Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Control Engineering and Information Technology

Efficient Neural Network Pruning Using Model-Based Reinforcement Learning

Master's Thesis

Author
Blanka Bencsik

Advisor
Dr. Márton Szemenyei

June 11, 2023

Contents

Kivonat i

Abstract ii

1 Introduction 1

2 Theoretical Background 3

2.1 Convolutional Neural Networks . . . 3

2.1.1 The Structure of Convolutional Neural Networks . . . 3

2.1.2 Training Neural Networks . . . 4

2.2 YOLOv4 Object Detector . . . 5

2.3 Neural Network Pruning . . . 7

2.4 Reinforcement Learning (RL) . . . 8

2.4.1 Reinforcement Learning Task . . . 8

2.4.2 Multi-Agent Actor-Critic Methods . . . 10

2.5 Transformer Neural Networks . . . 12

2.5.1 Processing Sequential Data . . . 12

2.5.2 The Transformer Architecture . . . 13

3 Preliminaries 15

3.1 Conventional Pruning Methods . . . 15

3.2 Reinforcement Learning-Based Pruning . . . 15

3.2.1 AutoML for Model Compression (AMC) . . . 16

3.2.2 Pruning Using Reinforcement Learning (PuRL) . . . 16

4 Problem Statement 18

5 Proposed Methods 20

5.1 Metrics . . . 20

5.2 Development Environment . . . 21

5.3 Training The YOLOv4 Object Detector . . . 21

5.4 Modifying the YOLOv4 Architecture During Pruning 22
5.4.1 Prunable Layers In The YOLOv4 Architecture 22
5.5 The State Predictor Network (SPN) 24
5.5.1 The SPN's Role In The System 24
5.5.2 Automatic Data Generation 25
5.6 Automatic Pruning Using Reinforcement Learning 26
5.6.1 State Space 26
5.6.2 Action Space 27
5.6.3 Reward Function 27
5.6.4 Training The RL Agent 28
5.6.5 Model-Based Learning 29
5.7 Generalization Of The Automatic Pruning System 30
5.7.1 Generalization Across Datasets 30
5.7.2 Generalization Across DNN Architectures 30
5.7.3 Torch Pruning Library 31
5.7.4 The Tranformer SPN Architecture 32

6 Results **34**
6.1 Towards The Optimal Solution 34
6.1.1 Model State Size 34
6.1.2 Overconfident Policy 36
6.1.3 Careful Hyperparameter Selection 37
6.2 The Pruned YOLOv4 Model 38
6.2.1 The Model's Performance 38
6.2.2 The Proposed Pruning System's Speed 40
6.3 Generalization Ability 42
6.3.1 Fine-Tuning The SPN On COCO 42
6.3.2 Training The Transformer SPN 45
6.3.3 Training The RL Agent With The Transformer SPN 46
6.4 Discussion 48

7 Conclusion **50**

Bibliography **52**

HALLGATÓI NYILATKOZAT

Alulírott *Bencsik Blanka*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2023. június 11.

Bencsik Blanka
hallgató

Kivonat

A mély neurális hálók az elmúlt tíz évben kimagasló eredményt mutattak az objektumdetektálás területén, alkalmazásuknak azonban gátat szab a nagy számításikapacitás-igényük és méretük, mely a hatalmas paraméterszámukkal magyarázható. Különösen jelentős ez a probléma az erőforráskorlátos eszközök esetén, ahol még a betanított modellek tárolása és futtatása is nehézséget okozhat. A probléma megoldására használatos a neurális háló ritkítás, melynek célja egy neurális háló modell paramétereinek lehető legnagyobb mértékű redukálása a pontosság romlásának elkerülése mellett.

Az eltávolítandó paraméterek kiválasztását jelentősen megnehezíti az, hogy a modell különböző rétegei nem egyformán érzékenyek a paraméterek törlésére. További nehezítő tényező, hogy a redukált modell pontosságromlását nem csak az éppen ritkítandó réteg érzékenysége befolyásolja, hanem az azt megelőző rétegek redukálásának mértéke is. A mély neurális háló modellekben ezen összefüggések feltérképezése a lehetséges variációk hatalmas száma miatt kézzel lehetetlen, csupán feltételezésekre támaszkodhatunk.

A probléma megközelíthető megerősítéses tanulás alkalmazásával, melynek során az ágens emberi beavatkozás nélkül igyekszik megtalálni az optimális eltávolítandó paraméterszámot a modell minden rétegéhez. A probléma ezen megközelítése jelenleg is nyitott téma az irodalomban, ám a létező megoldások legnagyobb hátulütője, hogy az ágens számára a két legfontosabb környezeti változót - a ritkaságot és a pontosságromlást - a modell tényleges ritkításával és validálásával határozzák meg futási időben, ami nagymértékben lelassítja a tanítási folyamatot.

Munkám során egy olyan megerősítéses tanulás alapú rendszert valósítok meg, mely a YOLOv4 objektumdetektor optimális ritkítását teszi lehetővé a tanítási folyamat idejének csökkentése mellett. Az eddig létező megoldásokhoz képest a rendszert egy olyan neurális hálóval egészítem ki, mely a ritkítandó háló pontosságváltozását és ritkaságát képes megbecsülni az addig törölt paraméterek mennyiségének és az adott réteghez választott redukáló tényező függvényében. Ezen állapotbecslő háló a környezet állapotának meghatározásához szükséges hosszadalmas műveletek szerepét veszi át, nagyságrendekkel növelve így a tanítás sebességét. Az állapotbecslő hálót önfelügyelt tanítás segítségével lett betanítva, automatikusan generált adatokon.

Mindemellett, a dolgozatban a ritkító rendszer általánosítóképességének kivizsgálására is sor kerül, továbbá, bemutatásra kerülnek az első, kulcsfontosságú fejlesztési lépések az adatbázisok és neurális háló architektúrák közti általáosíthatóság céljából. Ez többek közt magában foglalja a transzformer állapotbecslő háló tervezését és tanítását, mely lehetővé teszi a dinamikus bemenetek kezelését. A megvalósított módszer eredményei a teljes fejlesztési idő és a pontosságromlás és ritkaság tekintetében state-of the-art megerősítéses tanulás-alapú ritkító rendszerekkel, valamint kézzel szerkesztett ritkító szabályokkal kerülnek összevetésre.

Abstract

Deep Neural Networks (DNN) have achieved outstanding results in the field of object detection in the past decade. Unfortunately, this success usually comes at the cost of tremendous computational and memory capacity, due to the vast amount of parameters the DNNs comprise. These requirements make their deployment cumbersome, especially in resource-constrained devices, such as mobile phones or embedded systems. One popular approach to tackle this issue is network pruning, which is accomplished by systematically removing parameters from an existing, accurate DNN. By doing so, a smaller network is produced while maintaining most of the initial accuracy.

The process of choosing the parameters to be pruned is quite demanding, since the different layers in the DNN are not equally sensitive for removing parameters from them. Moreover, the deterioration of the accuracy is not only determined by the sensitivity of a current layer, but also by the amount of removed parameters from all the previous layers. The number of possible variations of these dependencies are so large, they cannot be tried out manually.

An innovative idea to mitigate the aforementioned challenges is to employ reinforcement learning (RL) and let the RL agent seek to find the optimal subset of parameters to be removed from a DNN without human interaction. This topic is currently an open issue in the literature, however, the existing solutions have one common drawback: they determine the main environmental state variables – the deterioration of the accuracy and the sparsity – by pruning and testing the model in run time, which slows down the training procedure extremely.

During my research, I orchestrate a RL-based automatic pruning system which is able to sparsify the YOLOv4 object detector optimally, by removing entire channels from its architecture to boost the utilization of hardware resources. Compared to existing solutions, the proposed system contains an additional State Prdeictor Network which can predict the main environmental variables used by the RL agent if the action and the model state are given as inputs. It replaces the role of long procedures that were performed to determine environment state, making the RL agent's training significantly faster. The State Prdeictor Network is trained via self-supervised learning on automatically generated data.

Furthermore, I have assessed the proposed system's generalization ability across datasets and have initiated the preliminary steps towards achieving generalization across different DNN architectures. These involve designing and training a transformer-based State Prdeictor Network, enabling it to effectively handle dynamic input sizes. The presented solution is evaluated by comparing its results to state-of-the-art RL-based pruning methods and self-designed handcrafted pruning rules, considering factors such as total development time, accuracy degradation and sparsity ratio.

Chapter 1

Introduction

DNNs have been firmly established as state-of-the-art (SOTA) approach to solve various tasks in the field of informatics in the past decade. However, their deployment in resource-constrained devices, such as mobile phones or embedded systems, remains challenging even nowadays, due to the their huge computational and memory costs. To address this challenge, the Neural Network Pruning process is employed, which aims to reduce the size of a trained, accurate DNN model while minimizing any potential accuracy degradation.

Evidently, in order to achieve minor accuracy degradation, one has to carefully design the strategy used for removing wights from the model, considering their amount and location. Over the past years, various studies have emerged, introducing different rule-based systems for efficiently pruning classifiers and detectors. Although some rule-based solutions are so well-constructed, that are able to remove over 90 % of the designated model's weights while maintaining high accuracy , these methodologies inherently pose a disadvantage as they require extensive human intervention. Consequently, they suffer from limitations in exploring the action space due to the vast number of possible parameter variations.

