Warsaw University of Technology

FACULTY OF
POWER AND AERONAUTICAL ENGINEERING

Institute of Aeronautics and Applied Mechanics

Master's diploma thesis

in the field of study Robotics and Automatic Control
and specialisation Robotics

Recognition of objects in the environments of autonomous vehicles in real-time

Iskel Fikiru Hordofa
student record book number 323715

thesis supervisor
Prof. Rzymkowski Cezary

WARSAW 2024

Recognition of objects in the environments of autonomous vehicles in real-time

Abstract. This thesis endeavors to investigate a deep learning-based obstacle avoidance system designed for autonomous vehicles, including mobile robots, through the utilization of Neural Architecture Search (NAS). In navigating complex environments, autonomous vehicles rely on sophisticated computer vision methodologies to promptly identify obstacles. The study provides a comprehensive evaluation of the efficacy of YOLO-NAS and its variants, underscoring their optimized performance in striking a desirable balance between precision and processing speed when employing identical hyperparameters. This effectiveness surpasses that of other manually crafted models in terms of operational efficiency. Moreover, the thesis accentuates the versatility of YOLO-NAS across various tasks and hardware configurations, positioning it as a valuable asset across a broad spectrum of applications. Through strategic pre-training, YOLO-NAS is endowed with a profound understanding, augmenting its proficiency in real-time object detection tasks.

Keywords: Object detection, Neural Network, Deep Learning, YOLO, YOLO-NAS

Rozpoznawanie obiektów w otoczeniu pojazdów autonomicznych w czasie rzeczywistym

Streszczenie. Streszczenie. Celem pracy jest analiza systemu unikania przeszkód przez pojazdy autonomiczne, w tym roboty mobilne, opartego na metodach głębokiego uczeniu się złożonych algorytmów sztucznych sieci neuronowych. Wykorzystanie zaawansowanych technik wizji komputerowego w wyposażeniu pojazdów autonomicznych, zapewnia zdolność postrzegania i analizy otoczenia, wykrywania przeszkód w czasie rzeczywistym i bezpiecznego poruszania się w złożonym otoczeniu. Praca przedstawia ocenę wydajności modelu sieci YOLO-NAS i jego wariantów, podkreślając jego wysoką wydajność w zakresie osiągania optymalnej równowagi pomiędzy dokładnością i szybkością przy użyciu tych samych hiperparametrów, w czym przewyższa inne modele zaprojektowane przez człowieka. Przeprowadzona analiza dodatkowo podkreśla możliwość dostosowania modelu YOLO-NAS do różnorodnych zadań i sprzętu, pozycjonując go jako cenne narzędzie o szerokim spektrum zastosowań. Strategia wstępnego treningu YOLO-NAS wyposaża ten model w „umiejętność" wszechstronnego zrozumienie rozwiązywanego zadania, zwiększając jego możliwości w zastosowaniach do wykrywania obiektów w czasie rzeczywistym.

Słowa kluczowe: Rozpoznawanie obiektów, sztuczne sieci neuronowe, głębokie uczenie maszynowe, YOLO, YOLO-NAS

załącznik nr 10 do zarządzenia
nr 46 /2016 Rektora PW

Politechnika Warszawska
Warsaw University of Technology

…….....
miejscowość i data
place and date

……..
imię i nazwisko studenta
name and surname of the student
……..
numer albumu
student record book number
…….…….
kierunek studiów
field of study

OŚWIADCZENIE

DECLARATION

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.
Under the penalty of perjury, I hereby certify that I wrote my diploma thesis on my own, under the guidance of the thesis supervisor.

Jednocześnie oświadczam, że:
I also declare that:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- *this diploma thesis does not constitute infringement of copyright following the act of 4 February 1994 on copyright and related rights (Journal of Acts of 2006 no. 90, item 631 with further amendments) or personal rights protected under the civil law,*
- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- *the diploma thesis does not contain data or information acquired in an illegal way,*
- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
- *the diploma thesis has never been the basis of any other official proceedings leading to the award of diplomas or professional degrees,*
- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- *all information included in the diploma thesis, derived from printed and electronic sources, has been documented with relevant references in the literature section,*
- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.
- *I am aware of the regulations at Warsaw University of Technology on management of copyright and related rights, industrial property rights and commercialisation.*

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płycie kompaktowej) oraz treść pracy dyplomowej w module APD systemu USOS są identyczne.
I certify that the content of the printed version of the diploma thesis, the content of the electronic version of the diploma thesis (on a CD) and the content of the diploma thesis in the Archive of Diploma Theses (APD module) of the USOS system are identical.

.....
czytelny podpis studenta
legible signature of the student

Contents

1. Introduction

The field of mobile robots is expanding quickly and has the potential to completely transform a variety of industries. Mobile robots are becoming increasingly common in our daily lives as the need for automation rises. These robots are capable of assisting from healthcare to industrial automation. The advent of autonomous vehicles has revolutionized the transportation industry, promising safer and more efficient travel. However, the realization of this vision hinges on the development of advanced technologies capable of enabling these vehicles to navigate their environment safely and efficiently. One such technology is real-time object recognition, which allows autonomous vehicles to identify and classify objects in their path.

Autonomous vehicles rely heavily on sensors and computer vision algorithms to perceive their surroundings. These systems must be able to process large amounts of data in real time to make quick decisions, often within milliseconds. The challenge lies in developing algorithms that can accurately recognize objects while maintaining high computational efficiency. Obstacle avoidance, which is essential for autonomous navigation, is one of the major problems in mobile robots. In the realm of mobile robotics, obstacle avoidance stands as a fundamental challenge, empowering robots to traverse cluttered environments with both safety and efficiency [1].
Traditional obstacle avoidance methods, such as sonar, laser scanning, and stereo vision, have served as the cornerstone of these systems. However, these methods have inherent limitations, particularly in terms of their ability to handle dynamic and unpredictable scenarios.

Recently, the integration of deep learning, a subset of machine learning, has brought about a paradigm shift in the field of autonomous vehicles. Deep learning algorithms, particularly Convolutional Neural Networks (CNNs), have proven to be highly effective in tasks such as image and speech recognition, natural language processing, and beyond. In the context of autonomous vehicles, deep learning has been instrumental in enhancing the perception of surroundings, driver behavior monitoring, path planning, sensor fusion, and intelligent control of vehicle dynamics.

Deep learning-based control algorithms are especially crucial for achieving full autonomy. They can manage complex lateral and longitudinal maneuverings and can effectively calculate steering commands for lateral control and acceleration and braking commands for longitudinal speed control of vehicle 1. However, the safety of deep neural networks can be unstable under adverse conditions, necessitating the use of varied datasets for training the automated control system.

This paper is focused on the application of deep learning algorithm called YOLO-NAS in real-time object recognition for autonomous vehicles.

2. Problem Statement

Developing real-time object detection systems for obstacle avoidance represents a nuanced and intricate task, necessitating meticulous consideration. The efficacy of such systems is contingent upon several pivotal factors, encompassing the quality of visual data, the intricacies of the underlying algorithms, and the overall architecture of the obstacle avoidance system. A thorough exploration of each of these components is imperative to optimize the system for superior performance.

In addressing this inquiry, a comprehensive examination of the influencing factors is undertaken, seeking avenues for their refinement and enhancement. The goal is to engineer an obstacle-avoidance system of the highest quality. Furthermore, it is essential to contemplate the broader implications of the obstacle avoidance solution, assessing how it contributes to heightened safety and enhanced mobility across diverse contexts. By prioritizing these fundamental aspects, a profound understanding of the challenges inherent in real-time object detection systems can be cultivated. This nuanced understanding serves as a foundation for the development of innovative solutions, leveraging advanced computer vision techniques to surmount these obstacles.

3. General Objective:

The general objective of this study is to develop a deep learning-based obstacle avoidance system for mobile robots using Neural Architecture Search(NAS) in Python. The aim is to enable mobile robots to perceive their environment, detect obstacles in real time, and navigate safely through complex surroundings in real time.

3.1 Specific Objectives:

- Collect and curate a dataset of labeled images for training the obstacle detection neural network.
- Preprocess the dataset by applying techniques such as image augmentation and normalization to improve the network's performance.
- Fine-tune and implement a neural network architecture suitable for obstacle detection and avoidance.
- Integrate the trained neural network into the mobile robot's control system for real-time obstacle detection and avoidance
- Evaluate the performance of the obstacle avoidance system using quantitative metrics such as accuracy, precision, recall, and F1 score.
- Compare the neural network-based approach with traditional methods of obstacle avoidance in terms of effectiveness, efficiency, and adaptability.
- Assess the robustness and generalization capabilities of the trained neural network by testing it in various environments and with different types of obstacles.
- Identify potential areas for further improvement and research in computer vision-based obstacle avoidance, such as multi-sensor fusion or incorporating depth information.
- Provide practical insights and recommendations for implementing computer vision-based obstacle avoidance systems for mobile robots using neural networks in Python.

4. General Theory

4.1 Image processing

Image processing is a method of performing operations on an image to extract information or enhance its features. It is a broad field that covers a range of techniques, including image acquisition, Pre-processing, segmentation, feature extraction, image analysis, and visualization. In general, image processing involves transforming an image into a more useful form for analysis or display. This can be achieved through a variety of techniques, such as filtering, edge detection, Thresholding, and morphological operations. Image processing has numerous applications in various fields, including medicine, remote sensing, surveillance, robotics, and entertainment. [2], [3]

4.2 Deep Learning and Neural Networks

Deep learning is another name for a multilayer artificial neural network or multilayer perceptron. The elementary bricks of deep learning are the neural networks, that are combined to form the deep neural networks.

Table 4.1. Comparing Neural Networks and Deep Learning

Feature	Neural Networks	Deep Learning
Structure	Single-layer or shallow neural networks (typically two to three layers)	Deep neural networks with multiple layers (typically more than three)
Learning Ability	Can learn simple patterns and make predictions	Can learn complex patterns, including abstract relationships, and make more accurate predictions
Data Requirements	Requires less training data	Requires more training data to learn complex patterns patterns
Computational Cost	Less computationally expensive	More computationally expensive, especially for deep neural networks with many layers
Applications	Image classification, fraud detection, spam filtering	Image recognition, natural language processing, speech recognition, medical diagnosis, self-driving cars

The term "deep" in deep learning refers to the use of multiple layers in the neural network. Deep learning models, often called deep neural networks (DNNs), are capable of learning and representing complex patterns and hierarchical features from data. [4] [5], [6]

Various types of deep learning systems exist, distinguished by their neural network architectures and operational principles. Examples include feed-forward neural networks, convolutional networks, recurrent neural networks, autoencoders, and deep beliefs. These methodologies have facilitated substantial advancements in sound and image processing domains, encompassing tasks such as facial recognition, speech recognition, computer vision, automated language processing, and text classification (e.g., spam detection). The

potential applications of these techniques are diverse. An illustrative instance is the AlphaGo program, which achieved world champion status in the game of Go in 2016 by leveraging deep learning methodologies.[7], [8], [9]

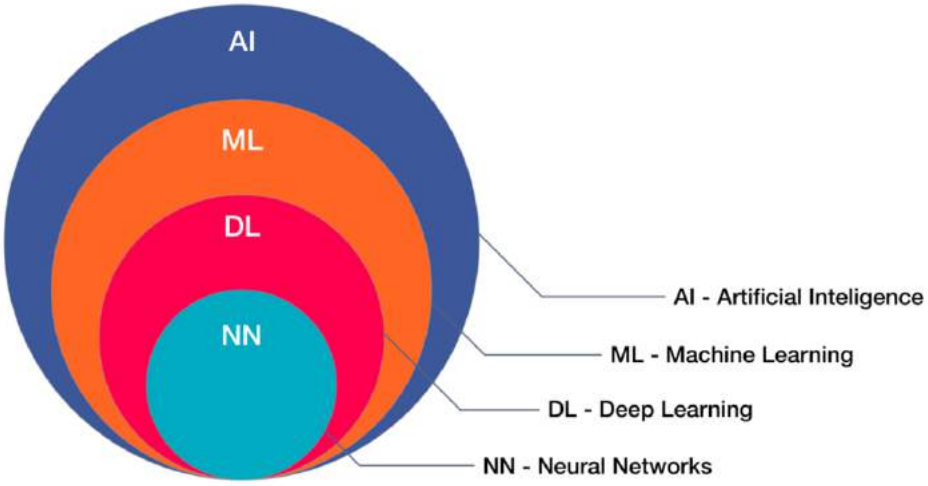


Figure 4.1. Neural network core of deep learning **an**

4.3 Neural Networks

Neural networks are the fundamental components of deep learning. They are composed of layers of interconnected nodes or neurons that process information. Artificial neural networks (ANNs) are computing systems that are designed to work in a similar way to the human brain.

4.3.1 The Biological Inspiration

The complexity of the brain has made it challenging for scientists to fully understand it, despite extensive research. Engineers have modified neural models to create a more useful and less biological approach, while still maintaining much of the original terminology.

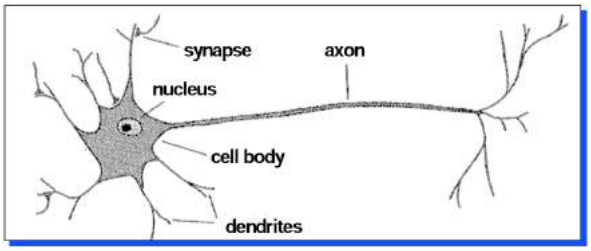


Figure 4.2. The structure of Neurons [4]

A neuron is composed of three main parts - the cell body, the dendrite (which branches out to receive input), and the axon (which branches out to send output). Connections between axons and dendrites are made via synapses. Electro-chemical signals are carried from the dendrites, through the cell body, and down the axon to other neurons.

4.3.2 Perceptron

A single neuron of a neural network is called a perceptron. A perceptron implements a mathematical function that operates on the input signals and generates outputs. Figure is an example of a perceptron. A perceptron is the simplest neural network.

4.3.3 The learning process

A perceptron model starts by multiplying all the input values and their weights and then sums these values to create a weighted sum. This weighted sum is then applied to the activation function "f" to obtain the desired result. This activation function is also known as a step function and is represented by "f". This step function or activation function plays an essential role in ensuring that the output signal is mapped between the required values (0,1) or (-1,1). Please note that the weight of the input data indicates the strength of the node. similarly, the value of the input bias allows the activation function curve to shift up or down. The input signals to a neuron come either from a source (a camera or sensing device) or from the outputs of other neurons.

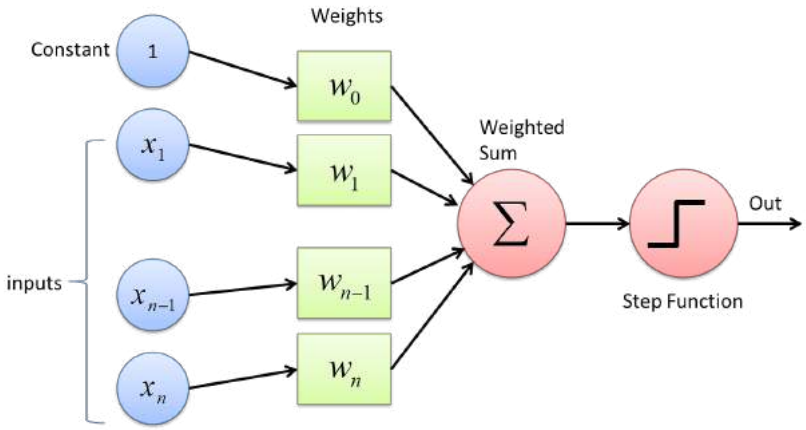


Figure 4.3. Perceptron

Mathematically, the output (y) of a neuron in a neural network perceptron can be expressed as:

$y = f(\sum_{i=1}^{n}(\omega_i.x_i) + b)$
where:
ω_i is the weight associated with the input x_i
b is the bias term,
f : activation
As shown in fig 4.3, One percepteron has the following components:

- **Input Nodes or Input Layer:** The input layer, also called input nodes, is the first layer of a neural network. It is responsible for receiving input data, which can be any numerical values, images or text. The input layer does not perform any calculations or transformations of the input data, but only forwards them to subsequent layers in the network. [10], [4]

- **Weight:** Weights are parameters that adjust the strength of the connections between neurons (nodes) in adjacent layers of a neural network. Each connection has an associated weight, and these weights are learnable and updated during the training process. The weights determine the impact of the input signals on the neurons in the next layer. A higher weight means that the corresponding input has a stronger influence on the neuron's output. During training, the neural network adjusts these weights to minimize the difference between the predicted output and the actual target output.

- **Bias:** Biases are additional parameters in a neural network that allow for fine-tuning the output of each neuron. Unlike weights, biases are not associated with specific inputs but are added to the weighted sum of inputs in each neuron. Role: Biases provide the neural network with flexibility, enabling it to account for situations where all inputs are zero or have low values. They allow neurons to activate even when the weighted sum of inputs is not sufficient to trigger an output.Like weights, biases are adjusted during training to improve the overall performance of the network. [10], [4]

- **Activation Function:** The primary purpose of an activation function is to determine whether a neuron should be activated (output a signal) or not, based on the input it receives. An activation function is a mathematical operation enacted on the output of a neuron (or node) within the architecture of a neural network. It introduces non-linearity to the network, allowing it to learn complex patterns and make more sophisticated decisions. The activation function takes the weighted sum of inputs and a bias term and produces the output of the neuron.[4]

4.3.4 Types of Perceptron:

Based on the layers, Perceptron models are divided into two types.

1. **Single-layer perceptron model:** This is one of the simplest types of artificial neural networks (ANN). The single-layer perceptron model consists of a feedforward network and also includes a threshold transfer function inside the model. The main goal of the single-layer perceptron model is to analyze linearly separated objects with binary outputs.
   In a single-layer perceptron model, its algorithms do not contain recorded data, so it starts with a non-constantly allocated input for weight parameters. Furthermore, it

sums all inputs (weight). If, after adding all inputs, the sum of all inputs is greater than the set value, the model will be activated and display the output value as +1. If the result is the same as the previously established threshold value, then the performance of this model is considered satisfactory and the weight requirements do not change. However, this model contains several discrepancies that arise when many input weight values are introduced into the model. Therefore, some changes should be necessary when entering the weights to find the desired result and minimize errors. A single-layer perceptron learns patterns that can be solved linearly. [10], [11]

2. **Multi-layer Perceptron** An artificial neural network, similar to the human brain, comprises several neurons, also known as perceptrons. A cluster of neurons processes the inputs. Every individual neuron within the group separately processes the inputs. The outputs from this cluster of neurons are transmitted to another individual neuron or cluster of neurons for subsequent processing. These neurons can be visualized as organized in layers, where the output of one layer serves as the input for the following layer. There is no limit to the number of layers you can use to train your neural network. The arrangement of neurons in a neural network, where multiple layers are used, is referred to as a multilayer perceptron (MLP). A multi-layer perceptron model shares the same structure as a single-layer perceptron model, but it includes a higher number of hidden layers.

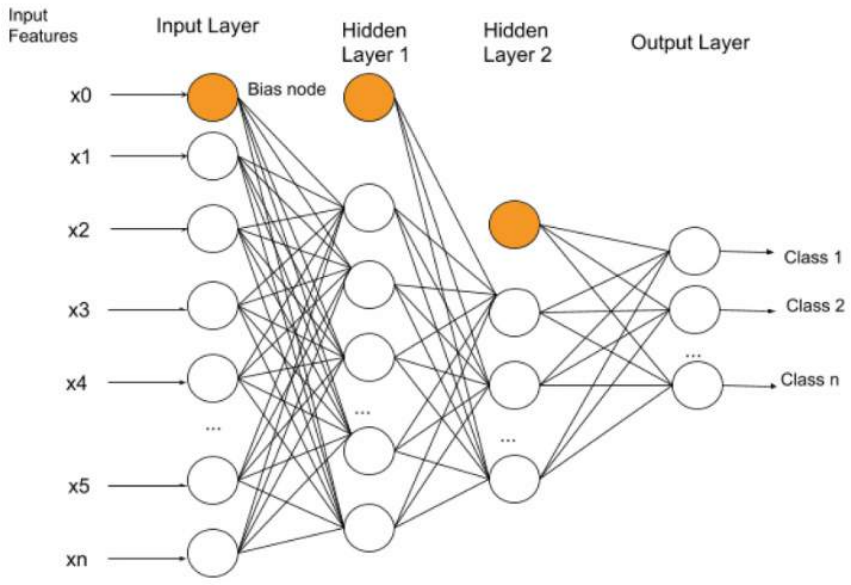


Figure 4.4. Multi layer perceptron [10]

also known as the backpropagation algorithm, which executes in two stages as follows:

- **Forward Stage:** Activation functions start from the input layer in the forward stage and terminate at the output layer.
- **Backward Stage:** In the backward stage, weight and bias values are modified as per the model's requirements. In this stage, the error between the actual output and demand originates at the output layer and propagates backward to the input layer.

A multi-layer perceptron model is a type of artificial neural network that has greater processing power and is capable of processing both linear and non-linear patterns. It is also capable of implementing logic gates such as AND, OR, XOR, NAND, NOT, XNOR, and NOR.
This model has multiple artificial neural networks with several layers, where the activation function is not linear as in a single-layer perceptron model. Instead, it can be executed with various activation functions such as sigmoid, TanH, ReLU, etc., for deployment. [10], [4]

4.4 Deep Learning

Deep learning is a type of artificial neural network or multilayer perceptron. It is a subset of machine learning that focuses on using artificial neural networks and representational learning. The term "deep" in deep learning refers to the use of multiple layers in the network. These methods can be supervised, semi-supervised, or unsupervised. [12]

Various deep-learning architectures, such as deep neural networks, deep belief networks, deep reinforcement learning, recurrent neural networks, convolutional neural networks, and transformers, have been successfully applied to several fields, including computer vision, voice recognition, signal processing, natural language processing (NLP), machine translation, bio-informatics, drug or medical purposes, climate forecasting, material inspection, and board game programs. These architectures have produced remarkable results that are comparable to, and in some cases, surpassing human expert performance. [13]

4.4.1 Deep Learning or Multilayer Perceptron Architecture

A multilayer perceptron consists of at least three types of layers: the input layer, the hidden layers, and the output layer. You can have more than one hidden layer. Each layer contains one or more neurons. A neuron performs some computation on the inputs it gets and generates outputs. The output from the neurons is sent as input to the next layer, except in the case of the output layer.

1. **Input layer:** This is the primary component of Perceptron, which accepts the initial data into the system for further processing. Every input node in the system holds a numerical value that is real and can be expressed as a number. In a neural network, the input layer receives raw input data like image, text, etc. It then passes it on to the next layer. Each node in the input layer represents a feature or an attribute of the input data. These nodes act as receptors for the raw input information and are connected to the nodes in the next layer, which is typically a hidden layer.
   Here are some key points about the input layer:
   a) Nodes/Neurons: The nodes in the input layer are also called input neurons. Each neuron corresponds to a specific feature or dimension of the input data.
   b) Input Features: If you're working with a dataset of, for example, images, each node

in the input layer might represent a pixel's intensity or color values. In the case of text data, each node could represent a unique word or a character.

c) No Processing: The nodes in the input layer do not perform any processing on the input data. They simply pass the raw input values to the next layer. The actual computation and learning occur in the subsequent layers, particularly in the hidden layers.

d) Connections: Each node in the input layer is connected to every node in the next layer (usually a hidden layer) through weights. These weights determine the strength of the connection between nodes and play a crucial role in the learning process of the neural network.

The input layer of a neural network consists of neurons that are equal in number to the features of the data. Along with these neurons, additional nodes called bias nodes can also be included in each layer. The primary purpose of the bias node is to provide control over the output of the layer. Although not strictly necessary in deep learning, it is still a common practice to add a bias node.

Total Number of Neurons in the input layer = The number of input features without a bias = (The number of input features + 1) with a bias [10]

2. **Hidden Layers:** Hidden layers in a neural network refer to the layers that come between the input layer and the output layer. They are called "hidden" because their outputs are not directly observable or part of the final network output during the training phase. At least one hidden layer is necessary for a neural network to function as this is where the learning occurs. The neurons in this layer perform the computations required for learning. Generally, for most cases, a single hidden layer is sufficient for learning. However, as required to model real-world situations, the number of hidden layers can be increased. As the number of hidden layers increases, the computation complexity of the network also increases, which leads to an increase in computation time.

   Total Number of Neurons in the hidden layer = A common practice is to take two-thirds (or 66 percent) of the number of neurons in the previous layer.

   For example, if the number of neurons in the input layer is 100, the number of neurons in the first hidden layer will be 66 and in the next hidden layer will be 43, and so on. Again, there is no magic number, and you should tune the neuron counts based on the model accuracy.

3. **Output Layer:** The output layer is the final layer in a neural network, responsible for producing the model's predictions or outputs based on the learned representations from the preceding layers. The number of neurons in the output layer depends on the problem type that the neural network is supposed to solve.

   Nodes in the Output Layer can be used for:

   - **For binary classification:** there is typically one node with a sigmoid activation

function, producing a probability output.

- **For multi-class classification:** when the network has to predict one of many classes, the output layer has as many neurons as the number of all possible classes.

- **For regression tasks:** there is usually a single node with a linear activation function, producing a continuous output/ when the network has to predict a continuous value, such as the closing price of stocks, the output node has only one neuron. [14], [4]

4.4.2 Forward propagation

A feedforward neural network is a type of artificial neural network comprising neurons interconnected in a manner that avoids cyclic connections. This architecture represents the most basic form of the neural network, where data propagation occurs unidirectionally, from the input layer through the hidden layer(s), and ultimately to the output layer. Unlike its counterparts, this network lacks any loopback or feedback mechanism, enabling a linear flow of information through its layers. [15] [10]

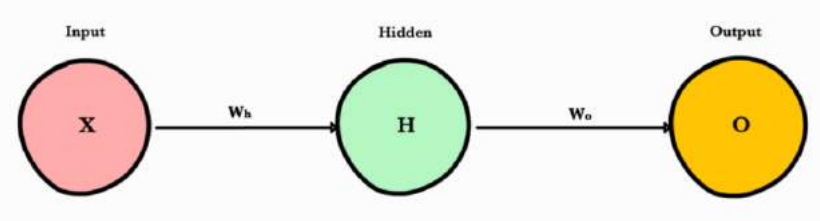


Figure 4.5. Forward Propagation [16]

a single pass of forward propagation translates mathematically to:

$$Prediction = A(A(XW_h)W_o)$$

Where A: is an activation function like ReLU, X is the input and W_h and W_o are weights.

Steps

- Calculate the weighted input to the hidden layer by multiplying X by the hidden weight W_h.
- Apply the activation function and pass the result to the final layer
- Repeat step 2 except this time X is replaced by the hidden layer's output, H [10], [4]

4.4.3 BackPropagation:

The main objective of back propagation is to modify or adjust the weights in the neural network in a way that corresponds to their contribution to the overall error. By consistently reducing the error associated with each weight, we can eventually obtain a set of weights that generate accurate predictions.
Here are the final 3 equations that together form the foundation of backpropagation. [17] [4]

$$OutputLayerErrorE_o = (O - y).R'(Z_o)$$

$$HiddenLayerErrorE_h = E_o.W_o.R'(Z_h)$$

$$Cost - WeightsDeriv \quad LayerError.LayerInputs$$

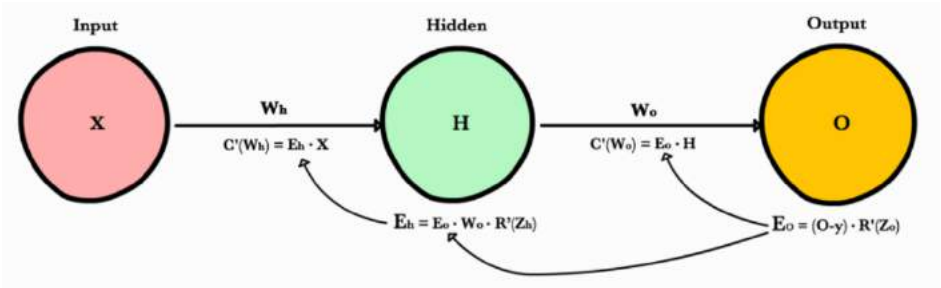


Figure 4.6. Backward Propagation [16]

4.4.4 Activation function

The activation function plays a crucial role in determining whether a neuron should be activated (turned on or off) based on whether its input is relevant for model prediction. This function normalizes the output of each neuron to a range between 0 and 1 or between -1 and 1. Different mathematical functions are utilized as activation functions for various purposes.[18], [19], [20]

we can broadly divide the activation function into two:

- **Linear Activation Function:** Commonly used in the output layer for regression tasks where a continuous range of values is desired. A linear activation function computes a weighted sum of its input without introducing non-linearity, and The output is a linear transformation of the input.

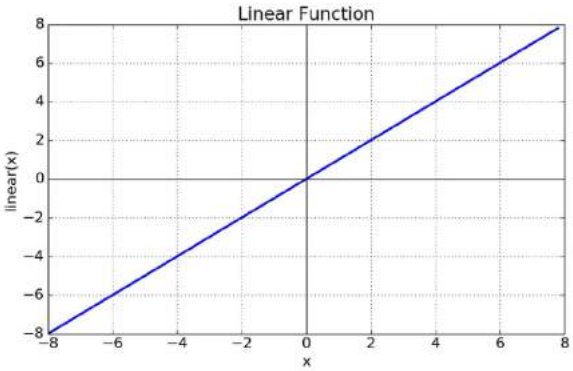


Figure 4.7. Linear Activation Function

Equation: $f(x) = cx$
where c is constant.

The output of the linear activation function varies from $-\infty$ to $+\infty$, as shown in Figure above. If you choose to use a linear activation function, the last layer of your neural network will simply be a linear function of the first layer, regardless of how many layers the network has. This means that your network can only learn linear dependencies between the input and output, which is insufficient for solving complex problems like computer vision. Therefore, using a linear activation function is not recommended for such problems. [20]

- **Non-linear Activation Function:** A non-linear activation function introduces non-linearity into the network, enabling it to learn complex relationships and representations. It facilitates the model's ability to generalize and differentiate outputs with diverse data.

