NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF NAVAL ARCHITECTURE & MARINE ENGINEERING
DIVISION OF SHIP STUDY AND MARINE TRANSPORT

Diploma Thesis

Development of an artificial neural network and corresponding software tool for 3D object recognition

Dikaios Christos

Supervisor: A. Gkinis, Assoc. Prof., NTUA

Athens, 2023

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΝΑΥΠΗΓΩΝ ΜΗΧΑΝΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΤΟΜΕΑΣ ΜΕΛΕΤΗΣ ΠΛΟΙΟΥ ΚΑΙ ΘΑΛΑΣΣΙΩΝ ΜΕΤΑΦΟΡΩΝ

Διπλωματική Εργασία

Ανάπτυξη νευρωνικού δικτύου και αντίστοιχου λογισμικού εργαλείου για την αναγνώριση τρισδιάστατων αντικειμένων

Δικαίος Χρήστος

Επιβλέπων: Α. Γκίνης, Αναπλ. Καθηγητής, Ε.Μ.Π.

Αθήνα, 2023

Table of Contents

Table of Figures

Abstract

This diploma thesis aims to explore the potential of artificial intelligence and machine learning techniques, and more specifically the field of object recognition, in the maritime business. It focuses on the development of a robust and accurate system for identifying and localizing mechanical components in complex systems that can be found in a vessel, such as piping networks. The end-goal is to develop an object recognition software tool that will be user-friendly and easy to use, without the need for explicit programming and fine-tuning.

The thesis reviews the state-of-the-art object recognition algorithms and analyzes their function and evolution over time. The proposed approach is based on deep learning techniques, particularly convolutional neural networks, for 3D object recognition in 2D images.

Additionally, the thesis includes a method to automate the process of generating a large and accurate dataset required for training a custom object detection CNN-based network. The developed system has potential applications in various fields, including shipbuilding, manufacturing, and industrial automation, where accurate object recognition can facilitate maintenance, inspection, and retrofitting tasks.

1. Introduction

1.1 Artificial Intelligence (AI)

Artificial Intelligence (AI) is a field of computer science that focuses on creating intelligent machines, capable of performing tasks that normally require human intelligence, such as learning, reasoning, problem-solving, perception and language understanding.

AI systems use algorithms and statistical models to analyze and process large amounts of data, learning from the patterns and relationships they find to make predictions, recommendations and decisions. These systems can be designed to perform a wide range of functions, from recognizing speech and images to driving cars, playing games and even creating new artwork.

There are many different approaches to AI, including rule-based systems, decision trees, neural networks, and deep learning, among others. Each of these approaches has its own strengths and weaknesses, and the choice of method depends on the specific problem that needs to be solved.

AI has numerous applications across a wide range of industries, including healthcare, finance, transportation, entertainment and more. In healthcare, for example, AI can be used to analyze medical images and assist in diagnosis, while in finance, it can be used to detect fraudulent transactions and make investment recommendations.

1.2 Machine Learning (ML)

Machine learning is a subset of artificial intelligence that involves teaching machines to learn from data without being explicitly programmed. It involves developing algorithms and models that can learn and improve on their own, based on the data they are trained on.

There are several types of machine learning algorithms, including supervised learning, unsupervised learning and reinforcement learning. Supervised learning involves training a machine to predict an output based on a set of input data, while unsupervised learning involves identifying patterns and relationships in data without any predefined output. Reinforcement learning involves training a machine to make decisions based on feedback from its environment.

Machine learning has many practical applications, including natural language processing, image recognition, fraud detection and personalized recommendations. It is used in a wide range of industries, including healthcare, finance and retail, among others.

However, it is important to note that machine learning algorithms are not infallible and can be prone to bias and errors. Properly training and evaluating these algorithms is essential to ensure their accuracy and effectiveness.

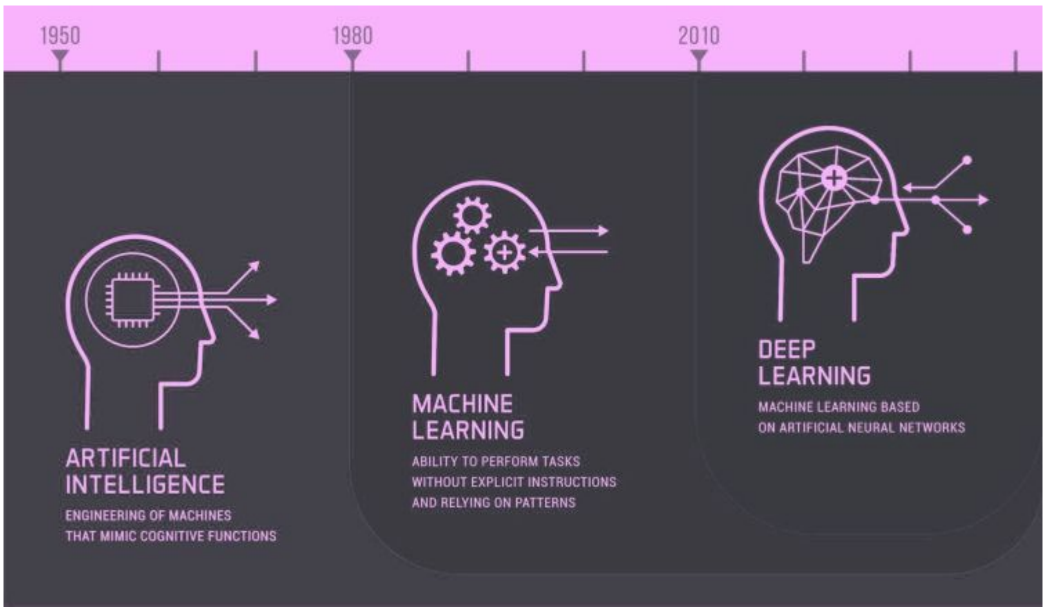
1.3 Deep Learning

Deep learning is a subfield of machine learning that involves the creation of neural networks with multiple layers, allowing for complex computations and pattern recognition. It has been driving advancements in artificial intelligence and has been applied to various fields, including computer vision, natural language processing and speech recognition.

At the heart of deep learning are neural networks, which are composed of multiple layers of interconnected nodes, or neurons. Each layer processes input data and passes the output to the next layer until a final output is produced. The number of layers and neurons in each layer can vary depending on the complexity of the task.

Deep learning has shown impressive results in a range of applications. For example, deep learning algorithms have achieved near-human level accuracy in image classification tasks, such as recognizing objects in images or detecting cancerous cells in medical images. Similarly, natural language processing models using deep learning techniques have been used to develop chatbots and virtual assistants capable of understanding and responding to human language.

However, deep learning is not without its limitations. One of the main challenges is the need for large amounts of labeled data to train the models effectively. Additionally, the complexity of the models can make it difficult to interpret how they make their predictions, leading to concerns about bias and fairness.


1950
1980
2010
ARTIFICIAL INTELLIGENCE
ENGINEERING OF MACHINES THAT MIMIC COGNITIVE FUNCTIONS
MACHINE LEARNING
ABILITY TO PERFORM TASKS WITHOUT EXPLICIT INSTRUCTIONS AND RELYING ON PATTERNS
DEEP LEARNING
MACHINE LEARNING BASED ON ARTIFICIAL NEURAL NETWORKS
Figure 1: Artificial Intelligence, Machine Learning and Deep Learning areas

1.4 Computer vision

Computer vision is a field of study that aims to teach computers to interpret and understand the visual world. It is an interdisciplinary field that combines computer science, mathematics, physics, and biology to develop algorithms and techniques that allow computers to "see" and "understand" images and videos.

The goal of computer vision is to replicate and enhance human vision capabilities, such as recognizing objects, identifying people, detecting and tracking movements and understanding complex scenes. The process of achieving this goal involves acquiring, processing, analyzing and interpreting visual data.

Computer vision is widely used in various applications, including medical imaging, robotics, autonomous vehicles, surveillance, augmented reality and gaming. For example, in medical imaging, computer vision techniques are used to detect and diagnose diseases from medical images, while in autonomous vehicles, they are used to identify and track pedestrians, vehicles and obstacles in real-time.

To achieve these applications, computer vision researchers and engineers use various techniques such as image processing, pattern recognition, machine learning, deep learning, and neural networks. These techniques allow computers to learn from vast amounts of data and make decisions based on that data.

1.5 Object detection / recognition

Object detection and object recognition are two closely related computer vision tasks that involve identifying objects in images or video frames. While they are often used interchangeably, they refer to different aspects of the overall task of understanding the visual world.

Object detection is the process of identifying the presence and location of objects within an image or video frame. It involves not only recognizing what objects are present, but also drawing bounding boxes around them to indicate their precise locations within the image. Object detection is typically performed using deep learning models such as convolutional neural networks (CNNs) that are trained on large datasets of annotated images.

Object recognition, on the other hand, is the process of identifying the specific category or type of object that is present in an image or video frame. This involves recognizing the object's shape, color, texture and other visual features, and mapping those features to a set of predefined categories or labels. Object recognition can be performed using a wide range of machine learning algorithms, including traditional computer vision techniques such as feature extraction and classification, as well as more advanced deep learning approaches such as CNNs and recurrent neural networks (RNNs).

While object detection and object recognition are distinct tasks, they are often used together in practical applications. For example, in self-driving cars, object detection is used to identify obstacles and other vehicles on the road, while object recognition is used to recognize traffic signs and signals. Similarly, in surveillance systems, object detection can be used to detect people and other objects of interest, while object recognition can be used to identify specific individuals or objects based on their appearance.

In recent years, there has been a tremendous amount of progress in both object detection and object recognition, thanks to advances in deep learning and the availability of large datasets such as ImageNet and COCO. These advances have made it possible to build systems that can identify and track objects in real-time with a high degree of accuracy and they have opened up new possibilities for applications in fields ranging from healthcare and agriculture to retail and entertainment.

2. Artificial Neural Networks (ANNs)

Artificial neural networks (ANNs) are a type of machine learning algorithm that is inspired by the structure and function of the human brain. ANNs consist of interconnected nodes, also known as neurons, that process and transmit information. These neurons are organized into layers, with each layer having a specific function.

The input layer receives data from the outside world, such as an image or text. The hidden layers process this information and extract meaningful features, while the output layer produces a prediction or classification based on the input.

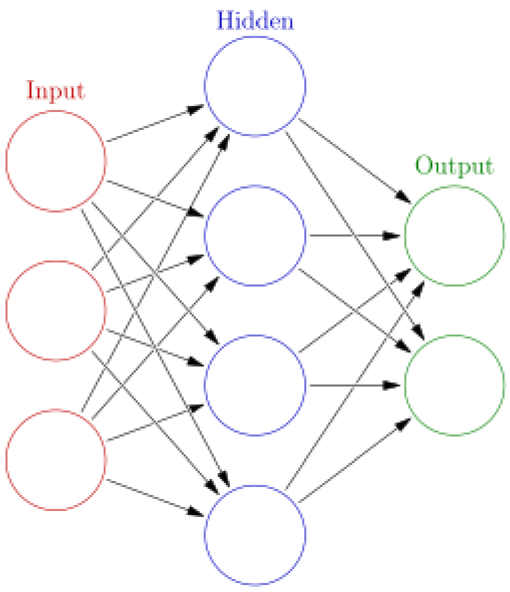


Figure 2: Structure of an ANN with a single hidden layer

One of the key advantages of ANNs is their ability to generalize to new data. Once an ANN is trained on a dataset, it can be applied to new, unseen data and still produce accurate predictions. This makes them a powerful tool for tasks such as image recognition, where the number of possible images is virtually infinite.

2.1 Weights

Weights are the parameters that are learned by the neural network during the training process. They represent the strength of the connections between neurons and determine how much influence one neuron has on another. Training an artificial neural network involves adjusting the weights of the connections between neurons to minimize the error between the predicted output and the true output.

2.2 Bias

Bias, or threshold, is an additional learnable parameter that influences the connections between neurons. It is the one that decides whether the activation function of the neuron will shift towards the positive or negative side, since it is added to the weighted sum of inputs in the activation function.

$$\mathrm{Output} = \mathrm{activation\ function}(\mathrm{x}_1\mathrm{w}_1 + \mathrm{x}_2\mathrm{w}_2 + \cdots + \mathrm{x}_\mathrm{n}\mathrm{w}_\mathrm{n} + \mathrm{bias})$$

where x_i are the inputs and w_i are the weights

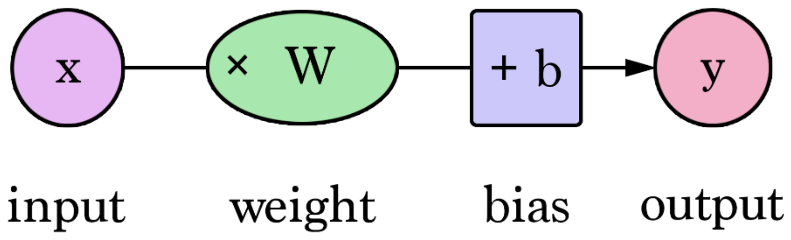


Figure 3: Representation of weight and bias impact on the processing of each neuron

2.3 Activation functions

In ANNs, activation functions are used to introduce non-linearity into the model, allowing it to learn complex patterns and relationships in the input data. An activation function takes a weighted sum of inputs and applies a non-linear transformation to generate the output of a neuron.

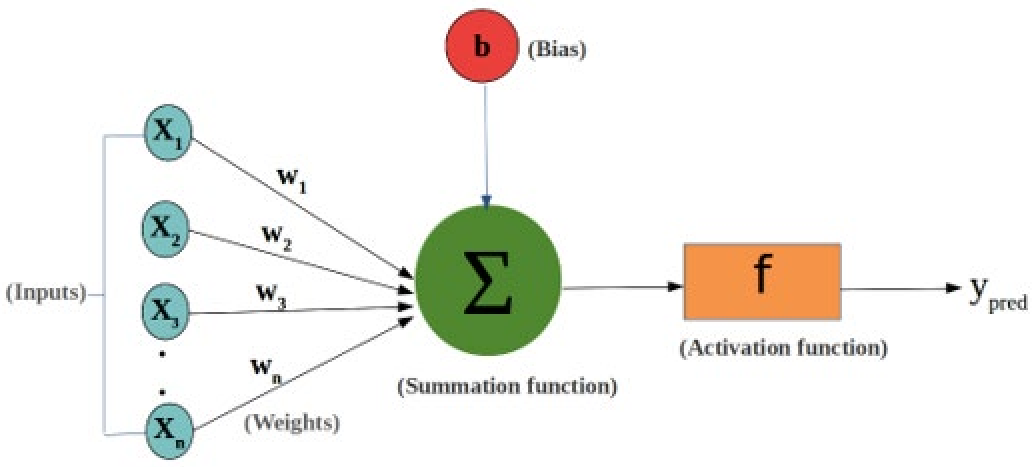


Figure 4: Plot of the activation function parameters

There are several types of activation functions, the most common of which are:

- **Sigmoid:** This activation function produces values between 0 and 1, making it useful for binary classification problems. However, it can suffer from the vanishing gradient problem, which can slow down learning. The vanishing gradient problem is a phenomenon that can occur during the training of artificial neural networks, where the gradients of the loss function with respect to the weights of the network become very small (i.e., close to zero) in certain layers. This can happen when an activation function used in a layer has a very small derivative, which can cause the gradients to decrease exponentially as they propagate through the network. This can make it difficult for the network to learn from the data, as the updates to the weights during training become too small to make a meaningful difference.

