VIP-IPA: Lane Detection Team

Final Report

Fall 2023

William Stevens, Mykhailo (Misha) Tsysin,

Vishal Urs, Karthik Selvaraj, and Gabriel Torres

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 work, we designed and implemented a multi-task neural network model according to the YOLOv4 architecture and trained on the BDD100k dataset, to give detections and classifications of objects, as well as segmentations and classifications of lane lines, in an image. While last semester's model was successful, there was room for improvement. This semester, we explored potential optimizations including the use of a vision transformer and other structural improvements to last semester's implementation. This process included the research, development, and testing of the transformer concept, which was revolutionized by its applications to natural language processing, specifically in chatbots such as ChatGPT. We focused on its applications to computer vision and object detection to benefit from its cutting-edge attention mechanism functionality, and implemented the concept into our model architecture as well as various other modifications to the model's control flow, allowing for increased performance and efficiency during training. These new developments will allow our multi-task model to perform the three vision tasks for autonomous driving at a lower computation cost while also achieving higher accuracy, thus creating an improved version of the model for future work to build upon.

2. Introduction

The goal of the project for this semester was to continue developing and testing algorithms to perform vision tasks for autonomous driving. The Spring 2023 implementation was successful in its development of a YOLOv4-Multi architecture that was able to perform lane and drivable area segmentation with promising accuracy. However, the object detection component of our network was not able to be completed. So, the main objective of the team this semester was to research and implement a new framework for performing object detection on our dataset. Our plan to achieve this goal started by first finishing the object detection component of last semester's YOLOv4 implementation, and then researching concepts external to YOLO for a new approach to object detection.

3. DATASET

We are continuing the use of the BDD100k dataset [1] from previous semesters because it contains all the necessary labels needed for the three tasks outlined in our goal. The dataset contains 100,000 driving videos and labeled images collected from more than 50,000 rides covering New York City and San Francisco Bay Area year-round with multiple weather and lighting conditions. Due to the diversity of geography, environment, and weather, training on the dataset results in a robust and adaptable model for encountering new environments or changes in seasons. Containing thirteen different object classes shown in Table I and nine different lane classes shown in Table II, the BDD100k dataset encapsulates many of the classes that the model could potentially be tasked to analyze. Another component of the dataset that was utilized for our purposes is the labeling of drivable areas, which is divided into two classes: direct and alternative. Direct drivable area is defined as the lane the vehicle is currently driving on and thus the region where the driver has priority over surrounding cars. Alternative drivable area is defined as any lane that the vehicle is currently not driving on but can do so through changing lanes [1]. The BDD100k dataset with its large size, wide variety, paired and highly detailed labeling, makes it a perfect fit for the success of our training purposes.

Category ID	**Class**
1	pedestrian
2	rider
3	car
4	truck
5	bus
6	train
7	motorcycle
8	bicycle
9	traffic light
10	traffic sign
11	other vehicle
12	trailer
13	other person

TABLE I: Labeled classes for object detection in BDD100k. [1]

Category ID	Class
0	crosswalk
1	double other
2	double white
3	double yellow
4	road curb
5	single other
6	single white
7	single yellow
8	background

TABLE II: Labeled lane categories for lane marking in BDD100k. [1]

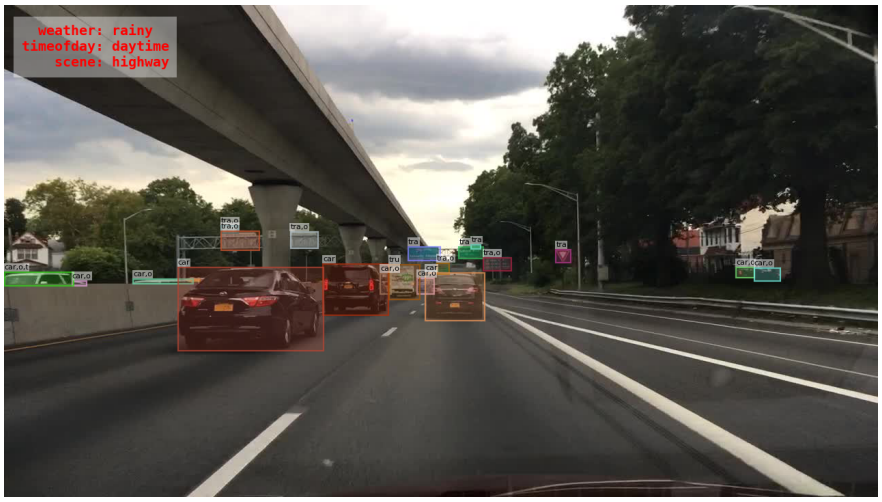

Fig. 1: Object detection visualization during the daytime. [1]

Fig. 2: Drivable area visualization during the nighttime. [1]

Fig. 3: Lane marking annotations during the nighttime. [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. 4.

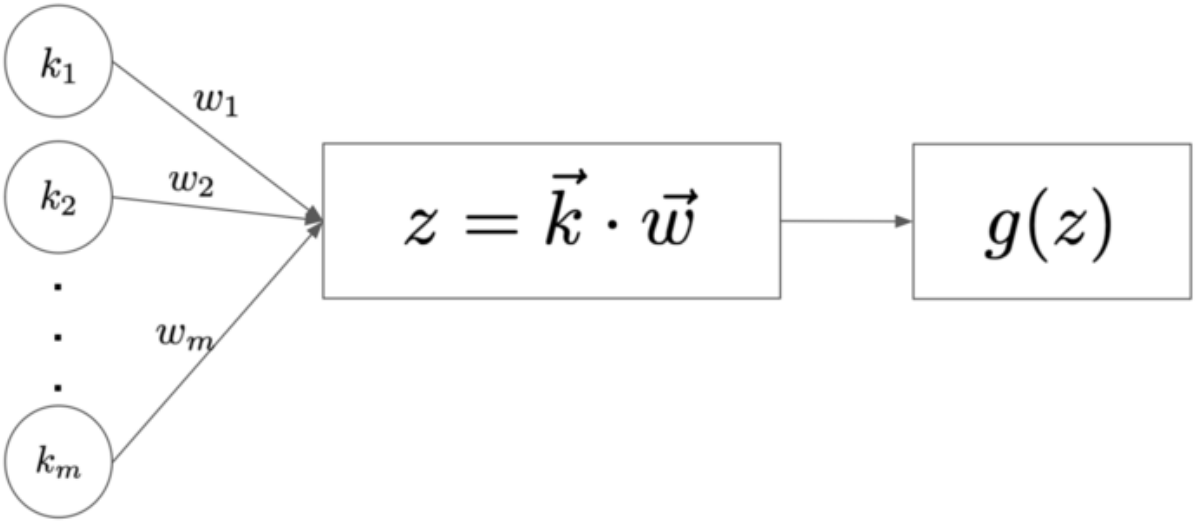

Fig. 4: Single Layer Perceptron

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. 4. 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:

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

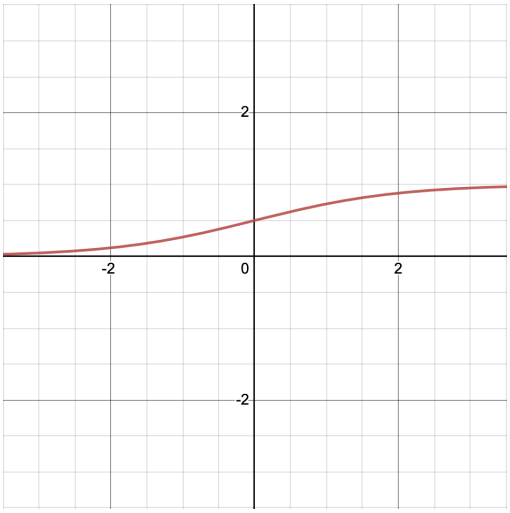

Fig. 5: Sigmoid Function

The Sigmoid function is a popular nonlinear activation function. As depicted in Fig. 5, 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):

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

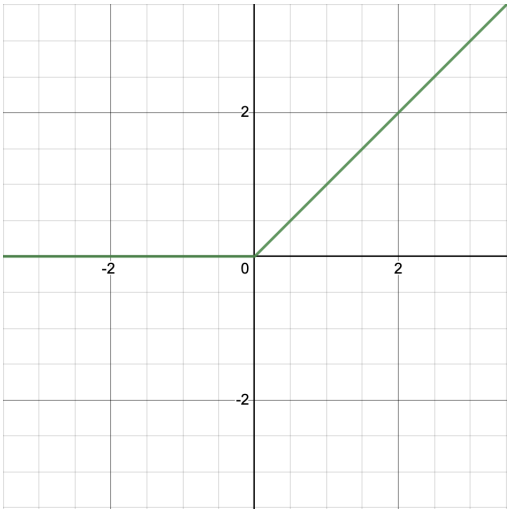

Fig. 6: ReLU Function

Illustrated in Fig. 6, 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):

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

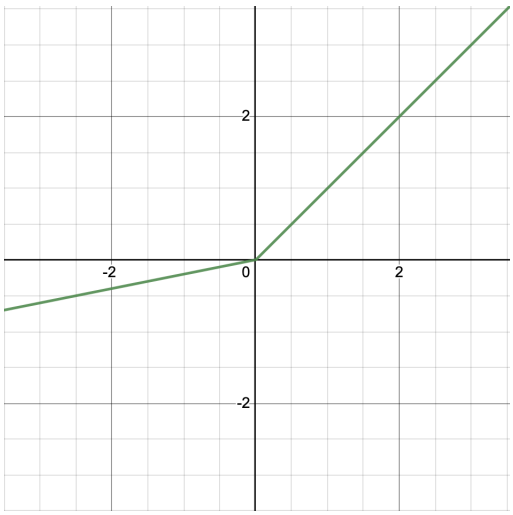

Fig. 7: Leaky ReLU Function

The Leaky ReLU function, shown in Fig. 7, 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."

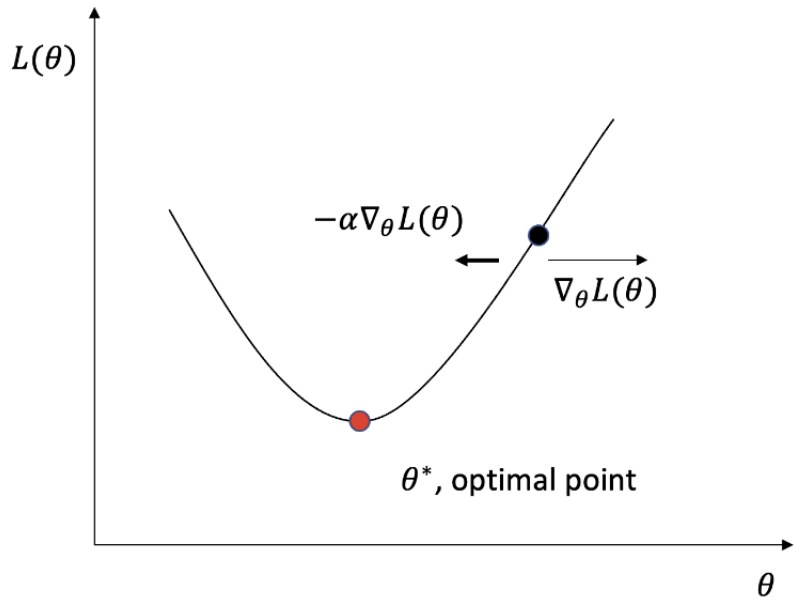

Fig. 8: Gradient Descent Visualized

Fig. 8 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. 8 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. 9 displays an example of a deep neural network.

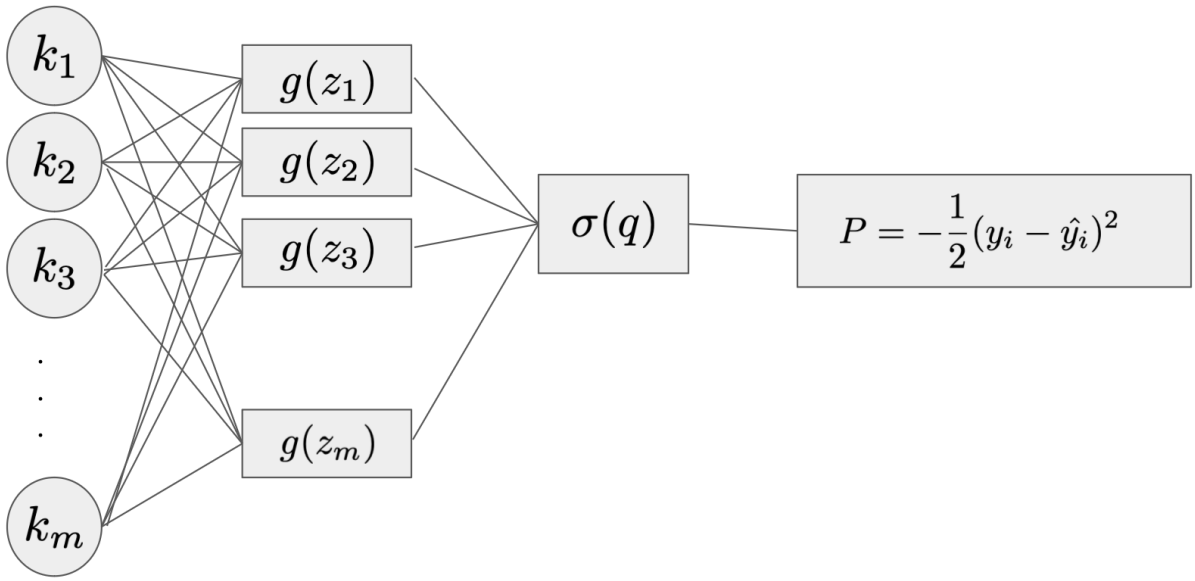

Fig. 9: Deep Neural Network.