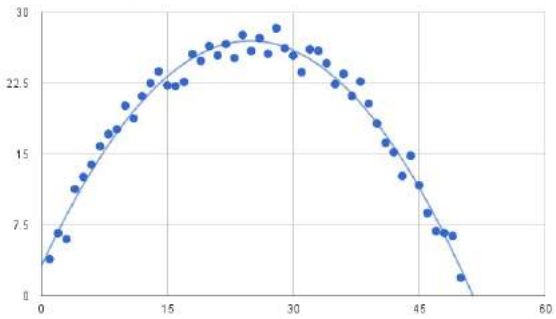


Figure 4.8. Non-Linear Activation Function

Essential for training deep neural networks as it allows the network to capture complex patterns and relationships in the data, They allow the network to capture intricate relationships between features, which is crucial for tasks like image recognition, natural language processing, and more.

21

1. **Sigmoid or Logistic Activation Function:** The sigmoid activation function calculates the neuron output using the sigmoid function, as shown here:

$$\phi(z) = \frac{1}{1+e^{-z}}$$

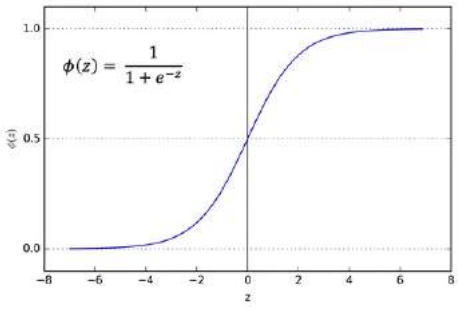


Figure 4.9. Sigmoid Activation Functioin

where z is calculated like

$$z = X_0 + \sum_{i=0}^{i=n} w_i x_i$$

The sigmoid function is a mathematical operation that always yields a value between 0 and 1. This makes the output spontaneous, without many overrides as the input value fluctuates. Additionally, the sigmoid function is very useful as it does not generate a constant value from the first-order derivatives because it is a non-linear function.

The sigmoid function is a mathematical function that always produces an output value between 0 and 1. This characteristic results in a smooth output without many jumps as the input value fluctuates. Another benefit of the sigmoid function is that it is nonlinear, and its first-order derivative does not generate a constant value.

2. **Tanh or hyperbolic tangent Activation Function**
   The Tanh function is a type of activation function, commonly used in neural networks. Its behavior is similar to the sigmoid activation function, but with some notable differences. The Tanh function is zero-centered, and The range of the tanh function is from (-1 to 1).

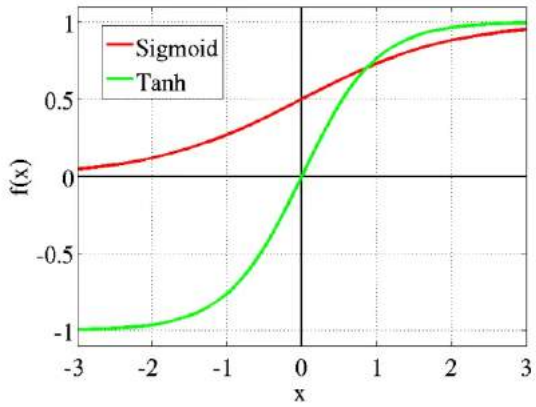


Figure 4.10. Tangent Activation Functioin

The TanH activation function calculates the neuron output using:

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

The TanH is useful when it comes to mapping the values; the negative inputs will be mapped strongly to negative, and the same for positive inputs too, and the zero inputs will be mapped near zero.

3. **ReLU (Rectified Linear Unit) Activation Function** The Rectified Linear Unit (ReLU) is a widely used activation function in deep learning, particularly for computer vision tasks. ReLU is preferred for image processing because it can handle non-negative values, which is a common characteristic of image pixels. One of the key advantages of ReLU is its computational efficiency, which makes it suitable for large-scale models. Furthermore, ReLU is a nonlinear function with a derivative, allowing for backpropagation and weight adjustment during training. This property is essential for the neural network to learn from the data and improve its performance over time.

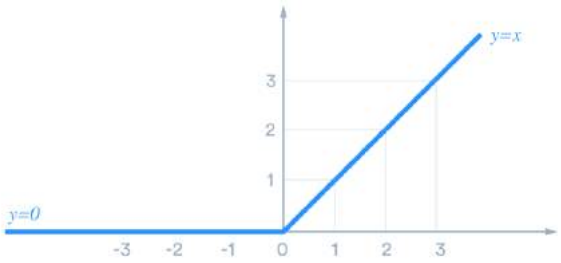


Figure 4.11. ReLU Activation Functioin

$$f(x) = max(0, x)$$

it ranges from 0 to $+\infty$

23

4. **Leaky ReLU Activation Function** The leaky rectified linear unit (ReLU) is a widely used activation function in deep learning models. The function introduces a small negative slope in the negative region, which allows for backpropagation for negative inputs, thereby avoiding the vanishing gradient problem. However, a drawback of this function is that it produces inconsistent outputs for negative values. This is because the negative slope of the activation function results in a non-zero output for negative inputs, which is not consistent with the expected behavior of an activation function. Nonetheless, the leaky ReLU remains a popular choice for deep neural networks due to its ability to improve the training of deep models.

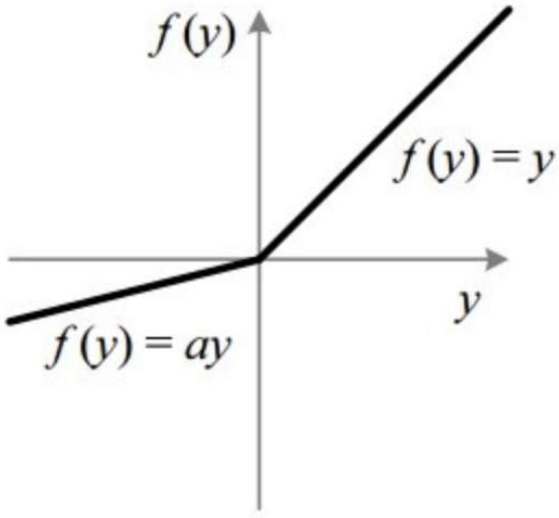


Figure 4.12. Leaky ReLU Activation Functioin

$$f(y) = \begin{cases} \alpha y & \text{for} \quad y < 0 \\ y & \text{for} \quad y \geq 0 \end{cases}$$

5. **SELU Actiovation Function** A scaled exponential linear unit (SELU) computes neuron outputs using the following equation:

$$f(\alpha, x) = \lambda \begin{cases} \alpha(e^x - 1) & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$$

where the value of λ = 1.05070098 and the value of α = 1.67326324. These values are fixed and do not change during backpropagation.[Orielly]

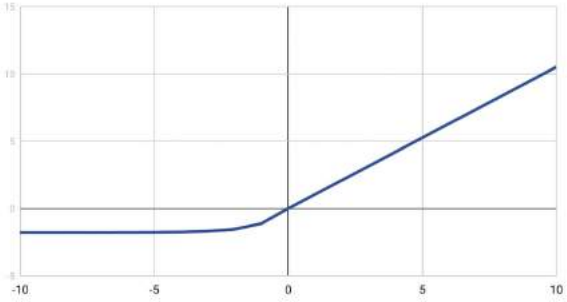


Figure 4.13. SELU Activation Functioin

SELU is an activation function that has the property of self-normalization. This means that with SELU, the entire network is self-normalizing, which makes it efficient in terms of computation and helps it converge faster. Additionally, SELU overcomes the issues of exploding or vanishing gradients that occur when the input features are too high or too low.

6. **Softplus Activation Function** The softplus activation function applies smoothing to the activation function value z . It uses the log of exponent as follows:

$$f(x) = lon(1 + e^z)$$

Softplus is also called the SmoothReLU function. The first derivation of the softplus function is $\frac{1}{1+e^z}$, which is the same as the sigmoid activation function.

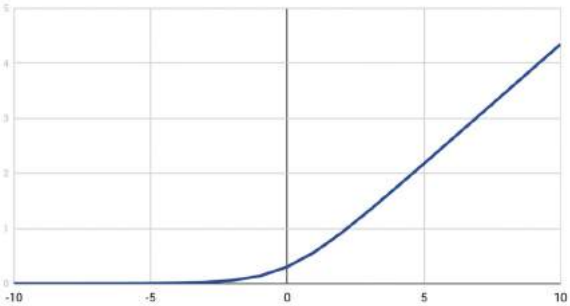


Figure 4.14. Softplus Activation Function

7. **Softmax** The Softmax function operates on a vector of real numbers by normalizing it to create a probability distribution that generates outputs between 0 and 1. The resulting probabilities have the property that the sum of all output values is equal to 1. This function is typically utilized as the activation function for the output layer of a classification neural network. The interpreted output values represent prediction probabilities for each class.

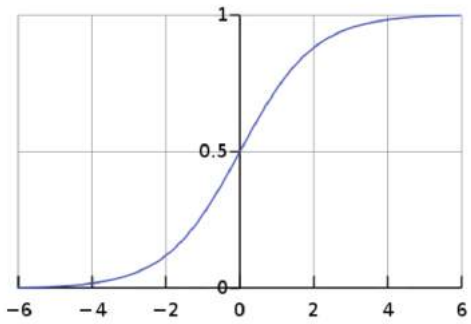


Figure 4.15. Softmax Activation Function

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

for i=1,...., k and z=(z_1...., z_k) ϵR^k

4.4.5 Loss Function/Error Function

A loss function is a function that measures how well a neural network models the training data by comparing the target and predicted output values. During training, the goal is to minimize the loss between the predicted and target outputs. The equation of error may be written in a simplified form as follows:

$$Error = Expectedoutcome - Predictedoutcome$$

When a neural network begins the learning process, it initializes weights and calculates output from each neuron using an activation function. It then computes the error, adjusts the weights, recalculates outputs, and re-evaluates errors, until it reaches the minimum error. The weights that give the minimum errors are considered the final weights and the network is considered "learned" at this stage.

In calculus, if the first derivative of a function is zero, then the function at that point is either a minimum or a maximum. The neural network training process aims to find the minimum point where the first derivative is zero. To achieve this, a neural network must have an **error function** that calculates the first derivative and identifies the points (weights and biases) where the error function is minimum. The type of error function used depends on the type of model being trained. These error functions are also called loss functions or simply losses.

The error functions are broadly divided into the following three categories: [10], [21]

- **Regression loss functions** are used when we want to train models to predict continuous value outcomes, typically a numerical value. To measure the difference between

the predicted and actual target values, different loss functions are employed. The following are some widely used regression loss functions:

— **Mean Squared Error (MSE) Loss:** This is the default error function for regression problems. This is the preferred loss function if the distribution of the target variable is normal or Gaussian. This function has numerous properties that make it especially suited for calculating loss. The difference is squared, which means it does not matter whether the predicted value is above or below the target value; however, values with a large error are penalized.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2$$

— **Mean Absolute Error (MAE) Loss:** MAE finds the average of the absolute differences between the target and the predicted outputs.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y^{(i)} - \hat{y}^{(i)}|$$

In some cases, this loss function serves as an alternative to MSE. As mentioned earlier, MSE is highly sensitive to outliers, which can significantly affect the loss due to the squared distance. To mitigate this, MAE is used when the training data has a substantial number of outliers.

— **The Mean Squared Logarithmic Error (MSLE)** The Mean Squared Logarithmic Error (MSLE) is a loss function that is commonly used in regression tasks. It is particularly useful when the target values span multiple orders of magnitude. MSLE measures the mean squared difference between the natural logarithm of the predicted values and the natural logarithm of the true values. This can be especially helpful when there is a wide variation in the scale of the target values. The formula for MSLE is as follows:

$$MSLE(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(log(1 + y^{(i))} - log(1 + \hat{y}^{(i)}))^2$$

where:
y^i : is the true target value for the i-th sample.
$\hat{y}^{(i)}$: is the predicted target value for the i-th sample.

— **Huber Loss:** The Huber loss, also referred to as the smooth L1 loss, is a commonly used loss function in regression tasks. It is designed to be less sensitive to outliers compared to the Mean Squared Error (MSE) loss function while retaining the benefits of a quadratic loss function for smaller errors.

$$Huber(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n} L_\delta(y^{(i))} - \hat{y}^{(i)})$$

where:

y_i : is the true target value for the i-th sample.
y^i : is the predicted target value for the i-th sample.
n : is the total number of samples.
$L_\delta(x)$ is defined as:

$$L_\delta(x) = \begin{cases} \frac{1}{2}x^2 & \text{for} \quad |x| \leq \delta \\ \delta(|x| - \frac{1}{2}\delta) & otherwise \end{cases}$$

Benefit: One big When training neural networks, using Mean Absolute Error (MAE) can cause a problem due to its large gradient, which can result in missing the minima at the end of training when using gradient descent. In contrast, as the loss gets closer to its minima, the gradient decreases when using Mean Squared Error (MSE), making it more accurate.
To address this issue, Huber loss can be very useful since it curves around the minima, decreasing the gradient. Additionally, Huber loss is more resistant to outliers than MSE. Therefore, it combines the desirable properties of both MSE and MAE.

— **Log-Cosh loss:** The Log-Cosh loss is a smooth and differentiable approximation of the Huber loss, often used in regression tasks. Similar to the Huber loss, it aims to be less sensitive to outliers than the Mean Squared Error (MSE) loss, but it has the advantage of being continuously differentiable.

$$Log - Cosh(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n} Log(cosh(y^{(i)} - \hat{y}^{(i)}))$$

where:
y_i : is the true target value for the i-th sample.
y^i : is the predicted target value for the i-th sample.
n : is the total number of samples.
$Cosh$: is the hyperbolic cosine function.
The optimization goal during training is to minimize the Log-Cosh loss, and the model adjusts its parameters to achieve predictions that result in smaller Log-Cosh loss values. The Log-Cosh loss is often considered when a balance between the robustness of the Huber loss and the differentiability of the MSE loss is desired.

— **Quantile loss function:** Quantile loss is a loss function used in quantile regression, where the goal is to predict not just a central tendency (like the mean in traditional regression) but rather different quantiles of the target distribution. It is particularly useful when you want to estimate a range of possible values for a prediction.

$$L_\gamma(y, y^p) = \sum_{i=y_i<y_i^p} (\gamma - 1).|y_i - y_i^p| + \sum_{i=y_i \geq y_i^p} (\gamma).|y_i - y_i^p|$$

The optimization goal during training is to minimize the Quantile Loss, and the model adjusts its parameters to achieve predictions that capture the desired

quantiles of the target distribution. Quantile regression is particularly useful in scenarios where understanding the variability or uncertainty in predictions is essential.

- **Binary classification loss functions** are used when we want to train models to predict a maximum of two classes (usually denoted as 0 and 1) and the true binary label, such as to compare and contrast different classes together.

  — **Binary Crossentropy Loss (Log Loss):** The Binary Crossentropy Loss (Log Loss) is the default loss function for binary classification problems and is considered better than other functions. It calculates a score that reflects the average difference between the actual and predicted probability distributions for predicting class 1. This score is minimized, and a perfect cross-entropy value is set to 0. The function can only be used when the target value is within the range of (0,1).

$$BinaryCrossentropy(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}[y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i)]$$

  It measures the cross-entropy between the true binary labels and the predicted probabilities. It penalizes deviations from the true labels by assigning higher penalties to confidently wrong predictions.

  — **Hinge Loss (SVM Loss):** This is used mainly in support of vector machine–based binary classification and can be used when the target variable is in the range (-1, 1).

$$Hinge - Loss(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n} max(0, 1 - y_i.\hat{y}_i)$$

  Penalizes misclassifications linearly and encourages correct predictions to have a margin of at least 1.

  — **Squared Hinge Loss:** The squared hinge loss function computes the squared value of the score hinge loss. This mathematical operation smooths the surface of the error function, rendering it numerically more tractable.

$$\text{Squared Hinge Loss}(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n} max(0, 1 - y_i.\hat{y}_i)^2$$

- **Multiclass classification loss functions** are used when our models need to predict more than two classes/are used to measure the difference between predicted class probabilities and true class labels in scenarios where there are more than two classes, such as object detection.

  — **Categorical Crossentropy Loss:** used for multiclass classification tasks with one-hot encoded true class labels. It is used to measure the cross-entropy be-

tween the true distribution and the predicted class probabilities.

$$CategoricalCrossentropy(y,\hat{y}_i) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m} y_{ij}\,log(\hat{y_{ij}})$$

— **Sparse Categorical Crossentropy Loss:** Similar to categorical crossentropy but used when true class labels are provided as integers rather than one-hot encoded vectors.

$$SparseCategoricalCrossentropy(y,\hat{y}) = -\frac{1}{n}\sum_{i=1}^{n} log(y_i[\hat{y}_i])$$

Sparse cross-entropy performs the same cross-entropy calculation of error without requiring that the target variable be one hot-encoded before training and is used When you have a large number of classes in the target, for example, predicting dictionary words.

— **Kullback-Leibler divergence (KLD) loss:** The Kullback-Leibler divergence (KLD) loss is a measure of the dissimilarity between one probability distribution and a reference baseline distribution. A value of 0 for the KL divergence loss indicates that both distributions are identical. This statistical metric quantifies the amount of information loss, expressed in bits, when the predicted probability distribution is employed to approximate the target probability distribution.
The KLD loss is a valuable tool for addressing complex problems, such as auto-encoders, which are utilized for learning high-dimensional representations. In the context of multiclass classification, KLD serves as a multiclass cross-entropy measure.

$$\text{KL Divergence}(P \parallel Q) = \sum_{i\in\chi} P(i)\log\left(\frac{P(i)}{Q(i)}\right)$$

where:
For discrete probability distributions, P and Q are defined on the same sample space, χ, the relative entropy from Q to P.

— **Cross-Entropy Loss (Sigmoid Crossentropy for Multilabel Classification):** Suitable for multilabel classification where each sample can belong to multiple classes. It measures the crossentropy for each class independently.

$$\text{Cross Entropy Loss}(y,\hat{y}) = -\frac{1}{n}\sum_{i=1}^{n}[y_i\,log(\sigma(\hat{y}_i)) + (1-y_i)\,log(1-\sigma(\hat{y}_i))]$$

4.4.6 Layers

In a neural network, layers are the building blocks that organize and structure the computation. Each layer contains a group of nodes, or neurons, that process information.

- **Convolutional Layer:** In convolutional neural networks (CNN), the convolution

layer performs a linear operation by multiplying the input with a weight (kernel or filter), and it plays a critical role in the network. The layer comprises two primary components:

— **Kernel (Filter):** A convolution layer can comprise more than one filter. The size of the filter should be smaller than the input dimension intentionally, as it allows the filter to be applied at different positions on the input. Filters are useful for identifying significant features in a given input. By applying more than one filter to the same input, different features can be extracted. The output from multiplying the filter with the input creates a dimensional array called the "feature map."

The Stride property controls the movement of the filter over the input. When the value is set to 1, the filter moves one column at a time over the input. When set to 2, the filter jumps two columns at a time as it moves over the input.

- **Dropout Layer:** A dropout layer is a type of layer commonly used in neural networks to prevent overfitting. It takes the output of the previous layer's activations and randomly sets a certain fraction, known as the dropout rate, of the activations to 0. This effectively cancels or "drops out" those activations, helping to prevent the network from becoming too reliant on any one feature or pattern. The dropout rate is a tunable hyperparameter that can be adjusted to measure performance with different values. Typically, it is set between 0.2 and 0.5, but it can be set arbitrarily depending on the specific application and dataset.
  Dropout is a technique used during training of neural networks. During training, some of the activations in a layer are randomly dropped or turned off. However, at test time, no activations are dropped, instead, they are scaled down by a factor of the dropout rate. This is to account for the fact that more units are active during test time than during training time. The idea behind dropout is to introduce some noise into the layer in order to disrupt any interdependent learning or coincidental patterns that may occur between units in the layer that are not significant. This helps to prevent overfitting and improve the generalization performance of the network.

- **Pooling layer:** Pooling layers often take convolution layers as input. A complicated dataset with many objects will require a large number of filters, each responsible for finding patterns in an image so the dimensionally of a convolutional layer can get large. It will cause an increase in parameters, which can lead to over-fitting. Pooling layers are a type of technique that is used to reduce the dimensionality of high-dimensional data. Similar to convolutional layers, pooling layers also have a kernel size and stride. The kernel size is typically smaller than the feature map, with a standard size of 2x2 and a stride of 2. There are two main types of pooling layers that are commonly used.
  The first type is the max pooling layer. The Max pooling layer will take a stack of feature maps (convolution layer) as input. The value of each node in the max pooling

layer is calculated by taking the maximum value of the pixels contained in a sliding window. This operation helps to reduce the size of the feature maps while preserving the most important information.
The other type of pooling layer is the:
Average Pooling layer. The average pooling layer calculates the average of pixels contained in the window. It's not used often but you may see this used in applications for which smoothing an image is preferable.

- **Fully-connected/Linear Layer:** A fully-connected layer, also called a linear layer, is a type of layer in a neural network where all the inputs from one layer are connected to every activation unit of the next layer. Usually, the last few layers in machine learning models are fully-connected ones, which output a class prediction based on the features learned in the previous layers.
To input a vector of nodes activated in the previous convolutional layers, the fully-connected layer passes it through one or more dense layers before sending it to the output layer.
An activation function is used to make a prediction before the vector reaches the output layer. While the convolutional and pooling layers tend to use a ReLU function, the fully-connected layer can use two activation functions depending on the classification problem:
- Sigmoid: is a mathematical function that is commonly used for binary classification problems. It is a logistic function that has a characteristic "S" shaped curve. - Softmax: A more generalized logistic activation function that ensures the values in the output layer sum up to 1. It is commonly used for multi-class classification.
The activation function outputs a vector with the same dimensions as the number of classes to be predicted. The output vector yields a probability between 1 and 0 for each class. [22], [23]

4.4.7 Optimizer

An optimizer is an algorithm or method used to adjust the parameters of the neural network (weights and biases) during the training process. The primary goal of an optimizer is to minimize the loss function, which measures the difference between the predicted output and the true target values.
During training, the neural network makes predictions, and the optimizer adjusts the model's parameters based on the error (loss) between these predictions and the actual target values. The optimization process involves finding the optimal set of parameters that minimize the loss, enabling the neural network to make accurate predictions on unseen data. [24], [10]
Some commonly used optimizers in neural networks include:

— **Adaptive gradient (Adagrad):** Adaptive gradient (Adagrad) is a technique that adjusts the learning rate according to a specific parameter.
  — This method ensures that parameters with higher gradients or frequent updates have a slower learning rate, so as not to overshoot the minimum value.
  — parameters with low gradients or infrequent updates have a faster learning rate, allowing them to be trained quickly.
  — It divides the learning rate by the sum of squares of all previous gradients of the parameter.
  — When the sum of the squared past gradients has a high value, it basically divides the learning rate by a high value, so the learning rate will become less.
  — Similarly, if the sum of the squared past gradients has a low value, it divides the learning rate by a lower value, so the learning rate value will become high.
  — This implies that the learning rate is inversely proportional to the sum of the squares of all the previous gradients of the parameter.

$$g_t^i = \frac{\partial J(w_t^i)}{\partial \mathbf{W}}$$

$$\mathbf{W} = \mathbf{W} - \alpha \frac{\partial J(\omega_t^t)}{\sqrt{\sum_{r=1}^{t} (g_r^i)^2 + \epsilon}}$$

where:
g_t^i the gradient of a parameter
α : the learning rate
ϵ : very small value to avoid dividing by zero

— **Adaptive delta(Adadelta):** Adadelta is a type of stochastic gradient descent algorithm that offers adaptive techniques for hyperparameter tuning. The name Adadelta is derived from "adaptive delta", where delta refers to the difference between the current weight and the newly updated weight.
Adadelta is an improved version of Adagrad that adjusts learning rates based on a moving window of gradient updates, instead of accumulating all past gradients. This allows Adadelta to continue learning even after many updates have been made.
The update rule in Adadelta eliminates the need to set a default learning rate, making it unnecessary to specify a learning rate.

$$v_t = \rho\, v_{t-1} + (1-\rho)\, \nabla_\theta^2 J(\theta)$$

$$\triangle\theta = \frac{\sqrt{\omega_t + \epsilon}}{\sqrt{v_t + \epsilon}} \nabla_\theta J(\theta)$$

$$\theta = \theta - \eta \triangle \theta$$

$\omega_t = \rho\omega_{t-1} + (1-\rho) \triangle \theta^2$

— **Adam Optimizer:** The Adam optimizer combines concepts from both RMSProp

and Momentum to help in computing adaptive learning rates for each parameter. It works as follows:

— Firstly, it computes the exponentially weighted average of past gradients (v_{dW})

— Secondly, it calculates the exponentially weighted average of the squares of past gradients (s_{dW})

— Thirdly, to counteract any bias towards zero, a bias correction is applied to these averages ($v_{dW}^{corrected}, s_{dW}^{corrected}$)

— Lastly, the parameters are updated using the information from the calculated averages

$$v_{dW} = \beta_1 v_{dW} + (1-\beta_1)\frac{\partial J}{\partial W}$$

$$s_{dW} = \beta_2 s_{dW} + (1-\beta_2)\left(\frac{\partial J}{\partial W}\right)^2$$

$$v_{dW}^{corrected} = \frac{v_{dW}}{1-(\beta_1)^t}$$

$$s_{dW}^{corrected} = \frac{s_{dW}}{1-(\beta_1)^t}$$

$$W = W - \alpha \frac{v_{dW}^{corrected}}{\sqrt{s_{dW}^{corrected}} + \epsilon}$$

where:

v_{dW}- the exponentially weighted average of past gradients

s_{dW}- the exponentially weighted average of past squares of gradients

β_1- hyperparameter to be tuned

β_2- hyperparameter to be tuned

$\frac{\partial J}{\partial W}$ - cost gradient with respect to current layer

W- the weight matrix (parameter to be updated)

α - the learning rate

ϵ - very small value to avoid dividing by zero

— **RMSProp Optimizer:** RMSProp optimizer is an adaptive learning rate optimization algorithm that helps to speed up convergence. It does this by keeping an exponentially weighted average of past gradient squares and dividing the learning rate by this average. This process results in a more efficient and faster convergence of the algorithm.

$$s_{dW} = \beta s_{dW} + (1-\beta)\left(\frac{\partial J}{\partial W}\right)^2$$

$$W = W - \alpha \frac{\frac{\partial J}{\partial W}}{\sqrt{s_{dW}^{corrected}} + \epsilon}$$

where:

s - the exponentially weighted average of past squares of gradients

$\frac{\partial J}{\partial W}$ - cost gradient with respect to current layer weight tensor

W - weight tensor

β- hyperparameter to be tuned

α- the learning rate

ϵ- very small value to avoid dividing by zero.

— **Stochastic Gradient Descent(SGD) Optimizer:** Stochastic Gradient Descent (SGD) is a widely used optimization algorithm for training neural networks and other machine learning models. It is a variant of the gradient descent optimization algorithm that processes each training example individually rather than using the entire dataset in each iteration. This property makes it well-suited for large datasets. Optionally, partition the dataset into mini-batches of a fixed size.

— For each mini-batch or individual example:
— Compute the gradient of the loss with respect to the model parameters.
— Update the model parameters in the opposite direction of the gradient to minimize the loss.

.

The update rule for the model parameters θ is given by:

$$\theta_t = \theta_{t-1} - \alpha \bigtriangledown J(\theta_{t-1}, x_i, y_i)$$

where:

α : is the learning rate, a hyperparameter that controls the size of the steps taken during optimization.

$\bigtriangledown J(\theta_{t-1}, x_i, y_i$ is the gradient of the loss function **J** with respect to the parameters θ for the example x_i, y_i

4.4.8 Regularization

Regularization techniques are widely used in machine learning to reduce overfitting and improve the generalization performance of models. These methods introduce constraints or penalties to the training process, which encourage the model to be simpler and more robust. By doing so, they help to prevent the model from fitting too closely to the training data, thereby improving its ability to generalize to new, unseen data. [23], [10], [25]

— **Data Augmentation:** Having more data is the surest way to get better consistent estimators (ML model), in contrast having a small dataset will lead to the

well-known problem of overfitting. Data augmentation refers to the technique of artificially increasing the size of a dataset by applying various transformations to the existing data. This is often used in deep learning, particularly in computer vision tasks, to improve the performance of neural networks. There are various types of data augmentation techniques, including:

1. Flipping: horizontally or vertically flipping an image.
2. Rotation: Rotating an image by a certain degree is the process of rotating the image based on a specific angle.
3. Zooming: zooming in or out of an image.
4. Translation: shifting an image horizontally or vertically.
5. Cropping: cropping a portion of an image.
6. Adding noise: adding random noise to an image.

These techniques can be used individually or in combination to generate new data samples. By increasing the size of the dataset, the neural network is exposed to more variations of the same data, which helps improve its ability to generalize and make accurate predictions on new, unseen data. [26]

— **Dropout:** is a technique used to reduce overfitting in neural networks. It works by randomly ignoring selected neurons during training, which prevents complex co-adaptations on the training data.
During the training process, some of the neurons are randomly dropped out, meaning that their contribution to the activation of downstream neurons is temporarily removed on the forward pass. Additionally, any weight updates are not applied to the neuron on the backward pass.
Simply put, the process of ignoring some of the neurons occurs during a particular forward or backward pass. The probability of randomly selecting nodes to be dropped out can be easily adjusted (e.g. 0.1%) each weight update cycle.
It is important to note that Dropout is only used during the training of a model and is not used when evaluating the model.

— **Early Stopping:** Early stopping is an alternative technique that can be used to prevent overfitting. This approach involves using the validation error to decide when to stop training the neural network.
The biggest challenge in training a neural network is determining how long to train the model. If you train the model too little, it will result in underfitting in both the train and test sets. On the other hand, if you train it too much, it will overfit the training set and perform poorly on the test sets.
The key is to train the network long enough that it can learn the mapping from inputs to outputs, but not so long that it overfits the training data. One possible solution is to treat the number of training epochs as a hyperparameter and train the model multiple times with different values. Then, you can select the number of epochs that results in the best accuracy on the train or a holdout test dataset.

However, this solution requires training and discarding multiple models, which can be time-consuming and resource-intensive. [23]

— **Ensembling:** Ensemble methods are machine-learning techniques that combine multiple models into one predictive model. There are two primary approaches to ensembling: Bagging and Boosting.:
  — **Bagging** Bagging, which stands for bootstrap aggregation, is a technique that reduces the variance of an estimate by averaging multiple estimates. It involves training a large number of "strong" learners in parallel. A strong learner is a model that is relatively unconstrained. Bagging then combines all the strong learners to obtain a smoother and more accurate prediction.
  — **Boosting** Boosting is a family of algorithms that convert weak learners into strong learners. It involves a sequence of models, with each one focusing on learning from the mistakes of the previous model. The final result is a combination of all the weak learners, which results in a single strong learner. Bagging utilizes complex base models to “smooth out” their predictions, while boosting uses simple base models to "boost" their combined complexity.