Research studies presenting the idea of employing RL for automating neural network pruning have emerged in the past 3-4 years. This field of research is still in its early phase, which accounts for the lack of standardized evaluation criteria to measure the efficiency of these methods. The task of automated pruning introduces new challenges to researchers in addition to those associated with traditional pruning methods. For instance, some include choosing the most suitable RL algorithm (agent), determining the frequency and calculation of rewards or the construction of the action and state space. Some papers already offer notable solutions for the aforementioned problems. Nevertheless, all the already existing solutions evaluate the pruned model on a smaller test set in order to obtain the sparsity ratio and the degradation of the model's accuracy. These metrics are the environmental variables used by the agent during its training, and performing time-consuming procedures to obtain them leads to extremely prolonged training time.

During my work, I addressed this particular problem to design and implement a novel automated RL-based pruning system suitable for the YOLOv4 object detector. The proposed system performs similarly efficient pruning as other existing methods, while significantly reduces the agent's training time and the overall development time. To accomplish this, I replaced the method used for determining the main environmental variables. Instead of loading and evaluating the pruned model at each iteration, I incorporated a so-called State Predictor Network (SPN) simulate the environment. The SPN was trained on automatically generated data and is able to predict the expected sparsity and accuracy degradation if the model's state and the chosen action are given as as inputs. As only the model's

architecture is needed to be fed to the agent, due to the reduced GPU memory requirement, the use of SPN allows for multi-agent training, which leads to faster and more stable training. To the best of my knowledge, this approach for pruning the YOLOv4 detector is unique, as no such solution has been introduced in the existing literature. The remainder of the system is constructed by incorporating ideas inspired by methods outlined in the literature.

My research also involves the examination of the proposed pruning system's generalization abilities across different datasets. In pursuit of the ultimate goal of generalizing the solution across different DNN architectures, I have designed and trained a transformer-based SPN that is capable of handling dynamic input sizes. This adaptability is crucial considering the varying number of layers present in different architectures, which directly influences the input of the SPN. While the successful integration of the new SPN architecture into the pruning pipeline awaits further development, I have evaluated its performance by comparing it to the initial SPN model.

My main contributions in the thesis are outlined as follows:

1. Training the YOLOv4 object detector on the KITTI dataset and implementing source code that allows the backpropagation on the modified architecture after pruning.
2. Planning the automatic data generation for the SPN using the previously trained YOLOv4 model. Designing the SPN, conducting experiments for finding optimal hyperparameters, and training it on the automatically generated data.
3. Choosing the RL agent, designing its state and action space and defining the reward function based on ideas inspired by existing publications. Integrating the SPN into the agent's training and experimenting with various hyperparameters.
4. Removing closely 50 % of the parameters from the initial YOLOv4 model with structured channel pruning, while preserving the accuracy with little to no degradation, using the proposed pruning system.
5. Evaluation of the achieved results by comparing them to SOTA methods considering the reduced model's performance and the efficiency of the automatic pruning system in terms of speed.
6. Investigation of the system's generalization ability across different datasets and deploying transfer-learning for its improvement.
7. Making initial efforts to achieving generalization of the system across different architectures by designing and training a transformer-based SPN that allows dynamic input sizes. In addition, its potential for the task was evaluated by comparing its performance to the original SPN model.

The thesis begins with providing a theoretical background on CNNs, object detectors, neural network pruning, RL and transformer-based NNs in Chapter 2. Chapter 3 offers a brief overview of the latest advancements and results in both traditional and automatic pruning techniques. This is followed by Chapter 4, which defines the problem statement and briefly introduces the proposed solution. In Chapter 5 I elaborate on the methods and ideas utilized during the development. Chapter 6 presents the achieved results, alongside with their comparison to SOTA methods, highlighting the challenges encountered during development and proposing solutions to address them. Finally, in Chapter 7, potential future improvements for the proposed automatic pruning system are discussed.

Chapter 2

Theoretical Background

2.1 Convolutional Neural Networks

2.1.1 The Structure of Convolutional Neural Networks

Artificial neural networks (ANNs) are algorithms whose working principle is inspired by the functioning of the human nervous system and are capable of recognizing various features of the input data. This form of artificial intelligence serves as the basis of many machine learning tasks. Neural networks' topologies can be diverse depending on the type of layers they are constructed from. The two most frequently used layer types are the fully connected (linear) and the convolutional layers. Fully connected layers connect all input neurons to all output neurons, while convolutional layers connect their output neurons to only a few input neurons that are located in their local neighborhood [1, 2].

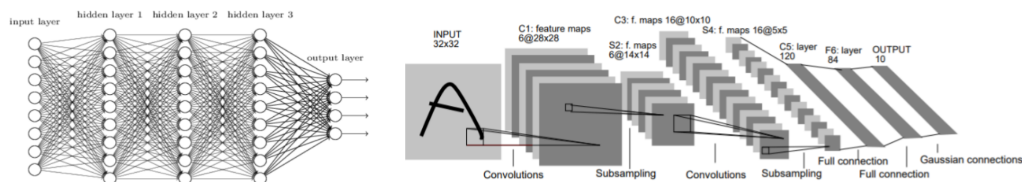

Figure 2.1: Fully connected [2] an convolutional neural network architectures [3].

A neural network (NN) that is constructed from convolutional layers is called convolutional neural network (CNN). It is most commonly used for solving image processing tasks due to its advantageous properties. In contrast to fully connected NN, it can effectively utilize the spatial structure of images and it comprises significantly less parameters because of the fewer connections between its layers [4]. The image is represented as a three-dimensional matrix to the input of the network, where the pixels correspond to the neurons. The filters in the first convolutional layer produce the output activation maps which serve as the input to the following convolutional layer and so on.

A CNN might also comprise various up- and down-scaling (so-called pooling) layers. The order in which the individual convolutional, linear and pooling layers follow each other in CNNs, the depth of the network and the size of the individual layers are altogether referred to as the network architecture [2]. Generally speaking, the first layers of the CNNs detect basic features in the images, like edges, corners, etc., while the following layers are able to perceive more complex feature properties as well [2].

2.1.2 Training Neural Networks

When training a NN model, the goal is to adjust the weights of the neurons in such a way that the output predicted by the model approximate the target output - also know as ground truth - as precisely as possible. In supervised learning the model's accuracy can be determined by defining a loss function (also known as cost function) and calculating its value for each input based on the two aforementioned outputs. Some widely used loss functions for this task are the Mean Square Error (MSE), the Hinge loss function and the Cross Entropy Loss. Often, a regularization parameter (R) is employed to add a penalty term to these loss functions, avoiding the so-called overfitting phenomenon: developing a model which is overly confident on the training data set, yet performs poorly on the validation data set. The two simplest regularization terms are the absolute and squared error of the weight matrix (L1 and L2 regularization). The resulting loss function:

$$L = \sum_{i}^{N} L_i + \lambda R(W) \tag{2.1}$$

Aiming to find the appropriate weights, the loss function needs to be minimized. This can be achieved using the Gradient Descent Algorithm, which leverages the fact that the derivative of the loss function can be calculated with respect to the weights. This gives the direction of the steepest descent along the error surface, thus, in each iteration, the algorithm alters the weight matrix in a way that it takes steps towards this direction until it reaches the global minimum [5]. The modification of the loss function can be described as:

$$W_{k+1} = W_k - \alpha \frac{\partial \|L\|^2}{\partial W} \tag{2.2}$$

where W denotes the weight matrix, L is the loss function and α is the learning rate (lr) which represents the step size by which we approach the global minimum [4]. It is common to supplement the Gradient Descent Algorithm with some sort of momentum which helps to avoid getting stuck in local minima. In this case, to determine the gradient, not only the local data, but the gradient calculated in the previous time step is also used with some weight [6]. Some widely used gradient-based optimization algorithms include Stochastic Gradient Descent (SGD), Adam, AdaGrad, and RMSProp.

The Backpropagation Algorithm is the most commonly used technique for training NNs utilizing the derivative of the loss function with respect to the weights. Its main idea is that a multilayer NN can be represented by a directed graph whose nodes are the neurons (weights) [5]. Through the chain rule, the algorithm computes the derivative of a node with respect to the input and an arbitrary weight by progressing from back to front. This is possible because the derivative of the last node is known: the output of the network is the loss function, whose derivative with respect to itself is always 1

By utilizing the knowledge gained so far, the working principle of CNNs can now be summarized. These types of NNs transform input data using their weight matrices and generate an output that highlights some typical characteristics of the data. However, this complex task cannot be handled using linear transformations alone. Convolution is a linear operation, therefore it is not sufficient for this task. To overcome this issue, usually

non-linear functions are incorporated into the architecture between certain layers. These are called activation functions, and some commonly used ones include the Rectified Linear Unit (ReLU), Leaky ReLU (Figure 2.2) or the Hyperbolic Tangent (Tanh).

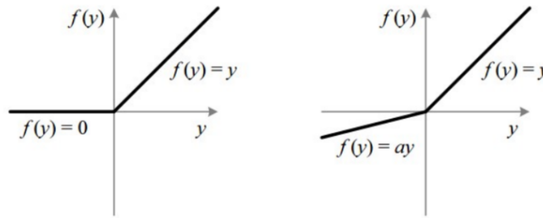

Figure 2.2: ReLU and Leaky ReLU activation functions[1].

2.2 YOLOv4 Object Detector

NN-based object detectors can be categorized into two main groups: region-based methods and single-shot detectors. Detectors belonging to the former group employ conventional segmentation methods to generate region proposals, which is followed by performing object detection on each proposal using CNNs [7, 8]. Meanwhile, single-shot detectors simultaneously perform object detection and classification without the need for prior generation of region proposals [9, 10, 11, 12, 13].