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 YOLO

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. 10 provides a high-level illustration of a CNN.

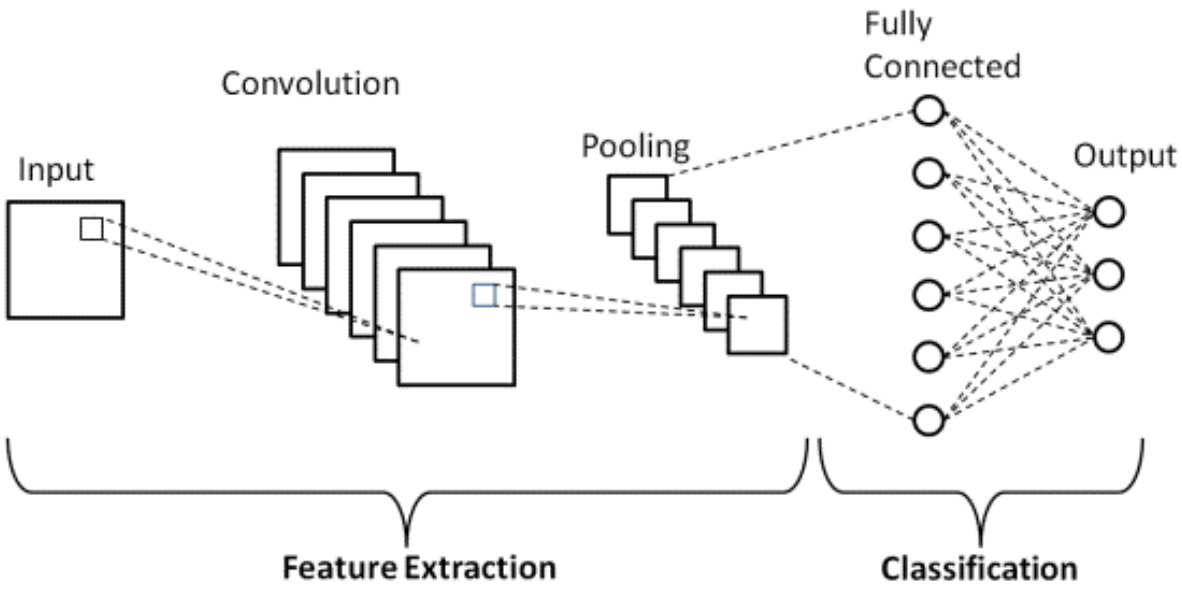

Fig. 10: Convolutional Neural Network (CNN)

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.

The You Only Look Once (YOLO) architecture that we explored comprises multiple Convolutional and Pooling layers. At a higher level, YOLOv1 is a single CNN architecture designed for object detection in images, videos, and even real-time input. YOLO can apply bounding boxes and class probabilities directly from the input image, without the need for region proposals or multiple stages of processing [2]. This makes YOLO a significantly faster object detection architecture compared to previous object detection methods; all this is done with sustaining high accuracy.

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:

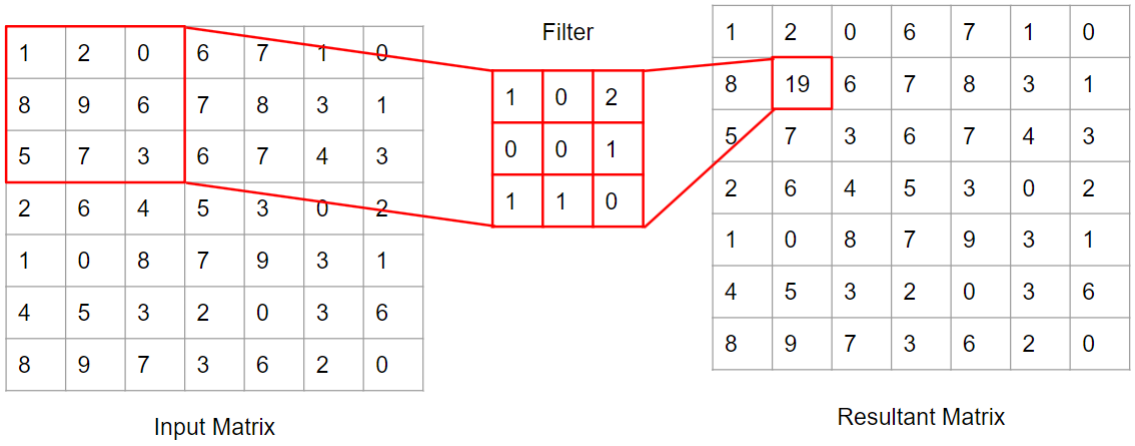

Fig. 11: Cross-Correlation Operation

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:

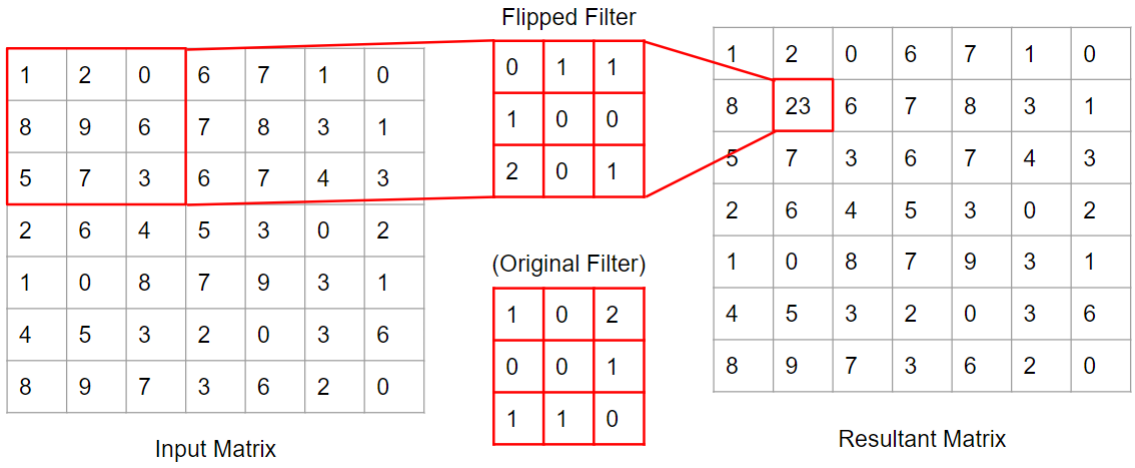

Fig. 12: Convolution Operation

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. YOLOv1 Architecture

YOLOv1 was a breakthrough in the object detection field. It was significantly faster than other popular architectures while maintaining similar accuracy. The YOLO concept models detection as a regression problem. It divides the input image into an $S \times S$ grid and for each grid cell, it predicts B bounding boxes. It also predicts C class probabilities and confidence scores for each box. These predictions are encoded as a $S \times S \times (B * 5 + C)$ tensor. Considering this design for detection, the YOLO architecture is illustrated in Fig. 13.

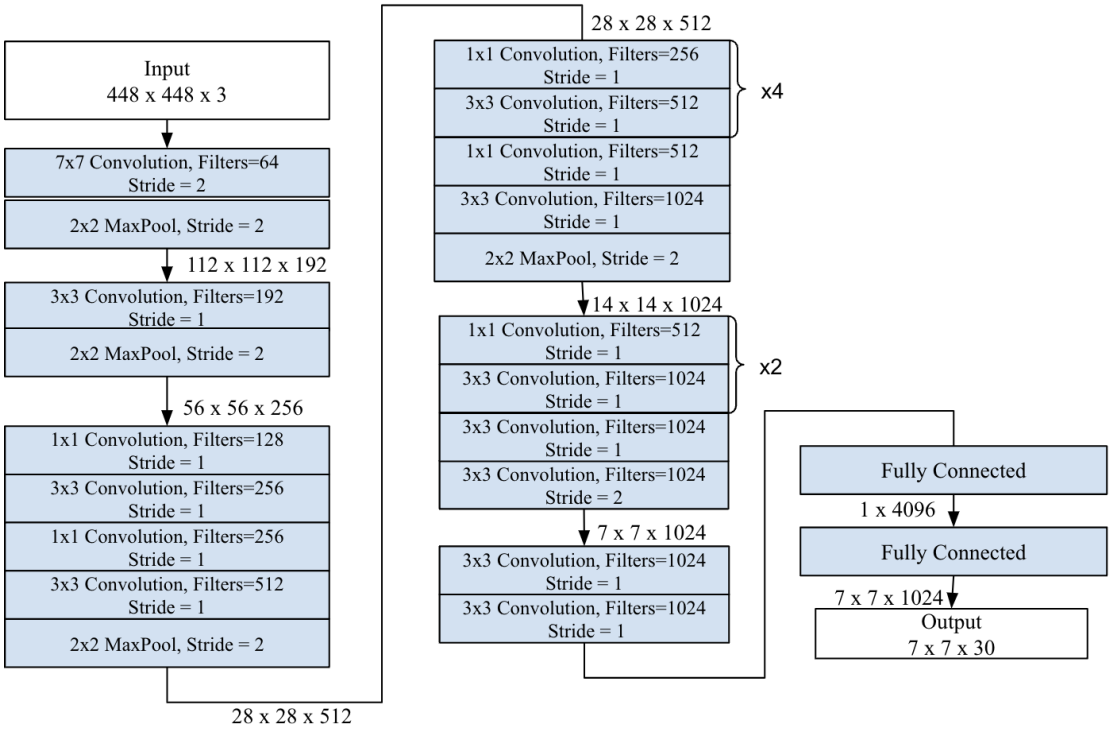

Fig. 13: YOLOv1 Architecture

The detection network has 24 convolutional layers followed by 2 fully connected layers. Alternating 1×1 convolutional layers reduce the dimensionality from preceding layers. Pooling layers are also applied to further reduce the dimensions and to minimize computation costs, especially during backpropagation. YOLO predicts multiple bounding boxes per grid cell. At training time only one bounding box predictor should be responsible for each object. As described in the YOLOv1 paper by Redmon et al [2] one predictor is assigned to be "responsible" for

predicting an object based on which prediction has the highest Intersection Over Union (IOU) with the ground truth. Each predictor gets better at predicting certain sizes, aspect ratios, or classes of objects, improving overall recall.

D. YOLOv1 Loss Function

As previously mentioned, loss functions are used as a metric to determine the performance of the current state of an architecture. This is no different with YOLOv1. However, YOLOv1 uses a more complex loss function to evaluate the architecture's performance. The loss function is:

$$\lambda_{coord} \sum_{i=1}^{S^2} \sum_{j=1}^{B} I_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] + \tag{21}$$

$$\lambda_{coord} \sum_{i=1}^{S^2} \sum_{j=1}^{B} I_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2] +$$

$$\sum_{i=1}^{S^2} \sum_{j=1}^{B} I_{ij}^{obj} (C_i - \hat{C}_i)^2 + \lambda_{noobj} \sum_{i=1}^{S^2} \sum_{j=1}^{B} I_{ij}^{noobj} (C_i - \hat{C}_i)^2 +$$

$$\sum_{i=1}^{S^2} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2$$

In Eq. 21, the first four components of the loss function have two of the same sums. The first sum iterates through each grid cell. Then the second sum iterates through the number of bounding boxes in those grid cells. As for the fifth component, the first sum is the same as its predecessors. However, the second sum iterates through the number of classes defined in the dataset. With an understanding of the summation, we can now take a look at the loss function's components individually.

The first component evaluates the bounding box location and dimensions:

$$\begin{aligned} &\lambda_{coord} \sum_{i=1}^{S^2} \sum_{j=1}^{B} I_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] + \\ &\lambda_{coord} \sum_{i=1}^{S^2} \sum_{j=1}^{B} I_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2] \end{aligned} \tag{22}$$

The $(x_i,\ y_i)$ is the ground truth value of the center box coordinates relative to the grid cell the box was detected in. Similarly, $(\hat{x_i},\ \hat{y_i})$ is the coordinate where YOLOv1 thinks the center box bounding box is relative to the same grid cell. Then the loss of the center coordinates is computed by taking the residual sum squared of the predicted and ground truth coordinates. The constant λ_{coord} is a parameter to determine how much the bounding box loss affects the overall loss function. In the YOLOv1 paper by Redmon et al [2], they set λ_{coord} to be 5. The I_{ij}^{obj} is either a 1 or 0 value depending on whether there is an object in that grid. If there is an object in the grid the value is 1. However, if there is no object in the grid then the value is 0. This is so that a non-existent object for the bounding box loss does not affect the overall loss. As for the second term, this determines the loss for the dimensions of the bounding box. The w_i, h_i are the width and height respectively, of the ground truth bounding box. The $\hat{w_i}$, $\hat{h_i}$ are the width and the height of the predicted bounding box. The reason these values are square-rooted is that the larger the bounding box is, the less impact a smaller difference makes. For example, if we had a ground truth bounding box of 10×10 pixels but the network detected a 5×5 pixel bounding box, the difference is 5 pixels for both width and height. However, if we were to have a ground truth bounding box of 100×100 pixels and the detected bounding box was 95×95 pixels, the difference is still 5 pixels. Although the difference is the same, the 5 pixel difference for the 95×95 pixel box is much closer to the ground truth than the 5×5 pixel bounding box. So, the square root mitigates the loss of larger bounding box dimensions.