  Bagging uses complex base models and tries to “smooth out” their predictions, while boosting uses simple base models and tries to “boost” their aggregate complexity.

— **Injecting Noise:** Noise is a phenomenon commonly introduced to the inputs as a technique for dataset augmentation. In situations where the dataset is small, the neural network may tend to memorize the training dataset instead of learning the general mapping from the inputs to the outputs. In other words, the model may only memorize specific input examples and their respective outputs. To solve this problem and enhance the structure of the mapping, one approach is to add random noise to the inputs.

  By adding noise, the network becomes less prone to memorizing the training samples since they keep changing all the time. This leads to smaller network weights and a more robust network, which in turn results in lower generalization error.

  It’s important to note that noise is only added during the training phase. No noise is added when the model is evaluated or used to make predictions on new data. Additionally, random noise can be added to other parts of the network during training. Some examples include:

  1. **Noise Injection on Weights** Adding noise to the weights of a model can be seen as a form of regularization that encourages the model to not be overly sensitive to small changes in the weights. By doing so, the model is able to avoid finding only local minima and instead find global minima surrounded by flat regions. This can result in a more robust and accurate model.
  2. **Noise Injection on Outputs** It is common for real-world datasets to have

errors in their output labels. In order to address this issue, one approach is to explicitly model the noise in the labels. An example of this is the technique of Noise Injection on Outputs, which can be achieved through a process known as label smoothing.

— **L1 Regularization:** A regression model that uses the L1 regularization technique is called Lasso Regression. The objective of L1 regularization is to push some of the weight coefficients to zero, effectively performing feature selection. This results in a sparse model where only a subset of the input features are used. [27]
The formula for L1 regularization is:
Mathematically:

$$\text{Loss} = \text{Error}(Y, \hat{Y})$$

Following formula calculates the error With L1 Regularization function

$$\text{Loss} = \text{Error}(Y - \hat{Y}) + \lambda \sum_{1}^{n} |\omega_i|$$

where:

$$\hat{Y} = \omega_1 x_1 + \omega_2 x_2 + ... + \omega_n x_n + b$$

L1 Regularization (or a variant of this concept) is a model of choice when the number of features is high Since it provides sparse solutions.

— **L2 Regularization:** L2 regularization is a technique used in regression models, and when it is applied, the model is called Ridge Regression. The primary difference between L1 and L2 regularization techniques is that L2 regularization adds a penalty term to the loss function, which is the squared magnitude of the coefficient. [27]
Mathematical formula for L2 Regularization.

$$\text{Loss} = \text{Error}(Y, \hat{Y})$$

$$\text{Loss} = \text{Error}(Y - \hat{Y}) + \lambda \sum_{1}^{n} \omega_i^2$$

4.4.9 Types of Deep learning architectures

Deep learning architectures are neural network models that are specifically designed to learn and make predictions from complex datasets like images, speech, and text.

There are a wide variety of algorithms and architectures used in deep learning to accomplish this task.

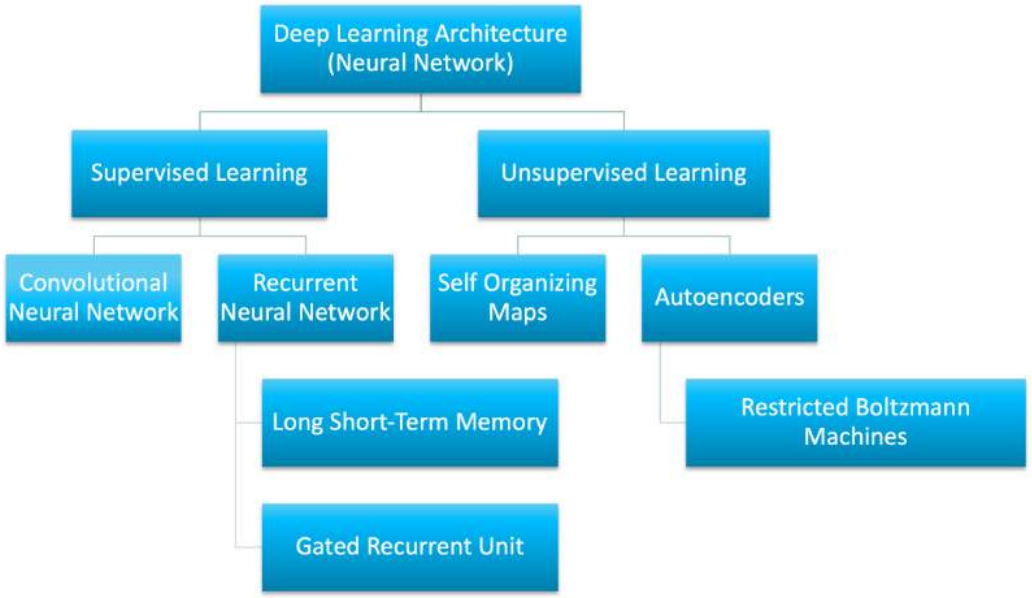


Figure 4.16. Deep learning architectures [28]

— **Supervised deep learning:** Supervised learning refers to the problem space wherein the target to be predicted is clearly labeled within the data that is used for training.

  — **Convolutional neural networks:** A CNN is a multilayer neural network used for image processing, inspired by the animal visual cortex. It was first created by Yann LeCun for recognizing handwritten characters. Early layers detect basic features, while subsequent layers combine these features to extract higher-level attributes of the input image. The LeNet CNN architecture performs feature extraction and classification through multiple layers, including convolutional and pooling layers, a fully connected multilayer perceptron, and an output layer.

39

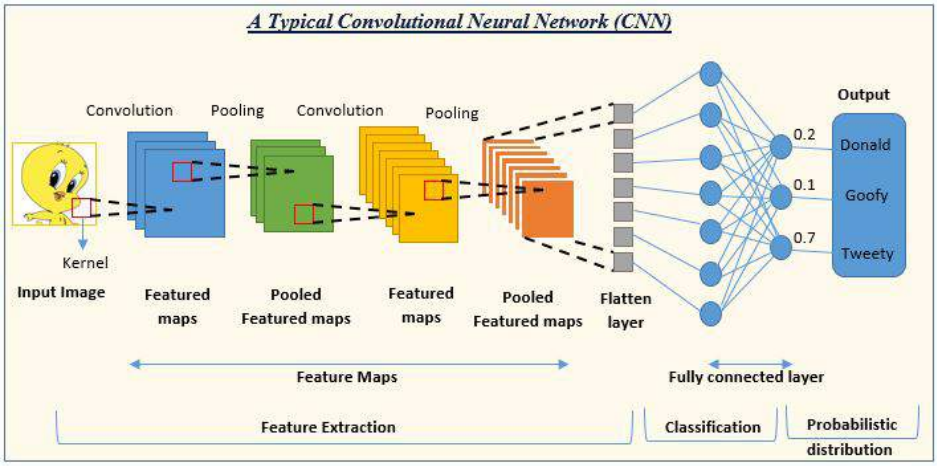


Figure 4.17. Convolutional Neural Network [29]

The network is trained through back-propagation. A Convolutional Neural Network (CNN) is a highly effective multilayer neural network that was inspired by the visual cortex of animals. It is primarily used for image processing applications. The first-ever CNN was created by Yann LeCun, which revolutionized the recognition of handwritten characters such as postal codes.
With its deep network architecture, CNN can detect basic features, such as edges, through its initial layers and then combine these features to extract higher-level attributes of the input image.
The LeNet CNN architecture, consisting of multiple layers that perform feature extraction and classification, is a prime example of CNN's effectiveness. It includes convolutional and pooling layers, a fully connected multilayer perceptron, and an output layer that identifies features of the image. The network is trained through back-propagation, making it an even more efficient tool. [30] [10]

— **The Gated Recurrent Unit (GRU) networks** In 2014, a simpler version of the LSTM network was introduced, known as the Gated Recurrent Unit (GRU). The GRU has two gates, an update gate and a reset gate, which replace the output gate present in the LSTM. The update gate determines how much of the previous cell contents should be kept, while the reset gate determines how to combine the new input with the previous cell contents. By setting the reset gate to 1 and the update gate to 0, a GRU can function as a standard RNN.[28]

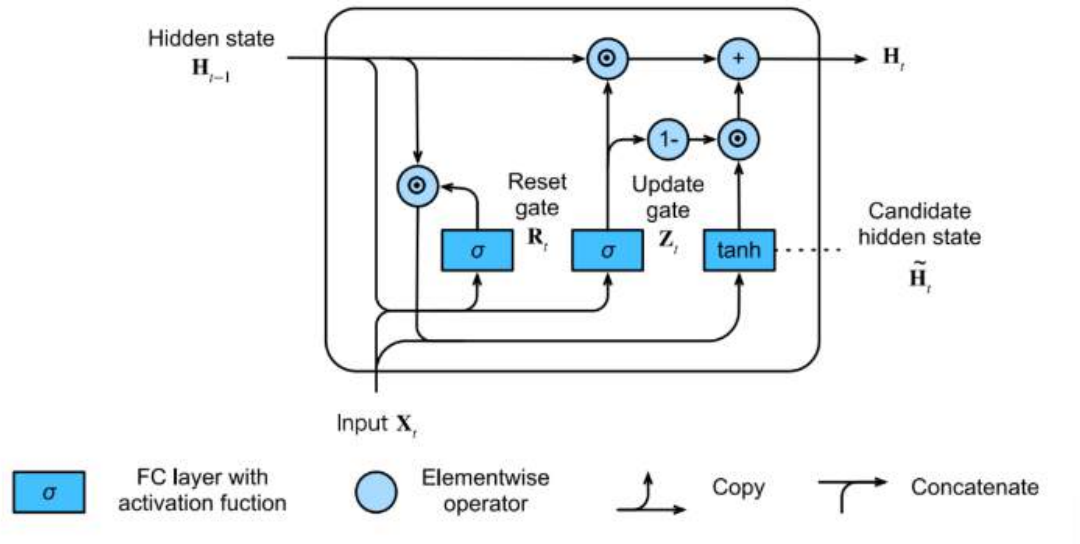


Figure 4.18. GRU networks [20]

The LSTM is more expressive and can lead to better results with more data, while the GRU is simpler, can be trained more quickly, and can be more efficient in its execution.

— **Recurrent Neural Network (RNN):** is a neural network that contains a hidden state which captures historical information up to the current timestep. The hidden state of the current state uses the same definition as the previous timestep, which makes the computation recurrent, hence its name.
The RNN is a foundational network architecture from which other deep learning architectures are built. The primary difference between a typical multilayer network and an RNN is that an RNN may have connections that feed back into prior layers (or the same layer) instead of completely feed-forward connections. This feedback allows RNNs to retain memory of past inputs and model problems in time.
RNNs include a rich set of architectures, and one popular topology is called Long Short-Term Memory (LSTM). The key differentiator is feedback within the network, which can appear from a hidden layer, the output layer, or some combination of both.
RNNs can be unfolded over time and trained using standard backpropagation or a variant called backpropagation through time (BPTT). [31]

41

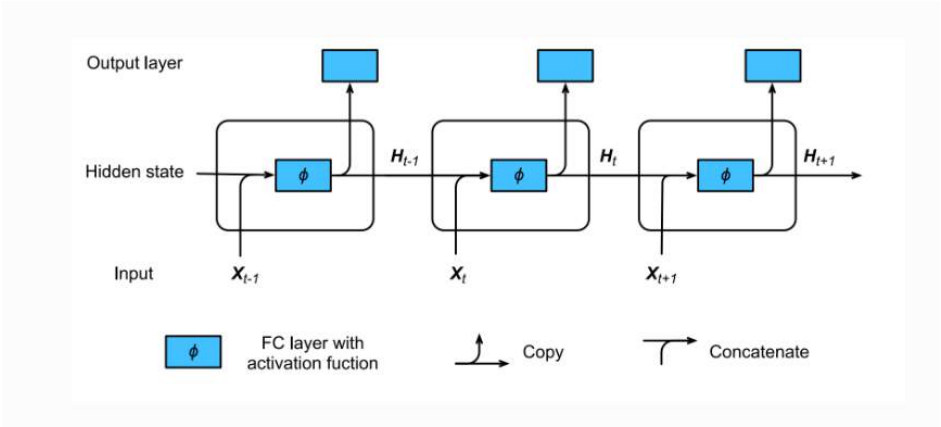


Figure 4.19. Recurrent Neural Network [20]

— **Gated Recurrent Unit (GRU) Layer:** GRU supports:
hidden gate the gating of hidden state,
Reset gate controls how much of the previous hidden state we might still want to remember.
Update gate controls how much of current hidden state is just a copy of the previous state The structure and math are as follow:

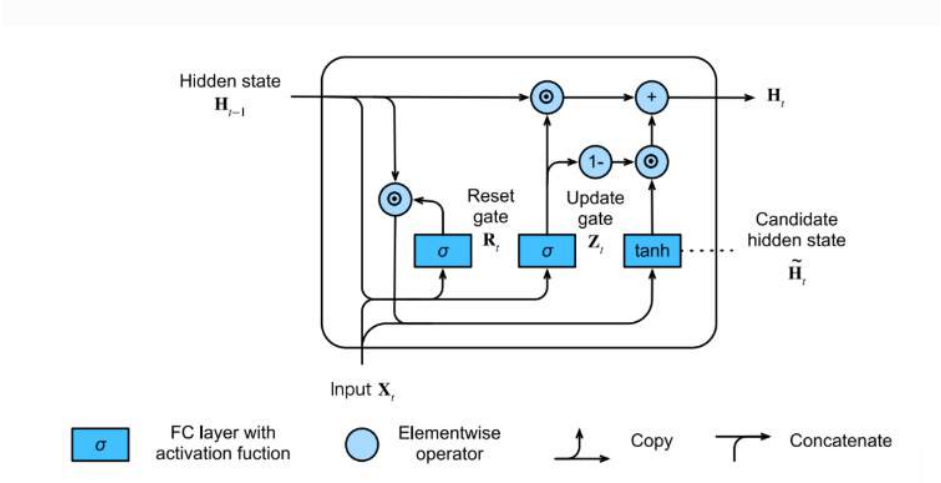


Figure 4.20. Gated Recurrent Unit [20]

— **Long short-term memory (LSTM):** Long Short-Term Memory (LSTM) is a type of recurrent neural network (RNN) architecture. It is specifically designed to address the vanishing gradient problem, which can occur in traditional RNNs. LSTMs are particularly useful for handling tasks that involve sequences of data, such as speech recognition, natural language processing, time series analysis, and more. The most remarkable feature of LSTMs is their ability to capture and remember long-term dependencies in sequential data while avoiding the vanishing gradient problem. This is a significant advantage over traditional RNNs, which can struggle with the training of long sequences. [30] [28]
Here are the main components and features of an LSTM:

Cell State (Ct):
The cell state in LSTM serves as a long-term memory. It remains unchanged throughout the chain of the LSTM network, with only some minor linear interactions. It works like a conveyor belt that runs through the entire sequence, and information can be added or removed from it as needed.

Hidden State (ht):

The hidden state in LSTM is essentially a short-term memory that plays a crucial role in capturing and retaining short-term dependencies in the sequence. It is determined by both the current input and the previous hidden state at each time step.
Input Gate: The input gate is responsible for determining the amount of newly received information that should be added to the cell state. It utilizes a sigmoid activation function to determine which values should be updated (values close to 1) and which values should be ignored (values close to 0).
Forget Gate: The forget gate is responsible for determining which information from the cell state should be disregarded. It takes into account the previous hidden state and the current input to decide which parts of the cell state are no longer important. It utilizes a sigmoid activation function to produce output values ranging from 0 to 1.
Cell State Update: Cell state update is determined by two gates - the input gate and the forget gate. These gates work together to decide what new information should be added to the cell state and what information should be removed from it. The input gate determines the new information to be stored, while the forget gate determines which information should be discarded.
Output Gate:
The output gate plays a crucial role in deciding the next hidden state. It utilizes two activation functions - sigmoid and tanh. Sigmoid determines which parts of the cell state should be outputted, while tanh generates a vector of new candidate values that are added to the hidden state. Layers

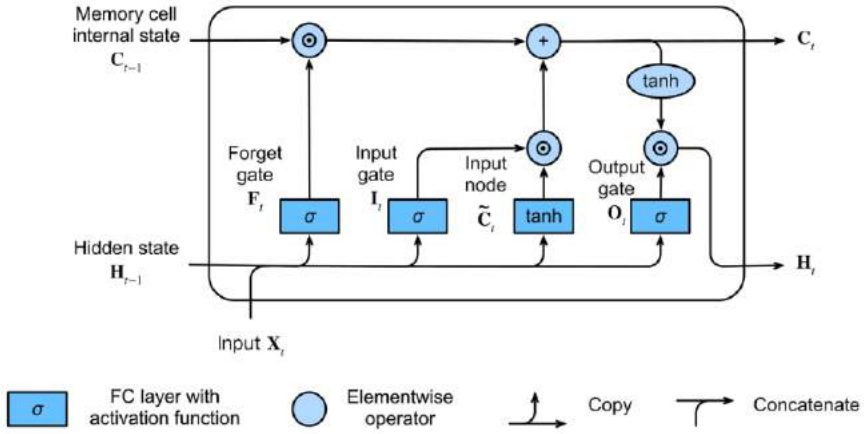


Figure 4.21. Long short-term memory [32]

4.4.10 Section Unsupervised learning

Unsupervised learning involves training models without target labels present within the data using unsupervised architectures.[15]

- **Self-organized maps:** Self-Organizing Maps (SOM) are a type of unsupervised neural network invented by Dr. Teuvo Kohonen in 1982, also known as Kohonen maps. Unlike traditional artificial neural networks, SOMs create clusters of input data by reducing input dimensionality. [30], [28]

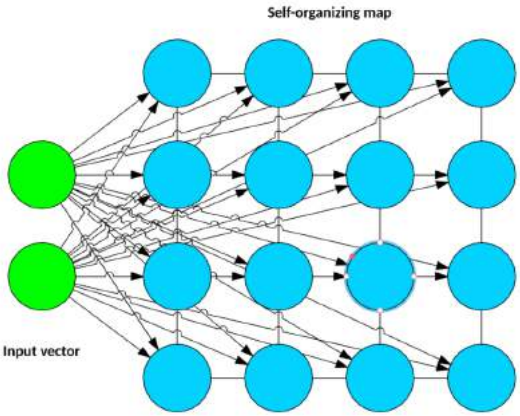


Figure 4.22. Self-organized map (SOM)

Example applications: Dimensionality reduction, clustering high-dimensional inputs to 2-dimensional output, radiant grade result, and cluster visualization

- **Autoencoders:** Autoencoders are a variant of artificial neural networks (ANNs) that consist of three layers: the input layer, the hidden layer, and the output layer. First, the input layer is encoded into the hidden layer using an appropriate encoding function. The number of nodes in the hidden layer is much less than the number of nodes in the input layer. This hidden layer contains a compressed representation

of the original input. Finally, the output layer aims to reconstruct the input layer by using a decoder function.

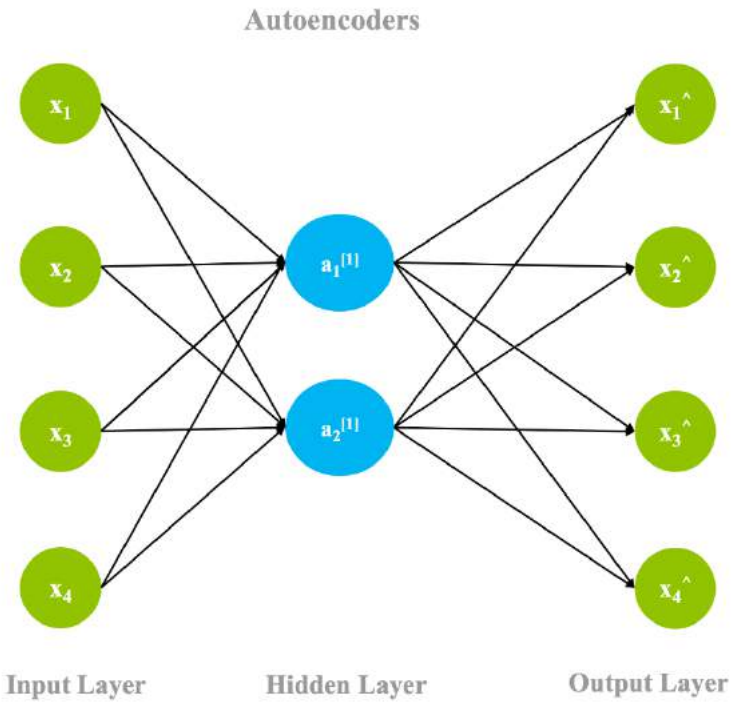


Figure 4.23. Autoencoders [28]

During the training phase, an error function is used to calculate the difference between the input and output layers, and the weights are adjusted accordingly to minimize the error.
Example applications: Dimensionality reduction, data interpolation, and data compression/decompression.

- **Restricted Boltzmann Machines:** A Restricted Boltzmann Machine (RBM) is a type of neural network that consists of two layers: input and hidden layers. As depicted in figure 4.24, every node in the hidden layer is connected to every node in the visible layer. Unlike traditional Boltzmann machines, where nodes in the input and hidden layers are interconnected, RBMs restrict the connections between nodes within a layer due to computational complexity.

45

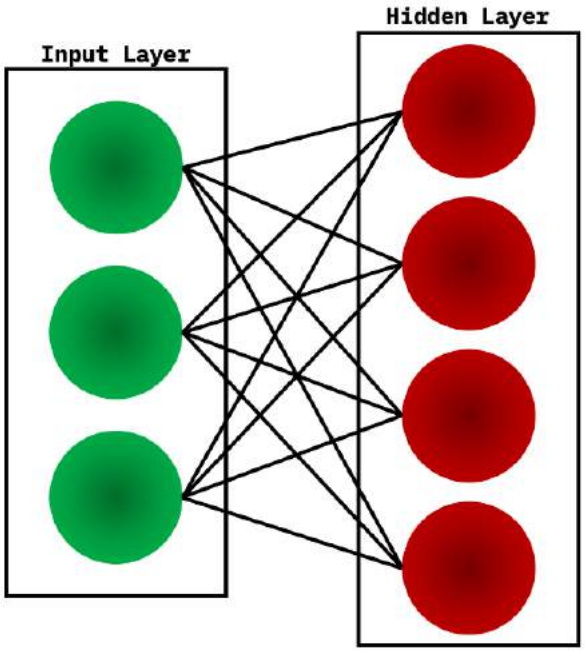


Figure 4.24. Restricted Boltzmann Machines [28]

During the training phase of RBMs, a stochastic approach is used to calculate the probability distribution of the training set. At the beginning of the training, each neuron is activated at random. The model also contains hidden and visible biases. The hidden bias is used in the forward pass to build the activation, while the visible bias helps in reconstructing the input. RBMs are also called generative models because the reconstructed input is always different from the original input. Additionally, due to the built-in randomness, the same predictions result in different outputs. RBMs is a deterministic model, and that makes it significantly different from Autoencoder. [28] [30]
Example applications: Dimensionality reduction and collaborative filtering.

- **Deep belief networks:** Deep belief networks (DBN) are multilayer networks that typically have several hidden layers, making them deep. Each pair of connected layers in a DBN is a Restricted Boltzmann Machine (RBM). The input layer of the DBN represents the raw sensory inputs, while each hidden layer learns abstract representations of this input. The output layer is treated differently and is responsible for network classification during training. The DBN is trained in two steps: unsupervised pretraining and supervised fine-tuning. [33], [15]

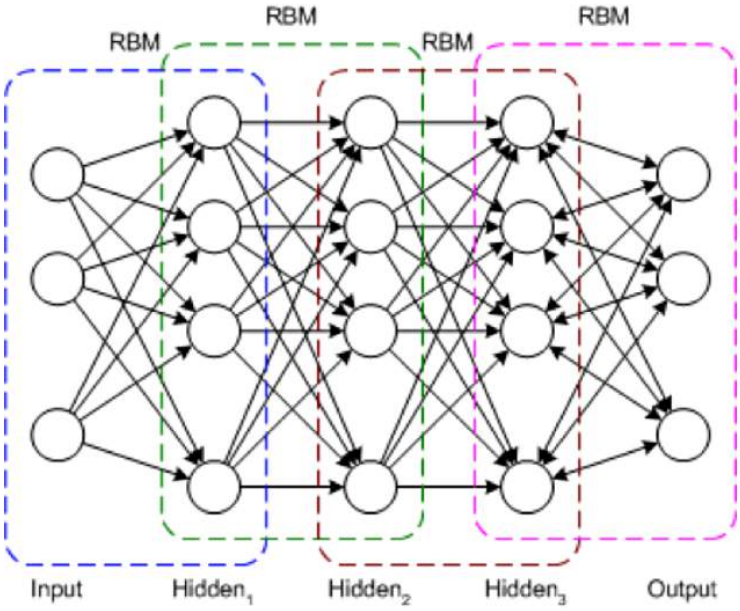


Figure 4.25. Deep belief networks architecture [28]

During unsupervised pretraining, each Restricted Boltzmann Machine (RBM) is trained to reconstruct its input. For instance, the first RBM reconstructs the input layer to the first hidden layer. Similarly, the next RBM is trained by using the outputs of the previous hidden layer as the inputs, and so on until each layer is pretrained. After pretraining, fine-tuning commences, and the output nodes are assigned labels to give them meaning, i.e., to signify what they represent in the context of the network. The final step involves applying full network training using either gradient descent learning or back-propagation to complete the training process.
Applications: object detection, information processing, natural language understanding processing, etc.

- **Deep stacking networks:** Deep stacking networks, also known as deep convex networks, are different from traditional deep learning frameworks. Although they consist of a deep network, they are actually a deep set of individual networks, each with its own hidden layers.
  This architecture was developed to solve one of the main problems with deep learning: the complexity of training. Each layer in a deep learning architecture exponentially increases the complexity of training, but the DSN views training as a set of individual training problems, instead of a single problem.
  The DSN consists of a set of modules, each of which is a subnetwork in the overall hierarchy of the DSN. For example, in one instance of this architecture, three modules are created for the DSN. Each module consists of an input layer, a single hidden layer, and an output layer. Modules are stacked one on top of another, where the inputs of a module consist of the prior layer outputs and the original input vector.

This layering allows the overall network to learn more complex classifications than would be possible given a single module. [34], [28]

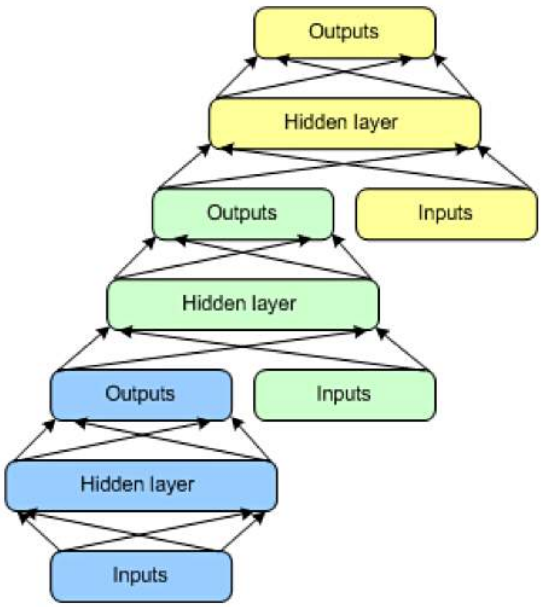


Figure 4.26. Deep stacking networks architecture [28]

Example applications: Information retrieval and continuous speech recognition

5. SOTA of object detection

5.1 Introduction to Object Detection

Object detection Object detection is a vital computer vision technique which is employed to identify instances of objects within images or videos. This technique is commonly implemented through the use of machine learning or deep learning algorithms, which are responsible for producing robust and accurate results. While humans are able to quickly recognize and locate objects of interest within visual data, the ultimate goal of object detection is to replicate this cognitive ability using computational methods. By detecting and localizing objects within visual data, object detection has become a crucial tool for various computer vision applications, including robotics, autonomous vehicles, and surveillance systems. [35], [10]

5.1.1 Why Object Detection

Identifying and localizing objects within an image or video is called object detection and is a fundamental task in computer vision. It plays a pivotal role in various real-world applications, contributing to advancements in technology and enhancing our daily lives. The importance of object detection lies in its ability to enable machines to interpret and understand visual information, allowing them to make informed decisions and interact intelligently with the environment.[36],

1. **Automation and Efficiency:** Object detection is a cornerstone in developing automated systems, empowering machines to perceive and respond to their surroundings. This is particularly crucial in industries such as manufacturing, where the automation of tasks, such as quality control and inventory management, leads to increased efficiency and reduced operational costs.
2. **Enhanced Security and Surveillance:** In the realm of security and surveillance, object detection is instrumental in identifying potential threats or anomalies. Surveillance cameras equipped with object detection algorithms can automatically detect and alert authorities to suspicious activities, enhancing public safety in crowded spaces, transportation hubs, and critical infrastructure.
3. **Autonomous Vehicles:** The advent of autonomous vehicles relies heavily on object detection to interpret the dynamic environment. Cars equipped with object detection systems can identify pedestrians, vehicles, and obstacles, enabling safer navigation and reducing the likelihood of accidents.
4. **Medical Imaging and Diagnosis:** In the field of healthcare, object detection plays a vital role in medical imaging. It aids in detecting and localizing abnormalities in radiological images, facilitating early diagnosis and improving patient outcomes.
5. **Retail and Customer Experience:** Object detection is utilized in retail for tasks such as inventory management, shelf monitoring, and cashierless checkout systems. These

applications streamline operations and enhance customer experience by reducing waiting times and optimizing stock levels.

Real-world Scenarios:

1. **Smart Cities:** Object detection is integral to developing smart cities. It can be used in urban environments for traffic management, waste management, and monitoring public spaces, contributing to more efficient and sustainable urban living.
2. **Search and Rescue Operations:** In disaster-stricken areas, object detection aids in search and rescue operations by identifying and locating individuals needing assistance. Drones equipped with object detection capabilities can quickly cover large areas, improving rescue efforts' efficiency.
3. **Environmental Monitoring:** Object detection is applied in environmental science for monitoring wildlife, tracking deforestation, and studying biodiversity. It enables researchers to gather crucial data for conservation and ecological studies.
4. **Augmented Reality:** Object detection is a key component in augmented reality applications, where virtual elements are seamlessly integrated with the real world. This technology enhances user experiences in gaming, education, and various interactive scenarios.