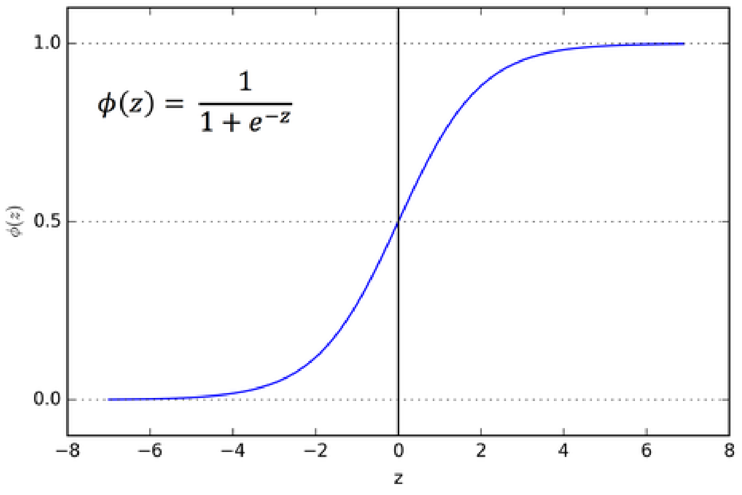


Figure 5: Plot of sigmoid function

- **Tanh (hyperbolic tangent):** This activation function produces values between -1 and 1, which can be helpful for classification problems with balanced data. Like the sigmoid function, it can also suffer from the vanishing gradient problem.

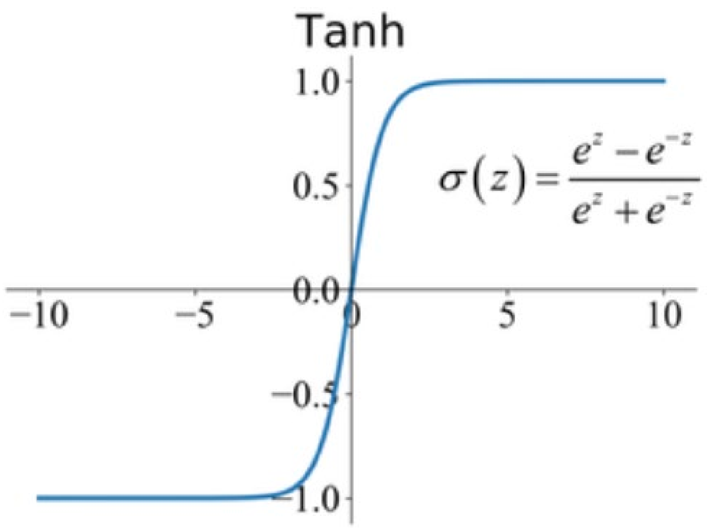


Figure 6: Plot of hyperbolic tangent function

- **ReLU (Rectified Linear Unit):** This activation function sets any negative inputs to 0 and leaves positive inputs unchanged. It is one of the most popular activation functions because it is simple and computationally efficient. The "dying ReLU" problem is a common issue that can occur when using the ReLU activation function in neural networks. The ReLU function returns a value of 0 for any negative input and returns the input value for any positive input. However, during training, some neurons in the network may end up receiving only negative input values, which causes the ReLU function to output 0. When this happens, the neuron is said to be "dead" and it will no longer participate in the training process since its gradient is always 0. If a large number of neurons become "dead," it can lead to a significant reduction in the network's capacity to learn and may result in poor performance. The "dying ReLU" problem can be especially problematic in deeper networks, where it can occur more frequently.

- **Leaky ReLU:** This activation function is similar to ReLU but allows small negative values, which can help prevent the dying ReLU problem.

- **Parametric ReLU (PReLU):** This activation function is similar to Leaky ReLU, but the negative slop is not fixed, but instead is learnable, which allows the network to adaptively change the slope during training.

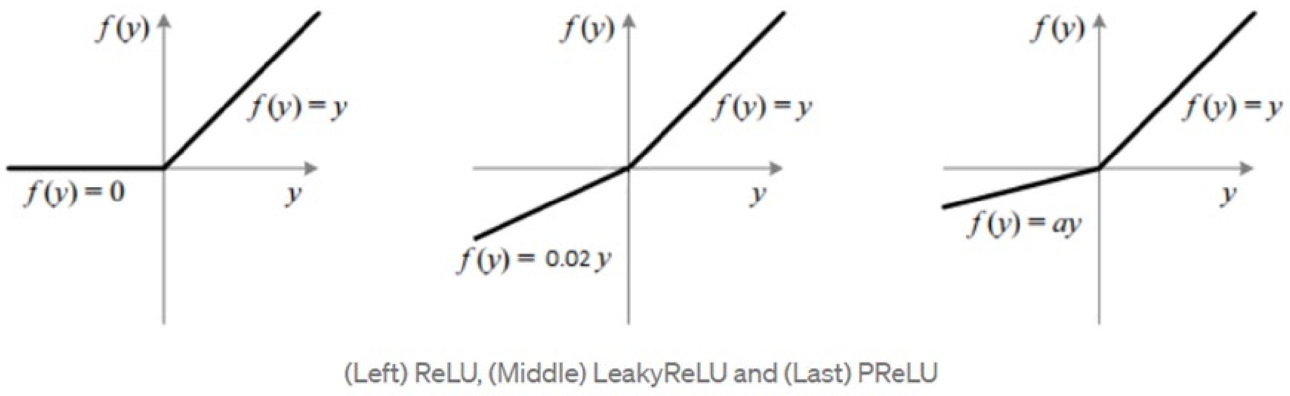


Figure 7: Plot of ReLU functions

2.4 Feedforward Neural Networks

Feedforward neural networks are a type of artificial neural network that processes information in a unidirectional way, meaning the input data flows through the network in a fixed direction from the input layer to the output layer without looping back on itself. These networks are commonly used in machine learning applications such as image classification, natural language processing and speech recognition. They are composed of multiple layers of interconnected nodes, with each node performing a simple mathematical function on the input it receives before passing it on to the next layer. The output of the final layer is then used to make a prediction or decision about the input data.

2.5 Feedback Neural Networks

Feedback neural networks, on the other hand, are neural networks that incorporate feedback connections, meaning they can loop back on themselves to influence their own output. These networks are also known as Recurrent Neural Networks (RNN) and are particularly useful for processing sequential data such as time series or natural language. In feedback networks, the output of a previous time step is fed back into the network as input for the current time step, allowing the network to learn patterns over time. Feedback neural networks have been used in a variety of applications, including speech recognition, language translation and image captioning.

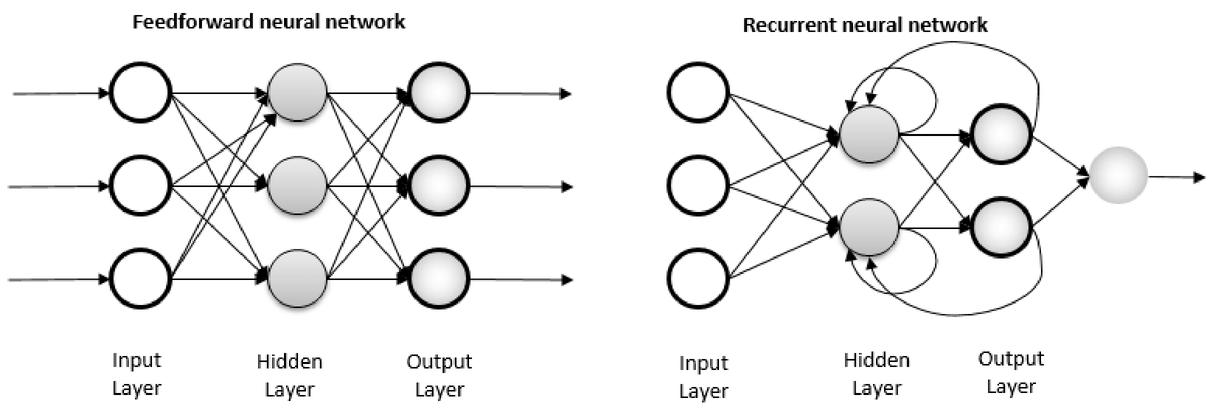


Figure 8: Feedforward and Recurrent Neural Networks architecture

3. Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a type of deep learning model commonly used for image recognition and classification tasks. They are inspired by the way the human visual cortex processes visual information, by detecting patterns and features at different levels of abstraction.

CNNs typically consist of multiple layers that perform different types of operations on the input image, such as convolution, pooling and non-linear activation functions. These layers work together to gradually transform the input image into a set of high-level features that can be used for classification or other tasks.

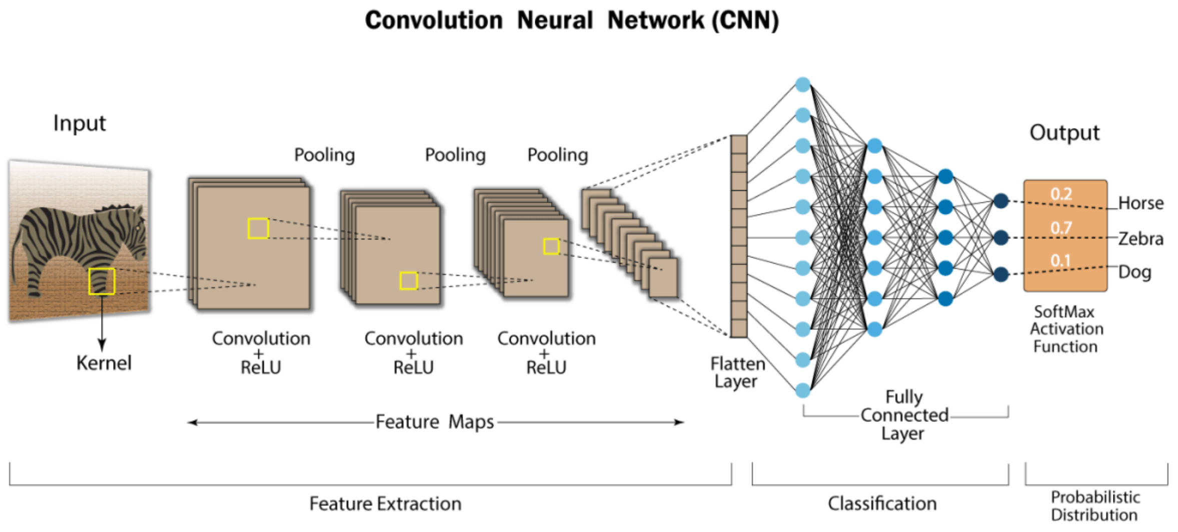


Figure 9: A typical Convolutional Neural Network

One important aspect of CNNs is their ability to learn features automatically from data, without the need for explicit feature engineering. This is achieved through the use of convolutional layers, which apply a set of filters to the input image to detect local patterns and features.

Overall, CNNs are a powerful and flexible tool for processing and analyzing visual data and have been used in a wide range of applications, including computer vision, natural language processing, and speech recognition.

3.1 Convolution

Convolution is a mathematical operation commonly used in signal processing, image processing and other related fields. The operation involves combining two functions to produce a third function that represents how one function modifies the other.

The basic idea of convolution is to slide one function (known as the kernel or filter) over the other function (known as the input signal) and compute the area of overlap between the two functions at each position. The result of the convolution operation is a new function that represents the modified input signal.

The convolution operation is often represented using mathematical notation, with the input signal represented by a function f(x) and the kernel represented by a function g(x). It is defined as:

$$(f * g)(x) = \int f(y)\, g(x - y)dy$$

where the integral is taken over all possible values of y. In other words, at each position x, the convolution operation involves multiplying the input signal f(y) by the kernel g(x - y), summing the product over all possible values of y, and assigning the result to the output function.

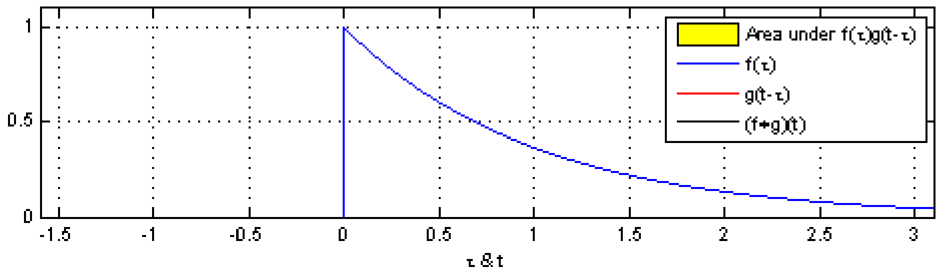


Figure 10: Depiction of convolution between two functions

3.2 Convolutional layers

Convolutional layers perform convolutions, which involve sliding a small, fixed-size filter (also called a kernel) over the input data (e.g., an image) and computing the dot product between the filter and the local region of the input it overlaps. This operation produces a single output value, which is typically added to a bias term and passed through a non-linear activation function (e.g., ReLU) to produce the output of the convolutional layer.

The main advantage of convolutional layers is their ability to capture local spatial patterns and hierarchies of features in the input data. By using multiple convolutional layers with increasing filter sizes and numbers of filters, CNNs can learn increasingly complex and abstract representations of the input data, ultimately enabling high-level recognition and classification tasks.

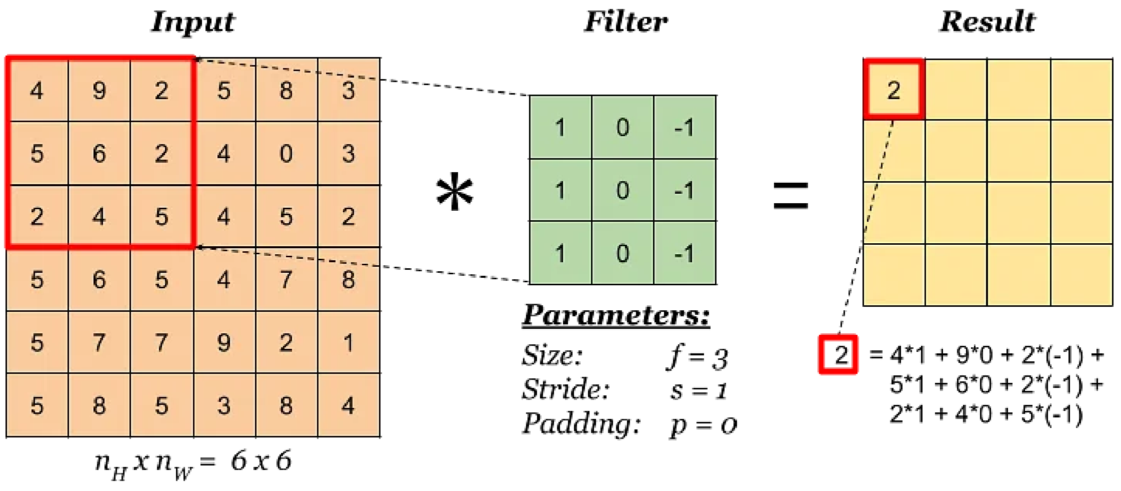


Figure 11: Convolutional Layer representation

Stride refers to the number of pixels by which the filter moves across the input during convolution. In other words, it determines the spacing between the locations at which the dot product is computed. A stride of 1 means that the filter moves by 1 pixel at a time, while a stride of 2 means that the filter moves by 2 pixels at a time. A larger stride can help to reduce the size of the output feature map and make the model more computationally efficient, but it may also lead to a loss of information.

Padding refers to the addition of extra pixels around the input, usually with a value of 0, in order to ensure that the output feature map has the same spatial dimensions as the input. Padding can be important for preventing the loss of information at the edges of the input, which can occur when the filter extends beyond the edges of the input. The amount of padding is usually determined by the size of the filter and the desired output spatial dimensions.

3.2 Pooling layers

Pooling layers operate on the output of convolutional layers by downsampling the spatial dimensions of the feature maps, while retaining their channel depth. This is typically achieved by dividing the feature maps into non-overlapping regions (e.g., 2x2 or 3x3) and replacing each region with a single value that summarizes its content. Common pooling operations include max pooling (taking the maximum value in each region), average pooling (taking the average value) and others.

The main purpose of pooling layers is to reduce the dimensionality of the feature maps, making them more computationally efficient to process and less prone to overfitting. Pooling can also help to increase the invariance of the network to small translations and distortions in the input data, since the pooling operation is relatively insensitive to small changes in the input.

