Contents

CHAPTER 1

Introduction

When comparing the human brain to machines, humans are better at interpreting data at a higher level and in a more abstract format, whereas machines excel in processing raw numerical data due to their high computational power. Additionally, by Moore's law, the number of transistors in a dense integrated circuit doubles every two years, leading to machines becoming more powerful over the year. Hence, if machines can interpret and understand the representation of objects beyond raw numerical data, they can process and react much faster than humans. Machine vision proves advantageous to humans in situations that require quick reactions, such as in everyday traffic. If executed correctly, machines equipped with vision can predict and prevent traffic accidents faster than humans, thus facilitating safer traffic flow.

Over the last decade, we have seen multiple raised startups, as well as big corporations, pour millions of dollars into research to try to obtain a piece of the autonomous vehicle market which is valued at billions of dollars [**autonomous_vehicle_market**]. One of the leading companies in the field of autonomous vehicles is Tesla. In October 2015, Tesla released the Tesla Version 7.0 software enabling the Autopilot feature for the Model S and promised that the car would be fully autonomous in 2017 [**tesla_software_v7**, **tesla_2017_promise**]. However, in July 2022, 2 different Tesla cars crashed while on autopilot, and each crash caused the death of a motorcyclist [**tesla_crash2**]. Additionally, in 2019, a Tesla autopilot crashed into a Honda Civic and killed two people [**tesla_crash1**]. Not to mention, there are 273 Tesla crashes involving the autopilot system reported just in 2019 [**autonomous_vehicle_market**]. As we can see, as of 2022, we have yet to be able to develop a fully autonomous system.

To be able to achieve a fully autonomous vehicle, it is crucial to identify which module or modules in the autonomous driving modular pipeline are at fault. The autonomous driving modular pipeline

consists of four components. These components are Perception and Localization, High-Level Path Planning, Behavior Arbitration, and Motion Controllers [Fig. **??**].

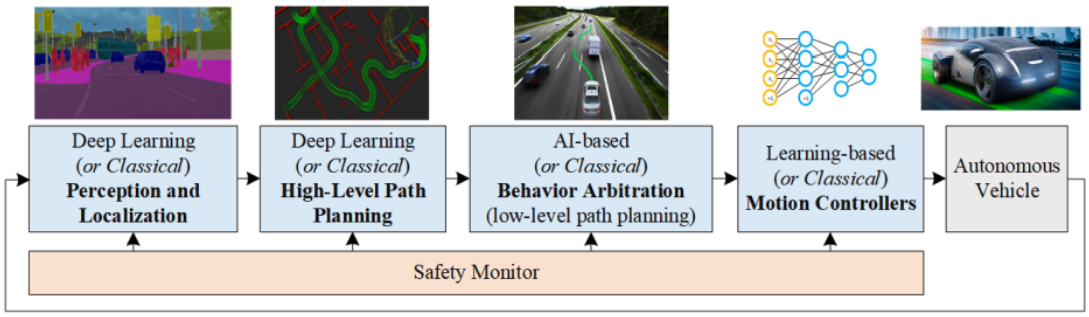


Figure 1.1: Deep Learning Based Autonomous Driving Modular Pipeline **[grigorescu_trasnea_cocias_macesanu_2020]**

To identify which module or modules are at fault, we need to understand and analyze each module of the modular pipeline. In this paper, we will only look at Perception and Localization, the first module in the autonomous vehicle pipeline. The Perception and Localization module is responsible for answering two questions: "Where is the car?" and "What is around the car?" **[liu_2020]**. To perform this function, the module utilizes the sensing hardware and the software to understand the scene. Such hardware includes LiDAR, cameras, and other sensors. Nonetheless, they are all used for getting the information surrounding the vehicle and feeding those data to the software functionality of the module.

In this study, we assume that the sensor (hardware) is reliable and able to pass the surrounding data to the software part of the module. With the recorded data, the software module must be able to understand the data scene, which is a very challenging task. Unlike human beings, machine sees and processes objects differently from our eyes and brain. While we see the object as a whole, the machine sees it as a grid of values that do not have any connection with one another. For this reason, the Computer Vision field tackles this problem and tries to create algorithms that are able to make sense of the object representation beyond the raw numerical data. The most important groups of algorithms in the Computer Vision field for understanding traffic scenes are video segmentation algorithms.

There are two types of video segmentation algorithms. The two are video semantic segmentation and video instance segmentation. In the video semantic segmentation algorithm, for each scene, objects in the scene are grouped and classified based on categories **[overview_cv_task]**. On the other hand, video instance segmentation detects each instance of each category for each scene

[**overview_cv_task**]. Comparing semantic segmentation and instance segmentation, we can see that instance segmentation proposes a higher accuracy and detail as it is able to see each instance of a class. That is, the instance segmentation algorithm will be able to distinguish between a car and another car behind it, while semantic segmentation considers these two cars as one. Further discussion of semantic segmentation and instance segmentation is provided in Section **??**.

Since video instance segmentation algorithms give higher detection accuracy, in this paper, we assume an algorithm from this group is used for the Perception and Localization module. Knowing a video is a sequence of images, many video instance segmentation algorithms were proposed based on an algorithm in image instance segmentation. One example of such an algorithm is MaskTrack R-CNN which includes a new tracking branch to the well-known image instance segmentation Mask R-CNN. For that reason, to understand video instance segmentation, we first need to understand and analyze image instance segmentation which is adapted from an object detection model. More specifically, this paper discusses the building blocks of the object detection model and its application in image segmentation tasks. We analyze and compare the performance of two well-known algorithms, the Mask R-CNN and You Only Look Once version 5 (YOLOv5), in the object detection task. Furthermore, we want to identify if failure to detect objects by these image object detection algorithms causes the autonomous driving pipeline to fail and result in accidents.

CHAPTER 2

Neural Network

The group of Instance segmentation algorithms is a subgroup of object detection algorithms. All algorithms in the object detection group are required to do two tasks [**overview_cv_task**].

1. Generates a bounding box surrounding each object in the image.
2. Classify the object in the bounding box.

We first discuss the algorithm used for the classification task. Since in each bounding box, there is exactly one object to classify, an algorithm from the image classification task, a subset of the computer vision task, is applied [**overview_cv_task**].

In the early day, as the first step toward artificial intelligence, a machine learning approach was proposed for the image classification task, but most were still designed manually by humans [**traditional_machine_learning**]. Conventional machine learning uses feature extraction functions to map raw data to feature vectors. The feature vector is the only suitable data format that allows the learning subsystem of machine learning to detect patterns and classify the input. For that reason, the accuracy and effectiveness of machine learning methods are heavily dependent on the feature extraction function. However, the feature extraction function's responsibility is to extract features unique to the object that the machine tries to detect. Thus, the function requires an extremely detailed design and immense domain expertise to extract the feature of one object. Another disadvantage is that each object requires a different feature extraction function as they have unique features [**traditional_machine_learning**]. The variety in features between objects causes the engineer to redesign the entire machine-learning architecture for each object which is a difficult task and inefficient.

As more studies go into the field, we start to move away from manually designed machine-learning methods to a more data-driven model. This data-driven algorithm group is now known as the artificial neural network.

Neural networks, also known as artificial neural networks (ANNs), are inspired by the human brain. Similar to the way the human brain processes and makes decisions, ANN is the core process of machine learning that gives the machine the ability to interpret the representation of raw binary data and move it toward artificial intelligence. An ANN algorithm is driven by data, the more data is supplied to the algorithm, the more accurate its interpretation ability becomes [**ai_data_driven**].

At the highest level of abstraction of ANN, there are two kinds of neural network architectures. The first and simplest one is feedforward architecture which allows its signal to travel from input to output [**lecun2015deep**]. The feedforward network is widely utilized in grid patterns processing tasks, like image and video frames. The second architecture is the recurrent neural network (RNN). RNN expands the feedforward network's functionality by adding feedback connections that allow the network to feed its output back to the network [**lecun2015deep**]. Feedback connections enable RNNs to be sufficient for processing sequences of data tasks.

As this paper's interest lies in detecting objects of a scene, which require classifying individual objects in a grid of pixel values, we will only discuss the detail of a fully connected feedforward neural network.

2.1 Feedforward Neural Network Architecture

Feedforward neural networks are the most basic type of deep learning model. Feedforward networks are designed to approximate a function that best maps the network's input to its output [**lecun2015deep**]. On a high level, a feedforward neural network algorithm has four phases. The first phase is forward propagation, where the data flows through the network and initializes every node's values. The second phase is error evaluation, which determines how well the network performs with its current nodes' value. The third phase is gradient descent, where the algorithm determines which part of the network causes it to perform poorly. The last phase is backpropagation, where the algorithm updates the node's value in the network to make it more accurate and better fit the input data set. These four phases are applied and repeated on an extensive data set to become more accurate over time. We discuss each phase of the algorithm in this section, but we first need to understand the basic structure of a feedforward neural network.

2.1.1 The Network Structure

To understand the feedforward network structure, we first define the idea of layers in the context of ANN. A **layer** in a neural network is a collection of neuron nodes at a specific network depth. A layer is represented by a column of nodes in Figure **??**. A feedforward network can be thought of as a stack of multiple layers. Each layer in the stack is responsible for transforming the layer's input to help make sense part of the representation for later layers. At a high level, the network consists of the **input layer**, single or multiple **hidden layers**, and the **output layer**. The number of hidden and output layers is the **depth** of the network. As an example, Figure **??** is a fully connected feedforward neural network with a depth of three.

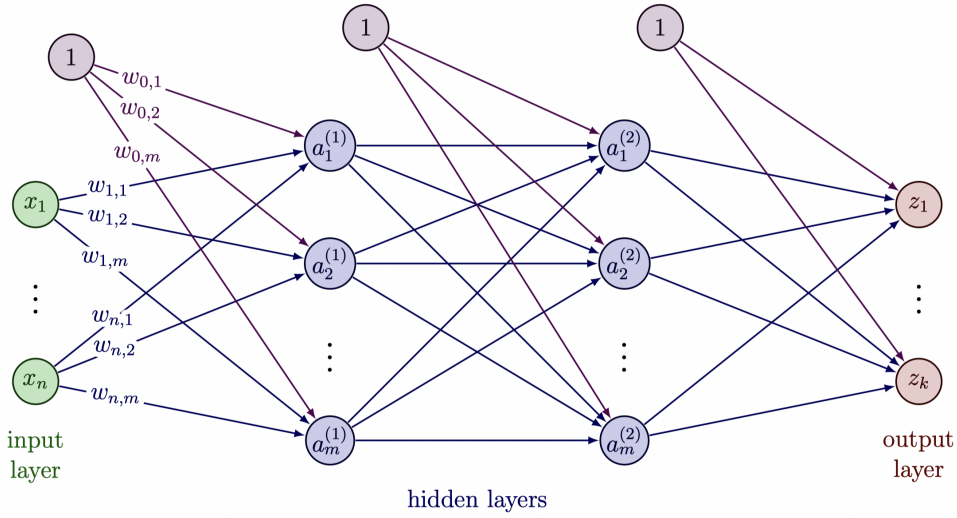


Figure 2.1: A Fully Connected Feedforward Neural Network

Each layer consists of one or more nodes. Each node in a layer represents an **artificial neuron**. The number of neurons in each layer can be different from one another. Each neuron holds a numerical value called the **activation signal**. In theory, the activation signal takes on any value in the real numbers set $\mathbb{R}$. However, in practice, studies have shown multiple advantages when normalizing the activation signal of every neuron **[lecun2012efficient]**. As an example, we consider a network that receives a 23×23 RGB image and then produces a dog or cat label for the object in the image. In the input layer, the number of neurons is the total pixel in the image in each color channel, that is $23 \times 23 \times 3$ neurons, and each neuron (denoted x_i) holds that pixel value normalized. On the other hand, the output layer only has two neurons (denoted z_i); one corresponds with the dog and the other with the cat. Unlike the input and output layers, where the number of neurons needed is straightforward, the number of neurons in each hidden layer and the number of hidden layers are

complicated to determine and remain outside the scope of this paper. However, studies have shown that networks with the same number of neurons in hidden layers perform better [**taylor2017neural**], and deeper networks result in more advanced learning with some issues like gradient vanishing [**lecun2015deep**].

Each neuron in a layer affects a neuron from the next layer through a **connection**. The connection is represented by a line between two neurons in Figure **??**. In fully connected networks, each neuron in a layer is affected by all the neurons in the previous layer. In other words, each neuron has a connection with every neuron in the previous and the next layer of the network. Associated with each connection is a numerical value representing the **weight**. The weight of a neuron affects the influence the neuron has on the next layer of the neural network. A small weight value means the activation signal of this neuron has a low effect on neurons in the next layer. On the other hand, a high weight value results in a more significant effect proposed by this neuron to the next layer and the network's output. When a neural network algorithm initializes, its weights are randomly assigned using a Gaussian or uniform distribution [**lecun2015deep**]. These weights are adjusted as the neural network algorithm processes to best approximate its mapping function. The weight from neuron a_i to a neuron a_j in the next layer is denoted as $w_{i,j}$ with the exception of bias's weight.

In addition to the weight, neurons in each layer other than the input layer are also influenced by a **bias node**. A bias node is represented by a node that always holds the value 1. The bias node has connections to every neuron in the layer it affects. Each bias's connection also has a weight associated with it. The connection between bias and neuron a_j in the affected layer has its weight denoted as $w_{0,j}$. The role of the bias node is like a threshold to the neuron. In other words, the bias determines how large the activation signal must be before that neuron gets propagated to the next layer of the network [**taylor2017neural**]. Similar to weight, when the network initializes, the bias's weight is randomly assigned a value, and this value is optimized as the algorithm process. In the forward process of the network, the algorithm will compute a net input that requires a multiplication between a neuron activation signal and its weight. Since the bias node always has an activation signal of 1, thus the bias weight is the variable that directly affects nodes in the next layer. Therefore, the bias's weight is often referred to as bias.

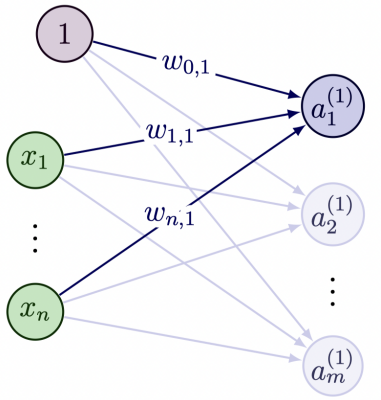
These basic building blocks create the internal structure of a feedforward neural network. Some internal parts are also known as **internal variables or network parameters**. The model parameters refer to variables that are learned during the gradient descent process, and they are not set manually by the developers. Weights and bias weights are examples of the network's parameters. In contrast with parameters, hyperparameter refers to variables manually set by the developers and sometimes

can be optimized through training [**lecun2015deep**]. Examples of hyperparameters are the learning rate and batch size, which we will touch on more in the gradient descent section.

With the basic structure of the fully connected feedforward neural network in mind, we will discuss the first phase of an ANN algorithm. That phase is forward propagation.

2.1.2 Forward Propagation

Forward propagation refers to the process by which the data move from the input layer to the output layer of the network. Regarding the image classification problem, the forward propagation process moves a list of raw pixel values through the network and gives out a number for each class label. These numbers are the **network's raw ouput**, and each number represents how likely the image is a member of this class. The forward propagation process uses two functions to evaluate the activation signal of each neuron in hidden and output layers. These two functions are the **summation function** and the **activation function** [**taylor2017neural**].



In a fully connected network, the summation function combines all neurons and their weight from the previous layer to create the net input for the current neuron. In general, the summation function is expressed as

$$net\ input\ of\ a_j = \sum_{i=1}^{n}(x_i w_{i,j}) + 1 \cdot w_{0,j}$$

where a_j represents the neuron that has the summation function compute its net input [**taylor2017neural**]. The neuron's net input is then transformed by an activation function.

The activation function is responsible for transforming the net input into an activation signal, indicating if this neuron will affect later network layers. Additionally, linear functions are closed under addition; that is, adding or subtracting multiple linear functions from or to another function results in a linear function. This fact implies that the feedforward network must have non-linearity terms if it approximates a non-linear function. The use of the activation function enables the network to introduce non-linearity terms to the algorithm. Furthermore, studies have shown that the output of a network - a network that only uses a linear activation function or does not use an activation function - will be a linear combination of its input, which means hidden layers have no effect [**He_2015_ICCV**].

There are numerous activation functions proposed, each with different strengths and weaknesses. However, in practice, there are four functions and their variants that are widely used by the ANN algorithm. These four activation functions are linear, sigmoid, hyperbolic tangent (tanh), and rectified linear unit (ReLU). The formula and graphical representation of each function are shown in Table **??**.

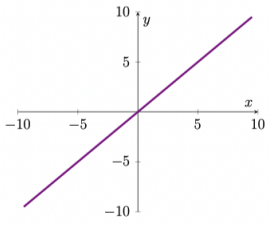
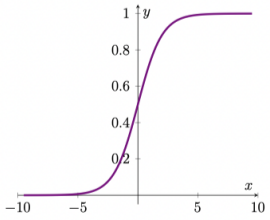
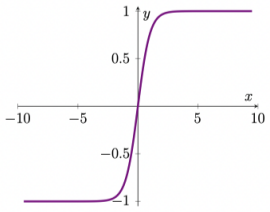
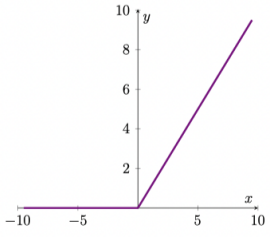
Activation Function	Graph	Formula
Linear		$\phi(x) = x$
Sigmoid		$\phi(x) = \frac{1}{1+e^{-x}}$
Tanh		$\phi(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
ReLU		$\phi(x) = max(0, x)$

Figure 2.2: Most Common Activation Function

To understand which activation function to choose for a network, we need to know some important strengths and weaknesses of each function. First of all, the linear function is simple; thus, it is forgiving and undemanding about the resources required for training compared to other activation functions. However, the linear function is close under addition; thus, the network will only be able to approximate a linear mapping function. In contrast, sigmoid and tanh are non-linear

functions, thus allowing the network to approximate a non-linear mapping function. Additionally, the sigmoid and tanh map the real number set to (0, 1) and (-1, 1), respectively. The mapping of sigmoid and tanh allow the activation signal to be normalized, thus improving training time and avoiding exploding gradients [**lecun2015deep**]. However, the mapping also causes sigmoid and tanh to suffer from vanishing gradients problems. Different from sigmoid and tanh, ReLU maps positive signals to itself, thus able to avoid the vanishing gradients problems. The first disadvantage of ReLU is it does not have any upper bound, which requires the network to have some additional normalization layer like batch normalization (BN), weight normalization (WN), and layer normalization (LN) to optimize the training process [**relu_optimization_2020**]. The second disadvantage of ReLU is it suffers from the dying ReLU problem, which is caused by the fact that all negative signals are mapped to 0 [**dying_relu**]. Despite the dying ReLU problem, ReLU proves to be very efficient and accurate in practice; thus, it is the most used activation function currently [**li2021survey**]. Furthermore, some variance of ReLU has been proposed to avoid the dying ReLU problem like Leaky ReLU and ELU. The problems possessed by these activation function is crucial when it comes to choosing an activation function and can be read more at LeCun et al., [**lecun2015deep**].

2.1.3 Error Evaluation

Once the forward propagation is completed, the algorithm needs to determine the correctness of the network with the current weight and bias. In order to estimate the network's correctness, the algorithm computes the difference between the network's output and the expected output. This process is known as the **error evaluation phase**, and it uses a **loss function** to quantify the difference. The three most used loss functions are **Mean Absolute Error** (MAE), **Mean Square Error** (MSE), and **Cross-Entropy**. MAE and MSE are primarily used in regression problems, while Cross-Entropy is mainly used in classification problems [**li2021survey**]. As the focus of this paper is on the image classification task, we will only focus on Cross-Entropy.

Cross-Entropy is a loss function that is always used in conjunction with a softmax layer [**taylor2017neural**]. A **softmax layer** is a layer that has the same number of nodes as the network's output layer. The goal of a softmax layer is to transform the raw output into probabilities for classes in the network's output, denoted $\hat{y}$. In other words, each neuron node activation signal is a class's probability after the softmax layer and is bounded by 0 and 1; and the probabilities of all classes must add to 1. The softmax layer uses a softmax function to map a raw output value to the range

$0-1$. The **softmax function** is defined as follow:

$$\hat{y}_i = \sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \qquad \text{with } i = 1, 2, 3, ..., k$$

where $z_1, z_2, z_3, ..., z_k$ are the value of network's raw output. We noted that the softmax function has non-negative and normalization properties. The function is non-negative because the e^z term is always positive, which results in both the numerator and denominator of the function being positive. Additionally, since numerator e^{z_i} denotes the value of a neuron in the output layer (corresponded with class) and the denominator is the sum in value of all neurons in the output layer; thus this fraction normalizes the value of neuron z_i with respect to all output neurons. Furthermore, the normalization property with respect to all output neurons allows the $\hat{y}$ value of all output neurons to be summed to 1. The use of a softmax layer is crucial for output interpretation. Since a raw output does not have a lower or an upper bound for its value, thus different images might result in a different value range for the class label. This wide range of value behavior makes the raw output extremely hard to interpret and compare with the output of other images. The softmax layer standardizes the output by mapping the raw output to the range $0-1$ before comparing and calculating the error.

The softmax layer enables us to represent the network's output as a probability distribution for the object's class with a higher value means the object is more likely to be a member of that class. Similarly, we can also represent our desired classification output as a probability distribution with 1 for the object's class and 0 for all other classes. With that in mind, we have the Cross-Entropy function able to quantify the difference between two probability distributions [**taylor2017neural**]; thus, we use it to determine how far the network's output is from the actual desired output. The Cross-Entropy value of a standardized output neuron can be computed as follow:

$$\text{CE of } \hat{y}_i = -\sum_{i=1}^{k} t_i \times \log(\hat{y}_i)$$

where t_i is the expected class$_i$'s value for the object present in our input image and $\hat{y}_i$ is the class$_i$'s probability for the object predicted by the network.

The Cross-Entropy for a neuron above describes the distance between the neuron's current activation signal and its expected value. By computing the Cross-Entropy for every output neurons, the sum of these Cross-Entropy values gives us the total error of the network. That is, the total Cross-Entropy describes how far the network is from its expected output in a single value and can

be formularized as follow:

$$\text{Total Cross-Entropy} = \sum_{i=1}^{k} \text{Cross-Entropy of } \hat{y}_i$$

where k is the number of output neurons i.e. the number of class that we are trying to classify. Since total Cross-Entropy gives us a single value to describe the total error in the network, thus by reducing the total Cross-Entropy value, the network will become more accurate. This concept of reducing the total Cross-Entropy value to make the neural network more accurate is the core idea of gradient descent.

2.1.4 Gradient Descent

Gradient descent is an optimization process in which our feedforward network evaluates how the network's parameters affect total error, thus giving insight into how to reduce the total error [**taylor2017neural**]. An ANN can have two or more parameters depending on the model; however, there are two that exist in any ANN model: weight and bias's weight. For simplicity, let us assume our network only has two learnable parameters – weight and bias. If we were to plot the total Cross-Entropy as a function of weight and bias in 3D space, we would result in a 3D surface that describes the total Cross-Entropy values at different combinations of weight and bias. The idea of gradient descent is to have the network's total Cross-Entropy value moving toward the global minimum, which will use a specific combination of weight and bias. There are three types of gradient descent methods; to understand those, we first define the idea of an epoch, batch, and iteration.

- An **epoch** refers to when the network sees the entire training data set exactly one time [**taylor2017neural**].
- A **batch** is the number of examples that pass through the network exactly once before updating the network's parameters [**taylor2017neural**].
- An **iteration** refers to the number of times a batch of data need to pass through a network to complete an epoch. This also states the number update for an epoch [**taylor2017neural**].