[10], [36]

5.2 Object detection stage

Object detection can be classified into two stages according to the detection process steps:

- **One-stage detector** is a simple regression problem that takes input and learns probability classes and bounding box coordinates. YOLO, YOLO v2, SSD, RetinaNet, etc., fall under one-phase detectors. Object detection is an advanced form of imaging classification where a neural network predicts objects in an image and draws attention to them in the form of bounding boxes. [37]
- **Two-stage detector** A two-stage detector completes detection in two steps. In the first step, regional design networks are used to create areas of interest with a high probability of being objects. In the second step, object detection is performed, which includes the final classification and regression of the bounding box of the detected objects. RCNN, Fast RCNN, SPPNET, Faster RCNN, etc., are some of the two-stage detectors. [38]

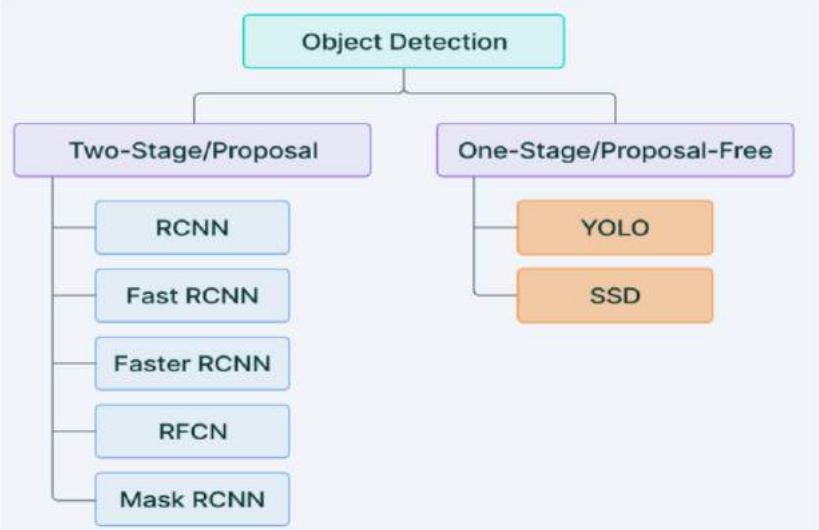


Figure 5.1. Stage or classification of Object Detection

5.3 Two-Stage/Proposal:

Two-stage object detection algorithms typically have two steps: region proposal and object classification or refinement. One popular framework is the region-based Convolutional Neural Network (R-CNN) and its variants.[38], [10]

5.3.1 Region-Based Convolutional Neural Network:

R-CNN A Region-Based Convolutional Neural Network (R-CNN) is a type of neural network specifically designed to detect objects in images. R-CNN uses region proposals generated by a selective search (SS) approach to pre-compute the priors. Since the features between these regions are not shared, we need to extract features individually for each region of interest (RoI) generated using an SS approach. Unlike other neural networks, R-CNNs are designed not only to classify objects but also to locate and delineate their boundaries within the image. The R-CNN architecture was first proposed in 2014 by Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik in their paper titled "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation." Since then, the architecture has undergone several iterations, including Fast R-CNN, Faster R-CNN, and Mask R-CNN.

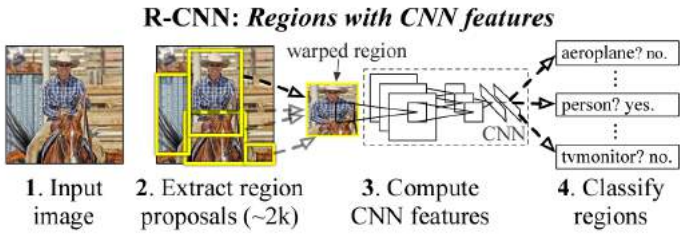


Figure 5.2. R-CNN Model(Source:[39])

51

R-CNN comprises of the following three modules:

1. **Region proposal:** The R-CNN algorithm starts by identifying regions in an image that may contain objects. These regions are known as region proposals. They are referred to as proposals because they may or may not contain objects, and the goal of the learning function is to remove areas that do not contain objects. Region proposals are bounding boxes around the objects.
2. **Feature extraction:** To identify objects in an image, the first step is to crop out the region proposals and resize them. These resized images are then sent through a standard CNN for feature extraction. The original research paper employed AlexNet for this purpose. The CNN extracts 4,096-dimensional feature vectors from each region.
3. **Classifier:** The extracted features are classified by using the standard classification algorithms, such as the linear SVM model

R-CNN was the first successful deep learning–based object detection system, but it suffered a serious issue concerning performance. Its time performance problem is because of the following:

- For feature extraction, each region proposal undergoes approximately 2,000 passes per image in the CNN.
- Three models are trained: CNN for feature extraction, classifier for image class prediction, and regression for bounding box refinement. Training is compute-intensive, increasing computation time.
- Due to the large number of regions, CNN predictions are slow for each proposal.

5.3.2 Fast R-CNN:

In 2015, Ross Girshick from Microsoft proposed a single model called "Fast R-CNN" to overcome the limitations of R-CNNs. This model learns and outputs region proposals and classifications directly, resulting in higher mAP on PASCAL compared to R-CNN.[40]

Fast R-CNN trains a deep VGG-16 network that is significantly faster than R-CNN (9x faster during training and 213x faster at test time). The model achieves this speed by evaluating the network and extracting features for the entire image once, instead of extracting features from each region of interest (RoI) as in R-CNN. To extract features from RoIs, Fast R-CNN uses the concept of RoI pooling, which is a special case of pyramid pooling used in SPPNet.

RoI pooling provides a feature vector of the desired length that is then used for classification and localization. This method is more efficient than R-CNN because the computations for overlapping regions are shared. Overall, Fast R-CNN significantly improves the speed and performance of object detection models.

Building blocks of Fast R-CNN:

1. Region proposal network

2. Feature extraction using CNN
3. RoI pooling layer: This is where the real magic of Fast R-CNN happens
4. Classification and Localization

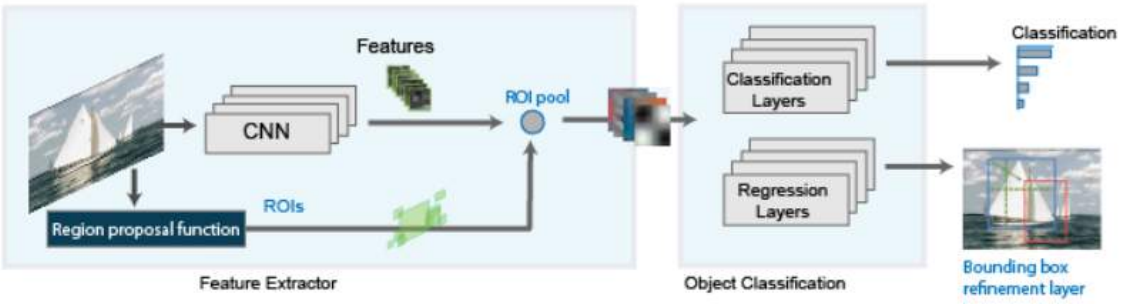


Figure 5.3. Architectural Design of Fast R-CNN [39]

The region proposal network and feature extraction modules work similarly to the ones used in R-CNN. However, instead of passing each cropped and re-scaled RoI (region of interest), the entire input image is processed through a feature extractor like VGG-16. This produces a convolutional feature map. The features (i.e., the convolutional feature map) are then combined with the region proposal network, which uses a selective search approach to create a fixed-length feature vector in the RoI pooling layer. These feature vectors are then passed along to the classification and localization modules. The classification module classifies K+1 (1 for background) object classes using a softmax probability. The localization module outputs four real-valued numbers for each K object class. [40]

5.3.3 Faster R-CNN

Faster R-CNN is an improved version of Fast R-CNN from the training speed and detection accuracy perspectives. Faster R-CNN added what they called a Region Proposal Network (RPN) in an attempt to get rid of the selective search algorithm and make the model completely trainable end-to-end.

53

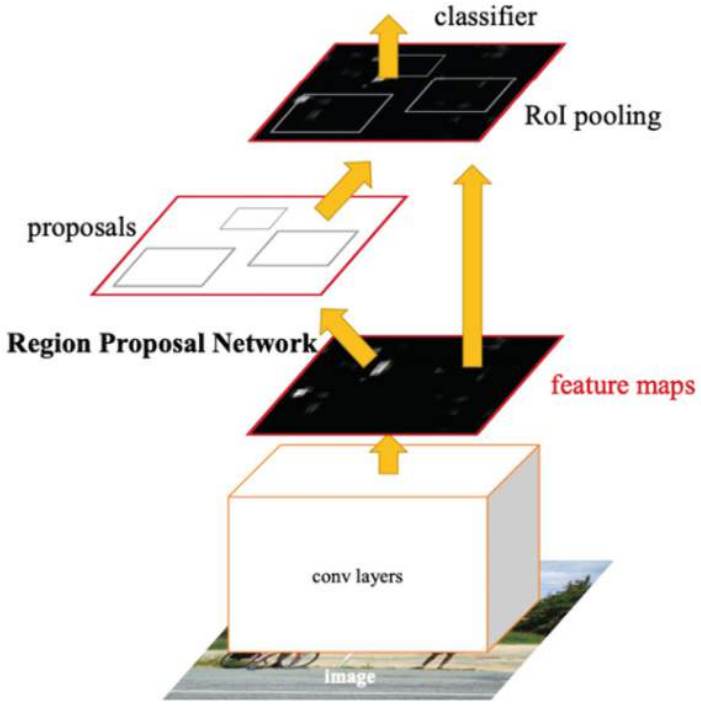


Figure 5.4. Architectural Design of Faster R-CNN [10]

1. **Region Proposal Network:** An RPN is a fully convolutional neural network that predicts object bounds and objectness scores simultaneously at each position of the image. An RPN is a deep CNN that takes an image input and generates the output as a set of rectangular object proposals. Each rectangular proposal has an "objectness" score.

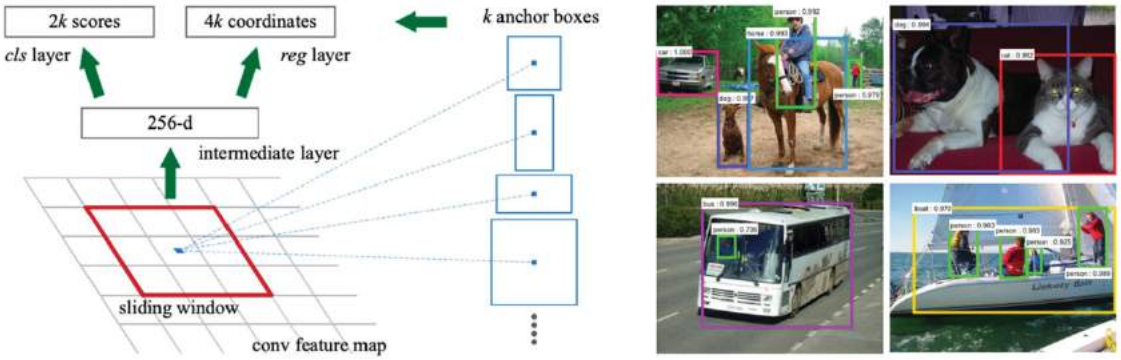


Figure 5.5. Region Proposal Networks (RPN) [image source: [41]

RPN has a classifier and a regressor. The concept of anchors has been introduced, which refers to the central point of the sliding window. The classifier determines the probability of a proposal having the target object. Regression regresses the coordinates of the proposals. At each sliding window location, the algorithm predicts multiple region proposals. If the maximum number of proposals at each window location is k, then the total number of bounding box coordinates will be 4k, while the

number of object classes will be 2k. One of the classes will indicate the probability of an object being present in the region, and the other will indicate the probability of no object being present. These region boxes at each window are referred to as anchors.

2. **Fast R-CNN:** The Faster R-CNN consists of two parts, with the second part being the detection network. This part is identical to the Fast R-CNN, which was described earlier. The Fast R-CNN uses input from the RPN to detect objects in images.

[10], [42]

5.3.4 Mask R-CNN

The Mask R-CNN extends the Faster R-CNN. The Mask R-CNN adds another branch for the object class, bounding box coordinates, and predicting an object mask. Here is how the Mask R-CNN differs from its predecessor, the Faster R-CNN:

- The Faster R-CNN algorithm produces two main results, namely: a predicted class label and the coordinates of the bounding box for the object of interest in an image.
- The Mask R-CNN has three outputs: a class label, bounding box coordinates, and an object mask.

The Mask R-CNN algorithm classifies each pixel of an image into a specific category, without distinguishing between individual object instances. To achieve this, it employs a technique known as pixel-to-pixel alignment between the output and input layers of the neural network. The classification of each pixel is then used to determine the masks in the region of interest (ROI).[43]

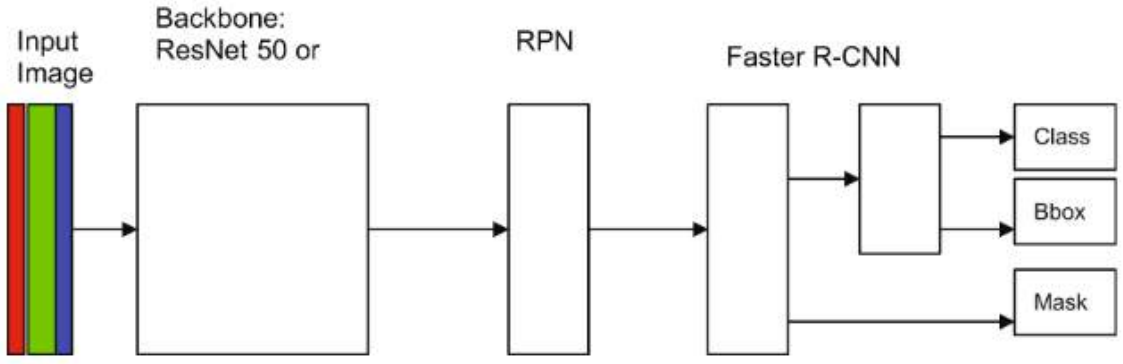


Figure 5.6. Mask R-CNN network architecture [43]

As shown in Figure 5.6, the network consists of three components **modules—backbone, RPN, and output head**.

- **Backbone** The backbone networks are commonly seen in object detection model architectures. The original paper describes using ResNet-50 and ResNet-101[Ref1] The backbone's main role is feature extraction. In addition to ResNet, a feature pyramid network (FPN) is also utilized to extract the feature details of the image.

55

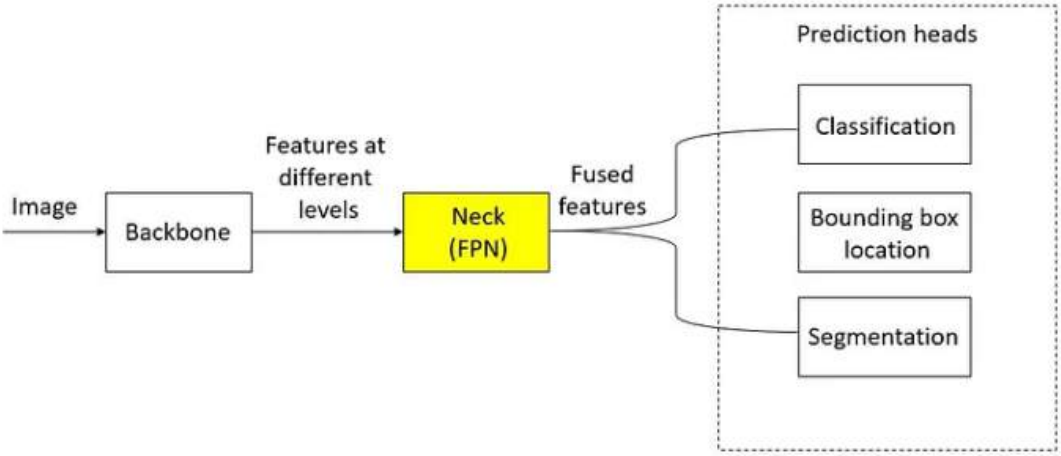


Figure 5.7. BackBone and FPN Tuning architecture [Ref]
[44]

The FPN consists of decreasing-size layers of a CNN, in which case each forward layer has fewer neurons.

- **Feature Pyramid Network (FPN):** is a neck network that combines features of different resolutions obtained from a backbone network, such as ResNet. A CNN-based backbone applies convolution layers to an input image, which results in a set of feature maps with decreasing resolution due to pooling or convolution with a stride different from one.
  The FPN consists of a bottom-up and top-down pathway. In the bottom-up pathway, a backbone network, such as ResNet, extracts features with diminishing spatial resolution. As the levels of resolution decrease, the semantic meaning of the feature maps increases, as indicated by the blue thickness of the box boundaries.
  The feature maps are fused in the top-down pathway to incorporate rich semantic meaning and precise spatial information, as illustrated in the figure. 5.8 below. [45]

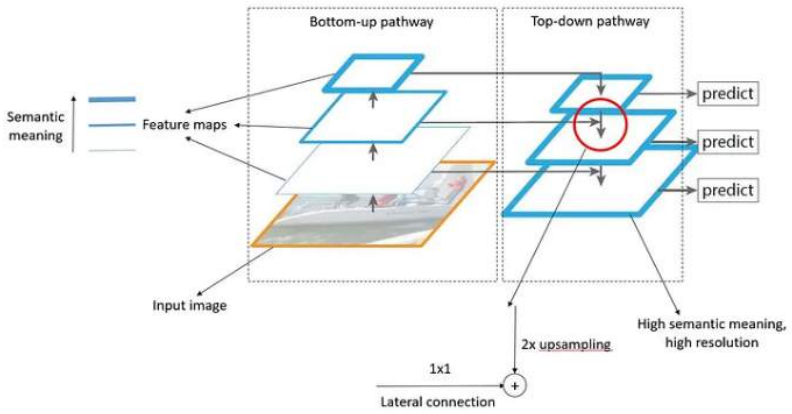


Figure 5.8. Feature Pyramid Network (FPN) [45]

56

- **Output Head:** The last module consists of the Faster R-CNN with an additional output branch. [43]

5.4 One stage detector:

They can process images in real-time and are suitable for applications where low latency is crucial, such as real-time object detection in video streams or robotics.

5.4.1 Single-Shot Multibox Detection:

SSD is primarily designed to solve object detection problems in real-time. An R-CNN and its variants are detectors that work in two stages. They have two specialized networks: one creates the region proposals to predict bounding boxes, and the other predicts the object classes. These detectors are relatively accurate but have a high computational cost. As a result, an R-CNN is not ideal for detecting objects in real-time streaming videos. A single-shot object detector predicts object classes and bounding boxes in one pass or single forward-pass[46]

- **SSD Network Architecture:** The SSD approach is a method that uses a feed-forward convolutional network to generate a set of fixed-size bounding boxes and scores, which helps detect the presence of object class instances in those boxes. A non-maximum suppression step is performed in the final stage to obtain the ultimate detections.

  — Single Shot MultiBox Detector (SSD) is a novel object detection framework that adopts grid-based image division instead of the conventional sliding window approach. The SSD approach divides the image into a grid pattern, where each grid cell is responsible for detecting objects within its corresponding region of the image. The objective of object detection is to predict the class and location of an object within its region. In case no object is present, the region is considered a background class, and the location is consequently ignored.
  During the training process, SSD requires only an input image and ground truth boxes for each object. In a convolutional manner, a small set of default boxes of different aspect ratios is evaluated at each location in several feature maps with different scales. For instance, 8x8 and 4x4 in (b) and (c). For each default box, SSD predicts the shape offsets and confidences for all object categories (c1, c2, ..., cp). Notably, these default boxes are first matched to the ground truth boxes during training.

57

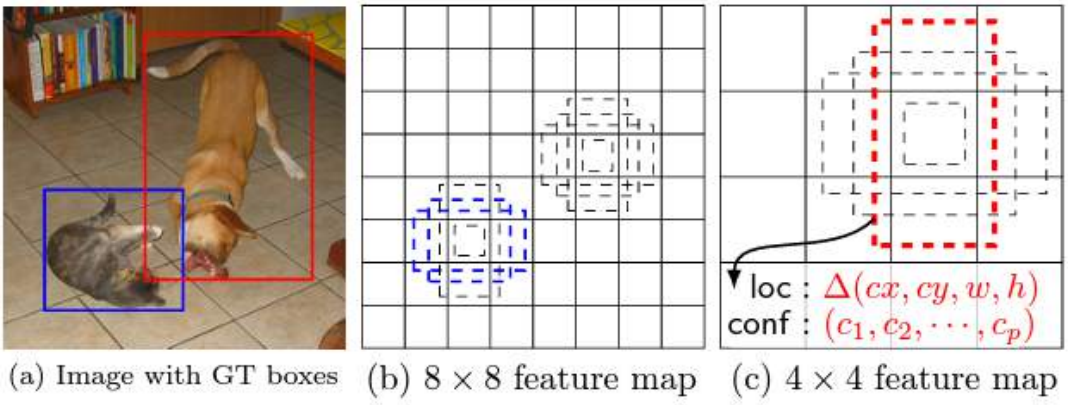


Figure 5.9. SSD:Single Shot Multi-Box Detector image source: [46]

— **Anchor Boxes:** Object detection is a process by which we aim to identify and locate objects as they appear within an image. This process is different from image classification because there may be multiple objects of the same or different classes present in the image, and object detection seeks to predict all of these objects accurately. In object detection, we use anchor boxes to help locate these objects in the image. Object detection models tackle this task by breaking the prediction step into two pieces: Ref: Anchor-box

1. First, they predict a bounding box through regression and
2. Second, by predicting a class label through classification.

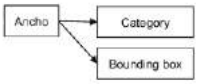


Figure 5.10. Anchor

To detect and locate multiple objects in an image, modern object detection models such as SSD, EfficientDet, and the YOLO models use anchor boxes as a starting point and then make adjustments accordingly. These anchor boxes are predetermined, and each one is responsible for a specific size and shape within a grid cell.

Anchors are rectangular shapes that are set at each convolution point of the feature map. In SSD, each grid cell can have multiple anchors or prior boxes assigned to it.

In Figure 5.11, there are five rectangular anchors (shown in red outlines) set at a point (shown in blue).

58

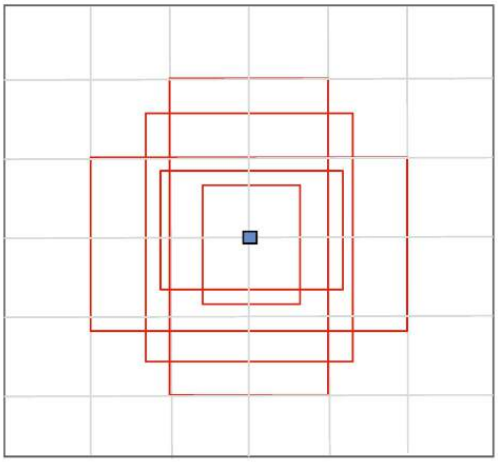


Figure 5.11. Anchor Boxes

In Single Shot Detector (SSD) network, multiple anchor boxes are selected at each location. These anchors serve as detectors, and the different sizes of these detectors enable the detection of objects of varying sizes. Smaller detectors can detect small objects, while larger detectors are capable of detecting larger objects.

— SSD does not use k-means to determine the anchor boxes, unlike some other object detection algorithms. Instead, it employs a mathematical formula to compute the anchor sizes. As a result, the sizes of SSD's anchor boxes are not dependent on the dataset used. In the SSD paper, these anchor boxes are referred to as "default boxes".
It is worth mentioning that in SSD (Single Shot Multibox Detector), anchors are predetermined and set as constants. At each convolution point, a set of fixed "default anchors" is placed. These anchors are associated with a set of default bounding boxes for multiple feature maps at the top of the network. They are laid out in a convolutional manner, tiling the feature map so that the position of each box relative to its corresponding cell remains fixed.

- **Model Architecture:** An SSD neural network consists of two components: **base network and prediction network.**
  1. **Base network:** The base network is a type of deep convolutional network that is used for feature extraction from input images. It is typically created by removing the fully connected layer of existing networks such as ResNet or VGG. In the case of SSD, the base network is truncated before any classification layer.
  2. **Detection network:** To the base network, attach some extra convolutional layers that will actually do the prediction of bounding boxes and object classes.

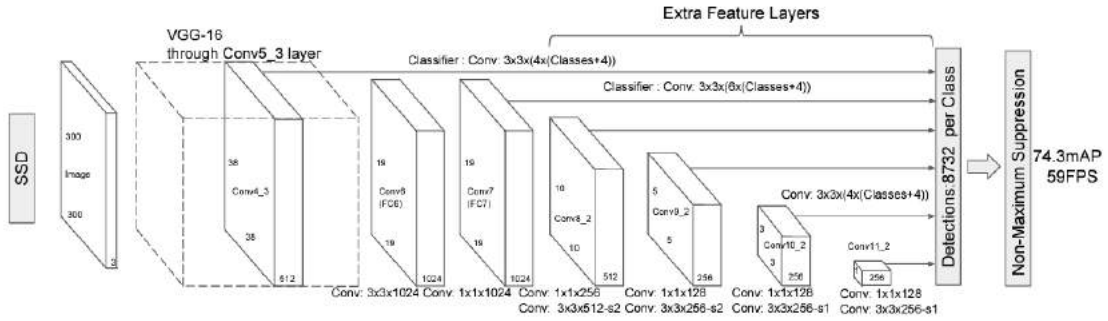


Figure 5.12. single shot detection models: SSD [46]

The detection network is characterized by the following features: [46]

— TThe layers in the network gradually decrease in size, enabling the detection of objects at multiple scales.

— Each feature layer in the network uses a different convolutional model for predicting detections.

— By adding a new feature layer (or using an existing layer from the base network), a fixed set of detection predictions can be generated using a set of convolutional filters

- **Matching System:** When training, we must choose default boxes that correspond to ground truth detection and train the network accordingly. For each ground truth box, we choose from default boxes that vary in location, aspect ratio, and scale. The SSD uses IoU (intersection over union) to match the default boxes with the ground truth. This is done by determining the overlap between them. The IoU-based overlap is also known as the Jaccard overlap. If the overlap between the default box and the ground truth is 0.5 or more, it is considered a match. This process of matching is repeated at each layer, which allows the network to learn at scale. Initially, the SSD uses the default boxes as predictions and then tries to regress and come closer to the ground truth bounding boxes. [10], [46]

5.5 Object Detection Evaluation Metrics

Object detection is a computer vision task where the goal is to identify and locate objects of interest within an image or a video. Several evaluation metrics are commonly used to assess the performance of object detection algorithms. The following are some of the detection metrics:

5.5.1 Intersection over Union (IoU)

Intersection over union (IoU), also known as the Jaccard index, is one of the most commonly used evaluation metrics in object detection algorithms. It is used to evaluate the performance of object detection by comparing the ground truth bounding box to the predicted bounding box and IoU. When it comes to object detection, we create training

sets by drawing bounding boxes around objects to label them. These bounding boxes are also known as ground truth in the training set. During the model learning process, the object detection algorithm predicts bounding boxes and then compares them with the ground truth. To determine the accuracy of the predicted bounding box, we use intersection over union (IoU) to evaluate to what extent the predicted bounding box overlaps with the ground truth.[10], [20], [46]

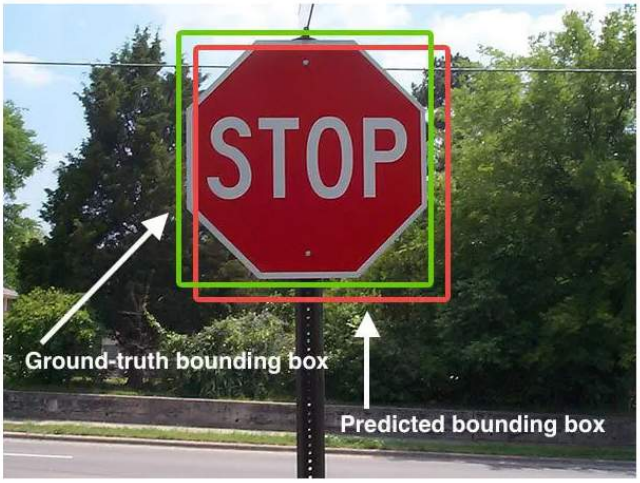


Figure 5.13. The predicted bounding box and Ground-truth bounding box [10]

Computing Intersection over Union can therefore be determined as:

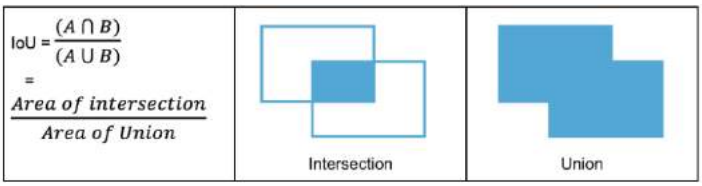


Figure 5.14. Intersection OverUnion

5.5.2 Mean Average Precision (mAP)

Mean Average Precision (mAP) is a frequently employed metric for evaluating the effectiveness of object identification and segmentation systems. Several object recognition techniques, including Faster R-CNN, MobileNet SSD, and YOLO, utilize mean Average Precision (mAP) as a metric to assess the performance of their models. The mean Average Precision (mAP) is utilized in various benchmark challenges, including Pascal, VOC, COCO, and others. The mean of Average Precision (AP) values are computed by averaging the AP values obtained for recall levels ranging from 0 to 1.[47], [10],
mAP formula is based on the following sub metrics:

- Confusion Matrix
- Intersection over Union(IoU)

61

- Recall,
- Precision

1. **Confusion Matrix:** A confusion matrix is a table that is often used to evaluate the performance of a classification algorithm on a set of labeled data for which the true values are known. It provides a summary of the classification results, breaking down the predicted and actual classes into four categories: true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). [47], [48]
   - **True Positives (TP):** This is when the model accurately predicts a label that matches the ground truth.
   - **True Negatives (TN):** This is when the model does not predict the label and is not part of the ground truth.
   - **False Positives (FP):** This is when the model predicts a label, but it is not a part of the ground truth (Type I Error).
   - **False Negatives (FN):** Finally, this is when the model does not predict a label, but it is part of the ground truth. (Type II Error).