One of the most popular single-shot detectors among the first to appear is YOLO (You Only Look Once). Its working principle begins with dividing the image using a SxS size grid (Figure 2.3). The network then predicts B number of bounding boxes for each cell, containing 5 parameters: x, y, w, h, c. Among these, x and y denote the coordinates of the bounding box's center with regard to the cell's top left corner, while w and h are the bounding box's width and height relative to the image size. The parameter c represents a confidence level, indicating the model's confidence regarding the presence of an object within the bounding box and the accuracy of its coverage. In addition, the model predicts C number of parameters for each bounding box, where C denotes the number of categories. The parameters' assign a probability value to each category indicating the likelihood of the object predicted by the respective bounding box belonging to the given category. This way, a single CNN can predict several bounding boxes and their corresponding class probabilities for an image, where each prediction takes the shape of SxSx(B*5+C) [9].

The first version of YOLO was published in 2016 and since then, different versions have been developed. The core difference in YOLOv2 [10] (2017) is the introduction of the anchor boxes. The width and height of the predicted bounding boxes are no longer calculated relative to the image size, but the size of the anchor boxes. The architecture of the YOLOv3 [11] detector is based on a more complex, 53 layer deep Darknet backbone and comprises a total of 106 layers. It performs the predictions with multiple scale factors and during the forward pass, it merges different activation maps in the architecture, which helps preserving the characteristics of smaller objects.

YOLOv4 (2020) is a widely used single-shot detector nowadays. Compared to the former versions it has been supplemented with numerous special techniques which the authors categorize into two groups: bag of freebies and bag of specials. Bag of freebies contain methods, that only change the training strategy or increase the training cost, but result in

[1] Source: `https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6`.

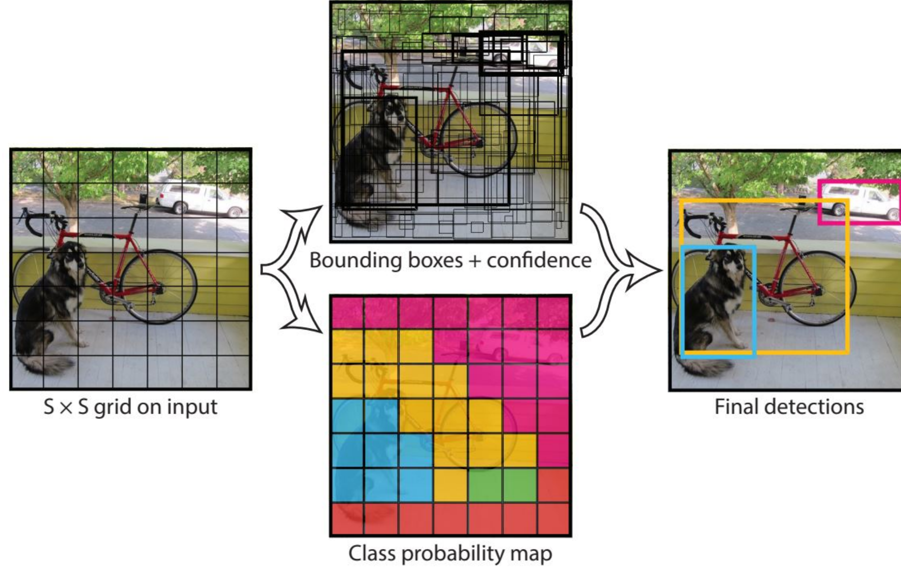

Figure 2.3: Working principle of the YOLO object detector [9].

the improvement of the model accuracy. For example, revolutionized data augmentation techniques like pixel-wise adjustments and combination of two images, focal loss and label smoothing to deal with semantic distribution bias in datasets, and the objective function of bounding box regression belong to this category. Bag of specials are plugins and post-processing methods that slightly increase the inference cost, but significantly improve the model accuracy. Some examples include Mish activation, Cross Stage Partial Connections (CSP), Spatial Pyramid Pooling, Spatial Attention Module and Path Aggregation Networks(PANet). Altogether, these extensions enlarge the perception field, improve the attention mechanism and enhance the capability of integrating visual features, which lead to SOTA detection results on MS COCO dataset [12].

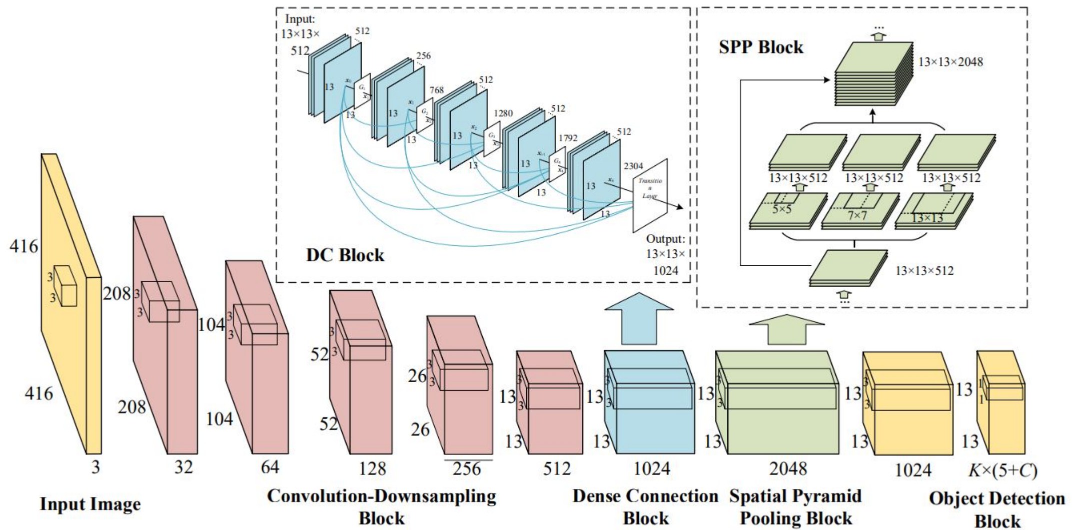

Figure 2.4: Architecture of the YOLOv4 object detector [14].

Since the beginning of this study, several new versions of YOLO have been released, introducing improvements on YOLOv4 which was leveraged in this thesis. YOLOv5 [15],

although released by different authors than the previous versions and lacking proper documentation, focuses on meeting the requirements of industrial use cases. YOLOv6 [16] brings enhancements in architecture, label assignment, loss function and quantization. YOLOv7 [17] introduces extended efficient layer aggregation, novel model scaling techniques, along with re-parametrization planning. YOLOv8 [18], developed by Ultralitics (like YOLOv5), further enhances the architecture and developer experience, establishing it as the current SOTA version.

2.3 Neural Network Pruning

Depending on the approach used for removing weights from a CNN we can distinguish between structured and unstructured CNN pruning. In the case of unstructured pruning, weights are removed from the convolutional layers individually, by setting their values to zero. Although, this indeed results in a sparser network regarding its parameter count, the network's architecture remains unchanged, precluding the decrease in computation cost. The structured pruning method, on the other hand, removes groups of weights, such as entire filters or channels. The removal of such weight groups results in a different, smaller architecture. Therefore, the number of matrix multiplication operations decreases, which allows the exploit of hardware resources and ultimately leads to speedup [19].

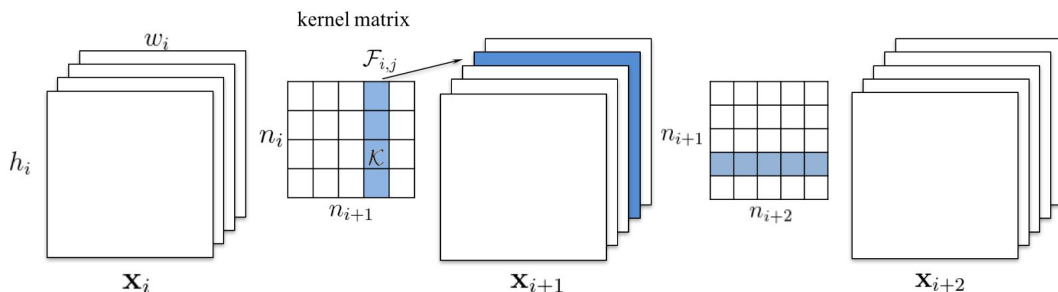

Figure 2.5: Process of structured pruning on CNNs [20].

Consider n_i as the number of input channels for the ith convolutional layer and let h_i and w_i denote the height and width of the input activation maps for the given layer. The ith convolutional layer transforms the input activation maps $x_i \in \mathbb{R}^{n_i \times h_i \times w_i}$, into the output activation maps $x_{i+1} \in \mathbb{R}^{n_{i+1} \times h_{i+1} \times w_{i+1}}$, which serve as inputs to the following convolutional layer. To achieve this transformation, n_{i+1} 3D filters $\mathcal{F}_{i,j} \in \mathbb{R}^{n_i \times k \times k}$ need to be applied on the n_i input channels, each filter producing one output activation map. The aforementioned 3D filters are composed by n_i 2D filers $\mathcal{K} \in \mathbb{R}^{k \times k}$ called kernels. As a result, the number of operations performed during the processing of a convolutional layer can be calculated as follows:

$$n_{op} = n_{i+1} * n_i * k^2 * h_{i+1} * w_{i+1} \tag{2.3}$$

If $\mathcal{F}_{i,j}$ is removed from a 3D filter, the corresponding $x_{i+1,j}$ output activation map will not be generated. This step saves $n_i * k^2 * h_{i+1} * w_{i+1}$ operations. The kernels in the $i+1^{th}$ layer that would have performed the transformations on the removed activation maps will also be removed. This further reduces the number of operations by $n_{i+2} * k^2 * h_{i+2} * w_{i+2}$.

Therefore, by removing m filters from the ith convolutional layer, both the ith and the $i+1^{th}$ layer's computation cost will drop by m/n_{i+1} [20].