The next component of the YOLOv1 loss function is the confidence loss:

$$\sum_{i=1}^{S^2}\sum_{j=1}^{B} I_{ij}^{obj}(C_i - \hat{C}_i)^2 + \lambda_{noobj}\sum_{i=1}^{S^2}\sum_{j=1}^{B} I_{ij}^{noobj}(C_i - \hat{C}_i)^2 + \tag{23}$$

In Eq. 23, C_i represents the ground truth confidence value, and $\hat{C}_i$ is the YOLOv1 predicted confidence value. To calculate the loss of both terms the difference squared of the confidence values are taken. However, the second term is configured differently. The λ_{noobj} is a parameter that determines how much the confidence loss affects the overall loss. In the YOLOv1 paper [2] they set λ_{noobj} to 0.5. Just like I_{ij}^{obj}, I_{ij}^{noobj} either takes a value of either 1 or 0. However, if there is no object present then the value of I_{ij}^{noobj} is 1. If there is an object present then I_{ij}^{noobj}

takes the value of 0. The $noobj$ is short for "no object". The reason for the second term is to evaluate the loss when there is no object present. Since YOLOv1 has to predict the B number of bounding boxes in a grid cell, what should happen when there is no object in the grid cell? Well, since we have to predict some sort of object in the grid cell, the confidence value of that object should be zero. So if the $\hat{C}_i$ is greater than zero we can calculate the loss of a non-existent object in the grid cell.

The last component is the classification loss, as shown in Eq. 24:

$$\sum_{i=1}^{S^2} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2 \tag{24}$$

The classification loss is relatively straightforward. The $p_i(c)$ is the class of the ground truth value and the $\hat{p}_i(c)$ is the predicted class. The loss is computed by taking the difference squared of the class values.

E. Evaluation Metrics

Mean Average Precision (mAP):

The mean average precision (mAP) metric is used to evaluate object detection models. 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 28.

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

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

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

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

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.

1) Intersection over Union (IOU):

The Intersection over Union (IOU) metric is used to evaluate the accuracy of predicted bounding boxes compared to the true bounding box location and dimensions. IOU is a measure of the amount of overlap between the predicted and groundtruth bounding boxes.

$$IOU = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{29}$$

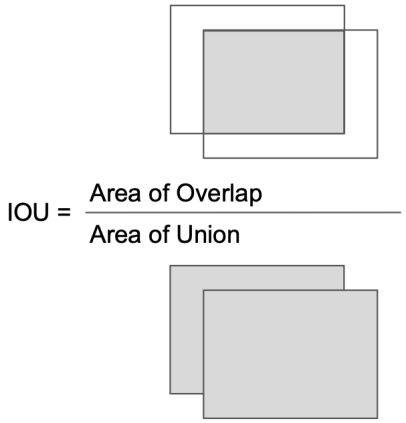

Fig. 14: Visual Representation of IOU

Using Eq. 29, we can compute the accuracy of the predicted bounding boxes compared to the groundtruth bounding boxes. If A is the predicted bounding box and B is the groundtruth bounding box, the resultant IOU calculation will be the metric used to evaluate bounding box accuracy.

2) Mean IOU:

The Mean IOU metric is used to evaluate the accuracy of predicted segmentation compared to the groundtruth segmentation. Since there are ten non-background segmentation classes in the dataset, a vector of ten IOU values is calculated for each image. Each of these vectors are then

averaged to produce a vector of ten mean IOU values, one for each segmentation class. In the case that a class is not present in the image, the corresponding entry for this class and image is ignored in the mean calculation.

6. YOLOv4-Multi

A. *Introduction*

In this section, we will recap the work done by the lane detection team prior to this semester. In Fall 2022, the team developed a model for performing unified object detection and lane segmentation through a combination of the YOLOv3 architecture [3] and the U-Net framework [4]. Following this success, the Spring 2023 team set out to adapt the YOLOv3-Multi model and incorporate various optimizations introduced in the official YOLOv4 paper [5] to improve on the accuracy results obtained by the Fall 2022 implementation. The team aimed to achieve this by improving the computation capabilities of the model through a more efficient feature extractor and activation function, and by increasing the accuracy of the model by incorporating an augmented feature aggregator to allow for a more effective fusion of high-level semantic features and low-level spatial features. A range of concepts was explored, including the Cross-Stage Partial Network, the Path Aggregation Network, the Mish activation function, CutMix data augmentation, and more. Many of these concepts were developed and added to the model to create the complete YOLOv4-Multi architecture, as shown in Fig. 15.

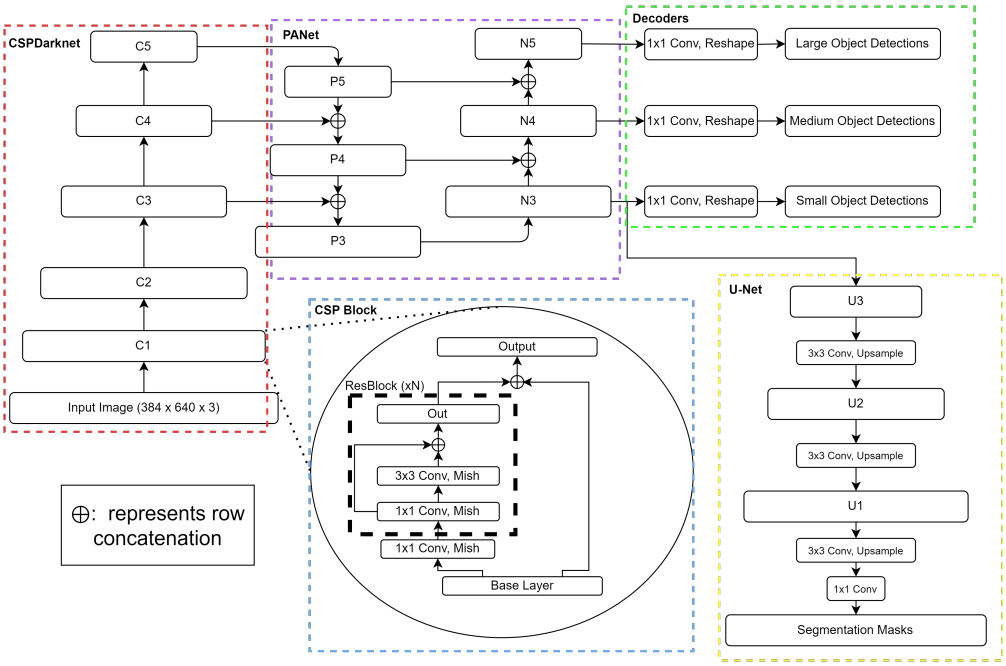

Fig. 15: Complete Model Diagram.

B. YOLOv4-Multi Architecture

1) Mish Activation:

For the activation function of the YOLOv4-Multi model, the Mish function was utilized. The Mish activation function is an improved version of the Rectified Linear Unit (ReLU) function, previously defined in Section 4.2. We define $ReLU(x)$ in Eq 30. The problem with ReLU is that it is not continuously differentiable due to its behavior when $x = 0$.

$$ReLU(x) = \begin{cases} x: & x > 0 \\ 0: & x \leq 0 \end{cases} \tag{30}$$

There is a corner at the origin of ReLU, which means that the derivative must be approximated when performing backpropagation. The Mish activation function, as described in the official paper by Misra [6], was derived by a Neural Architecture Search (NAS) which searches the non-linear function space to determine optimal activation functions. Mish builds upon the Swish function, which is a self-gated version of the classic sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$. Swish $S(x)$ in Eq 31 achieves self-gating by multiplying the sigmoid by the input x, allowing the function to be unbounded above. This feature, which is shared by ReLU, prevents gradient saturation and improves gradient flow.

$$S(x) = \frac{x}{1 + e^{-x}} \tag{31}$$

Swish, unlike ReLU however, is continuously differentiable and non-monotonic (not strictly increasing/decreasing). These two features, shown in Fig. 16, are beneficial as continuous differentiability eases derivative computation and non-monotonicity allows for the preservation of small negative weights, thus preventing model overfitting.

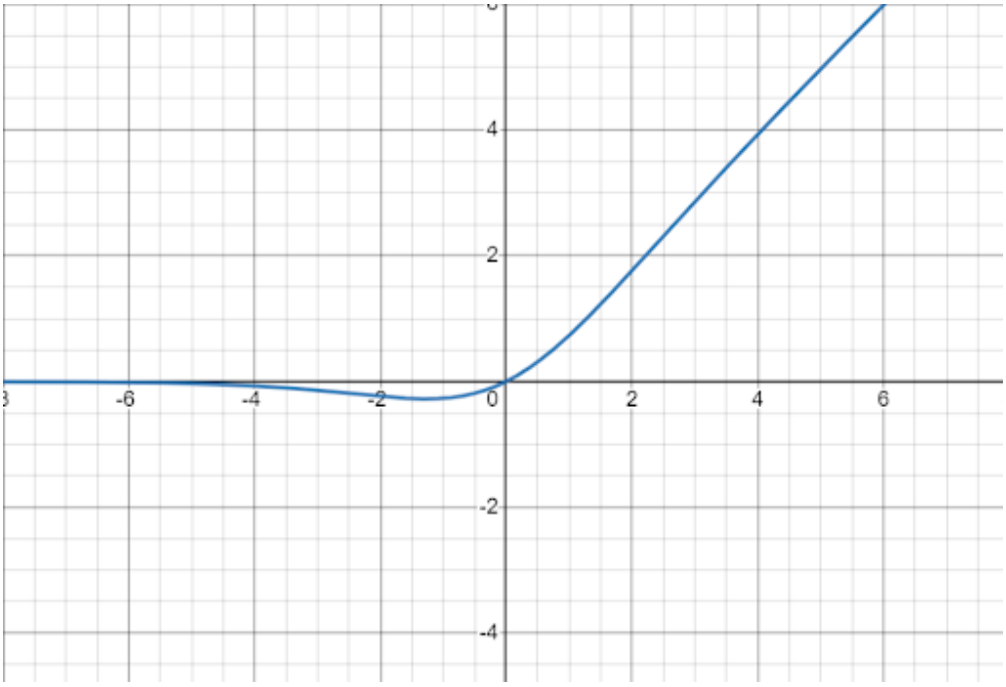

Fig. 16: The Swish function.

While Swish has proven to be a more effective activation function than ReLU, the NAS conducted by Misra determined that the Mish activation function was more optimal. Mish, similar to Swish, is based on the integral of the sigmoid function, also called the SoftPlus function: $\int \sigma(x) = ln(1 + e^x)$. Mish takes the hyperbolic tangent of the SoftPlus function, and multiplies by the original input x, as shown by $M(x)$ in Eq 32.

$$M(x) = x\text{Tanh}(ln(1 + e^x)) \tag{32}$$

These additions to SoftPlus allow Mish to have the same beneficial traits as Swish: continuous differentiability, non-monotonicity, and unboundedness above (See Fig. 17); while achieving better accuracy than Swish.

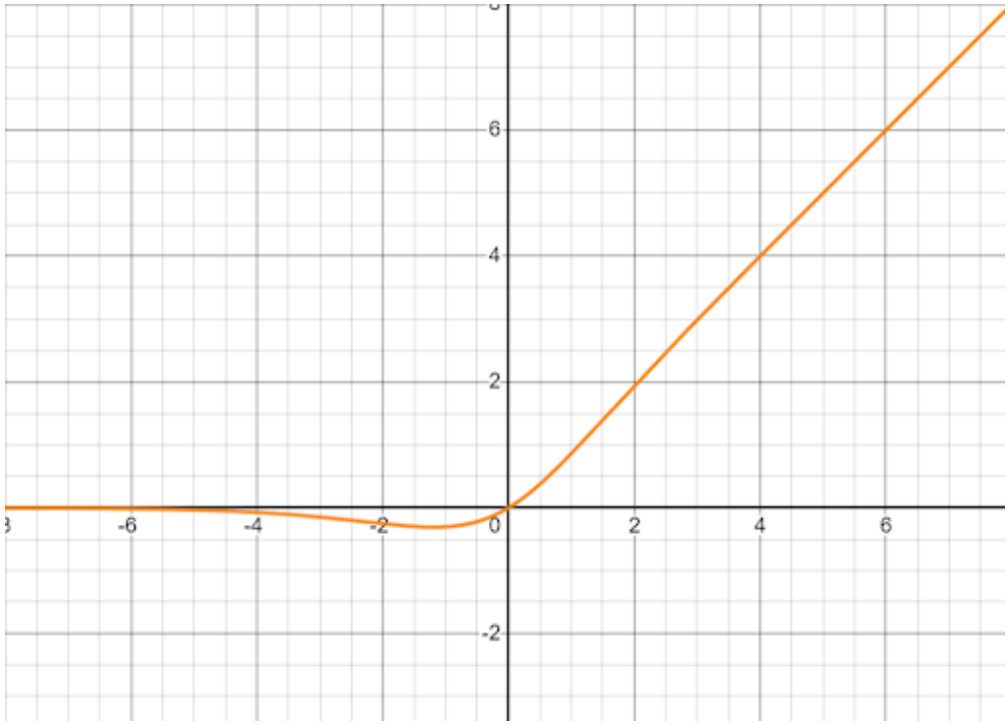

Fig. 17: The Mish function.