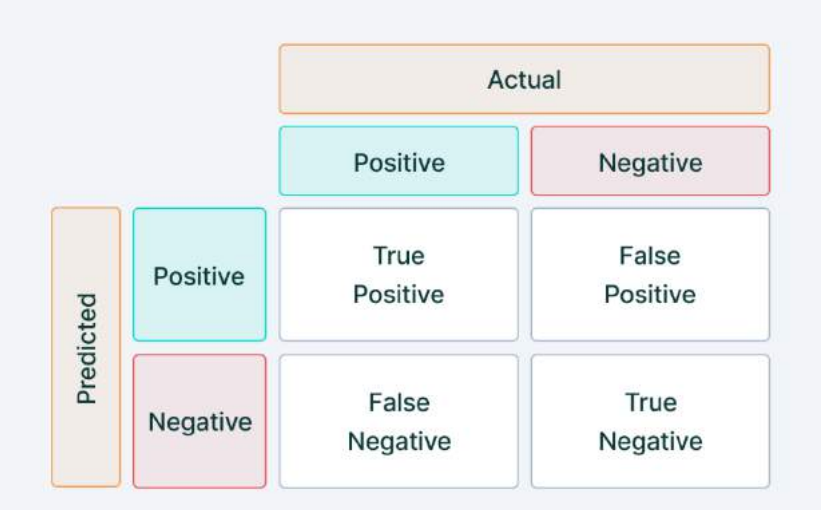


Figure 5.15. Confusion matrix table

2. **Precision:** Precision measures how well you can find true positives(TP) out of all positive predictions. (TP+FP).

$$Precision = \frac{TP}{TPTP + FP}$$

3. **Recall:** Recall measures how well you can find true positives(TP) out of all predictions(TP+FN).

$$Recall = \frac{TP}{TPTP + FN}$$

62

5.6 YOLO

YOLO is one of a real-time object detection algorithm that is aimed and designed to be fast and accurate. It uses a single convolutional neural network to simultaneously predict the bounding boxes along with class probabilities of objects in an image. What sets YOLO apart from other object detection algorithms is that it trains on the full image and is set up to solve regression problems. This means that it does not require a complex processing pipeline, which makes it extremely fast.YOLO is a system that simplifies object detection by treating it as a single regression problem. It predicts bounding box coordinates and class probabilities directly from image pixels. With YOLO, you can detect objects in an image with just one look and get information about what objects are present and where they are located. [49], [10]

5.6.1 Detection Algorithm

The YOLO design enables end-to-end training and real-time speeds while maintaining high average precision.
It divides the image into an SxS grid and for each grid cell predicts **B** bounding boxes, confidence for those boxes, and C-class probabilities. These predictions are encoded as a tensor.

$$SxSx(B*5+C)$$

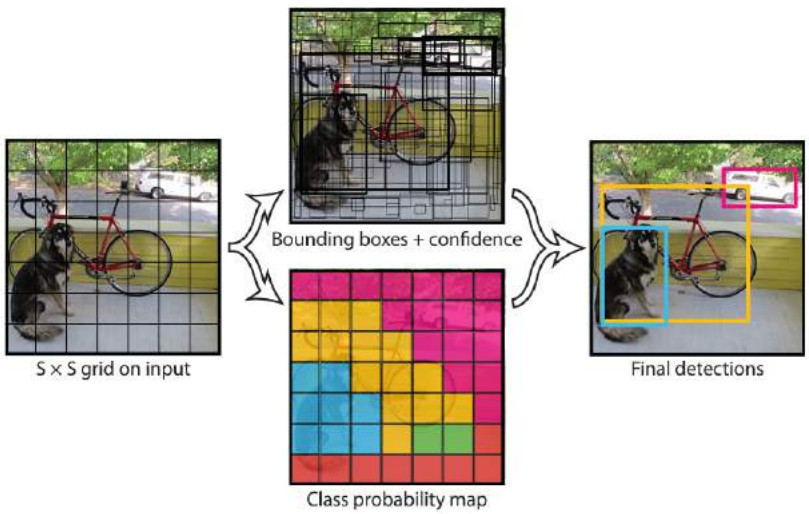


Figure 5.16. YOLO Model [49]

63

Each bounding box consists of 5 predictions: x, y, w, h, and confidence. confidence scores reflect how confident the model is.

$$Confidence = Pr(Object) * IOU_{pred}^{truth}$$

5.6.2 YOLO Architectural Design

The YOLO network architecture is inspired by GoogLeNet and is composed of 24 convolutional layers and 2 fully connected layers for image classification. The YOLO network uses 1x1 reduction layers followed by 3x3 convolutional layers, in contrast to the inception modules used in GoogLeNet. The full network is shown in Figure[5.17].

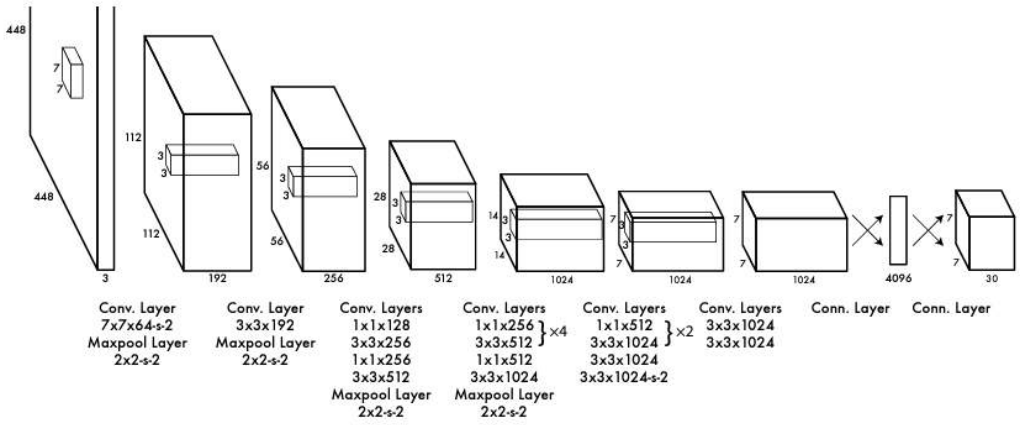


Figure 5.17. YOLO Model (Image source Original paper [49])

The convolutional layers were pre-trained on the ImageNet 1000-class competition dataset. Following an average-pooling layer and a fully connected layer, YOLO uses the first 20 convolutional layers from Figure 5.17 for pretraining. The final layer predicts both class probabilities and bounding box coordinates. A leaky rectified linear activation is used for all layers except the final layer, which uses a linear activation function.

$$\phi(x) = \begin{cases} x, & \text{if } x > 0 \\ 0.1x, & \text{otherwise} \end{cases}$$

5.6.3 Limitations of YOLO

- It struggles with small objects that come in groups, such as flocks of birds.
- It can predict only one class of objects within a cell grid.
- It does not predict well if the object has an unusual aspect ratio that was not seen in the training set.

- The accuracy of YOLO is lower than that of Faster R-CNN and some of the state-of-the-art algorithms.

5.7 Transfer Learning and Pre-trained Models:

Transfer learning is a technique in deep learning where a pre-trained model for a particular task is used as a starting point for another model that performs a similar task. This approach provides a faster and easier way to update and retrain a network as compared to training a network from scratch. It is commonly used in various applications, such as object detection, image recognition, and speech recognition. [50]

Transfer learning is a popular technique because:

- Using transfer learning, you can train models with less labeled data by utilizing popular models that have already been trained on large datasets.
- It can reduce training time and computing resources. With transfer learning, the weights are not learned from scratch because the pre-trained model has already learned the weights based on previous learning.
- You can take advantage of model architectures developed by the deep learning research community, including popular architectures such as GoogLeNet and ResNet.

Figure 5.18. Transfer Learning (Image source: [51])

5.7.1 Pre-trained Model

There are several pre-trained models available for object detection. Below are some of the most popular ones:

- Alexnet, googlenet(ImageNet), goolgenet(Places365), resnet18, resnet50, resnet101, vgg16, vgg19, inceptionv3, inceptionresnetv2, squeezenet, densenet201, mobilenetv2, shufflenet, xception, nasnetmobile, nasnetlarge.

When selecting a neural network, it's crucial to take into account its accuracy, speed, and size. Usually, there's a trade-off between these factors. To make a well-informed decision, you can refer to the graph below which compares the ImageNet validation accuracy with the time taken for prediction using the neural network. [48]

65

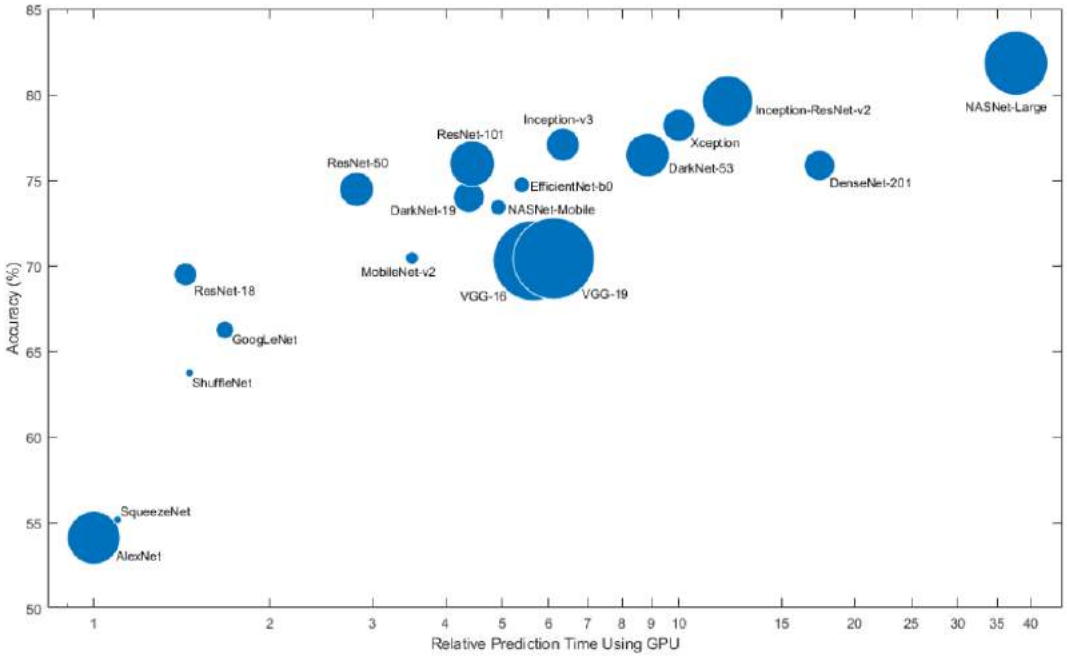


Figure 5.19. Compare Pretrained Neural Networks [52]

66

6. Implemented object detection algorithm and Evolution of YOLO model

This study aims to explore the performance of the YOLO-NAS and its variants on object detection tasks. Specifically, we evaluate the results of YOLO-NAS-S, YOLO-NAS-M, and YOLO-NAS-L on a dataset from the RoboFlow universe. RoboFlow provides various datasets that are structured accordingly, making it an ideal choice for this study. The dataset used in this study comprises 2481, 222, and 50 images for training, validation, and testing, respectively.

Figure 6.1. Image-1: from Training dataset

Figure 6.2. Image-1: from Training dataset

Figure 6.3. Image-1: from validation dataset

Figure 6.4. Image-1: from validation dataset

Figure 6.5. Sample image from dataset I am going to fine-tune

The image used to test the model is completely different from the dataset. Source: iStock

After fine-tuning and extensive training, we will test our models on a set of images obtained from iStock image stocks. The test dataset comprises a variety of images with different object sizes, orientations, and backgrounds. Some samples of our test dataset are provided in the figure below.

Figure 6.6. Car-1

Figure 6.7. Car-2

Figure 6.8. Car-3

Figure 6.9. Car-4

Figure 6.10. Sample image for testing the model after and before the training. This image is taken from Istock by Getty Images for testing the model only.

The image used to test the model is completely different from the dataset. Source: iStock

To train and test my models, I will utilize Google Colab, which provides powerful computers specifically designed for deep learning purposes. The machine I used for our experiments had an NVIDIA Tesla T4 GPU with high RAM, enabling us to train and test our models efficiently.

Our experiments involved training the models on the training dataset and evaluating their performance on the validation and test datasets. I will analyze the models' performance based on various metrics, including mean average precision (mAP), accuracy, and time taken for inference. I will also compare the performance of YOLO-NAS variants.

The results of our study will shed light on the performance of YOLO-NAS and its variants on object detection tasks and provide insights into their strengths and weaknesses. This study will contribute to the development of more efficient and accurate object detection

models that can be applied in various domains, including self-driving cars, surveillance systems, and robotics.

6.1 YOLO

YOLO (You Only Look Once) is an object detection system for real-time object detection. Ross Girshick, Ali Farhadi, Santosh Divvala, and Joseph Redmon were the ones who first introduced it in 2016. YOLO is known for its high speed and accuracy. The system is designed to classify and detect multiple objects in an image using a single forward pass of the neural network. Traditional object detection systems use a sliding window approach where they apply a classifier at each location and scale of the image. This process is computationally expensive and time-consuming. On the other hand, YOLO divides the input image into a grid of cells, and each cell is responsible for detecting an object. This approach reduces the number of bounding boxes that need to be processed and allows for real-time detection; this makes it suitable for a wide range of applications such as self-driving cars, surveillance systems, and robotics. Additionally, YOLO can detect objects at different scales and aspect ratios, making it robust to variations in object size and orientation. [53] [49]

6.2 The Evolution of YOLO

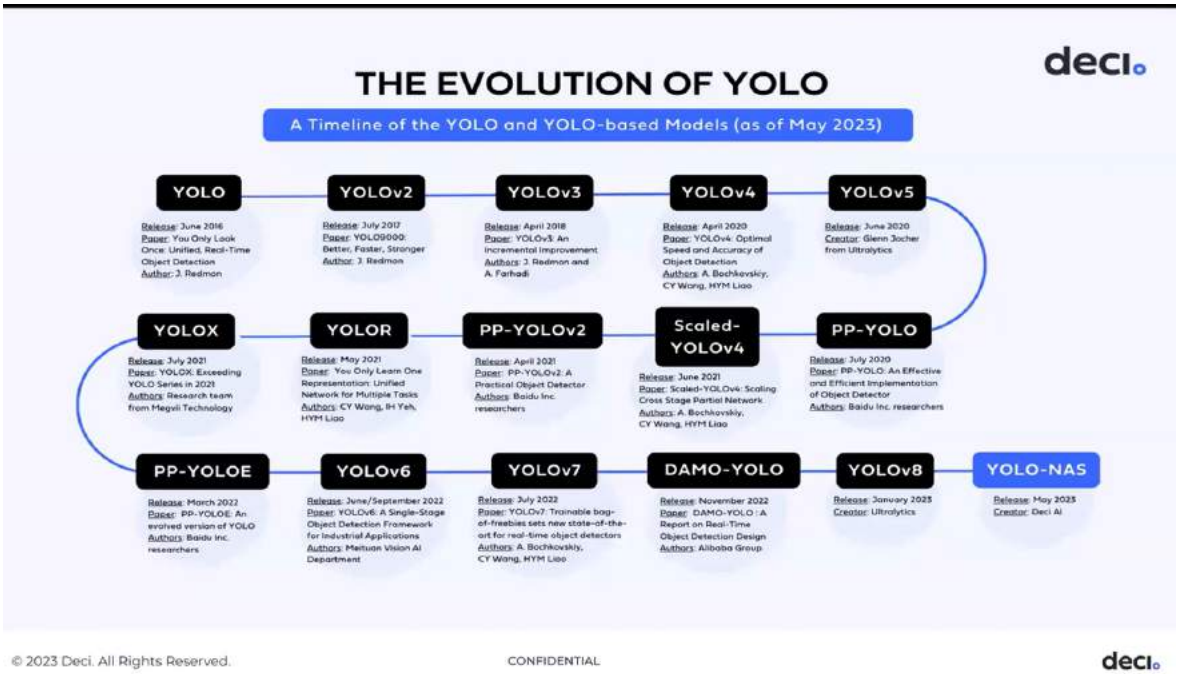


Figure 6.11. The evolution of yolo [Source: Deci. ai webinar]

1. **YOLO:** YOLO was introduced in 2016 for real-time object detection. Achieved a groundbreaking mAP of 63.4% on the PASCAL VOC2007 dataset. YOLO's efficiency in one network pass. Localization error limitations due to object count, aspect ratios,

and down-sampling. [49]

2. **YOLOv2 Improvements:** YOLOv2 was introduced in 2017 with 9000+ categories. Improved with batch normalization, high-resolution classifiers, and anchor boxes. Joint training for classification and detection.[54]

3. **YOLOv3 Advancements:** YOLOv3 (2018) achieved 60.6% mAP on MS COCO, 2x faster than previous versions. Introduced logistic regression for objectness scores and anchor box priors. [55]

4. **YOLOv4 Global Impact:** YOLOv4 (2020) maintained YOLO's philosophy with bag-of-freebies and bag-of-specials. Enhanced accuracy with image adjustments like mosaic augmentation and DropBlock. [56]
5. **YOLOv5 Speed and Efficiency:** YOLOv5 (2020) by Ultralytics in PyTorch achieved 50.7% AP with user-friendly features. [57]

6. **YOLOX Anchor-Free Innovation:** YOLOX (July 2021) exceeded previous YOLO versions, anchor-free with center sampling. Employed MixUP, Mosaic augmentations for improved accuracy. [58]

7. **YOLOR Multi-Task Learning:** YOLOR (May 2021) focused on a unified network for multiple tasks. Applied multi-task learning for classification, detection, and pose estimation.

8. **PP-YOLOv2:** Upgrades include ResNet101, PAN, Mish Activation, increased input size, and modifications to IoU-aware branch. [59]

9. **Scaled-YOLOv4 Flexibility:** Scaled-YOLOv4 (CVPR 2021) introduced scaling-up and scaling-down techniques. YOLOv4-tiny for low-end GPUs, YOLOv4-large for cloud GPUs. [60]

10. **PP-YOLO:** A YOLOv3-based model by Baidu, leveraging PaddlePaddle, introducing ten tricks for accuracy without sacrificing speed.[61]

11. **PP-YOLOE:** Evolved from PP-YOLOv2 with anchor-free architecture, new backbone, TAL, ET-head, and VFL/DFL. [62]

12. **YOLOv6 Industrial Framework:** YOLOv6 (September 2022) for industrial applications with anchor-free detection. YOLOv6-L achieved 52.5% AP and 70% AP50 at 50 FPS. [63], [53]
13. **YOLOv7 Speed and Accuracy:** YOLOv7 (July 2022) surpassed others in speed and accuracy. E-ELAN, model scaling, and "bag-of-freebies" approach for efficiency. [64],

[53]

14. **DAMO-YOLO Real-Time Improvements:** DAMO-YOLO (November 2022) by Alibaba for real-time object detection. Introduced MAE-NAS, Efficient-RepGFPN, ZeroHead, and knowledge distillation. [65]

15. **YOLOv8 :** YOLOv8 (January 2023) by Ultralytics with anchor-free prediction and faster NMS. Mosaic augmentation during training for improved accuracy. YOLOv8 offers five scaled versions: YOLOv8n (nano), YOLOv8s (small), YOLOv8m (medium), YOLOv8l (large), and YOLOv8x (extralarge). YOLOv8x outperformed YOLOv5 on the MS COCO dataset with an impressive 53.9% AP at 640 pixels and fast speed.[66], [53]

6.3 YOLO-NAS

YOLO-NAS is a cutting-edge object detection model created using Deci's Neural Architecture Search technology, AutoNAC™. It offers unmatched real-time object detection capabilities and production-ready performance, surpassing other models such as YOLOv5, YOLOv6, YOLOv7, and YOLOv8.

The YOLO-NAS model undergoes a multi-phase training process that involves pre-training on Objects365, COCO Pseudo-Labeled data, Knowledge Distillation (KD), and Distribution-Focal-Loss (DFL).

During the pre-training phase, the model is trained on Objects365, a comprehensive dataset that consists of 2 million images and 365 categories. Depending on the model variant, this process can take 25-40 epochs, as each epoch requires 50-80 minutes on 8 NVIDIA RTX GPUs.

The COCO dataset offers an additional 123,000 images without labels. These images are utilized to create pseudo-labeled data. First, an accurate model is trained on COCO to label these images. Then, the labeled images are used to train our model alongside the 118,000 train images.

The YOLO-NAS architecture incorporates Knowledge Distillation (KD) and Distribution Focal Loss (DFL) techniques to improve its training process. Furthermore, the YOLO-NAS training procedure employs various datasets, including both labeled and unlabeled data, as well as supervised and unsupervised training methods.[67], [53]

71

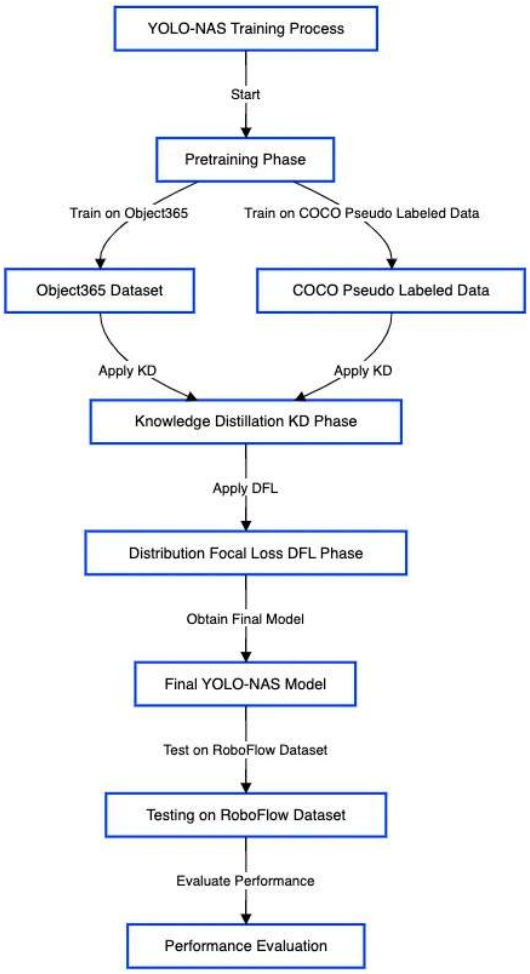


Figure 6.12. An overview of the YOLO-NAS training process [68]

6.3.1 Architectural Features of YOLO-NAS

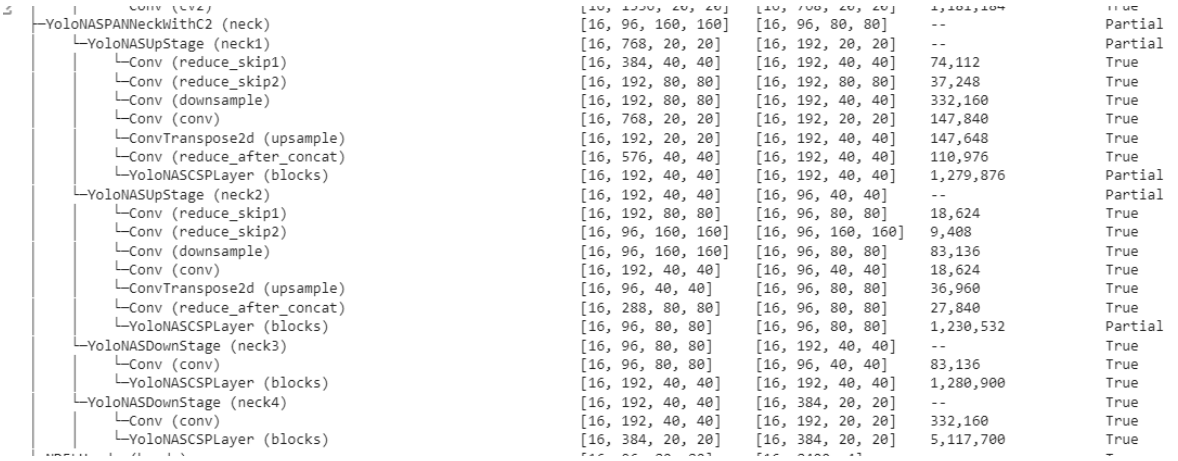
It is considered one of the most efficient object detection algorithms due to its unique architecture, which is designed to minimize the computational cost of the algorithm while maintaining high accuracy.

1. **Quantization Aware Blocks and Selective Quantization:**
    - **Hybrid Quantization Method:** YOLO-NAS uses a mixed quantization method that includes both Quantization-Specific Parameters (QSP) and Quantization-Centric Initialization (QCI) blocks. These leverage re-parameterization and 8-bit quantization, inspired by Chu et al.'s methodology.
    - **Selective Quantization:** The model strategically quantizes specific parts rather than uniformly affecting all layers. This selective quantization balances accuracy and latency, addressing information loss, a common issue in standard quantization techniques.
    - **Layer Selection Algorithm:** YOLO-NAS utilizes a sophisticated layer selection algorithm to decide which layers to quantize. It evaluates the impact of each

layer on accuracy and latency, carefully considering the implications of toggling between 8-bit and 16-bit quantization.

- **Performance in Limited Resources:** Uniquely designed for environments with limited resources, YOLO-NAS ensures superior performance. Its optimized approach to quantization preserves accuracy while being efficient, striking a crucial balance for advanced object detection tasks.

2. **Detection Head:** A standout feature of YOLO-NAS is its detection head design, which predicts a distribution probability for size regression. This approach is particularly useful in scenarios where the object sizes vary significantly. YOLO-NAS is an object detection model that predicts a range of possible sizes instead of a single fixed size. This unique approach improves its accuracy in detecting objects of varying scales. In addition, YOLO-NAS uses a probabilistic approach to size regression which makes it suitable for knowledge distillation. This means that it facilitates a more nuanced transfer of knowledge from a complex, high-capacity teacher model to a simpler, more efficient student model. All these features make YOLO-NAS a practical choice for diverse applications.
3. **Neck:** The YOLO-NAS network has a highly advanced pyramid-attention neck that combines both top-down and bottom-up information flows.



├─YoloNASPANNeckWithC2 (neck)	[16, 96, 160, 160]	[16, 96, 80, 80]	--	Partial
│ └─YoloNASUpStage (neck1)	[16, 768, 20, 20]	[16, 192, 20, 20]	--	Partial
│ │ └─Conv (reduce_skip1)	[16, 384, 40, 40]	[16, 192, 40, 40]	74,112	True
│ │ └─Conv (reduce_skip2)	[16, 192, 80, 80]	[16, 192, 80, 80]	37,248	True
│ │ └─Conv (downsample)	[16, 192, 80, 80]	[16, 192, 40, 40]	332,160	True
│ │ └─Conv (conv)	[16, 768, 20, 20]	[16, 192, 20, 20]	147,840	True
│ │ └─ConvTranspose2d (upsample)	[16, 192, 20, 20]	[16, 192, 40, 40]	147,648	True
│ │ └─Conv (reduce_after_concat)	[16, 576, 40, 40]	[16, 192, 40, 40]	110,976	True
│ │ └─YoloNASCSPLayer (blocks)	[16, 192, 40, 40]	[16, 192, 40, 40]	1,279,876	Partial
│ └─YoloNASUpStage (neck2)	[16, 192, 40, 40]	[16, 96, 40, 40]	--	Partial
│ │ └─Conv (reduce_skip1)	[16, 192, 80, 80]	[16, 96, 80, 80]	18,624	True
│ │ └─Conv (reduce_skip2)	[16, 96, 160, 160]	[16, 96, 160, 160]	9,408	True
│ │ └─Conv (downsample)	[16, 96, 160, 160]	[16, 96, 80, 80]	83,136	True
│ │ └─Conv (conv)	[16, 192, 40, 40]	[16, 96, 40, 40]	18,624	True
│ │ └─ConvTranspose2d (upsample)	[16, 96, 40, 40]	[16, 96, 80, 80]	36,960	True
│ │ └─Conv (reduce_after_concat)	[16, 288, 80, 80]	[16, 96, 80, 80]	27,840	True
│ │ └─YoloNASCSPLayer (blocks)	[16, 96, 80, 80]	[16, 96, 80, 80]	1,230,532	Partial
│ └─YoloNASDownStage (neck3)	[16, 96, 80, 80]	[16, 192, 40, 40]	--	True
│ │ └─Conv (conv)	[16, 96, 80, 80]	[16, 96, 40, 40]	83,136	True
│ │ └─YoloNASCSPLayer (blocks)	[16, 192, 40, 40]	[16, 192, 40, 40]	1,280,900	True
│ └─YoloNASDownStage (neck4)	[16, 192, 40, 40]	[16, 384, 20, 20]	--	True
│ │ └─Conv (conv)	[16, 192, 40, 40]	[16, 192, 20, 20]	332,160	True
│ │ └─YoloNASCSPLayer (blocks)	[16, 384, 20, 20]	[16, 384, 20, 20]	5,117,700	True

Figure 6.13. Neck Design

This design element assists the network in effectively capturing and utilizing multi-scale information. In this pyramid-attention structure, the top-down pathway captures high-level semantic information, while the bottom-up pathway focuses on smaller details. The attention mechanism in this pyramid structure ensures that the network focuses on the most relevant features at different scales, which improves its ability to detect objects with varying sizes and complexities.

4. **Backbone:** The backbone of YOLO-NAS is a significant component of its architecture, which is the result of an advanced Network Architecture Search (NAS) process. The process utilizes Deci's proprietary NAS technology, AutoNAC, to tailor the network structure for object detection tasks with unparalleled precision. This innovative

73

approach ensures that the network structure meets the specific demands of the tasks at hand.

- **quantization-aware RepVGG:** During the NAS process, they have incorporated quantization-aware RepVGG blocks into the model architecture, ensuring that our model architecture would be compatible with Post-Training Quantization (PTQ).

