Contents

CHAPTER 1

Introduction

When comparing the human brain with the machine, we can see that humans excel at interpreting data on a high level and in a more abstract format. In contrast, the machine has higher computational power when it comes to raw numerical data. Additionally, by Moore's law, the number of transistors in a dense integrated circuit doubles every two years, which is associated with the computational power of machines becoming more and more powerful. Therefore, it is reasonable to believe that if a machine can interpret and understand the representation of objects beyond raw numerical data, then the machine can process and react much faster than humans. In other words, vision in machines proves to be very beneficial to humans when it comes to situations requiring quick reactions. Such situations can be seen or encountered every day through traffic. If done correctly, the machine with vision can predict and avoid traffic accidents faster than humans. Thus, the machine can help create a 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. One of the leading companies in the field of the autonomous vehicle 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. However, in July 2022, 2 different Tesla cars crashed while on autopilot, and each crash caused the death of a motorcyclist. Additionally, in 2019, a Tesla autopilot crashed into a Honda Civic and killed two people. Not to mention, there are 273 Tesla crashes involving the autopilot system reported just in 2019. 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. 1.1].

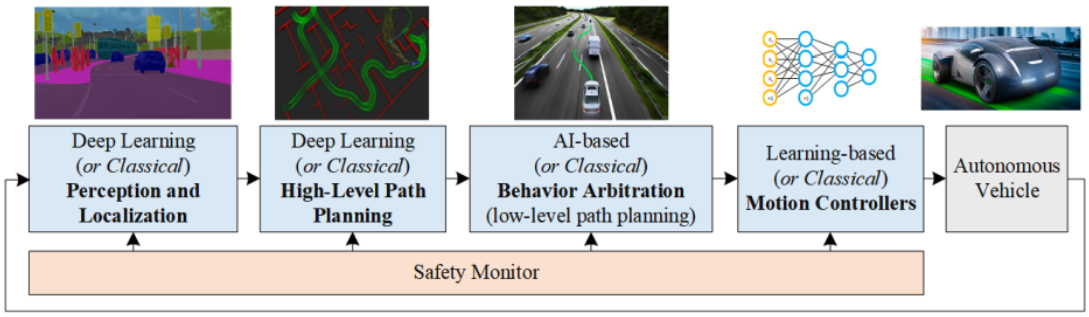


Figure 1.1: Deep Learning Based Autonomous Driving Modular Pipeline [7]

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?" [14]. To perform this function, the module needs to utilize the sensing hardware and the software to understand the scene. Such hardware can be LiDAR, cameras, or other sensors. Nonetheless, they are all used for getting the information surround the vehicle and feeding those data to the software functionality of the module.

Getting the information surround the vehicle is straightforward as it simply records and passes those recorded data to the software functionality. However, understanding the scene is a more challenging task and not as straightforward. Unlike human beings, the 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 exist. 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. On the other hand, video instance segmentation detects each instance of each category for each scene. Comparing semantic segmentation and instance segmentation, we can see that instance segmentation proposes a higher accuracy 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, a truck, and a bicycle if present instead of just vehicles class like semantic segmentation.

Since video instance segmentation algorithms give higher detection accuracy, in this paper, we will assume an algorithm from this group will be 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. More specifically, this paper will discuss the building block of image instance segmentation algorithms. We then analyze and compare the performance of two well-known image instance segmentation algorithms, the Mask R-CNN and You Only Look Once version 7 (YOLOv7). Furthermore, we will want to identify if misidentified by these image instance segmentation algorithms causes the autonomous driving pipeline to fail and result in accidents.

CHAPTER 2

Neural Network

Instance segmentation algorithm group is a subgroup of the object detection algorithm group. All algorithms in the object detection group required to do two tasks. Those tasks are generate a bounding box surrounding each object in the image and clasify the object in each bounding box. We will first discuss the algorithm that have been use for this classification task. Since in each bounding box, there is exactly one object need to be classify, an algorithm from the image classification task, a subset of computer vision task, will be applied.

In the early day, a set of algorithms called machine-learning was proposed for the image classification task, but most were still designed manually by humans. Conventional machine-learning use feature extraction functions to map raw data to feature vector. The feature vector is the only suitable data that allow the learning subsystem of machine-learning to detect patterns and classify the input. For that reason, the accuracy and effectiveness of machine-learning method 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. 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 the manually designed machine-leaning algorithm 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 architecture. 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 feeded to the algorithm, the more accuracy its interpretation ability become.

At the highest level of abstraction of ANN, there exist two types of neural network architecture. The first and simpler one is feedforward architecture which only allows its signal to travel from input to output. The feedforward network is widely utilized in grid pattern processing tasks. 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. Feedback connections enable RNN 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

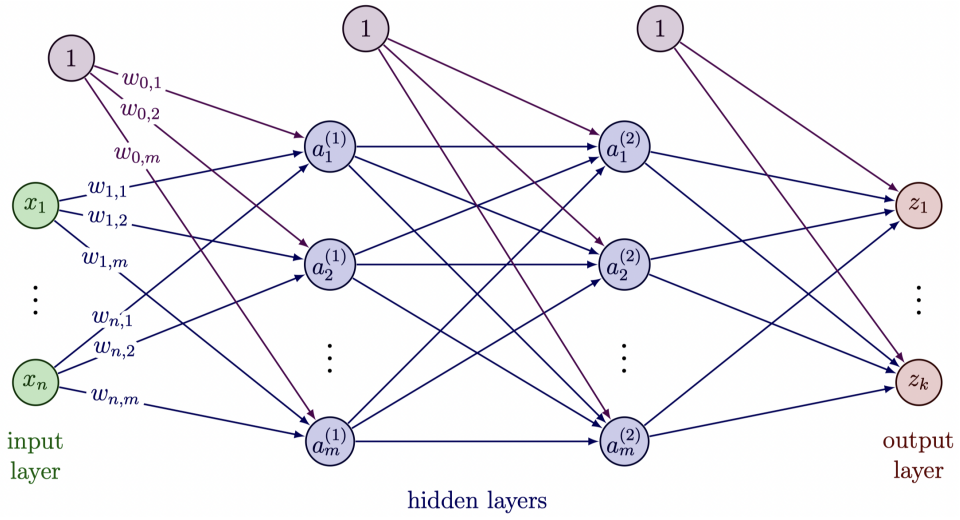


Figure 2.1: A Fully Connected Feedforward Neural Network

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 [11]. On a high level, a feedforward neural network algorithm can be decomposed into 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

does with its current setup. The third phase is gradient descent, where the algorithm determines which part of the network cause it to perform poorly. The last phase is backpropagation, where the algorithm updates the internal part of 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 precise over time. We will discuss more about 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 need to 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 node in Figure 2.1. 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. On 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 2.1 is a fully connected feedforward neural network with a depth of three.