2) Cross-Stage Partial Network:

The development of the Cross-Stage Partial Network (CSPNet) as described in the official paper by Wang et al. [7], was motivated by the desire to reduce the computation cost of convolutional neural networks, thus allowing for cheaper and less complex processors to run object detection tasks at reasonable speeds. The main intuition behind CSPNet is that there are many duplicate gradient calculations performed in the backpropagation of standard CNNs. The authors of the CSPNet paper theorized that many of these duplicate calculations could be removed by integrating early feature maps with later ones, in a manner similar to the concept of residual connections.

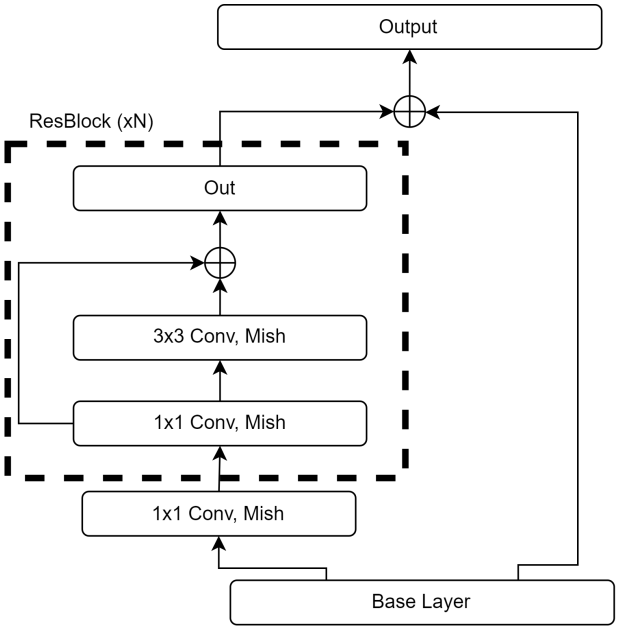

Fig. 18: A Cross-Stage Partial Layer.

The authors designed cross-stage partial connections within the convolutional layers, which partitioned the feature maps of the input into two paths, where one path carried out standard convolutions and activations, whereas the other underwent no operations. These paths, shown in Fig. 18, would later be merged across stages before being passed to the next layer. It was found that generally, the introduction of these extra gradient flow connections allowed gradient computation to skip over duplicate calculations, and resulted in a general reduction of network computation costs by around 20% without any loss of model accuracy.

3) Path Aggregation Network:

The feature aggregator implemented was the Path Aggregation Network (PANet) as described in the official paper by Liu et al. [8]. PANet is an augmentation of the Feature Pyramid Network [9] used in the Fall 2022 YOLOv3-Multi implementation. PANet adds a second "Bottom-Up" path to the end of the standard FPN architecture, and adds additional lateral connections to connect both bottom-up paths with the central top-down path, as shown in Fig. 19. These additional lateral connections and extra bottom-up path allow for low-level features, which often contain important spatial information, to take a shortcut and efficiently aggregate with the rich semantic

information extracted from the high-level features.

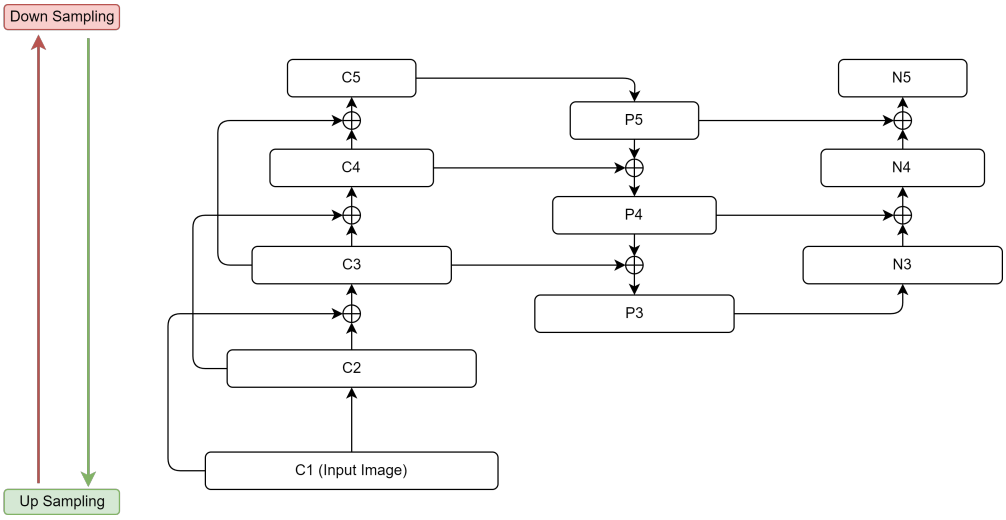

Fig. 19: CSPDarknet with PANet.

4) Detection Decoder:

For transforming the model's output into detections, a similar version of the decoder used in Fall 2022 was utilized. The YOLOv4-Multi model analyzes features at three different scales, obtained from the feature maps in N3, N4, and N5 (See Fig. 15 or 19), in order to detect objects of varying sizes.

To arrive at the desired output matrices necessary for visualizing and quantifying the detections, a series of convolutions and reshapes is performed, as shown in Fig. 20. The N5 layer will have 1024 12×20 feature maps, the N4 layer will have 512 24×40 feature maps, and the N3 layer will have 256 48×80 feature maps. Each of these layers are sent through a 1×1 convolution to pool all their channels into 54 channels. These three outputs are then reshaped to reintroduce the three RGB color channels into the detections, achieving three RGB grids of varying sizes, where each 18-value grid-cell represents a detection at a given grid-cell size. The 18 values signify the five bounding elements for the object (center x-coordinate, center y-coordinate, box width, box height, and objectness confidence), as well as the 13 object class probabilities.

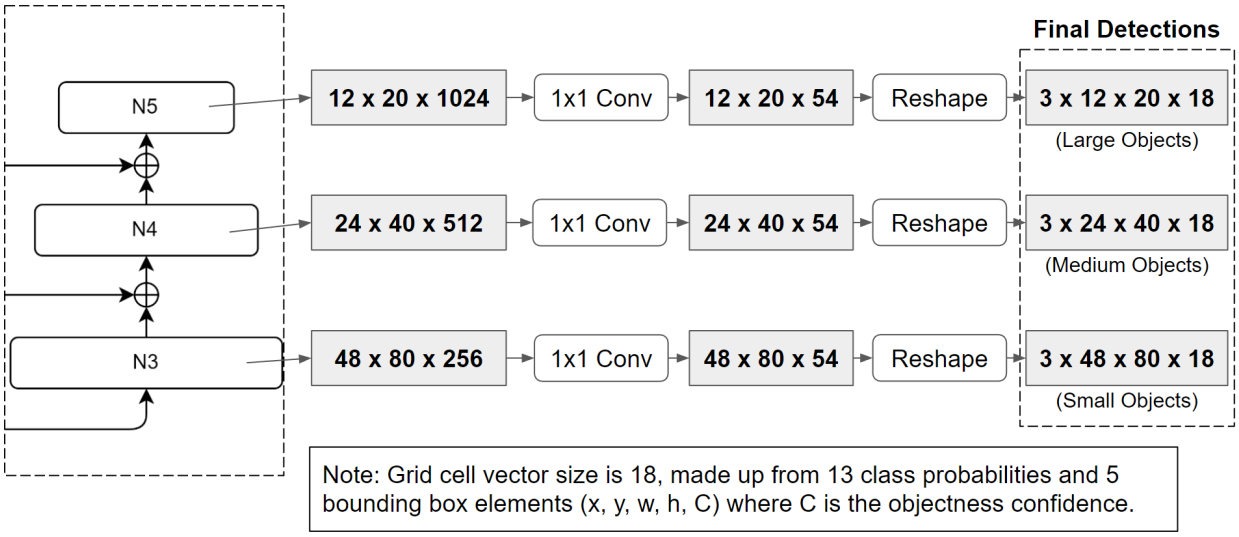

Fig. 20: Detection Decoder.

5) Segmentation Decoder:

As for obtaining segmentations from the model's output, the team utilized the second half of the U-Net architecture developed by the Spring 2022 team. The YOLOv4-Multi model takes the N3 feature maps (See Fig. 15 or 19) and sends a copy to the segmentation decoder to achieve the desired segmentation masks.

The N3 layer's feature maps are sent through the U-Net component as shown in Fig. 21. The U-net component carries out a series of 3×3 convolutions and 2D upsamples to eventually return to the original input image resolution of 384×640. However, U-Net ends with 32 channels, so these channels are pooled down to 12 to represent the segmentation masks for the 12 classes in our dataset (8 lanes and background, 2 drivable areas and background).

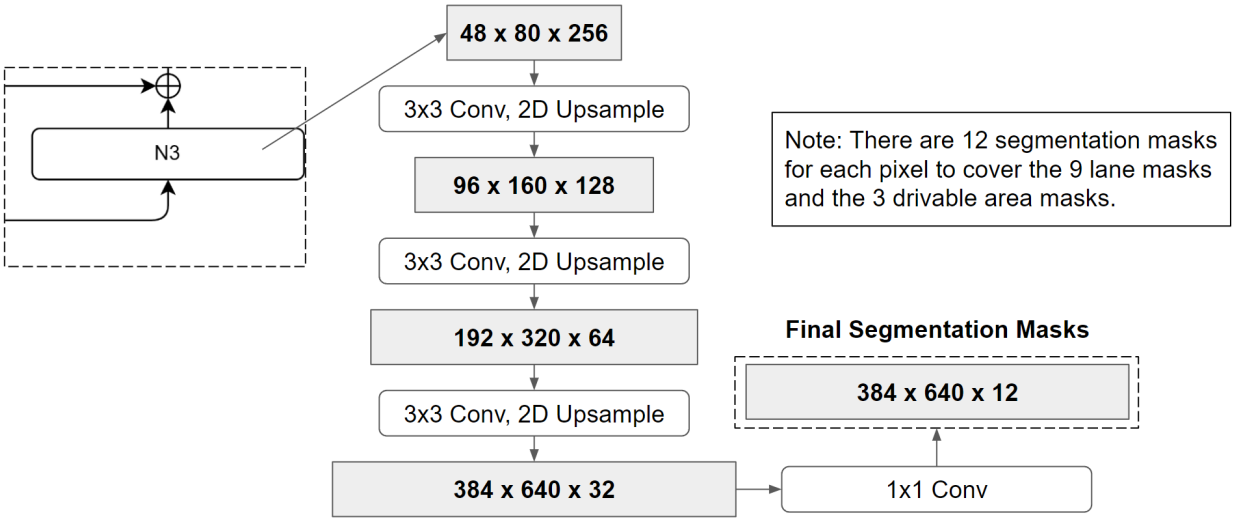

Fig. 21: Segmentation Decoder.

C. Loss Functions

par To quantify the network's object detection accuracy each after iteration, the YOLOv4-Multi architectures used the same two loss functions as used in the Fall 2022 YOLOv3-Multi implementation: Focal Loss and Complete-IOU loss [3].

Focal loss [10] in Eq 33 accounts for accuracy with respect to the classification results. It is an augmentation of standard cross-entropy loss that includes a modulating factor to account for class imbalances in training. For example, the modulating factor will de-emphasize loss on common and easy object classification tasks, and increase emphasis for tasks that appear less frequently, and are thus more difficult [10]. It will effectively balance the loss function based on the frequency of certain classification tasks to ensure that the model dedicates equal learning power to each class.

$$FocalLoss(p_t) = -\alpha_t(1-p_t)^{\gamma}log(p_t) \tag{33}$$

Where a_t is the class balancing factor, γ is the focusing parameter, and p_t is the class probability p if the class matches the ground truth, and $1-p$ if the class does not match the ground truth.

Complete-IOU loss [11] defined by Eq 34 accounts for accuracy with respect to bounding box detection. It is an augmentation of standard IOU loss which calculates the intersection-over-

union between the model's detected bounding boxes and the ground truth data. In addition to computing IOU, Complete-IOU introduces two more factors: the distance between the centers of the detected bounding box and the ground truth, and the consistency of the bounding boxes' aspect ratios [11].

$$C_{IOU} = 1 - IOU + \frac{\rho^2}{diag^2} + \frac{v^2}{1 - IOU + v} \tag{34}$$

Where IOU is the intersection over union of the detection and the ground truth, ρ is the distance between bounding box centers, $diag$ is the diagonal length of the convex of bounding boxes, v is the aspect ratio consistency, w_{gt}, h_{gt} is the width and height of ground truth, w_d, h_d is the width and height of the detection, and $v = \frac{4}{\pi^2}(tan^{-1}(\frac{w_{gt}}{h_{gt}}) - tan^{-1}(\frac{w_d}{h_d}))^2$.