It is a common practice to fine-tune the reduced model after the pruning process to regain its accuracy that usually somewhat degrades during pruning. It is worth mentioning that the accuracy of the pruned model may outperform that of the original model. The explanation for this is that pruning is basically a regularization method, as the removal of the weights reduces the model's confidence, thereby mitigating the overfitting as well [19].

2.4 Reinforcement Learning (RL)

2.4.1 Reinforcement Learning Task

Machine learning algorithms can be divided into multiple groups based on the characteristics of the training database. The first major group is supervised learning, where alongside the training data, we have access to their corresponding training labels as well which contain the expected output, also called ground truth. In this case, the task of the algorithm is to predict the labels as precisely as possible. During the training process, at each iteration the predicted labels are compared to the ground truth, thus, the algorithm can adjust to the problem and will perform more and more accurately over the time. In contrast, in the case of unsupervised learning, the ground truth remains unknown to the machine learning algorithm. As a result, its task is to discover similarities and differences in the training data by uncovering hidden structures and perform categorization based on these findings. The combination of the two aforementioned learning approaches is known as semi-supervised learning, where only a certain portion of the training data is labeled. Self-supervised learning should also be mentioned, where, similarly to supervised learning, the labels are provided, but were generated automatically [1].

In the context of categorizing machine learning algorithms, reinforcement learning (RL) constitutes the third major group. The task for the learning algorithm (agent) is to solve the machine learning task independently, without relying on enormous labeled training datasets. To do so, the agent interact with an external environment which can be described with states (Figure 2.6). By observing these states, the agent chooses an action at each step which, after the execution, causes the environment to change its state. Unlike in supervised learning, the correct action remains unknown to the agent. Instead, the environment rewards the agent based on the quality of the chosen action. Therefore, the agent aims to discover a strategy for choosing the actions that will result in the highest possible reward [21].

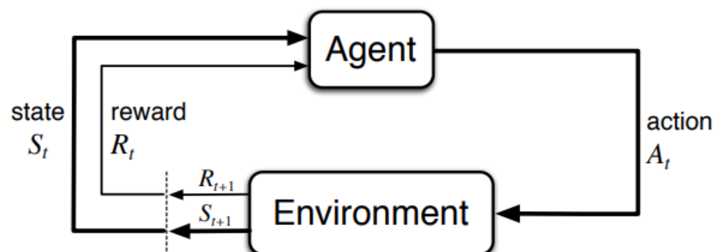

Figure 2.6: Interaction of the RL agent and the environment [21].

In mathematical terms, the problem introduced in the previous paragraph can be described by the Markov decision process. Formally, it can be written as $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma)$, where $\mathcal{S}$ and $\mathcal{A}$ denote the state and action space respectively, $\mathcal{R}(s, a)$ is the expected reward for a given state-action pair,$\mathcal{P}(s, a)$ denotes the state-transition probabilities and γ stands for the discount-rate, which is used to weigh previous rewards [21]. The agent's decision strategy is given by the policy; a function that assigns an action to each state:

$$\pi(a, s) = P(a_t = a | s_t = a) \tag{2.4}$$

By following the policy π, the agent transitions the environment from one state to another, earning a corresponding reward. This state-action-reward sequence is referred to as trajectory. With the help of that, we can define the value function. While the reward indicates the goodness of an action in an immediate sense, the value function defines what is beneficial in the long run; it returns the expected total reward the agent may accumulate starting from a given state following the π policy:

$$V^{\pi}(s) = E[\sum_{t \geq 0} \gamma^r r_t | s_0 = s, \pi] \tag{2.5}$$

Similarly, the the Q-function can be defined which assigns the expected total reward to a state-action pair:

$$Q^{\pi}(s, a) = E[\sum_{t \geq 0} \gamma^r r_t | s_0 = s, a_0 = a, \pi] \tag{2.6}$$

Depending on the functions employed by the agent while attempting to find the optimal policy, various RL methods can be distinguished. In the case of Q-Learning the agent strives the optimal Q-function, which is then used to derive the policy. The Q-function satisfies the Bellman-equation, which states that the highest achievable total reward from a given state-action pair equals the sum of the currently received reward and the highest achievable total reward from the next state [21]. Using this, we can define the parameter update rule and the cost function:

$$\hat{Q}(s, a, \theta) = E[r + \max_{a'} \hat{Q}(s', a', \theta)] \tag{2.7}$$

$$\Delta Q = r + \max_{a'} \hat{Q}(s', a', \theta_{i-1}) - Q(s, a, \theta_i) \tag{2.8}$$

$$L_i(\theta_i) = E[\Delta \theta_i^2] \tag{2.9}$$

$$\frac{\partial L_i(\theta_i)}{\partial \theta_i} = E[\Delta Q \frac{\partial Q(s, a, \theta_i)}{\partial \theta_i}] \tag{2.10}$$

where θ denotes the network's parameters, ΔQ is the error of the Bellman-equation and L_i is the cost function. According to this, the Q-function is updated at each iteration until it converges to the optimum.

Another type of RL methods is the Policy Gradient algorithms, that seek to learn the policy directly without incorporating the value function. The simplest algorithm in this category is the Monte Carlo Policy Gradient (REINFORCE). Its working principle can be described as follows: given a state, the agent assigns probabilities to the actions, then takes the action with the highest probability, for which it receives a reward from the environment. The agent receives a reward from the environment based on this chosen action. If the reward is high, the network is updated to increase the probability of choosing that action in the future. Conversely, if the reward is low, the probability of selecting that action is reduced. Through this iterative process, by the end of the training, the sequence of the actions, or the policy, converges towards the optimum. The cost function of the REINFORCE algorithm:

$$J(\theta) = E[r(\tau)] = \oint_{\tau} r(\tau)p(\tau, \theta) \tag{2.11}$$

where θ denotes the NN's parameters, τ is the trajectory obtained by following the policy, $r(\tau)$ is the reward function and $p(\tau)$ represents the probability of taking an action given a trajectory. The cost function's derivative with respect to θ is proportional to the derivative of the logarithm of the network's output ($\pi_\theta(a_t|s_t)$). The latter can be calculated using the Monte-Carlo estimation method [22]:

$$\nabla_\theta J(\theta) = \sum_{t \geq 0} r(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t) \tag{2.12}$$

2.4.2 Multi-Agent Actor-Critic Methods

One typical issue with Policy Gradient methods is the rewards' exclusive non-negative values, meaning that the agent's decision is always reinforced, even in the case of a poor decision, but to a lesser extent. A possible approach for overcoming this issue is to only consider a reward good if it is higher than the expected total reward at a given state. To achieve this, we can define advantage function as the difference between the Q-function associated with the state-action pair and the value function associated with the state:

$$A^{\pi_\theta}(s, a) = Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s, a) \tag{2.13}$$

Using this, the cost function of the Policy Gradient is calculated as follows:

$$\nabla_\theta J(\theta) = \sum_{t \geq 0} (Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s, a)) \nabla_\theta \log \pi_\theta(a_t|s_t) \tag{2.14}$$

Methods based on the introduced concept are called Actor-Critic methods, where the names denote two different networks: the actor network is responsible for finding the

optimal policy using the Policy Gradient algorithm, whereas the critic network aims to produce the advantage function using Q-learning [21].

Actor-Critic methods excel in solving different RL tasks and have several variants,including Asynchronous Advantage Actor-Critic (A3C), A2C, Deep Deterministic Policy Gradient (DDPG) and Proximal Policy Optimization. Among these, A2C and Proximal Policy Optimization will be presented in more detail in this thesis, as they play a key role in my research.

The previously introduced RL methods have significant advantages and drawbacks compared to one another, yet they share one common characteristic: only a single one agent interacts with the environment at a time. However, a framework introduced in 2016 revolutionized this approach by enabling multi-agent training. The framework uses the A2C algorithm and allows several agents to run on multiple instances of the environment simultaneously [23]. These agents follow different policies, therefore are able to explore various regions of the environment. At designated iterations, known as episodes in RL, the trained network is updated based on the combined experiences of the different agents. This approach leads to leads to a more stable training process, eliminating the need for the Experience Replay method previously used for the same purpose [24]. As a result, the algorithm's memory and computation costs are reduced.

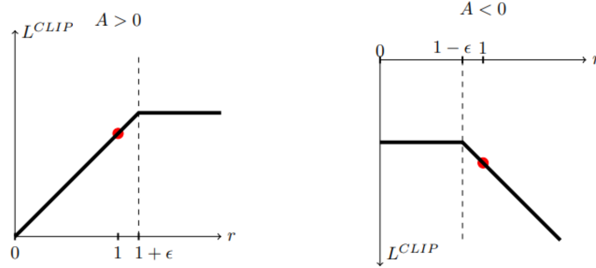

Figure 2.7: Working principle of the PPO actor cost function [25].

Another widely used, multi-agent SOTA algorithm is the Proximal Policy Optimization (PPO) [25]. Its core innovation lies in the special actor cost function, which aims to solve the overconfident policy phenomenon. To do so, the algorithm takes into account the policy from the previous step and restricts the probability of taking a specific action to be significantly higher than in the previous step. Let r_t denote the ratio between the probability of the current and probability of the previous policy:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{2.15}$$

Knowing the sequence of the observed actions and states, if the probability of taking action a_t is higher than in the previous policy, r_t will be greater than 1. Based on this, the cost function can be defined as:

$$r_t^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t] \tag{2.16}$$

where ϵ is the so-called cutting factor, whose value is usually around 0.2. According to this, if the r_t ratio falls within the 1-e range, the error equals the product of the r_t ratio

and the advantage function A_t. Otherwise, if the r_t ratio falls outside the 1-e range, the product value is truncated (Figure 2.7).