As an example, if our image classification training data set has 1000 images with a batch size of 250, then we need 4 iterations to pass the entire training set through the network and complete one epoch.

The three types of gradient descent are **Full-Batch Gradient Descent** (BGD), **Stoc-hastic Gradient Descent** (SGD), and **Mini-Batch Gradient Descent** (MGD). Each method impacts when the weight and bias will be updated during training, thus resulting in different pros and cons.

The first gradient descent method is BGD which is a one iteration method. Since BGD only updates weight and bias after an epoch, it is undoubtedly the slowest of the three in terms of training time [**taylor2017neural**]. However, by having the batch size equal to an epoch, BGD guaranteed to find the local minimum on a non-convex 3D surface, and the global minimum on a convex 3D surface [**taylor2017neural**]. Thus BGD enables the network to adjust the weight and bias to move toward the optimal solution over each epoch.

The second gradient descent method is SGD which is an n iteration method where n is the number of training examples in one epoch. Since SGD updates the weight and bias after each training example passes through the network; thus it is much faster in updating weight and bias than BGD [**taylor2017neural**]. Despite having a faster update, SGD suffers from a high variance of training examples since improving the error for one example does not equate to improving the error for other examples in the training set. Hence, SGD is faster for large training sets but might never reach the global minimum of the loss function [**taylor2017neural**]. As an additional note, using SGD requires the training set to be shuffled before input to the network, as the order of samples can introduce unknown bias to the model.

The third and most used method nowadays is MGD. MGD has the advantage of both BGD and SGD as it allows the developer to choose the trade-off between training time and accuracy through batch size. **Batch size** is one of the network's hyperparameters which is bounded by 1 and n, where n is the number of training examples in one epoch. A larger batch size moves the network behavior toward BGD behavior, while a smaller batch size will cause the network behavior to resemble SGD. Studies have shown the value of batch size should be a relatively small power of 2 bounded by 1 and a few hundred with a reasonable default value of 32 [**bengio2012practical**, **masters2018revisiting**]. Some studies also propose the use of an adaptive batch size where the network starts with a small batch size, then increases after each update [**lecun2012efficient**]. However, the rate of increase of batch size in these proposals is still challenging to determine and generalize; thus, most successful networks still only use a fixed batch size.

By choosing one of the three methods of gradient descent, we now know when the network's weights and bias get updated. To know how much the weights and biases need to be updated to bring the model closer to the global minimum of the loss function, we need to know how much a change in weight or bias affects a change in the loss function. For this reason, the algorithm uses partial derivate to quantify the rate of change of the total Cross-Entropy with respect to a change in a specific weight or bias. This process is also known as computing the gradient for each weight and

bias in the network. The gradient for weight j can be computed using the following formula:

$$\frac{\Delta \text{Tot. CE}}{\Delta w_j} = \sum_{i=1}^{k} \frac{\Delta \text{CE of } \hat{y}_i}{\Delta w_j}$$

where $\hat{y}_1, \hat{y}_2, ..., \hat{y}_k$ are the standardized output of neurons affected by weight w_j. To understand how to evaluate the gradient, we consider a simple network shown in Figure **??**.

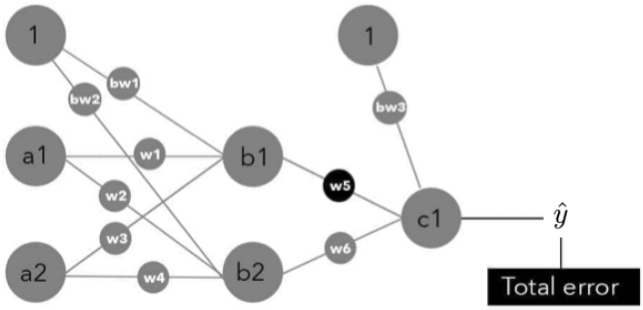


Figure 2.3: Simple Neural Network **[taylor2017neural]**

For this example, we will calculate the gradient of the weight w_5 and weight w_1. To calculate the gradient of weight w_5, we need to calculate the rate of change of the total Cross-Entropy with respect to weight w_5. Since this simple network only has one output neuron, denoted c_1, then the rate of change of total Cross-Entropy is the rate of change of the Cross-Entropy of $\hat{y}$, where $\hat{y}$ is the standardized value of output neuron c_1. Thus we have the gradient of weight w_5 as:

$$\frac{\Delta \text{Tot. CE}}{\Delta w_5} = \frac{\Delta \text{CE of } \hat{y}}{\Delta w_5} \tag{2.1}$$

However, weight w_5 does not affect the Cross-Entropy of $\hat{y}$ directly, but instead affects the net input of output neuron c_1, which intern affect the activation signal of c_1. Then and only then, output neuron c_1 affects the standardized value $\hat{y}$ and its Cross-Entropy. For that reason, to compute the gradient of weight w_5, we need to link weight w_5 to the Cross-Entropy of $\hat{y}$ by performing chain rule operations on the partial derivative. Apply chain rule to Equation **??** for weight w_5's gradient, we have:

$$\frac{\Delta \text{Tot. CE}}{\Delta w_5} = \frac{\Delta \text{CE of } \hat{y}}{\Delta \hat{y}} \times \frac{\Delta \hat{y}}{\Delta c_1} \times \frac{\Delta c_1}{\Delta c_1 \text{ net\_input}} \times \frac{\Delta c_1 \text{ net\_input}}{\Delta w_5} \tag{2.2}$$

Similarly to weight w_5, the gradient of weight w_1 can also be computed using the partial derivative with the chain rule. Weight w_1 affects the net input of neuron b_1, then its activation signal. Then,

neuron b_1 impacts the net input of neuron c_1 and its activation signal. Neuron c_1 then affects $\hat{y}$ and its Cross-Entropy value. Notice that from neuron c_1 onward, the change is the same as part of weight w_5's gradient formula; thus, we can expand and adapt Equation **??** to compute the gradient of weight w_1. The gradient of weight w_1 can be evaluated as follow:

$$\frac{\Delta \text{Tot. CE}}{\Delta w_1} = \frac{\Delta \text{CE of } \hat{y}}{\Delta \hat{y}} \times \frac{\Delta \hat{y}}{\Delta c_1} \times \frac{\Delta c_1}{\Delta c_1 \text{ netin}} \times \frac{\Delta c_1 \text{ netin}}{\Delta b_1} \times \frac{\Delta b_1}{\Delta b_1 \text{ netin}} \times \frac{\Delta b_1 \text{ netin}}{\Delta w_1}$$

where "netin" stand for the net input. These examples showcase the computational process for the gradient of an in-network and an out-network weight, i.e., w_1 and w_5, respectively. The same computation process for the gradient is applied to all network weights and biases, even with a more complex and higher depth network. Once the gradient is calculated for all the weights and biases, the algorithm knows how much a particular weight or bias changes the network's total error and is thus ready to update each weight and bias accordingly.

2.1.5 Backpropagation

Backpropagation refers to a phase in which the algorithm uses gradients of weight or bias to update their value and bring the network's total error to the global minimum [**taylor2017neural**]. Since the gradient gives the slope of the loss function with respect to weight or bias, and backpropagation updates their value to approach a local or global minimum, thus gradient descent along with backpropagation process enables the neural network to have a behavior similar to the idea of learning through the process of optimizing network's parameters. The formula to update the value of weight w_j is:

$$\text{new } w_j = \text{old } w_j - \left(\frac{\Delta \text{Tot. CE}}{\Delta w_j} \times \eta \right) \tag{2.3}$$

where old w_j is the current value of weight w_j, $\frac{\Delta \text{Tot. CE}}{\Delta w_j}$ is the gradient of weight w_j, and η refers to the algorithm's learning rate.

Learning rate, denoted η, is a network's hyperparameter, and it determines how fast the network learns. In the updating weight function, equation **??**, the learning rate directly affects the size of the step when the algorithm moves toward the global minimum for total error. A large learning rate means a bigger step toward minimum, thus resulting in faster learning, while a smaller learning rate value results in slower learning [**taylor2017neural**]. However, a large learning rate can also affect the network's ability to reach the global minimum, as it can overstep and pass the optimal value.

Studies have shown the value of a leaning rate for a multi-layer ANN should be between 10^{-16} and 1 with a reasonable default value of 0.01 [**bengio2012practical**].

As the backpropagation phase is completed, all weights and biases in the network are updated, and all four phases of the algorithm are repeated on the next batch, where the size of each batch depends on the method of gradient descent. These phases are repeated until the algorithm's total error function reaches a global minimum [**taylor2017neural**]. At that moment, a test set is passed through the network, and the network's total error is used to evaluate the performance of the network.

2.2 Convolutional Neural Network

Convolutional Neural Network (CNN) is a type of feedforward neural network designed to process images. A simple structure of a CNN consists of five types of layers. These layers are the input, convolutional, pooling, fully connected, and output layers [**o2015introduction**]. The fully connected layer is responsible for the actual classification process. Both fully connected and output layers behave the same as in a fully connected feedforward network discussed in Section **??**. Figure **??** shows a simple CNN structure for handwritten digit classification.

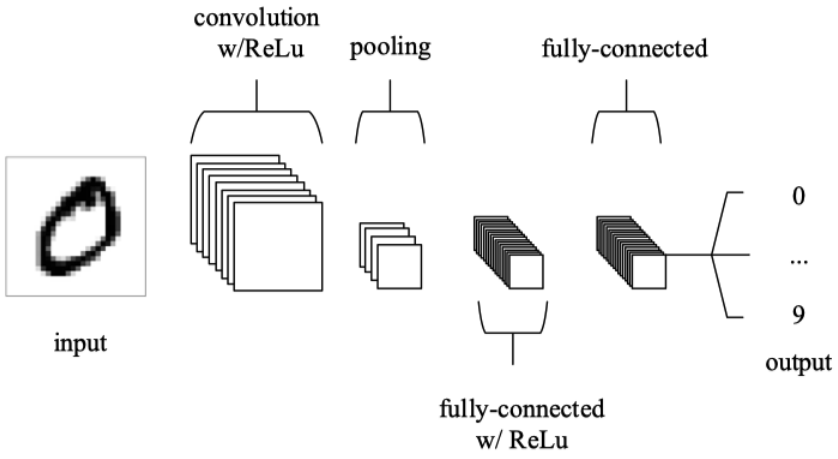


Figure 2.4: Simple CNN structure for handwritten digit classification [**o2015introduction**]

Since an image is a 2D grid pattern data, it is possible to flatten the image and pass it through a fully connected feedforward neural network for classification directly. However, there are multiple benefits when using CNN over a standard feedforward network. The two most important benefits of CNN are spatial interaction capturing and data downsampling.

The first significant benefit is spatial interaction. **Spatial interaction** in an image refers to the connection between two or more pixel values [**goodfellow_book**]. These connections are essential since pixels next to one another tend to describe a feature of an object, while pixels far from each other describe a different feature of the same object or a completely different object; thus, spatial interaction enhances feature extraction. On the other hand, if an image is flattened, pixels that appear close to one another will be very far away in the network, thus losing their meaning.

The second important benefit is data downsampling. **Data downsampling** refers to reducing the number of weights in the network [**goodfellow_book**]. Consider a 64×64 RGB image, in a fully connected feedforward network, each neuron in the network needs to consider values from $64 \times 64 \times 3 = 12,288$ neurons from the previous layer. In other words, each neuron has $12,288$ incoming connections, and the network needs to compute the gradient and do backpropagation for $12,288$ weights per neuron per network's layer. Thus, the computational and memory usage are still expensive despite the image being a low-resolution photo. Therefore, if a network could downsample the data, it would be more efficient, have less training time, and require less computational power.

CNNs are able to capture the spatial interaction and perform data downsampling using the convolutional and pooling layer before passing to a fully connected layer for classification. To further understand CNN architecture, we will discuss convolutional and pooling layer functionality in detail.

2.2.1 Convolutional Layer

The **Convolutional layer** is responsible for making spatial connections and extracting features from the image [**cnn_feature_extraction**]. These tasks can be achieved with the use of learnable kernels. A kernel is a grid of data that has a smaller width and height but has the same depth as the input image. For example, a 64×64 RGB image requires the kernel to have the size of $x \times x \times 3$ where x is in the range [2, 64]. There are various types of kernels, and each type is designed to target a specific task [**o2015introduction**]. These tasks include blurring, sharpening, edge detection, and more. When applying the kernel to the input image, it slides from left to right and top to bottom. As it slides, the kernel's activation signal is computed by performing a scalar product of the kernel with the subregion of the image covered by it, as shown in Figure **??**. The kernel's activation signal represents how likely the feature – the feature that the kernel is trying to extract – is present at the current spatial position of the input image. The resulting grid of the kernel's activation signal is

the feature map. Each kernel has an associate activation map. If multiple kernels are applied to an image, then the output feature map of these kernels will be stacked along the depth dimension.

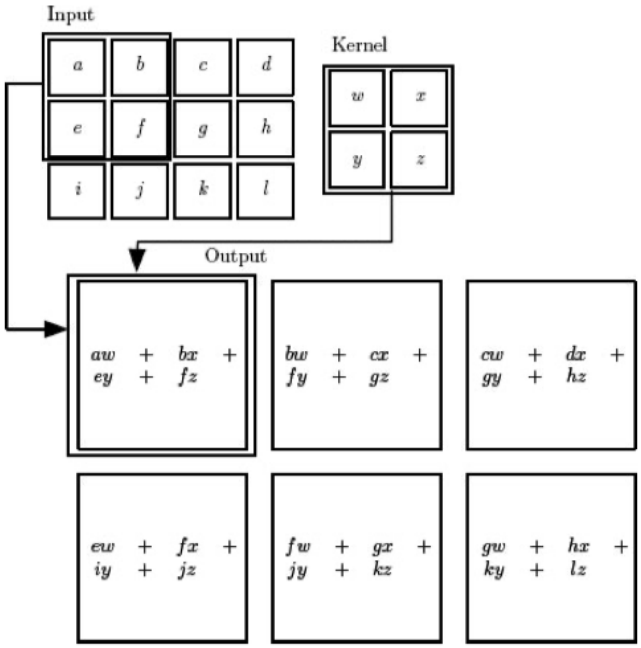


Figure 2.5: Scalar product for a kernel's activation signal **[lecun2015deep]**

Notice that a kernel's width and height must be equal, and the kernel itself must be symmetric **[o2015introduction]**. Studies have shown that using symmetric kernels enables the algorithm to extract the inverse of a feature, thus improving the generalization for feature extraction. Additionally, since the kernel's activation signal is only based on a subregion of the image that the kernel applied to, this kernel neuron will only be connected with the neurons associated with pixels in this subregion in the network, thus reducing the number of weight in the network. The width and height of this subregion are also known as the **receptive field size**. Reconsider our 64×64 RGB image example, if the recaptive field is 4, then each neuron has $4 \times 4 \times 3 = 48$ connections. Thus, the network only needs to compute gradient descent and do backpropagation for 48 weights for this neuron instead of $12,288$ weights.

Other than width and height, we can also change the **stride** and the **zero-padding** to fine-tune the kernel behavior. The stride enables the kernel to slide through the image with a more significant step **[o2015introduction]**. That is, if the stride is 1, then the kernel will move to the left and down one pixel at a time and calculate the kernel's activation signal. Besides the stride, zero-padding also changes the convolutional behavior. The zero-padding value is the number of zero rows and

columns surrounding the border of the input image. The zero-padding allows the algorithm to emphasize the border of the input image. Along with stride and zero-padding, we denote a kernel as

$$\text{kernel width} \times \text{kernel height, zero-pading size, /stride value}$$

And the feature map size can be computed as follow:

$$\text{feature size} = \left(\frac{(\text{image size} - \text{kernel size}) + 2 \times \text{padding size}}{\text{stride}}\right) + 1 \tag{2.4}$$

2.2.2 Pooling Layer

Unlike the convolutional layer, the pooling layer only has one purpose: to reduce the number of neurons in the network [**o2015introduction**]. The use of the pooling layer helps result in fewer parameters for the network and thus requires less computational power. Similar to the kernel, the pooling layer also slides through a grid of values. However, instead of sliding through the input image, the pooling layer slides through the feature map or the convoluted image to reduce the size of the feature map. There are two types of pooling layers, and they are **max-pool** and **average-pool**. As the name suggested, a max-pool layer will extract the largest activation signal in a subregion of the feature map while removing all other activation signals in the same subregion. An example of a max-pool function is shown in Figure **??**. On the other hand, an average-pool layer takes the average value of all the activation signals in the subregion. The choice of which pooling layer to use depends on whether we care about every feature in the image equally, then the average-pool is used, or if we only care about the more prominent feature, then max-pool is used. Pooling also uses the stride to change how big of a step the pooling layer slides through the feature map, denoted $/x$ where x is the stride value. Since the pooling layer also slides through a grid of values and outputs exactly one value for the subregion, thus a similar function to Equation **??** is used to calculate the output size for the pooling layer. The output size can be calculated as follow:

$$\text{pooling's output size} = \left(\frac{(\text{feature size} - \text{pooling size})}{\text{stride}}\right) + 1$$

As an example, reconsider our 64×64 RGB image example, by applying a $4 \times 4, 0,\ /1$ kernel and a $2 \times 2,\ /2$ max-pool layer, the convoluted image size before passing to fully connected layer for classification is:

$$\text{output size} = \left[\left(\frac{(64-4) + 2 \times 0}{1} + 1\right) - 2\right] \times \frac{1}{2} + 1 = 30$$

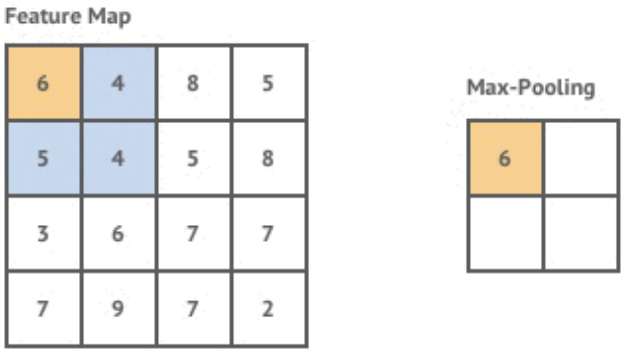


Figure 2.6: 2×2, /2 max-pool on **[zeiler2014visualizing]**

Thus, the algorithm is able to reduce from $12,288$ weights to $30 \times 30 \times 3 = 2700$ weights. In practice, it is common to have more than one kernel and one output apply to an input image.

CHAPTER 3

Computer Vision Tasks And Evaluation Metrics

3.1 Object Detection, Instance Segmentation, and Other Tasks

The field of computer vision aims to enable machines the ability to comprehend and derive meaning from visual scenes. This field comprises several tasks for processing images and videos, including but not limited to image classification, object detection, semantic segmentation, and instance segmentation [**overview_cv_task**]. However, for the purpose of our study, we will focus specifically on object detection and instance segmentation. In this section, we will define these two tasks, then compare them with related tasks such as image classification and semantic segmentation. Followed by a discussion of the metrics used to evaluate object detection and instance segmentation models.

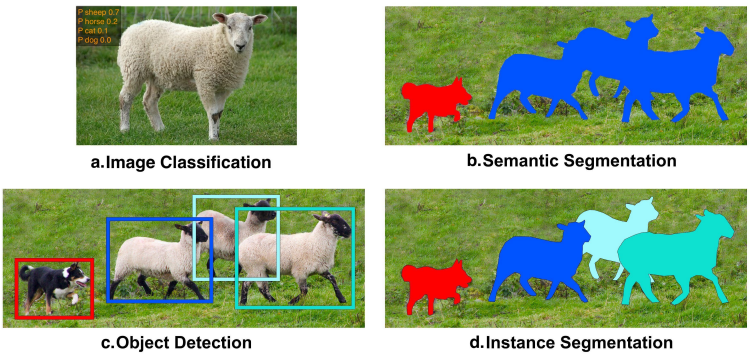


Figure 3.1: Different Computer Vision Tasks [**diff_detection_segmentation_task_fig**]

We begin with image classification, the most fundamental task in the computer vision field. The task involves assigning an object category label to an input image, assuming the input image contains exactly one object [**overview_cv_task**]. This implies that the task allows the object classification

label to be given to the entire image without specifying where the object is. As shown in Figure **??**a, the image classification model predicts the probability that the image shows a sheep, a horse, a cat, or a dog. Although image classification is comparatively simpler than other computer vision tasks, it presents several significant challenges due to the variability of image appearance caused by changes in scale, orientation, lighting, occlusions, and other factors.

The second task is semantic segmentation. In contrast to image classification, semantic segmentation is unconcerned with the presence or absence of objects in the input image. The objective of the task is to classify each pixel of the input image into one of several predefined classes or categories [**overview_cv_task**]. By classifying each pixel, the task generates a detailed mapping between image pixels and classes that can be used to locate and outline objects of different classes within the image. However, when classifying each pixel without consideration of object location, the task removes the depth dimension of the image. That is, if two objects of the same class are positioned behind one another, then the semantic segmentation model will consider them as one object of this class, thus losing dimension information. Considering Figure **??**b, we noted the three sheep are at different locations in the depth dimension, but the semantic segmentation mask visualizes these sheep as one object in 2-dimensional space.

The third task is object detection, an improvement over image classification. The goal of object detection models is to locate and categorize each object within a given image [**overview_cv_task**]. The process of object detection involves two main subtasks: object localization and object classification. Object localization determines the location and size of each object in an image by predicting a bounding box around the object. Once the object is localized, it can be classified using an image classification model. Compared to semantic segmentation, object detection retains all spatial information of the object in the image but loses the pixel accuracy mask. This is illustrated in Figures **??**c and **??**d, where object detection is able to detect all three sheep, but it is unclear whether a pixel within the bounding box refers to the sheep or the grass behind it.

The last task we want to discuss is instance segmentation. For every object in a given image or video frame, the instance segmentation model must identify all pixels belonging to the object and assign a category label to the object [**overview_cv_task**]. Identifying all pixels that belong to the object creates the object's mask, which may then be used to locate pixel-precise locations and the object's outline. While both object detection and instance segmentation involves locating objects, the former offers information about the location and scale of the detected objects, whereas the latter goes further by identifying all pixels associated with the object. On the other hand, the main distinction between instance and semantic segmentation lies in the level of detail they provide. While semantic

segmentation divides an image into classes such as "vehicle" or "animal", instance segmentation provides more details by distinguishing between each individual object within those classes. It accomplishes this by assigning each object a unique label for identification. This means that instance segmentation can determine not only the class of each object in an image but also where it is located and how many there are - something semantic segmentation cannot achieve on its own. Considering Figure **??**, we observed that instance segmentation creates a pixel-accurate outline of all four animals, in contrast to the bounding box generated by object detection. In addition, the instance segmentation model's pixel-by-pixel mask must differentiate between the three sheep in contrast to semantic segmentation. In other words, the instance segmentation model combines object detection and semantic segmentation strengths while eliminating their weaknesses.

R-CNN and YOLO are two popular families of models used for object detection and instance segmentation. In Chapters **??** and **??**, we will delve into the variations of the R-CNN and YOLO models, respectively. However, before we proceed, it is essential to understand the metrics used for assessing the performance of an object detection or instance segmentation model. In the next subsection, we will discuss the metrics utilized for evaluating these models.

3.2 Evaluation Metrics

As previously mentioned, an object detection model is responsible for localizing and classifying objects in an image by predicting a bounding box for each object and assigning a class label. Therefore, to evaluate the performance of an object detection model, the accuracy of the predicted bounding box and the given class label must be evaluated. Similarly, while evaluating an instance segmentation model, it is critical to analyze the quality of the pixel-wise mask in addition to the bounding box and category label. As a result, in this section, we will go over the metrics used to assess the accuracy of the bounding box, category label, and pixel-wise mask.