Each layer consist 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, like improved training time and avoiding overfitting problems [section 2.3], when normalizing the activation signal of every neuron [12]. 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 layer, 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 neurons' hidden layers perform better [21], and deeper networks result in more advanced learning with some issues like gradient vanishing and more [section 2.3]. Discuss overfiting and gradient vanishing problem in section 2.3

Each neuron in a layer can affect a neuron from the next layer through a **connection**. The connection is represented by a line between two neurons in Figure 2.1. In fully connected networks, each neuron in a layer will be affected by all the neurons from the previous layer. In other words, each neuron will have a connection with every neuron in the previous and next layer of the network. Associate with each connection is a numerical value representing the **weight**. The weight of a neuron infers how influence the neuron will be 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 of 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 initialize, its weights are randomly assigned using a Gaussian or uniform distribution [11]. Discuss Gaussian or uniform distribution in section 2.3 These weights are adjusted as the neural network algorithm process to best approximate its mapping function. The weight from neuron a_i to a neuron a_j in the next layer will be denoted as $w_{i,j}$ with the exception of bias's weight.

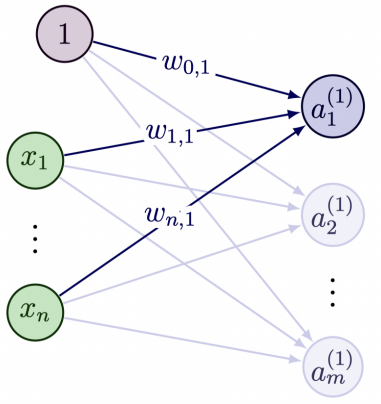
In addition to the weight, neurons in each layer other than the input layer are also influenced by a **bias node**. A bias node is represented by a node that always holds value 1. The bias node has connections to every neuron in the layer it affects. Each bias's connection also has a weight associated with it. The connection between bias and neuron a_j in the affected layer will have 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 significant the activation signal of a neuron must be before that neuron gets propagated to the next layer of the network. 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 refers as the 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 learnable in 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. 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 function are the **summation function** and the **activation function** [21].



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 can be 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. The neuron's net input will then be 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 will result in a linear function. This fact implied 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 [8].

There are numerous activation functions proposed, each with different strengths and weaknesses. However, in practice, there are four functions and their variance that are widely used by the ANN algorithm. These four activation functions are the linear, sigmoid, hyperbolic tangent (tanh), and

rectified linear unit (ReLU). The formula and graphical representation of each function are shown in Table 2.2.

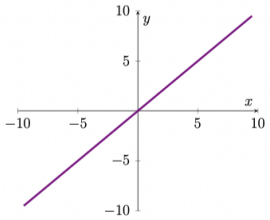
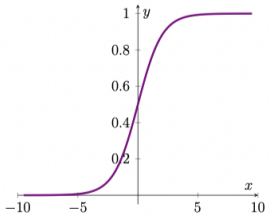
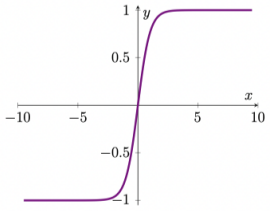
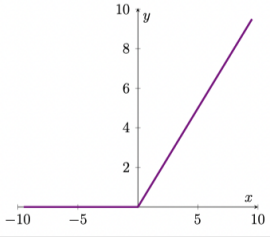
Activation Function	Graph	Formula
Linear		$\phi(x) = x$
Sigmoid		$\phi(x) = \frac{1}{1+e^{-x}}$
Tanh		$\phi(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
ReLU		$\phi(x) = max(0, x)$

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 forgiving and undemanding about the resources required for training. 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, sigmoid and tanh map 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 [11] [section

2.3]. However, the mapping also causes sigmoid and tanh to suffer from the vanishing gradients problems. Different from sigmoid and tanh, ReLU map positive signal to itself, thus able to avoid the vanishing gradients problems. The first disadvantage of ReLU is it does not have any upper bound, which require the network to have some additional nomalization layer like batch normalzation (BN), weight normalizaiton (WN), and layer normalizaiton (LN) to optimize the training process [3] [section 2.3]. The second disadvantage of ReLU is it suffer from the dying ReLU problem which cause by the fact that all negative signal is map to 0 [section 2.3]. Despite the dying ReLU problem, ReLU proves to be very efficient and accurate in practice; thus, it is the most used activation function currently [13]. 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., [11].

2.1.3 Error Evaluation

Once the forward propagation is completed, the algorithm will need 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 [13]. 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. 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 use 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. 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 image 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, 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 on how to reduce the total error. 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** refer to when the network see the entire training data set exact one time.
- A **batch** is the number of example that pass through the network exact one before updating the network's parameters.
- An **iteration** refer to number of time a batch of data need to pass through a network to complete an epoch. This also state the number update for an epoch.

As an example, if our image classification training data set have 1000 images with the batch size of 250, then we need 4 iteration to pass the entire training set throught 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. However, by having the batch size equal to an epoch, BGD guaranteed to find a global minimum on the Cross-Entropy 3D convex surface. 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 pass through the network; thus it is much faster in updating weight and bias than BGD. Despite having a faster update, SGD suffers from a high variance of training examples since improving error for one example does not equate to improving 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. As an additional note, using SGD requires the training set to be shuffled before input to the network as the order of example 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 [1, 15]. Some studies also propose the use of an adaptive batch size where the network starts with small batch size, then increases after each update [12]. 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 show in Figure 2.3.