However, too much pooling can also lead to loss of spatial information and make it more difficult for the network to localize objects accurately. Therefore, the number and size of pooling layers, as well as the pooling operation used, are typically chosen based on the specific requirements of the task and the available computational resources.

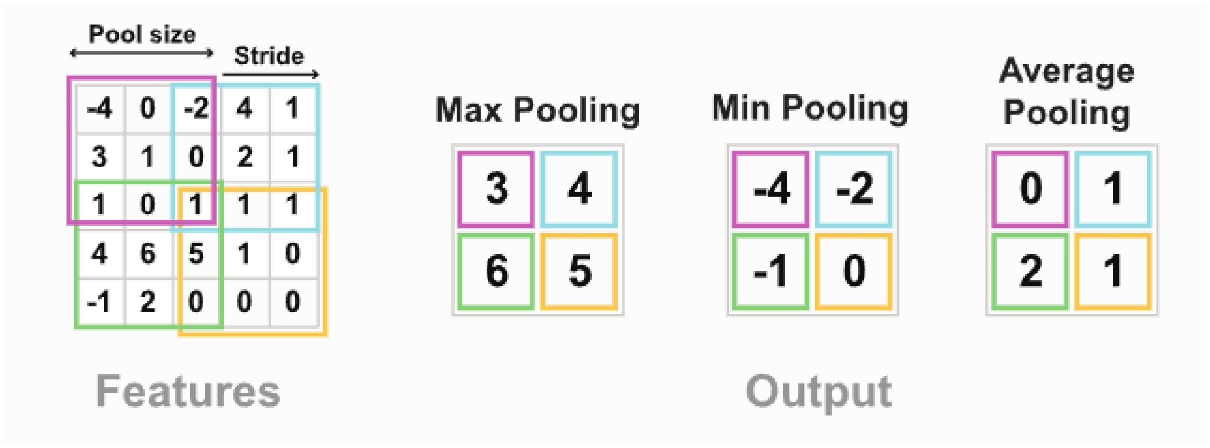


Figure 12: Pooling Layer representation

3.3 Fully connected layers

Fully connected layers, also known as dense layers, are a type of neural network layer that connects every neuron in one layer to every neuron in the next layer. They are often used as the final layer in neural networks to map the learned features to the output labels.

The purpose of fully connected layers is to perform a nonlinear mapping from the high-level features learned by the previous convolutional and pooling layers to the output classes or values. This enables the network to make complex, non-linear decisions based on the learned features. However, fully connected layers can also introduce a large number of parameters into the network, making it more prone to overfitting and requiring more training data and computational resources.

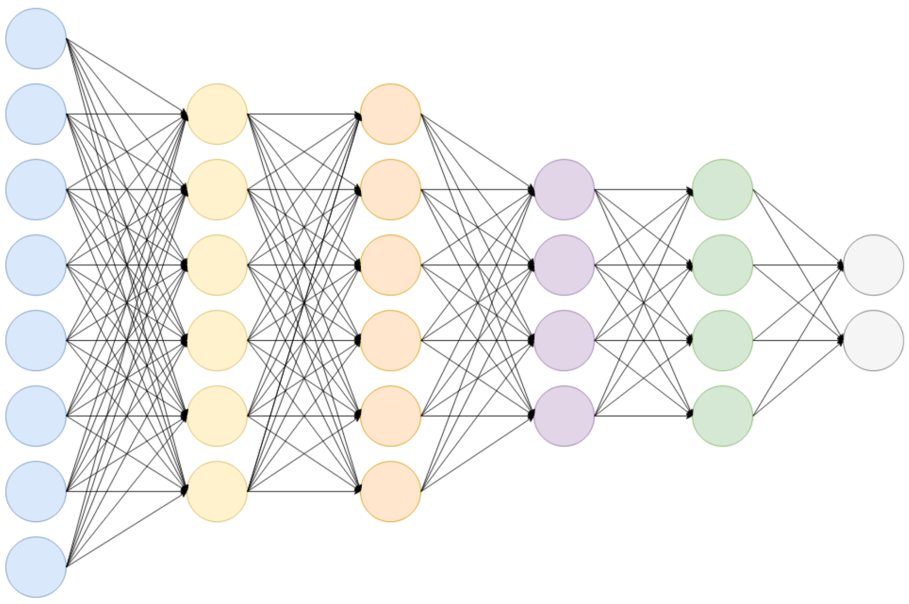


Figure 13: Fully connected layers representation

4. Training

Training a CNN involves several steps, each of which requires careful consideration and tuning to achieve optimal performance. With the right architecture, data preprocessing, hyperparameter tuning and optimization techniques, a well-trained CNN can achieve state-of-the-art performance on a wide range of tasks.

4.1 Supervised Learning

Supervised learning is a type of neural network training in which the model learns to predict output values based on input data and known output values. In supervised learning, the neural network is provided with labeled training data, which consists of input data and corresponding output data. The network then learns to map the input data to the correct output values by adjusting the weights and biases of the neurons. It is well-suited for problems where there is a clear mapping between the input and output data.

4.2 Unsupervised Learning

Unsupervised learning, on the other hand, is a type of neural network training in which the model learns to identify patterns and structure in unlabeled data. In unsupervised learning, the neural network is provided with input data but without corresponding output data.

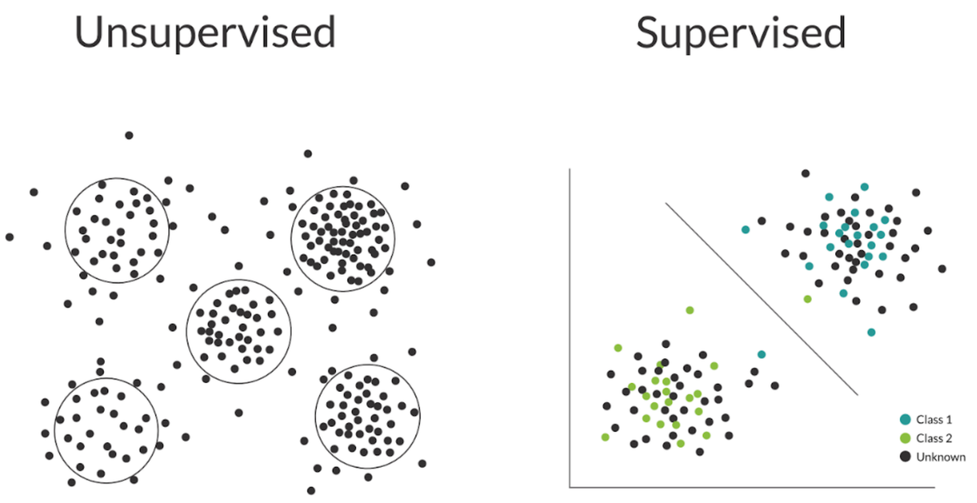


Figure 14: Supervised vs Unsupervised Learning

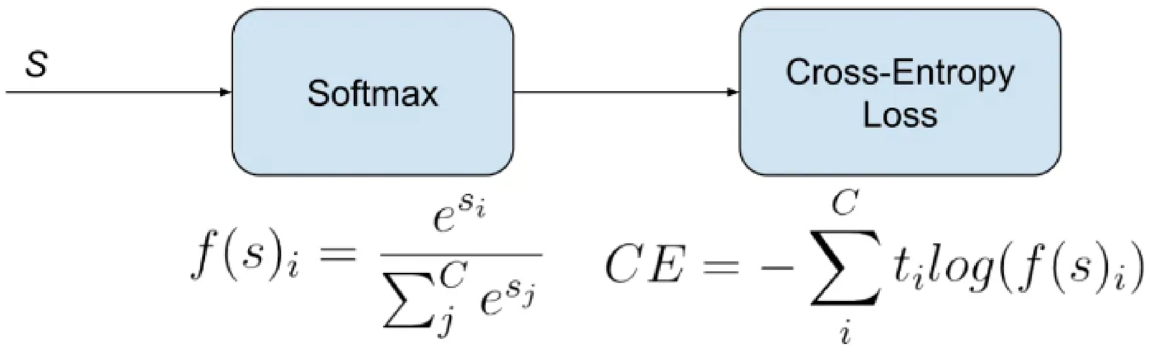
4.3 Loss function

The loss function plays a critical role in the training of neural networks. It is a measure of how well the neural network is performing on a given task and its optimization is the primary objective during training.

The loss function is essentially a mathematical function that calculates the difference between the predicted output of a neural network and the actual output. This difference is known as "error" or "loss". The objective of training is to minimize this error or loss, which means the neural network is getting closer to the desired output.

There are various types of loss functions and the choice of loss function depends on the type of task the neural network is being trained to perform. For example, in a classification task, the most commonly used loss function is the cross-entropy loss. In regression tasks, mean squared error or mean absolute error loss functions are commonly used.

The choice of loss function also depends on the type of activation function used in the neural network's output layer. For example, if the output layer uses a softmax activation function, cross-entropy loss is a suitable choice. On the other hand, if the output layer uses a linear activation function, mean squared error loss is more appropriate.



$$f(s)_i = \frac{e^{s_i}}{\sum_j^C e^{s_j}} \qquad CE = -\sum_i^C t_i log(f(s)_i)$$

Figure 15: Cross-Entropy Loss function's mathematical notation and the corresponding Softmax activation function

4.4 Backpropagation

Backpropagation is a method of computing the gradient of a neural network's loss function with respect to its weights. This gradient can then be used to update the weights in order to minimize the loss function.

The backpropagation algorithm works by propagating the error backwards through the neural network, starting from the output layer and moving towards the input layer. At each layer, the error is split among the neurons in proportion to their contribution to the output of the layer and then the gradients are computed with respect to the weights and biases of the layer.

Once the gradients have been computed for all layers, the weights and biases can be updated using a technique such as gradient descent or one of its variants. This process is repeated many times until the network converges to a set of weights and biases that minimize the loss function.

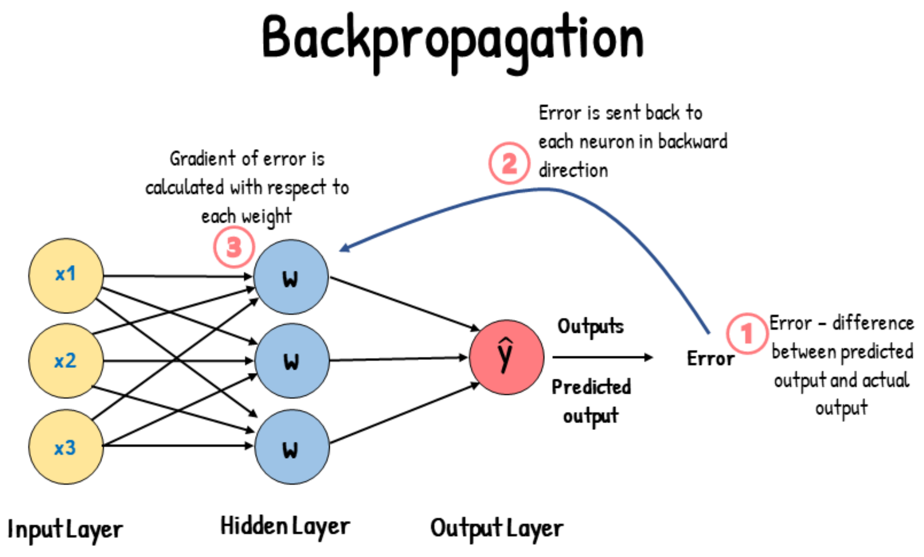

Backpropagation
Gradient of error is calculated with respect to each weight
3
Error is sent back to each neuron in backward direction
2
x1
x2
x3
w
w
w
ŷ
Outputs
Predicted output
Error
1
Error – difference between predicted output and actual output
Input Layer
Hidden Layer
Output Layer

Figure 16: Representation of Backpropagation

4.5 Overfitting / Underfitting

Overfitting and underfitting are common problems in the training of neural networks. Overfitting occurs when a model is trained to fit the training data so closely that it performs poorly on new, unseen data. This is often due to the model being too complex or having too many parameters, allowing it to memorize the training data rather than learning generalizable patterns. On the other hand, underfitting occurs when a model is too simple and unable to capture the underlying patterns in the data. This often leads to poor performance on both the training and test data. To avoid overfitting, techniques such as regularization, early stopping and dropout can be used, while to avoid underfitting, increasing model complexity, gathering more data and adjusting hyperparameters can be helpful.

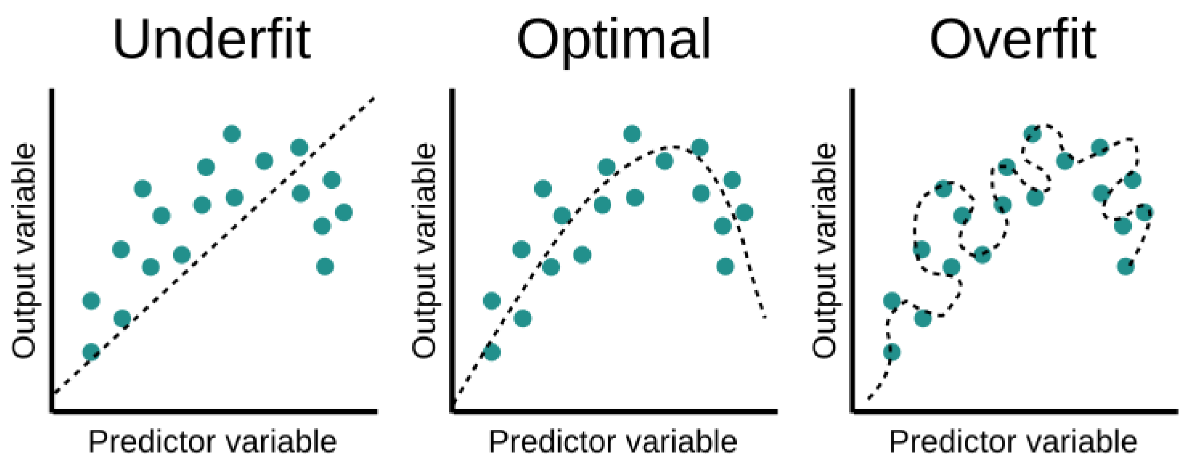


Figure 16: Overfitting and Underfitting

4.6 Hyperparameters

Hyperparameters in neural network training are parameters that cannot be learned during the training process, but must be set before training begins. These parameters govern the architecture of the network and the details of the learning algorithm. Some common hyperparameters in neural network training include:

- **Learning rate:** determines how quickly the model should update its parameters based on the error during training.
- **Number of epochs:** determines the number of times the training process should loop through the entire training dataset.
- **Batch size:** determines the number of training examples used in each iteration of the optimization algorithm.
- **Number of hidden layers:** determines the number of layers of neurons in the neural network.
- **Number of neurons in each hidden layer:** determines the number of neurons in each layer of the neural network.
- **Activation functions:** determines the function applied to the output of each neuron in the neural network.
- **Regularization:** determines the degree of regularization applied to the neural network to avoid overfitting.
- **Optimization algorithm:** determines the specific algorithm used to update the weights and biases of the neural network during training, such as stochastic gradient descent or Adam.

Choosing appropriate hyperparameters is crucial for successful neural network training, as poorly chosen hyperparameters can lead to overfitting or slow convergence during training.

4.7 Regularization techniques

Regularization techniques are used in neural network training to prevent overfitting, which occurs when the model becomes too complex and fits the training data too closely, resulting in poor performance on new data. Some common regularization techniques in neural network training include:

- **L1 and L2 regularization:** These methods add a penalty term to the loss function that discourages large weights in the network. L1 regularization adds the absolute values of the weights, while L2 regularization adds the squared values of the weights.
- **Dropout:** This technique randomly drops out some of the neurons during training, which helps prevent overfitting by reducing co-adaptation between neurons.
- **Early stopping:** This method stops the training process before it reaches the maximum number of epochs if the performance on a validation set has stopped improving.
- **Data augmentation:** This involves creating new training examples by applying random transformations to the existing data, such as flipping, rotating, or adding noise. This can help the model generalize better to new data.
- **DropConnect:** This method is similar to dropout but instead of dropping out neurons, it drops out connections between neurons.