In essence, this cost function operates the same way as the earlier published TRPO method [26]. However, TRPO is a highly complex algorithm, and due to PPO's simpler implementation of the same idea, it has become a more frequently used method.

2.5 Transformer Neural Networks

2.5.1 Processing Sequential Data

While feedforward NNs have revolutionized classical machine learning, they suffer from limitations when handling sequential data. Due to their lack of an explicit inner state and requirement for fixed-size inputs, they are unsuitable for capturing long-term dependencies within sequences of varying lengths. This is a prominent obstacle for their employment for solving natural language processing (NLP) tasks like language modeling or machine translation, nevertheless, makes them less efficient in video processing, navigation or any kind of continuous data processing tasks.

Recurrent Neural Networks (RNNs) address the aforementioned limitations by incorporating an inner state that is responsible for retaining information from earlier time steps. The RNN layer is a key part of the architecture which combines the current input with the inner state from the previous time step while determining its output. After the forward pass, the RNN layer is rolled out, and its states at different time steps are considered as traditional NN layers with outputs, allowing the use of the backpropagation algorithm [22, 27] (Figure 2.8).

This approach, however, raises several challenges. Firstly, as training progresses, the rolled out network grows, gradually slowing down the training process due to increased computational complexity. Secondly, the weights are identical in every instance of the RNN layer in the rolled out network, which will likely result in the exploding or vanishing gradient phenomenon during backpropagation. Furthermore, due to the sequential nature of RNNs as sequence models, parallel computation becomes unfeasible [28, 27].

The first issue can be mitigated by limiting the maximal size of the rolled out network. As for the second issue, the creators of the Long Short-Term Memory (LSTM) [29] cell have constructed a structure where the derivative of the current inner state with respect to the previous inner state is consistently close to 1. This approach indeed reduces the occurrence of gradient instability, the slow training and the parallelization obstacles, however, still remain.

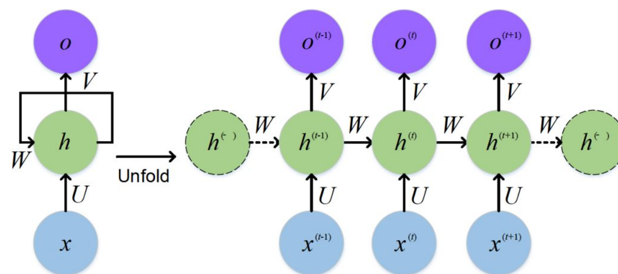

Figure 2.8: Standard and unrolled RNN [30].

2.5.2 The Transformer Architecture

Transformer NNs were introduced in 2017 in the paper called Attention Is All You Need, and as the creative name indicates, they rely entirely on the self-attention mechanism, in contrary to former end-to-end memory networks that use recurrence and convolution [31]. With that, they have become the foundational architecture for cutting-edge NLP models and have shown great results in vision-related tasks as well [32].

A fundamental concept in sequence analysis is to consider a sequence that comprises an arbitrary number of fixed sized elements, known as embeddings. An example of such embeddings is the representation of words in a sentence using word embeddings. The self-attention mechanism represents a sequence by computing one embedding's relation to the others in the sequence. In order to incorporate trainable parameters, the authors employ the Query (Q), Key (K) and Values (V) approach. Query is the selected embedding which calculates a dot product with all embedding in the sequence, referred to as keys. The resulting output is divided by the square root of the dimension of the Keys and subjected to a softmax function, yielding the weights. These weights are then multiplied by the corresponding embeddings, known as values, to obtain the final output. As all queries, keys and values are represented as matrices, their values can be trained as neural networks [31]. They call the aforementioned type of attention Scaled Dot-Product Attention and is calculated as follows:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.17}$$

where d_k denotes the dimension of keys.

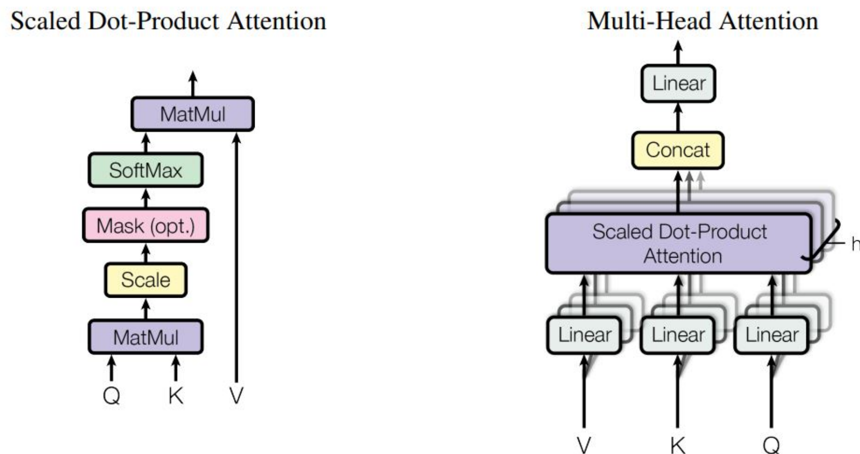

Figure 2.9: Scaled Dot-Product Attention and Multi-Head Attention [31].

The paper proposes a technique known as Multi-Head Attention, aiming to create a richer representation of the input, which leads to increased performance. To achieve this, the queries, keys, and values undergo linear projections using learned linear transformations. Each of these projections are then fed to a Scaled Dot-Product Attention block in parallel, referred to as heads. The final output is obtained by concatenating the resulting attentions from the different blocks. To inject information about the positions of the embeddings in

the sequence - for instance, about the positions of the words in a sentence -, positional encoding is used. The authors employ sine and cosine functions with varying frequencies for this purpose, which proves effective for text data, but is unsuitable image data [31].

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}}) \tag{2.18}$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}}) \tag{2.19}$$

where i denotes the dimension, while pos is the position. This means, that each dimension of the positional encoding is associated with a sinusoid, where the wavelengths form a geometric progression ranging from 2π to $10000 * 2\pi$.

The complete architecture of the transformer model is illustrated in Figure 2.10. The model takes the input sequence which is constructed by the embeddings. After performing positional encoding, the output is passed through the encoder block. The encoder is composed by a multi-head self-attention mechanism and a simple, position-wise funny connected NN. Its primary task is to map the input sequence from symbolic form to continuous representations. The decoder block's construction is similar to that of the encoder, and its purpose is to produce the output sequence from the continuous representation. To do so, it generates symbols iteratively, considering the previously generated symbols to generate the subsequent ones at each step [31].

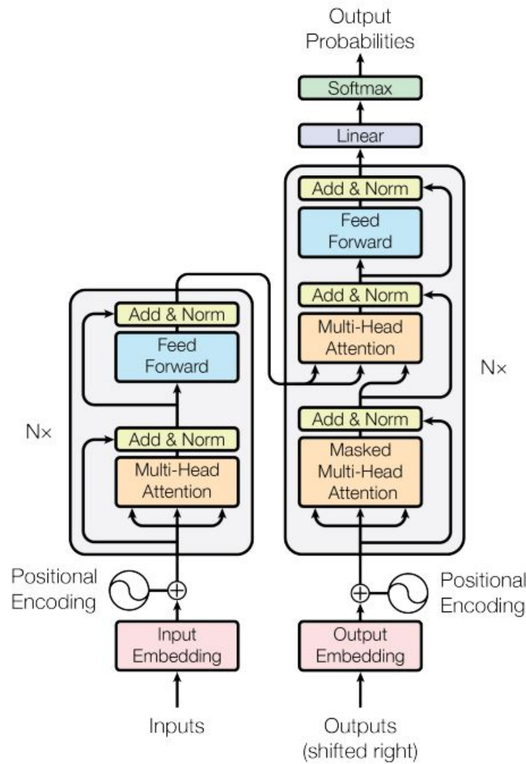

Figure 2.10: The Transformer model architecture [31].

Chapter 3

Preliminaries

3.1 Conventional Pruning Methods

The NN procedure has been known since the 1980s, however, the demand for its application began to grow in the past decade with the widespread use of DNNs. The first publications focused on constructing carefully-designed rules to select those parameters, that could be removed from the model without causing major performance degradation. Researchers soon discovered that magnitude-based pruning outperforms random pruning. Consequently, many studies began to examine the optimal magnitude threshold for removing a certain parameter [33]. Other researches explored more complex rules that considered not only weight magnitudes, but also the impact of architectural dependencies on the overall structure of the network [34, 35].

Regarding the compression of YOLO-type detectors, various conventional pruning methods have been proposed previously. Although these methods rely on manufactured rules, they are so well-designed that they have achieved outstanding results. For example, YOLObile [36] prunes YOLOv4 with a block-punched pruning scheme and achieves 93 % sparsity with 8.3 % mAP loss. Another notable solution is YOLO-Tight [37] which leverages sparsity training for pruning YOLOv3, and prunes approximately 90 % of the model's parameters with little to no accuracy degradation.

The comparison of all these great result is cumbersome due to the lack of standardized metrics for evaluating the effectiveness of pruning methods [19]. Furthermore, when comparing results across different network architectures, direct comparisons become even more challenging. However, despite the great results presented above, conventional pruning methods require human intervention, making model compression time-consuming and sub-optimal as the enormous search space cannot be explored entirely manually. As a solution to this problem, recent research has leveraged RL to automate the pruning process.

3.2 Reinforcement Learning-Based Pruning

In the following subsections, SOTA RL-based pruning approaches will be introduced, that primarily inspired my work.