- **Spatial Pyramid Pooling (SPP):** YOLO-NAS backbone has Spatial Pyramid Pooling (SPP) at the end to capture global context; it is a pooling layer that removes the fixed-size constraint of the network, i.e., a CNN does not necessitate a fixed or predetermined picture size input. Placing SPP at the end of the YOLO-NAS backbone allows the model to capture global context information, which is crucial for understanding the overall context of the scene. This can be beneficial for detecting objects that may span a larger region in the image.[67]

Source

```
from torchinfo import summary

summary(model = model,
        input_size = (16,3,640,640),
        col_names = ['input_size',
                     'output_size',
                     'num_params',
                     'trainable'],
        col_width = 20,
        row_settings = ['var_names'])
```

```
Collecting torchinfo
  Downloading torchinfo-1.8.0-py3-none-any.whl (23 kB)
Installing collected packages: torchinfo
Successfully installed torchinfo-1.8.0
```

Layer (type (var_name))	Input Shape	Output Shape	Param #	Trainable
YoloNAS_L (YoloNAS_L)	[16, 3, 640, 640]	[16, 8400, 4]	--	Partial
├─NStageBackbone (backbone)	[16, 3, 640, 640]	[16, 96, 160, 160]	--	Partial
│ └─YoloNASStem (stem)	[16, 3, 640, 640]	[16, 48, 320, 320]	--	False
│ │ └─QARepVGGBlock (conv)	[16, 3, 640, 640]	[16, 48, 320, 320]	(1,344)	False
│ └─YoloNASStage (stage1)	[16, 48, 320, 320]	[16, 96, 160, 160]	--	Partial
│ │ └─QARepVGGBlock (downsample)	[16, 48, 320, 320]	[16, 96, 160, 160]	(41,568)	False
│ │ └─YoloNASCSPLayer (blocks)	[16, 96, 160, 160]	[16, 96, 160, 160]	388,034	Partial
│ └─YoloNASStage (stage2)	[16, 96, 160, 160]	[16, 192, 80, 80]	--	Partial
│ │ └─QARepVGGBlock (downsample)	[16, 96, 160, 160]	[16, 192, 80, 80]	(166,080)	False
│ │ └─YoloNASCSPLayer (blocks)	[16, 192, 80, 80]	[16, 192, 80, 80]	1,058,435	Partial
│ └─YoloNASStage (stage3)	[16, 192, 80, 80]	[16, 384, 40, 40]	--	Partial
│ │ └─QARepVGGBlock (downsample)	[16, 192, 80, 80]	[16, 384, 40, 40]	(663,936)	False
│ │ └─YoloNASCSPLayer (blocks)	[16, 384, 40, 40]	[16, 384, 40, 40]	6,787,333	Partial
│ └─YoloNASStage (stage4)	[16, 384, 40, 40]	[16, 768, 20, 20]	--	Partial
│ │ └─QARepVGGBlock (downsample)	[16, 384, 40, 40]	[16, 768, 20, 20]	(2,654,976)	False
│ │ └─YoloNASCSPLayer (blocks)	[16, 768, 20, 20]	[16, 768, 20, 20]	11,802,114	Partial
│ └─SPP (context_module)	[16, 768, 20, 20]	[16, 768, 20, 20]	--	True
│ │ └─Conv (cv1)	[16, 768, 20, 20]	[16, 384, 20, 20]	295,680	True
│ │ └─ModuleList (m)	--	--	--	--
│ │ └─Conv (cv2)	[16, 1536, 20, 20]	[16, 768, 20, 20]	1,181,184	True

Figure 6.14. Backbone Structure

AutoNAC plays a crucial role in the NAS process by determining the optimal sizes and structures of different stages in the YOLO-NAS backbone. This involves configuring the block type, number of blocks, and channels in each stage meticulously. Consequently, AutoNAC guarantees that every part of the backbone is meticulously optimized for its specific function. The NAS-generated backbone is not just a result of automated design but also of intelligent decision-making that considers various factors such as computational efficiency, accuracy, and speed. This ensures that the backbone contributes effectively to the overall robustness and adaptability of

74

YOLO-NAS. These advancements lead to a superior architecture with exceptional object detection capabilities and outstanding performance compared to its predecessors.

6.3.2 Automated Neural Architecture Construction (AutoNAC) engine

Deci created the optimization engine AutoNAC, which Deci-AI uses. This engine applies Neural Architecture Search (NAS) to enhance the architecture of a deep learning model. The main goal is to improve the performance of the model when it is executed on specific hardware while maintaining or even improving its accuracy. The AutoNAC engine is hardware aware, data-aware and considers all the components in the inference stack, including compilers and quantization.
AutoNAC is essentially a NAS engine takes three components as inputs. [69]

- **Task:** Firstly, we need to decide which model to build, such as an object detection model.

- **Data characteristics:** Deci Group requires the NAS Engine to be data-aware or capable of understanding the characteristics of the data so that it can create a model that is well-suited for the task at hand. For instance, if the objective is to detect small objects, the model would likely be different from that required for detecting large objects, as the receptive fields would differ.

- **Inference environment and Hardware:** When it comes to computer vision on edge, the priority is to achieve real-time performance or improve some level of latency or throughput while being hardware-aware, compilation-aware, and quantization-aware. To achieve this, we need to consider factors such as model size, latency, and throughput. Decis Group's goal is to optimize these factors as part of the Neural Architecture Search (NAS) process. Manually finding the optimal architecture for object detection models can be a laborious and inefficient process. To address this issue, Deci utilized AutoNAC, a tool that uses neural design space incorporating state-of-the-art (SOTA) architectural design principles and Deci's novel neural elements, to discover novel object detection models. These models were optimized to minimize inference latency computed over NVIDIA's T4 cloud GPU, which is a widely used computing device. We achieve this by feeding all three inputs into the AutoNAC engine, which generates a new architecture.

75

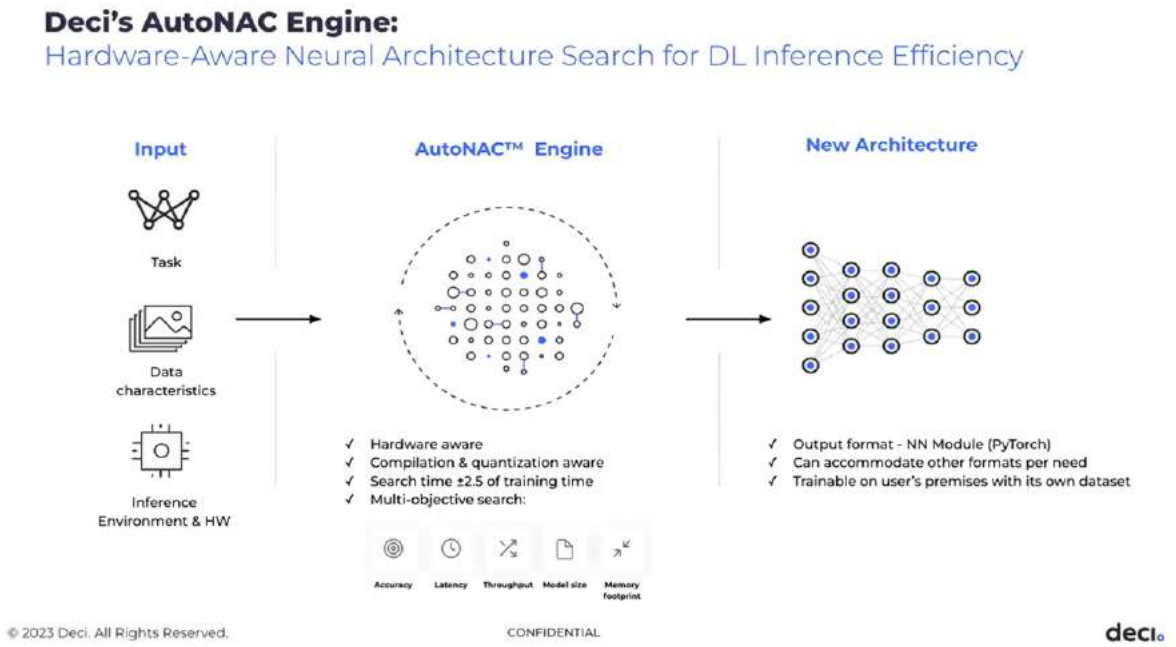


Figure 6.15. Deci's AutoNAC Engine;
Hardware-Aware Neural Architecture Search for DL Inference Efficiency [52][70]

6.3.3 Under the hood View of Auto-NAC

NAS algorithms can methodically search through the vast space of potential architectures, effectively locating novel and optimized configurations that human intuition might miss. By automating the process, these algorithms can quickly evaluate and compare an unimaginably large number of candidate architectures. They will eventually find a solution that strikes the best balance between accuracy, speed, and complexity. The NAS engine will be used when we already have data and want to build a model. it takes all three inputs mentioned above, and the search space is automatically created under the hood, considering all the possibilities of neural architecture as an input. For example, optimal sizes, block types, number of blocks, and channel counts in every stage. [69] [52]

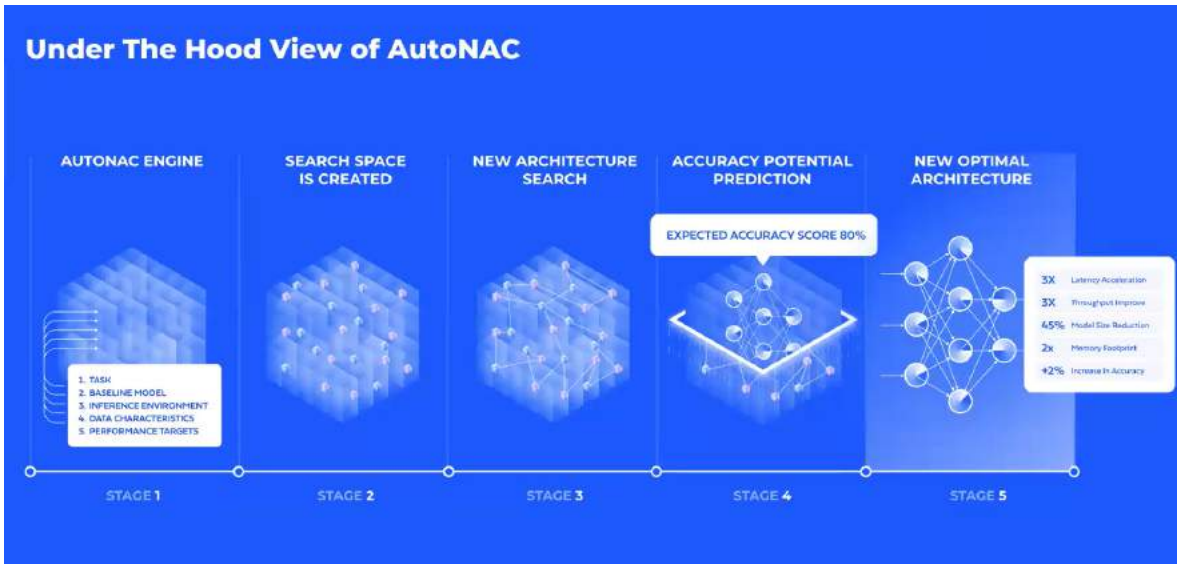


Figure 6.16. AutoNAC Engine Search Space [69]

Finally, AutoNAC Engine explores and maps the efficiency frontier, searching for an architecture that best balances latency vs. throughput. Deci Group samples three points of this frontier to create the YOLO-NASS, YOLO-NASM, and YOLO-NASL architectures.

6.3.4 How to find the best architecture out of the search space

while the search space has trillions of candidate architectures, the task of identifying the optimal architecture becomes increasingly challenging. Within the realm of Deci, the selection of the most suitable candidate from the numerous models within the extensive search space is guided by two distinct evaluation criteria. [69]

1. **Bench Marking:** Understanding how the model will behave on the expected hardware or using the techniques of Latency Vs. throughput for the candidate architecture.

2. **Accuracy Potential Predictions:** The fundamental selection methodology of best candidate out of the search space involves the provision of an AI model, designated to assess the accuracy or potential accuracy of neural network models. Leveraging principles from neural architecture search, YOLO-NAS systematically navigates the expansive design space of possible neural network configurations, employing optimization algorithms to identify architectures anticipated to yield high accuracy on the specified object detection task.

77

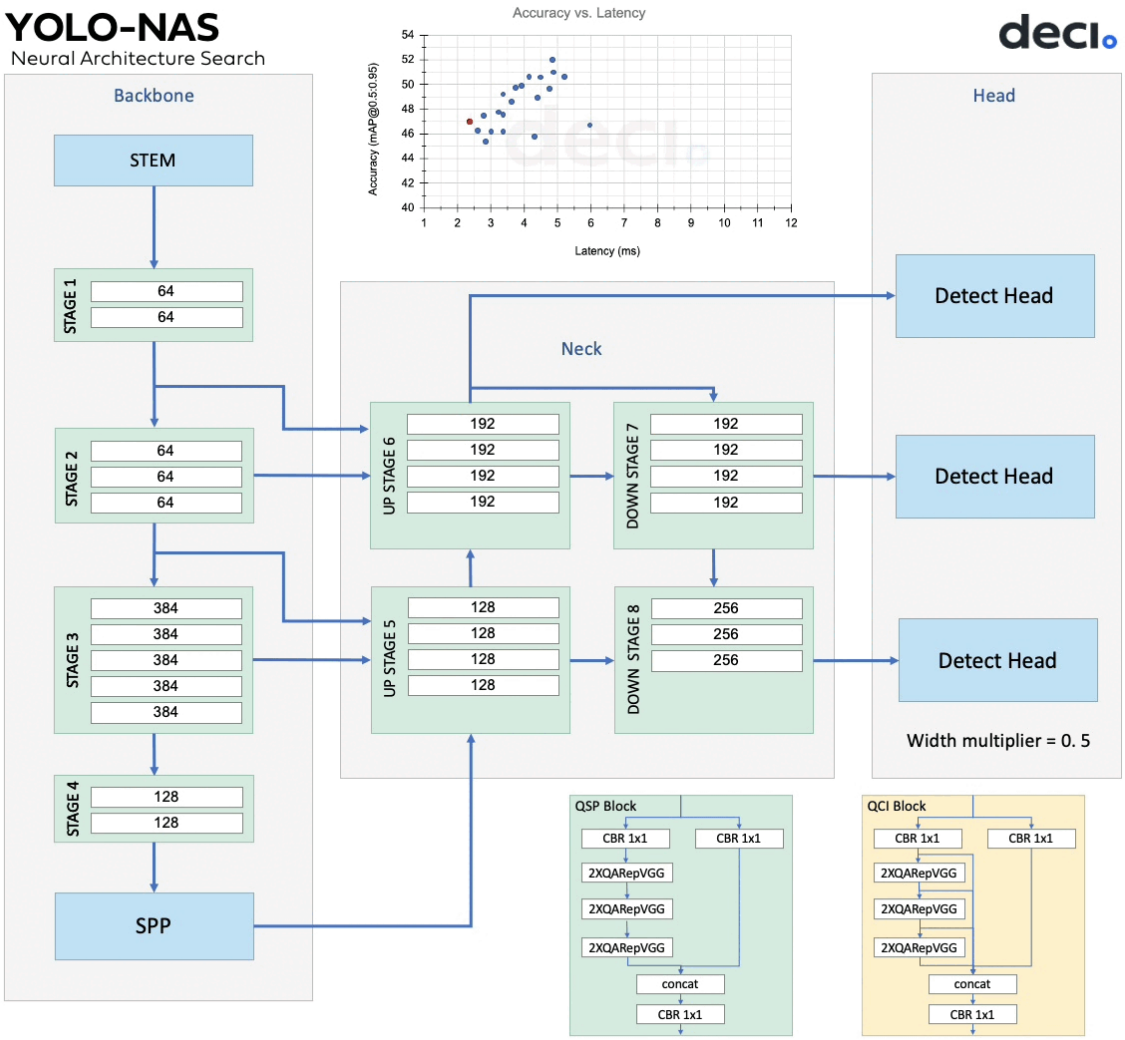


Figure 6.17. YOLO-NAS Architecture [67]. The architecture is found automatically via a Neural Architecture Search(NAS) system called AutoNAC to balance latency vs. throughput. They generated three architectures called YOLO-NASS(small), YOLO-NASM(medium), and YOLO-NASL(large), varying the depth and the position of the QSP and QCI blocks [67]

6.3.5 Fine-tuning of the selected model On custom data set

Fine-tuning refers to the process of making minor adjustments to achieve the desired output or performance. In deep learning, pre-existing trained neural networks can be used to program another deep learning algorithm from the same domain by using their weights. These weights are responsible for connecting each neuron in one layer to every neuron in the next layer of the neural network. Fine-tuning can be used to speed up the training process and overcome a small dataset as it already contains vital information from the pre-existing deep learning algorithm. The YOLO-NAS architecture and pre-trained weights are an excellent starting point for fine-tuning downstream tasks and define a new frontier in low-latency inference. When fine-tuning, using strong pre-trained weights often leads to higher model accuracy on new datasets. YOLO-NAS was trained on the RoboFlow100

78

dataset, which is a collection of 100 datasets from diverse domains, to demonstrate its ability to handle complex object detection tasks. The RF100 dataset is a benchmark for existing YOLO models, enabling us to compare YOLO-NAS's performance against them and showcase its advantages.[67]

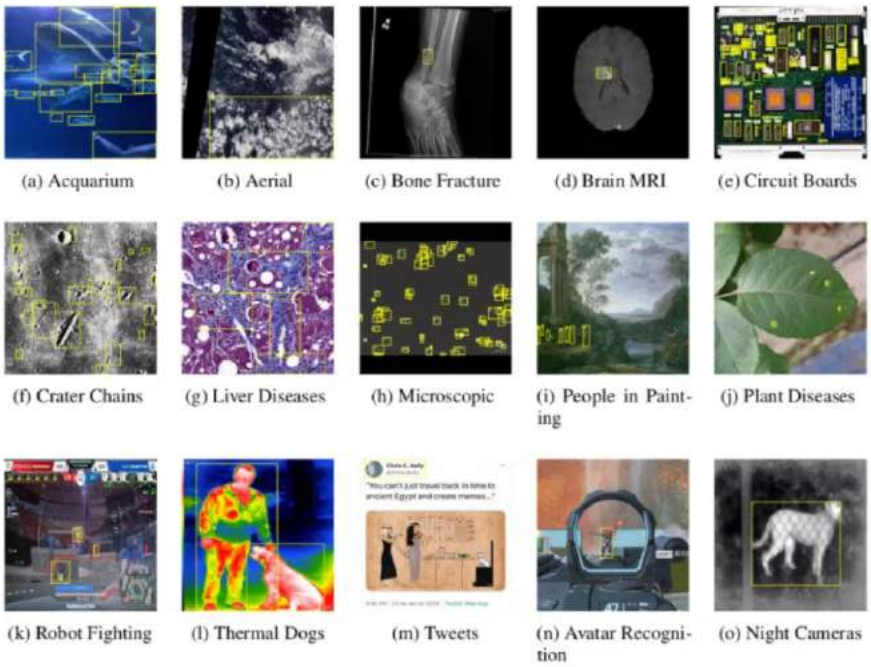


Figure 6.18. Examples of annotated images in the RF100 benchmark [67]

The following hyperparameters ensure a robust and consistent training process, allowing for a fair comparison of the model's performance across different datasets.
Deci followed the RF100 repository's training protocol to ensure a fair comparison.
The model was trained for 100 epochs on a single T4 GPU with 16GB of VRAM, using consistent settings across all datasets.[67], [71]

- Learning rate: a learning rate of 5e-4 is used, while for the Medium version, the learning rate is set to 4e-4.
- Weight decay: 1e-4 (excluding bias and BatchNorm layers)
- Exponential moving average (EMA) with a decay factor of 0.99
- Batch size: 16
- Image resolution: 640×640

79

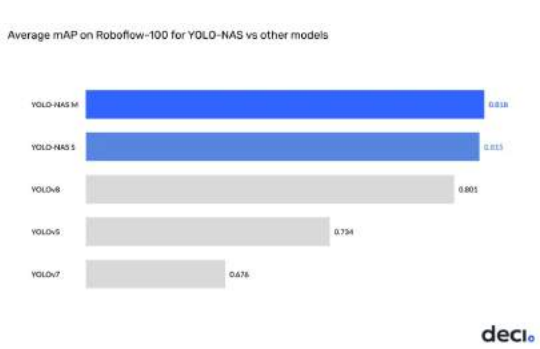


Figure 6.19. Average mAP on Roboflow-100 for YOLO-NAS vs other models.

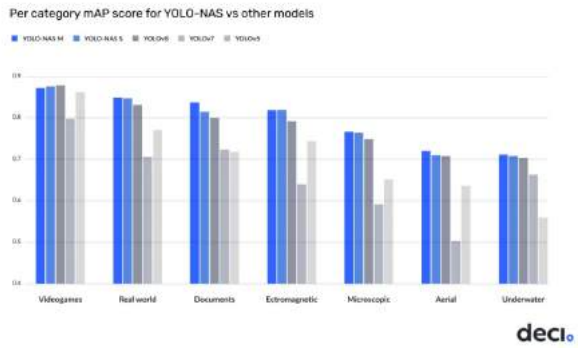


Figure 6.20. Per category mAP score for YOLO-NAS vs other models. Note: For Yolo vV5/v7/v8

Figure 6.21. Average mAP and Per category mAP score for YOLO-NAS vs other models. [67]

Figure 6.19 are the results obtained by focusing on the "Small" and "Medium" YOLO-NAS variants, and figure: 6.20 is a per-category breakdown of YOLO-NAS's performance on the RF-100 dataset, compared to the performance of v5/v7/v8 models. [67], [70]

6.3.6 Object Detection Evaluation Metrics

Object detection metrics are measurements that are used to assess the effectiveness of object detection algorithms. These metrics are essential in determining the accuracy and efficiency of object detection models, as they provide insights into the model's ability to identify and locate objects within images. Furthermore, they help in identifying false positives and false negatives. This information is critical in evaluating and improving the model's performance. There are many evaluation metrics that are applicable across different object detection algorithms, including the YOLO-NAS model. [72]

- **Intersection over Union (IoU):** Intersection over Union (IoU) is a fundamental measure that quantifies the overlap between a predicted bounding box and a ground truth bounding box. It plays a crucial role in evaluating the accuracy of object localization.
- **Average Precision (AP):** AP calculates the area under the precision-recall curve, producing a single value that encompasses precision and recall performance.
- **Mean Average Precision (mAP):** mAP calculates the average AP values across multiple object classes, making it useful in multi-class object detection scenarios for comprehensive model performance evaluation.
- **Precision and Recall:** Precision and Recall are two important metrics used to evaluate the performance of classification models. Precision measures the percentage of true positives among all positive predictions, indicating the model's ability to avoid false positives. Recall, on the other hand, calculates the percentage of true positives among all actual positives, indicating the model's ability to detect all instances of a class. These metrics are often used together to provide a more complete picture of a model's performance.

80

- F1 Score: The F1 Score is a measure of a model's performance that takes into account both precision and recall. It provides a balanced assessment while considering false positives and negatives.

6.3.7 Benefits of YOLO-NAS over other models

Optimized Efficiency:
YOLO-NAS is an advanced model that has been designed to achieve an optimal balance between accuracy and speed, making it more efficient than other human-designed models. This optimization is essential for real-time object detection applications as it enhances inference speeds and improves resource utilization. YOLO-NAS outperforms other models in terms of efficiency, making it the go-to choice for those who demand high-performance object detection capabilities."
Adaptability to Diverse Tasks and Hardware:
The architecture of YOLO-NAS allows it to adapt to various object detection tasks, including complex scenarios and hardware. Its versatility makes it a valuable tool across a wide range of applications.
Training on Prominent Datasets:
The architecture of YOLO-NAS allows it to adapt to various object detection tasks, including complex scenarios and hardware. Its versatility makes it a valuable tool across a wide range of applications.
Detection and Localization Accuracy:
The YOLO-NAS architecture is a specialized object detection model that addresses the challenges associated with detecting small or intricate objects. Its heightened detection capabilities and improved accuracy in pinpointing precise locations make it a superior choice for a diverse range of applications, particularly those where identifying small or elusive objects is crucial. In contrast, the YOLOv8 object detection model, while impressive, is limited in its ability to detect small objects accurately and localize them with precision. Despite its proficiency in various scenarios, YOLOv8 falls short when compared to the specialized capabilities of YOLO-NAS in handling the challenges posed by diminutive objects. [71]

81

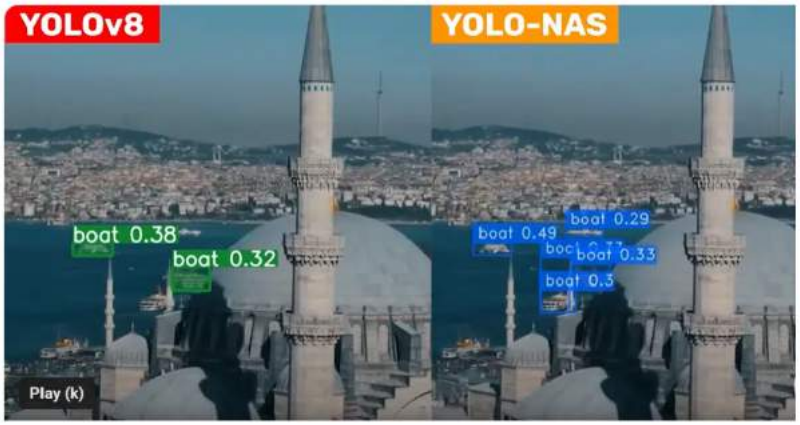


Figure 6.22. Performance evaluation on YOLO-V8 vs. YOLO-NAS (Image Source: Youtube)

6.3.8 Data set

The "Smart City Cars Detection Computer Vision Project" stands as a meticulously curated dataset within the Roboflow Universe, purposefully designed for object detection, with a specific emphasis on vehicular recognition. Encompassing various vehicle types, including buses, cars, motorbikes, and trucks this dataset plays a crucial role in advancing computer vision applications. Regarding key attributes, the project adopts an object detection framework with distinct classes such as bus, car, motorbike, and truck. The dataset comprises 2753 after preprocessing applied from 1099 original images, with data allocation distributed as 90% for training, 8%for validation, and 2% for testing.

During preprocessing, the image is resized to fit within a 640x640 frame, with black edges if necessary. Augmentations include rotating between -15° and +15°, shearing horizontally and vertically by ±15°, adjusting hue between -25° and +25°, saturating between -40% and +40%, applying blur up to 4 pixels, introducing noise up to 3% of pixels, and utilizing mosaic augmentation. For bounding boxes, cropping ranges from 0% minimum zoom to 20% maximum zoom, and brightness is adjusted between -30% and +30%. The model generates three outputs per training example.

This dataset holds substantial utilitarian significance across various domains. It facilitates real-time traffic monitoring and analysis, contributing to Employing the Smart City Cars Detection model to process live video feeds of traffic, recognize various vehicle types and their numbers, and adjust traffic light timings dynamically. This optimization aims to reduce congestion and enhance the overall traffic flow within a smart city environment.

Additionally, it Utilizes the Smart City Cars Detection model for monitoring parking areas, distinguishing between different vehicle classes occupying spaces (cars, trucks, buses, motorbikes), and offers drivers real-time updates on parking space availability through smart parking applications, Integrate findings from the Smart City Cars Detection model to analyze the distribution of diverse vehicle types in specific locations or across the entire city. This information aids city planners in devising focused infrastructure development plans, such as dedicated bus lanes, motorcycle parking, or optimized truck routes,

Input the Smart City Cars Detection model data into machine learning algorithms for the autonomous generation of precise forecasts regarding future traffic patterns. This includes predicting peak hours and identifying locations prone to congestion, enabling proactive measures to be implemented. Mostly, Employ the Smart City Cars Detection model to observe bus utilization within a city, recognizing both bus occupancy and frequency. Utilize this information to enhance public transportation routes, determining whether additional buses or different types of public transport are required for more efficient service to the population.

The "Smart City Cars Detection Image Dataset" dataset epitomizes substantive potential across domains, ranging from traffic management and security to autonomous vehicles, fleet management, and traffic accident analysis.[73]

83

7. Object detection Implementations and fine-tuning on custom dataset

1. **Environment Setup:** Google CoLab is a free research environment provided by $Google^{Tm}$. Using Google Colab for training deep learning models, including YOLO-NAS, provides you with access to free GPU resources. Before we start training, we need to prepare our Python environment and start by installing the required packages. The YOLO-NAS model is distributed using a $super-gradients$ package. In addition, we will install $roboflow$ and $supervision$. Roboflow will enable us to download the dataset from the Roboflow Universe, while Supervision will allow us to visualize the outcomes of our training. We use `!pip install` to install the required package. [74]

Listing 1. Installing the required package

```
!pip install super-gradients
!pip install roboflow
!pip install supervision
```

2. **Load YOLO-NAS Model** I have automated the selection of the model. To support the open-source community, the Deci group has released the capability to fine-tune YOLO-NAS models on the RF100 dataset within the SuperGradients library. To perform inference using the pre-trained model, we first need to choose the model size. YOLO-NAS offers three different model sizes: `yolo_nas_s`, `yolo_nas_m`, and `yolo_nas_l`.

Listing 2. importing models from the supergradients

```
#[Original Version]
from super_gradients.training import models

model = models.get(yolo_nas_l, pretrained_weights="coco").to(DEVICE)

#[The Modified Version]
import torch #gives us access to some helpful neural network things,
such as various neural network layer types

DEVICE = 'cuda' if torch.cuda.is_available() else "cpu"  #Check if
GPU is available

MODEL_ARCH_OPTIONS = ['yolo_nas_l', 'yolo_nas_m', 'yolo_nas_s']

# Display options to the user
print("Choose a model architecture:")
for i, MODEL_ARCH in enumerate(MODEL_ARCH_OPTIONS, start=1):
```

```
        print(f"{i}. {MODEL_ARCH}")

    # Take input from the user
    while True:
        try:
            user_input = int(input("Enter your choice number: "))
            if 1 <= user_input <= len(MODEL_ARCH_OPTIONS):
                selected_model_arch = MODEL_ARCH_OPTIONS[user_input - 1]
                print(f"Selected Model Architecture: {selected_model_arch}")
                break
            else:
                print("Invalid input. Please enter a valid number.")
        except ValueError:
            print("Invalid input. Please enter a valid number.")
```

3. **Dataset Download and Directory Structure:** To perform model fine-tuning, the acquisition of pertinent data is imperative. The requisite dataset structure for this endeavor adheres to the YOLO format. I am utilizing a dataset from the Roboflow universe, encompassing a vast repository of over 100,000 open-source computer vision datasets. Significantly, the dataset obtained from Roboflow is inherently organized to align with the prescribed folder structure mandated by YOLO standards.