These techniques can be used individually or in combination to improve the performance and generalization ability of neural networks.

4.8 Optimization algorithms

There are various optimization algorithms that can be used in neural network training. The most commonly used and most effective ones are:

- **Stochastic Gradient Descent (SGD):**

SGD is used to minimize the loss function in neural network training. It updates the weights of the neural network in the direction of the negative gradient of the loss function. In other words, it calculates the gradient of the loss function with respect to the weights and adjusts the weights in the direction that decreases the loss. SGD updates the weights after processing each mini-batch of training examples, which makes it faster than regular gradient descent. However, because it updates the weights in small steps, it can take longer to converge to the minimum of the loss function.

- **Adaptive Moment Estimation (Adam):**

Adam is an adaptive optimization algorithm that adjusts the learning rate for each weight based on the historical gradient information. It computes the learning rate for each weight by combining the estimates of the first and second moments of the gradients. In other words, it uses a moving average of the gradients and the squared gradients to adjust the learning rate. This makes Adam an effective optimization algorithm for problems with noisy and sparse gradients. Adam also uses bias correction to adjust the estimates of the moments during the early stages of training.

5. You Only Look Once (YOLO)

5.1 Original YOLO

YOLO, which stands for "You Only Look Once," is a real-time object detection system that has revolutionized the field of computer vision. It was developed by Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi in May 2016 and has since become one of the most popular algorithms for object detection.

YOLO is different from traditional object detection algorithms in that it performs detection and classification in a single stage, as opposed to two-stage algorithms such as Faster R-CNN or SSD. The entire detection process consists of three steps: resizing the input image to 448 x 448, running a single CNN on the complete image and then thresholding the resulting detections by the model's confidence, thereby removing duplicate detections.

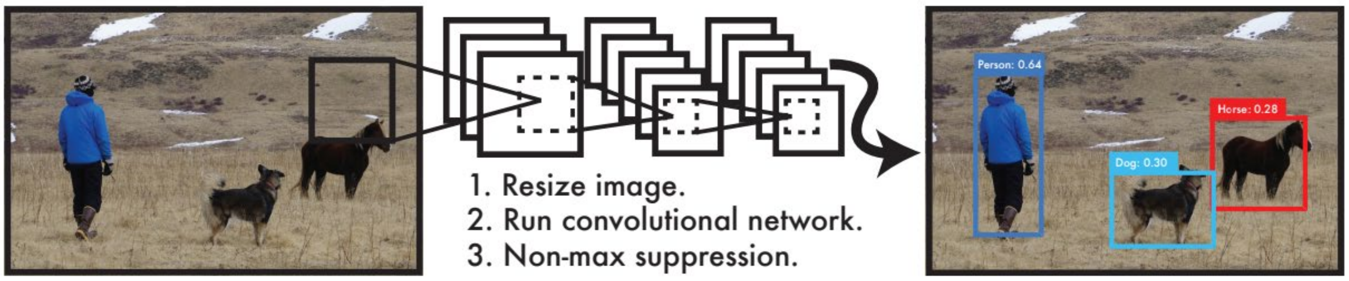


Figure 17: The YOLO detection system

5.1.1 Definitions

The below definitions that are used in YOLO are critical for the understanding of the way the model works:

- **Intersection over Union (IoU):**

IoU (Intersection over Union) is a commonly used evaluation metric in computer vision tasks, particularly in object detection and segmentation. It measures the overlap between two sets of bounding boxes or regions of interest (ROIs) and determines how well the predicted bounding boxes or regions match the ground truth annotations.

To calculate the IoU, the intersection between the predicted and ground truth bounding boxes/regions is divided by the union of the two regions. The result is a value between 0 and 1, where a score of 1 indicates a perfect match between the predicted and ground truth regions.

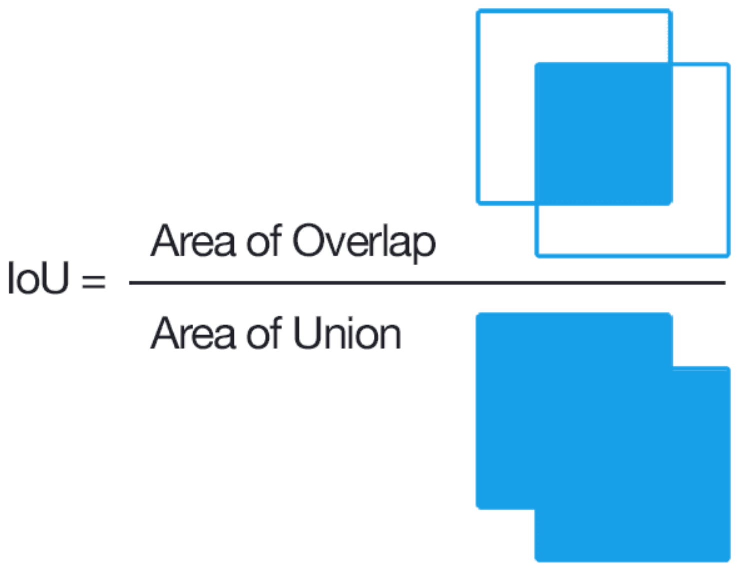


Figure 18: Intersection over Union calculation

- **Mean Average Precision (mAP)**

mAP is a measure of the accuracy and completeness of a model's predictions, taking into account both the precision and recall of the predictions. It is calculated by first computing the average precision (AP) for each class and then taking the mean of those values. AP is a measure of how well the model correctly identifies objects in an image for a particular class, considering all possible thresholds for the confidence score of the predictions.

To calculate AP, the model's predictions are sorted by their confidence scores and then the precision and recall are computed at each threshold. Precision is the ratio of true positive predictions to the total number of positive predictions, while recall is the ratio of true positive predictions to the total number of actual positive instances in the dataset. The precision-recall curve is then plotted and the area under the curve (AUC) is computed, which represents the AP.

$$\textbf{Precision} = \frac{\text{True Positive}}{\text{Actual Results}} \quad \text{or} \quad \frac{\text{True Positive}}{\text{True Positive + False Positive}}$$

$$\textbf{Recall} = \frac{\text{True Positive}}{\text{Predicted Results}} \quad \text{or} \quad \frac{\text{True Positive}}{\text{True Positive + False Negative}}$$

$$\textbf{Accuracy} = \frac{\text{True Positive + True Negative}}{\text{Total}}$$

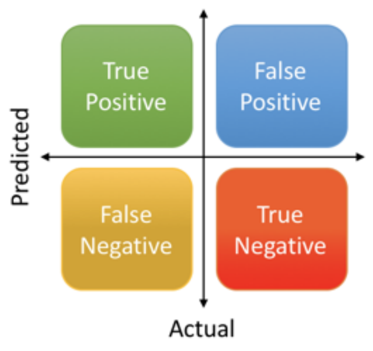


Figure 18: Precision and Recall definitions

mAP is an important metric because it provides an overall measure of a model's performance across all classes, rather than just for a single class. A higher mAP indicates better performance in identifying objects in an image.

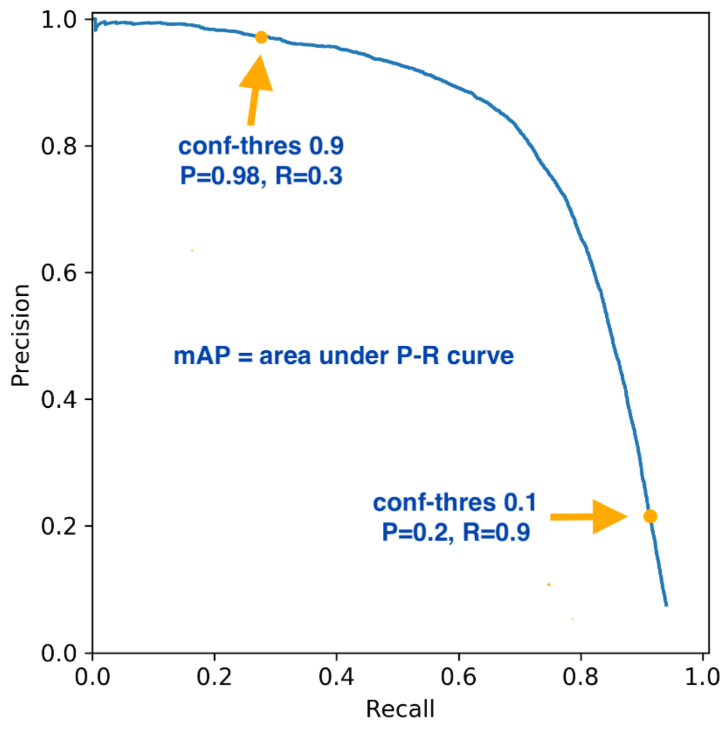


Figure 19: Precision and Recall curve

- **Non-maximum suppression**

Non-maximum suppression (NMS) is a technique commonly used in object detection and computer vision to eliminate duplicate detections of the same object. It involves selecting the highest-scoring bounding box that overlaps with other bounding boxes above a certain threshold and suppressing the rest. This threshold is typically based on the IoU metric. By using NMS, we can obtain a more accurate and robust object detection output.

5.1.2 Architecture

The original YOLO model consists of 24 Convolutional layers with intermediate Maxpool layers, which do the image feature extraction, followed by two fully connected layers that predict the output probabilities and coordinates.

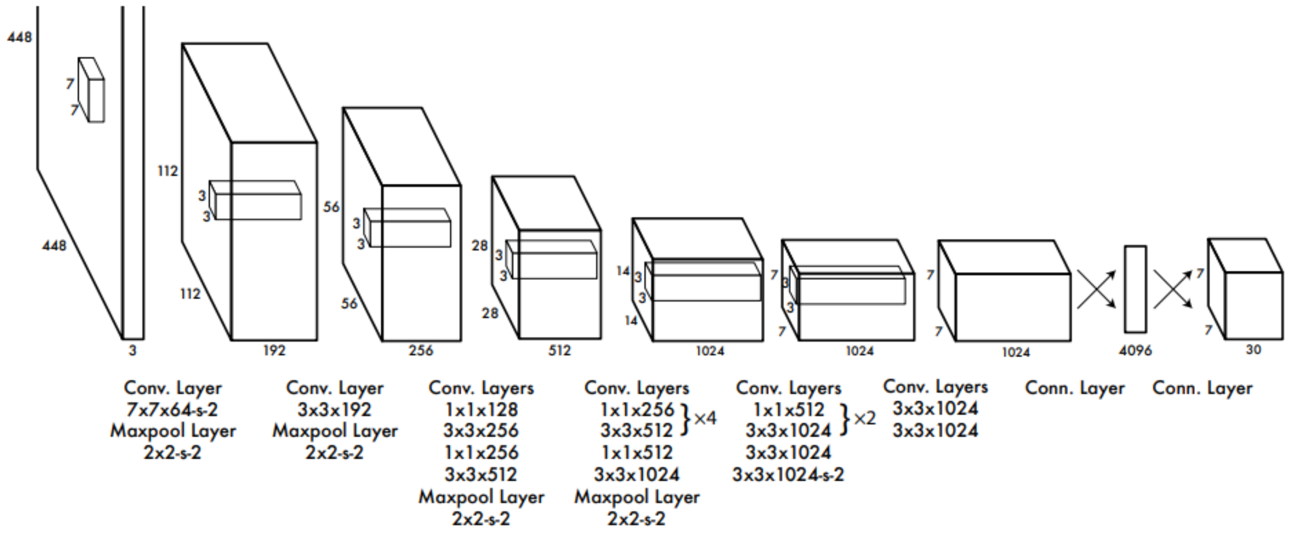

448
448
3
112
112
192
56
56
256
28
28
512
14
14
1024
7
7
1024
1024
4096
30
Conv. Layer 7x7x64-s-2 Maxpool Layer 2x2-s-2
Conv. Layer 3x3x192 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x128 3x3x256 1x1x256 3x3x512 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x256 3x3x512 }×4 1x1x512 3x3x1024 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x512 3x3x1024 }×2 3x3x1024 3x3x1024-s-2
Conv. Layers 3x3x1024 3x3x1024
Conn. Layer
Conn. Layer

Figure 20: Original YOLO architecture

5.1.3 Loss function

The loss function that is used in YOLO is based on the sum-squared error function, mainly because it was easier to optimize. It was retrofitted in a way to remedy its weaknesses in order to produce the highest mAP possible.

More specifically, the sum-squared error function weights localization error equally with classification error, which is not optimal. Also, in every image many grid cells do not contain any object, which pushes the “confidence” scores of those cells towards zero, often overpowering the gradient from cells that do contain objects. This can lead to model instability, causing training to diverge early on. To face this issue, the loss from bounding box coordinate predictions was increased, while the loss from confidence predictions for boxes that don’t contain objects was decreased, using two parameters, λ_{coord}, which is set to 5, and λ_{noobj}, which is set to 0.5.

Another issue of the sum-squared error is that it weights errors equally in large and small boxes. In reality, small deviations in large boxes matter less than in small boxes, so to address this, the YOLO loss function predicts the square root of the bounding box dimensions instead.

The final loss function is:

$$
\begin{aligned}
&\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
&+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\
&+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2 \\
&+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2 \\
&+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2
\end{aligned}
$$

Figure 21: YOLO loss function

where I_i^{obj} denotes if an object appears in cell $\boldsymbol{i}$ and I_{ij}^{obj} denotes that jth bounding box predicted in cell $\boldsymbol{i}$ is responsible for that prediction

5.2 YOLOv2

The 2nd version of YOLO, or as the paper was called "YOLO9000: Better, Faster, Stronger", is an improvement to the original YOLO model, developed by Joseph Redmon and Ali Farhadi in December 2016. The new features that are proposed try to face the weaknesses of the original YOLO model, such as the significant amount of localization errors that it produces and its low recall compared to region proposal-based methods.

In order to address these issues, YOLOv2 uses, among others, the below features:

- **Batch normalization:** it is a technique used to normalize the input of each layer of a neural network. It works by subtracting the mean and dividing by the standard deviation of the values in each batch of data. This helps to stabilize the training process by reducing the internal covariate shift problem, which occurs when the distribution of the input to a layer changes during training. By normalizing the input to each layer, batch normalization can improve the performance of the network, reduce overfitting, and speed up training. By adding batch normalization to all of the convolutional layers, there is an increase of at least 2% on the mAP of the model.

- **Convolutional with Anchor Boxes:** The YOLO algorithm predicts bounding box coordinates directly using fully connected layers on top of the convolutional feature extractor. By predicting offsets instead of coordinates, the problem is simplified and easier for the network to learn. To use anchor boxes in YOLOv2, the fully connected layers are removed, and one pooling layer is eliminated to increase the resolution of the convolutional layers. The network is also shrunk to operate on 416 input images, resulting in an output feature map of 13 × 13. With anchor boxes, the class and objectness predictions are decoupled from the spatial location, and every anchor box is used for class and objectness prediction. Although the use of anchor boxes results in a small decrease in accuracy, the increase in recall indicates that the model has more potential for improvement. The YOLO algorithm predicts only 98 boxes per image, whereas YOLOv2 model predicts more than a thousand.