For Fall 2023, Focal loss was maintained for classification accuracy, but augmented by IOU loss as a segmentation accuracy.

IOU loss measures the overlap accuracy between a groundtruth mask and a predicted segmentation mask for a given class. The IOU loss across all classes is shown in Eq 35. It takes the bitwise-and of the two binary masks, and divides by the bitwise-or of the masks. If the divisor is zero, we simply add the bitwise-and of the masks, which will represent a false positive segmentation of a class that does not exist in the image.

$$\mathcal{L}_{IOU} = 1 - \frac{1}{C}\sum_{c=1}^{C} IOU_c \tag{35}$$

$$IOU_c = \frac{|P_c \cap G_c|}{|P_c \cup G_c|} \tag{36}$$

Where C is the number of segmentation classes, IOU_c is the computed IOU for class c (defined in Eq 29), P_c is the predicted segmentation mask for class c, and G_c is the groundtruth segmentation mask for class c.

D. Evaluation Metrics

For evaluating the accuracy of the model's detections and segmentations, the same metrics as the Fall 2022 implementation were used. Intersection over union, and the connected Mean

IOU metric, were used to measure the segmentation accuracy of our model. These metrics are defined in Section 5.5. Since the team was not able to get complete object detection results in Spring 2023, the mean average-precision metric was not needed for evaluation.

E. Segmentation Results

The YOLOv4-Multi model developed in Spring 2023 was trained for 300 epochs with evaluation metric results shown in Table III. The metric used for measuring the segmentation task was mean IOU. The drivable area segmentation results saw the greatest accuracy increase from Fall 2022, jumping from just about 54% to a much higher accuracy of about 90%. However, a persistent issue with the model was the low lane segmentation IOUs. Just like the Fall 2022 model, the YOLOv4-Multi model predicted thick lane lines compared to the groundtruth lanes, which are only a couple of pixels wide. Again, for individual classes, the segmentation accuracy was higher for the more frequently appearing classes, such as single white lanes and road curbs. While most of these lane line accuracy results were not satisfactory, the accuracy increased for nearly all classes, save for the crosswalk class. The results of the YOLOv4-Multi model can be further contextualized through a visualization of the model outputs shown in Fig. 22 - 25.

Fall 2022 (YOLOv3)		Spring 2023 (YOLOv4)	
Segmentation Class	mIOU(%)	Segmentation Class	mIOU(%)
Drivable Area	54.42	Drivable Area	89.92
Lane Predictions (all)	10.4	Lane Predictions (all)	16.87
Single White Line Lanes	25.3	Single White Line Lanes	40.54
Road Curbs	18.8	Road Curbs	32.8
Cross Walks	7.6	Cross Walks	3.59
Single Yellow Lines	6.4	Single Yellow Lines	14.8
Double Yellow Lines	2.1	Double Yellow Lines	5.81
Double White Lines	2.0	Double White Lines	3.7

TABLE III: Comparison of Segmentation Evaluation Results.

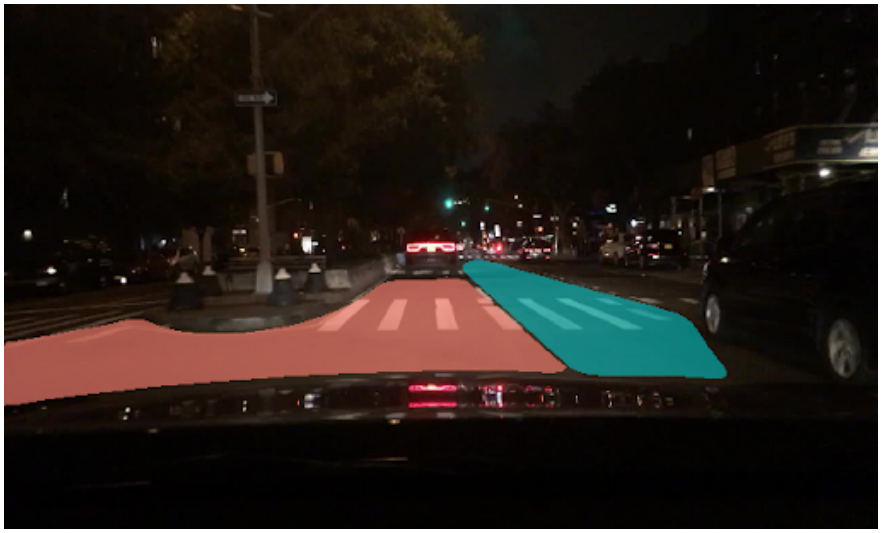

Fig. 22: YOLOv4-Multi Drivable Area Results (1).

Fig. 23: YOLOv4-Multi Lane Line Results (1).

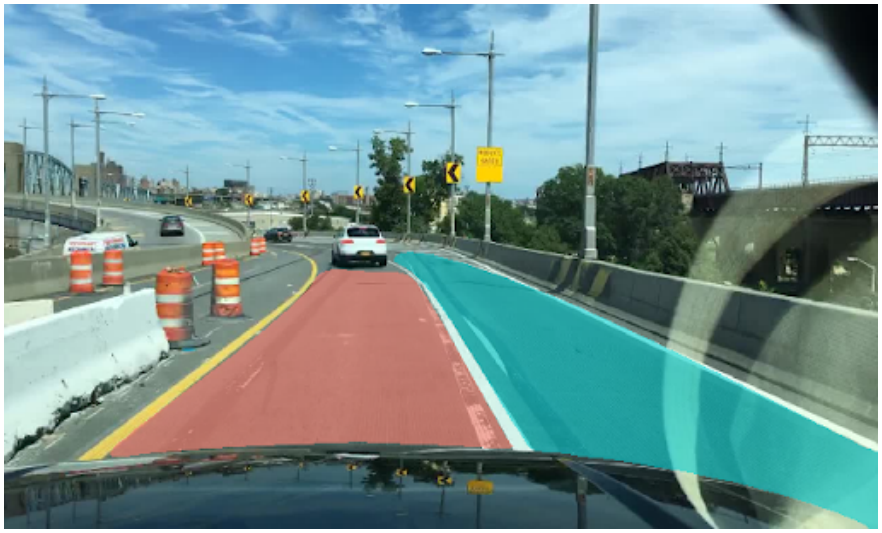

Fig. 24: YOLOv4-Multi Drivable Area Results (2).

Fig. 25: YOLOv4-Multi Lane Line Results (2).

7. Detection Transformers

A. Introduction

As previously mentioned, the main goal for this semester was to to research and implement a new framework for performing object detection on our dataset. Our team set out to achieve this goal by investigating the concept of transformers. Transformers were novel machine learning technique proposed in the 2017 paper "Attention Is All You Need" [12], and have since been instrumental in the development of revolutionary text processing and generation tools like ChatGPT. While such tools are not relevant to our vision task of object detection, recent papers such as "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale (ViT)" [13] and "End-to-End Object Detection with Transformers (DETR)" [14] have demonstrated that the transformer concept can be applied to object detection tasks with similar success. Over the course of this semester, 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. 26, flatten them, and then apply a transformer network described in Section 7.2. While this approach works well for classification tasks, it is sometimes unclear how we can extract features of different sizes for detection. Another problem with ViT is that it did not specify the loss function appropriate for detection. Fortunately, the DETR network was able to solve all of these issues. The paper introduced a different way of feature extraction as well as the loss function appropriate for transformer architectures. So, we ended up adapting the DETR implementation to fit our vision task and dataset, for the purpose of performing unified object detection and classification.

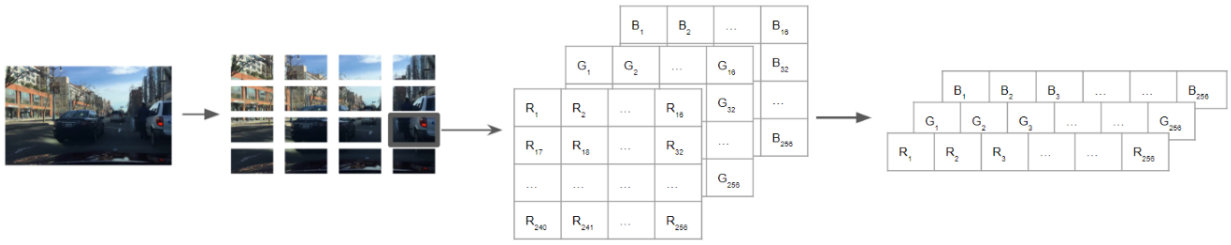

Fig. 26: Patch formation for Vision Transformer

B. Transformer Concept

The original transformer proposal introduced in "Attention Is All You Need" [12] 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 37 below as described in the paper [12].

$$\text{Attention}(Q, K, V) = \text{SoftMax}(\frac{QK^T}{\sqrt{d_k}})V \tag{37}$$

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 38 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.

$$\text{SoftMax}([x_1, ..., x_n]) = [p_1, ..., p_n] \text{ where } p_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{38}$$

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) [12]. This component, as shown in Fig. 27, 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.

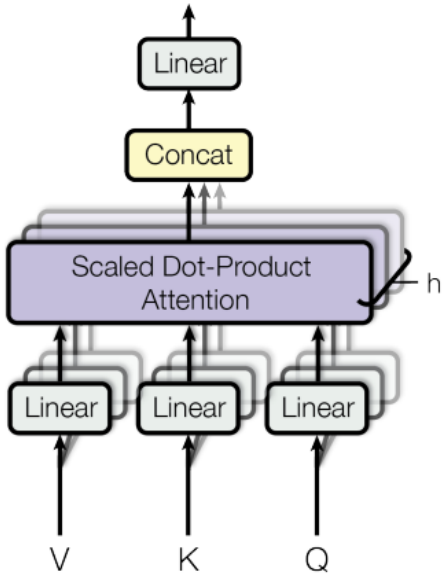

Fig. 27: Multi-Head Self Attention. [12]

C. Transformer Architecture

Having established the notion of Multi-Head Self Attention, we can now use it to build our network. The transformer network was initially proposed in [12], and the original detection transformer paper closely followed the original design [14]. Here, we will quickly go over the general structure of the network and subsequently will talk a bit more about how to convert it to computer vision applications. The structure of the original transformer is presented in Fig. 28.

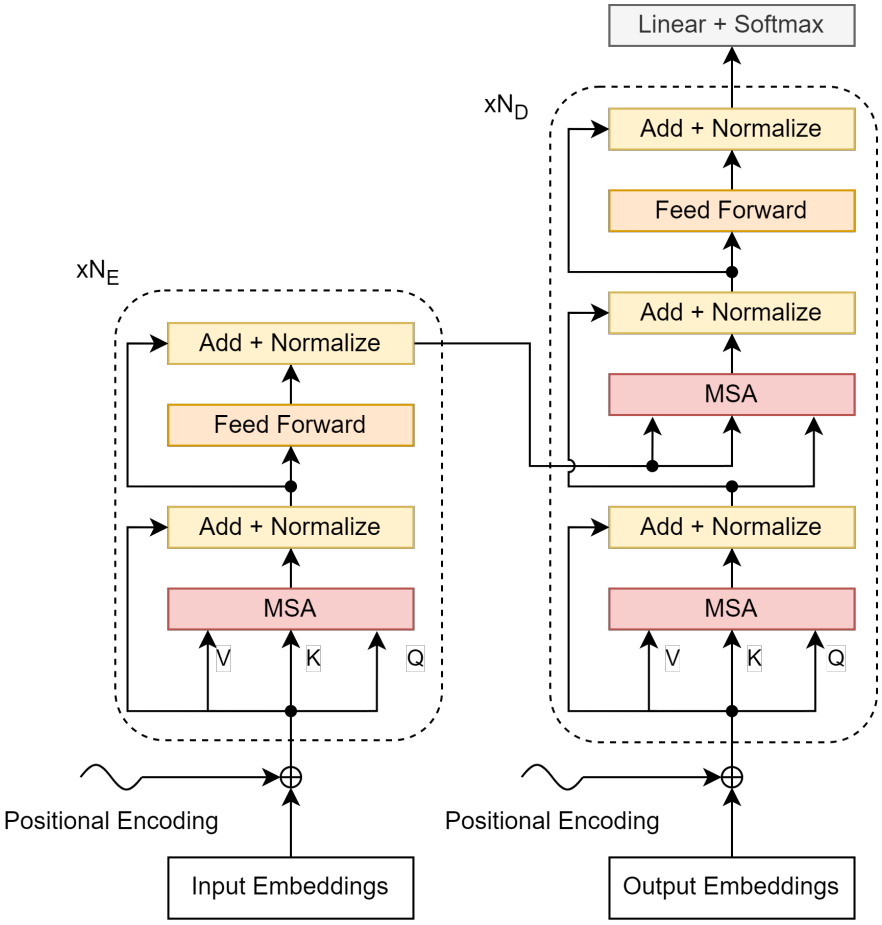

Fig. 28: Original Transformer architecture

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 essentially a "nice" way of assigning vectors to non-uniform data, such as words. In our application, this is not an issue thus we will not focus here on how embeddings work. Think of them for now as raw input to the network. Before going into the encoder block, we add position encoding to our vectors. This is needed to make sure that the network has a way to infer the order of input features. We will talk more about how positional encoding works in our application later.