3.2.1 Intersection over Union (IoU) Metric

We start by discussing the evaluation metric for bounding box accuracy in the object detection task. In this task, the model predicts a bounding box for each object in the image to describe the object's location and size. The bounding boxes are commonly represented as a tuple $(x_{min}, y_{min}, x_{max}, y_{max})$ indicating the coordinates of the top-left and bottom-right corners of the box. To evaluate the accuracy of the predicted bounding box, we compare its coordinates with the coordinate of a ground-truth

bounding box. Each ground-truth bounding box is a manually annotated rectangle that precisely encloses an object in the image or video frame. With the ground-truth boxes, **Intersection over Union (IoU)**, also known as the Jaccard index or Jaccard similarity coefficient, the metric for measuring the degree of overlap between the predicted bounding box and the ground-truth bounding box for a given object [**generalized_iou**]. In other words, given two rectangles represent the predicted bounding box and the ground-truth box, then the IoU can be expressed as:

$$\text{IoU} = \frac{Area(BB_p \cap BB_t)}{Area(BB_p \cup BB_t)} \tag{3.1}$$

where BB_p is the predicted bounding box, and BB_t is the ground-truth bounding box. The IoU metric is defined in the range $[0, 1]$, where a value closer to 0 indicates no overlap between bounding boxes, and 1 indicates a perfect overlap. Since $Area_{rectangle} = height \times width$, by comparing the area of two rectangles, the IoU metric incorporates the size and aspect ratio of the two boxes in comparison. As a result, the IoU metric is invariant to the size and aspect ratio. Due to its simplicity and invariance properties, IoU is widely used to measure the accuracy of predicted bounding boxes in various computer vision tasks [**generalized_iou**]. Despite its success as a metric, IoU has a major weakness that prevents it from being used as a loss function. As shown in Equation **??**, we noted that as long as the overlapping area is 0, i.e., $Area(BB_p \cap BB_t) = 0$, then IoU will be 0. In other words, when the predicted and ground-truth bounding boxes are not overlapping, the resulting IoU score does not indicate whether they are near or far from each other. The Generalized Intersection over Union (GIoU) was proposed to resolve this issue, as described in [**generalized_iou**].

3.2.2 IoU Threshold and Confidence Threshold

Next, we can assess the model's accuracy in classifying the object within each bounding box. Each bounding box requires the model to predict a categorical label and a confidence score. The label denotes the object's class within the bounding box, and the score reflects the model's confidence in identifying the object as belonging to this class. To assess the model's accuracy, we compare the predicted bounding box label with the ground-truth bounding box label. Before comparing the predicted box and ground-truth box labels, it is crucial to require that the degree of overlap between them meets a certain threshold t_{IoU}. If IoU $\geq t_{IoU}$, the detection is considered as a positive sample, whereas if IoU $< t_{IoU}$, the detection is considered as a negative sample. For each positive sample, we check if the model's confidence is higher than a predetermined threshold, $t_{confidence}$. If the confidence score meets the threshold, we compare the predicted label with the ground-truth label. Conversely,

the comparison will result in a false if a bounding box is classified as a negative sample or has a confidence score lower than $t_{confidence}$. Therefore, to correctly classify a predicted bounding box, the model must satisfy three conditions: (1) the IoU score $\geq t_{IoU}$, (2) the confidence score $\geq t_{confidence}$, and (3) the predicted label matches the ground-truth label. If any of these conditions are not met, the model has incorrectly classified the predicted box.

The first condition (IoU score $\geq t_{IoU}$) is required because, assuming the second and third conditions are met, if the predicted bounding box and the ground-truth bounding box have little or no overlap, then the prediction should be considered false, and the model's accuracy should be reduced. The value of t_{IoU} depends heavily on the task the model tries to accomplish. For tasks that require absolute precision, such as human surgery assistance, t_{IoU} should be set high. However, a much lower threshold is more forgiving for tasks like checking the presence of a glass of water on the table. For evaluating the overall performance of the model, a t_{IoU} value of 0.5 can be used, as demonstrated in the PASCAL VOC benchmark [**pascal_voc_2015**]. Alternatively, the model can be evaluated at each $t_{IoU} \in \{0.50, 0.55, ..., 0.95\}$, as shown in the COCO benchmark [**coco_2014**]. On the other hand, the second condition (confidence score $\geq t_{confidence}$) is necessary because the score directly affects the model's behavior in predicting accurately versus finding all occurrences. This behavior is denoted as the tradeoff between precision and recall, which we will discuss later in this section.

As an example, consider an image with 3 cars and 1 human. After processing the image with our object detection model, 2 cars and 1 human are identified with confidence scores of 0.76, 0.72, and 0.58, and IoU scores of 0.89, 0.32, and 0.52, respectively. The results are shown in the following table:

ground-truth labels	car	car	car	human
predicted labels	car	car	none	human
predicted confidence scores	0.76	0.72	0	0.58
predicted IoU scores	0.89	0.32	0	0.52

Table 3.1: Example's representation: Input image contains 3 cars and 1 human. The model predicts 2 cars and 1 human with respective confidence scores of 0.76, 0.72, and 0.58, and IoU scores of 0.89, 0.32, and 0.52

Assuming the confidence threshold $t_{confidence} = 0.6$, any predicted bounding box with a confidence score lower than 0.2 will be removed. Additionally, assuming the IoU threshold $t_{IoU} = 0.5$, then the truth bounding box with an IoU score lower than 0.5 will be removed [**metrics_survey_2020**]. In this example, one of the predicted car is 68% offset from its truth location, hence the corresponding

truth car label being removed. Thus the predicted labels and ground-truth labels are mapped one to one as follows:

ground-truth labels	car	none	car	human
predicted labels	car	car	none	none

Please note that changes in IoU and confidence thresholds can impact the correspondence between predicted and truth labels. For instance, if the IoU threshold (t_{IoU}) is adjusted to 0.7, the model will only recognize the first car as a ground-truth object, while the second car and human object (sample) will not be recognized. Similarly, if the confidence threshold ($t_{confidence}$) is set to 0.8, the model will not be able to detect any object, resulting in all predicted labels being classified as None. Therefore, it can be said that the value of t_{IoU} determines whether a truth label is considered in the comparison, while $t_{confidence}$ determines whether a predicted label is considered in the comparison [**metrics_survey_2020**].

3.2.3 Confusion Matrix

The mapping between the predicted and ground-truth labels offers insight into the model's ability to detect and classify samples at a particular confidence and IoU threshold. We can quantify this insight using a confusion matrix, a 2 x 2 matrix that assesses the object detection model's performance in distinguishing between two categories [**confusion_matrix_2017**]. Since the confusion matrix assumes that there are only two categories, when multiple categories are in the mapping, each category will have its unique confusion matrix. In other words, for each category, the confusion matrix measures the model's ability to differentiate between samples belonging to this category and those that do not. The confusion matrix comprises four different combinations of predicted and ground-truth labels, as shown:

		Predicted	
		Negative	Positive
Ground-Truth	Negative	True Negative (TN)	False Positive (FP)
	Positive	False Negative (FN)	True Positive (TP)

The **true negative (TN)** represents the count of times the model correctly recognizes a sample not belonging to a specific class. On the other hand, the **false negative (FN)** represents the number of ground-truth samples where the model fails to detect an object of this class at that position. The **false**

positive (FP) is the count of samples that the model identifies as objects of this class at that position where there is no object or the IoU score is low. Lastly, the **true positive (TP)** denotes the number of ground-truth samples accurately recognized by the model. Consider Example **??**. Since we have two classes, we will have a confusion matrix for each class. Take the car class, for example, then:

- True negative: The number of times the model *correctly* classifies *a none-car object as not a car*.
- False negative: The number of times the model *incorrectly* classifies *a car object as not a car*.
- False positive: The number of times the model *incorrectly* classifies *a none-car object as a car* or classifies *a car object as a car but at a wrong location*.
- True positive: The number of times the model *correctly* classifies *a car object as a car*.

By utilizing these definitions, we can create the confusion matrix for the car class and, similarly, for the human class, with thresholds $t_{IoU} = 0.5$ and $t_{confidence} = 0.6$ as follow:

Confusion Matrix for Car Class

		Predicted	
		Negative	Positive
Ground Truth	Negative	1	1
	Positive	1	1

Confusion Matrix for Human Class

		Predicted	
		Negative	Positive
Ground Truth	Negative	3	0
	Positive	1	0

where in the car confusion matrix, FP= 1 because the ground-truth label is "none" while the predicted label is "car", FN= 1 because the ground-truth label is "car" while the predicted label is "none", and TN= 1 because both ground-truth and predicted label ("human" and "none", respectively) are not "car".

3.2.4 Accuracy Metric

With the confusion matrix, we can assess the overall object detection model's performance at a particular threshold by computing the accuracy, precision, and recall metrics. First, the **accuracy (ACC) metric** describes how the model performs in detecting and classifying samples of all known classes. It is determined by dividing the number of correct detections by the total number of predictions made [**szeliski_cv_book**], as follows:

$$ACC = \frac{TN + TP}{TN + FN + TP + FP} \tag{3.2}$$

However, the accuracy (ACC) metric is only meaningful when all classes are equally important and the input data has an approximately equal number of samples belonging to each class. The ACC metric can be misleading when some specific classes dominate the input data. For example, suppose a model processes an image of 495 cars and 5 humans. If the model classifies all 500 detections as cars, then the ACC be $\frac{0+495}{0+0+495+5} = 0.99$. This result indicates that the model is 99% correct in detecting the presence of cars and humans, even though it was unable to detect any humans.

3.2.5 Precision Metric

An alternative metric that can be utilized is precision. **Precision metric** assesses the dependability of the model in identifying a detection as positive for a specific class [**metrics_survey_2020**]. Unlike ACC, precision assesses the model's reliability in classifying a particular class rather than across all classes. Thus, the precision performance of the model is different for each class at a given threshold. Precision is defined in the range of $[0, 1]$ as:

$$Precision = \frac{TP}{TP + FP} \tag{3.3}$$

A precision score of 0 signifies that all samples identified by the model as belonging to this particular class are incorrect, while a score of 1 indicates that all predictions for this class are accurate. However, precision does not reflect instances where the model fails to identify some samples of this class. Take the car class in Table **??** as an example. Let's assume the model correctly classifies the second car as a car at a higher IoU value of 0.8, then the mapping between the ground-truth and predicted labels is:

ground-truth labels	car	car	car	human
predicted labels	car	car	none	none

Therefore, the precision score for this class is $\frac{2}{2+0} = 1$, indicating that when the model predicts a sample as a car, the ground-truth label for that sample is indeed a car 100% of the time. However, it is worth noting that the precision score does not factor in the car that the model missed in our example.

3.2.6 Recall Metric

As previously stated, precision only measures the correctness of detections but not the model's ability to detect all samples of a particular class. **Recall metric** are often utilized in conjunction with precision to overcome this limitation. The recall is mathematically defined in the range of $[0, 1]$ as:

$$Recall = \frac{TP}{TP + FN} \tag{3.4}$$

A value of 0 denotes that the model has missed identifying all ground-truth samples belonging to this class, while a value of 1 indicates that the model has detected all ground-truth samples of this class [**metrics_survey_2020**]. However, similar to other metrics, recall is not flawless, as it does not account for instances where the model classifies all negative samples as belonging to this class. For instance, consider the car class in Table **??**. If our model predicts all ground-truth samples as belonging to the car class, the car class's recall is $\frac{3}{3+0} = 1$, suggesting that the model can identify all ground-truth cars. However, as illustrated, the recall value fails to account for the model's misclassification of a non-car (human) object as a car. This error can be detected by the false positive (FP) term in the precision metric. Therefore, precision and recall are typically used together to evaluate an object detection model.

Using the definition of precision and recall, we noted that the two metrics measure the model's accuracy in classifying objects of a particular class at a particular confidence threshold $t_{confidence}$. We denote objects belonging to this class referred to as positive samples, while those that do not belong are negative samples. Based on Equation **??**, we can see that as the precision approaches 1, the true positive (TP) term increase while the false positive (FP) term decrease. This implies that the model becomes more confident in classifying a sample as positive as precision increases. On the other hand, Equation **??** shows that as the true positive (TP) term increases and the false negative (FN) term decreases, the recall approaches 1. This indicates that a higher recall value means the model identifies more samples as positive.

3.2.7 Precision-Recall Curve

As it might be seen, there is a tradeoff between precision and recall. Suppose we establish a high $t_{confidence}$ value for the model and only classify a sample as positive when its confidence score surpasses this threshold. In this case, the model may miss multiple ground-truth positive samples, resulting in high precision but low recall. Conversely, suppose we set a low $t_{confidence}$ value; then, the

model may classify most samples as positive, even those that are incorrectly classified, resulting in the model having low precision but high recall. Nonetheless, an object detector is deemed as high performance at a particular IoU threshold if its precision remains high as its recall increases [**metrics_survey_2020**]. Since the model's precision and recall vary depending on the confidence threshold $t_{confidence}$, we can calculate both metrics for each threshold value to examine the tradeoff between optimizing for one over the other. These computed metric values can be plotted on a two-dimensional plane to visualize the tradeoff, also known as the **precision-recall curve**. Figure **??** shows examples of the precision-recall curve for a dataset and how the curve change at different IoU threshold value.

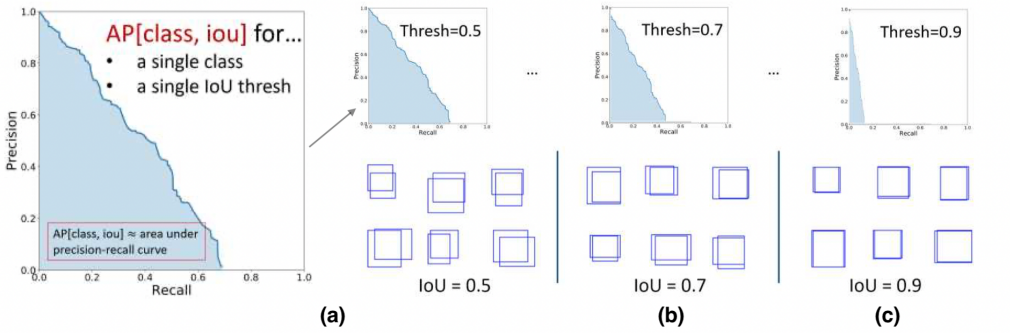


Figure 3.2: **(a)** The precision-recall curve for a single class at the IoU threshold of 0.5 for a dataset. **(b, c)** the same precision-recall curve as (a) but with IoU threshold of 0.7 and 0.9, respectively. [**szeliski_cv_book**]

3.2.8 F-score (F1-score) Metric

Optimizing a model for precision or recall, similar to choosing an IoU threshold value, heavily depends on the task the model tries to complete. If the model is used for a high-accuracy and detailed task, such as surgery, it should be optimized for precision metrics. On the other hand, if the task is more focused on catching as many instances as possible, such as low-cost home-based cancer diagnosis, the model should be optimized for recall. In cases where both precision and recall are equally important, we can determine the optimal combination of the two metrics by comparing their F-scores at different confidence thresholds. The F-score is the harmonic mean, a type of numerical average, of the precision and recall metrics [**fscore_2017**]. For each pair of precision and recall, the F-score can be calculated as follows:

$$\text{F-score} = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{3.5}$$

Given that the F-score is the harmonic mean of precision and recall metrics for a specific class at a confidence threshold $t_{confidence}$, comparing the F-score at different $t_{confidence}$ values helps determine the optimal combination of precision and recall. The highest F-score obtained represents this optimal combination, allowing us to select the ideal confidence threshold value.

3.2.9 Average Precision (AP) Metric

Another metric that can be used to evaluate the performance of a model for each class is the average precision (AP). While the F-score describes the confidence threshold value at which the model achieves the highest performance for a particular class, the average precision (AP) summarizes the model's performance across various confidence thresholds for that class. In other words, the AP condenses the precision-recall curve for a specific class into a single value representing the average of all precisions. The AP is an estimate of the area under the precision-recall curve. There are different versions of AP; the adopted version for PASCAL VOC and COCO benchmark is the N-point interpolation AP [**n_point_interpolation_ap**]. Let P and R denote precision and recall, respectively. The N-point interpolation AP is defined as follows:

$$AP_N = \frac{1}{N} \sum_{R \in \mathbb{R}_N} P_{interp}(R) \qquad with \qquad P_{interp}(R) = \max_{R':R' \geq R} P(R') \tag{3.6}$$

where the set of N interpolation $\mathbb{R}_N$ is $\left\{0, \frac{1}{N}, \frac{2}{N}, ..., \frac{N}{N}\right\}$. The term $P(R')$ is the value of precision at recall R'. The condition $\max_{R':R' \geq R}$ implies that the $P_{interp}(R)$ is the highest precision value among all recall point R' that are larger than R, instead of being the precision observed at the recall R [**n_point_interpolation_ap**]. As an example, consider using 11-point interpolation to the precision-recall curve shown in Figure **??**. With this precision-recall curve, the 11-point interpolation AP is:

$$AP_{11} = \frac{1}{11}(1 + 0.666 + 3 \times 0.428 + 6 \times 0) \approx 26.818\%$$

We noted that instead of using the precision value at each recall in $\left\{0, \frac{1}{10}, \frac{2}{10}, ..., \frac{10}{10}\right\}$, the AP_{11} use the maximum precision value of all the remaining recall points. While the N-point interpolation technique simplifies the computation of the average precision (AP), it is an approximation method that yields a non-differentiable AP value. The difficulties in optimizing non-differentiable values make it challenging to use AP as a loss function. To address this limitation, recent research has proposed metrics like probability-based detection quality (PDQ) and Smooth-AP [**pdq_metric_2020**, **smoth_ap_metric_2020**].

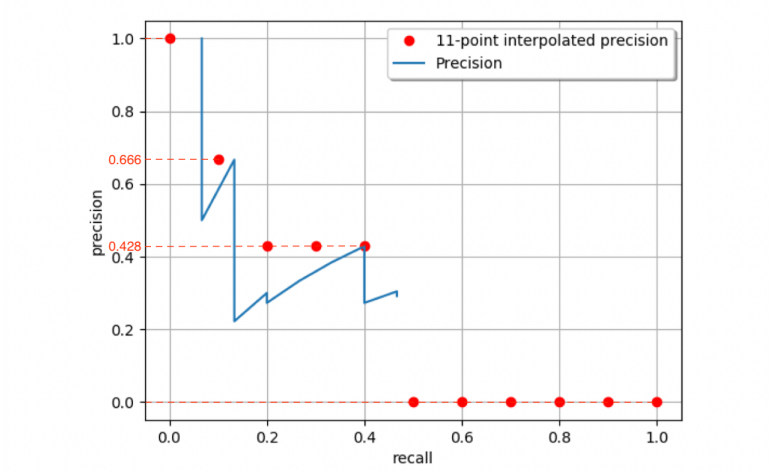


Figure 3.3: Applied 11-point interpolation method on the precision-recall curve **[metrics_survey_2020]**

3.2.10 Mean Average Precision (mAP) Metric

The mean average precision (mAP) of a class can be calculated using the average precision (AP) for that class at a specified IoU threshold **[szeliski_cv_book]**. The mAP quantifies the model's accuracy in detecting objects of all classes above a particular IoU threshold. Let K be the number of classes our model is able to predict for, then mAP is defined as follows:

$$mAP = \frac{1}{K} \sum_{i=1}^{i=K} AP_i \tag{3.7}$$

where AP_i is the average precision value of the ith class.

In summary, evaluating object detection models is a five-step process. First, the Intersection over Union (IoU) between the predicted and ground-truth bounding boxes is calculated to determine the detection's location correctness. Second, a confusion matrix is created by comparing the predicted labels against the ground-truth labels at each IoU and confidence threshold. Third, the model's reliability and sensitivity can be quantified from the confusion matrix at each combination of the two thresholds by computing the precision and recall metrics. Moreover, the precision-recall curve and F-score can be computed to offer insights into the balance between the model's reliability and sensitivity. Fourth, the average precision (AP) metric is calculated to quantify the model's accuracy in detecting objects of a specific class at each IoU threshold. Finally, the mean average precision (mAP) metric represents the model's accuracy in detecting objects across all classes at a given IoU threshold.

In comparison to object detection, instance segmentation models provide a more detailed output by generating a pixel-wise mask of the object, along with the object's bounding box and classification label. Therefore, to fully assess the performance of an instance segmentation model, we need to evaluate the accuracy of the detection at the pixel level. Similar to the bounding box, the accuracy of the object's mask can be measured using the Intersection over Union (IoU) metric, denoted as *mask IoU* [**instance_segementation_metric_2022**, **mask_rcnn_2017**]. The IoU for the object's mask is the ratio of overlapping pixels in the predicted and ground-truth masks over the total number of pixels in both. Consider representing the masks' bounding box as matrices, denoting a value of 1 to the pixels belonging to the object (i.e., inside the mask) and 0 to the pixels not (i.e., outside the mask). The union of the two mask matrices is the number of 1s in either matrix. The intersection is the number of 1s present in the output matrix of the element-wise product (Hadamard product) between the two masks' matrices. After computing the IoU for each pair of object masks, the subsequent steps for evaluating the model are identical to those used for object detection. This entails calculating the confusion matrix, precision-recall curve, F-score, AP, and mAP, but utilizing the mask's IoU rather than the bounding box's IoU.

With the understanding of object detection, instance segmentation, and their evaluation metrics, we will discuss R-CNN and YOLO, the two most popular algorithm families in these tasks, in Chapters **??** and **??**, respectively. In Section **??**, we discussed the structure and building blocks of convolutional neural networks (CNNs). We also stated how CNNs classify objects in the image classification task in the discussion. However, the image classification task assumes the image has exactly one object, and the model classifies the entire image based on that one object. Therefore, if we consider each bounding box as its own image, we can utilize a CNN to identify the object's class within the bounding box. This is the main idea behind the different variations of R-CNN and YOLO, which are designed for object detection and instance segmentation task.

CHAPTER 4

R-CNN Variation

Since the interested domain for this paper is the object detection and instance segmentation task, we will analyze the Mask R-CNN. Mask R-CNN is a popular algorithm that tries to solve both object detection and instance segmentation problem in the Computer Vision field. However, Mask R-CNN is the improved version of Faster R-CNN and Fast R-CNN, which is based on the R-CNN algorithm. Therefore, to fully understand Mask R-CNN, we will start with understanding the building block of R-CNN, which is designed for object detection tasks.

4.1 R-CNN

R-CNN, also known as regional-based convolutional neural networks, is an object detection algorithm. The algorithm was developed by a group of researchers at UC Berkeley in 2015. Since its development, the R-CNN model has revolutionized the field of computer vision. The model was designed to detect up to 80 different types of objects in images and provide large-scale object recognition capabilities. Additionally, before the existence of RCNN, most algorithms in object recognition task used support vector machine (SVM) with blockwise orientation histograms like Histogram of Oriented Gradients (HOG) [**svm_hog**] or Scale-Invariant Feature Transform (SIFT) [**svm_sift**]. SVM dominated the space until 2012. In 2012, a CNN algorithm showed an astounding image classification accuracy on ImageNet Large Scale Visual Recognition Challenge (ILSVRC). Since then, more studies have been created into developing and improving algorithms utilizing CNN. The R-CNN model is our first attempt to build an object detection model that extracts features using a pre-trained CNN.