3.2.1 AutoML for Model Compression (AMC)

AMC [38] was the first RL-based pruning approach to be published, and it surpassed conventional rule-based compression methods by achieving a higher compression ratio while better preserving the initial accuracy. It can perform both structured and unstructured pruning on several classifier NNs and the R-CNN object detector. The number of weights removed from a given layer is expressed as a percentage (sparsity), and unwanted weights are selected using a magnitude-based search. It deploys the DDPG actor-critic algorithm as the RL agent, and its state space consists of the following 11 parameters:

$$(t, n, c, h, w, stride, k, FLOPs[t], reduced, rest, a_{t-1}) \tag{3.1}$$

where t is the index of the layer being pruned, the size of the filter is $n \times c \times k \times k$, and the size of the input is $c \times h \times w$. $FLOPs[t]$ is the FLOPs of layer t, *reduced* is the total number of reduced FLOPs in previous layers, *rest* is the number of remaining FLOPs in the following layers, while a_{t-1} denotes the chosen action in the previous step. AMC exploits a continuous action space that denotes the sparsity of the given layer (pruning ratio). The agent receives a reward only after pruning all the layers in the model (sparse rewards) that is computed by using the following reward function:

$$R_{Param} = -Error * log(Param) \tag{3.2}$$

where *Error* denotes the accuracy deterioration after pruning and is calculated by evaluating the pruned model on a validation dataset without fine-tuning.

AMC achieves great results; when pruning the R-CNN detector, 50 % sparsity is achieved with 0.1 % higher mAP than the initial. The algorithm, however, only converges after a significant number of iterations due to the sparse rewards.

3.2.2 Pruning Using Reinforcement Learning (PuRL)

The PuRL [39] automated pruning method addressed this issue by applying a training procedure that rewards the agent after pruning each layer in the NN (dense rewards). With this improvement, PuRL needs 85% fewer RL episodes than AMC when pruning ResNet-50 trained on ImageNet. This method focuses mainly on sample efficiency and accuracy, thus, it performs only unstructured pruning. It uses the Deep Q-Network as the RL-agent, and its state space is defined as follows:

$$s = (l, a, p) \tag{3.3}$$

where l is the index of the layer to be pruned, a is the current accuracy achieved on the test set after fine-tuning for one epoch, and p corresponds to the proportion of weights pruned thus far. The action space consists of pruning coefficients (α) that determine the amount of pruning. The pruning criteria is a magnitude threshold derived from the α value and the standard deviation of the weights in a given layer. All the weights that have a smaller absolute magnitude than this threshold have to be removed:

$$PrunedWeights_i(\alpha) = \{w||w| < \alpha\sigma(w_i)\} \tag{3.4}$$

where $\sigma(w_i)$ is the standard deviation of weights in layer i. PuRL uses a discrete action space i.e. $\alpha \in \{0.0, 0.1, 0.2, .., 2.2\}$. The dense reward approach requires the use of a complex reward function that calculates the reward at each step in the episode:

$$R(s) \ = \ -\beta(max(1 - \frac{A(s)}{T_A}, 0) \ + max(1 - \frac{P(s)}{T_P}, 0\ )) \tag{3.5}$$

Here, $A(s)$ and $P(s)$ denote the test accuracy and sparsity at state s and β corresponds to a fixed scaling factor of 5. This reward function ensures that the agent optimises for the desired accuracy (T_A) and sparsity (T_P).

Even though PuRL needs considerably fewer episodes than AMC to obtain convergence, neither of them publish run times in their papers. In both methods, the accuracy degradation is determined by evaluating the pruned model on a test dataset. In the case of PuRL, the pruned model is additionally fine-tuned for one epoch before validation. Furthermore, due to dense rewards, these steps are carried out in each episode as many times as the number of layers in the neural network. Therefore, in the aforementioned methods, the RL agent might converge after a few episodes of training, but their training procedure is excessively time-consuming.

Chapter 4

Problem Statement

Considering the insights gained from the previous chapter, the objective of this study is to design and implement a RL-based automatic pruning system for the YOLOv4 object detector that achieves pruning results of comparable quality to those reported in existing literature while significantly reducing the training time of the RL agent. According to my observations, the prolonged training time of RL agent in the existing publications can primarily be attributed to the validation of the pruned model in each episode to obtain environmental variables.

Therefore, I propose a novel solution to replace this part of the system. Instead of performing validation on the pruned model during the agent's training, the environmental variables are determined by a pre-trained State Predictor Network (SPN). The SPN is trained beforehand on automatically generated data using the YOLOv4 model that is desired to be pruned. With this modification, the characteristics of the YOLOv4 model are integrated into the SPN, eliminating the need to load the model during RL agent training. As a result, the time-consuming part of the existing automatic pruning systems is concentrated before training, thus, the RL agent's training can be performed rapidly, allowing extensive exploration of the optimal training settings. The pruning system removes entire channels from the YOLOv4 model, enabling more effective utilization of hardware resources. The remaining parts of the system, like the selection of the deleted channels, reward function, state and action space of the agent, were inspired by existing solutions in the literature.

After achieving outstanding results for the aforementioned problem, the secondary goal of this study is to investigate the generalization ability of the automatic pruning system across various datasets and DNN architectures. Firstly, I examine its performance in the former case, and further explore the enhancement achieved through transfer learning with the pre-trained SPN model. To ensure generalization across different DNN architectures, an SPN model that can handle dynamic input sizes is required. For this task I propose a transformer-based SPN model and compare its performance to that of the original SPN model.

The flowchart of the proposed automatic pruning system is illustrated in Figure 4.1. The pruning process consist of two main stages. The first stage involves data generation for SPN training, by generating random pruning coefficient (α) sequences. These are then used to prune the trained YOLOv4 model and the pruned models' performance is evaluated on the validation dataset. Subsequently, the SPN is trained. In the second stage, the initial state of the YOLOv4 model is fed to the RL agent which iterates through its layers and determines an α action for each layer. After each action is generated, along with the model

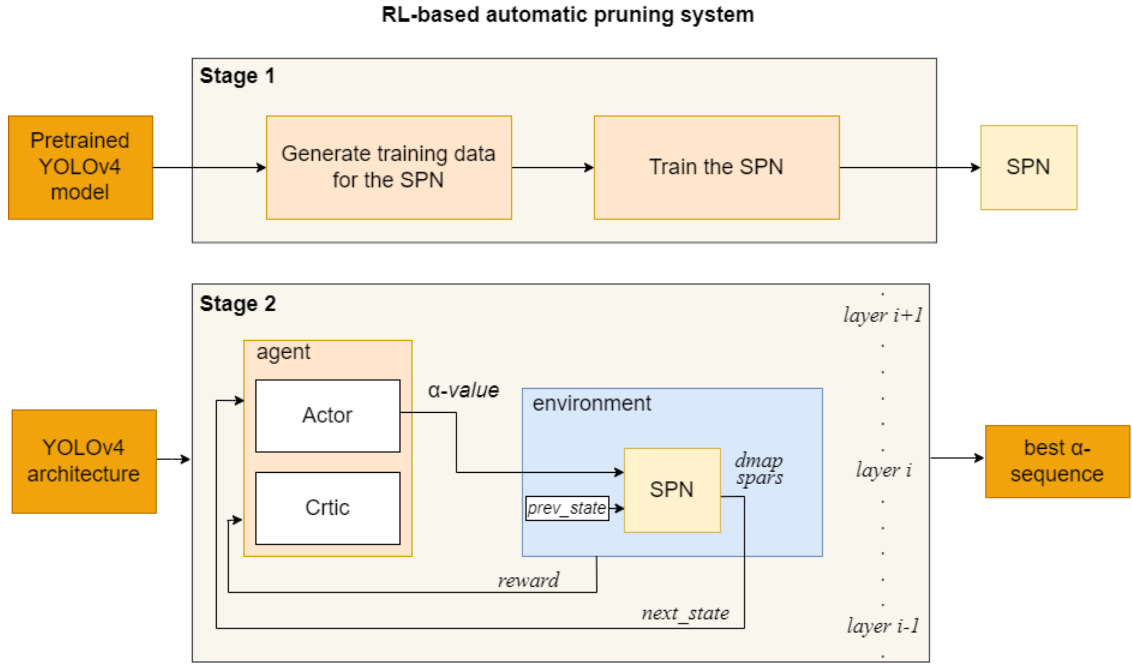

Figure 4.1: Flowchart of the proposed RL-based automatic pruning system.

state, it is passed to the SPN. The SPN predicts the next state and the environmental variables required to calculate the reward, which is then used to reinforce the agent. At the end of the RL agent's training process, an α sequence is obtained, which is used to prune the initial YOLOv4 model in a single step.

In summary, the proposed system's task is to automatically discover an α sequence of length n, that enables pruning the initial YOLOv4 model to create a network with the minimum number of parameters while ensuring the smallest possible accuracy degradation. Here, n denotes number of the prunable layers in the YOLOv4 architecture, while in terms of RL, the α sequence represents the policy being learned by the agent.

Chapter 5

Proposed Methods

5.1 Metrics

To enhance the readability of the upcoming sections, in this section, I introduce the evaluation aspects used in this thesis. They are categorized into the following groups: evaluation of a fundamental object detector, assessment of the pruned model, performance of the SPN and the pruning system. For measuring these aspects, I employed some conventional metrics, accompanied with several self-designed ones, which are detailed below.

Performance of object detectors:

- Precision [%]: Indicates the proportion of the correctly detected objects out of all detections.
- Recall [%]: Indicates the proportion of the correctly detected objects out of all ground truth objects.
- Mean Average Precision (mAP) [%]: The average precision calculated for each category individually, averaged across all the categories. This is a widely used, comprehensive indicator of the goodness of an object detector as it measures the trade-off between precision and recall.