Listing 3. Dataset folder structure YOLO format

```
/directory_to_your_dataset
|-- train
|   |-- images
|   |   |-- image1.jpg
|   |   |-- image2.jpg
|   |   |-- ...
|   |-- labels
|       |-- image1.txt
|       |-- image2.txt
|       |-- ...
|-- valid
|   |-- images
|   |-- labels
|-- test
|   |-- images
|   |-- labels
```

Each image file should have a corresponding label file in the YOLO format, which contains one line per object in the image. Each line should have the following format: `<class_id> <x_center> <y_center> <width> <height>`

where <class_id> is an integer representing the class of the object, and <x_center>, <y_center>, <width>, and <height> are the normalized coordinates of the bounding box. In the process of importing data from the Roboflow universe, the inclusion of a data.yaml file is an automated or default feature. This file serves as a dictionary, encapsulating the names of classes associated with the imported data. The order of the class names should match the <class_id> in the label files.

4. **Dataset Parameters for Training YOLO NAS** After preparing your data, you need to create a Python dictionary that outlines the key elements of your dataset's structure, as such:

```
    dataset_params = {
    'data_dir': LOCATION,
    'train_images_dir': 'train/images',
    'train_labels_dir': 'train/labels',
    'val_images_dir': 'valid/images',
    'val_labels_dir': 'valid/labels',
    'test_images_dir': 'test/images',
    'test_labels_dir': 'test/labels',
    'classes': CLASSES
}
```

Using the dataset_params we can easily create the datasets in the required format later on, Where LOCATION is the path to the main directory where your dataset resides, and CLASSES is the list of class names you created earlier. It's important to specify the number of training epochs, batch size, and data processing workers.

```
EPOCHS = 60,
BATCH_SIZE = 16,
WORKERS = 8
```

5. **Setting-up dataloaders:** SuperGradients provides ready-to-use dataloaders for the datasets it supports. Import the required modules for SuperGradients dataloaders.[74]

Listing 4. Setting-Up Dataloaders

```
from super_gradients.training import dataloaders
from super_gradients.training.dataloaders import
(coco_detection_yolo_format_train, coco_detection_yolo_format_val)
```

For training, validation, and testing sets, create data loaders with a given batch size and workers Below we use batch_size=16 and num_workers=2. Note that we use coco_detection_yolo_format_val for training and testing data to instantiate the dataloader.

Listing 5. Setting-Up Dataloaders

```
from IPython.display import clear_output
```

```
train_data = coco_detection_yolo_format_train(
    dataset_params={
        'data_dir': dataset_params['data_dir'],
        'images_dir': dataset_params['train_images_dir'],
        'labels_dir': dataset_params['train_labels_dir'],
        'classes': dataset_params['classes']
    },
    dataloader_params={
        'batch_size': BATCH_SIZE,
        'num_workers': 2
    }
)

val_data = coco_detection_yolo_format_val(
    dataset_params={
        'data_dir': dataset_params['data_dir'],
        'images_dir': dataset_params['val_images_dir'],
        'labels_dir': dataset_params['val_labels_dir'],
        'classes': dataset_params['classes']
    },
    dataloader_params={
        'batch_size': BATCH_SIZE,
        'num_workers': 2
    }
)

test_data = coco_detection_yolo_format_val(
    dataset_params={
        'data_dir': dataset_params['data_dir'],
        'images_dir': dataset_params['test_images_dir'],
        'labels_dir': dataset_params['test_labels_dir'],
        'classes': dataset_params['classes']
    },
    dataloader_params={
        'batch_size': BATCH_SIZE,
        'num_workers': 2
    }
)
clear_output()
```

6. **Model Instantiation for Fine-Tuning** Choose the YOLO-NAS variant (e.g., `yolo_nas_l`) and launch the model, tailoring the number of classes to your dataset.

```
from super_gradients.training import models
model = models.get(selected_model_arch , num_classes=len(dataset_params
['classes']), pretrained_weights="coco")
```

7. **Training Parameters and Metrics:** While utilizing SuperGradients, define your training parameters thoughtfully because they will affect your training process and significantly impact your model's performance. To help you get started, here's a brief overview of the most crucial parameters and some advanced features you should consider. [74]

Essential Training Parameters

- `max_epochs`: Tells the upper limit for the number of training cycles.
- **loss:** Select an appropriate loss function for your model and dataset.
- **optimizer:** Choose an optimizer and modify it if needed using `optimizer_params`.
- `train_metrics_list`/`valid_metrics_list`**:** Specify metrics for evaluating training and validation performance. These are implemented as Torchmetric objects.
- `metric_to_watch`**:** Select a primary metric to determine checkpoint saving.

Advanced Features and Integrations

- **Monitoring Tools:** There a lot of monitoring tools like Weights, Tensorboard, ClearML, Biases, or for tracking training progress. Custom integrations can be achieved using Base SGLogger.
- **Enhancing the Training :** there are so many features in supergradients SuperGradients like Exponential Moving Average(EMA), Zero Weight Decay on Bias and Batch Normalization, Weight Averaging, Batch Accumulation, and Precise BatchNorm for optimized training.
- **Custom Metrics:** SuperGradients is customizable allows create custom metrics.

Loss and Detection Metrics Setup:
Setup Loss and Detection Metrics: correct number of classes for functions like PPYoloELoss and DetectionMetrics_050 to align with dataset specifics.

```
from super_gradients.training.losses import PPYoloELoss
from super_gradients.training.metrics import DetectionMetrics_050
from super_gradients.training.models.detection_models.pp_yolo_e
 import PPYoloEPostPredictionCallback
```

```
train_params = {
    # ENABLING SILENT MODE
    'silent_mode': True,
    "average_best_models":True,
    "warmup_mode": "linear_epoch_step",
    "warmup_initial_lr": 1e-6,
    "lr_warmup_epochs": 3,
    "initial_lr": 5e-4,
    "lr_mode": "cosine",
    "cosine_final_lr_ratio": 0.1,
    "optimizer": "Adam",
    "optimizer_params": {"weight_decay": 0.0001},
    "zero_weight_decay_on_bias_and_bn": True,
    "ema": True,
    "ema_params": {"decay": 0.9, "decay_type": "threshold"},
    # ONLY TRAINING FOR 10 EPOCHS FOR THIS EXAMPLE NOTEBOOK
    "max_epochs": 10,
    "mixed_precision": True,
    "loss": PPYoloELoss(
        use_static_assigner=False,
        # NOTE: num_classes needs to be defined here
        num_classes=len(dataset_params['classes']),
        reg_max=16
    ),
    "valid_metrics_list": [
        DetectionMetrics_050(
            score_thres=0.1,
            top_k_predictions=300,
            # NOTE: num_classes needs to be defined here
            num_cls=len(dataset_params['classes']),
            normalize_targets=True,
            post_prediction_callback=PPYoloEPostPredictionCallback(
                score_threshold=0.01,
                nms_top_k=1000,
                max_predictions=300,
                nms_threshold=0.7
            )
        )
    ],
    "metric_to_watch": 'mAP@0.50'
}
```

8. **Instantiate the training:**

Listing 6. Commencing the Training

```
trainer.train(model=model, training_params=train_params,
train_loader=train_data, valid_loader=val_data)
```

9. **Fetching the Best(Optimal) Model after Training:** Following intensive training, wherein the model underwent rigorous learning iterations, we are now focused on retrieving the best-trained model. This phase involves selecting the most refined and proficient version of the model, one that has successfully absorbed and generalized patterns from the training data. The goal is to identify the model that exhibits superior performance, ensuring its readiness for deployment in real-world scenarios. This meticulous retrieval process is crucial for obtaining a highly capable and reliable model that aligns with the desired objectives and expectations. In SuperGradients, this involves selecting the most effective set of weights from your training runs.
Using Checkpoint Averaging

Listing 7. Load trained model

```
from super_gradients.training import models

# Path to the best weight checkpoint
averaged_checkpoint_path = 'checkpoints/Experiment_name/average_model.pth'

# Retrieve the model with averaged weights
best_model = models.get(
        selected_model_arch,
        num_classes=len(dataset_params['classes']),
        checkpoint_path=f"{CHECKPOINT_DIR}/{EXPERIMENT_NAME}/ckpt_best.pth"
).to(DEVICE)
```

10. **Evaluate trained model**

Listing 8. Model Evaluation on Test Data

```
trainer.test(
    model=best_model,
    test_loader=test_data,
    test_metrics_list=DetectionMetrics_050(
        score_thres=0.1,
        top_k_predictions=300,
        num_cls=len(dataset_params['classes']),
        normalize_targets=True,
        post_prediction_callback=PPYoloEPostPredictionCallback(
            score_threshold=0.01,
            nms_top_k=1000,
            max_predictions=300,
            nms_threshold=0.7
```


90

```
                    )
                )
            )
```

11. **Model Evaluation on Test Data:** Evaluating a model on test data is a pivotal phase within the machine learning workflow, gauging the effectiveness of a trained model on new data. The primary objective is to ascertain the model's generalization capability for novel, unseen instances. This section outlines the prevalent metrics and procedures employed in assessing a model using test data.

Listing 9. Inference with trained model

```
Images ='Path_to_image'
images_predications = best_model.predict(Images, iou=0.5, conf=0.5).show()
```


91

8. Result and discussion

The YOLO-NAS has YOLO-NAS-L, YOLO-NAS-M, and YOLO-NAS-S variants, and in the result and discussion I am tailored to discuss individual performance characteristics of each model, and every model has three paths to choose from; the best model, average model, and latest model, and particularly I performed inference using the best weight of each model.

8.1 Training and Implementation Details

A batch size of 16 was used for all models, and they were trained for 60 epochs on NVIDIA Tesla T4 GPU with High-RAM. Before commencing training, various hyperparameters were adjusted. The "average best models" option was enabled, and warm-up models were set up with a linear epoch step. The initial learning rate was set to 1e-6 during warm-up, and the learning rate decay factor was set to 3. The initial learning rate was set to 5e-4, and the learning rate decay mode was cosine. The cosine-final learning rate ratio was set to 0.1, and the optimizer was Adam. The weight decay in optimizer parameters was set to 0.0001, and no weight decay was applied to bias and batch-normalization. Furthermore, exponential moving averaging was used with a decay factor of 0.9, and the decay type was set to a threshold. The "mixed precision" option was enabled during training. All models were implemented and trained on Google Colab.

8.1.1 YOLO-NAS-S

The performance evaluation of inference on both models(i.e., before training and after training) is conducted utilizing a metric known as Intersection over Union (IoU) with a value of 0.5 and above and a confidence level of 0.5 and above. The images used to evaluate the model's performance are foreign images.

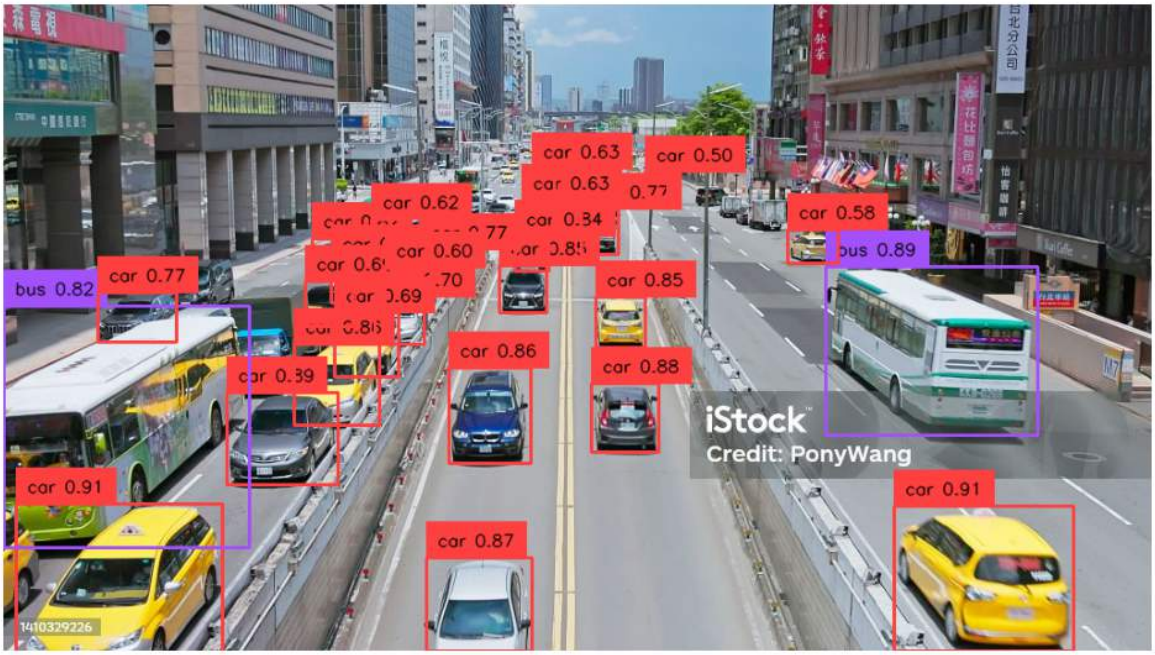


Figure 8.1. YOLO-NAS-S: Inference on the Model before training

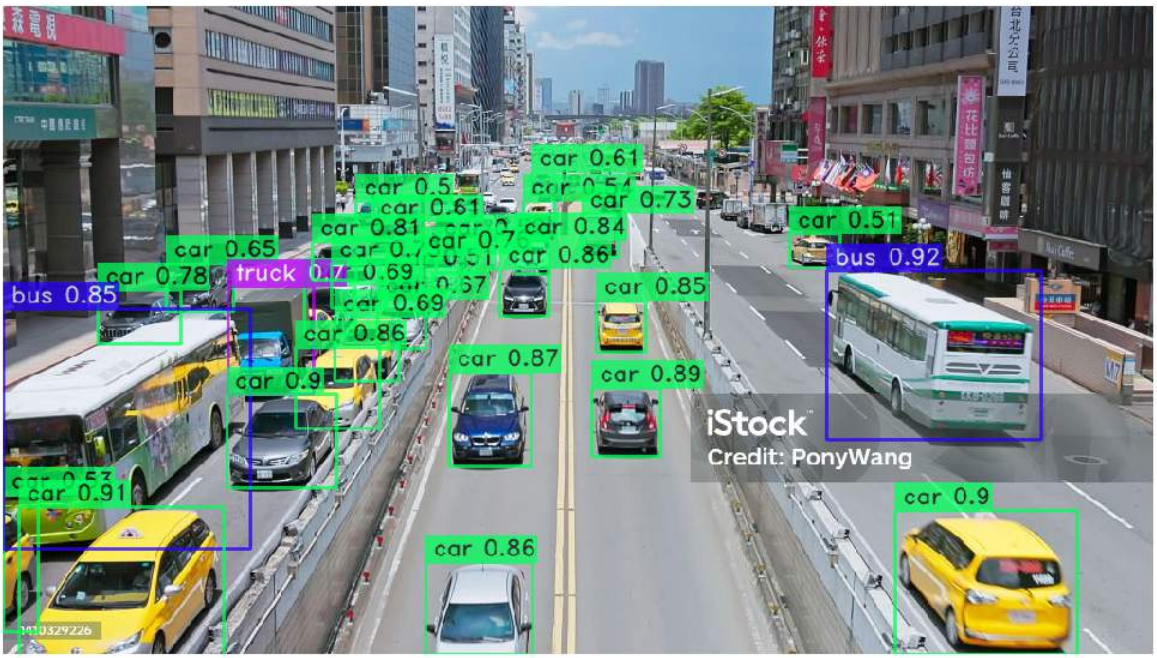


Figure 8.2. YOLO-NAS-S: Inference on the Model after training

Figure 8.3. YOLO-NAS: Inference on the model before and after training .

Both models exhibit almost the same level of performance, but as can be seen from both images, the trained model outperforms with a slight difference where it shows a slightly better trade-off between precision and recall, also the dataset used for training is annotated mostly for not distant or short range, as a consequence the model is also exhibiting that it is struggling to detect the distant classes, sometimes the fine-tuned model is good at detecting distant objects.

93

Figure 8.4. Visualization of the performance of YOLO-NAS-S with bounding boxes and corresponding obstacle classes predicted by the model.

94

The Confusion Matrix illustrates the effectiveness of the overall YOLO-NAS-S in detecting obstacles across four distinct classes. Notably, the matrix reveals a notable variation in the true positive values. Specifically, the highest true positive rate, standing at 1 which is not good, is achieved in detecting the 'motorbike' class. Conversely, the 'truck' class exhibits the lowest true positive rate, measured at 0.55.

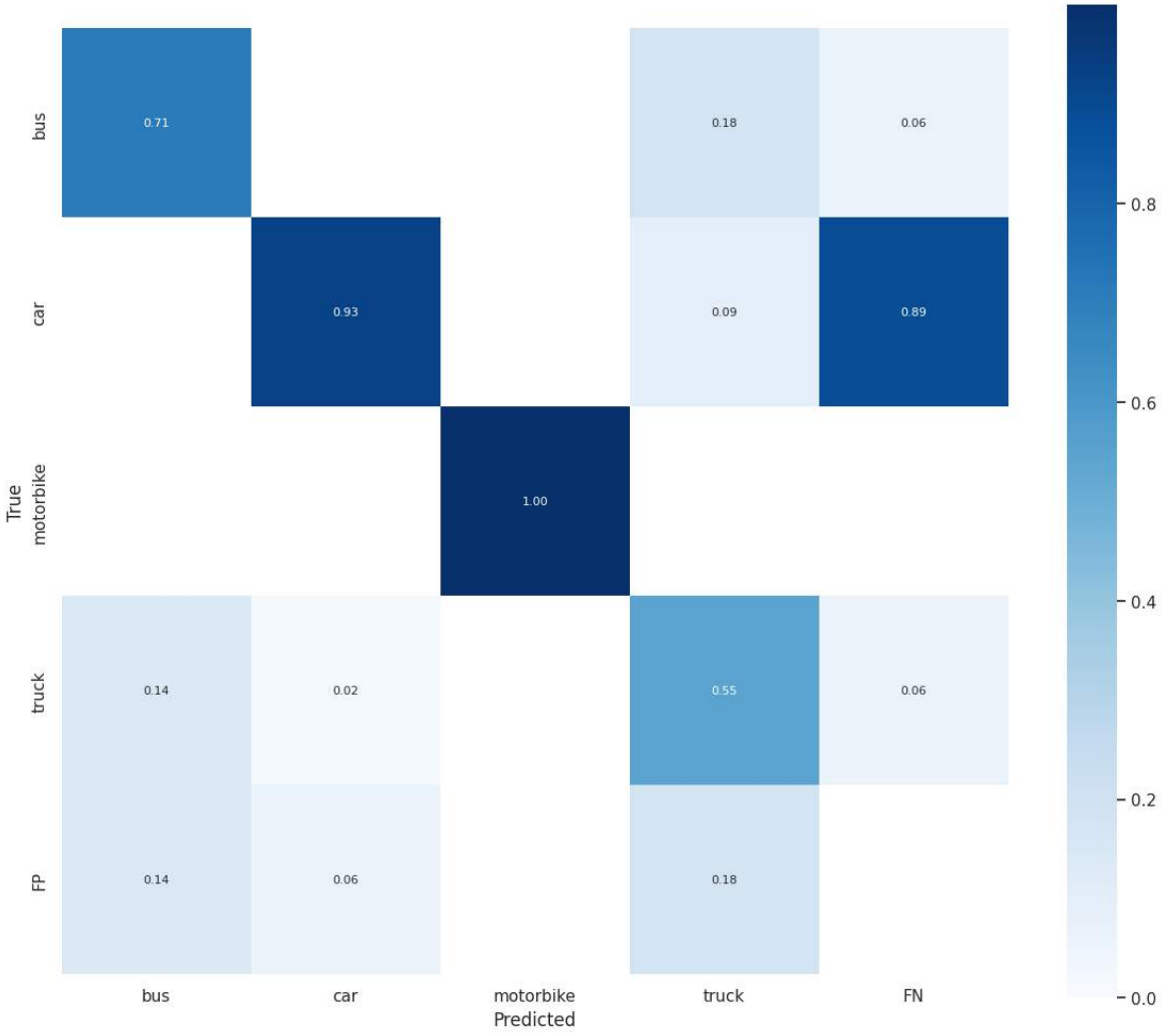


Figure 8.5. YOLO-NAS-S Overall results

95

8.1.2 YOLO-NAS-M

The performance evaluation of inference for models is the same for all. As shown in 8.11, The pre-trained model produces false positives, and after extensive training, the trained model exhibits very good results in detecting the short-range objects as YOLO-NAS-S model because of the dataset properties.

Figure 8.6. YOLO-NAS-M: Inference on the Model before training

Figure 8.7. YOLO-NAS-M: Inference on the Model after training

Figure 8.8. YOLO-NAS-M: Inference on the model before and after training .

96

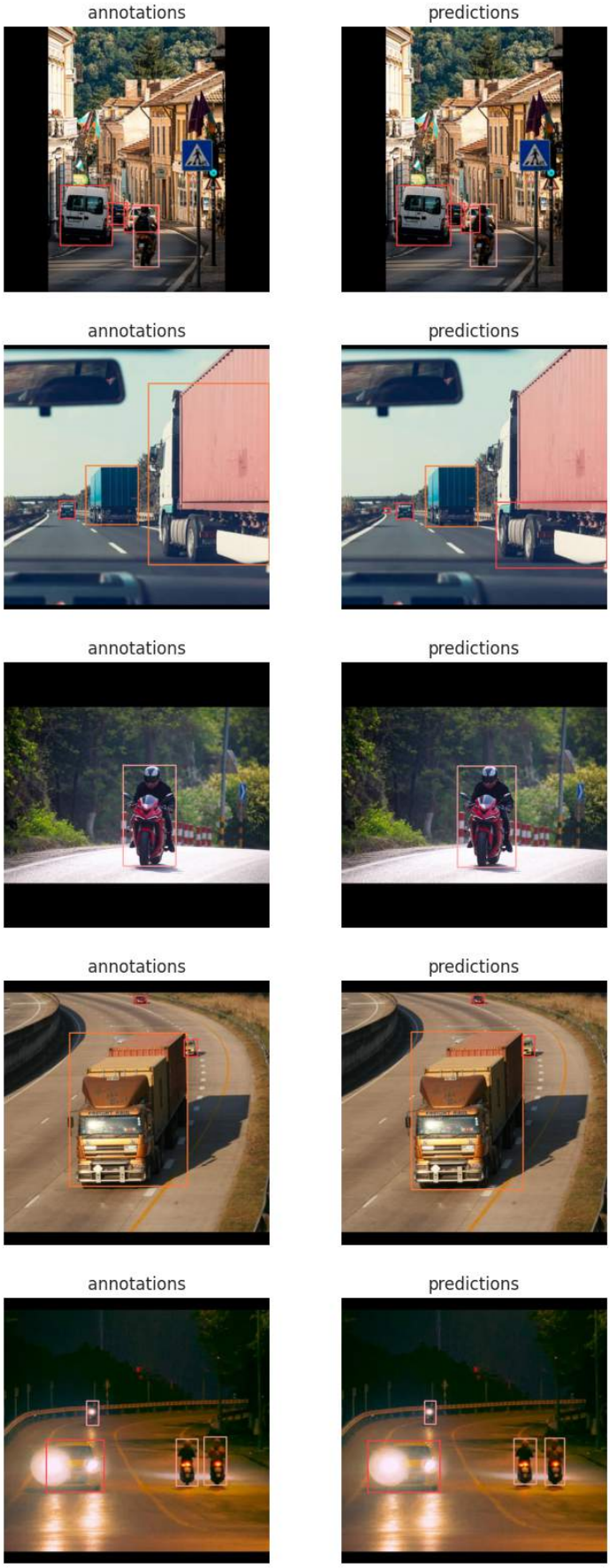


Figure 8.9. Visualization of the performance of YOLO-NAS-M with bounding boxes and corresponding obstacle classes predicted by the model.

97

The Confusion Matrix illustrates the effectiveness of the overall YOLO-NAS-M in detecting obstacles across four distinct classes. Notably, the matrix reveals a notable variation in the true positive values. Specifically, the highest true positive rate, standing at 0.93 which is good, is achieved in detecting the 'motorbike' class. Conversely, the 'truck' class exhibits the lowest true positive rate, measured at 0.56 which is better than the small model, from this we can tell that it is slightly better at balancing the trade-off between recall and precision, this is due to the size of the dataset in the backbone of the pre-trained model. The table shows that the model is losing to detect a car class or generates a significant number of false negatives for this category. This is likely attributed to the model's struggle in detecting distant cars.

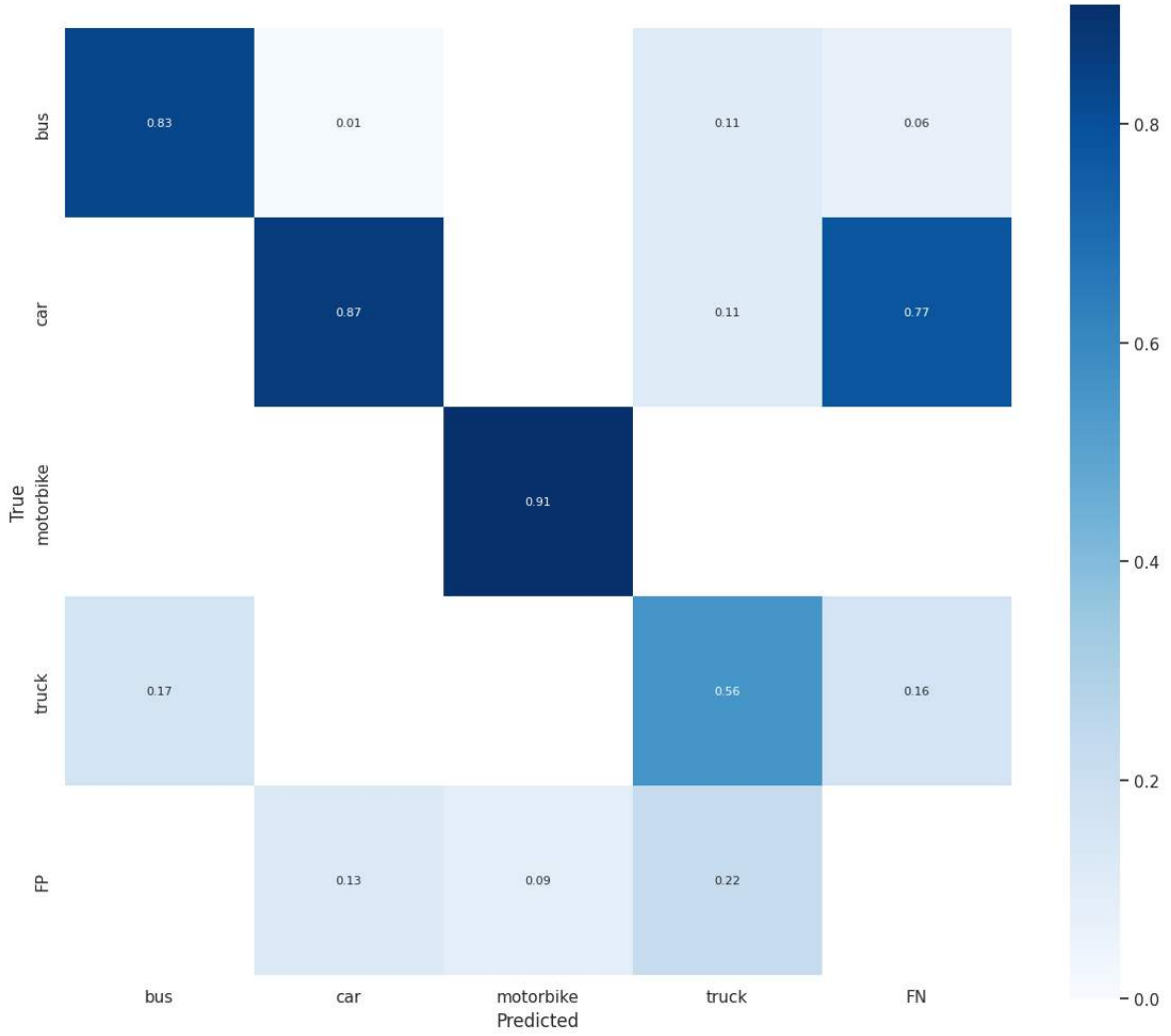


Figure 8.10. YOLO-NAS-M Overall results

8.1.3 YOLO-NAS-L

The performance evaluation of inference for models is the same for all. As the model size of the pre-trained model increases, the model exhibits an increase in the trade-off between precision and recall and I will discuss extensively in Table: 8.2

98

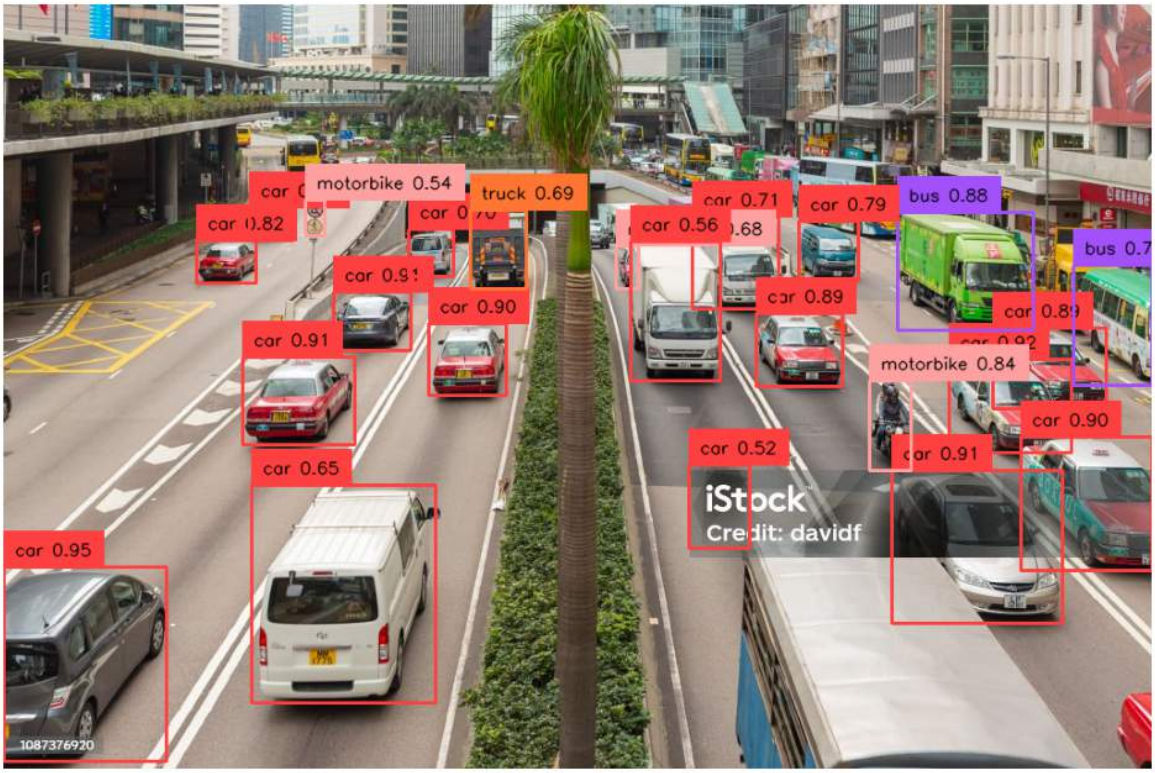


Figure 8.11. YOLO-NAS-L: Inference on the Model before training

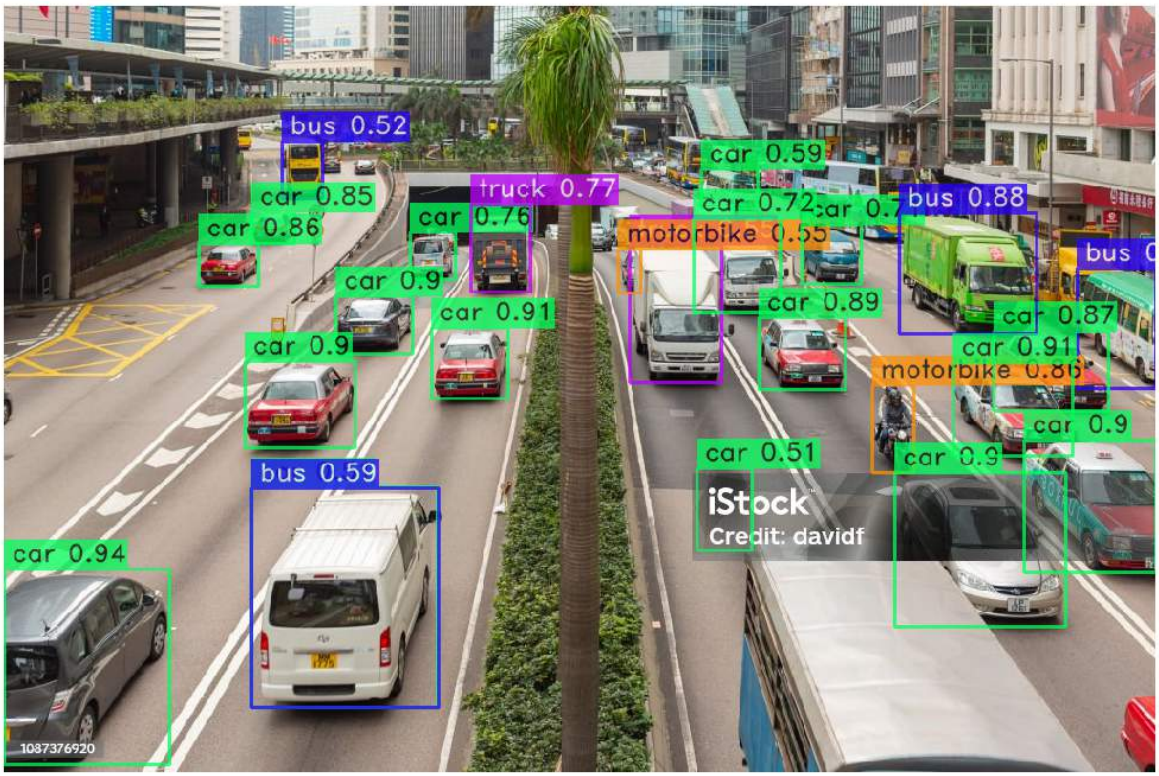


Figure 8.12. YOLO-NAS-L: Inference on the Model after training

Figure 8.13. YOLO-NAS-L: Inference on the model before and after training .