The first part of the encoder network is the multi-headed self-attention (labeled as MSA). We split the input into keys, queries, and values and look for dependencies between the features. After the MSA block, we use Layer Norm as well as a skip connection which both help with

gradient flow as was shown in earlier sections about ResNet and CSPNet. The last two stages of the encoder block are a simple feed-forward network with one or two layers followed by the normalization and skip connection that was already used before. 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 part of the transformer need to be decoded into our desired output. The decoder network is very similar to the encoder in terms of building blocks. We start by output embeddings (which can be learned or specified by hand), add positional encoding, and proceed toward the decoder blocks. In the decoder block, we have two attention layers. The first one (at the bottom of the picture) is self-attention since we split the same input into keys, queries, and values. And the second is cross-attention. Here, we use the keys and values from the output of the encoder. And the queries are taken from the features running through the decoder. By doing so, our decoder can gradually incorporate the information from the encoder into our final output. Similarly to the encoder block, we finish by applying a feed-forward network. Note that all of the 3 blocks (self-attention, cross-attention, and feed-forward) use layer norm and a skip connection. After we pass through N_D decoder blocks we are ready to convert our information into the desired format. The original paper used a linear layer as well as softmax. But this can be changed based on the application.

D. Transformers for Vision and DETR Architecture

The detailed architecture of DETR as used in our implementation is presented in Fig. 29. As we can see, the general architecture is very similar to the one described in the previous section, but there are some differences. First of all, the transformer network does not take the raw image directly as input. Instead, we use a feature extractor to generate useful features. Such feature extractors can be, for instance, a convolutional backbone. In our implementation, we used a ResNet-50 network pre-trained on the COCO dataset. This makes sure that the transformer accepts only features providing complex information about the image contents.

The second thing that looks different is the way positional encodings are added. Instead of only one encoding at the beginning of each stage (encoder and decoder), we add positional encodings only to the keys and queries of encoder self-attention as well as the keys of the

decoder's cross-attention. Finally, the output of the decoder stage produces N_Q queries each one representing a potential detection. Notice that the network essentially outputs a set of predictions with no particular structure. That might lead to problems when it comes to evaluating the loss – we simply do not know which box from the ground truth to compare our results to. This problem is solved with bipartite matching loss, which will be described later.

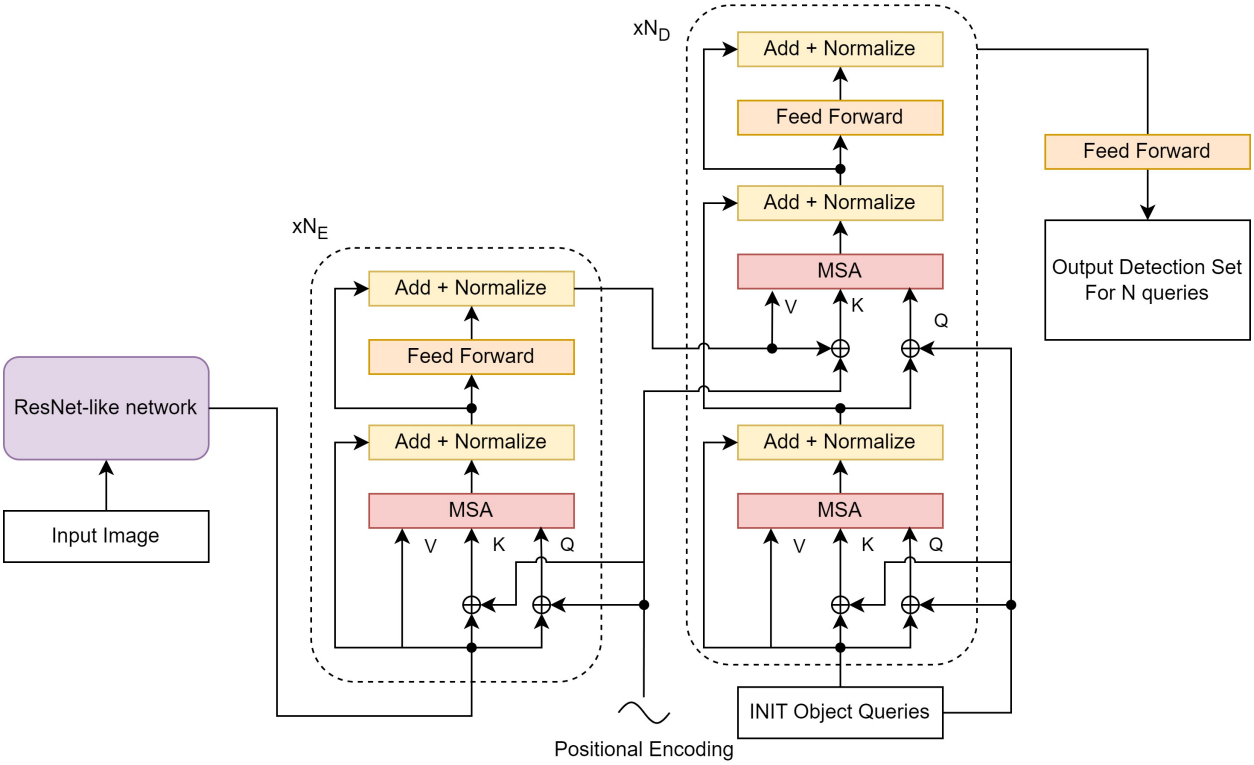

Fig. 29: Detection Transformer Full Architecture

Now, let us talk about some implementation details regarding the network which were not clear to us at the beginning.

1) Attention masking: The detection transformer, unlike most CNN models we described earlier is capable of working with images of different sizes. To accomplish this, we choose a canvas size of (W_c, H_c) where $W_c = \max_{i \in \text{images}} W_i$, where W_i is the width of image i, similarly for H_c. Then, we fit our image in the top left corner and pad everything else with zeros. Now, we have an image of size (W_c, H_c) with some pixels being irrelevant. To make sure that these values do not propagate through the network, we use attention masking. Whenever we are propagating a value through the network we will propagate a pair $(\textbf{values}, \textbf{mask})$, where **mask** will have

ones (or zeros if you like) whenever the value is irrelevant. Let us see how the masking will work in the attention mechanism. Attention, computed in (37), 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 there corresponds to a query and each column to a key. Before anything else, we see that these are essentially the corresponding attentions as represented by query-key correspondence. Thus, if we make certain elements of this matrix very small (actually, set them to large negative numbers, since we are applying softmax subsequently) we will make sure 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 (keys) to $-\inf$ for the features where the mask value is 1.

2) Positional encoding: We mentioned positional encodings a couple of times before, so here is a short description of how they work. If we look at the attention mechanism, we can see that the procedure is invariant to the permutations of input elements. If we swap two features, all the corresponding rows (or columns) in Q, K, and V will be also swapped and at the end, we will obtain a swapped result. This is not desirable since we want our model to know if some elements are closer in the input sequence or further apart. While this is obvious in the NLP domain, where the sentences "I go to Purdue" and "Purdue go I to" are clearly different, it turns out that order is also very important for image features. After our feature extractor, we obtain a set of features of dimension (C, H_f, W_f) where C is the number of features and (H_f, W_f) is the size of each feature. Every 2-D feature map along the first dimension corresponds to different aspects of the input image, which might be related in different ways between each other. Thus we want to have a sense of the ordering, absolute and relative.

Positional encodings P are usually of the same size as the feature set X they encode. Let us assume that our input X is a matrix of shape $(d_x \times d_{model})$. We need to encode the position along the first dimension. There are two common approaches that we looked into: Use manually constructed sine and cosine waves or just make these encodings learnable. The second choice is relatively straightforward amounting to just introducing a new learnable parameter that is added to the input. The first one, used in the original implementation of transformers in [12] works as follows.

The idea is to get a set of unique sine and cosine sequences for each position in X. If t is the position along the first dimension, let us define

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

where T is a parameter sometimes called temperature which is set to 10000 in our implementation. Now, at position t, we can define the following d_{model}-dimensional vector:

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

These 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 give us the positional encoding matrix:

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

Finally, this matrix is added to the original input: $X \to X + P$. Note that in our case the input is not of size $(d_x \times d_{model})$ but generally of dimension $(d_{features} \times W \times H)$. In this case, we will similarly apply the positional encodings with positional index t being along the first (features dimension). The only difference is that the sines and cosines will be calculated independently along both W and H dimensions which will lead to unique sequences.

E. Bipartite Matching Loss

To quantify the networks's object detection accuracy after each iteration, the DETR architecture employs bipartite matching loss [14].

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 [14], 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 39), 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. 30.

$$\hat{\sigma} = \underset{\sigma \in \sigma_N}{\arg\min} \sum_{i}^{N} \mathcal{L}_{Hungarian}(y_i, \hat{y}_{\sigma(i)}) \tag{39}$$

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, $\mathcal{L}_{Hungarian}$ is the matching cost betweena 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.

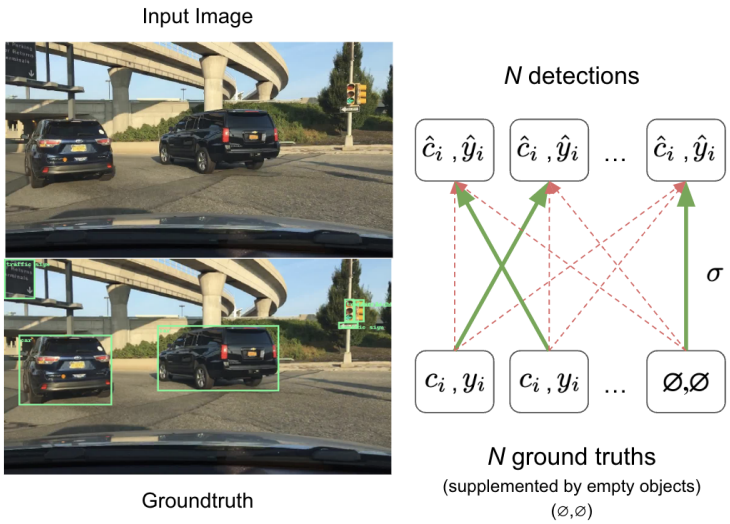

Fig. 30: Bipartite Matching Loss. [14]

The Hungarian Loss function that the Bipartite matching minimizes is defined in the DETR paper [14] 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 40, 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.

$$\mathcal{L}_{Hungarian}(y_i, \hat{y}_i) = \sum_{i=1}^{N} \left(-log(\hat{p}_{\hat{\sigma}(i)}(c_i)) + \mathbb{1}_{\{c_i \neq \varnothing\}} C_{IOU}\right) \tag{40}$$

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 6.3.

F. Data Augmentation

This semester was the first year when we successfully implemented a custom extensive data augmentation scheme. The augmentations can be logically separated into 2 types. "Simple,"

involving operations only on a single image, and "complex," where multiple images are combined together to get a mixed input. The simple augmentations involved:

- Random perspective transofrmation. This augmentation, as can be inferred from its name applies a random projective transformation which can be described with the following matrix expression:

$$\begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{41}$$

  where the coordinates (x_1, x_2, x_3) correspond to image coordinates (x, y) as:

$$x = \frac{x_1}{x_3}, y = \frac{x_2}{x_3} \tag{42}$$

  and the values h_{ij} are general projective coefficients. This transformation can be intuitively thought of in the following way. Given the original image, move the four corners arbitrarily, and try to "fill" the resulting quadrilateral with the "content" of the original image.
- Random crop. Since the previous transformation might lead to the image being partially out of original bounds or might create some "empty space" in the image, we will crop it so the image is fully present inside the window.
- Random horizontal flip. This transformation flips the image symmetrically with respect to the vertical axis. If the original image is described by a matrix $X_{i,j}$ of size (W, H), then the resulting image $Y_{i,j} = X_{W-i,H-j}$
- Photometric distortion. This stage takes care of colors so that our network can perform well with different image tones. It randomly adjusts such parameters as brightness, hue, contrast, and saturation. The description of each of those values is quite involving when dealing with simple RGB color maps and the logic behind their usage is often associated with the humans' perception of color. Thus the detailed description is outside of the scope of this report.

Not mentioned in the list above, but all the geometric transformations require extreme care as it is very easy to "lose" a bounding box or to run into numerical inconsistencies.

After applying the transformations above with certain probability, our pipeline then either outputs these images into the trainer or goes one step further and applies one of the "complex" augmentations described below.