In a board picture, the R-CNN model is composed of three modules [Fig. **??**]. The first module's purpose is to generate regions of interest (RoI), i.e., regions that possibly contain an object. The second module utilizes a CNN to extract out feature vector from each proposed region. The third

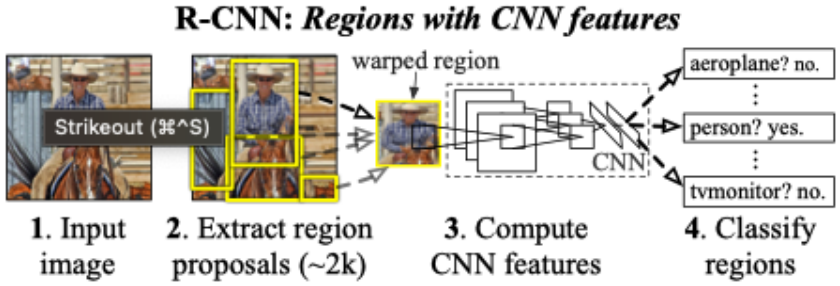


Figure 4.1: R-CNN overall architecture **[Girshick_R_CNN_2013]**

module then performs classification for each region using a pre-trained SVM algorithm on the feature vectors provided by the second module. As we can see R-CNN algorithm tries to find the location of each object, extract the object features, and classify these features. Next, we will dive deeper into each module of R-CNN.

4.1.1 The First Module: Region Proposal

In the first module, the algorithm used in this step must be able to propose some region of interest (RoI). A region of interest is a smaller part of the original image that could contain an object.

Given this problem, one might consider the bruce-force method by examining every small rectangle area of the original image as a potential region of interest in a systematic way. This method is the main concept behind the sliding-window technique, a type of exhaustive search algorithm used in the region proposal task. The sliding window technique involves running a window of a predefined size across the image, such that each subsection of the image is considered as a potential region proposal. Like the CNN kernel, the windows' size, stride, and padding are all customizable and can be adjusted to suit different types of images. In addition, multiple scales can also be incorporated into the sliding window technique, which allows for more robust performance in scenarios where objects may appear at different sizes within the same image. Since the sliding window technique examines every location within the image by considering every pixel as part of some RoI, it will not miss any potential object location. However, considering every location in the image as an RoI is unnecessary due to objects most likely not appearing everywhere in the image. Furthermore, classifying every sub-image at different sizes will require tremendous computational power. Therefore, even though it allows our model to detect all possible locations, the sliding window technique should not be used in autonomous cars' vision because it is computationally expensive.

In recent years, many studies have shown interest in improving the efficiency of the region proposal algorithm. One notable algorithm that gives the same strength as the sliding window technique is selective search. The selective search was designed to combine the strength of both exhaustive search and segmentation [**selective_search_2013**]. The strength that selective search inherits from sliding windows is the ability to find all the possible locations that can be a potential region of interest. Additionally, selective search utilizes the underlying image structure to cluster pixels into different regions, taken from the strength of the segmentation technique. Selective search also aims to complete three goals. These three goals are to capture all scales, diversification, and fast to compute. Capture all scale is the idea that the algorithm must be able to detect objects of different sizes in the image. Next, diversification requirements refer to a method of grouping regions. The algorithm must be able to combine regions containing part of an object into one region. Additionally, for instance, like a person inside a car or a person in front of a car, the algorithm must be able to separate the region for the car and the region for the person under diversification requirement. Lastly, fast computing requires the algorithm not to demand heavy computational power.

As for selective search overall behavior, it can be thought of as two steps. The first step is to perform **bottom-up segmentation** – described in Efficient Graph-Based Image Segmentation by Felzenszwalb and Huttenlocher [**felzenszwalb_huttenlocher_2004**] – to generate initial sub-segmentation. The second step is to combine similar sub-segmentation recursive using a similarity score between subsegments. The similarity score is a combination of four similarity grouping criteria. These four criteria are color similarity, texture similarity, size similarity, and fill similarity [**selective_search_2013**]. The reason that color and texture similarity between regions is needed is that the same object most likely will have the same texture or shade of color. In the original paper, the color and texture similarity criteria score can be described by an equation only based on a fixed number of values taken from the histogram of each color channel. Thus, these two similarity scores require minimal to no computational power to compute. Similarly, two regions that have the majority of the area overlap with each other are most likely described the same object. Therefore, the use of a fill similarity score allows the algorithm to merge regions that mostly overlap and keep regions that are not overlapping separate from each other. The fill similarity score can be calculated using the following equation:

$$S_{fill}(r_i, r_j) = 1 - \frac{size(BB_{ij}) - size(r_i) - size(r_j)}{size(image)}$$

where r_i, r_j are two considering RoI, and BB_{ij} is the bounding box around i and j. Lastly, the similarity

in size between the two regions takes into consideration to make sure that the algorithm identifies all locations for different objects at different scales. The size similarity score can be calculated using the following equation:

$$S_{size}(r_i, r_j) = 1 - \frac{size(r_i) + size(r_j)}{size(image)}$$

where r_i, r_j are two considering RoI. As we can see, the similarity score computation is fast and take into account the diversification of the image data set. With these two steps, the selective search can quickly classify foreground and background, then downsample the number of potential RoI that existed in foreground segmentation. Therefore, utilizing selective search allow the model to reduce the number of falsified RoI compared to sliding window and requires lower computational power.

In the original paper of R-CNN, in the first module – region proposal – utilize selective search algorithm [**Girshick_R_CNN_2013**]. However, as mentioned before, the R-CNN model can be thought of as three separate modules; thus, one can experiment with R-CNN with different region proposal techniques.

4.1.2 The Second Module: Feature Extraction With CNN

On completion of the first module of R-CNN, which generates all regions of interest within an image, these RoIs are passed to the AlexNet architect for feature extraction [**Girshick_R_CNN_2013**]. AlexNet, a CNN architect, was proposed by Alex Krizhevsky at the University of Toronto in 2012 with the guidance of Ilya Sutskever and Geoffrey E. Hinton [**AlexNet_2017**]. AlexNet was trained using ImageNet, which at the time was the most extensive picture data collection. As AlexNet required its input to have a fixed resolution with the same width and height, the photos in ImageNet were resized to 256×256 pixels. Prior to publication, AlexNet competed in ILSVRC-2010 and obtained top-one error rates of 37.5%. In other words, 37.5% of the time, the model assigns the highest score to the correct label. In the same competition, AlexNet also achieves top-5 error rates of 17.0%, i.e., the correct label in the top 5 predicts 17% of the time.

AlexNet is regarded as one of CNN's most significant innovations. AlexNet, with 60 million parameters and 650,000 neurons as of 2012, is one of the largest neural networks ever suggested. The architecture of AlexNet consists of eight learned layers. Five convolutional layers are followed by three layers that are fully connected. Due to hardware limitations at the time, it was not possible to fit a massive network like AlexNet on a single GPU. For this reason, the model's architecture is distributed across two separate GPUs, and training must pass via both GPUs. Due to this limitation, for each layer in the network, the model has half of the neurons for that layer on each GPU and only

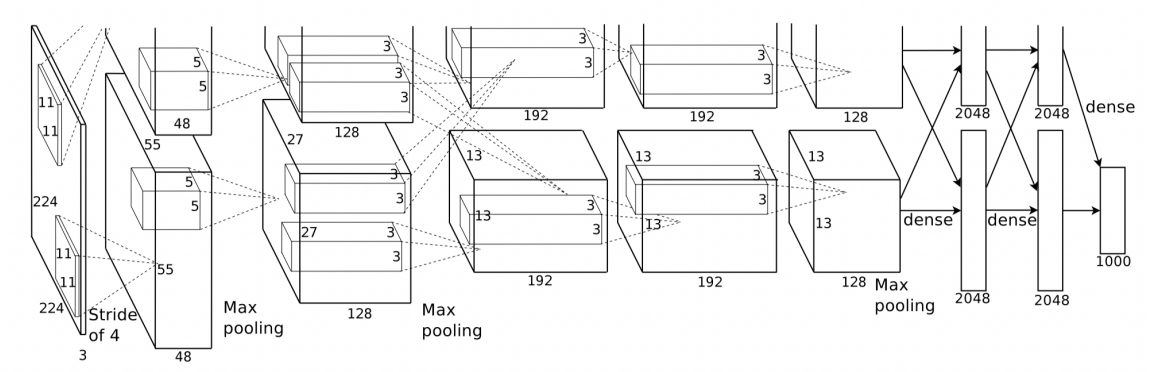


Figure 4.2: AlexNet's overall architecture [**AlexNet_2017**]

requires particular layers to perform GPU communication. For example, in the overall architecture of AlexNet presented in Figure **??**, we note that neurons of the first layer only connect to neurons of the second layer on the same GPU. On the contrary, we notice that all neurons of the second layer are fully connected with neurons in the third layer. The notion of spreading neurons across multiple GPUs is known as the parallelization scheme. The parallelization scheme is a topic that remains outside the scope of this paper, and GPU memory size is no longer a significant problem due to the current advancement of technology. Therefore, understanding the parallelization scheme is likely to be optional for the analysis of a CNN model.

In addition to being one of the largest networks and employing a parallelization strategy, AlexNet was among the first neural networks to employ ReLUs. The ReLUs activation function was employed instead of the more common *Tanh* and *Sigmoid* functions at the time because AlexNet was designed to reduce learning time. Using ReLUs, the model has a faster learning rate, which, according to the author, would vastly increase the performance of large models with a large data set. In addition, AlexNet implements local response normalization (LRN) to assist ReLUs in generalization. The non-trainable LRN layer is utilized following the first and second network layers. Following the first, second, and fifth layers, AlexNet employs a max-pooling layer with a size of 3 and a stride of two 2. The author notes that having an overlapping pooling layer — i.e., size 3 > stride 2 — decreases overfitting in general [**AlexNet_2017**].

AlexNet employs data augmentation and dropout techniques to address the issue of overfitting when training an extensive network with a large data set. For data augmentation, AlexNet randomly selects images into batches and resizes them to a resolution of 224×224. It then generates a copy with horizontal reflection image transformation and a copy with modified intensity for the three RGB channels. The CPU generates these transformed images while the GPU trains on the previous batch,

allowing the data augmentation process to be done without incurring any additional performance costs. In addition to data augmentation, AlexNet also uses the dropout technique. If the output of any hidden neurons is less than or equal to 0.5, the network sets its output to 0. This technique reduces the computation required because neurons with 0 need not be considered in the rest of the forward pass or updated during backpropagation. This method also permits neurons in the network to be independent of one another, as no neurons are guaranteed to persist with each training sample.

R-CNN model utilizes the AlexNet architect implemented on the Caffe framework. R-CNN model passes each generated RoI as a separate image to AlexNet for feature extraction. Since the RoI role is to find each object location in the image, thus AlexNet can assume each RoI only have exactly one object. AlexNet also requires image input of fixed resolution 224×224; thus, R-CNN transform to the required size by warping all pixels in a bounding box of 227×227.

4.1.3 The Third Module: Classification With Pre-trained SVM

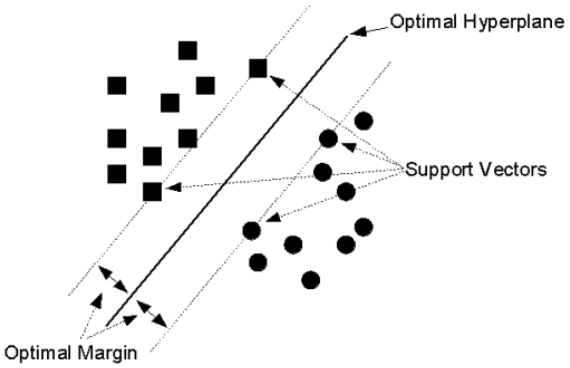


Figure 4.3: 2D linear SVM visualization **[2d_svm_Tzotsos]**

After the second module, R-CNN is able to obtain multiple features for each RoI. Each RoI with extracted features then be given to each trained linear SVM classifier in each class for evaluation. Since each class has its own SVM classifier, thus the SVM only needs to distinguish between objects belonging to the class and objects that are not belonging to the class. Linear SVM classifier can be thought of as trying to draw a $N-1$ dimension separator in the N-dimension space where N is the number of features of a specific class. The separator in SVM must be linear, and it tries to separate objects in the class and objects outside of the class. The best-fit separator has the highest margin between any point in the class and any point outside of the class. An example of 2-dimensional linear SVM is visualized by figure **??** where the circle represents the object in this class, and the

square is the object outside the class. Given that the class-specific SVM know what features the class possesses. The separation between objects in and out of the class can be done with little computation power, as we already have the extracted features. Each class SVM is then given the RoI a score representing the likelihood that the object in RoI belongs to this class. Therefore, for each RoI, the class with the highest SVM score is assigned to be the class for the object.

4.1.4 R-CNN Result and Drawback

On the PASCAL VOC dataset 2012, by combining these three modules, R-CNN achieved 53.3% in **mean Average Precision (mAP)**, a metric that measures the overall accuracy of the object detection network mentioned in Sec. Evaluation section number. R-CNN also achieves the mAP of 31,4% and ranks first on the ILSVRC2013 dataset in terms of accuracy [**Girshick_R_CNN_2013**]. Despite achieving an incredible breakthrough in using Convolutional Networks and having a high object detection accuracy, R-CNN's performance is marred by a number of disadvantages. These issues include multi-stage training, high runtime and space complexity, reliance on non-learnable algorithms, and slowness [**fast_rcnn_og**].

As described by the architecture of R-CNN, the network is divided into three modules and runs sequentially. The network attempts to feed the input of one module with the module's output coming before it. In other words, the first module must completely process the image before the second module is running. Therefore, the network's modules must wait on one another and be trained individually, thus creating the multi-stage training problem. Secondly, since the first module generates 2000 proposed RoIs before the second module can start running, the networks must cache these proposed RoIs on the disk. Similarly, the generated feature for each RoI must be stored on the disk before processing by SVM. The need to write and read multiple time for each RoI cause a high order of runtime and space complexity. The runtime and space complexity is even higher when we consider overlapping RoIs; the network recalculates features and classification for the overlapping portion of overlapping RoIs. Thirdly, the selective search algorithm used in the first module is a non-learnable algorithm. Therefore, the algorithm runtime and accuracy would not improve through training the network. Additionally, through the error analysis, the author notices the mass amount of localization inaccuracy. Thus, for each proposed region, after being scored by the SVMs, the region is piped to a class-specific bounding-box regressor to generate a new bounding box. Finding the bounding box within the region of interest allows the model to improve localization accuracy but exacerbates the model runtime performance as it generates the bounding box at least

twice for each object in the image. Lastly, with a processing speed of 47s per image, the R-CNN model is slow and thus has limited real-world application.

4.2 Fast R-CNN

The author of R-CNN later implemented Fast R-CNN to reduce the runtime and space complexity while improving detection accuracy. Fast R-CNN is implemented in Python and C++. Like the R-CNN model, Fast R-CNN can be used along with any convolutional neural network. In the proposal Fast R-CNN model, the author utilized VGG16, one of the deepest CNN in 2015, as the backbone CNN for the model. Comparing the performance of Fast R-CNN with VGG16 versus R-CNN with VGG16 on the PASCAL VOC 2012 dataset while having the same setup, the experiment showed that Fast R-CNN is 9 times faster at train-time and 213 times faster at test-time while achieving a higher mAP score [**fast_rcnn_og**]. In the next section, we will mention the keynote of VGG16 architecture, followed by the discussion of the Fast R-CNN model and its design decisions that lead to a higher mAP.

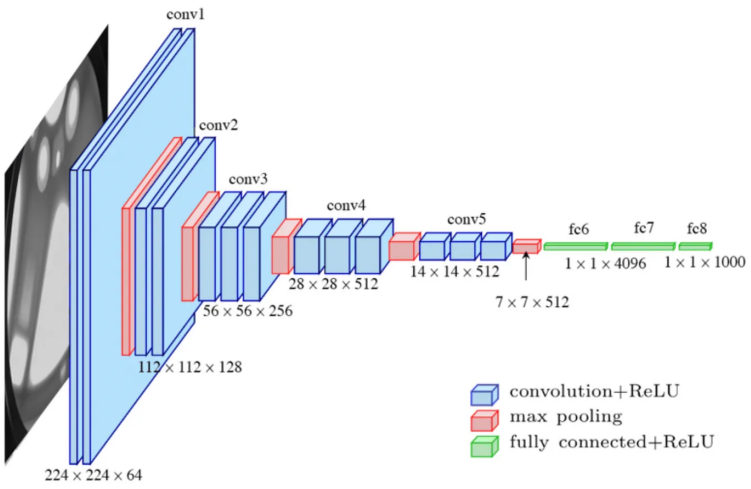
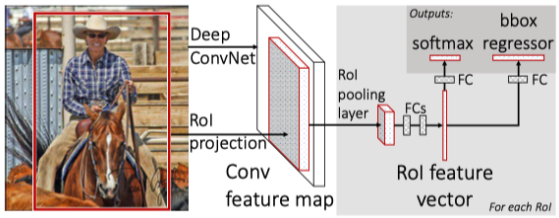


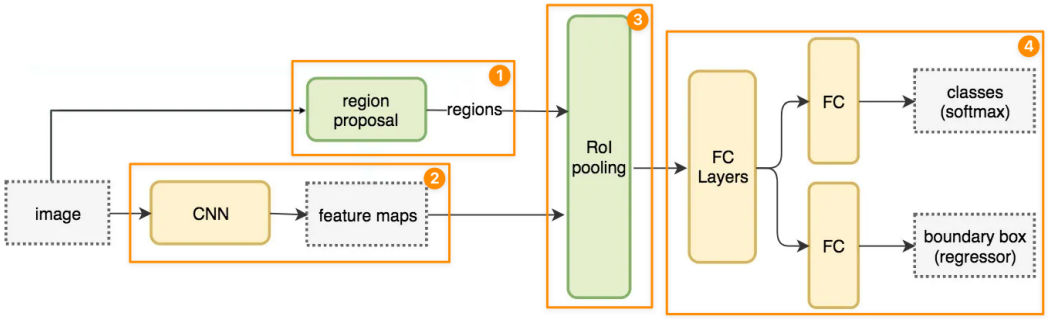
Figure 4.4: VGG16 architecture [**vgg16_architect_2014**]

The VGG16 is a CNN model developed by the Visual Geometry Group (VGG) at the University of Oxford in 2014 [**vgg16_2014**]. The VGG16 architecture is notably deeper than AlexNet, comprising 16 learnable layers – 13 convolutional layers and 3 fully connected layers – and 5 non-trainable max-pooling layers [Fig. **??**]. VGG16 takes an RGB 224×224 image as input and generates a 7×7 feature map, downsampled by a factor of 32 from the input image resolution [**deconv_rcnn_2018**]. A set of fully connected layers is then processed in this feature map to produce the predicted

classification label for the image. Unlike AlexNet, which uses a combination of 3×3 and 5×5 filters, VGG16 uses only 3×3 filters throughout the network with smaller strides and padding. By employing a strategy like utilizing a pair of stacked 3×3 layers in place of a single 5×5 layer, VGG16's design enabled it to demonstrate that elevating the depth of a network to 16 layers can yield substantial improvements for existing CNNs. Fast R-CNN adapts any CNN model for object detection by performing three changes. The first change is replacing the last pooling layer with an RoI pooling layer (named RoIPool). For VGG16, an RoI will be used in place of the $7 \times 7 \times 512$ max-pool layer in Fast R-CNN. The second change is replacing the last fully connected layer with two sibling layers, i.e., replacing the fc8 layer for VGG16. Lastly, Fast R-CNN will adjust the input layer of the CNN model to accept images of any size and proposed RoIs for that particular image as input. We will discuss these changes in more detail later in this section.



(a) Overall architecture [**fast_rcnn_og**]

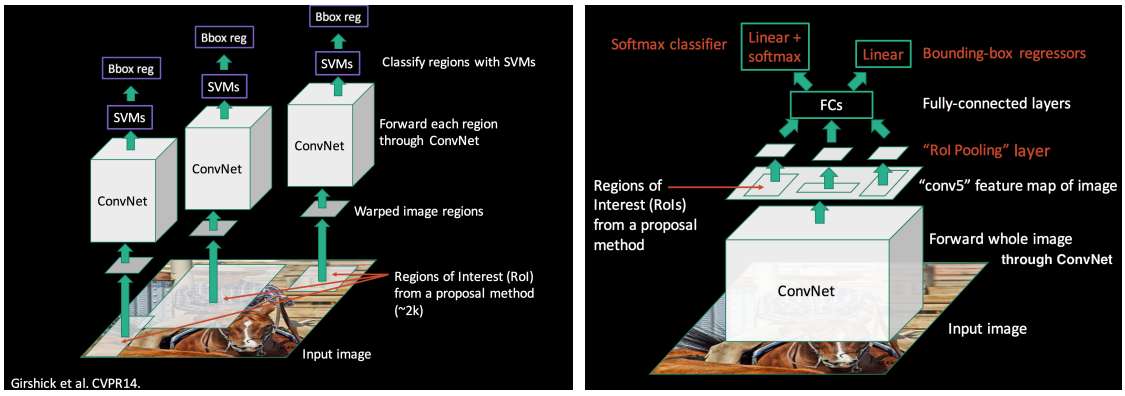


(b) Network flow [**rcnn_vari_flow_chart**]

Figure 4.5: Fast R-CNN overall architecture and network flow

The overall architecture of Fast R-CNN can be thought of as four stages [Fig. **??**]. The network takes an image and a set of RoI as inputs. Comparing Fast R-CNN with R-CNN, which takes an image as an input and then generates RoIs with selective search, the type of input data between the two models is not equivalent. Thus we assume Fast R-CNN takes an image as input and performs the selective search algorithm as the first stage of the model. In the second stage, Fast R-CNN generates a feature map for the entire image by running the input image through a CNN. The CNN

used for Fast R-CNN performance measurement in the original paper is VGG16. In the third stage, the model uses RoIPool layer to extract the feature grid corresponding to the proposed RoI from the image feature map generated in stage two for each proposed RoI. The RoIPool layer is also used for downsampling the RoI feature grid of any size to a pre-defined fixed-length feature vector. In the fourth stage, each RoI feature vector is processed by multiple fully connected layers and then branched into the two sibling output layers – softmax classification and bounding box regression. The model's learning with two output branches is possible with the use of multi-task loss.



(a) R-CNN architecture with stages **(b)** Fast R-CNN architecture with stages

Figure 4.6: R-CNN vs. Fast R-CNN architecture comparison [**rcnn_vs_faster_custom_fig**]

Comparing the architecture of R-CNN and Fast R-CNN, there are three main differences between the two models [Fig. **??**]. The first difference is that Fast R-CNN generates the feature map for the entire input image instead of each RoI individually. This means Fast R-CNN only applies CNN to image one and shares the feature maps across RoIs. Fast R-CNN behavior holds several advantages compared to treating RoIs individually, like in the R-CNN model. These advantages are reducing the use of disk storage, reducing redundancy operation performed on overlapping RoI, and sharing computation power and memory used between RoIs in the same image, thus improving the performance at test-time [**fast_rcnn_og**]. Sharing the feature map and memory data between RoIs also allows the model to be trained faster. The author reported that training Fast R-CNN by examining multiple RoIs in an image allows the model to convert roughly 64 times faster compared to when trained with RoIs from different images.

The second difference is the inclusion of the RoI pooling layer (named RoIPool). This layer is responsible for extracting RoI feature grids of varying sizes and downsampling them to a fixed pre-defined size. To extract the RoI feature grid for any input RoI, the RoIPool layer first maps the

top left corner point of the input RoI box, which is defined on the input image, to a corresponding pixel in the feature map. Then, the width and height of the input RoI are reduced by the same factor that the feature map is downsampled from the original image. As we will perform a max-pooling operation on the pixels' value, the size of the original projected RoI must be rounded down to the nearest integer because we cannot take a partitioned pixel. After projecting the top left corner and rounding the scaled-down RoI size from the input layer onto the feature map, we have the offset rounded projected RoI. The RoI feature grid is then formed by every feature map pixel that lies inside this offset rounded projected RoI. Since objects in the input image can have different sizes and aspect ratios, thus the projected RoI feature grid can also vary in size. However, the input of a fully connected layer must be of the same pre-defined size, which is why the input's size-independent downsampling operation is necessary. The RoIPool layer achieves size-independent downsampling by dividing the RoI feature grid into a pre-defined $W \times H$ grid of the RoI bin (or RoI bin), where $W \times H$ is the required dimension for the following fully connected layer input. In other words, the layer divides the $w \times h$ RoI feature grid into RoI bins of equal size, each with an approximate size of $\frac{w}{W} \times \frac{h}{H}$. Here, $\frac{w}{W}$ and $\frac{h}{H}$ represent the number of pixels along the width and height of the RoI bin, respectively, and are rounded down to the nearest integer. In contrast to the traditional pooling layer described in Sec. **??**, which used a sliding technique dependent on the input size, the division into a grid of equal RoI bins enables the RoIPool layer to have a fixed output grid size regardless of the size of the layer's input RoI. The RoIPool layer then applies max-pooling to the pixel values in each RoI bin, thereby effectively reducing the size of any projected RoI to a pre-defined $W \times H$ size.