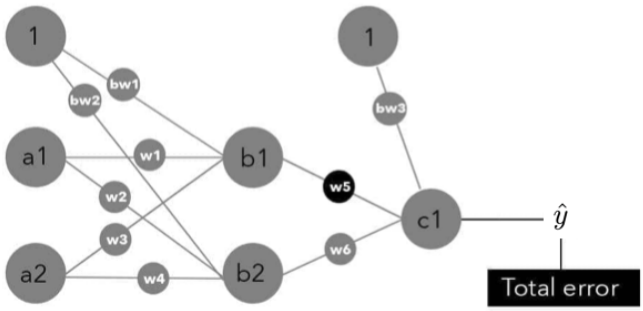


Figure 2.3: Simple Neural Network [21]

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 2.1 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 affect $\hat{y}$ 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 2.2 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 paticular weight or bias changes the network's total error, and thus ready to update each weight and bias accordingly.

2.1.5 Backpropagation

Backpropagation refer to a phase in which the algorithm using gradients of weight or bias to update their value and bring the network's total error to global minimum. Since the gradient give the slope

of the loss function with respect to a weight or bias, and backpropagation update their value to approaching global minimum, thus gradient descent along with backpropagation process enable neural network to has 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 refer to the algorithm's learning rate.

Learning rate, denoted η, is a network's hyperparameter and it determine how fast the network learn. In the updating weight function, equation 2.3, the learning rate directly affect the size of the step when the algorithm move toward global minimum for total error. Large leaning rate means bigger step toward minimum, thus result in faster learning, while smaller learning rate value result in slower leaning. However, a large leaning rate can also affect the network's ability to reach global minimum as it can over step and pass optimal value. Studies has shown 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 [1]. Consider put the learning rate paragraph into [section 2.3]

As the backpropagation phase is completed, all weights and biases in the network get 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. At that moment, a test set will be passed through the network, and the network's total error will be returned to determine the accuracy of the algorithm.

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. 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 2.1. A simple CNN structure for handwritten digit classification is shown in Figure 2.4.

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

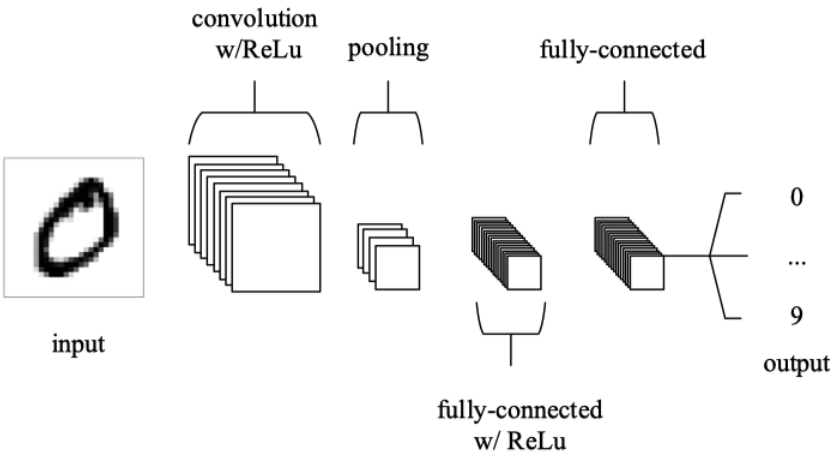


Figure 2.4: Simple CNN structure for handwritten digit classification [16]

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. 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. Consider a 64×64 RGB image, in a fully connected feedforward network, each neuron in the network will need to consider value from $64 \times 64 \times 3 = 12,288$ neurons from the previous layer. In other words, each neuron will have $12,288$ 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 downsampling the data, it would be more efficient, have less training time, and require less computational power.

CNN is 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 structure, we will discuss convolutional and pooling layer functionality in detail.

2.2.1 Convolutional Layer

Convolutional layer is responsible for making the spatial connection and extracting features from the image. 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 will require the kernel to have the size of $x \times x \times 3$ where x bounded by 2 and 64 inclusively. There are various types of kernels, and each type is designed to target a specific task. 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 2.5. 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.

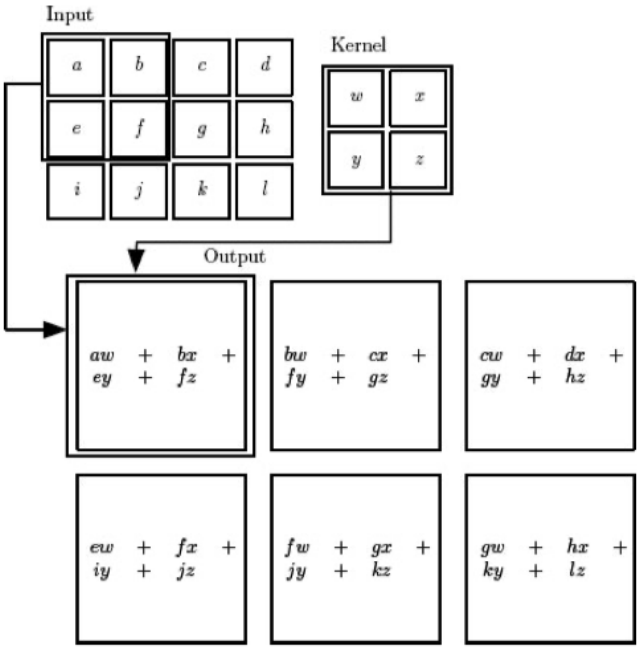


Figure 2.5: Scalar product for a kernel's activation signal [11]

Notice that a kernel's width and height must be equal, and the kernel itself must be symmetric. 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 will only have $4 \times 4 \times 3 = 48$ connections. Thus, the network will only need 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. 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 denoted a kernel as