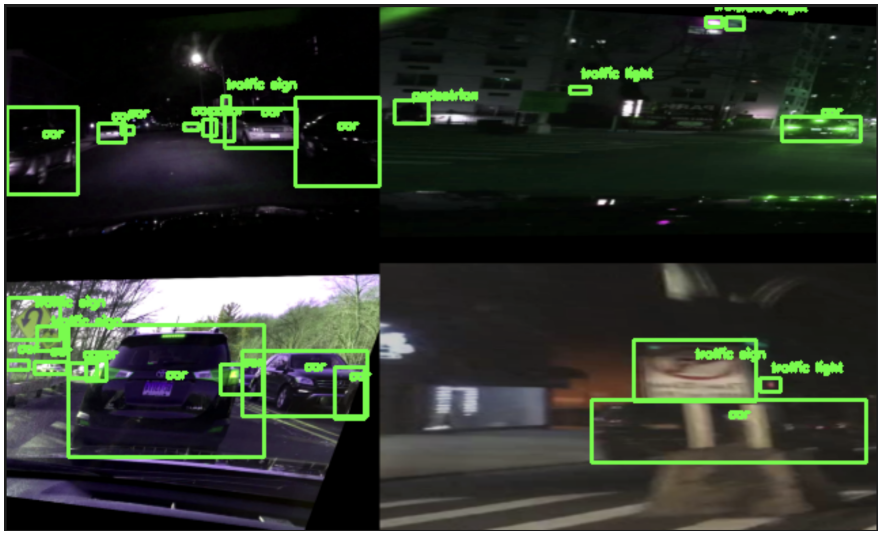

Fig. 31: Mosaic augmentation result

1) Mosaic Augmentation: Mosaic Augmentation was originally proposed in the implementation of YOLOv5, which rivaled YOLOv4's CutMix. We found mosaic augmentation to be more logical than CutMix and thus decided to implement it. The algorithm is very simple. Suppose the image has its bottom left corner at point $(0, 0)$ and the top right corner at point (x, y). We uniformly choose a center point (x_c, y_c) where $x_c \sim Unif(ax/2, (1-a)x/2)$ and $y_c \sim Unif(ay/2, (1-a)y/2)$ where $0 < a < 1$ is a parameter usually set above 0.8. After the center is chosen, we will randomly select four images from the dataset and resize them to sizes

$$(x_c, y_c), (x_c, y - y_c), (x - x_c, y - y_c), (x - x_c, y_c)$$

. We perform resizing using a bilinear interpolation. After this, the resized images are put in the four quadrants. The example of mosaic augmentation applied to an image is shown in Fig. 31.

2) MixUp augmentation: This augmentation was also introduced in the YOLOv3-v4 series. The idea is to merge two images by their pixel values directly. The example is shown in the Fig. 32. Given two images X_1 and X_2 can corresponding one-hot encoded labels and boxes l_1 and l_2,

we generate a new sample by choosing a random parameter λ and generating a new datapoint:

$$\tilde{X} = \lambda X_1 + (1 - \lambda) X_2$$

$$\tilde{l} = \lambda l_1 + (1 - \lambda) l_2$$

What it does is essentially connect points in the sampling distribution and create new samples somewhere on the line. We noticed that this augmentation did not work very well, so we decided to assign a relatively small probability to it.

Fig. 32: Mixup augmentation result

G. Evaluation Metrics

For evaluating the accuracy of our model's detections, we used the same metrics as the Spring 2023 implementation, as described in Section 5.5. The Mean Average Precision (mAP) metric was used as a baseline to allow us to compare our network's performance to the results achieved by Facebook's implementation, described in the DETR paper [14].

H. Results

The DETR network we developed this semester was trained for 100 epochs with evaluation metric results shown in Table IV. As described, the metric we used was Mean Average Precision in order 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, we found that large objects had a mAP of 40.2, medium objects had 21.0, and small objects had 4.0. Although our accuracies are lower than Facebook's implementation across the board, our results are 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. 33 - 38. Note that Fig. 33 - 36 are images from the BDD100k dataset, and Fig. 37 - 38 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 IV: Metric Comparison of DETR Implementation.

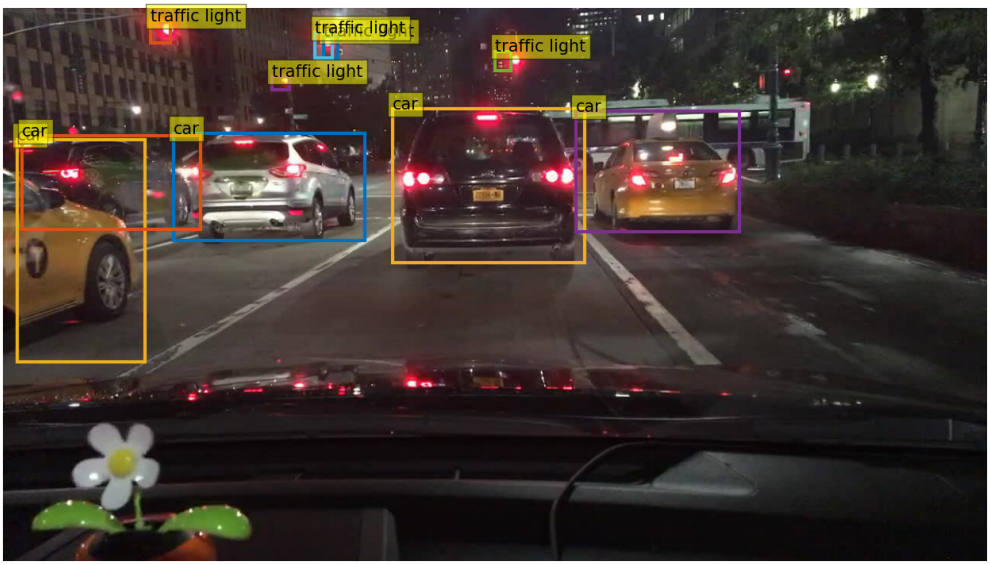

Fig. 33: Results of DETR model (BDD100k)

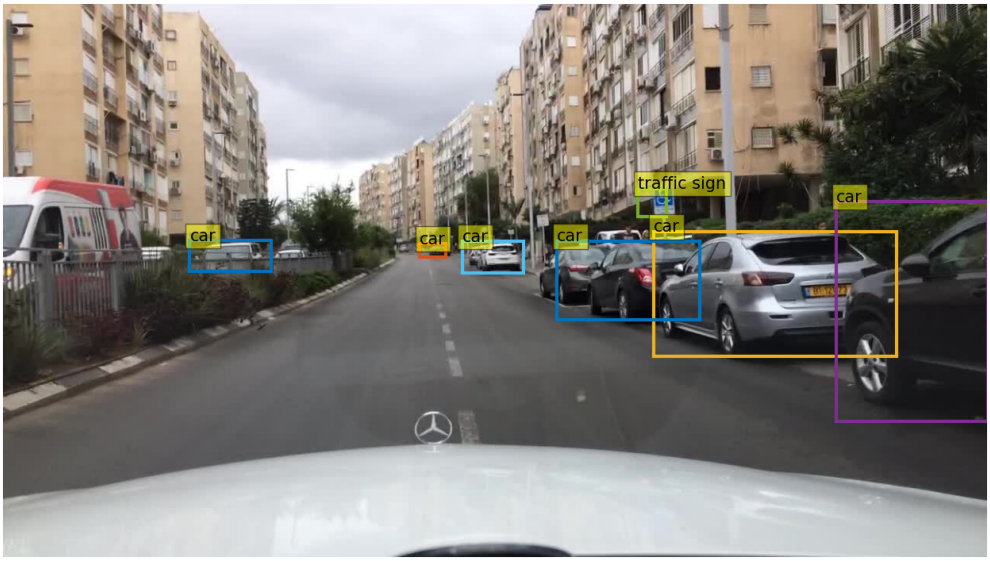

Fig. 34: Results of DETR model (BDD100k)

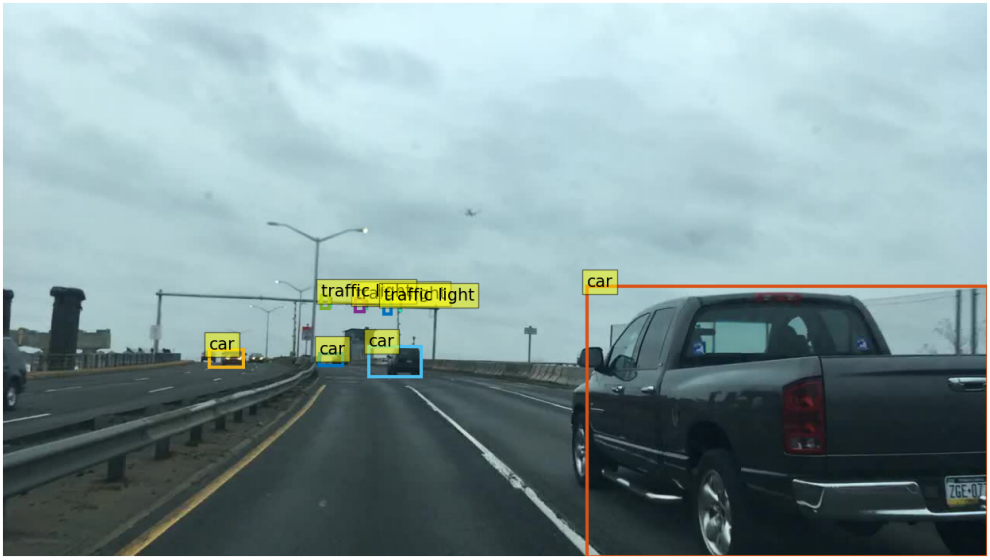

Fig. 35: Results of DETR model (BDD100k)

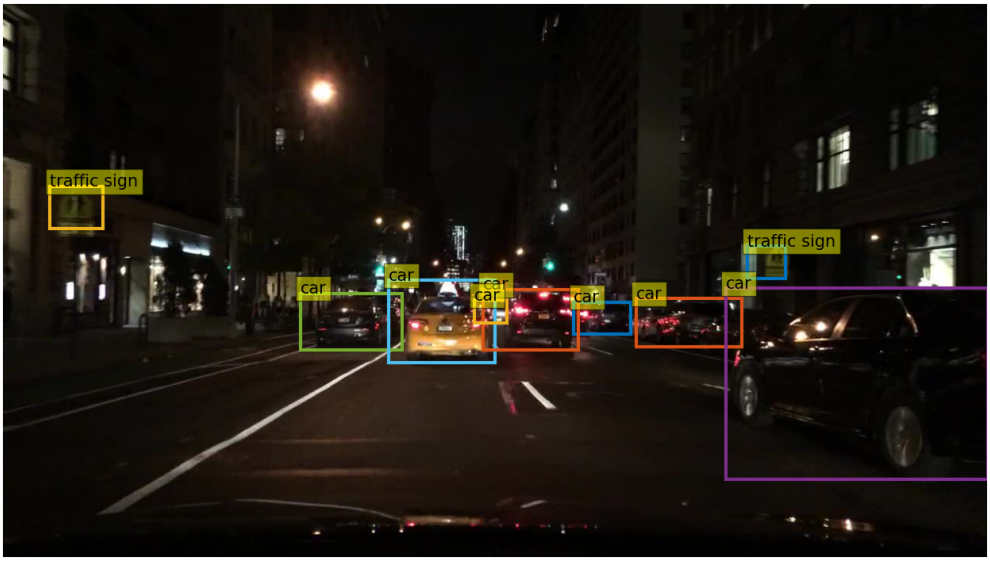

Fig. 36: Results of DETR model (BDD100k)

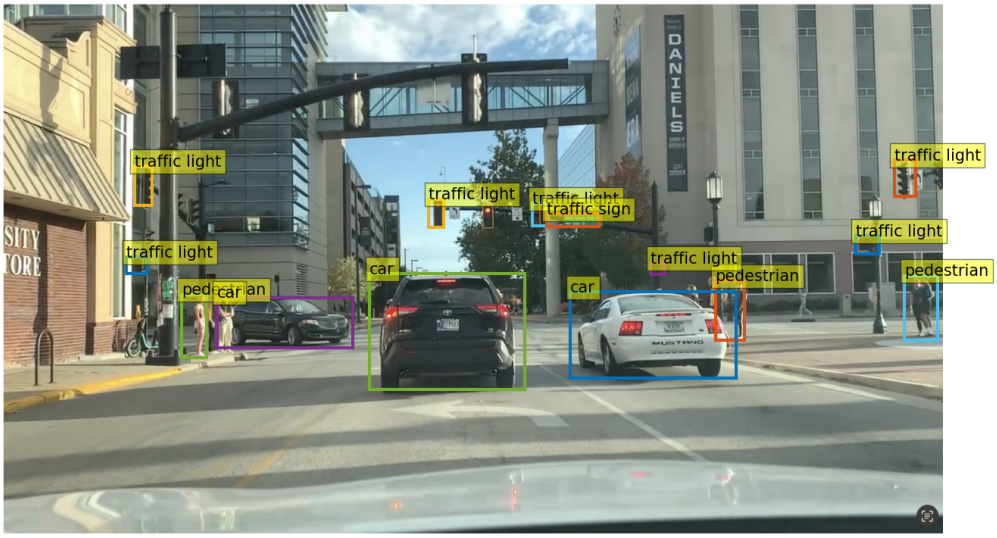

Fig. 37: Results of DETR model (Purdue)