The third difference is going from multi-stage training in R-CNN to single-stage multi-task training in Fast R-CNN. In R-CNN, the model must be completely trained with class-specific SVM before being trained with class-specific bounding box regressor, and these tasks also are performed in the same sequence in the test time. On the contrary, Faster R-CNN has the softmax classifier and bounding box regressor as sibling output layers. Fast R-CNN model generates a multi-task loss L for each RoI and uses the loss L as a metric to jointly train both the softmax classifier and bounding box regressor branches. The multi-task loss L is generated from the difference between the truth label, truth box, and predicted label, predicted box perspectively. The author suggested that employing a multi-task learning scheme would improve performance, as the network's shared components must be general and precise enough to produce correct results for both classifier and bounding box regressor branches [**fast_rcnn_og**]. The author also reports that Fast R-CNN with multi-task learning consistently achieved higher mAP scores than stage-wise training across different CNN implementations.

These changes in architecture allow the Fast R-CNN model to achieve a processing runtime of 0.3 seconds per image, excluding the time needed for object proposal [**fast_rcnn_og**]. However, when factoring in the runtime for object proposal, such as the Selective Search algorithm, Fast R-CNN is almost 7.67 times slower, taking 2.3 seconds per image [**selective_search_2013**]. Additionally, the RoIPool layer in Fast R-CNN causes the model to undergo quantization. Quantization is the process of reducing the precision of an input from a large set of possible values to a smaller set of discrete values. Recall that the RoIPool layer initially projects the input RoI box to the appropriate location, rounded down to the nearest pixel in the feature map. The layer then subdivides the projected RoI, expressing the size of each RoI bin in the projected RoI in terms of pixels. In other words, the RoIPool layer quantizes these sizes and coordinates from the continuous non-negative real number set to the discrete natural number set. While quantization enables the layer to perform a valid max-pooling operation on pixel values, it also introduces a loss of precision and information in our model. The loss in precision is caused by the projected RoI deviating slightly from its actual coordinate. The loss in pixel data is due to the RoI RoI feature grid cannot always be divided perfectly without remainder.

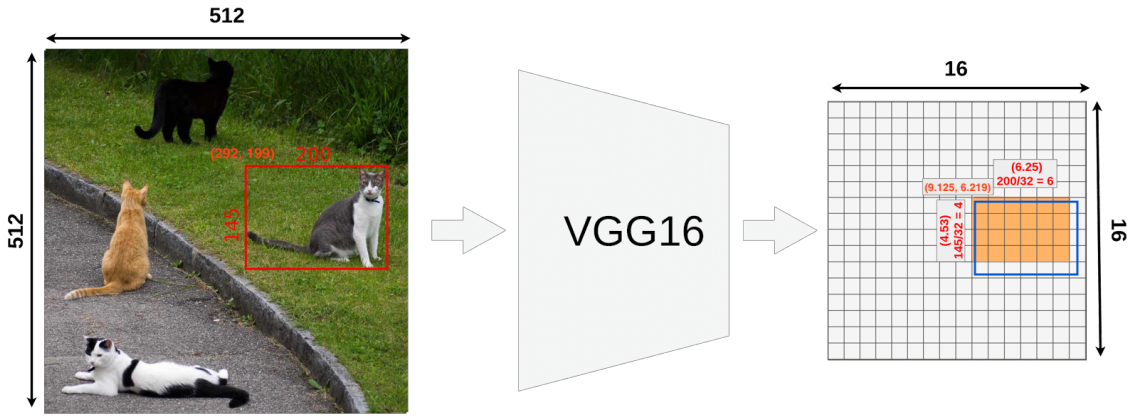


Figure 4.7: RoI projection to feature map [**roi_pooling_problem**]

As an example, assume that we are processing an input image of 512×512 with VGG16, and the first fully connected layer expects 3×3 as input [Fig. **??**]. Consider the input RoI with the top left corner coordinate of $(292, 199)$ and the size of 145×200. As mentioned earlier, VGG16 has a scale-down factor of 32 from the input image to the feature map space. With this information, we

can compute the corresponding coordinate and size of the input RoI in the feature map as follows:

$$\text{Feature map RoI corner coordinate:} \quad \left(\left\lfloor \frac{292}{32} \right\rfloor, \left\lfloor \frac{199}{32} \right\rfloor\right) = (\lfloor 9.125 \rfloor, \lfloor 6.21875 \rfloor) = (9, 6) \quad (4.1)$$

$$\text{Feature map RoI size:} \quad \left\lfloor \frac{200}{32} \right\rfloor \times \left\lfloor \frac{145}{32} \right\rfloor = \lfloor 6.25 \rfloor \times \lfloor 4.53125 \rfloor = 6 \times 4 \quad (4.2)$$

When projecting to the feature map space, Fast R-CNN offsets and resizes the original projected RoI (shown as the blue rectangle in Fig. **??**) to align perfectly with n feature map pixels (shown as the orange area in Fig. **??**), where n is the number of feature map pixels that can fit entirely within the original projected RoI. As shown in Fig. **??**, we lose some pixels data (the white part inside the blue rectangle) and have additional noise data (the orange part outside the blue rectangle). Consider the quantization of the conner's vertical position at 6.21875 to 6 in feature map space. When we factor in the scaling of 32 times, this means our projected RoI is shifted by $(6.21875 - 6) \times 32 = 7$ pixels vertically compared to the input RoI in the original image space. Similarly, the projected RoI is offset by 4 pixels horizontally, and the quantization process results in a loss of $(0.25 + 0.53125) \times 32 = 25$ pixels during projection.

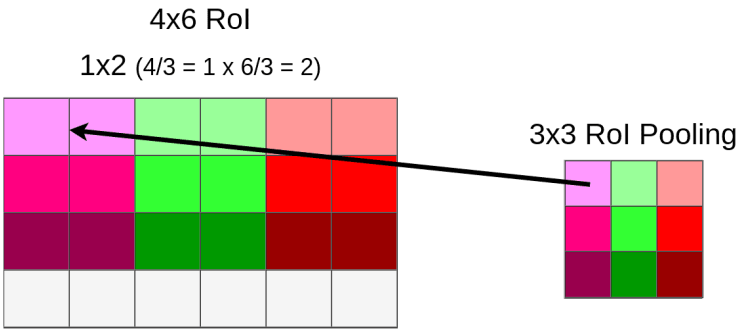


Figure 4.8: The RoI feature grid is divided into RoI bins, as shown on the left. Performing max-pooling on each RoI bin results in a 3×3 output matrix, as shown on the right. Each cell in the input image represents a feature map pixel, and each RoI bin contains 2 pixels and has a unique color. Each cell in the output 3×3 matrix represents the highest feature map pixel out of all pixels in each corresponding RoI bin. The white cells on the left do not belong to any RoI bin and are not used in the RoI pooling process.[**roi_pooling_problem**]

After projecting and quantizing the input RoI, we obtain an RoI feature grid. In our example, this grid has dimensions of 6×4 and is located at position (9, 6). However, the next layer in our network requires inputs of size 3×3. To accommodate this, we divide the RoI feature grid into a grid of RoI bins, each with a size of $\frac{6}{3} \times \frac{4}{3} \approx 2 \times 1.3333$. Since these sizes are expressed in terms of pixel counts, we must quantize them to the nearest integer values, resulting in RoI bins of size 2×1.

As shown in Fig. **??**, this quantization results in the loss of information for the bottom row of pixels in the RoI feature grid, corresponding to a loss of $6 \times 32 = 192$ pixels in the input image space. In addition, the pooling operation can also cause small offsets in the position of the RoI bins, leading to further loss of information. Combined with every loss that happens in the RoIPool layer, the model has loss $192 + 25 = 217$ pixels. In addition, the RoI used for classification is offset by 7 pixels vertically and 4 pixels horizontally. While the loss of 217 pixels due to quantization and pooling may seem small compared to 29,000 pixels in the input RoI of size 200×145, thereby can be overlooked in the object detection task. However, as the goal of our study lies in the instance segmentation task, where the model outputs a mask containing every pixel belonging to the object, every pixel counts. Nonetheless, these losses in information and offset are for one input RoI. Due to the fact that object identification tasks typically detect multiple objects per image, the loss of information and offset caused by quantization may compromise the quality of the model.

To address the quantization issue in the RoIPool layer, RoIAlign was proposed as a method for achieving size-independent downsampling without quantization. The RoIAlign is utilized with Mask R-CNN, an instance segmentation model that will be discussed in greater detail in Sec. **??**. Since Mask R-CNN is built on top of Faster R-CNN, which is also the subsequent significant improvement over the Fast R-CNN model, we will discuss the Faster R-CNN model in the following section. Faster R-CNN proposes the addition of the region proposal network (RPN) to resolve the bottleneck in object detection performance caused by object proposal runtime [**faster_rcnn_2015**]. Instead of attempting to reduce the runtime of the object proposal algorithm, RPN's primary objective is to share computation with the object classification module, thereby allowing the object detection model to generate object proposals at no additional cost. We will further discuss the region proposal network (RPN) used in conjunction with Fast R-CNN to create Faster R-CNN in section **??**.

4.3 Faster R-CNN

In 2016, the object detection Faster R-CNN model was proposed as an improvement over the Fast R-CNN model. The Faster R-CNN model is composed of two modules. The first module is responsible for generating RoIs with a CNN. The second module is a classification CNN. At its core, Faster R-CNN is a Fast R-CNN model with a region proposal network (RPN) [Fig. **??**]. In other words, the RPN generates RoIs, then Fast R-CNN takes these generated RoIs as input and generates bounding box locations and class scores. Through testing with different datasets like PASCAL VOC and COCO, Fast R-CNN with RPN as a region proposal method consistently achieved

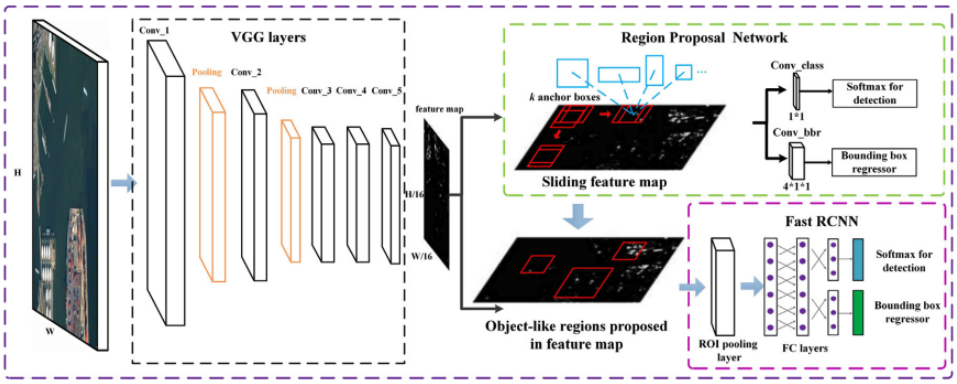


Figure 4.9: Faster R-CNN overall architecture **[faster_rcnn_architecture_fig]**

higher accuracy scores compared to when generating RoIs with the selective search algorithm. As an example, when testing the Fast R-CNN based on VGG-16 on the combined dataset of PASCAL VOC 2007 and 2012, the use of RPN to generate RoIs resulted in a higher mAP score of 73.2% compared to the Selective Search algorithm, which achieved an mAP score of 70% **[faster_rcnn_2015]**.

4.3.1 Region Proposal Network (RPN)

The Region Proposal Network (RPN) is a fully connected convolutional network engineered to propose regions of interest (RoIs) regardless of object scales and aspect ratios. RPN accepts images of any size as input and outputs several bounding boxes, each with a score representing the possibility of an object residing in the box under consideration. The main advantage of RPN over Selective Search is, while Selective Search is implemented in CPU, RPN is implemented in GPU. Utilizing the computation power of GPU instead of CPU help boost the performance of region proposal tremendously. Moreover, since models like Fast R-CNN also use GPU and convolutional layers to generate detection, RPN can share computation with Fast R-CNN through a multi-task loss. Consequently, reduce the computational power needed for generating RoIs to almost none when added on top of the computation already needed by the object detection module. As a comparison, RPN is able to generate RoIs in 0.01 seconds as opposed to 2 seconds with Selective Search while achieving a higher accuracy score overall **[faster_rcnn_2015]**. Furthermore, unlike Selective Search, which is a fixed algorithm and not trainable, RPN is a learnable neural network. Thus the performance will increase when more data is fed and/or a deeper network is being used.

The RPN model can be thought of as a two-stage process. In the first stage, RPN applies a CNN over the inputted image and generates a feature map for the entire image. The same operation is taken in the first stage of Fast R-CNN. Thus, Faster R-CNN can have a CNN to generate the image

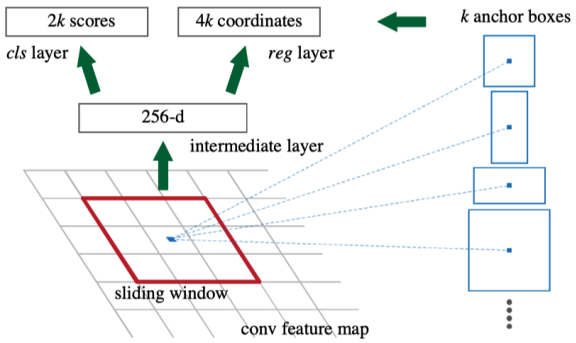


Figure 4.10: Region Proposal Network (RPN) [**faster_rcnn_2015**]

feature map, and then the image feature map proceeds as input for the second stage of RPN and Fast R-CNN. Thereby sharing computation between the two modules of Faster R-CNN. In the second stage, RPN slides a 3×3 convolutional layer, then branched to two siblings 1×1 layers. The first branch is a bounding box regressor (named *reg*) that generates a set of four values representing the coordinate of the bounding box. The second branch is a classifier (named *cls*) that produces a score for the probability that the object is in the window. At each sliding window location, RPN creates an anchor at the center of the window. Then, with 3 different scaling factors and 3 different aspect ratios, RPN generates 9 new proposals with the anchor at the center [Fig. **??**]. Since RPN behavior is based on the sliding window technique, it exhausted all possible locations, and with the use of 9 different sizes and aspect ratios, RPN gains the translation invariant property. That is, RPN is guaranteed to be able to generate the object's location even if some translation operation is performed on the object.

Similar to Fast R-CNN, the RPN model is trained end-to-end with a multi-task loss. The multi-task loss is derived based on the classification loss and regression loss across anchor points from the *cls* and *reg* branches, respectively. The classification loss evaluates the accuracy of classifying objects versus non-objects, while the regression loss assesses the accuracy of predicting bounding box coordinates. If RPN samples all anchor points, then the model will favor negative false as the majority of RPN's generated predicted bounding boxes does not overlap with any truth box. To resolve this problem, RPN employs a sample-dropping technique, then randomly selects 256 samples from the remaining samples to help balance positive and negative samples [**faster_rcnn_2015**]. For sample dropping, RPN marks all predicted boxes with over 70% overlap with any truth box as positive and those with less than 30% overlap as negative. Predicted boxes that are neither marked as positive nor negative are dropped. Then, RPN applies non-maximum suppression (NMS) to reduce the remaining samples, since the generation of 9 proposals at each anchor location may result in highly overlapping boxes. During NMS, any two proposed bounding boxes that overlap by more

than 70% with each other are compared, and the one with a higher *cls* score is kept while the other is dropped. With that being said, the RPN training process consists of three steps. In step one, the pre-trained VGG-16 model is used to initialize the shared CNN, and weight values are randomly assigned to the RPN's unique layers. Next, RPN is trained with the stochastic gradient descent (SGD) method, using mini-batches of 256 predicted bounding boxes from the remaining proposals after sample-dropping described above. In the last step, The model uses the previously described multi-task loss during each iteration to perform backpropagation.

4.3.2 Training

Although it is possible to train the RPN and Fast R-CNN components independently in Faster R-CNN as described previously. However, because the network learns significantly differently for object proposal and classification tasks, optimizing the shared convolutional layers of one task may not fully optimize them for the other task. To address this issue, Faster R-CNN employs a four-step alternative training procedure. Faster R-CNN trains an RPN model separately in the first phase, as described previously. Faster R-CNN trained a separate Fast R-CNN network in the second phase, which uses the RoIs generated by RPN in the first step as input for the classification task. Once the second phase is completed, Faster R-CNN trains the unique layer to RPN using the trained Fast R-CNN. This means that all Fast R-CNN layers are being fixed during this step, while the 3×3 convolutional layer and two siblings 1×1 layers unique to RPN are being trained. In the last phase, Faster R-CNN fixed the shared convolutional layers and layers specific to the RPN, so training occurs on Fast R-CNN's unique layers. This 4-step alternative training process is carried through each minibatch of SGD. The author of Faster R-CNN mentions two other shared schemes, approximate joint training and non-approximate joint training, in addition to the alternative training scheme [**faster_rcnn_2015**]. Unfortunately, both methods are impeded by issues beyond this study's scope; therefore, we will not go into depth.

By combining the VGG16 and RPN with the Fast R-CNN model and training with the union training set of PASCAL VOC 2007 + 2012, Faster R-CNN, using the top 300 proposals ranked by RPN's *cls* branch, achieved an mAP score of 73.2% when tested on the PASCAL VOC 2007 test set. This is 3.2% higher than when replacing RPN with Selective Search, which generated 2000 proposals for Fast-RCNN. Moreover, when using only the top-ranked 100 proposals, Faster R-CNN still achieved an mAP score of 55.1%, indicating that RPN's top-ranked proposals are highly accurate and that the *cls* branch is important. Furthermore, Faster R-CNN can process 5 images per second,

which is 10 times faster than the processing rate achieved by Fast R-CNN with Selective Search, which is 0.5 images per second [**faster_rcnn_2015**].

Faster R-CNN has been a significant breakthrough in the field of object detection. It achieved a much higher mAP score while maintaining a near real-time processing rate compared to its predecessor, Fast R-CNN. However, it is important to note that Faster R-CNN is limited to performing object detection and does not support instance segmentation. Instance segmentation involves not only detecting objects in an image but also segmenting each object at a pixel level. Additionally, Faster R-CNN still suffers from the quantization issue caused by the RoI pooling layer, as described in the previous section. In the next section, we will discuss Mask R-CNN, which adds instance segmentation capabilities to Fast R-CNN and resolves the quantization issue.

4.4 Mask R-CNN

In 2017, the Mask R-CNN model was introduced, which enhanced the Faster R-CNN model and enabled instance segmentation by adding a new branch called the mask branch. The mask branch predicts a binary mask for the object in each proposed RoI on a pixel-by-pixel basis. The mask branch works in parallel with the *reg* and *cls* branches of the Fast R-CNN model, which were described in Sec. **??**. In addition to the mask branch, Mask R-CNN also introduces the RoIAlign layer, which projects proposed RoIs onto the feature map and downsamples them to the input size requirement of the next layer. The RoIAlign layer replaces the RoIPool layer to achieve the same functionality without the quantization issue.

The Mask R-CNN architecture consists of three main components [Fig. **??**]. These components are a backbone network, a region proposal network (RPN), and the Fast R-CNN model with the addition of a mask branch. The backbone network is a convolutional neural network (CNN) that generates the image's feature map by performing feature extraction on the input image. The second component, RPN, takes the image's feature map as input and generates RoI proposals as described in Sec. **??**. These proposals are then fed into the Fast R-CNN model with a mask branch to predict the classification, the bounding box location, and a binary mask for the object that lies inside each RoI proposal.

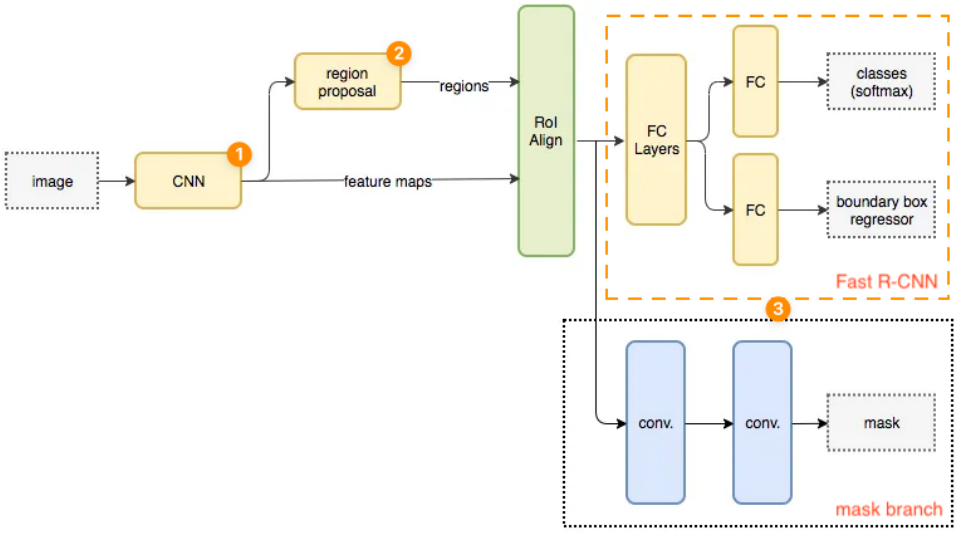


Figure 4.11: Mask R-CNN network flow **[rcnn_vari_flow_chart]**

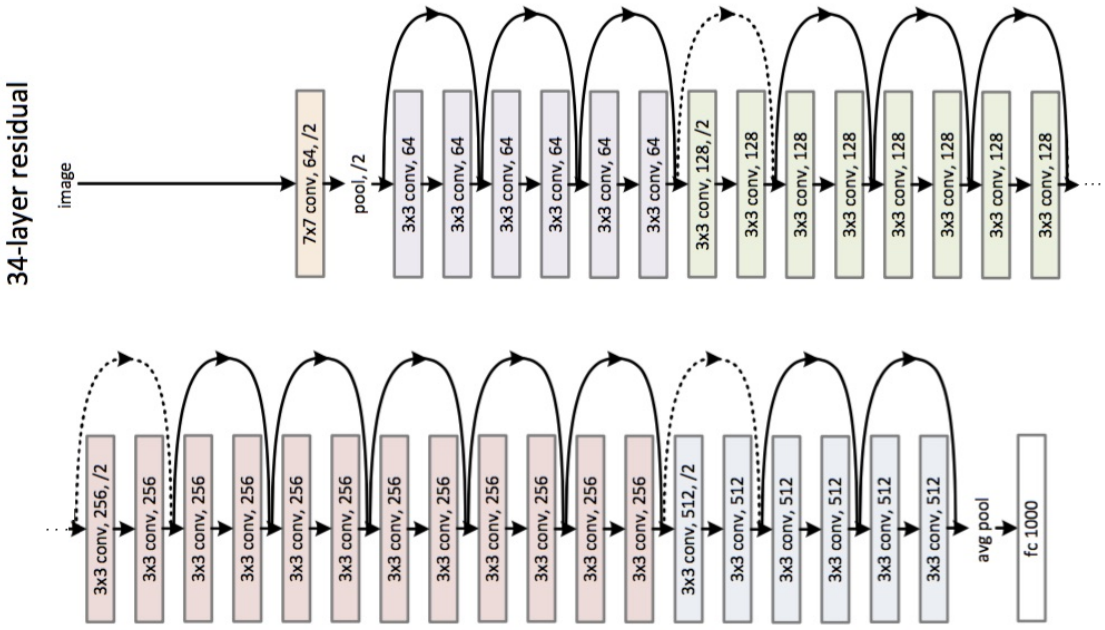


Figure 4.12: ResNet34 overall architecture **[resnet_2016]**

4.4.1 The Backbone Convolutional Neural Network

Similar to its predecessor, the backbone CNN used to generate feature map within the network model can be any deep CNN like AlexNet and VGG16. The backbone CNN used for Mask R-CNN in

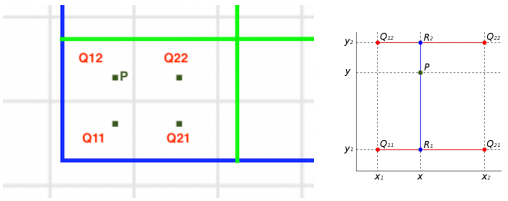
the original paper is the ResNet model. There are different implementations of ResNet with varying numbers of layers, including a 34-layer structure depicted in Fig. **??**. Like other CNN models, ResNet uses multiple convolutional layers, mostly 3*3, for feature extraction. ResNet then employs both max-pooling and average-pooling layers for downsampling. ResNet also utilizes a fully connected layer with 1000 neurons for classification. However, this fully connected layer will be dropped to fit with the inplementation of Fast R-CNN as described in Sec. **??**. As an important note, ResNet was the first network to introduce the use of identity shortcut connections, which further improve the runtime and address problems with high-depth networks. These connections skip over certain convolutional layers based on certain conditions. The detail of identity shortcut connection remains outside the scope of this paper but can be learned more at He et al., [**resnet_2016**]. The feature map generated by ResNet34 will then be passed to the RPN and the RoIAlign layer.