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

An 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: reducing the number of neurons in the network. The use of the pooling layer help result in fewer parameters for the network and thus require less computational power. Similar to the kernel, the pooling layer also slide through a grid of value. 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 2.6. 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 the step the pooling layer will slide through the feature map, denoted $/x$ where x is stride value. Since pooling layer also slide throught a grid of value and output exactly one value for the

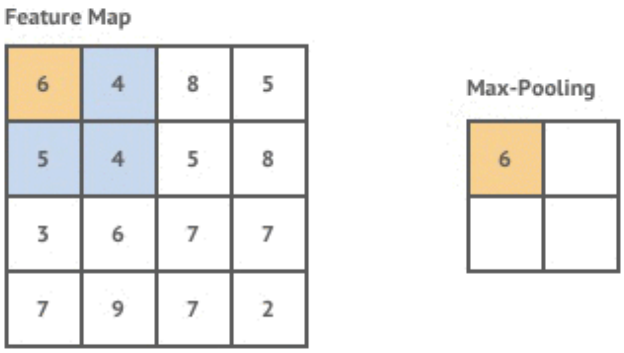


Figure 2.6: $2 \times 2,\ /2$ max-pool on [25]

subregion, thus a similar function to Equation 2.4 is used to calculate the output size for pooling layer. The output size can be calculate 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:

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

Thus, the algorithm 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 ouput apply to an input image.

2.2.3 Aditional term

TODO:

- overfitting problems
- gradient vanishing problem
- gaussian and uniform distribution

- exploding gradients
- batch normalzation (BN), weight normalizaiton (WN), and layer normalizaiton (LN), Local Response Normalization (LRN – find source that LRN is not that helpful)
- dying ReLU
- input saturation

CHAPTER 3

Instance Segmentation

An instance segmentation task is a computer vision task that involves determining the boundaries and identities of individual objects within an image or video. Unlike semantic segmentation, which classifies each pixel in an image as belonging to a particular class, instance segmentation assigns a unique label to each object in a scene and then identifies its boundaries. For example, in a picture of two vehicles one after another on the street, instance segmentation would be able to distinguish between the two vehicles by assigning them different labels and draw bounding box around them.

The main distinction between instance segmentation and semantic segmentation lies in the level of detail they provide. While semantic segmentation will divide an image into classes such as "vehicle" or "animal", instance segmentation will offer more details by distinguishing between each individual object within those classes. It does this by assigning a unique label to each object for identification. This means that with instance segmentation, it's possible to identify not only what type of thing is present within an image but also where it is located and how many there are - something semantic segmentation cannot do on its own.

On the other hand, instance segmentation and object dectection are quite similar on a high level. Upon comparing instance segmentation with object detection tasks reveals some further distinctions between the two tasks. While both involve locating objects within an image or video stream, object detection focuses on providing information about the location of the detected objects while instance segmentation goes further by identifying their outlines as well as providing additional information about them like the object size. Additionally, whereas object detection is used to detect multiple types of objects within one scene, instance segmentation is more focused on identifying individual instances within one specific type of object (e.g., two cars).

By the definition of instance segmentation task, we known that any algorithm which goal is to

complete this task must do two things. First, the algorithm must detect the location of the object, draw a bounding box srounding the object and the outline of the object. Secondly, the algorithm must be able to identify the class of the object resign inside the bouding box we draw from previous step. The algorithm must also aware the number of instance in this class at the second step.

From section 2.2, we have discussed about the structure and building block of a convolutional neural networks (CNNs). Throgh the discussion we also state how CNNs able to classify object in image classification task. However, by the definition of image classification task, the task assume the image have exactly one object and the algorithm will classify the class for entire image based on that one object. Therefore, we know that if we were to consider each bouding box as it own image, we can utilize a CNN to identify the class of the object within the bounding box. This is the main idea behind defferent variation of R-CNN for instance segmentation and objectt detection task.

CHAPTER 4

R-CNN Variation

Since the interested domain for this paper is the instance segmentation task, we will analyze the Mask R-CNN. Mask R-CNN is a popular algorithm that tries to solve 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 an object detection algorithm.

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) or Scale-Invariant Feature Transform (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, R-CNN model is composed of three modules [Fig. 4.1]. 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

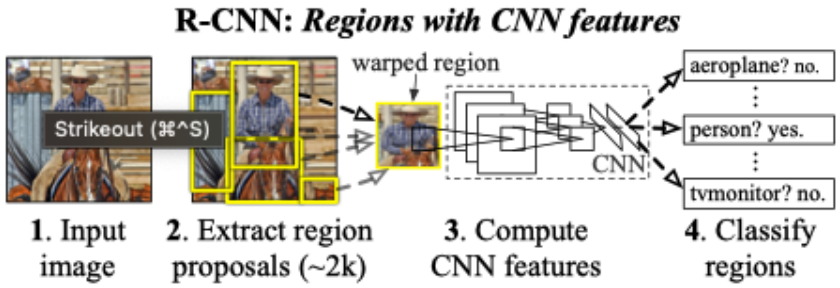


Figure 4.1: R-CNN overall architecture [6]

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 a 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.

consider describe segmentation statregy: segmentation able to generate multiple foreground and background segmentation and is trained to scoring if a forground segment is an object

In recent years, many study have show interest in improve the efficient of region proposal algorithm. One notable algorithm that given the same strength as the sliding window technique is selective search. Selective search was designed to combines the strength of both exhaustive search and segmentation [23]. 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 ustilize the underlying image structure to cluster pixels into different regions, taken from the strength of segmentation technique. Selective search also aim to complete three goals. These three goals are capture all scale, diversification, and fast to compute. Capture all scale is the idea that the algorithm must be able to detect object at different size in the image. Next, diversification requirement refer to method of grouping region. The algorithm must be able to combine region contain part of an object into one region. Additionally, for instance like a person inside a car or person in front of a car, the algorithm must be able to seperate the region for the car and the region for the person under diversification requirement. Lastly, fast to compute require the algorithm to not demain heavy computational power.