99

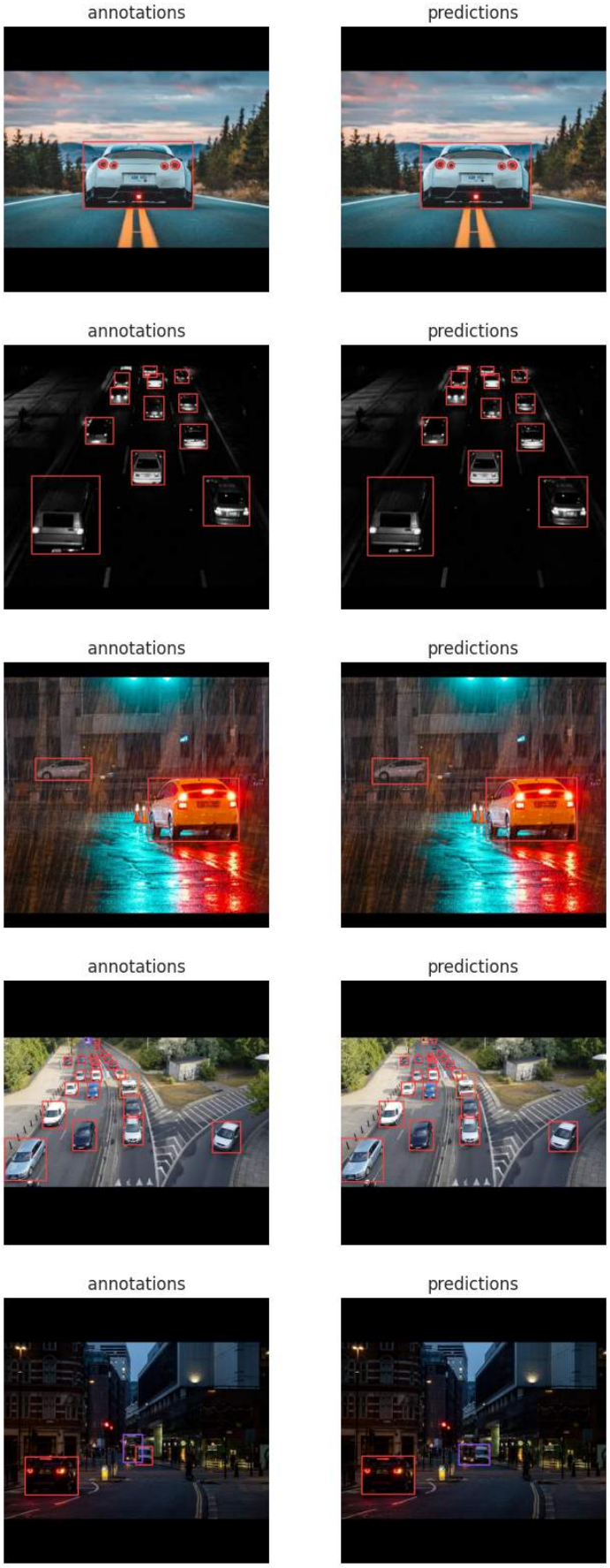


Figure 8.14. Visualization of the performance of YOLO-NAS-L with bounding boxes and corresponding obstacle classes predicted by the model.

The Confusion Matrix illustrates the effectiveness of the overall YOLO-NAS-L in detecting obstacles across four distinct classes. Notably, the matrix reveals a notable variation in the true positive values. Specifically, the highest true positive rate, standing at 0.91 which is good, is achieved in detecting the 'motorbike' class. Conversely, the 'bus' class exhibits the lowest true positive rate, measured at 0.44 which is very poor.

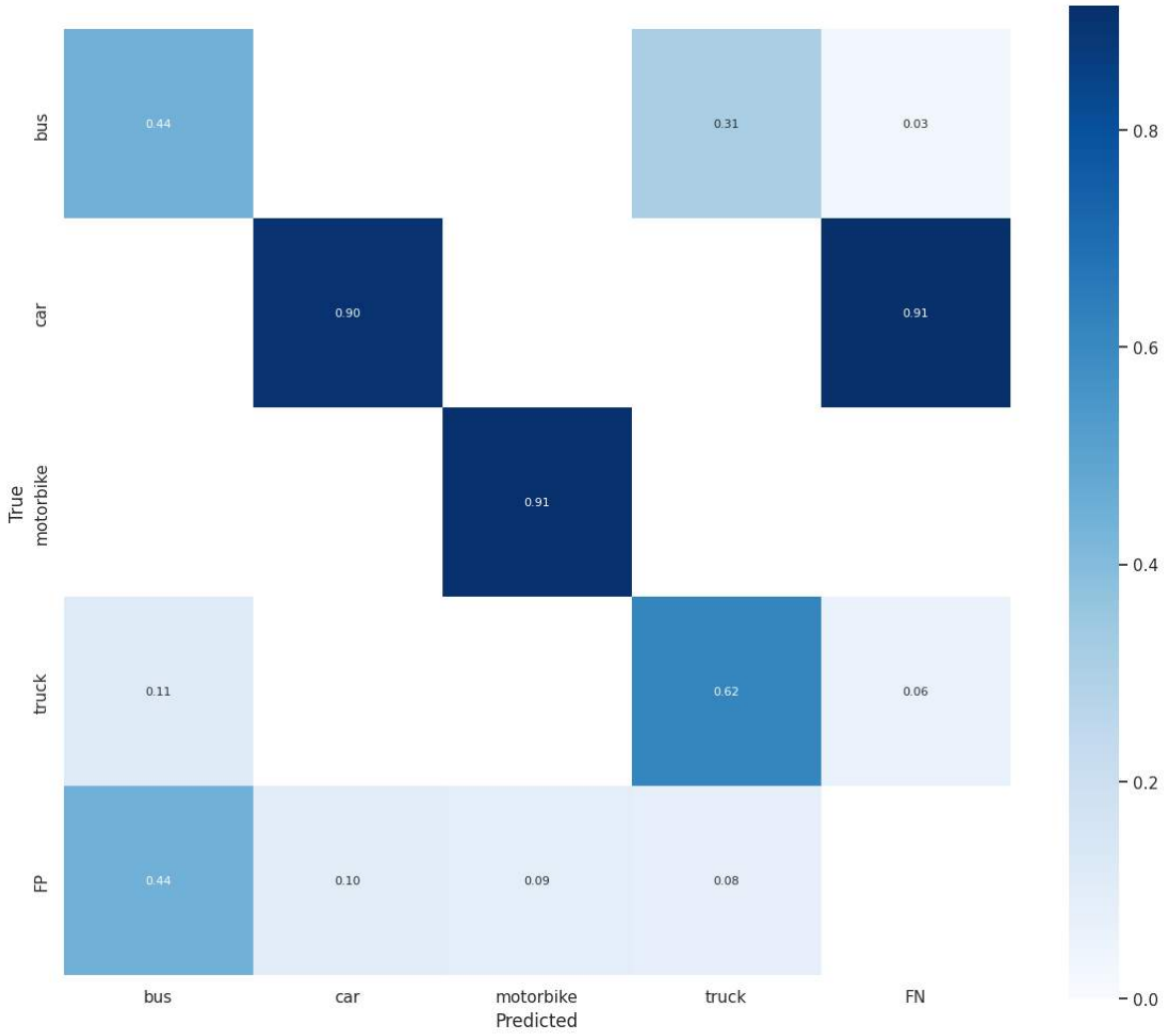


Figure 8.15. YOLO-NAS-L Overall results

101

8.2 Analysis of Performance for YOLO-NAS

Table 8.1. Over-all results of the three models from the confusion matrix

YOLO-NAS-SMALL					
		Bus	Car	Motorbike	Truck
	Bus	0.71			
	Car		0.93		
	Motorbike			1	
	Truck				0.55
YOLO-NAS-MEDIUM					
		Bus	Car	Motorbike	Truck
	Bus	0.83			
True	Car		0.87		
	Motorbike			0.91	
	Truck				0.56
YOLO-NAS-LARGE					
		Bus	Car	Motorbike	Truck
	Bus	0.44			
	Car		0.90		
	Motorbike			0.91	
	Truck				0.62
	Predicted				

The primary objective of table 8.1 is to provide a comprehensive analysis of the True Positive (TP) rate of three different models upon completion of their training. The table presents a comparative evaluation of the models' performance in detecting various classes, thus enabling researchers to identify the model that performs best. It is evident from the results that YOLO-NAS-M outperforms the other models in detecting all classes. Therefore, researchers can also fine-tune this model for future applications that require high accuracy in detecting various classes, similar to the training dataset mentioned in 6.3.8.

Table 8.2. Analysis of Performance for YOLO-NAS

Models	Loss_cls	Loss_iou	Loss_dfl	Loss	Precision@0.5	Recall@0.5	mAP@0.5	F1@0.5
NAS-Small	0.638	0.122	0.729	1.308	0.0717	0.9327	0.7812	0.1323
NAS-Medium	0.6198	0.1194	0.7262	1.2814	0.0831	0.9308	0.8339	0.1513
NAS-Large	0.6000	0.1154	0.7355	1.256	0.098	0.9813	0.8707	0.1779

The performance metrics presented in Table 8.2 highlight interesting facets of the behavior of the different YOLO-NAS variants. First, the Recall scores for all models are exceptionally high, above 0.93 in all cases, which means that these models are very good at detecting the classes. This could be crucial in scenarios where the priority is to detect as many positive cases as possible, even at the risk of detecting some false classes.

Conversely, the precision scores present a contrasting narrative. The scores for all models are significantly lower, particularly in comparison to the high recall values. YOLO-NAS-L, the largest model, offers the best Precision at 0.098, YOLO-NAS-M at 0.0831, and YOLO-NASs at 0.0717. These low Precision scores suggest that while the models are good at capturing

positive cases, they also have a high rate of false positives, this false positive is mostly coming from one class that is 'car' We can clearly see that in figure 8.5, 8.10 8.15

The F1-score, grows with the model size, reflecting that larger models achieve a better balance between these two measures.

The mAP@0.50 scores show that model size matters in delivering the best overall performance and that the larger models deliver the best overall performance.

In summary, while all the YOLO-NAS models exhibit a strong ability to capture positive cases, they struggle with precision, indicating high false positives. Though marginally, the models' complexity seems to positively influence precision and F1-score, suggesting potential benefits of using larger models if computational resources and training time permit. However, the precision-recall trade-off must be carefully considered based on the specific requirements of the detection task.

103

9. Summary

The paper discusses the implementation of Neural Architecture Search (AutoNAC) to optimize YOLO-NAS for latency and throughput, resulting in three new architectures (YOLO-NASS, YOLO-NASM, and YOLO-NASL). The models were fine-tuned on a specific dataset from the Roboflow universe. Their performance was evaluated on a separate dataset using various performance metrics such as Intersection over Union (IoU), Mean Average Precision (mAP), Precision, Recall, and F1 Score.

After extensive training, notably, YOLO-NAS-S achieved a mAP@0.5 of 0.7812, an inference time of 5.38e+3, a recall of 0.9327, and an F1 Score of 0.1323. YOLO-NAS-M attained a mAP@0.5 of 0.8339, an inference time of 7.78e+3, a recall of 0.9308, and an F1 Score of 0.1513. Lastly, YOLO-NAS-L demonstrated a mAP@0.5 of 0.8707, an inference time of 9.31e+3, a recall of 0.9327, and an F1 Score of 0.1779.

Based on the results, it is recommended to utilize YOLO-NAS-M for object detection tasks as it strikes a balance between accuracy and efficiency. YOLO-NAS-M is found to be efficient for real-time applications while maintaining good performance in identifying and localizing objects in images.

Generally, this model is good where we mostly focus on producing or capturing the more positive cases, and this will sometimes over-predict or may detect when it is not.

It would be better in the future to fine-tune a big dataset that has been correctly labeled and purposefully collected for the special job to see if it will balance the precision and recall or may result in a good F1 Score.

References

[1] P. Wenzel, T. Schön, L. Leal-Taixé, and D. Cremers, "Vision-based mobile robotics obstacle avoidance with deep reinforcement learning", in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2021, pp. 14 360–14 366.

[2] B. Wilhelm and J. B. Mark, *Digital image processing: An algorithmic introduction using java*, 2016.

[3] V. Tyagi, *Understanding digital image processing*. CRC Press, 2018.

[4] M. A. Nielsen, *Neural networks and deep learning*. Determination press San Francisco, CA, USA, 2015, vol. 25.

[5] L. Arnold, S. Rebecchi, S. Chevallier, and H. Paugam-Moisy, "An introduction to deep learning", in *The European Symposium on Artificial Neural Networks*, 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:52860197.

[6] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning",, vol. 521, no. 7553, pp. 436–444, May 2015. DOI: 10.1038/nature14539.

[7] N. Barla, "What is deep learning? a guide to deep learning use cases, applications, and benefits", Dostęp zdalny (FEBRUARY 14, 2023): https://clear.ml/blog/what-is-deep-learning, 2023.

[8] P. P. Shinde and S. Shah, "A review of machine learning and deep learning applications", in *2018 Fourth international conference on computing communication control and automation (ICCUBEA)*, IEEE, 2018, pp. 1–6.

[9] R. Y. Choi, A. S. Coyner, J. Kalpathy-Cramer, M. F. Chiang, and J. P. Campbell, "Introduction to machine learning, neural networks, and deep learning", *Translational vision science & technology*, vol. 9, no. 2, pp. 14–14, 2020.

[10] S. Ansari, *Building Computer Vision Applications Using Artificial Neural Networks*. Springer, 2020.

[11] S. Knerr, L. Personnaz, and G. Dreyfus, "Single-layer learning revisited: A stepwise procedure for building and training a neural network", in *Neurocomputing: algorithms, architectures and applications*, Springer, 1990, pp. 41–50.

[12] R. Vargas, A. Mosavi, and R. Ruiz, "Deep learning: A review", 2017.

[13] M.-P. Hosseini, S. Lu, K. Kamaraj, A. Slowikowski, and H. C. Venkatesh, "Deep learning architectures", *Deep learning: concepts and architectures*, pp. 1–24, 2020.

[14] O. Calin, *Deep learning architectures*. Springer, 2020.

[15] A. Coates, A. Ng, and H. Lee, "An analysis of single-layer networks in unsupervised feature learning", in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, JMLR Workshop and Conference Proceedings, 2011, pp. 215–223.

[16] ajitjaokar, "The mathematics of forward and back propagation", Dostęp zdalny (April 30, 2019): https://www.datasciencecentral.com/the-mathematics-of-forward-and-back-propagation/, 2019.

[17] R. Hecht-Nielsen, "Theory of the backpropagation neural network", in *Neural networks for perception*, Elsevier, 1992, pp. 65–93.

[18] S. Sharma, S. Sharma, and A. Athaiya, "Activation functions in neural networks", *Towards Data Sci*, vol. 6, no. 12, pp. 310–316, 2017.

[19] A. Apicella, F. Donnarumma, F. Isgrò, and R. Prevete, "A survey on modern trainable activation functions", *Neural Networks*, vol. 138, pp. 14–32, 2021.

[20] B. Fortuner, *Machine learning glossary*, Dostęp zdalny (14.03.2019): `https://github.com/bfortuner/ml-glossary`, 2019.

[21] J. Heaton, "Ian goodfellow, yoshua bengio, and aaron courville: Deep learning: The mit press, 2016, 800 pp, isbn: 0262035618", *Genetic programming and evolvable machines*, vol. 19, no. 1-2, pp. 305–307, 2018.

[22] J. Schmidhuber, "Deep learning in neural networks: An overview", *Neural networks*, vol. 61, pp. 85–117, 2015.

[23] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[24] D. Choi, C. J. Shallue, Z. Nado, J. Lee, C. J. Maddison, and G. E. Dahl, "On empirical comparisons of optimizers for deep learning", *arXiv preprint arXiv:1910.05446*, 2019.

[25] C. Bishop, "Pattern recognition and machine learning", *Springer google schola*, vol. 2, pp. 531–537, 2006.

[26] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning", *Journal of big data*, vol. 6, no. 1, pp. 1–48, 2019.

[27] J. Kukačka, V. Golkov, and D. Cremers, "Regularization for deep learning: A taxonomy", *arXiv preprint arXiv:1710.10686*, 2017.

[28] S. Madhavan and M. T. Jones, "Deep learning architectures", *IBM Developer*, 2017.

[29] S. Shah, "Convolutional neural network: An overview", Dostęp zdalny (15 Mar, 2022): `https://www.analyticsvidhya.com/blog/2022/01/convolutional-neural-network-an-overview/`, 2022.

[30] L. Alzubaidi, J. Zhang, A. J. Humaidi, *et al.*, "Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions", *Journal of big Data*, vol. 8, pp. 1–74, 2021.

[31] A. Sherstinsky, "Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network", *Physica D: Nonlinear Phenomena*, vol. 404, p. 132 306, 2020.

[32] Z. Yu, K. Wang, Z. Wan, S. Xie, and Z. Lv, "Popular deep learning algorithms for disease prediction: A review", *Cluster Computing*, vol. 26, no. 2, pp. 1231–1251, 2023.

[33] I. Sohn, "Deep belief network based intrusion detection techniques: A survey", *Expert Systems with Applications*, vol. 167, p. 114 170, 2021.

[34] L. Deng, "A tutorial survey of architectures, algorithms, and applications for deep learning", *APSIPA transactions on Signal and Information Processing*, vol. 3, e2, 2014.

[35] Z.-Q. Zhao, P. Zheng, S.-t. Xu, and X. Wu, "Object detection with deep learning: A review", *IEEE transactions on neural networks and learning systems*, vol. 30, no. 11, pp. 3212–3232, 2019.

[36] A. R. Pathak, M. Pandey, and S. Rautaray, "Application of deep learning for object detection", *Procedia computer science*, vol. 132, pp. 1706–1717, 2018.

[37] H. P. Sabina N Aneesa M. P, "Object detection using yolo and mobilenet ssd", *ENGINEERING RESEARCH TECHNOLOGY (IJERT)*, vol. 11, 2022.

[38] L. Du, R. Zhang, and X. Wang, "Overview of two-stage object detection algorithms", in *Journal of Physics: Conference Series*, IOP Publishing, vol. 1544, 2020, p. 012 033.

[39] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[40] R. Girshick, "Fast r-cnn", in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440–1448.

[41] Y. P. Chen, Y. Li, and G. Wang, "An enhanced region proposal network for object detection using deep learning method", *PloS one*, vol. 13, no. 9, e0203897, 2018.

[42] A. Salvador, X. Giró-i-Nieto, F. Marqués, and S. Satoh, "Faster r-cnn features for instance search", in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2016, pp. 9–16.

[43] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn", in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2961–2969.

[44] L. Zhu, F. Lee, J. Cai, H. Yu, and Q. Chen, "An improved feature pyramid network for object detection", *Neurocomputing*, vol. 483, pp. 127–139, 2022.

[45] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 2117–2125.

[46] W. Liu, D. Anguelov, D. Erhan, *et al.*, "Ssd: Single shot multibox detector", in *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*, Springer, 2016, pp. 21–37.

[47] R. Padilla, S. L. Netto, and E. A. Da Silva, "A survey on performance metrics for object-detection algorithms", in *2020 international conference on systems, signals and image processing (IWSSIP)*, IEEE, 2020, pp. 237–242.

[48] H. Li, B. Singh, M. Najibi, Z. Wu, and L. S. Davis, "An analysis of pre-training on object detection", *arXiv preprint arXiv:1904.05871*, 2019.

[49] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[50] K. You, Y. Liu, J. Wang, and M. Long, "Logme: Practical assessment of pre-trained models for transfer learning", in *International Conference on Machine Learning*, PMLR, 2021, pp. 12 133–12 143.

[51] R. Murugan and T. Goel, "E-diconet: Extreme learning machine based classifier for diagnosis of covid-19 using deep convolutional network", *Journal of Ambient Intelligence and Humanized Computing*, vol. 12, no. 9, pp. 8887–8898, 2021.

[52] B. D. Team, "Introduction to automated neural architecture construction technology", Remote access (April 29, 2020): ://deci.ai/resources/autonac/.

[53] J. Terven, D.-M. Córdova-Esparza, and J.-A. Romero-González, "A comprehensive review of yolo architectures in computer vision: From yolov1 to yolov8 and yolo-nas", *Machine Learning and Knowledge Extraction*, vol. 5, no. 4, pp. 1680–1716, 2023.

[54] J. Sang, Z. Wu, P. Guo, *et al.*, "An improved yolov2 for vehicle detection", *Sensors*, vol. 18, no. 12, p. 4272, 2018.

[55] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement", *arXiv preprint arXiv:1804.02767*, 2018.

[56] R. Gai, N. Chen, and H. Yuan, "A detection algorithm for cherry fruits based on the improved yolo-v4 model", *Neural Computing and Applications*, vol. 35, no. 19, pp. 13 895–13 906, 2023.

[57] T.-H. Wu, T.-W. Wang, and Y.-Q. Liu, "Real-time vehicle and distance detection based on improved yolo v5 network", in *2021 3rd World Symposium on Artificial Intelligence (WSAI)*, IEEE, 2021, pp. 24–28.

[58] Z. Ge, S. Liu, F. Wang, Z. Li, and J. Sun, "Yolox: Exceeding yolo series in 2021", *arXiv preprint arXiv:2107.08430*, 2021.

[59] X. Huang, X. Wang, W. Lv, *et al.*, "Pp-yolov2: A practical object detector", *arXiv preprint arXiv:2104.10419*, 2021.

[60] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "Scaled-yolov4: Scaling cross stage partial network", in *Proceedings of the IEEE/cvf conference on computer vision and pattern recognition*, 2021, pp. 13 029–13 038.

[61] X. Long, K. Deng, G. Wang, *et al.*, "Pp-yolo: An effective and efficient implementation of object detector", *arXiv preprint arXiv:2007.12099*, 2020.

[62] S. Xu, X. Wang, W. Lv, *et al.*, "Pp-yoloe: An evolved version of yolo", *arXiv preprint arXiv:2203.16250*, 2022.

[63] C. Li, L. Li, H. Jiang, *et al.*, "Yolov6: A single-stage object detection framework for industrial applications", *arXiv preprint arXiv:2209.02976*, 2022.

[64] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors", in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 7464–7475.

[65] X. Xu, Y. Jiang, W. Chen, Y. Huang, Y. Zhang, and X. Sun, "Damo-yolo: A report on real-time object detection design", *arXiv preprint arXiv:2211.15444*, 2022.

[66] F. Jacob Solawetz, "What is yolov8? the ultimate guide.", Remote access (JAN 11, 2023): `https://blog.roboflow.com/whats-new-in-yolov8/#what-is-yolov8`, 2023.

[67] B. D. R. Team, "Yolo-nas by deci achieves sota performance on object detection using neural architecture search", Remote access (May 3, 2023): `https://deci.ai/blog/yolo-nas-object-detection-foundation-model/`, 2023.

[68] R. Alake, "Yolo-nas uncovered: Essential insights and implementation techniques for machine learning engineers", Remote access (Jun 12, 2023): `https://richmondalake.medium.com/yolo-nas-uncovered-essential-insights-and-implementation-techniques-for-machine-learning-engineers-87ee266b37f6`, 2023.

[69] B. D. R. Team, "Webinar: The making of yolo-nas, a foundation model, with nas", Remote access (July 27, 2023): `https://deci.ai/resources/webinar-making-yolo-nas-neural-architecture-search/`, 2023.

[70] B. D. A. Team, “An overview of state of the art (sota) dnns”, Remote access (October 18, 2023): https://deci.ai/blog/sota-dnns-overview/, 2023.

[71] B. D. A. Team, “Yolov8 vs. yolo-nas showdown: Exploring advanced object detection”, Remote access (December 4, 2023): https://deci.ai/blog/yolov8-vs-yolo-nas-showdown-exploring-advanced-object, 2023.

[72] a.-v. (glenn.jocher@ultralytics.com (3), “Performance metrics deep dive”, Remote access (18.11.2023): https://docs.ultralytics.com/guides/yolo-performance-metrics/, 2023.

[73] p4p 2023, *P4p cars dataset*, https://universe.roboflow.com/p4p-2023/p4p-cars, Open Source Dataset, visited on 2024-01-24, Aug. 2023. [Online]. Available: https://universe.roboflow.com/p4p-2023/p4p-cars.

[74] S. Aharon, Louis-Dupont, Ofri Masad, *et al.*, *Super-gradients*, 2021. DOI: 10.5281/ZENODO.7789328. [Online]. Available: https://zenodo.org/record/7789328.

List of Symbols and Abbreviations

COCO – Common Objects in Context
DNN – Deep Neural Networks
ANN – Artificial Neural Network
MLP – Multilayer Perceptron
ReLU – Rectified Linear Unit
SELU – Scaled Exponential Linear Unit
MSE – Mean Squared Error
MAE – Mean Absolute Error
MSLE – Mean Squared Logarithmic Error
SVM – Support Vector Machine
KLD – Kullback-Leibler Divergence
CNN – Convolutional Neural Network
RMSProp – Root Mean Square Prop
SGD – Stochastic Gradient Descent
GRU – Gated Recurrent Unit
LSTM – Long-Short term Memory
RNN – Recurrent Neural Network
BPTT – Back-Propagation Through Time
SOM – Self Organized Map
RBM – Restricted Boltzmann Machines
DBN – Deep Belief Networks
DNS – Deep Stacking Networks
SOTA – State-of-Art
YOLO – You Look Only Once
R-CNN – Region-Based Convolutional Neural Network
RPN – Region Proposal Network
FPN – Feature Pyramid Network
SSD – Single Shot Detection
IoU – Intersection Over Union
mAP – Mean of Average Precision
VOC – Visual Object Classes
TP – True Positives
TN – True Negatives
FP – False Positives
FN – False Negatives
VGG – Visual Geometry Group
PP-YOLO – Paddle Paddle *You Look Only Once*
MAE-NAS – Maximum Entropy Neural Architecture Search
GPU – Graphics Processing Unit
E-ELAN – Extended Efficient Layer Aggregation Network

ET-head – Efficient Task-Aligned Head
VFL – Varifocal
DFL – Distribution Focal Loss
FPS – Frames Per Second
PAN – Path Aggregation Network
YOLO-NAS – *You Look Only Once Neural Architectural Search*
KD – Knowledge Distillation
AutoNAC – Automated Neural Architecture Construction
QSP – Quantization-Specific Parameters
QCI – Quantization-Centric Initialization
NAS – Neural Architectural Search
SPP – Spatial Pyramid Pooling
AI – Artificail Intelligence
YOLO-NASS – YOLO-NAS-Small
YOLO-NASM – YOLO-NAS-Medium
YOLO-NASL – YOLO-NAS-Large
RF – Roboflow
EMA – Exponential Moving Average
AP – Average Precision
SGLogger – Super-Gradient Logger

List of Figures

List of Tables