4.4.2 The RoIAlign Layer

The RPN generates a list of RoIs on the input feature map as described in Sec. **??**. It is important to note that the RPN does not necessarily ensure that the proposed RoIs will completely cover an exact natural number of feature map pixels, as it is only concerned with the existence of an object at any given location in the feature map. Following this, the feature map RoI proposal and the feature map produced by the backbone CNN are passed to the RoIAlign layer.

Recall the RoIPool layer from Fast R-CNN, the layer is responsible for mapping proposed RoI of varying to the feature map and downsampling them to a fixed pre-defined size by performing max-pool operations, described in Sec. **??**. One issue of RoIPool is it required to perform quantization because it need the RoI bins in the RoI feature grid to perfectly cover a natural number of pixel in order to perform max-pooling operation on the feature map pixel value, which lie in each RoI bin. During it operation, RoIPool layer perform quantization twice. The first quantization happen when RoIPool map proposed RoIs onto the feature map. The second quantization happen is when RoIPool perform downsampling by dividing the RoI feature grid into RoI bin. These quantization cause the extracted features to be offset from the input image features, and the model loss a small number of pixels data for each proposed RoI. While it is acceptable for object detection model as missing some pixels does not affect the general location and size of the object, pixel offset and pixel loss have a large negative affect in instance segmentation task, where the model have to identify all pixels belong to the object.

Similar to RoIPool, RoIAlign is a layer that aligns a proposed RoI onto a feature map and



(a) tobe added **(b)** tobe added

Figure 4.13: **(a)** A feature map RoI bin, represent by blue and green boundary. The dark green points inside the bin are regularly spaced points. Q_{ij} reresent the center of a feature map pixel, which has a coordinate and a feature pixel value. Q_{11}, Q_{21}, Q_{12}, and Q_{22} are the four nearest feature map pixel to point P.

downsamples the feature map RoI to a pre-defined size. However, RoIAlign does not have the quantization issue that RoIPool has because it uses max-pooling on the bilinear interpolation value of four regularly spaced points in each RoI bin rather than the feature map pixel value. This means that RoIAlign does not need to use complete pixels to perform the max-pool operation, eliminating the need for quantization. To achieve this, RoIAlign divides the feature map RoI into a grid of equal-sized RoI bins and calculates the location of the four regularly spaced points in each RoI bin. The number of RoI bins along the width and height of the RoI feature map is pre-defined by the input size requirement of the next layer in the network. With the calculated RoI bins, the four regularly spaced points are located at $(\frac{W}{3}, \frac{H}{3})$, $(\frac{2\times W}{3}, \frac{H}{3})$, $(\frac{W}{3}, \frac{2\times H}{3})$, and $(\frac{2\times W}{3}, \frac{2\times H}{3})$ in each RoI bin, where $W \times H$ be the size of the RoI bin [Fig. **??**]. Bilinear interpolation is then used to calculate the value of each point based on the four nearest feature map pixels. Since our regularly spaced points are in a grid of square pixels, thus our four nearest feature map pixels will create four pairs that either have the same x coordinate or the same y coordinate. Let the four nearest feature map pixels to a regularly spaced point P at (x, y) are Q_{11}, Q_{21}, Q_{12}, Q_{22} at (x_1, y_1), (x_2, y_1), (x_1, y_2), (x_2, y_2), respectively, as shown in Figure **??**. Then the bilinear interpolation value of point P can be calculated based on linear interpolation over $x-$coordinate and then $y-$coordinate as follow:

Let linear interpolation over $x-$coordinate be:

$$f(R_1) = f(x, y_1) = \left(\frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21})\right)$$
$$f(R_2) = f(x, y_2) = \left(\frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22})\right)$$

Then the bilinear interpolation of point P can be computed by performing linear

interpolation over $y-$coordinate of the linear interpolation result over $x-$coordinate.

$$
\begin{aligned}
f(P) &= \frac{y_2 - y}{y_2 - y_1} f(R_1) + \frac{y - y_1}{y_2 - y_1} f(R_2) \\
&= \frac{y_2 - y}{y_2 - y_1} f(x, y_1) + \frac{y - y_1}{y_2 - y_1} f(x, y_2) \\
&= \frac{y_2 - y}{y_2 - y_1} \left(\frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}) \right) \\
&\quad + \frac{y - y_1}{y_2 - y_1} \left(\frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}) \right)
\end{aligned} \tag{4.3}
$$

Since bilinear interpolation technique is based on the value of the four nearest pixel and weight them accordingly based on their location relative to point P, it give RoIAlign a smooth way to estimate the value at point P without shifting. Similarlly, the bilinear interpolation value for the other three regularly spaced points in each RoI bin are calculated. With the value of the four regularly spaced points, RoIAlign perform a max-pool operation on these points to choose a value represent the RoI bin. RoIAlign repeat this process for all RoI bins in the feature map RoI, thus result in a fixed predefine output matrix.

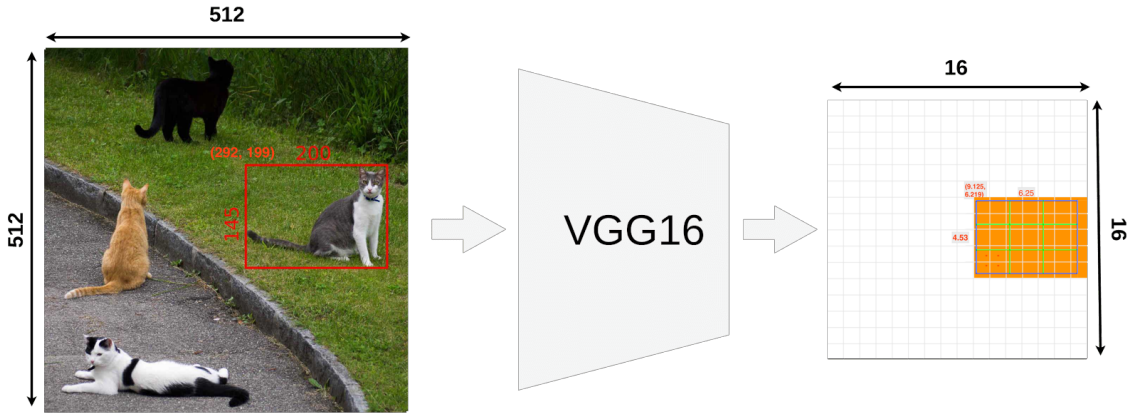


Figure 4.14: Mapping the proposed RoI onto the feature map using RoIAlign. The RoI (red rectangle) on the image space (left) aligns with the proposed RoI (blue rectangle) on the feature map space (right). The rectangles bounded by green or blue border inside the feature map RoI are RoI bins. The four red dots inside each RoI bins represent the four regularly spaced point for max pool operation. [**roi_pooling_problem**]

Reconsider the example mentioned in Sec. **??**, we process an image of size 512×512 image with Mask R-CNN without the mask branch and with VGG16 as the backbone CNN. This setup is the same setup as mentioned in the Fast R-CNN model except for replacing the RoIPool layer with the RoIAlign layer and using RPN to generate RoI proposals. By using RoIAlign, the proposed RoI location is the same between the input image and the image feature map [Fig. **??**]. Furthermore,

when downsampling the feature map RoI, RoIAlign does not lose information like RoIPool. Instead, RoIAlign uses bilinear interpolation on each point inside each RoI bin and samples from the surrounding feature map pixels. While both layer output a 3×3 matrix for each proposed RoI, in our example RoI proposal, RoIPool only sample 6×3 feature map pixels, 1.67 times smaller compared to 6×5 feature map sampled by RoIAlign [Fig. **??**, Fig. **??**, and Fig. **??**]. This allows the RoIAlign layer's output matrix to convey more information and helps maintain the required input size for the next layer.

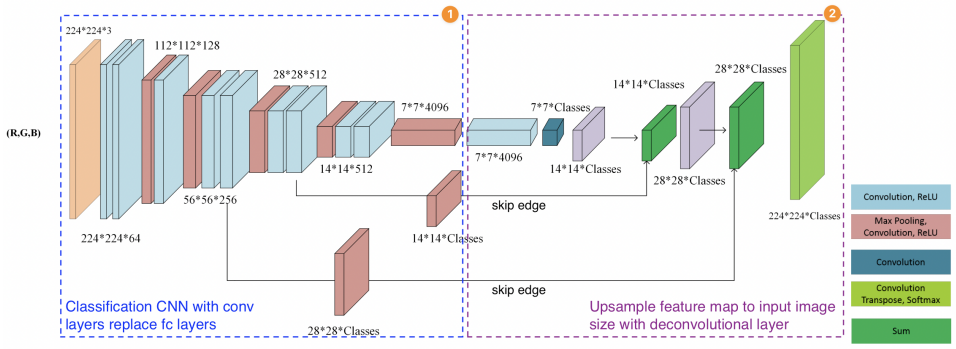


Figure 4.15: Fully Convolutional Network (FCN) with stride 8 [**fcn_archite_2018**]

4.4.3 The Mask Branch

Additional to the RoIAlign layer, Mask R-CNN introduce mask branch running parallel with the bounding box regressor branch and classification branch as included in Fast R-CNN model. The mask branch is based on the fully convolution neural network, which first proposed for the semantic segmantation task. We will use FCN as acronym for the fully convolution network, oppose to fully connected network, for the remaining of this section.

The FCN model is based on the classification CNN model, like AlexNet and VGG16, with the ability to predict classification for each pixel in the image. The FCN overall architecture consists of two stages [Fig. **??**]. The first stage is known as an encoder. The encoder responsible for dowsampling the input image and generating a feature map along with classification for each feature map pixel. Similar to the traditional classification CNN, the encoder can generate a feature map with a set of convolutional layers. In a classification CNN, once the feature map is generated, a fully connected layer will be used to classify the object in the feature map, which introduce two problem to the network. The first problem is the network will lose spatial location of all feature map pixels. This is because to convert feature map to a fully connected layer, the feature map must be flattern to

1-dimensional matrix of size $n_1 \times 1$ where n_1 is the number of neurons in the first fully connected layer. Then the first layer's neurons connect and activate second layer's neurons by performming a matrix multiplication with the transposed weight matrix of size $n_1 \times n_2$, where n_2 is the number of neurons in the second fully connected layer. That is $[n_1 \times n_2]^T[n_1 \times 1] = [n_2 \times 1]$, thus correctly result in the second fully connected layer with dimension of $n_2 \times 1$. However, due to fully connected layer matrix multiplication, these layers are required to have a fixed size. Thus result in the second problem of the fully connected layer is required the network to have a fix input size. The issue of fully connected layers is not desired in the encoder network as it must maintain the spatial location of all feature map pixels, so that the decoder can upsampling the subsampled (encoded) feature map. Recall equation **??**, which use to calculate the feature size, after apply a convolution layer. Assumming the kernel size is equal to the layer's input feature map and we do not use zero-pading, then the network's layer will result in a feature map of size 1×1. By applying n_1 kernels where each kernel have the same size as the layer's input feature map, the network will achieve the same n_1 neurons as when use fully connected layer, while remove the shortcomming of using the fully connected layer. Therefore, the encoder part of FCN able to perform feature map generation and feature map pixel classification without the loss of pixels' spatial location by replacing the fully connected layers with convolution layers in a traditional classification CNN.

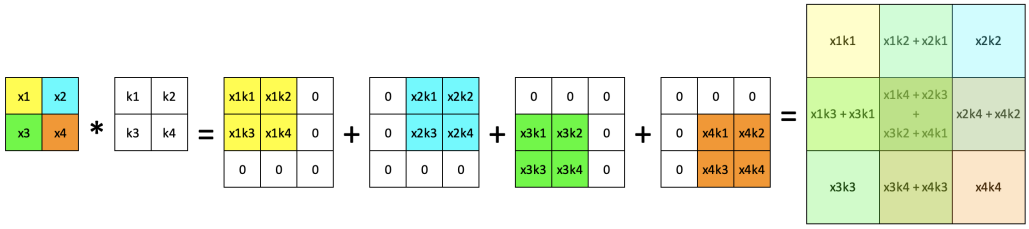


Figure 4.16: tobe added

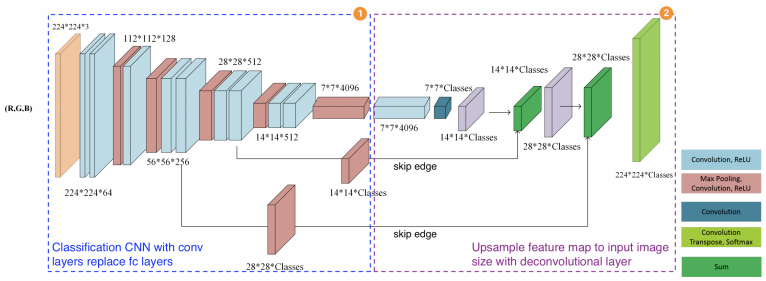
With the spatial location and class label of all pixels in the feature map generated by the encoder, the decoder, second stage of FCN model, can upsampling these classified feature map pixels to the input image space. By upsampling the classified pixel, the decoder will be able to project objects' pixel-wise mask from feature map space to the input image space. The decoder perform upsampling operation by using the learnable transposed convolution layer, also known with other names like fractionally strided convolution, backward strided convolution, or upconvolution [**transposed_convolution_layer_2016**]. When upsampling an input feature matrix, transpose convolution layer first apply a learnable kernel to each element of the input matrix by performing a

scarlar matrix multiplication. The learnable kernel have it's weight randomly initialize or with an upsampling technique like bilinear interpolation. The location of the multiplicated matrix on the output matrix is determined by the kernel size and stride. That is the result of our scarlar matrix multiplication, between input matrix element and kernel matrix, will slide on the output matrix with the stride of $\left\lfloor \frac{1}{f} \right\rfloor$, where f is the scale factor between the imput image and out put image. Note that if our kernel size is larger than kernel stride, then the multiplicated matrix is overlaping with one another on the output matrix, and thus we concatenate the overlaped cell of these multiplicated matrix together. An example of overlaping transpose convolution layer is shown in Figure **??**, where kernel size is 2 and stride is 1. Additionally, transposed convolution is responsible for negate the size change of the convolution operation, thus we can invert the Equation **??** to see the relationship between kernel size, kernel stride, and zero padding interact with one another as follow:

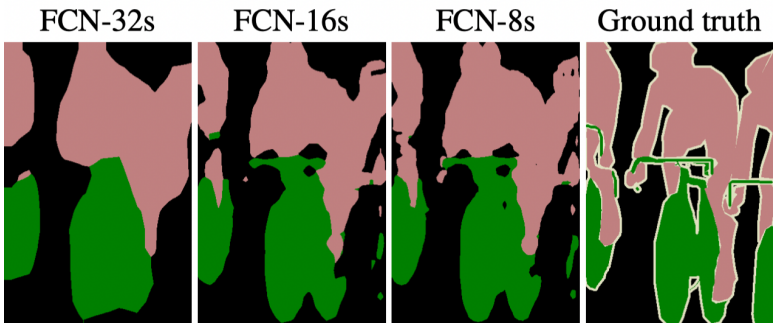
$$\text{output size} = \text{input size} - 1 \times \text{kernel stride} - 2 \times \text{zero-padding size} + \text{kernel size}$$

Let's consider the example of upsampling a 2×2 feature map to the image size of 32×32, which corresponds to a scale factor of 16. Without using any zero-padding, the kernel size must be $32 - (2 - 1) \times 16 = 16$ for both the width and height of the kernel. The FCN model that performs a scale factor of 16 after downsampling for classification is referred to as FCN-16s. In the original FCN paper, the authors proposed three variations of the FCN model: FCN-32s, FCN-16s, and FCN-8s. The architecture of the FCN-8s model is illustrated in Figure **??**.

The skip connection is a key element in improving the accuracy of segmentation masks produced by Fully Convolutional Networks (FCN). As seen in Figure **??**, while FCN-32s can provide a rough idea of object locations, it has low pixel accuracy. To address this issue, FCN-16s and FCN-8s use skip connections and reduce the scale factor, as shown in Figure **??**. The skip connection allows the FCN to retain the original feature map before downsampling, which is then concatenated with a classified upsampled feature map of the same size. This process reduces the scaling factor needed for upsampling and enables the original feature map to contribute in predicting missing pixels. With the additional information from the skip connection, the FCN can project the classified pixels more accurately onto the denser space, resulting in improved segmentation masks.



(a) The FCN-8s architecture [**fcn_archite_2018**]



(b) The output mask for different version of FCN model

Figure 4.17: tobe added

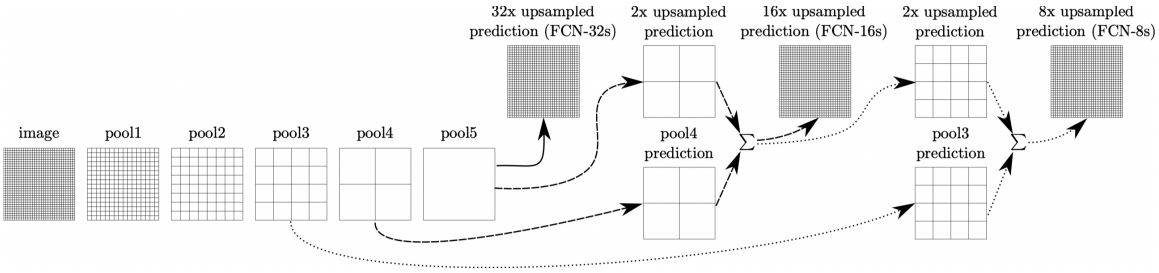


Figure 4.18: FCN skip connection

CHAPTER 5

YOLO Variation

In addition to the Region-based Convolutional Neural Network (R-CNN) family, the other family of model we will discuss in this section is You Only Look Once (YOLO). Similar to R-CNN family, YOLO family also designed to perform object detection and can be expand to other task like instance segmentation or semantic segmentation. However, unlike R-CNN family, the model in YOLO family frame the object detection task as a single stage regression problem instead of double stage regression problem cite yolov3.

The models in R-CNN family is consider as double stage method because the model first try to generate region of interest (RoI), then classify each RoI using two different algorithm or network. That is, while RoI generation can be done with a greedy algorithm (like Selective Search) or a fully connected convolutional network (like RPN), the model use another CNN for feature extraction (like AlexNet, VGG16, and ResNet) and adapt it to perform classfication for each generated RoI [**Girshick_R_CNN_2013**, **fast_rcnn_og**, **faster_rcnn_2015**]. For this reason, the object detection task is seperate into two tasks: localization task and classfication task, where each task is completed by a different network.

On the other hand, the models in YOLO family is utilized a single convolution neural network to simultaneously predict both bounding box and the classfication label for each object in the input image [**yolov1_2016**]. Therefore YOLO model family is designed to only evaluate an image once with a single CNN for object detection task. Hence the name You Only Look Once. In this chapter we will disscuss different version of YOLO model.

5.1 YOLOv1

The first YOLO model was proposed in the "You Only Look Once: Unified, Real-Time Object Detection" paper in 2015 [**yolov1_2016**]. Since there will be improvement version of the model, we denote this first model is YOLOv1, which is widely accepted in the computer vision research community [**understand_cnn_vs_yolo**]. The YOLOv1 is designed to be a realtime object detection model. The YOLOv1 model is a three steps process, as shown in Firgure **??**. In the first step, the model take an image of any size as input and resize the image to the fix size of 448×448. In the second step, the model predict mutiple bounding box, each with a objectiveness confidence score, and multiple classification for each detected object. Since the model predict multiple bounding box and classfication for each detected object, a non-maximum suppression (NMS) algorithm, as described in Subsection **??**, is applied to assign one bounding box and one classification label per predicted object in the last step.

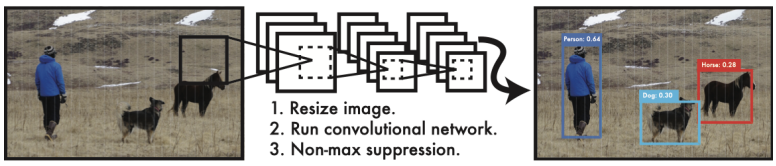


Figure 5.1: YOLOv1 object detection process [**yolov1_2016**]

In the original paper, the authors claim that YOLOv1 model have three main benifit [**yolov1_2016**]. First, YOLOv1 is extremely fast, processing 45 images per second compare to 7 images per second achieved by the Faster R-CNN model with VGG16 backbone. However, this is achieved at the cost of 9.8% reduction in mean average precision (mAP) score, i.e., 63.4% and 73.2% for mAP score of YOLOv1 and Faster R-CNN, respectively. The second benifit is YOLOv1 learn a general representation of object, thus it tend to perform better compare to R-CNN based model when predicting for other domain like artwork. The third benifit it it see the entire image during bounding box regression and classification compare to R-CNN classifier and bounding box regressor only see the RoI. This change allow YOLOv1 to encodes contextual information about classes and thus reduce the number of false positive.