As for selective search overall behavior. It can be thought as two steps. The first step is perform bottom-up segmentation – descibed in Efficient Graph-Based Image Segmentation by Felzenszwalb and Huttenlocher [4] – to generate initial sub-segmentation. The second step is combine similar sub-segmentation recursive using a similarity score between subsegment. The similarity score is a combination of four similarity grouping criteria. These four criteria are color similarity, texture similarity, size similarity, and fill similarity [23]. The reason that color and texture similarity between regions is needed is due to same object mostlily will have the same texture or shade of color. In the original paper, the color and texture similarity criteria score can be descibed by an equation only based on fixed number of values taken from histogram of each color channel. Thus, these two similarity score require minimal to no computational power to compute. Similarly, two region that have majority of the area overlap on each other are most likely described the same object. Therefore, the use of fill similarity score allow the algorithm to merge regions that are mostly overlap and keep regions that are not overlaping seperate from each other. The fill similarity score can be caculated 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 two regions take into consideration to make sure ensures that the algorithm identify all locations for different object at different scale. The size similarity score can be caculated 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 accout the diversify of the image data set. With this two steps, selective search can quickly classify foreground and background, then downsampling number of potential RoI existed in forground segmentation. Therefore, by utilize selective search allow the model to reduce number of falsify RoI compare to sliding window and require lower computational power.

In the original paper of R-CNN, in the first module – region proposal – utilize selective search algorithm [6]. However, as mention before, the R-CNN model can be thought of as three seperate modules, thus one can experiment R-CNN with different region proposal technique.

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 [6]. 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 [10]. 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 competes 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

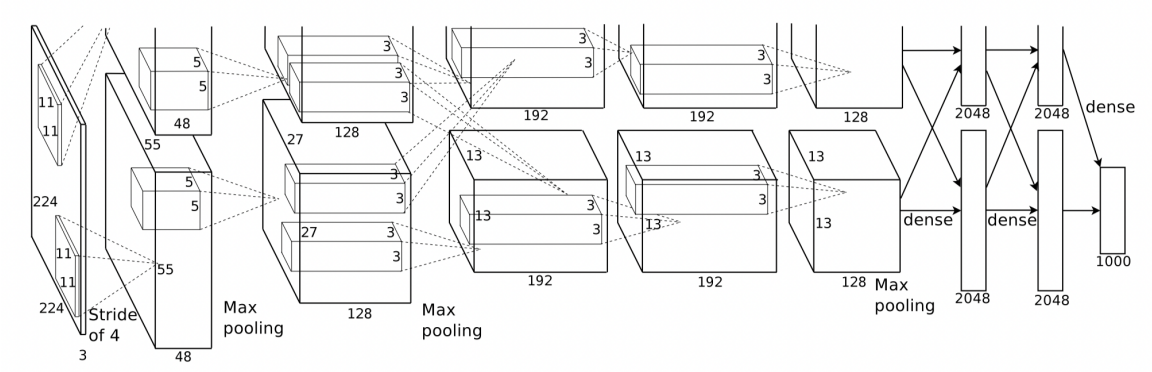


Figure 4.2: AlexNet's overall architecture [10]

requires particular layers to perform GPU communication. For example, in the overall architecture of AlexNet presented in Figure 4.2, 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 [10].

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 utilize the AlexNet architect implemented on the Caffe framework. R-CNN model pass each generated RoIs as seperated imaged to AlexNet for feature extraction. Since RoIs role is to find each object location in the image, thus AlexNet can assume each RoI only have exactly one object. AlexNet also require 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

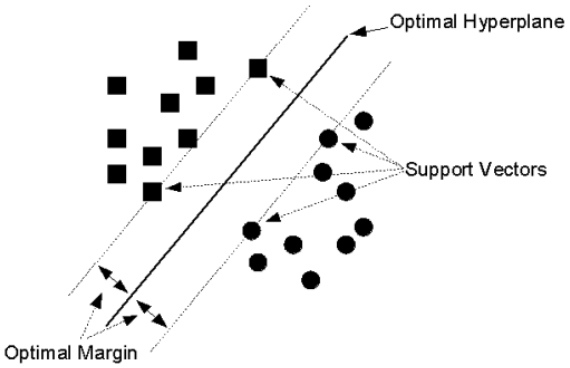


Figure 4.3: 2D linear SVM visualization [22]

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 4.3 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 [6]. 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 [5].

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, a 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 [5]. In the next section, we will mention the keynote of VGG16 architecture, follow by the discussion of Fast R-CNN model and its design decisions that lead to a higher mAP.

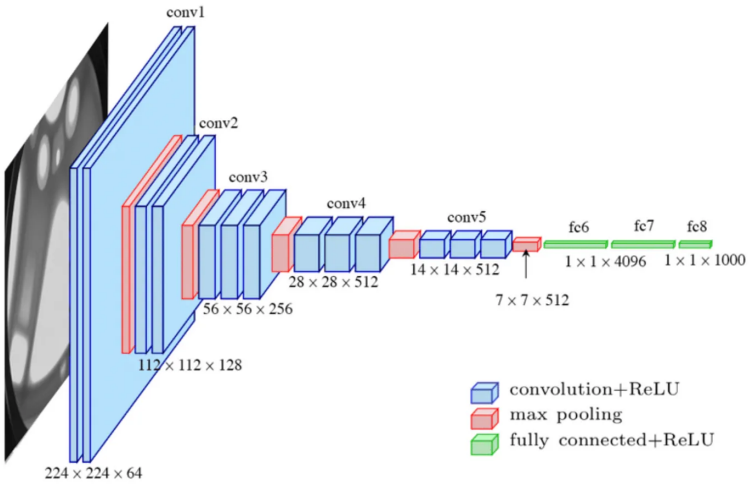
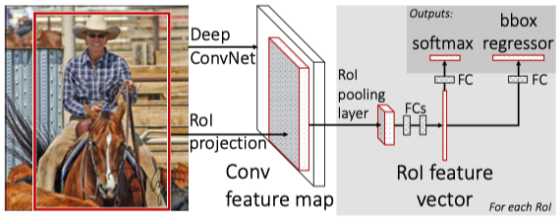