Performance of the pruned model:

- Number of parameters (#params) [M]: The number of parameters present in the NN model.
- Performance degradation (*dmap*) [%]: This self-designed metric indicates the extent of the performance degradation of the pruned model. Taking into consideration the benefits of the mAP metric, I utilized that for constructing the dmap metric. It shows the change of the pruned model's mAP compared to that of the initial model. Therefore, a smaller dmap is desirable. Its formula is defined as follows:

$$dmap = 1 - \frac{mAP_{pruned}}{mAP_{initial}} \tag{5.1}$$

- Sparsity (*spars*) [%]: Indicates the proportion of the removed parameters from the initial model. A higher sparsity is desirable. Its formula is designed as follows:

$$sparsity = 1 - \frac{\#params_{pruned}}{\#params_{initial}} \tag{5.2}$$

Performance of the SPN:

- Accuracy of *dmap*/*spars* with a margin [%]: The SPN's task is to predict two of the aforementioned metrics: the mAP deterioration and the sparsity. To measure this performance, I designed a metric that indicates the proportion of the predictions that fall within a specifies margin. For example, if the accuracy of *dmap* with a 2 % margin is 90 %, it means, that 90 % of the SPN's predictions for the *dmap* fall within a 2 % radius of the ground truth *dmap*.
- Mean absolute error for *dmap*/*spars* [%]: Indicates the mean absolute difference between the predicted and the ground truth *dmap* or *spars* values.
- Maximum error for *dmap*/*spars* [%]: The maximum difference between the predicted and the ground truth *dmap* or *spars* values.

Performance of the pruning system:

- Performance of the pruned model: To measure the quality of a pruning system, obviously, one of the most important indicators is the performance of the pruned model that it produced.
- Speed: On the other hand, the speed of the pruning system is also a crucial aspect. To measure this, I take into consideration the time needed from loading the trained model to obtaining the best performing pruned model, including the data generation for the SPN, training the SPN and the hyperparameter fine-tuning for the RL agent. Also referred to as total development time.

5.2 Development Environment

The source code of the proposed pruning system is implemented in a Python 3.8.10 environment powered by an Intel(R) Core(TM) i5-8400 CPU and an NVIDIA Titan X (Pascal) graphics card. Some parts of the generalization development were carried out on an Intel(R) Xeon E5-2698 v4 CPU and NVIDIA Tesla V100 graphics card, in Python 3.6.9 environment. All the deep learning specific approaches are deployed using the Pytorch [40] machine learning framework.

5.3 Training The YOLOv4 Object Detector

The development process begins with the preparation of the baseline model, that will serve as the input for the automatic pruning system. The chosen baseline is the YOLOv4 object detector, implemented according to [41]. The pre-trained Darknet weights on the MS COCO dataset are publicly available for such purposes [42]. Due to hardware restrictions, however, a smaller dataset is used to train the YOLOv4 model through transfer learning.

The selected KITTI dataset is collected from traffic scenarios, containing 9 different object categories and approximately 7500 pictures [43]. The dataset is split into train, validation, and test sets in a ratio of 60 %, 20 %, and 20 %, respectively. The model is trained until it reaches 72.6 % mAP (Figure 5.1) with the hyperparameter settings summarized in Table 5.1.

Table 5.1: Training hyperparameters for YOLOv4 object detector on KITTI dataset.

epochs	batch size	img size	optimizer	lr scheduler	initial lr	final lr	weight decay	device
500	16	544	Adam	LambdaLR	1e-3	0.1	5e-4	Titan X

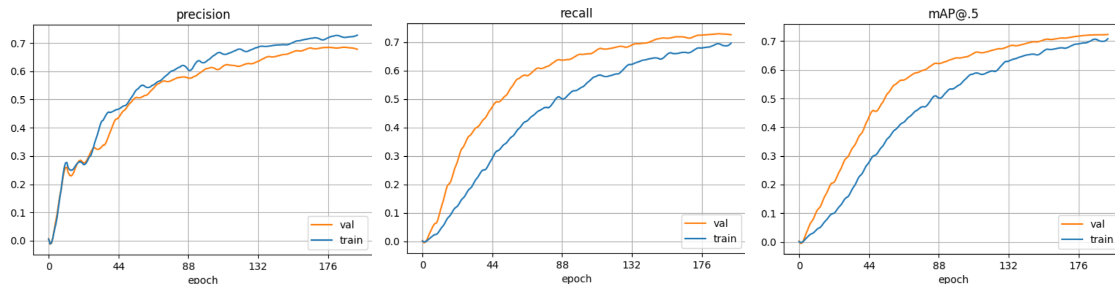

Figure 5.1: Learning curves of YOLOv4 object detector on KITTI dataset.

5.4 Modifying the YOLOv4 Architecture During Pruning

The goal of this study extends beyond achieving a sparse network in terms of parameter count, focusing on speedup and optimizing hardware utilization. To accomplish this, structured pruning is implemented, which involves removing entire channels from the neural network. For each layer, an α pruning coefficient is provided by the agent which indicates the channels to be eliminated from the given layer. The precise concept behind this will be discussed in the upcoming Section 5.6.2 in more detail. However, to understand the current section, it is sufficient to know, that α is a discrete variable that can take on 23 different values. This section discusses the prunability of the YOLOv4 architecture, potential obstacles and introduces the algorithm that modifies the architecture during the pruning process.

5.4.1 Prunable Layers In The YOLOv4 Architecture

During the pruning process, the algorithm alternately removes channels from the input and the output channels of the layers, starting with the output channels of the 1st convolutional layer. The complexity of the YOLOv4 architecture makes this process challenging, as it contains several connections between its layers (Figure 5.2). Therefore, when modifying a layer, careful consideration must be given to adjusting the dependent layers accordingly.

The architecture of the YOLOv4 object detector is built up by the following layer types:

- Convolutional: Simple convolutional layer.
- Maxpool: Simple max pooling layer.

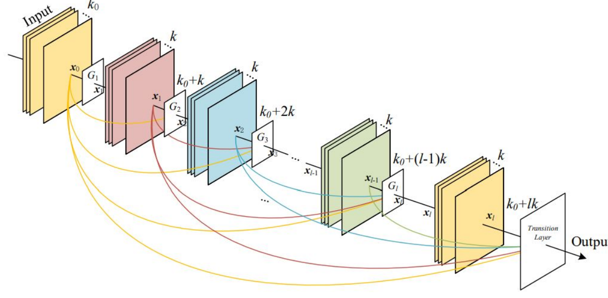

Figure 5.2: Dense connections in the YOLOv4 architecture [14].

Table 5.2: Working principle of the architecture modifier algorithm.

Layer Type	dim	Action
Convolutional	0	deleted_channels += get_indices(α) dim_next = dim XOR 1 (*new indices*)
Convolutional	0	deleted_channels += deleted_channels[-1] dim_next = dim XOR 1 (*indices according to prev. layer's out. channels*)

FeatureConcat
Features to concat:
layer_indices
	0/1	ii = []
		for j in layer_indices
		    if dims[j] == 0
		        ii += indices[j]
		deleted_channels += ii
		dim_next = 1

WeightedFeatureFusion Features to concat: *layer_indices*	0/1	i ← *index of layer, where output channel number is MIN from layer_indicies* deleted_channels += deleted_channels[i] dim_next = 1
ConvBeforeYOLO	1	deleted_channels += deleted_channels[-1] dim_next = dontcare
ConvBeforeYOLO	1	deleted_channels += [] (*no pruning*) dim_next = dontcare
YOLOLayer	0/1	deleted_channels += [] (*no pruning*) dim_next = dims[-1]
Maxpool Upsample	1	deleted_channels += deleted_channels[-1] dim_next = 0 dims[-1] = 0

- Upsample: Simple upsample layer.
- FeatureConcat: This layer concatenates the channels of the designated layers. The size of the resulting layer is the sum of the designated layers' sizes.
- WeightedFeatureFusion: This layer computes the sum of the designates layers' channels. The size of resulting layer will be the same as the size of the smallest among the designated layers.
- YOLOLayer: The detection layer. It generates the parameters of the detected objects from the input activation map.

- ConvBeforeYOLO: This is also a simple convolutional layer which is not distinguished from the others in the architecture. However, as it has meaningful characteristics considering the algorithm design, it needs be distinguished.

To introduce the architecture modifier algorithm, a few additional parameters need to be defined. The *dim* and *dim_next* parameters indicate whether the input or output channels will be pruned in the ith and $i+1^{th}$ layers respectively. If $dim = 0$, the algorithm removes channels from the output channels, while in case of $dim = 1$, the input channels are pruned. For the simple convolutional layers, the algorithm alternates between pruning the output and input channels, therefore, in each pruning step $dim_next = \text{dim XOR 1}$. The *deleted_channels* container stores the indices of the removed channels for each layer, while the *dims* array contains the *dim_next* value for the ith layer. With this knowledge in mind, the working principle of the architecture modifier algorithm is introduced by Table 5.2.

After considering the dependencies, the introduced algorithm allows new α values for only 44 layers out of the total 160, which will be referred to as "prunable layers" moving forward. Given that the removed output channels from a designated layer will also affect the input channels of the subsequent layer, the algorithm prunes channels from a total of 88 layers.

5.5 The State Predictor Network (SPN)

5.5.1 The SPN's Role In The System

The State Predictor Network simulates the external environment in this task; taking the model state (*model_state*) and a sequence of pruning coefficients (α) sequence as input, it predicts the model's sparsity and the accuracy degradation, measured by *spars* and *dmap*. Regarding its architecture, it consists of 3 fully connected hidden layers with 256, 512 and 256 neurons respectively, and ReLU activation is applied between them (Figure 5.3).