5.1.1 Network Architecture

The YOLOv1 model is a convolutional neural network (CNN) based on the GoogLeNet model for image classification cite GoogLeNet. The YOLOv1 model consist of 24 convolutional layer and end

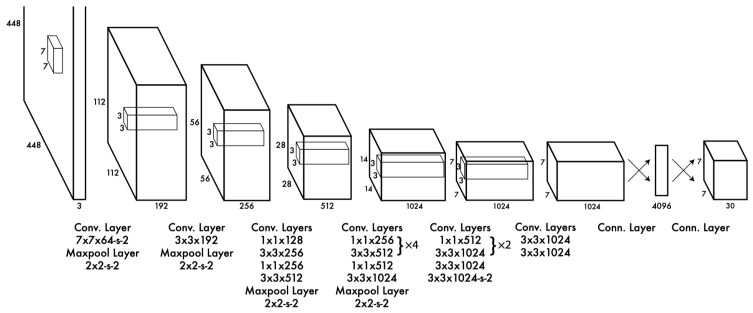


Figure 5.2: YOLOv1 architecture **[yolov1_2016]**

with 2 fully connected layer. The overall architecture of YOLOv1 netword is shown in Figure **??**. The author replace the inception layers in GoogLeNet with 1×1 reduction layer and 3×3 convolutional layer pair further explain the inception modules and reduction layer. All the layers in the yolov1 network, with the exception of the final layer, utilize the leaky ReLU activation function cite leaky ReLU, described as:

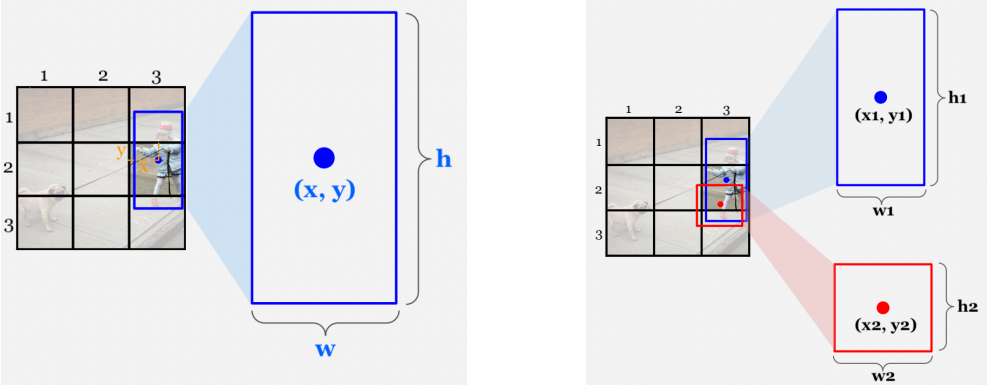
$$\phi(x) = xifx > 0, 0.1xotherwise$$

As we can see in Figure **??**, the last convolutional layers in the network produce a feature map of size $7 \times 7 \times 1024$. This feature map is then processed by two fully connected layers. The last fully connected layer is responsible for predicts both the bounding box and the class label probability **[yolov1_2016]**. This layer use a linear activation function. The classfication process in the last fully connected layer is simmilar to other CNN where the layer is randomly initialize and can be optimized through training. On the other hand, YOLOv1 proposed a new bounding box regression method that able to predict all bounding box of all objects present in the image at once, instead of processing each RoI individually one-by-one like the regressor implemented in Faster R-CNN. We will discuss this bouding box generation process in the next subsection.

5.1.2 Bounding Box Generation

To predict bounding box, the YOLOv1 model divide the image into an $S \times S$ grid of equal cells. Each grid cell is then predict B bounding boxes and C probabilities for the C supported classes **[yolov1_2016]**. The S and B values are hyperparameter and can be fine tune throught experiment. The C value is the number of classes in the training dataset. In other words, $C = 20$ if the training dataset is PASCAL VOC **[pascal_voc_2015]** and $C = 80$ if the training dataset is COCO dataset **[coco_2014]**.

Each bounding box is represented by 5 values: coordinate x, coordinate y, width w, height h, and a confidence score. The (x, y) coordinates is the center of the bounding box relative to a grid cell. The width w and height h is the dimension of bouding box in 2D space. The value of w and h are normalized with respect to the input image width and height, thus they are bounded by [0, 1]. The confidence score denote the model confidence in saying there is an object present in this cell, i.e., the objectiveness probability.



(a) tobe added **(b)** tobe added

Figure 5.3: tobe added

As an example, consider processing an image with $S = 3$, $B = 1$, and clasifying between two class human and dog ($C = 2$), as demonstrated in Firgure **??**. Since $B = 1$, which means we only predict one bounding box per cell, then the cell_{32} will return a tensor represent the predicted bounding box as:

$$ceil's\ output = \begin{bmatrix} x & y & w & h & conf & p_{human} & p_{dog} \end{bmatrix}$$

where $conf$ is the confidence score, p_{human} and p_{dog} are probability that the object belong to the human and dog class, respectively. Noted that when we only predicting 1 bounding box per ceil, the YOLOv1 model predict $(4 + 1 + 2)$ values for each ceil, where 4 value describe the bounding box location, 1 for confidence score, and 2 probality values with one for each class. Therefore we say that when $B = 1$, the model predict $(4 + 1 + C)$ for each ceil.

Now, we consider the example reside in Figure **??**, which is the same setup as previous example but with $B = 2$. In this second example, we predict two bounding boxes per cell, then the cell_{32} return the following tensor for the two bounding boxes:

$$ceil's\ output = \begin{bmatrix} x_1 & y_1 & w_1 & h_1 & conf_1 & x_2 & y_2 & w_2 & h_2 & conf_2 & p_{human} & p_{dog} \end{bmatrix}$$

The ceil output a $(4 + 1) * 2 + C$ elements tensor for when the model predict two bounding boxes per ceil. Therefore, we can generalize the ceil's prediction is encoded as $(4 + 1) * B + C$ tensor.

The computed prediction for each ceil in the grid is stacked side by side create the depth for the image. That is we divides a 2-dimentional image into a grid of $S \times S$ ceil, we predict a $(4 + 1) * B + C$ tensor for each ceil, these prediction create the third dimention of the image. Therefore the prediction for the image is encoded as $S \times S \times [(4 + 1) * B + C]$ tensor.

The YOLOv1 architecture showned in Figure **??** is for predicting 20 class in PASCAL VOC with $S = 7$ and $B = 2$, thus the model predicting a $7 \times 7 \times 30$ tensor which encoded multiple bouding boxes and classfication for each ground-truth object.

5.1.3 Training

The first 20 convolutional layers of YOLOv1 is pretrained with the input size of 224×224 for classification task on the ImageNet2012 dataset [**ImageNet_dataset**]. Then four new convolutional layers and 2 fully connected layer are added. These new layers are randomly initialize. Additionally, the input size is increase to 448×448 for object detection task [**yolov1_2016**]. With the initialized network, the model generate multiple bounding boxes and classification as described previously. While the YOLOv1 model apply NMS to choose which predicted bounding box to keep at inference time, the author used a different scheme to choose which predicted bounding box to contribute to the loss function. The scheme is choosing the predictor with predicted box that has the highest IoU with a ground truth box. This lead to each predictor have different specilization, which mean each predictor is better at predicting certain size, aspect ratio, or object's class [**yolov1_2016**].

The YOLOv1 model is trained to optimize for the sum-square error which encode both the bounding box coordinate loss and the classification loss [**yolov1_2016**]. While sum-square error allow eazier optimization, it have some shortcomming and not ideal if the model need to optimize for average precision. The first critical shortcomming is the loss weight localization error and classification error equally [**yolov1_2016**]. In an image, since the majority of the cells does not contain any object, which mean confidence score for these cell are 0, thus cause model always have a poor performance for classification task. This also cause bouding box error to have little affet on the total error. Thus a scalar term is added to the loss to weight the bouding box error higher than the classification error [**yolov1_2016**]. The second critical shortcomming is sum-squared error weight offset in large bounding box and small bounding box equally [**yolov1_2016**]. This is not ideal as offset by certain pixels have a larger affect on the smaller bouding box than the larger bounding box,

due to total number of pixels in large bounding box is larger than the smaller bounding box. To patially resolve this, the YOLOv1 model perform the error calculation on the squareroot of bounding box width and height instead of the width and height directly.

5.2 YOLO9000 (YOLOv2)

YOLO9000, also known as YOLOv2, is an improvement of the YOLOv1 model [**yolo9000_2017**]. The real time object detection model YOLOv2 is first introduced in 2016. The name YOLO9000 is stem from the fact that the model is capable of detecting over 9,000 object categories in real-time, which is significantly more than the original YOLO model. This is achieved by finetunning the model with the WordNet language database, which also is the database that ImageNet labels set pulls from.

5.2.1 Accuracy Improvement

The YOLOv2 model propose five changes that improve the accuracy of YOLOv1. The first change is the addition of **batch normalization** on convolutional layer, which improve YOLOv1 mAP score by more than 2%. Batch normalization remove the need of dropout technique [**dropout_2014**], which drop certain activated neuron to avoid overfiting problem [**szeliski_cv_book**]. The second change is utilize **higher resolution classifier**. While YOLOv1 only train on 224×224 images for classification task, YOLOv2 further train the classifier with 448×448 images. The higher resolution classifier increase the YOLOv1 mAP score by 4%. The third change is the use of **passthrough layer**. Since convolutional layer decrease the image spatial dimension gradually, thus it become more diffcult to detect smaller object in the image as more convolutional layers are used. For this reason, the passthrough layer help bring the image detail from the higher spatial dimension to the lower spatial dimension map, simmilar to skip connection in ResNet [**resnet_2016**].

Since each divided cell in YOLOv1 only predict exactly 2 bounding boxes, 1 confidence score, and 1 classification label (the highest probability class), thus the model has poor performance in detecting object appear near together, especially small object. To address this problem, YOLOv2 model propose the fourth and fifth change.

The fourth change is utilizing **anchor boxes** at each cell instead of randomly initialize bounding box like in YOLOv1. The size and aspect ratio of the anchor box is heavily depend on the domain that the model will be applied to. For example, when apply the model to traffic detection task, then the main shape and size the model need to look for are pedestrian and different vehicle. Thus

starting the anchor box at these shape and size improve the runtime for both training and inference. For this reason, a k-means clustering algorithm is used to find the top-k common dimension in the training set [**yolo9000_2017**]. The anchor boxes is then used to predict the bounding box. The YOLOv2 model predict 5 parameters for each bounding box $[t_x, t_y, t_w, t_h, and t_o]$, where t_x, t_y and t_w, t_h are center and dimention of the bounding box in relation to the cell and the image dimention, respectively. These 5 parameters are the same as the prediction make by YOLOv1. Given that the cell is (c_x, x_y) offset from the top-left conner of the image, and the anchor box have the dimension of p_w, p_h, then the predicted bounding box has the center at (b_x, b_y) with dimension of b_w, b_h, and can be computed with constraint by sigmoid function, as shown in Firgure **??**.

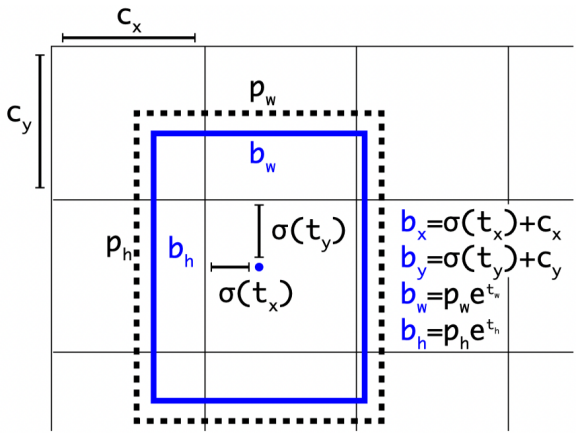


Figure 5.4: YOLOv2 bouding box (in blue) generation offset from the anchor box (in black) [**yolo9000_2017**]

Since the anchor boxes is used to predict the bounding box, thus remove the need of the fully connected layer. In addition to the removal of fully connected layer, YOLOv2 model also move the classification label from cell level to anchor box level. In other words, YOLOv2 model predict a classification label for each anchor box [**yolo9000_2017**]. Let the image be divided to an $S \times S$ cell grid, each cell generate B anchor boxes, and the dataset have C catagorise, then the image detection is encoded as a $S \times S \times B * (4 + 1 + C)$ tensor. The achor box scheme remove the assumption of one object per grid cell and improve the mAP score by 5% approximately.

The fifth change is **multi-scale training**. Since YOLOv2 remove the fully connected layer, thus it can process the image of any size. Therefore, instead of training with a fixed size, the model choose a new input image dimension every 10 batches. Additionally, since the model is downsample by 32 times during it process, to avoid quantization, the input image should be a multiple of 32: 320, 352, ..., 608. This trainning scheme force the model to be able to predict object at different resolution,

which train the network to predict object of different scale. In addition to multi-scale training, the model also perform data agmentation process including crop, rotattion, hue and saturation shift, and exposure shift to expand the training dataset and futher generalize the model.

5.2.2 Runtime Improvement

The YOLOv2 further simplify the archietecture used in YOLOv1. The YOLOv2 model use a new classification model, namely Darknet-19. The overall archietecture of Darknet-19 is shown in Firgure **??**. Compare to the GoogLeNet, Darknet-19 is smaller with 5.58 billion operations instead of 8.52 billion operations, while have higher top-5 accuracy in classification task after training [**yolo9000_2017**].Compare to YOLOv1's CNN archietecture, Darknet-19 only has 19 convolutional layers and 5 max-pooling layers, while YOLOv1's CNN consist of 24 convolutional layers, 4 max-pooling layers, and 2 fully connected layers. The Darknet-19 is first trained with the ImageNet 1000 for classification task. The Darknet-19 model is then adapted for object detection task by replacing the last convolutional layer with three 3×3 convolutional layer that output 1024 channels, folowed by 1×1 convolutional layer to convert $S \times S \times 1024$ to $S \times S \times B * (4 + 1 + C)$.

5.3 YOLOv3

The YOLOv3 model is the next incresemental improvement of YOLO family after YOLO9000. The YOLOv3 model is introduced in 2018 [**yolov3_2018**], which consist of four main minor changes to YOLOv2 archietecture.

The first change is replacing the softmax function with independent logistic for the classifier. While the softmax function require the sum of all class's probabilities to 1, independent logistic consider each class independently and the probability of any two class do not affect one another [**yolov3_2018**]. This improve the model's classifier from multi-class classifier to multi-class and multi-label classifier. That is, when the classifier is multi-class and multi-label, there are no competition between label. For example, if a pedestrian is a women and the supported label include both "pedestrian" and "women", then when use independent logistic classifier the probability for both label should be high for this object, while softmax classifier will only give high probability to one of the two labels.

While the format of the predicted bounding in YOLOv3 is the same as YOLOv2 (Figure **??**), the calculation of the confidence score and loss function are different compare to YOLOv2. This is the

Type	Filters	Size/Stride	Output
Convolutional	32	3×3	224×224
Maxpool		$2 \times 2/2$	112×112
Convolutional	64	3×3	112×112
Maxpool		$2 \times 2/2$	56×56
Convolutional	128	3×3	56×56
Convolutional	64	1×1	56×56
Convolutional	128	3×3	56×56
Maxpool		$2 \times 2/2$	28×28
Convolutional	256	3×3	28×28
Convolutional	128	1×1	28×28
Convolutional	256	3×3	28×28
Maxpool		$2 \times 2/2$	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Maxpool		$2 \times 2/2$	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	1000	1×1	7×7
Avgpool		Global	1000
Softmax			

Figure 5.5: Darknet-19 architecture [**yolo9000_2017**]

second change the YOLOv3 proposed compare to YOLOv2. The YOLOv3 predicts the bounding box's confidence (objectness) score is predicted using a logistic regression [**yolov3_2018**]. This is trained by setting the confidence score of the bounding box anchor to 1 if the anchor box has highest IoU score with the ground-truth box compare to other anchor. During training any anchor box that has IoU score more than 0.5 with a ground-truth box but the IoU score is not the highest among anchor boxes for this object, then that anchor box will not contribute to the loss function. If the model does not predict a bounding box anchor for a ground-truth object, the objectness loss will increase and contribute no affect to localization loss and classification loss. Simmilar to YOLOv1 and YOLOv2, YOLOv3 also optimize for sum-square error loss which combine the localization, classification, and objectness loss into one metric.

The third change is detection at 3 different scales per cell. At each cell, the model predict B bounding box for each feature map scale i.e., the current feature map and two upscaled feature map.

Assume we divide the image into $S \times S$ cells and the dataset is COCO 80 labels, then at each scale the model predict $S \times S \times [3_{bbox} * (4 + 1 + 80)]$, thus the model generate a $S \times N \times [3_{scale} * 3_{bbox} * (4 + 1 + 80)]$ [**yolov3_2018**]. The YOLOv3 model predict 9 anchor boxes per cell. This 9 anchor boxes is choosen using the k-means clustering algorithm and divided into each scale evenly. For example, the YOLOv3 model trained for COCO dataset will have the following 9 anchor boxes:

$$Scale - 1 :[(10 \times 13), (16 \times 30), (33 \times 23)]$$

$$Scale - 2 :[(30 \times 61), (62 \times 45), (59 \times 119)]$$

$$Scale - 3 :[(116 \times 90), (156 \times 198), (373 \times 326)]$$

The fourth change is the feature extractor. Inspired by Darknet-19 and ResNet, the author proposed a new CNN archietecture, namely Darknet-53. The overall architechture of Darknet-53 is shown in Firgure **??**.The network consist of 53 convolutional layers, 3×3 or 1×1 convolutional layers, and using shortcut connections [**yolov3_2018**]. The shortcut connections is introduced in ResNet model and is used to skip over certain convolutional layers based on certain conditions, which improve runtime and address problems with high-depth networks [**resnet_2016**]. The Darknet-53 is two times faster than the ResNet-152 model while perserve the model's accuracy. Other than the loss computation method, the Darknet-53 training is the same as YOLOv2 which include batch normalization, multi-scale training, and data augmentation.

5.4 YOLOv5

The YOLOv5 is the fifth entry of the YOLO family. The model is published in 2020 by Ultralytics teams [**yolov5_github**]. The name YOLOv5 is still controlversal to this date because it is considered as less innovative compare to the YOLOv4 model. While YOLOv4 has a significant change in structure and use some state-of-art algorithm like MISH activation function and GIOU(Generalized Intersection over Union) loss function, YOLOv5 is more focus on ease of use, model size control, and enhancing training data [**yolov5_review**]. Unfortunately, YOLOv5 model have never had a formal research paper detail explaining the implementation detail, but it has a well documented and supported API developed and maintain by the Ultralytics teams [**yolov5_github**].

At it core, YOLOv5 is YOLOv3 structure with flexible control of model size [**yolov5_review**]. This is reflected by the API where there are 5 versions of YOLOv5: nano, small, medium, large, and xlarge. Where nano is the smallest version with approximate 12.7 million parameters and xlarge is

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figure 5.6: Darknet-53 architecture [**yolov3_2018**]

the largest version with approximate 141.8 million parameters [**pytorch_yolov5**]. With the different in size, without a doubt, nano version is much faster than the xlarge version at 4.3ms compare to 22.4ms. However, nano version also have a smaller mAP score compare to the xlarge version at 61.9% compare to 72.0% on the COCO evaluation set at IoU threshold of 0.5.

CHAPTER 6

Experiments: Mask R-CNN and YOLOv5 Comparision

We perform a comparision between the Mask R-CNN and the YOLOv5 algorithm on the object detection task. We use 16,445 anotated images in the validation set of the nuImages dataset. In this chapter, we will provide an overview of the nuImages dataset and describe the metrics implementation we employed in our experiment program. Finally, we will discuss the performance of the two models.

6.1 nuImages Dataset

The nuImages dataset is a large dataset that is designed for computer vision tasks in autonomous driving applications. It was developed by the Motional team and is available for personal and educational use. The dataset comprises 93,000 images that were carefully selected from 500 driving logs. Each log provides data from the entire sensor suite of an autonomous vehicle, including six cameras, one LIDAR, five RADARs, GPS, and IMU. The images in the nuImages dataset were selected based on two criteria. The first 75% of the dataset consists of images that are challenging for object detection models to detect objects within them due to weather and lighting conditions, as well as rare classes like motorcycles and bicycles. These weather and lighting conditions are essential for autonomous driving applications as the vehicle needs to operate in rain, snow, storms, and during nighttime. The remaining 25% of the dataset was sampled uniformly to avoid bias towards a particular class or condition. Using these two schemes ensures that the nuImages dataset has a balanced distribution of classes, weather, and lighting conditions.

The 93,000 images in the data set are divided into three distinct subsets. Precisely, the training subset consists of 67,000 images, the validation subset has 16,000 images, and the test subset comprises 10,000 images. Notably, the training and validation subsets contain annotated images,

while the test subset does not. The dataset can be obtained at [**nuimages_dataset**]. To use the nuImages dataset for evaluating the object detection model, we only concern ourselves with the Metadata and Sample directories in the dataset. The Metadata directory provides information about all the images in each subset, including the mapping between the image to their class, mask, and bounding box location, if available. Meanwhile, the Samples directory contains traffic scene images extracted from each camera mounted on the vehicle.

In conjunction with the dataset, the Motional team also developed the nuscenes-devkit library to provide an API to simplify accessing these data. We build the *NuImgSample* class in *src/nuim_util.py* to represent each image in the dataset. Each *NuImgSample* object utilizes the NuImages object from the API to extract important information such as the original image path, mask, bounding box location, and classification label for each object in the image. During initialization, *NuImgSample* also maps the nuImages object's label to one of the supported labels or removes the object entirely if it is not supported. The supported labels are bicycle, bus, car, motorcycle, person, and truck. The mapping from nuImages catogorical labels to supported label is available in *src/label_mapping.py*. This mapping and filtering process is crucial because the nuImages labels (ground-truth label set), Mask R-CNN labels (predicted label set 1), and YOLOv5 (predicted label set 2) are all different. This process ensures that the ground-truth and predicted labels match if the model correctly detects the object.

After creating instances of the NuImageSample class with the extracted images from the dataset, the *get_truth_objs* function in *src/nuim_util.py* is used to process these instances and generate a list of TruthObject instances. Each TruthObject instance represents an object in the original image and contains essential information about the object such as its classification label, bounding box coordinates in the format (x_{min}, y_{min}, x_{max}, y_{max}), and the object's mask matrix. A TruthObject instance is shown as:

```
truth_object.TruthObject object at 0x285cfa763 contains:
    t_label: car
    t_bbox: [785, 478, 890, 563]
    t_mask: [[[0, 0, ..., 0], ..., [0, 0, ..., 0]]]
```

By using the TruthObject class to represent each object in the ground truth image, our code takes advantage of the benefits of object-oriented programming (OOP) design properties. This approach makes it easy to access the necessary information about each object and ensures that the code remains loosely coupled.

6.2 Prepare Models for Comparision

6.2.1 Mask R-CNN Preparation

For our comparison, we utilized the pre-trained Mask R-CNN model for the COCO challenge, which is implemented in the *torchvision* library [**pytorch_mrcnn**]. This implementation uses the ResNet50 CNN backbone, as discussed in Section **??**, and can be accessed through the *maskrcnn_resnet50_fpn* function provided by the *torchvision.models.detection* module. The function expects a list of tensors as input, where each tensor represents an image with dimensions of [number of image channels, image width, image height]. To convert an image to a tensor, we used the *read_image* helper function provided by the *torchvision.io* module, which takes the path to the image storage location as an input string. Since the model uses pre-trained weights for the COCO challenge, we must apply the preprocessing operations on the image tensor provided by *MaskRCNN_ResNet50_FPN_Weights.COCO_V1.transforms* function before passing it to the *maskrcnn_resnet50_fpn* function.

The *maskrcnn_resnet50_fpn* function employs the Mask R-CNN model to predict four lists – labels, confidence scores, bounding boxes, and masks – for each image tensor. The elements in these four lists correspond to each other location-wise and provide information about the detected objects. The list of confidence scores is sorted from highest to lowest, and the order of the other three lists changes accordingly. For instance, the first element in the scores, labels, bounding boxes, and masks list describes the detected object that the model is most confident about among all detected objects.

To eliminate the need to maintain location correspondence between four lists, the PredictObject class, defined in *src/predict_obj.py*, is used. Each class instance represents a predicted object within the image and contains the object's label, confidence score, bounding box, and mask, as demonstrated:

```
predict_object.PredictObject object at 0x299dee950 contains:
    p_label: car
    p_score: 0.959472981564043
    p_bbox: [789.85, 483.3, 879.51, 554.54]
    p_mask: [[[0, 0, ..., 0], ..., [0, 0, ..., 0]]]
```

Similar to the TruthObject, the PredictObject follows the OOP principle, enabling easy access to information without affecting program efficiency. Since the instances of this class are passed across functions as references rather than actual copies, their usage does not impact program efficiency, as documented in [**python3_docs**]. Furthermore, since the model is pre-trained for the COCO challenge, which consists of 80 categories, we remove all objects with labels that are not in the

supported labels set before creating the list of PredictObject instances. This ensures that the label comparison between the ground-truth and predicted labels is valid, and there is no need for label mapping since all the supported labels exist in the COCO challenge's label set.

