its components, including Lane Attention. Once the team had gained a solid understanding of these mechanisms, the LaneSegNet architecture was written in code and trained on the OpenLanev2 dataset. I wrote the code for the Spatial Cross-Attention mechanism, the self-attention in the LaneSegNet decoder, the Lane Attention mechanism, and the feed-forward network that adjusts reference points in the decoder. The team had to meet in person to collectively set up the dataset and initial training process. After hours of training, we obtained very sub-par results compared to the research paper, but our evaluation metrics and visual results were much better than originally anticipated. I had some difficulties keeping up with my code submissions and documentation this semester and would like to be more on top of my work in future semesters, but I was still able to contribute to the team's work and success this semester. In this report, I wrote the sections for Spatial-Cross Attention, the LaneSegNet Encoder: BEVFormer, Lane Attention, and the LaneSegNet Decoder. I have had the opportunity to grow and learn a lot these past two semesters. I am looking forward to diving deeper into image processing next semester by tackling other problems in computer vision with my teammates.

D. Gabriel Torres

As freshman in first year engineering, this was my second semester on VIP-IPA and the Lane Detection Team. This semester we built on the work we had achieved last semester. Last semester we implemented a detection transformer architecture (DETR) to detect and classify objects when driving. However, this semester we wanted to expand beyond normal object detection and try to train an architecture that could create topological information, something similar to a bird's eye view (BEV). This BEV would be created from multiple camera angles on a car that created a 360° view. We decided to model our architecture after LaneSegNet to create the BEV. I looked into how the LaneSegNet loss worked and found out that it was very similar to the DETR loss. Both architectures used the hungarian algorithm but differ in their loss functions. I noticed that in both architectures to compute the hungarian algorithm they used a function from numpy. However since numpy does computations on the CPU, I wondered if I could make a hungarian alogrithm on GPU. Currently on pytorch, there is no function that emulates the hungarian algorithm, so I went ahead tried to create one using python and pytorch. I was able to create a working version of the hungarian algorithm but to my surprise, it actually took longer than the CPU version from numpy. One of the reasons why is probably because I used two for loops and a while loop in python to make the algorithm which is never a good sign. Even though my own version of the hungarian algorithm did not pan out, I still learned a lot about how the algorithm works and how it applies to the LaneSegNet loss. As a team, we were able to train our architecture to get relatively decent results. However, getting the model to work in a real time scenario is something we are currently looking into. Being on this team I had the opportunity to grow my understanding of how many mathematical concepts (derivatives, matrices) can be applied to the field of image processing and neural networks to create architectures that can detect objects or infer topological information.

E. Gaurish Lakhanpal

This was my first semester in VIP and on the lane detection team. Since I was new to the VIP group and image processing and analysis, I spent much of my time learning the fundamentals required to understand the various ideas and architectures used within LaneSegNet. I began by watching Dr. Delp's intro videos which taught me how how images are taken, and how mathematical concepts such as cross-correlation/convolution can be used to process images. With this knowledge, I dug deeper into algorithms used for image processing, beginning with neural networks and gradient descent/backpropagation. Once I'd understood neural networks, I learned about convolutional neural networks which are type of neural network that can learn convolutional filters and is especially effective at extracting information from images. Having learned about basic neural networks and CNNs, I looked into the shortcomings of these techniques and discovered how issues such as the vanishing gradient problem could arise. Investigating the shortcomings led me to ResNet which introduced skip connections and proposed a more effective architecture for convolutional neural networks. Notably, ResNet50 (a type of residual network) is an important piece of LaneSegNet due to the ResNet50 backbone in the BevFormer, which is responsible for extracting important features the initial images. I then learned about recurrent neural networks, which process sequential data and serve as an important prerequisite for transformers. Once I understood the basics of RNNs, I learned about the attention mechanism at the core of transformers and LaneSegNet.

On a weekly basis I programmed the ideas that I was learning about. I implemented mathematical ideas such as convolution and activation functions in python and used Pytorch to implement larger model architectures.

Near the end of the semester, I worked with the team to write code for LaneSegNet and train it on the Bluepill and Gilbreth clusters. We achieved decent results, but inference times are too slow to for real-time performance. I enjoyed the opportunity to go beyond basic libraries and look forward to learning more in future semesters. I especially want to learn more about model weight initialization and how loss functions are intrinsically tied to probabilistic distributions.