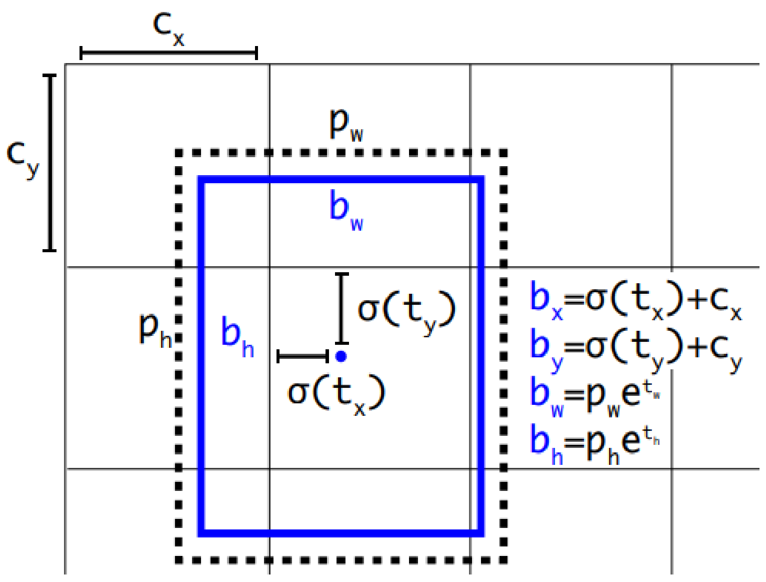


Figure 22: Bounding box dimensions and location with the use of anchor boxes

- **Multi-Scale Training:** In the original YOLO algorithm, the input resolution was set to 448 × 448. However, with the addition of anchor boxes, the resolution was changed to 416 × 416. As the YOLOv2 model only uses convolutional and pooling layers, it can be resized dynamically, making it robust to running on images of different sizes. To train this into the model, the network is changed every few iterations, with the input image size randomly chosen every 10 batches from multiples of 32 ranging from 320 × 320 to 608 × 608. This approach forces the network to learn to predict well across a variety of input dimensions, allowing the same network to predict detections at different resolutions. YOLOv2 offers a trade-off between speed and accuracy, with the network running faster at smaller sizes.

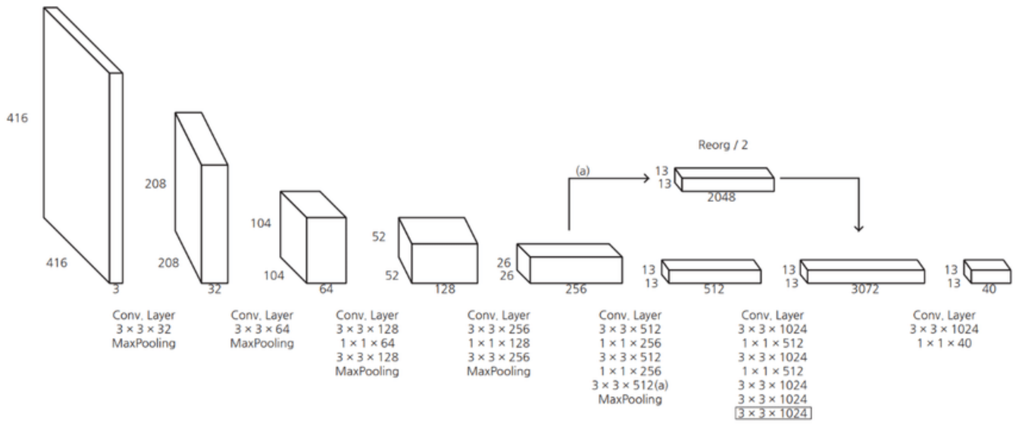

416
416
3
208
208
32
104
104
64
52
52
128
26
26
256
Reorg / 2
(a)
13
13
2048
13
13
512
13
13
3072
13
13
40
Conv. Layer 3 × 3 × 32 MaxPooling
Conv. Layer 3 × 3 × 64 MaxPooling
Conv. Layer 3 × 3 × 128 1 × 1 × 64 3 × 3 × 128 MaxPooling
Conv. Layer 3 × 3 × 256 1 × 1 × 128 3 × 3 × 256 MaxPooling
Conv. Layer 3 × 3 × 512 1 × 1 × 256 3 × 3 × 512 1 × 1 × 256 3 × 3 × 512(a) MaxPooling
Conv. Layer 3 × 3 × 1024 1 × 1 × 512 3 × 3 × 1024 1 × 1 × 512 3 × 3 × 1024 3 × 3 × 1024 3 × 3 × 1024
Conv. Layer 3 × 3 × 1024 1 × 1 × 40

Figure 23: YOLOv2 architecture

5.3 YOLOv3

YOLOv3 (You Only Look Once version 3) was developed by Joseph Redmon and Ali Farhadi in April 2018. It is the third iteration of the YOLO algorithm, which stands out for its high accuracy and real-time performance.

The most significant improvement over the previous versions is the use of a feature extractor called Darknet-53, which has 53 convolutional layers and is capable of extracting more complex features than previous versions, thus significantly improving accuracy. In addition, YOLOv3 uses three different scales to detect objects, which allows it to identify objects of different sizes and shapes more accurately.

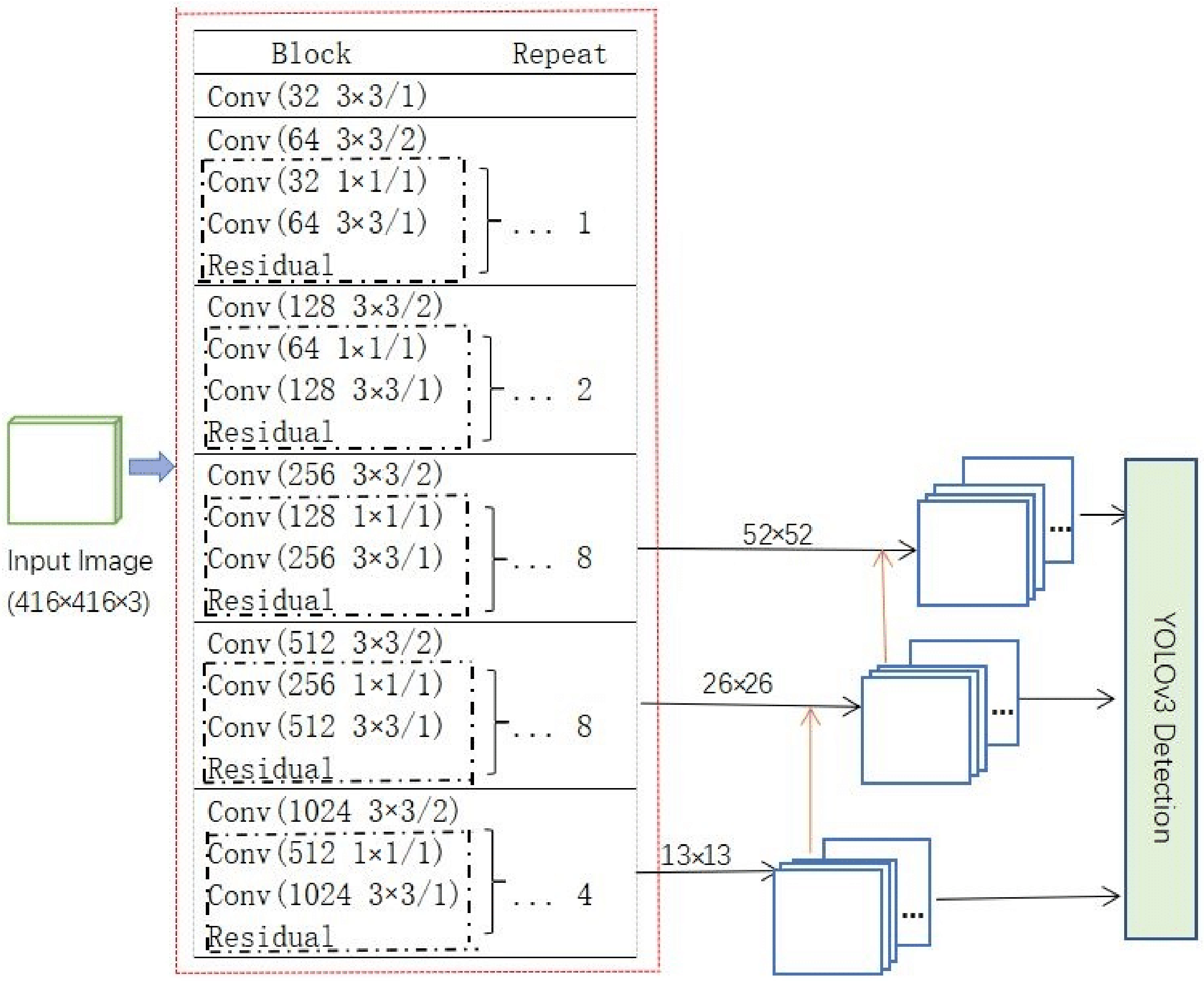


Figure 24: YOLOv3 architecture with Darknet-53 backbone and the three prediction scales

Another major improvement in YOLOv3 is its ability to detect objects at different levels of granularity. YOLOv3 can detect objects at the level of individual pixels, which means it can accurately locate small objects even in cluttered scenes. It is also designed for real-time performance, with the ability to process up to 60 frames per second on a conventional GPU.

5.4 YOLOv4

YOLOv4 (You Only Look Once version 4), the fourth iteration of the YOLO model, was developed by Alexey Bochkovskiy with the help of Chien-Yao Wang and Hong-Yuan Mark Liao in April 2020, since the YOLOv3 was an open-source model, available to everyone for experimenting and improving.

Yolov4 introduces two new sets of features that are used to optimize training and detection, called "bag of freebies" and "bag of specials":

- **Bag of freebies:** Typically, a conventional object detector is trained offline, which motivates researchers to develop better training methods that can enhance the object detector's accuracy without increasing inference cost. These methods are known as "bag of freebies," which refers to strategies that only modify the training procedure or increase its cost. One common bag of freebies approach used in object detection methods is data augmentation, which aims to increase the variability of input images to improve the model's robustness to different environmental conditions.
  There are several data augmentation methods, including photometric distortions and geometric distortions, which make pixel-wise adjustments to images. Additionally, some researchers use data augmentation to simulate object occlusion issues, achieving good results in image classification and object detection. Other data augmentation methods involve using multiple images together, such as MixUp and CutMix, to adjust labels and reduce texture bias learned by CNN.
  Some bag of freebies methods address the problem of data imbalance between different classes, which can cause semantic distribution bias. Hard negative example mining and online hard example mining are often used in two-stage object detectors, but they are not applicable to one-stage detectors. To address data imbalance, focal loss and label smoothing have been proposed. Knowledge distillation and label refinement networks have also been introduced to refine soft labels.

Finally, the objective function of bounding box (BBox) regression is another bag of freebies method. Traditional object detectors use mean square error (MSE) to perform regression directly on the center point coordinates and height and width of the BBox. IoU loss has been proposed to address the issue of treating points as independent variables, with researchers continuing to improve it with GIoU loss and DIoU loss.

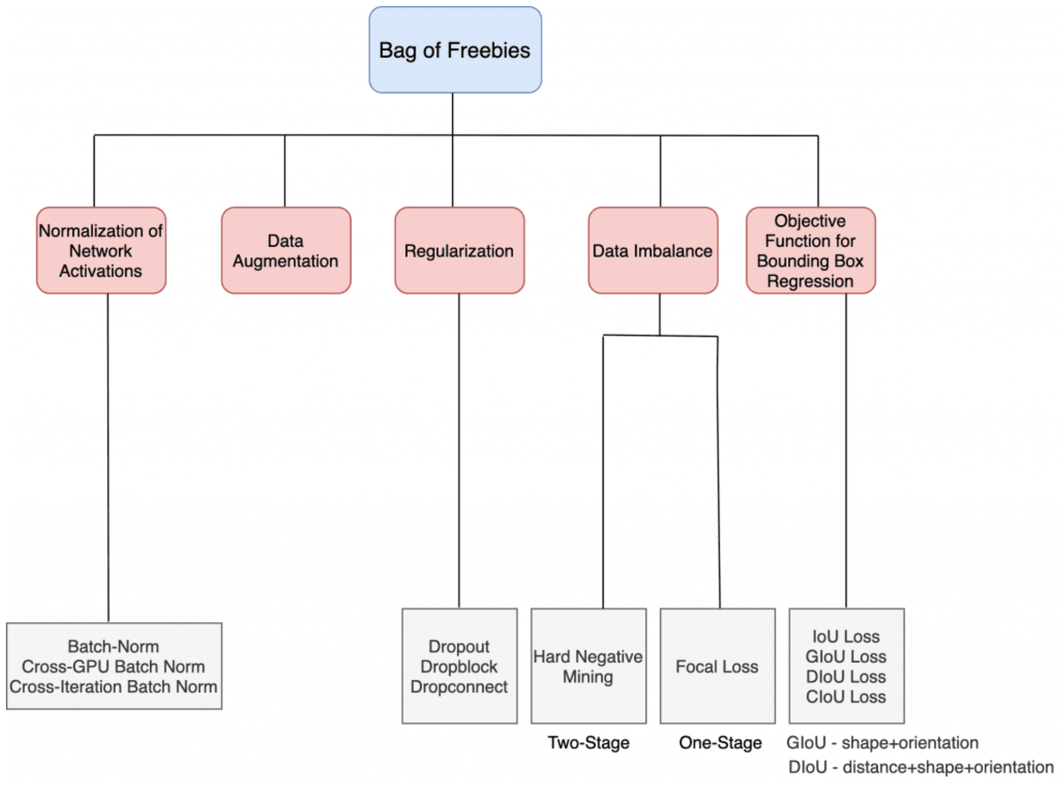


Figure 25: Bag of Freebies methods

- **Bag of Specials:** We refer to plugin modules and post-processing methods as "bag of specials" if they increase the inference cost slightly but greatly enhance the accuracy of object detection. These modules can be used to improve specific attributes of a model such as feature integration capability, receptive field, or attention mechanism.
  The Bag of Specials includes various plugin modules that can enhance the receptive field of a model, such as Spatial Pyramid Pooling (SPP), Atrous Spatial Pyramid Pooling (ASPP) and Receptive Field Block (RFB).
  SPP was originally developed as a Spatial Pyramid Matching (SPM) method and later integrated into convolutional neural networks (CNNs) using max-pooling instead of a bag-of-word operation. However, its one-dimensional output made it infeasible to use in Fully Convolutional Networks (FCNs). Thus, the SPP module was improved by concatenating max-pooling outputs with kernel sizes ranging from 1 to 13 and stride equal to 1. This design allows for larger max-pooling kernels, which effectively increase the receptive field of backbone features.
  Another common type of plugin module used in object detection is the attention module. Attention modules can be divided into channel-wise and point-wise attention, represented by Squeeze-and-Excitation (SE) and Spatial Attention Module (SAM), respectively. SE modules can improve the accuracy of ResNet50 in the ImageNet image classification task by 1% top-1 accuracy at the cost of only increasing the computational effort by 2%. However, on a GPU, SE modules can increase inference time by approximately 10%. On the other hand, SAM only requires 0.1% extra calculation and can improve ResNet50-SE 0.5% top-1

accuracy on the ImageNet image classification task without affecting the speed of inference on the GPU.
Finally, the Bag of Specials includes various post-processing methods, with Non-Maximum Suppression (NMS) being the most commonly used in deep learning-based object detection. NMS can be used to filter out bounding boxes that predict the same object poorly and only retain candidate bounding boxes with higher response. This approach can significantly improve object detection accuracy by reducing false positives.