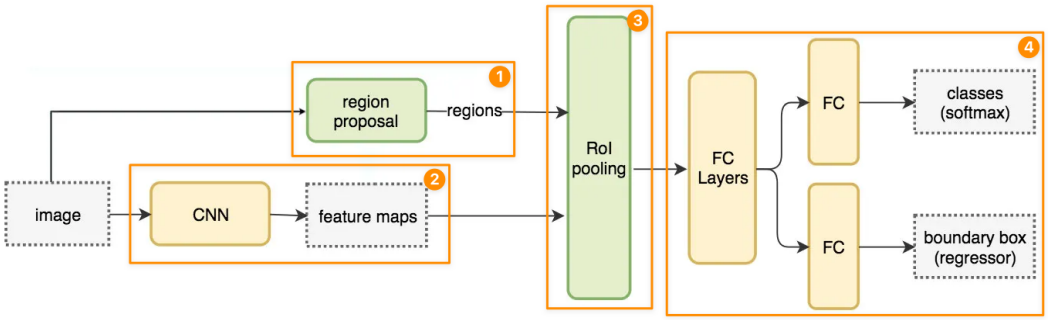
Figure 4.4: VGG16 architecture [20]

The VGG16 is a CNN model developed by the Visual Geometry Group (VGG) at the University of Oxford in 2014 [19]. 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. 4.4]. 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 [26]. A set of fully connected layers is then processed 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. 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 [5]

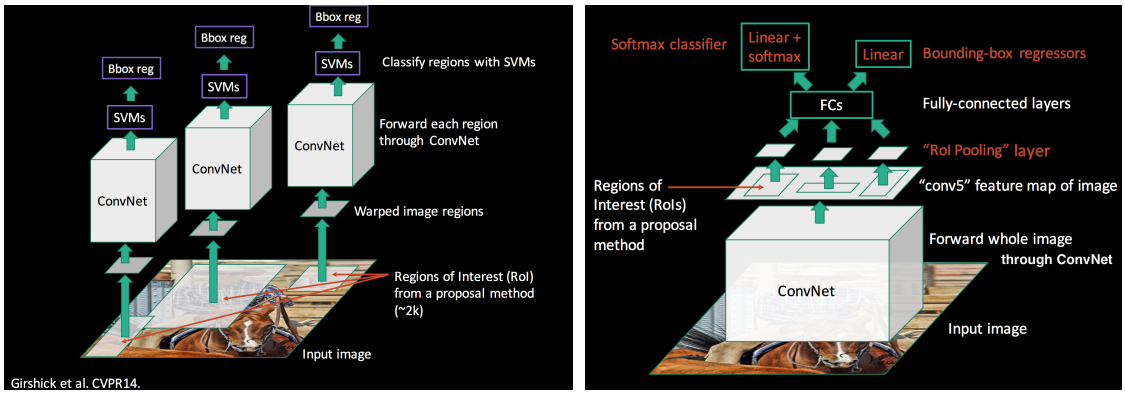


(b) Network flow [9]

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. 4.5]. 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 use RoI pooling layer to extracts the feature grid corresponding to the proposed RoI from the image feature map generated in stage two for each proposed RoI. The RoI pooling 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 [17] **(b)** Fast R-CNN architecture with stages [17]

Figure 4.6: R-CNN vs. Fast R-CNN architecture comparision

Comparing the architecture of R-CNN and Fast R-CNN, there are three main differences between the two models [Fig. 4.6]. 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 [5]. 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. 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 RoI pooling 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 RoI pooling layer achieves size-independent downsampling by dividing the RoI feature grid into a pre-defined $W \times H$ grid of subwindow, 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 subwindows 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 subwindow, respectively, and are rounded down to the nearest integer. In contrast to the traditional pooling layer described in Sec. 2.2.2, which used a sliding technique dependent on the input size, the division into a grid of equal subwindows enables the RoI pooling layer to have a fixed output grid size regardless of the size of the layer's input RoI. The RoI pooling layer then applies max-pooling to the pixel values in each subwindow, 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 [5]. 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 [5]. 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 [23]. Additionally, the RoI pooling 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 RoI pooling 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 subwindow in the projected RoI in terms of pixels. In other words, the RoI pooling 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.

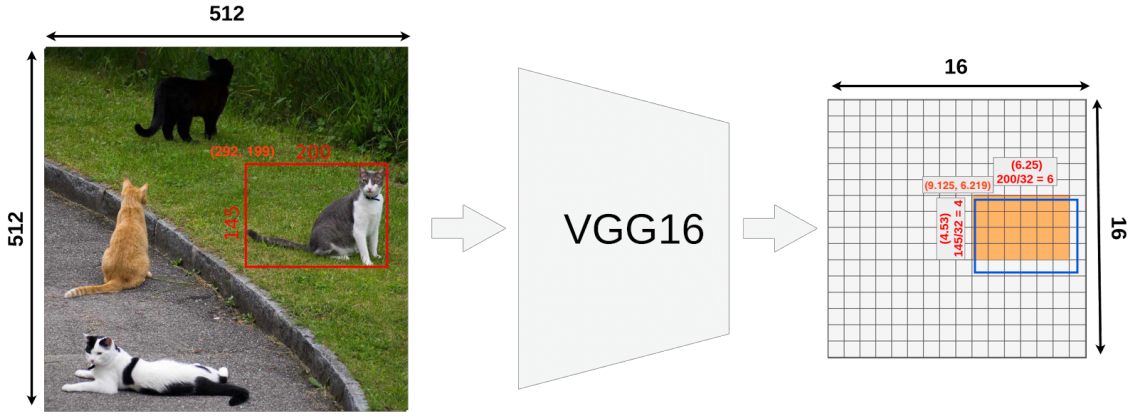


Figure 4.7: RoI projection to feature map [24]

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. 4.7]. 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:

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

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

When projecting to the feature map space, Fast R-CNN offsets and resizes the original projected