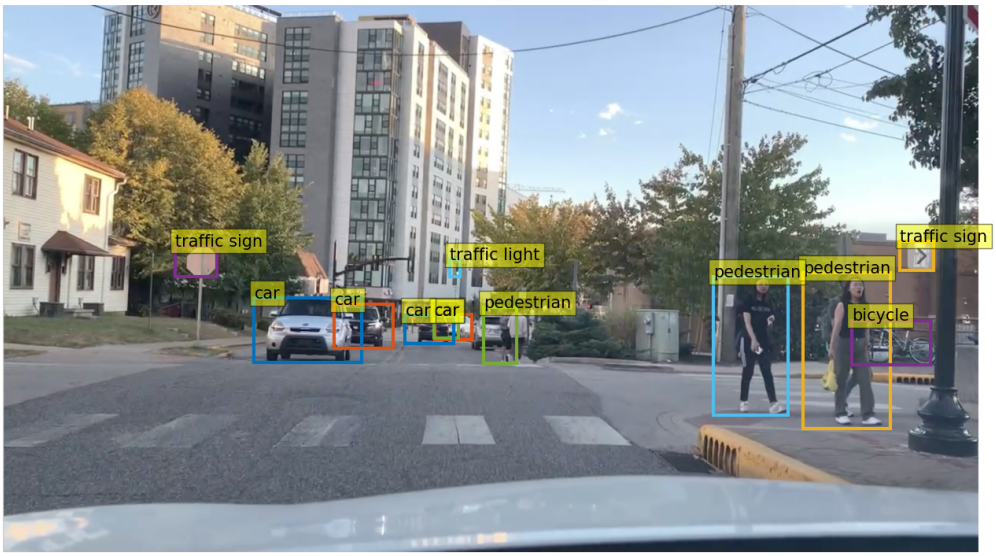

Fig. 38: Results of DETR model (Purdue)

8. Future Work

A. DETR improvements

Although we have managed to do a lot of work on the detection transformer, there are certainly some ways to improve the system. First, the biggest problem of DETR is the number of parameters and the amount of resources necessary to train the network. In our implementation, the model had $94M$ parameters, which require at least 4 GPUs to train efficiently. This constraint limited our training time since the machine available to us had only 4 GPUs that had to be shared with other teams. Also, one of the GPU slots had connectivity issues that further limited our training time. With that in mind, we were only able to train the network a few times with minimum hyperparameter tuning. In the future, we plan to expand the types and sizes of our models to determine the sweet spot for our application which should balance speed and accuracy.

B. SWIN transformers

As the next step, we also plan to look at another transformer application called SWIN transformer proposed by Liu et al. in 2021 [15]. The idea behind the SWIN transformer is very similar to ViT. Recall that in ViT, the original image of size (H, W) is cut into patches of a certain size (H_p, W_p) totaling $\frac{H}{H_p} * \frac{W}{W_p}$ patches. Each of these patches is flattened and considered as a part of the input sequence. Now, suppose we want to run a detection task where very small objects are present. That means our patches should remain small, while the resolution of the input images is steadily growing with new advances in camera technologies. The result is that the number of patches grows as a square of the linear size of the image, and the number of operations required to compute a self-attention grows as a fourth power of linear size which is unsustainable in the long run.

Instead, the authors of the SWIN transformer proposed to calculate attention only within a window. At each stage, the attention between parts is calculated only within a small window, thus allowing total quadratic complexity in image linear size. At each stage, the windows are shifted allowing us to calculate the attention between the patches which are in different windows at the previous stage. The network showed promising results on COCO and we think is a good candidate for next semester's investigation.

9. Conclusion

The results obtained this semester are promising, as our implementation of the DETR network is able to perform effective object detection and classification at scale, achieving slightly less accuracy than the Facebook implementation due to training constraints. The visual results are equally promising, as nearly all objects are detected, correctly classified, and bounded tightly by the detection boxes. However, there are some cases, such as in Fig. 34, where two cars that overlap are being detected as a singular car. Additionally, some bounding boxes struggle with cars that are not entirely visible and cut off by something. For example, Fig. 33 displays a taxi being detected as a skinny car due to its second half being cut off by the frame, rather than the detection extending outside the image like the truck in Fig. 35. So, with these discrepancies and our average precision of 33.4, the model still has room for improvement and further optimization. Some of the future work mentioned in Section 8 could be performed to combat these shortcomings. In summary, our final DETR model was able to successfully run the object detection task we set out to achieve, and achieved promising results for the team to continue to improve in the future.

REFERENCES

[1] 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.

[2] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," May 2016. [Online]. Available: https://arxiv.org/abs/1506.02640v5

[3] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," Apr 2018. [Online]. Available: https://arxiv.org/abs/1804.02767

[4] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," May 2015. [Online]. Available: https://arxiv.org/abs/1505.04597

[5] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," Apr 2020. [Online]. Available: https://arxiv.org/abs/2004.10934

[6] D. Misra, "Mish: A self regularized non-monotonic activation function," Aug 2020. [Online]. Available: https://arxiv.org/abs/1908.08681

[7] C.-Y. Wang, H.-Y. M. Liao, I.-H. Yeh, Y.-H. Wu, P.-Y. Chen, and J.-W. Hsieh, "Cspnet: A new backbone that can enhance learning capability of cnn," Nov 2019. [Online]. Available: https://arxiv.org/abs/1911.11929

[8] S. Liu, L. Qi, H. Qin, J. Shi, and J. Jia, "Path aggregation network for instance segmentation," Sep 2018. [Online]. Available: https://arxiv.org/abs/1803.01534

[9] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," Apr 2017. [Online]. Available: https://arxiv.org/abs/1612.03144

[10] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," Feb 2018. [Online]. Available: https://arxiv.org/abs/1708.02002

[11] Z. Zheng, P. Wang, W. Liu, J. Li, R. Ye, and D. Ren, "Distance-iou loss: Faster and better learning for bounding box regression," Nov 2019. [Online]. Available: https://arxiv.org/abs/1911.08287

[12] 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

[13] 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

[14] 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

[15] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," Mar 2021. [Online]. Available: https://arxiv.org/abs/2103.14030

10. Appendix

A. *William Stevens*

This was my fourth semester in VIP-IPA, and my third semester on the Lane Detection team. Misha and I were the only team members who returned from last semester, so it fell upon us to direct the team and get the new team members up to speed in order for us continue the prorgress from last semester. For my own work, I spent most of my time researching the concept of transformers and how they could be applied to our project. Transformers were suggested by us to Professor Delp and some graduate students, as they have been a novel concept in machine learning that was recently applied to computer vision. Some initial readings directed me first towards the orignal 2017 paper on transformers "Attention is All You Need," [12] which then led me towards papers that took the concept and applied it to computer vision. I researched, and later helped to implement, the concept of the Vision Transformer (ViT) introduced in "An Image is Worth 16x16 Words" [13], as well as the Detection Transformer (DETR) proposed in "End-to-End Object Detection with Transformers (DETR)" [14]. I also learned about many related concepts, such as the unique loss function used in the DETR implementation, bipartite matching loss. After these concepts had been sufficiently researched and presented to our mentors, I worked on implementing and testing them in Python. Roughlythe last month of the semester was spent working on the training script for the DETR model and various postprocessing and evaluation algorithms in order to produce final results, both visual and numerical.

On a more weekly basis, I directed the work of the team alongside Misha during our virtual meetings outside of normal class time. I made plans for and contributed slides to the weekly presentations. In addition, I helped design and create our research talk for participation and presentation in the Undergraduate Research Conference. Within this report, I was responsible for writing the abstract, introduction, Detection Transformer introduction/concept/matching loss/results, and the conclusion. I also adapted the dataset and the YOLOv4-Multi sections from last semester's report to fit into the context of this semester.

B. Mykhailo (Misha) Tsysin

This semester was my second and final semester in the VIP team and in the Lane Detection team. Over the two semesters, my goal was to research different ways of detecting objects on roads using various tools. After successfully implementing YOLO-based models in the past semester, I started this semester by polishing the old implementation of YOLO. I managed to fix most of the bugs and achieve better overall results. Next, my focus switched to a new promising set of networks based on transformers. I started the research by reading the foundational papers from the NLP domain and then slowly transitioning the knowledge to vision applications. Having realized the potential flaws of the Vision Transformer model, we switched towards the detection transformer, which proved to be a good choice. After learning all the necessary aspects of the network architecture, I implemented the code and trained the model on the BDD100k dataset. Since the model requires a lot of computational resources, we were not able to reproduce the original author's results. Still, the results we got were very good, with large object detection mAP being the best. Having joined the team with very limited knowledge of image processing, after successfully implementing the two networks, I feel like I learned a lot. For this report, I worked on the functional description of our final model: detection transformer. I was mainly responsible for the transformer and DETR architecture, augmentations, and evaluation metrics.

C. Vishal Urs

This was my first semester in the VIP Image Processing Analysis (IPA) team. My knowledge in the field of machine learning and image processing was rudimentary when I joined the team. During my time here, I have been exposed to several fascinating ideas and machine learning techniques. I have been able to make practical connections between concepts from my academic classes and the work I have done in this team. I was able to understand the role of convolution in image processing and the role of linear algebra in representing images as matrices of values. Through the development of grayscaling, edge detection, and blurring algorithms, I have been able to build a concrete foundation of machine learning concepts and image processing techniques. Alongside this, I worked with my teammates to construct our own implementation of a convolutional neural network (CNN) trained on the MNIST dataset. I have been exposed to multiple object detection architectures through the different research papers we explored in the quest for an optimal object detection technique. YOLOv1 is one such architecture that was a breakthrough in object detection and computer vision. It was faster and more efficient than its contemporaries. Karthik, Gabriel, and I worked together to develop our implementation of the YOLOv1 object detection architecture as a step toward building detection models. We were not able to test it on the BDD100k dataset, but we were able to train it on VOC2007. Our results on the VOC2007 dataset were up to our expectations, which indicates the proper functionality of our model. However, our purpose is to develop detection models for application in autonomous driving; the BDD100k dataset suits this purpose. Our primary focus going forward will be to train and test our architecture on BDD100k. Once we attain positive results, I hope to work on more optimized methods of object detection with the more experienced members of the team. Through this experience, I have made steady progress and have been able to learn a lot. With time, we discover new models and techniques that are better suited for lane detection. There is always something to improve on. There is always something to learn.

D. Karthik Selvaraj

This semester was the first semester I was part of the VIP Image Processing and Analysis (IPA) team. I came into this semester with some knowledge of machine learning and signal processing through courses such as ECE 47300 and ECE 30100. However, I learned much more about the concepts covered in these courses and how to apply them to computer vision tasks. In general, I have gained a lot of knowledge on image processing and machine learning techniques. Previously, I had learned about the convolution operation and convolutional neural networks (CNN) but never understood that CNNs do not actually use convolution. I now know that CNNs use the cross-correlation operation. I also employed various introductory image processing techniques such as grayscaling, image blurring, and histogram manipulation. Additionally, I further strengthened my understanding of the gradient descent algorithm and its application of the chain rule while also learning more about how the gradient matters during the training process. Over the course of this semester, Gabriel, Vishal, and I worked with the Brain Tumor Detection Team to train a CNN in PyTorch on the MNIST handwritten digit dataset. Gabriel, Vishal, and I worked on implementing the gradient descent algorithm for the CNN. We ran into trouble with writing the gradient descent algorithm from scratch, so we decided to implement the CNN in PyTorch after coding the other portions of the CNN from scratch. We were able to achieve a very accurate network. After this, Gabriel, Vishal, and I worked on implementing YOLOv1 in PyTorch. We trained our model on the VOC2007 dataset instead of the BDD100k dataset and achieved the results that we expected. Unfortunately, we did not have time to train our model on the BDD100k dataset, but this will be our first goal for next semester. In the future, we hope to also work on the other architectures the Lane Detection team has developed and further our understanding of object detection and image segmentation. Over the course of the semester, I have gained new insight into image processing, computer vision, and machine learning in general. I have utilized my mathematics degree time and time again to gain a deeper understanding of these concepts. I look forward to continuing to learn and grow in this team.

E. Gabriel Torres

This semester was my first semester doing VIP. Before joining the VIP VIPER lab, I had some knowledge about image processing. However, my knowledge was severely limited because I used third-party libraries like OpenCV to do edge detection and image blurring. So I never understood the math behind manipulating images. Since I was new to image processing, I implemented algorithms to grayscale, Sobel edge detect, and blur images. Then with the fundamentals of image processing under my belt, I helped the Lane Detection team re-implement YOLOv1. I had to learn the basics of machine learning, like what a loss function is or the point of an activation function. Then I was able to build the architecture for YOLOv1, while Karthick and Vishal did the training and loss function respectively. I was able to train the model on the VOC 2007 dataset, to see if we had constructed the model correctly. It turns out that the model is able to find the correct location of the objects but has trouble classifying and identifying the true size of the objects. Next semester, we will try to figure out the model's inability to correctly size and classify objects. Then hopefully we can get some results training on BDD100k.