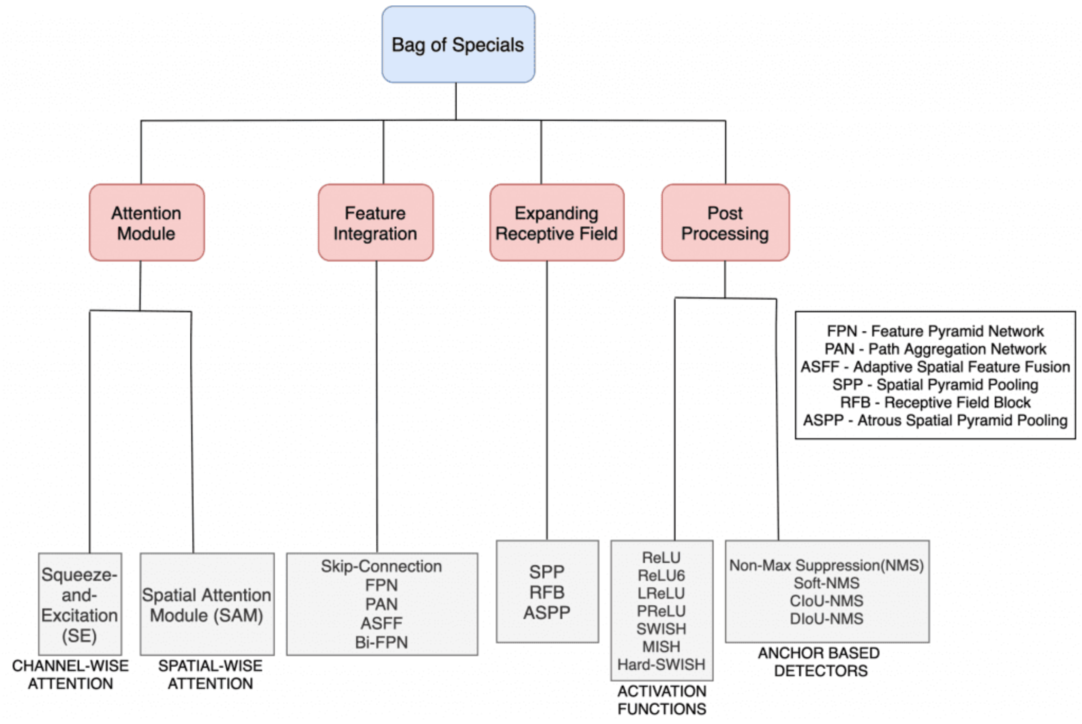

Bag of Specials
Attention Module
Feature Integration
Expanding Receptive Field
Post Processing
FPN - Feature Pyramid Network
PAN - Path Aggregation Network
ASFF - Adaptive Spatial Feature Fusion
SPP - Spatial Pyramid Pooling
RFB - Receptive Field Block
ASPP - Atrous Spatial Pyramid Pooling
Squeeze-and-Excitation (SE)
CHANNEL-WISE ATTENTION
Spatial Attention Module (SAM)
SPATIAL-WISE ATTENTION
Skip-Connection FPN PAN ASFF Bi-FPN
SPP RFB ASPP
ReLU ReLU6 LReLU PReLU SWISH MISH Hard-SWISH
ACTIVATION FUNCTIONS
Non-Max Suppression(NMS) Soft-NMS CIoU-NMS DIoU-NMS
ANCHOR BASED DETECTORS

Figure 25: Bag of Specials modules

YOLOv4 consists of:

- **Backbone:** CSPDarknet53
- **Neck:** SPP, Path Aggregation Network (PAN)
- **Head:** Yolov3

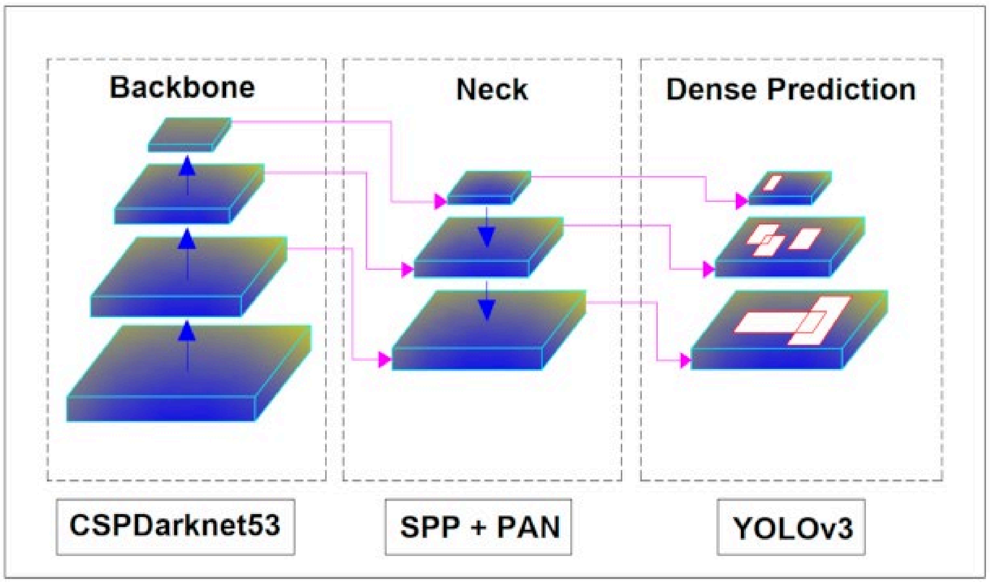


Figure 26: YOLOv4 architecture

YOLOv4 uses:

- **Bag of Freebies for backbone:** Cutmix and Mosaic data augmentation, DropBlock regularization, Class label smoothing
- **Bag of Specials for backbone:** Mish activation, Cross-stage partial connections (CSP), Multi-input weighted residual connections (MiWRC)
- **Bag of Freebies for detector:** CIoU-loss, Cross mini-Batch Normalization (CmBN), DropBlock regularization, Mosaic data augmentation, Self-Adversial Training, Eliminate grid sensitivity, Using multiple anchors for a single ground truth, Cosine annealing scheduler, Optimal hyper-parameters, Random training shapes
- **Back of Specials for detector:** Mish activation, SPP-block, SAM-block, PAN path-aggregation block, DIoU-NMS

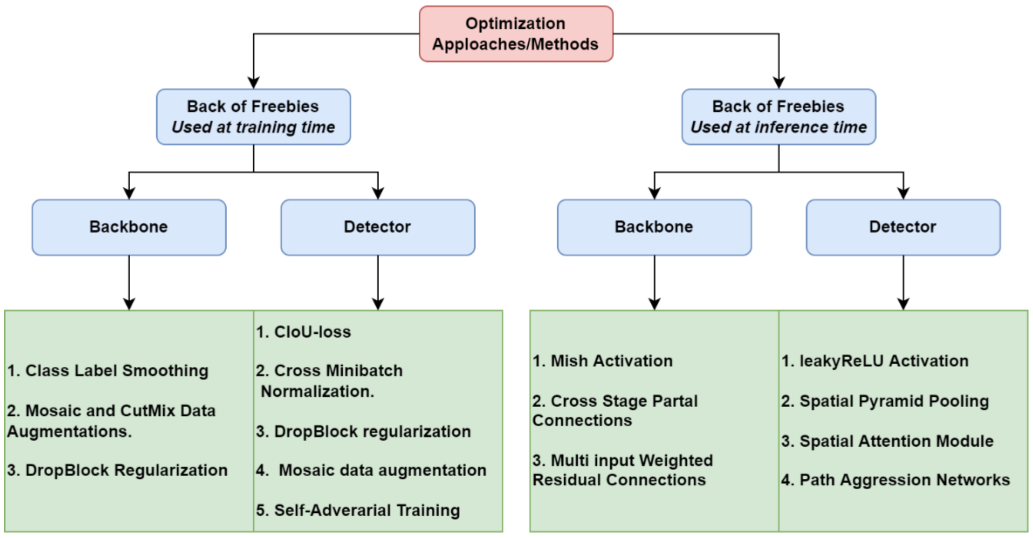

Optimization Apploaches/Methods
Back of Freebies *Used at training time*
Back of Freebies *Used at inference time*
Backbone
Detector
Backbone
Detector
1. Class Label Smoothing
2. Mosaic and CutMix Data Augmentations.
3. DropBlock Regularization
1. CIoU-loss
2. Cross Minibatch Normalization.
3. DropBlock regularization
4. Mosaic data augmentation
5. Self-Adverarial Training
1. Mish Activation
2. Cross Stage Partal Connections
3. Multi input Weighted Residual Connections
1. leakyReLU Activation
2. Spatial Pyramid Pooling
3. Spatial Attention Module
4. Path Aggression Networks

Figure 27: YOLOv4 BoF and BoS

5.5 YOLOv4-tiny

YOLOv4-tiny is the compressed version of YOLOv4 designed to train on machines that have less computing power. YOLOv4-tiny utilizes a couple of different changes from the original YOLOv4 network to help it achieve these fast speeds. First and foremost, the number of convolutional layers in the CSP backbone are compressed with a total of 29 pretrained convolutional layers. Additionally, the number of YOLO layers has been reduced to two instead of three and there are fewer anchor boxes for prediction.

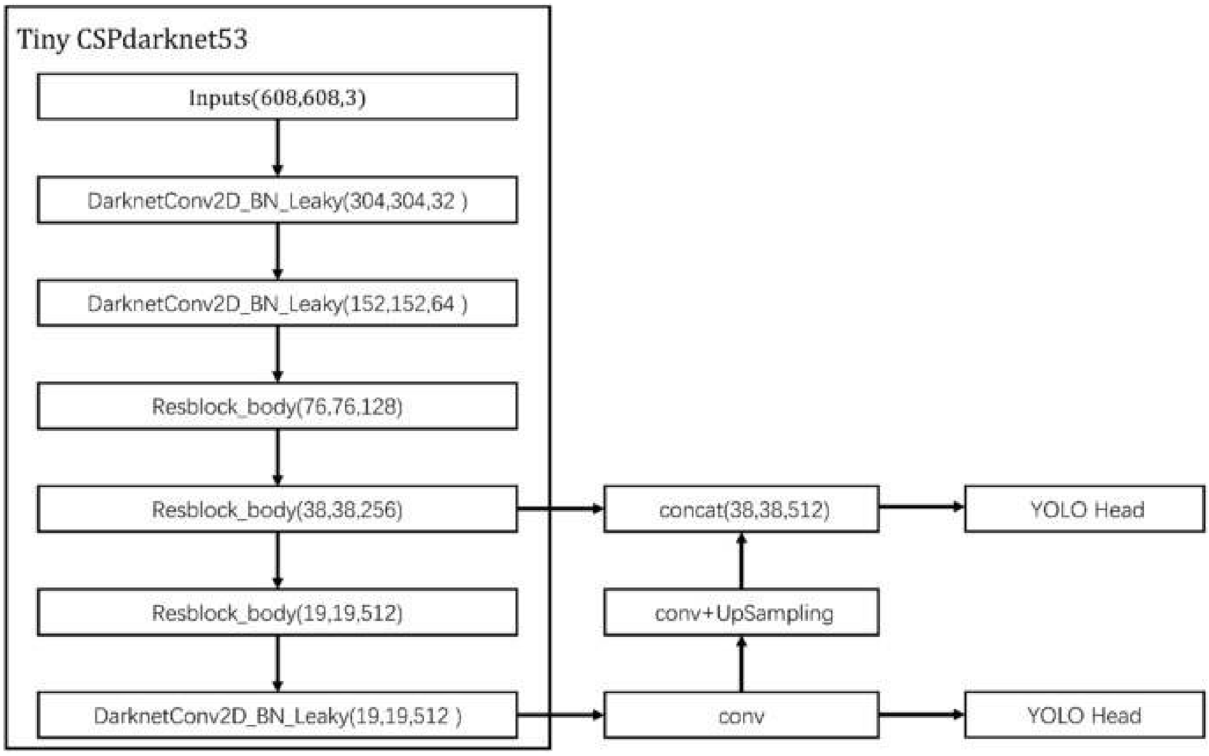


Figure 28: YOLOv4-tiny architecture

6. Case study

6.1 General Description – Goal

The main goal of this study was to develop a software tool for 3D object recognition, which can be custom trained to detect and identify any items the programmer chooses. In this case, it was trained to recognize mechanical components that can be found inside a vessel, such as pipes, flanges, valves and electric motors.

Additionally, the abovementioned software should be able to run on conventional devices, without the need for high-end computational components, that would make it inaccessible to common users. In order to achieve this, an object recognition model that requires low computing power was needed, without losing out on accuracy and speed. For this reason, the YOLOv4-tiny model was chosen as the ideal option.

Lastly, all of the software tools that are available on the internet until now, like Darknet and PyTorch, have a complex setup process, which requires the installation of various dependencies and applications that work in the background. This study aimed to create a software tool that can perform object recognition without the need for excessive preparation and setup and will be more user-friendly. Unity was deemed to be the best environment to create an application like that.

6.2 Used tools

6.2.1 Unity

Unity is a popular game engine that enables developers to create high-quality, interactive 2D and 3D games for various platforms such as desktop, mobile, and virtual reality (VR) devices. It was first released in 2005 by Unity Technologies and has since become one of the most widely used game engines in the world.

One of the primary advantages of Unity is its ease of use. The engine offers a visual editor that allows developers to create game scenes by dragging and dropping assets and scripts. Additionally, Unity supports a wide range of programming languages, including C#, which is commonly used by game developers. This flexibility makes Unity an accessible tool for developers of all skill levels.

Another key feature of Unity is its cross-platform capabilities. Games developed using Unity can be easily deployed to various platforms such as Windows, macOS, iOS, Android, and many more. This makes it possible for developers to reach a wider audience and maximize the potential of their games.

Unity also offers a variety of tools and features that can enhance the development process. For example, the engine supports real-time lighting and physics simulation, which can help create more immersive and realistic game experiences. Additionally, Unity's Asset Store provides a vast library of pre-made assets and tools that developers can use to speed up their workflow and improve the quality of their games.

6.2.1.1 Unity Barracuda

Unity Barracuda is an open-source deep learning framework developed by Unity Technologies. It is designed to enable developers to easily incorporate machine learning models into their Unity projects, allowing for advanced AI and data-driven game mechanics.

Barracuda supports a wide range of deep learning frameworks such as TensorFlow, ONNX, and PyTorch, and can be used with Unity's own machine learning tools such as ML-Agents. This flexibility makes it possible for developers to use their preferred machine learning framework while still taking advantage of Unity's game development capabilities.

One of the key benefits of Barracuda is its high performance. The framework is designed to run efficiently on a range of hardware, including CPUs, GPUs, and even mobile devices. This makes it possible to run complex machine learning models in real-time within a Unity game.

Overall, Unity Barracuda is a powerful tool for game developers looking to incorporate machine learning into their projects. Its high performance and flexible framework make it possible to create advanced AI and data-driven game mechanics with ease.

6.2.2 TensorFlow

TensorFlow is an open-source machine learning framework developed by Google Brain team in 2015. It is designed to build and deploy machine learning models at scale, allowing developers to create complex neural networks and other machine learning algorithms with ease.

The core component of TensorFlow is its computational graph, which represents a series of mathematical operations that are applied to input data to produce an output. This graph is highly flexible and can be optimized for a wide range of hardware configurations, including CPUs, GPUs, and TPUs.

One of the key advantages of TensorFlow is its ability to handle large-scale datasets efficiently. This is achieved through its use of data flow graphs, which allow developers to parallelize computations across multiple machines and GPUs. Additionally, TensorFlow supports distributed computing, which enables developers to train and deploy models across multiple nodes in a cluster.

TensorFlow has a wide range of pre-built libraries and tools, including Keras, a high-level API for building and training deep learning models, and TensorBoard, a visualization toolkit that helps developers to monitor and optimize their models. Additionally, TensorFlow supports a wide range of programming languages, including Python, C++, and Java.

6.2.2.1 Keras

Keras is an open-source neural network library written in Python. It is designed to enable the creation of deep learning models with minimal code. Keras was developed with a focus on enabling fast experimentation, and it has become a popular choice for building neural networks.

One of the main advantages of Keras is its high-level interface, which abstracts away many of the low-level details of building a neural network. Keras provides a simple and intuitive API for building and training models, which allows users to quickly prototype and experiment with different architectures.

6.2.3 ONNX

ONNX (Open Neural Network Exchange) is an open-source project developed by Microsoft and Facebook that allows deep learning models to be transferred between different frameworks, making it easier to deploy and optimize them on a variety of hardware platforms. ONNX defines a standard format for representing machine learning models, allowing them to be interoperable across different software frameworks.