RoI (shown as the blue rectangle in Fig. 4.7) to align perfectly with n feature map pixels (shown as the orange area in Fig. 4.7), where n is the number of feature map pixels that can fit entirely within the original projected RoI. As shown in Fig. 4.7, we lose some pixels data (white part inside the blue rectange), and have additional noise datas (orage part outside the blue rectange). 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. After projecting and quantizing the input RoI, we obtain an RoI feature

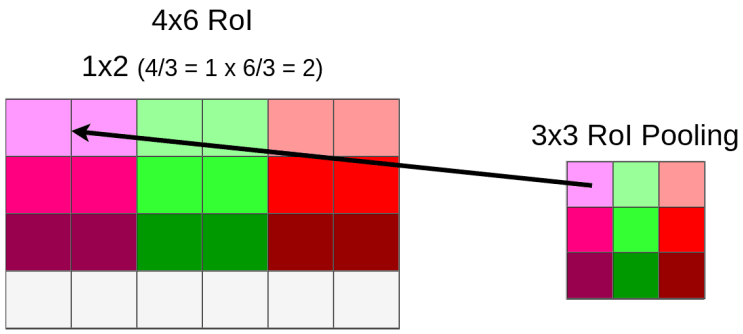


Figure 4.8: The RoI feature grid is divided into subwindows, as shown on the left. Performing max-pooling on each subwindow results in a 3×3 output matrix, as shown on the right. Each cell in the input image represents a feature map pixel, and each subwindow contains 2 pixels and has a unique color. Each cell in the output 3×3 matrix represents the highest feature map pixel out of all pixels in each corresponding subwindow. The white cells on the left do not belong to any subwindow and are not used in the RoI pooling process.[24]

grid. In our example, this grid has dimensions of 4×6 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 subwindows, each with a size of $\frac{4}{3} \times \frac{6}{3} \approx 1.3333 \times 2$. Since these sizes are expressed in terms of pixel counts, we must quantize them to the nearest integer values, resulting in subwindows of size 2×1. As shown in Fig. 4.8, 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 subwindows, leading to further loss of information. Combined every loss happen in the RoI pooling 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 145×200, 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 loss in information and offset is 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 RoI pooling 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. 4.4. 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 [18]. 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.

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. 4.9]. In another words, the RPN generate RoIs, then Fast R-CNN takes these generated RoIs as input and generate 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 higher accuracy scores compared to when generating RoIs with the selective search algorithm. One example is testing the same Fast R-CNN based on VGG-16 on the combined dataset of PASCAL VOC 2007 and 2012, the model with the Selective Search algorithm achieved an mAP score of 70%, which is not as accurate as when using RPN to generate RoIs at 73.2% in mAP score [18].

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 any size image as input and outputs several bouding boxes, each with a score representing the possibility

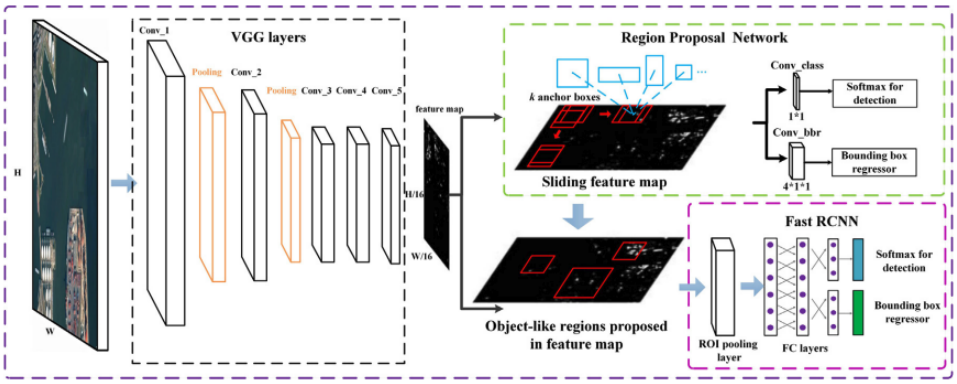


Figure 4.9: Faster R-CNN Overall Architecture [2]

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 able to generate RoIs in 0.01 second oppose to 2 seconds with Selective Search, while achieving higher accuracy score overall [18]. 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 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 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 that generates a set of four values representing the coordinate of the bounding box. The second branch is a classifier that produces a score for the probability that the object is in the window. At each sliding window location, RPN create an anchor at the center of the window. Then, with 3 different scaling factors and 3 different aspect ratios, RPN generate 9 new proposals with the anchor at the center [Fig. 4.10]. 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.

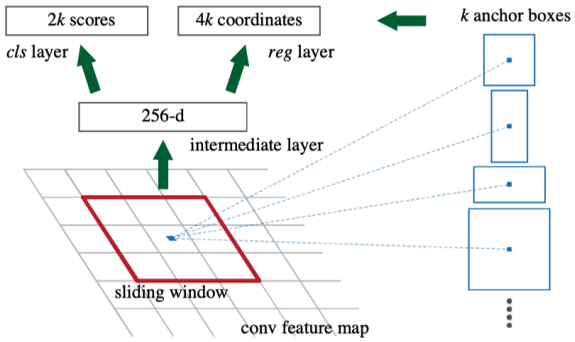


Figure 4.10: Region Proposal Network (RPN) [18]

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 over two classes (object vs. not object) and the regression loss over two coordinates (predict bounding box vs. truth bounding box), summed across all anchor points. If RPN samples all anchor points, then the model will favor negative false as the majority of RPN's generated box does not overlap with any truth box. To resolve this problem, RPN employs a sample-dropping technique and then randomly selects 256 samples to help balance positive and negative samples [18]. For sample dropping, RPN mark all predicted box with more than 70% of overlapping with any truth box as positive and any predicted box with less than 30% of overlapping with any truth box as negative, then drop any predicted box that is neither classified as positive nor negative. With that being said, the RPN training process consists of 3 steps. In the first step, the model initialized the share CNN with a pre-trained model like VGG-16 and randomly assigned weight values for the newly added layer in RPN. Then RPN is trained with the stochastic gradient descent (SGD) method, where each mini-batch consists of 256 samples that either overlap more than 70% or less than 30%. At each iteration, the model uses the multi-task loss as described previously to perform backpropagation.