6.2.2 YOLOv5 Preparation

We utilized the pre-trained YOLOv5 model developed by Ultralytics teams as the second model for our comparison. YOLOv5 has five versions, ranging from most minor to most extensive in the number of neuron nodes and depth layers. For our comparison, we used the most extensive version, known as YOLOv5x. The YOLOv5 model is available through PyTorch and can be loaded into our program using the *hub.load* function provided in the *torch* library. Similar to the Mask R-CNN model, we loaded the pre-trained COCO weight for the YOLOv5 model. However, unlike Mask R-CNN, once initiated in our program, the YOLOv5 model takes the path of an image and produces a Pandas data frame. Each row in the data frame represents a detected object in the image. We convert this data frame into a list of PredictObject instances, similar to the process described in Subsection **??**. This conversion ensures that other functions in our program can process the YOLOv5 model output in the same manner as the Mask R-CNN model output. One key difference is that YOLOv5 does not generate a mask for each object as is an object detection model; therefore, the mask field for each PredictObject instance defaults to a None value.

6.3 Metric Implementation Detail

Recall from Section **??**, to evaluate the performance of an object detection model, we need to perform the following five steps:

1. Calculate the Intersection over Union (IoU) between the predicted and ground-truth bounding boxes to determine the detection's location correctness.

2. Create a confusion matrix for each categorical label at a particular IoU and confidence threshold by comparing the predicted labels against the ground-truth labels at that configuration.

3. Calculate the precision and recall metrics for each confusion matrix. These two metrics indicate the model's reliability and sensitivity in classifying objects of this category at each threshold pair. Additionally, these two metrics can be used to plot the precision-recall curve, which signifies the tradeoff between the two metrics across confidence thresholds. An object detection

model is considered high performance at a confidence threshold if the precision remains high as recall increases. The F-score is then used to find the combination of precision and recall at which the model achieves its highest reliability and sensitivity performance across all confidence thresholds.

4. Calculate the average precision (AP) to determine the model's overall performance in classifying a category at a specific IoU threshold.

5. Finally, calculate the mean average precision (mAP) to assess the model's accuracy in detecting objects across all categories at a given IoU threshold.

This section provides an overview of how we integrate metrics into our comparison software. Our comparison software focuses on assessing the performance of Mask R-CNN versus the YOLOv5 model in object detection tasks. Since the output of the two models is formatted to produce the same data structure, a list of PredictObject instances, we can further process their output in a similar manner. The following discussion will focus on evaluating the object detection performance of the Mask R-CNN model, unless otherwise specified.

6.3.1 Prediction and Ground-truth Ojbect IoU Calculation

As stated in Sections **??** and **??**, the ground-truth data is provided in the form of a list of TruthObject instances, whereas each model produces a list of PredictObject instances. It should be mentioned that each item in the TruthObject list represents an object present in the image, and the list is not ordered in any particular way. On the other hand, each instance in the PredictObject list represents an object detected by the model in the image and is sorted from the most confident detection to the least confident detection. It is important to note that there is no direct correspondence between the items in the two lists with respect to their location. Therefore, it is not possible to determine which PredictObject instance corresponds to the detection of a particular TruthObject instance.

In order to map the elements in the PredictObject and TruthObject lists, we employ a greedy brute force algorithm implemented in the *preds_choose_truths_map* function in the *src/metrics.py* file. This function takes in the two lists and produces a list of hashmaps, where each hashmap maps a PredictObject instance to a TruthObject instance and provides their corresponding Intersection over Union (IoU) value. An example output of the *preds_choose_truths_map* function is as follows:

```
1 [
2     {
```

```
        "iou_score": 0.959472981564043,
        "pred_obj": <PredictObject object at 0x299dee950>,
        "truth_obj": <TruthObject object at 0x2f4359f50>
    },
    {
        "iou_score": None,
        "pred_obj": <PredictObject object at 0x292a48210>,
        "truth_obj": None
    },
    {
        "iou_score": None,
        "pred_obj": None,
        "truth_obj": <TruthObject object at 0x2f435a490>
    }
]
```

The main idea behind the greedy mapping algorithm is to assign the PredictObject instance to the TruthObject instance with the highest IoU score among all available TruthObject instances. The greedy algorithm follows the following two steps. Firstly, for each instance in the PredictObject list, we calculate the IoU value between that instance and each available TruthObject instance. The IoU calculation between any PredictObject instance and TruthObject instance is straightforward as they both have their bounding box's top-left and bottom-right corner coordinates. These coordinates are used to calculate the intersection and union area between the two boxes, and then the IoU as shown in the *get_box_iou* function in the *src/metrics* module. Secondly, the PredictObject instance is mapped to the TruthObject instance that has the highest IoU score among all the available TruthObject instances. Then, this mapped TruthObject instance is marked as used or unavailable for subsequent instances in the PredictObject list.

This greedy mapping scheme effectively achieves two goals. The first goal is to assign each PredictObject to a TruthObject instance if available. When the number of instances in the PredictObject list exceeds the number of instances in the TruthObject list, any additional PredictObject instances will be assigned a value of None, and vice versa. The second goal is to reduce the model's overall performance further when a high-confidence detection has a lower IoU score with its corresponding ground-truth object compared to when a low-confidence detection has the same IoU score with

its corresponding ground-truth object. By allowing the high-confidence PredictObject instances to choose their mapping first, the lower-confidence PredictObject instances will have fewer available TruthObject instances to choose from, forcing them to have a lower priority compared to the higher-confidence PredictObject instances.

6.3.2 Confusion Matrix Generation

Using the mapping between the PredictObject and TruthObject lists, we can establish a label mapping between their classification labels, IoU, and confidence scores. This mapping consists of four lists: the ground-truth label list, the predicted label list, the confidence score list, and the IoU score list, similar to the representation in Table **??**. Each element in these lists corresponds to the element of the other list in terms of location.

The label mapping is then utilized to filter by IoU and confidence thresholds before generating the confusion matrix. In *src/main.py* file, the *IOU_THRESHOLDS* variable defines a list of IoU thresholds to be filtered by, while the *CONFIDENCE_THRESHOLDS* variable defines a list of confidence thresholds. In case the IoU score between a pair of PredictObject and TruthObject instances is lower than the IoU threshold, the truth label associated with the TruthObject instance is converted to "none". This conversion ensures that we treat the low IoU score detection as a False Positive (FP) [**metrics_survey_2020**]. Similarly, if the confidence score is below the confidence threshold, the predicted label corresponding to that PredictObject instance is set to "none". This effectively treats low confidence score detections as False Negatives (FN).

The resulting filtered label map is then fed into the *multilabel_confusion_matrix* function of the *sklearn.metrics* module [**skm_multilabel_confusion_matrix**]. This function takes in the ground-truth and predicted label lists as input. Additionally, since we have more than two categories in our two lists, we also need to provide a list of supported category labels to the function. The *multilabel_confusion_matrix* function returns a list of confusion matrices, with each matrix corresponding to a label in the list of supported categories in terms of position. The current image's resulting hashmap will be included in the total hashmap of all processed images within the batch. This task is executed by the *update_lic_cmatrix* function, which is implemented in the *src/main_utils.py* file.

Further explanation and an example of the filtering process and the generation of a confusion matrix are demonstrated in Subsections **??** and **??**. These processes are implemented in the *multilabel_cmatrix* function within the *src/metrics.py* file. It should be noted that the model will have a

unique confusion matrix for each categorized label at each IoU threshold and confidence threshold. Therefore, the *multilabel_cmatrix* function returns a three-level nested hashmap, where the key at each level is the label name, IoU threshold value, and confidence threshold value. The value of the most inner hashmap is a 2×2 confusion matrix. A sample output of the nested hashmap used to store confusion for a batch of images is:

```
{
    "bicycle": {
        0.5: {
            0.65: [
                [108, 7],
                [8, 8]
            ],
            0.75: [...],
            0.85: [...]
        },
        0.7: {...},
        0.9: {...}
    },
    "bus": {...},
    "car": {...},
    "motorcycle": {...},
    "person": {...},
    "truck": {...}
}
```

where the confusion matrix is stored in a hashmap first by label, then by IoU threshold values of 0.5, 0.7, and 0.9, and finally by confidence threshold values of 0.65, 0.75, and 0.85. The displayed confusion matrix is for the "bicycle" category at an IoU threshold $t_{IoU} = 0.5$ and a confidence threshold $t_{confidence} = 0.65$.

6.3.3 Accuracy, Precision, Recall, and F-score Calculation

In order to determine the accuracy (ACC), precision, recall, and F-score of a label at a specific IoU and confidence threshold, the corresponding confusion matrix, as stated in the previous subsection,

is utilized. Recal from Section **??**, these metrics can be computed with the following formulas:

$$\text{Precision} = \frac{TP}{TP + FP} \qquad \text{Recall} = \frac{TP}{TP + FN}$$
$$\text{ACC} = \frac{TN + TP}{TN + FN + TP + FP} \qquad \text{F-score} = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

where the numerator of a metric equals 0, the metric will be assigned a value of 0. This is because all terms in the numerator are also used in the denominator. By assigning a value of 0 to the metric when the numerator is 0, regardless of the denominator, we can effectively avoid any instances of invalid division by 0. This serves as a sufficient condition for ensuring the metric remains valid in such edge cases.

These metrics are implemented in *calc_accuracy*, *calc_precision*, *calc_recall*, and *calc_f1* functions in the *src/metrics* module. Additionally, we also have the helper function *calc_aprf1*, implemented in the same module, which calls these four functions to calculate accuracy, precision, recall, and f-score for a given confusion matrix. After computing the four metrics for each confusion matrix, the F-score with the highest value for a particular IoU threshold can be determined by comparing the F-scores at all confidence thresholds within the corresponding IoU threshold hashmap. The highest F-score indicates the level of confidence at which the model achieves the best combination of precision and recall values. Note that since these metrics are computed for each confusion matrix, thus we will also have a three-level nested hashmap, similar to the nested hashmap used to store the confusion matrix. An example of the metric nested hashmap's output for a batch of images is:

```
{
    "bus": {
        0.5: {
            0.65: {
                "accuracy": 0.925,
                "precision": 0.5,
                "recall": 0.6666666666666666,
                "f1": 0.5714285714285715
            },
            0.75: {...},
            0.85: {...},
            "highest_f1": 0.5714285714285715,
```

```
            "highest_f1_at_conf": 0.75,
        },
        0.7: {...},
        0.9: {...}
    },
    "car": {...},
    "bicycle": {...},
    "person": {...},
    "motorcycle": {...},
    "truck": {...}
}
```

6.3.4 Average Precision (AP) and Mean Average Precision (mAP) Calculation

As stated in Subsection **??**, by computing the precision and recall metrics for each label at various combinations of IoU and confidence threshold, we can determine the Average Precision (AP) metric. This metric represents the model's performance in detecting objects of a particular class at a given IoU threshold. We adopt the N-point interpolation AP for our comparison, similar to the PASCAL VOC and COCO benchmarks. Recall that the formula for N-point interpolation AP is as follows:

$$AP_N = \frac{1}{N} \sum_{R \in \mathbb{R}_N} P_{interp}(R) \qquad with \qquad P_{interp}(R) = \max_{R':R' \geq R} P(R') \tag{6.1}$$

where the set of N interpolation $\mathbb{R}_N$ is $\left\{0, \frac{1}{N}, \frac{2}{N}, ..., \frac{N}{N}\right\}$. We also recall that the $P_{interp}(R)$ term is the highest precision value among all recall point R' that are larger than R, instead of being the precision observed at the recall R, as stated in Subsection **??**.

The *calc_ap* function in the *src/metrics* module is responsible for computing the AP metric. This function requires a list of precision and a list of recall, as well as the number of interpolation points N, as input parameters. It should be noted that the function assumes that the precision and recall lists have corresponding elements based on their index. To obtain the $P_{interp}(R)$ value, a simple yet effective algorithm is employed. The key concept of this algorithm is that the $P_{interp}(R)$ value will always be equal to the maximum precision value of all the subsequence precision points in the list at any given precision point. By traversing the precision list in reverse order, keeping track of the highest precision value encountered so far, and updating each precision value that is lower than the

current highest precision, we can obtain the list of $P_{interp}(R)$ values in linear (O(1)) runtime. However, since we must interpolate N recall points equally spaced in the range [0, 1] for N-point interpolation, the interpolated recall points might not be present in the recall list. In this case, we will interpolate the smallest higher recall point. For instance, if we need to interpolate the recall value of 0.55, but our sorted recall list is [..., 0.5, 0.58, ...], then the $P_{interp}(0.55)$ value will be equal to $P_{interp}(0.58)$.

Once the AP for each class label has been computed, calculating the mean Average Precision (mAP) is a simple process that involves applying the formula outlined in Subsection **??**. The formula can be expressed as:

$$mAP = \frac{1}{K} \sum_{i=1}^{i=K} AP_i$$

where K be the number of catagory label, and AP_i is the average precision value of the ith class.

6.4 Experiment Result

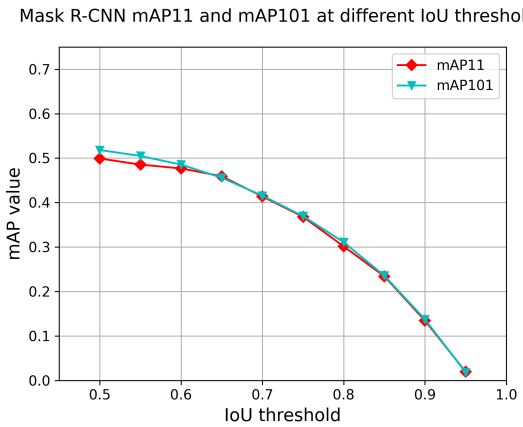
In our comparision, we look at the 11-point interpolation mean average precision (mAP) and 101-point interpolation mAP at 10 different IoU thresholds from 0.5 to 0.95 with the step size of 0.05 (Figure **??** and **??**). This configuration is the same of the AP metrics of the COCO benchmarks [**coco_2014**]. We notice that the mAP_{11} as good of approximation as the mAP_{101} in approximating the area under the precision-recall curve, while mAP_{11} is faster as it require less operations compare to mAP_{101}. However, for the remaining of the comparision experiment, we utilize the mAP_{101} as it is more accurate and the evaluation runtime is not important in our comparision software.

We start our experiment with evaluate the Mask R-CNN model performance on the nuImages validation set. When we look at the mAP_{101} at IoU threshold of 0.5, the pretrain Mask R-CNN model is accurate at 51.82%, signal that the model able to detect human and vehicle more than 50% of the times without the need of domain training. However, the mAP_{101} score drop significiantly as the IoU threshold increase. This indicate that the region proposal network (RPN) and bounding box regressor are struggling in correctly proposing RoI and finetuning it to align well with the ground-truth object.

In constrast to Mask R-CNN, the pretrain YOLOv5 model only have the mAP_{101} score of 47.18% at IoU threshold of 0.5, which indicate that the model is struggling with able to detect the object present in the image. However, YOLOv5 mAP_{101} score is gradually decrease as we increase the IoU threshold until 0.85. This means that YOLOv5 detector is strong in localization task with majority of predicted bouding box, that has IoU larger than 0.5, will have the IoU score of 0.85 approximately.

t_{iou}	mAP11	mAP101
0.50	0.4991	0.5182
0.55	0.4854	0.5048
0.60	0.4767	0.4853
0.65	0.4592	0.4559
0.70	0.4136	0.4154
0.75	0.368	0.3696
0.80	0.3015	0.3108
0.85	0.2341	0.2355
0.90	0.1343	0.1373
0.95	0.0193	0.018

(a) mAP_{11} and mAP_{101} values

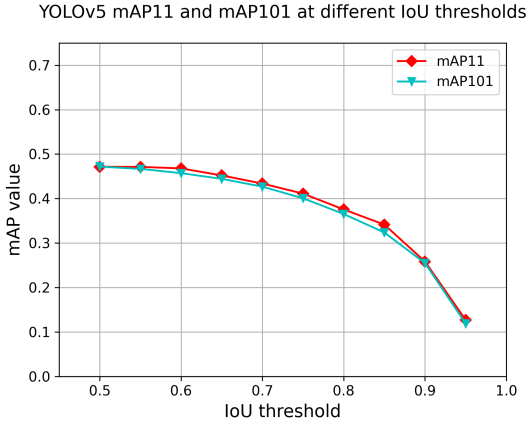


(b) mAP_{11} and mAP_{101} curves

Figure 6.1: Mask R-CNN model 11-point interpolation mAP (mAP_{11}) and 101-point interpolation mAP (mAP_{101}) at different IoU thresholds. The IoU threshold list is defined as [0.5:0.05:0.95], which means the IoU thresholds are from 0.50 to 0.95 with an equal step size of 0.05

t_{iou}	mAP11	mAP101
0.50	0.4715	0.4718
0.55	0.4712	0.4668
0.60	0.4679	0.4571
0.65	0.4520	0.4443
0.70	0.4337	0.4268
0.75	0.4112	0.4005
0.80	0.3755	0.3652
0.85	0.3414	0.3236
0.90	0.2582	0.2548
0.95	0.1271	0.1190

(a) mAP_{11} and mAP_{101} values



(b) mAP_{11} and mAP_{101} curves

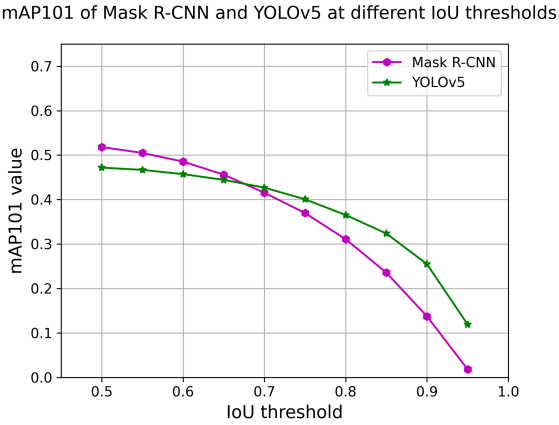
Figure 6.2: YOLOv5 model 11-point interpolation mAP (mAP_{11}) and 101-point interpolation mAP (mAP_{101}) at different IoU thresholds. The IoU threshold list is defined as [0.5:0.05:0.95]

By comparing the mAP_{101} score of the Mask R-CNN and YOLOv5 models, we can clearly see that Mask R-CNN is better at detecting the present of object compare to YOLOv5, however it have a poor localization performance. With poor localization, many detection make by Mask R-CNN is consider as wrong (False Positive) as IoU threshold increase, and thus reduce the overall mAP_{101} score significiantly. On the other hand, YOLOv5, with the high accuracy localization, become a better detection model as the IoU threshold approaching 1 (Firgure **??**). Therefore, for autonomous vehicle application where both objectiveness and localization are important, YOLOv5 is a better

suited model compare to Mask R-CNN model. In addition to mAP_{101} score, the break down AP_{101} score for each class of the two model at IoU thresholds of 0.55, 0.65, 0.75, and 0.85 is shown in Figure **??**. Nonetheless, eventhough YOLOv5 is better compare to Mask R-CNN, at IoU threshold of 0.85 and confidence threshold of 0.5, YOLOv5 still unable to detect majority of ground truth object. The visualization of number of missed detection for Mask R-CNN and YOLOv5 model are shown in Figure **??**

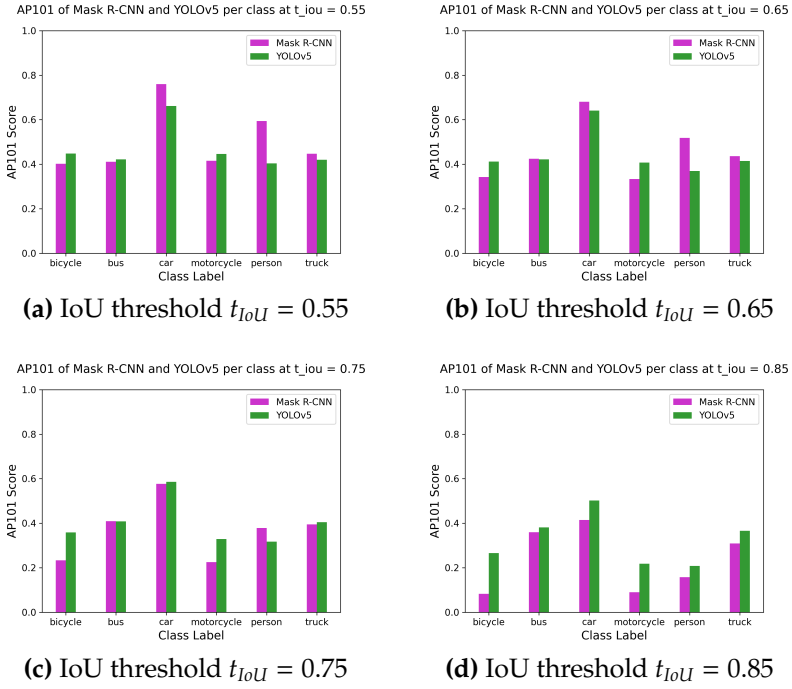
t_{iou}	Mask R-CNN	YOLOv5
0.50	0.5182	0.4718
0.55	0.5048	0.4668
0.60	0.4853	0.4571
0.65	0.4559	0.4443
0.70	0.4154	0.4268
0.75	0.3696	0.4005
0.80	0.3108	0.3652
0.85	0.2355	0.3236
0.90	0.1373	0.2548
0.95	0.0180	0.1190

(a) mAP_{101} values



(b) mAP_{101} curves

Figure 6.3: Mask R-CNN's mAP_{101} and YOLOv5's mAP_{101} at different IoU thresholds. The IoU threshold list is defined as [0.5:0.05:0.95]



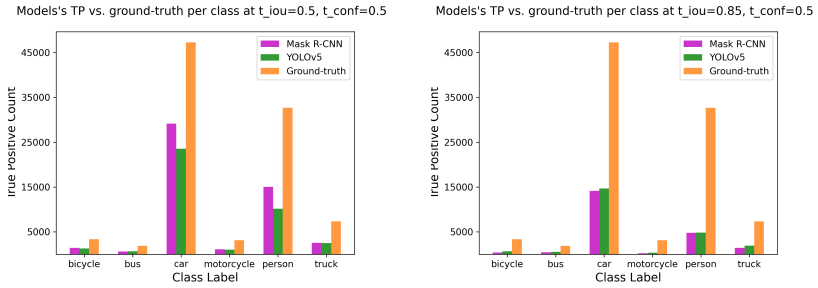
(a) IoU threshold $t_{IoU} = 0.55$

(b) IoU threshold $t_{IoU} = 0.65$

(c) IoU threshold $t_{IoU} = 0.75$

(d) IoU threshold $t_{IoU} = 0.85$

Figure 6.4: AP_{101} score of Mask R-CNN and YOLOv5 model for each supported label at IoU thresholds of 0.55, 0.65, 0.75, and 0.85



(a) True Positive at IoU threshold of 0.5 and confidence threshold of 0.5

(b) True Positive at IoU threshold of 0.85 and confidence threshold of 0.5

Figure 6.5: Number of detected objects by Mask R-CNN and YOLOv5 versus the number of ground-truth objects. The comparisons are at IoU thresholds of 0.5, 0.85, and the confidence threshold of 0.5. The comparison is displayed for each supported label.

CHAPTER 7

Future Work

CHAPTER 8

Conclusion