The ONNX format allows users to export their trained models from their original framework to the ONNX format and then import them into another framework. This enables users to take advantage of the strengths of different frameworks and use them in combination to build complex models. The ONNX format is particularly useful for deploying machine learning models to a range of devices, from mobile phones to cloud servers, as it supports a wide range of hardware accelerators.

6.3 Steps in Case Study

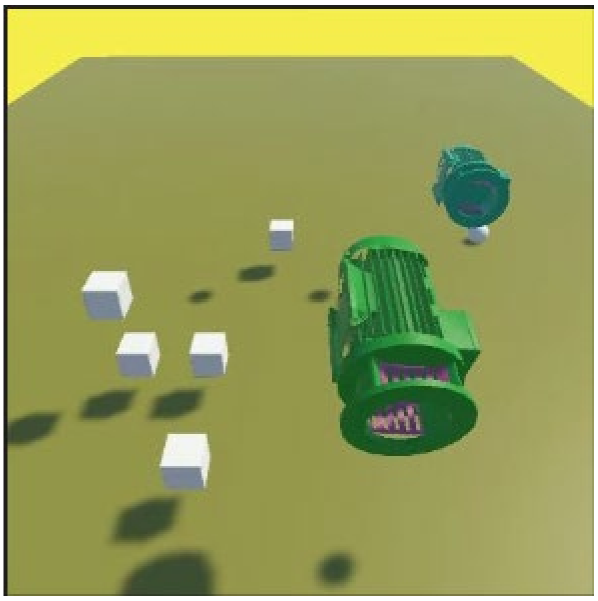
6.3.1 Dataset preparation

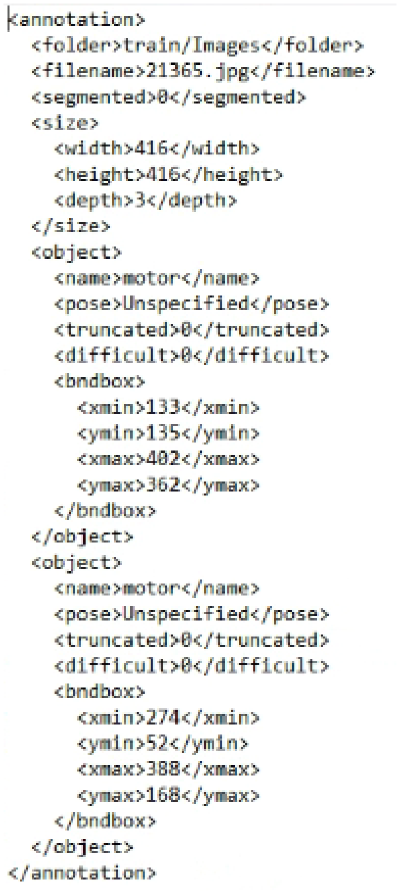
In order to train a CNN to recognize objects in images, it is necessary to ''feed'' the model with images that are already labelled and have bounding boxes drawn around each object. It is highlighted that for every object class (in this case, the classes will be 4: pipe, flange, valve, motor) about 2000 different images are required. This means that in order to train the model, it is required to have a minimum of 8000 images, with the accompanying annotation files that contain the information about the objects' classes and bounding boxes.

The annotation file that is needed for every image is an .xml file that has the following format:

```
<annotation>
 <folder>train/Images</folder>           //specifies the folder that contains the image
 <filename>0001.jpg</filename>           //the image's name (images should be in .jpg format)
 <segmented>0</segmented>                //specifies if the image is segmented or not (0 or 1)
 <size>
  <width>416</width>                     //image width
  <height>416</height>                   //image height
  <depth>3</depth>                       //image depth (3 is for RGB images)
 </size>
 <object>                                //this segment is repeated for every object in the image
  <name>motor</name>                     //object's class
  <pose>Unspecified</pose>               //specifies the skewness or orientation of the object
  <truncated>0</truncated>               //specifies if the object is fully or partially visible
  <difficult>0</difficult>               //specifies if the object is difficult to recognize
  <bndbox>                               //bounding box coordinates
   <xmin>12</xmin>                       //xmin and ymin are the coordinates of the top-left corner
   <ymin>94</ymin>
   <xmax>188</xmax>                      //xmax and ymax are the coordinates of the bottom-right
   <ymax>200</ymax>                      //corner
  </bndbox>
 </object>
 </annotation>
```

An example of an image and its annotation file is the following:





```xml
<annotation>
  <folder>train/Images</folder>
  <filename>21365.jpg</filename>
  <segmented>0</segmented>
  <size>
    <width>416</width>
    <height>416</height>
    <depth>3</depth>
  </size>
  <object>
    <name>motor</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>133</xmin>
      <ymin>135</ymin>
      <xmax>402</xmax>
      <ymax>362</ymax>
    </bndbox>
  </object>
  <object>
    <name>motor</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>274</xmin>
      <ymin>52</ymin>
      <xmax>388</xmax>
      <ymax>168</ymax>
    </bndbox>
  </object>
</annotation>
```

Figure 29: Image with two motors and the corresponding annotation file

Creating a dataset of at least 8000 sample images from data found on the internet is a task that requires hundreds of hours to complete, since it requires drawing the bounding boxes and writing the annotation files manually. Since there is no available dataset online that contains mechanical components, it was necessary to automate this process. In order to achieve the automation of the dataset preparation, a Unity project was used, where the images and annotation files were generated by the project.

Initially, the meshes of the required objects were created in Blender, a 3D modelling software, and then imported in the Unity project. Then, a scene was created in the Unity project with the use of the imported meshes on a custom design. When the project starts running, it takes screenshots of the scene and automatically creates the annotation files, while rotating the objects, changing their colour and spinning the camera angle, in order for the images to differentiate between them and give different perspectives of the scene. This process was repeated for every different object class for about 2000 images, and then it was used with a combination of classes, similar to a vessel's engine room.

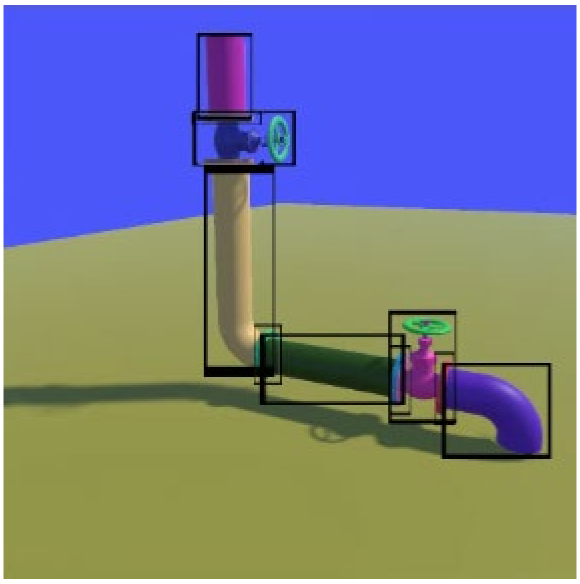


Figure 30: Screenshot of Unity scene with a custom design of pipe, flange and valve meshes

A total number of 25040 images and their corresponding annotation files were created in order to be used in the training process.

6.3.2 Training

The training was done with the use of the below Keras implementation of YOLOv4-tiny (https://github.com/bubbliiiing/yolov4-tiny-keras).

After installing the required dependencies and downloading the pre-trained YOLOv4-tiny weights file, the voc_annotation.py script was used in order to split the dataset into two categories. The ''train'' category, which contains 90% of the images, is used to train the model, while the ''val'' category, that contains the rest of the images, is used to evaluate the model and correct the training process.
Then, a .txt file named cls_classes was created, which contains the four classes' names, each on a separate line.
Lastly, the train.py script, which contains all of the parameters and hyperparameters, was updated with the new file paths and it was ran on the Command Prompt. At this point, it should be noted that the new Nvidia GPUs with Ampere technology do not support the TensorFlow version 1.13 that the above software requires, so training was conducted using an Nvidia GeForce 1050 Ti GPU. Training lasted about 3 days and the output was an .h5 file that contains the weights of the network.

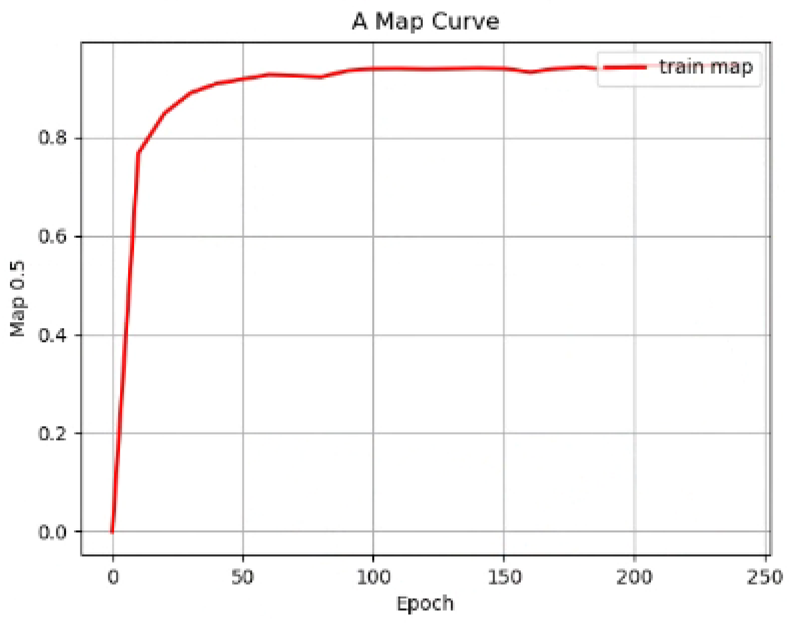


Figure 31: Graph showing mAP calculations throughout the 250 epochs of training

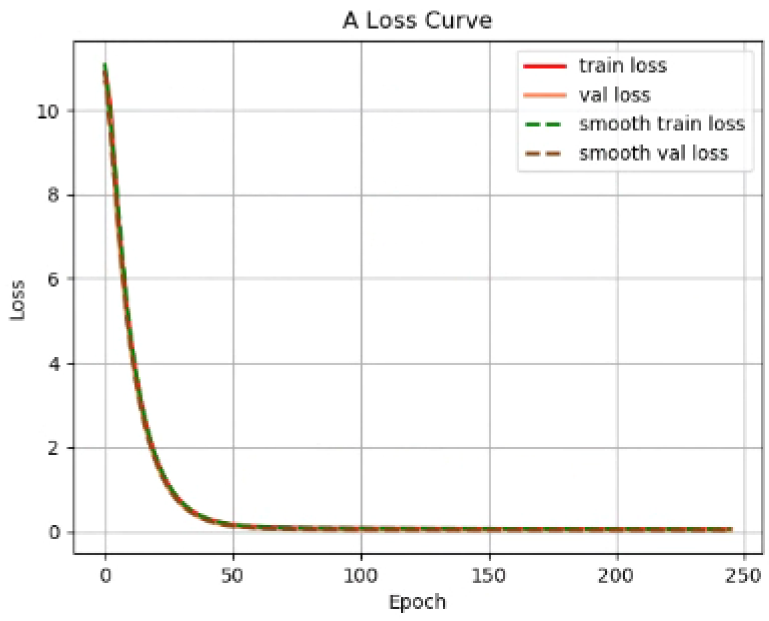


Figure 32: Graph showing train loss and val loss throughout the training process

In the end, the below lines were added in the yolo.py script in the generate method:

```
self.yolo_model.summary()
json_string = self.yolo_model.to_json()
open('yolov4_tiny_custom.json', 'w').write(json_string)
```

and the predict.py script was used, in order to generate a .json file that will be used in future steps.

6.3.3 Conversion to ONNX format

In order to be able to use the trained model in Unity, the weights needed to be converted to ONNX format. The conversion happened in three steps:

- Conversion from Keras format (.h5) to TensorFlow format (.pb)

Using the below script and the two files generated during the training process (the .h5 weights file and the .json file), a ''saved model'' folder was created containing the .pb weights file

```
import tensorflow as tf
model = tf.keras.models.model_from_json(open('yolov4_tiny_voc.json').read(),
custom_objects={'tf': tf})
model.load_weights('yolov4_tiny_voc.h5')
model.save('saved_model')
```

- Conversion from TensorFlow format (.pb) to ONNX format (.on)

https://github.com/onnx/tensorflow-onnx
Using the above GitHub repository, after installing the required dependencies, the below command converts the saved_model folder to an .onnx weights file:

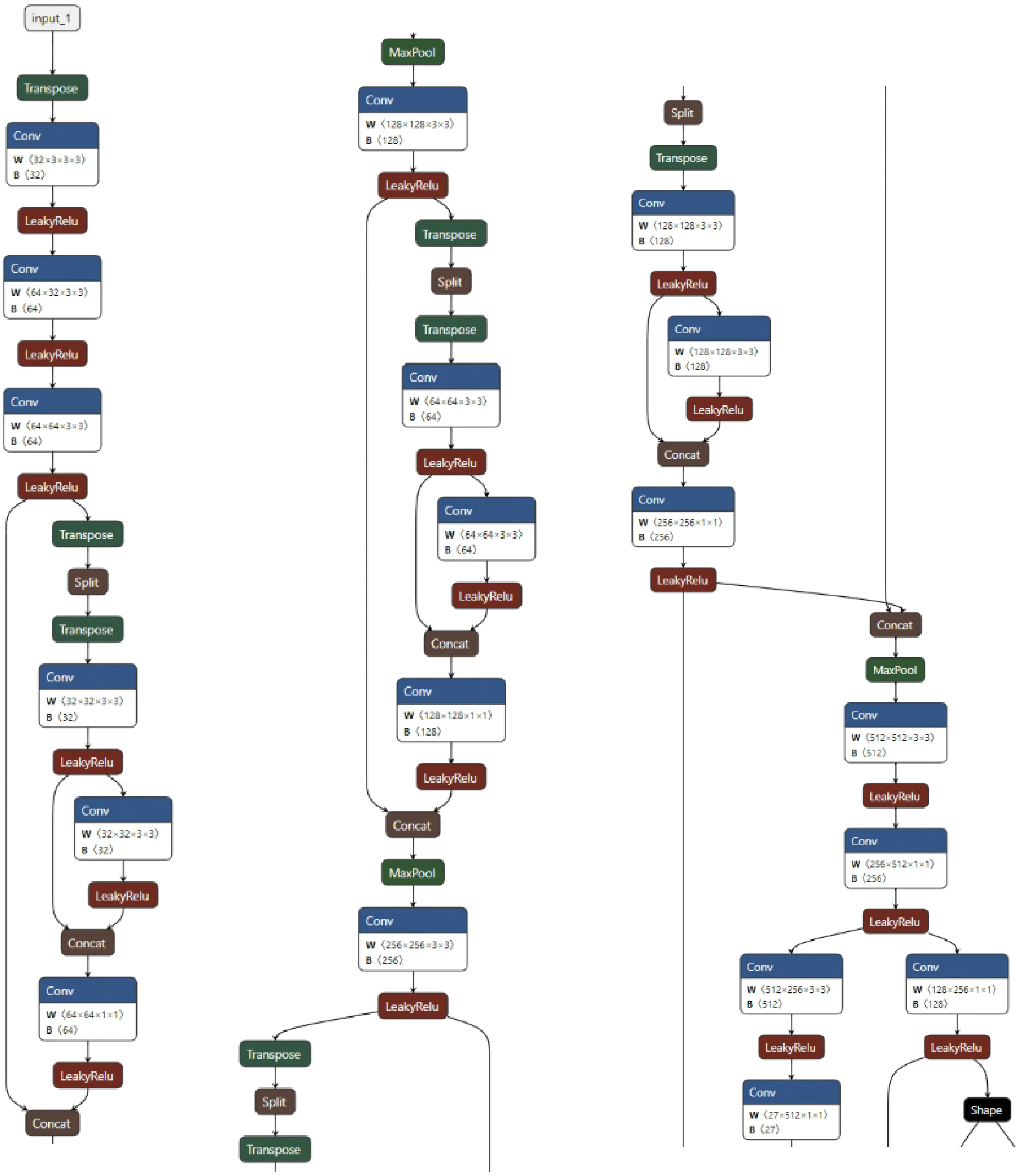
```
python -m tf2onnx.convert --saved-model tensorflow-model-path --opset 9 --output model.onnx
```