With the understanding of end-to-end training for RPN and Fast R-CNN, we can now train the two components of Faster R-CNN independently. However, when training RPN and Fast R-CNN separately, optimizing the shared convolutional layers for the object proposal task does not guarantee that these layers will be optimized for the classification task because the network learns significantly differently for the two tasks. 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 [18]. Unfortunately, both methods are impeded by issues beyond this study's scope; therefore, we will not go into depth.

4.4 Mask R-CNN

CHAPTER 5

YOLO Variation

CHAPTER 6

Evaluation

TODO: - write about Intersection-over-Union (IoU)

CHAPTER 7

Conclusion

References

[1] Yoshua Bengio. "Practical recommendations for gradient-based training of deep architectures". In: *Neural networks: Tricks of the trade*. Springer, 2012, pp. 437–478 (pages 13, 15).

[2] Zhipeng Deng et al. "Multi-scale object detection in remote sensing imagery with convolutional neural networks". In: *ISPRS Journal of Photogrammetry and Remote Sensing* 145 (May 2018). DOI: 10.1016/j.isprsjprs.2018.04.003 (page 37).

[3] Yonatan Dukler, Quanquan Gu, and Guido Montúfar. *Optimization Theory for ReLU Neural Networks Trained with Normalization Layers*. 2020. DOI: 10.48550/ARXIV.2006.06878. URL: https://arxiv.org/abs/2006.06878 (page 10).

[4] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. "Efficient graph-based image segmentation". In: *International Journal of Computer Vision* 59.2 (2004), pp. 167–181. DOI: 10.1023/b:visi.0000022288.19776.77 (page 25).

[5] Ross Girshick. "Fast r-cnn". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1440–1448 (pages 29–33).

[6] Ross B. Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *CoRR* abs/1311.2524 (2013). arXiv: 1311.2524. URL: http://arxiv.org/abs/1311.2524 (pages 24, 26, 29).

[7] Sorin Grigorescu et al. "A survey of Deep Learning techniques for autonomous driving". In: *Journal of Field Robotics* 37.3 (2020), pp. 362–386. DOI: 10.1002/rob.21918 (page 2).

[8] Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. Dec. 2015 (page 8).

[9] Jonathan Hui. *What do we learn from region based object detectors (faster R-CNN, R-FCN, FPN)?* Feb. 2019. URL: https://jonathan-hui.medium.com/what-do-we-learn-from-region-based-object-detectors-faster-r-cnn-r-fcn-fpn-7e354377a7c9 (page 31).

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Communications of the ACM* 60.6 (2017), pp. 84–90 (pages 26–27).

[11] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), pp. 436–444 (pages 5, 7, 9–10, 17).

[12] Yann A LeCun et al. "Efficient backprop". In: *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48 (pages 6, 13).

[13] Zewen Li et al. "A survey of convolutional neural networks: analysis, applications, and prospects". In: *IEEE Transactions on Neural Networks and Learning Systems* (2021) (page 10).

[14] Shaoshan Liu. "Computer vision for perception and localization". In: *Engineering Autonomous Vehicles and Robots* (2020), pp. 77–96. DOI: 10.1002/9781119570516.ch6 (page 2).

[15] Dominic Masters and Carlo Luschi. "Revisiting small batch training for deep neural networks". In: *arXiv preprint arXiv:1804.07612* (2018) (page 13).

[16] Keiron O'Shea and Ryan Nash. "An introduction to convolutional neural networks". In: *arXiv preprint arXiv:1511.08458* (2015) (page 16).

[17] *RCNN, fast RCNN, and faster RCNN algorithms for object detection explained*. June 2022. URL: http://www.sefidian.com/2020/01/13/rcnn-fast-rcnn-and-faster-rcnn-for-object-detection-explained/ (page 32).

[18] Shaoqing Ren et al. "Faster r-cnn: Towards real-time object detection with region proposal networks". In: *Advances in neural information processing systems* 28 (2015) (pages 36–39).

[19] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2014. DOI: 10.48550/ARXIV.1409.1556. URL: https://arxiv.org/abs/1409.1556 (page 30).

[20] T Sugata and C Yang. "Leaf App: Leaf recognition with deep convolutional neural networks". In: *IOP Conference Series: Materials Science and Engineering* 273 (Nov. 2017), p. 012004. DOI: 10.1088/1757-899X/273/1/012004 (page 30).

[21] Michael Taylor. *Neural Networks: A visual introduction for beginners*. Blue Windmill Media, 2017 (pages 6, 8, 13).

[22] A. Tzotsos and D. Argialas. "Support Vector Machine Classification for Object-Based Image Analysis". In: *Object-Based Image Analysis: Spatial Concepts for Knowledge-Driven Remote Sensing Applications*. Ed. by Thomas Blaschke, Stefan Lang, and Geoffrey J. Hay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 663–677. ISBN: 978-3-540-77058-9. DOI: 10.1007/978-3-540-77058-9_36. URL: https://doi.org/10.1007/978-3-540-77058-9_36 (page 28).

[23] J. R. Uijlings et al. "Selective search for object recognition". In: *International Journal of Computer Vision* 104.2 (2013), pp. 154–171. DOI: 10.1007/s11263-013-0620-5 (pages 25, 34).

[24] *Understanding region of interest (roi pooling)*. URL: https://erdem.pl/2020/02/understanding-region-of-interest-ro-i-pooling (pages 34–35).

[25] Matthew D Zeiler and Rob Fergus. "Visualizing and understanding convolutional networks". In: *European conference on computer vision*. Springer. 2014, pp. 818–833 (page 19).

[26] Wei Zhang et al. "Deconv R-CNN for small object detection on remote sensing images". In: *IGARSS 2018-2018 IEEE International Geoscience and Remote Sensing Symposium*. IEEE. 2018, pp. 2483–2486 (page 30).