- Conversion to a Unity Barracuda compatible format

Unity Barracuda version 1.0.4 does not support the 'Split' node, so in order to solve this issue, the following script was used, which replaces the Split nodes of the network:

```
import numpy as np
import onnx
from onnx import checker, helper
from onnx import AttributeProto, TensorProto, GraphProto
from onnx import numpy_helper as np_helper
def scan_split_ops(model):
 for i in range(len(model.graph.node)):
   # Node type check
   node = model.graph.node[i]
   if node.op_type != 'Split': continue
   # Output tensor shape
   output = next(v for v in model.graph.value_info if v.name == node.output[0])
   shape = tuple(map(lambda x: x.dim_value, output.type.tensor_type.shape.dim))
   shape = (shape[3], shape[3])
   # "split" attribute addition
   new_node = helper.make_node('Split', node.input, node.output, split = shape, axis = 3)
   # Node replacement
   model.graph.node.insert(i, new_node)
   model.graph.node.remove(node)
model = onnx.load('yolov4_tiny.onnx')
model = onnx.shape_inference.infer_shapes(model)
scan_split_ops(model)
checker.check_model(model)
onnx.save(model, 'yolov4_tiny_barracuda.onnx')
```

The final result is an ONNX file that contains the weights and the structure of the network, which can be used in Unity and perform the desired 3D object recognition.



input_1
Transpose
Conv W ⟨32×3×3×3⟩ B ⟨32⟩
LeakyRelu
Conv W ⟨64×32×3×3⟩ B ⟨64⟩
LeakyRelu
Conv W ⟨64×64×3×3⟩ B ⟨64⟩
LeakyRelu
Transpose
Split
Transpose
Conv W ⟨32×32×3×3⟩ B ⟨32⟩
LeakyRelu
Conv W ⟨32×32×3×3⟩ B ⟨32⟩
LeakyRelu
Concat
Conv W ⟨64×64×1×1⟩ B ⟨64⟩
LeakyRelu
Concat
MaxPool
Conv W ⟨128×128×3×3⟩ B ⟨128⟩
LeakyRelu
Transpose
Split
Transpose
Conv W ⟨64×64×3×3⟩ B ⟨64⟩
LeakyRelu
Conv W ⟨64×64×3×3⟩ B ⟨64⟩
LeakyRelu
Concat
Conv W ⟨128×128×1×1⟩ B ⟨128⟩
LeakyRelu
Concat
MaxPool
Conv W ⟨256×256×3×3⟩ B ⟨256⟩
LeakyRelu
Transpose
Split
Transpose
Split
Transpose
Conv W ⟨128×128×3×3⟩ B ⟨128⟩
LeakyRelu
Conv W ⟨128×128×3×3⟩ B ⟨128⟩
LeakyRelu
Concat
Conv W ⟨256×256×1×1⟩ B ⟨256⟩
LeakyRelu
Concat
MaxPool
Conv W ⟨512×512×3×3⟩ B ⟨512⟩
LeakyRelu
Conv W ⟨256×512×1×1⟩ B ⟨256⟩
LeakyRelu
Conv W ⟨512×256×3×3⟩ B ⟨512⟩
Conv W ⟨128×256×1×1⟩ B ⟨128⟩
LeakyRelu
LeakyRelu
Conv W ⟨27×512×1×1⟩ B ⟨27⟩
Shape

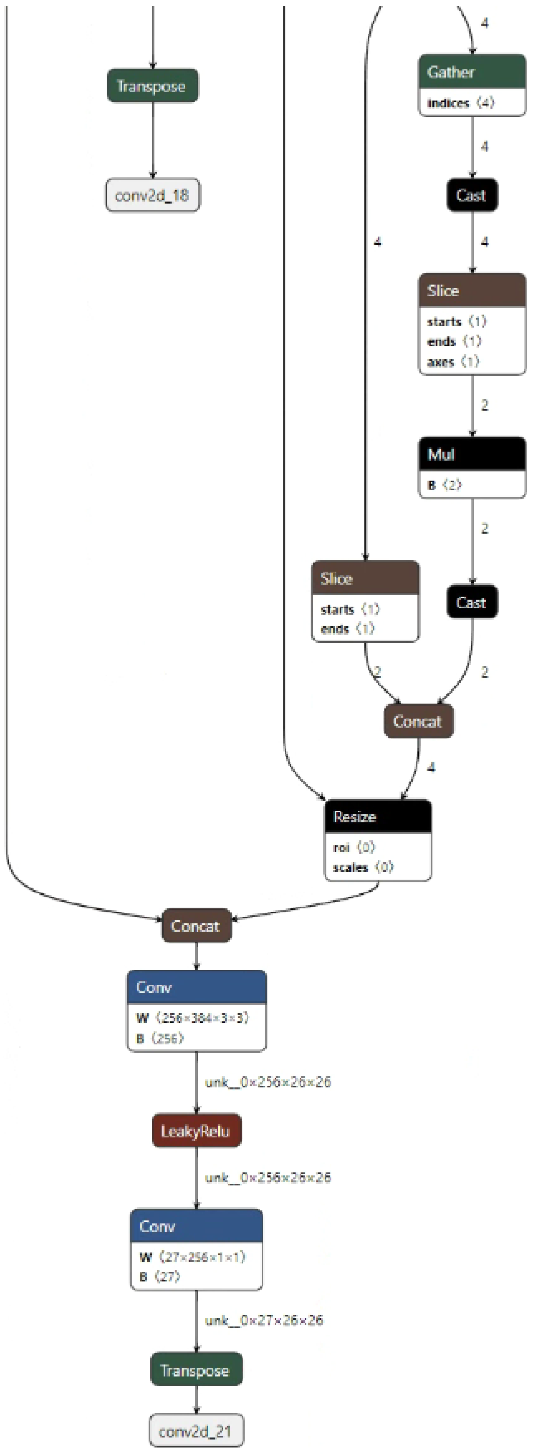


Figure 33: Architecture of the .onnx file

6.3.4 Implementation in Unity

For the final step of this study, a project by Keijiro Takahashi, a Unity developer was used (https://github.com/keijiro/YoloV4TinyBarracuda). After importing the final ONNX weights file into the project and assigning it to the Resource Set, a few changes were also made. First of all, in the "Marker" script, the "_labels" string was changed to contain the model's 4 classes. Also, in the "ObjectDetector" script, the names of the output layers in lines 131 and 132 were changed as shown below, in order to correspond with our model's output layers:

"Identity" → "conv2d_18"
"Identity_1" → "conv2d_21"

At this point, it is possible to choose in the “Image Source” script of the “Visualizer” game object the desired source type, between image, video or a webcam, and the project is ready to perform the desired object recognition task.

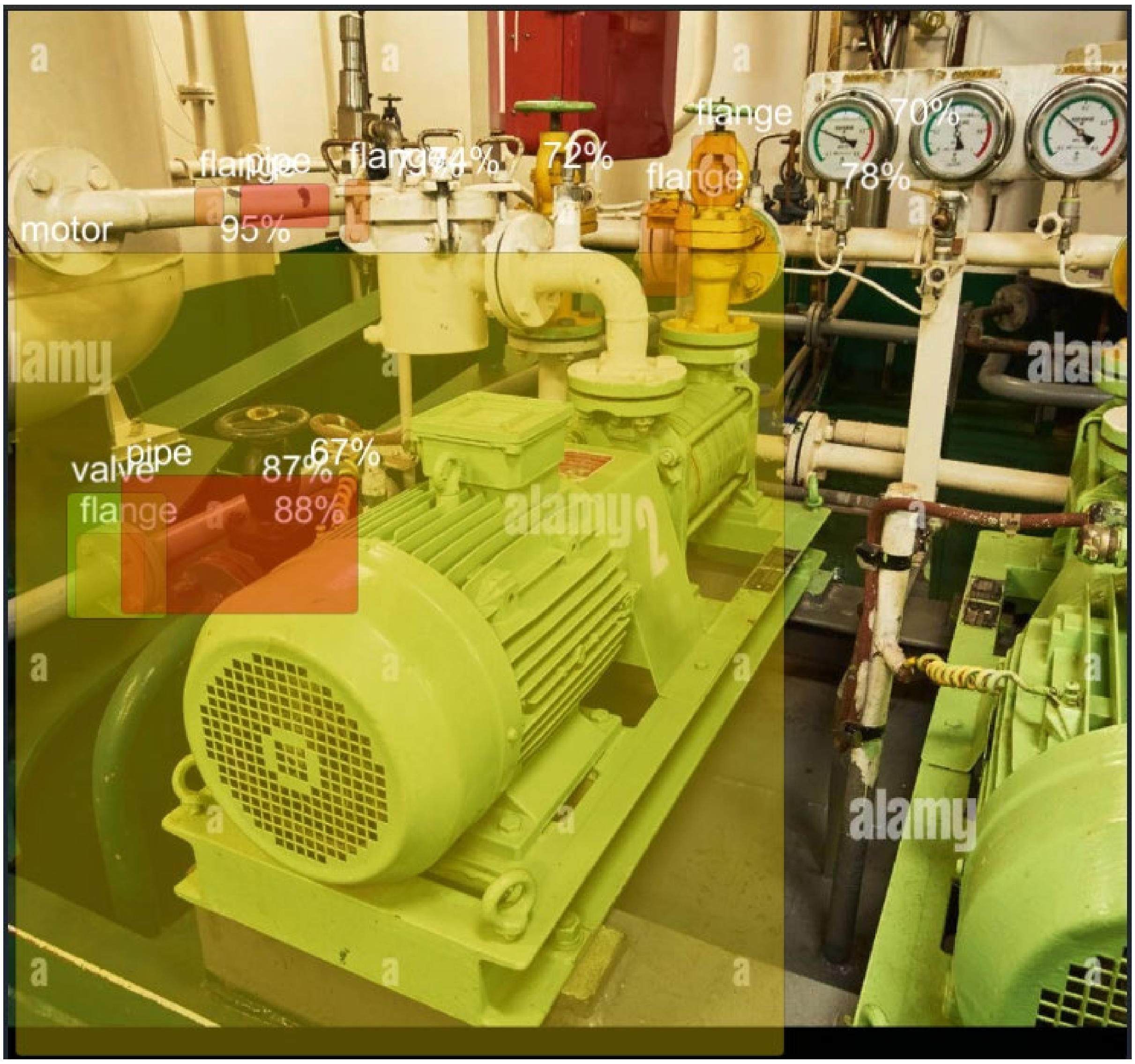


Figure 34: Output of the model with a threshold of 0.65

In the above picture, the model was able to detect most of the machinery parts with an adequate accuracy. The threshold of detection was set to 0.65, meaning that the model will show only predictions with confidence score above said threshold.

7. Conclusion and Future Work

7.1 Conclusion

The present thesis describes and analyses the process of creating a software tool capable of performing 3D object recognition, without the need of explicit programming and computer science knowledge. In this case, the desired objects were mechanical components that can be found inside a vessel, but it can be used to detect any object class, provided that a dataset with enough sample images is provided or generated by the user. The training part of the CNN is a more complex process, which requires several parameters to be monitored and a dataset that is vast and correctly labelled, but as soon as the model is trained efficiently, the detection part is a simple and easy task. The end-product is both effective and user-friendly, and can be used by anyone desiring to automate an object recognition process, such as inventory management.

Additionally, in this time of rapid advancement of AI technology, it is important that anyone who wishes to invest time and effort on AI applications can learn their basic concepts and principles, in order to be able to utilize them in efficient ways and help each industry evolve. This thesis provides a comprehensive overview of the state-of-the-art techniques and methods used for 3D object recognition, as well as the strengths and limitations of each approach.

7.2 Future work

First of all, the end-product of the Unity project can be further optimized with a user-friendly UI and built into an application that can run on any device without the need for the Unity platform. This way, the 3D object recognition task can be performed everywhere and by anyone, even inside a vessel's engine room in real-time using a smartphone's camera.

One other improvement can be the complete automation of the training process. The Unity project that was used for the dataset preparation is a huge improvement compared to the manual processing of each image, but considering the rapid evolve of AI these days, it should be possible to assign the task of image labelling to a machine. This way, by using images from an existing engine room's environment, the training process will be much more efficient and the model will perform object recognition with almost 100% accuracy.

Lastly, this thesis can serve as a foundation for the application of more state-of-the-art object recognition models, such as YOLOv7, which is an improvement to the previous YOLO models, but it is not yet able to run on the Unity Barracuda environment.

8. References

1. Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi (2016)
   You Only Look Once: Unified, Real-Time Object Detection.
   https://arxiv.org/abs/1506.02640

2. Joseph Redmon, Ali Farhadi (2016)
   YOLO9000: Better, Faster, Stronger
   https://arxiv.org/abs/1612.08242

3. Joseph Redmon, Ali Farhadi (2018)
   YOLOv3: An Incremental Improvement
   https://arxiv.org/abs/1804.02767

4. Alexey Bochkovskiy, Chien-Yao Wang, Hong-Yuan Mark Liao (2020)
   YOLOv4: Optimal Speed and Accuracy of Object Detection
   https://arxiv.org/abs/2004.10934

5. Sergey Ioffe, Christian Szegedy (2015)
   Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift
   https://arxiv.org/abs/1502.03167

6. Vinod Nair, Geoffrey E. Hinton (2010)
   Rectified Linear Units Improve Restricted Boltzmann Machines
   https://www.cs.toronto.edu/~fritz/absps/reluICML.pdf

7. Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov (2014)
   Dropout: A Simple Way to Prevent Neural Networks from Overfitting
   https://jmlr.org/papers/v15/srivastava14a.html

8. Pengguang Chen, Shu Liu, Hengshuang Zhao, Jiaya Jia (2020)
   GridMask Data Augmentation
   https://arxiv.org/abs/2001.04086

9. Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, Rob Fergus (2013)
   Regularization of Neural Networks using DropConnect
   https://proceedings.mlr.press/v28/wan13.html

10. Ilya Loshchilov, Frank Hutter (2017)
    SGDR: Stochastic Gradient Descent with Warm Restarts
    https://arxiv.org/abs/1608.03983

11. Diederik P. Kingma, Jimmy Ba (2017)
    Adam: A Method for Stochastic Optimization
    https://arxiv.org/abs/1412.6980

12. Sangdoo Yun, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, Youngjoon Yoo (2019)
CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features
https://openaccess.thecvf.com/content_ICCV_2019/papers/Yun_CutMix_Regularization_Strategy_to_Train_Strong_Classifiers_With_Localizable_Features_ICCV_2019_paper.pdf

13. Chien-Yao Wang, Hong-Yuan Mark Liao, I-Hau Yeh, Yueh-Hua Wu, Ping-Yang Chen, Jun-Wei Hsieh (2019)
CSPNet: A New Backbone that can Enhance Learning Capability of CNN
https://arxiv.org/abs/1911.11929

14. Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun (2015)
Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition
https://arxiv.org/abs/1406.4729

15. Zhuliang Yao, Yue Cao, Shuxin Zheng, Gao Huang, Stephen Lin (2021)
Cross-Iteration Batch Normalization
https://arxiv.org/abs/2002.05712

16. Shu Liu, Lu Qi, Haifang Qin, Jianping Shi, Jiaya Jia (2018)
Path Aggregation Network for Instance Segmentation
https://openaccess.thecvf.com/content_cvpr_2018/papers/Liu_Path_Aggregation_Network_CVPR_2018_paper.pdf