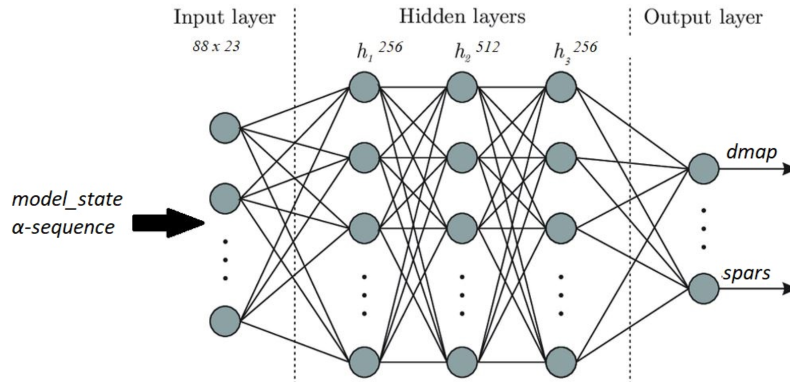

Figure 5.3: State Predictor Network.

The model state is represented by a 2D matrix with the dimensions of $[N_{players} \times N_{features}]$, where $N_{players}$ denotes the number of prunable layers in the architecture (which is 44 as discussed earlier), while $N_{features}$ refers to the number of observed features. Here, the only feature observed is the sparsity reached thus far (*spars*), which simplifies the state matrix to a state array. In the beginning of each episode, the model state is initialized with -1 values. As each layer is pruned, the corresponding cell in the state array is updated. For example, when pruning the first layer, since no parameters have yet been removed,

the model state array will consist of 44 elements, all set to -1. When pruning the second layer, the first value in the array will change to $spars_0$ which represents the model's sparsity achieved after pruning the first layer based on the predictions from the SPN. It is worth noting that during the initial stages of pruning, the model state primarily contains -1 values, leading to potential imbalanced training. This is why a 1D feature vector is chosen, focusing solely on capturing the sparsity achieved.

The SPN was trained on automatically generated data using self-supervised learning. It is important to note, that the generated training samples may not cover all the possible variation of the α sequence. Consequently, during the training of the RL agent, the SPN's predictions may not be is not precise enough in some cases. However, I would like to highlight here, that the primary task of the SPN is not to provide highly accurate predictions for every single case, but to learn to predict the most typical cases accurately. By doing so, it guides the agent towards the right direction and accelerates the convergence which highly depends on several hyperparameters. Adjusting a single hyperparameter often requires more than 50 training episodes to determine if it leads to convergence. When leveraging the SPN, the agent leads to the optimal policy significantly faster. Nevertheless, the final results should be determined by performing actual pruning and validation on the YOLOv4 model, using the best α sequence suggested by the automatic pruning system.

5.5.2 Automatic Data Generation

As discussed earlier, there are 44 prunable layers in the YOLOv4 architecture, and for each layer an α variable determines the channels to be removed. As the α can take on 23 different values, when pruning a model, there are a total of 23^{44} possible variations to construct the α sequence, resulting in a value of magnitude 10^{59}. Obviously, generating that many samples is preposterous, however, it is not even necessary. In certain cases, extending an α sequence with a varying α values does not yield substantial differences. One typical example for this is when the initial layers of the model are pruned with high α, given that a higher alpha value implies a higher number of removed channels. In such instances, the accuracy degradation is radical even in the early stages, and choosing higher or lower α values for the subsequent layers have minimal impact, the mAP of the pruned model will remain close to zero. While it is still essential to document these scenarios, a considerably less amount of samples is sufficient to train the model to recognize this pattern. On the contrary, when the pruned model performs well even after pruning more than half of its layers, it becomes less sensitive to assigning high alpha values to the remaining layers. In such cases, the changes in mAP caused by different α values are smaller, therefore, it becomes necessary to collect a larger number of samples to thoroughly documents these patterns.

Algorithm 1 Automatic data generation for training the SPN.

```
model ← pre-trained YOLOv4 model
base_nParams ← calculate the number of parameters in pre-trained YOLOv4 model
base_mAP = test(model)
for layer_i in model do
    if layer_i is prunable then
        α = random(0.0, 2.2)
        model, state_features = prune_network(model, layer_i, α)
        mAP = test(model)                                          ▷ On a dataset of 500 images
        nParams ← calculate number of parameters in pruned model
        dmap = calc_dmap(base_mAP, mAP)
        spars = calc_sparsity(base_nParams, nParams)
        save(dmap, spars, state_features)
```

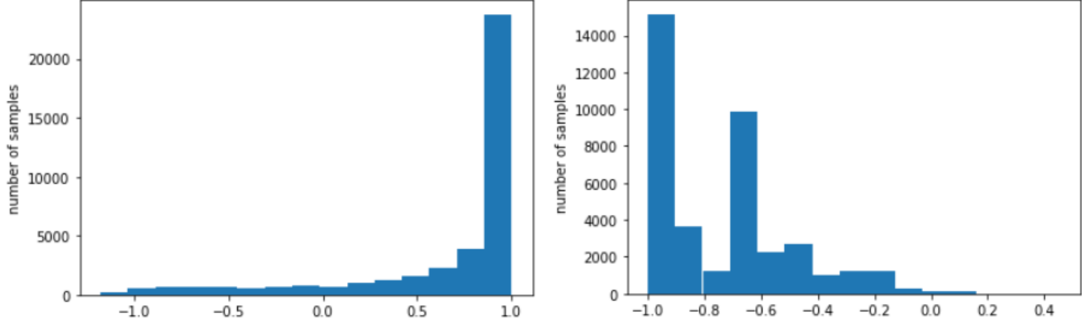

Figure 5.4: Distribution of the *dmap* (left) and *spars* (right) labels.

The data generation process follows the Algorithm 1. Initially, the trained YOLOv4 model is loaded and evaluated using the validation dataset to obtain its mAP metric and the number of parameters in the model. These metrics are considered as baseline when calculating accuracy degradation and determining sparsity. The algorithm generates random α values layer-by-layer, forming an α sequence. At each layer, the initial model is pruned using the corresponding sequence, and the resulting pruned model is evaluated on the same validation dataset. The validation results and the model parameters are then saved for future use as input for the SPN and the RL agent. In addition to the samples generated by this algorithm, some manually constructed α sequences are also added to the dataset to expand the coverage of possible patterns. At the end of the data generation process, approximately 57000 samples were collected. As depicted in Figure 5.4, the labels of the generated data show imbalanced distribution, which needs to be taken into account and addressed in future development.

5.6 Automatic Pruning Using Reinforcement Learning

5.6.1 State Space

At each step in the episode the agent receives the model state (S_i) as an input, which is presented by a 2D matrix with dimensions of $[N_{players} \times N_{features}]$, the same way as in the case of the SPN. In other words, the model state is a feature vector (s) for every prunable layer in the model. Here, the number of observed features $N_{features} = 6$:

$$s = (in_chs, out_chs, k, stride, pad, spars) \tag{5.3}$$

In the feature vector the first 5 features refer to the convolutional layer: in_chs and out_chs denote the number of input and output channels, k is the filter size, $stride$ and pad denote the stride and padding, while $spars$ corresponds to the percent of weights pruned until the ith layer. The size of he state S_i is always the same, regardless of the layer being pruned. In case of those layers that have not yet been pruned, the feature vector is filled with -1 values.

5.6.2 Action Space

The action space consists of α actions that determine the amount of pruning for each layer. As structured channel pruning is performed in this study, the pruning criteria is a magnitude threshold for the norms of the channels. The threshold is derived from the standard deviation of the norms of the channels $\sigma(ch_i)$ and the α value. All channels that have a smaller absolute magnitude than this threshold have to be removed from the ith layer:

$$RemovedChannels_i(\alpha) = \{ch||ch| < \alpha\sigma(ch_i)\} \tag{5.4}$$

This approach, inspired by the PuRL method [39], modifies Equation 3.4 to make it suitable for the structured pruning problem. The adaptation is necessary since they perform unstructured pruning, thus examine the standard deviation of weights in a layer. Similar to the PuRL, a discrete action space is utilized here, with a step size of 0.1: $\alpha \in \{0.0, 0.1, 0.2, .., 2.2\}$.

5.6.3 Reward Function

Designing a proper reward function for a RL agent that is trained with dense rewards is not trivial. Superficially, the expectations for an adequate reward function are as follows: if the chosen action results in high *spas* and low *dmap*, the agent should receive a very good reward; for low *spars* and high *dmap* a very bad reward should be given, and for both small *dmap* and *spars* a medium reward should be assigned.

If the agent was only rewarded after pruning the entire network, this theory would be quite straightforward to put into practice. However, using dense rewards comes with the following problem: it may happen, that an action results in a certain *spars* and *dmap* combination, which categorizes as poor performance according to the aforementioned expected working principle. Meanwhile, if the position of the layer the action was assigned to, is also taken into account, the chosen action is a highly advantageous decisions in reality. For example, if only a small portion of the parameters is removed from the initial layers, the *dmap* will not decrease significantly, however neither will the *spars*. This will not lead to a high reward, however, in fact these are good decisions, since later the agent can still achieve the desired result by pruning the upcoming layers. Consequently, the agent will more likely choose actions for the first layers that result a high *spars* and high *dmap*. Unfortunately, such decision has a detrimental effect, as it is impossible for the model to recover from this aggressive pruning.

The designed reward function, inspired by the complex reward function of the PuRL method, aims to tackle this issue. The original reward function was modified in order to make the importance of sparsity and accuracy degradation more scalable:

$$R(s) = -\beta(c_{dmap} * max(\frac{dmap(s) - T_{dmap}}{1 - T_{dmap}}, 0) + c_{spars} * max(1 - \frac{spars(s)}{T_{spars}}, 0)) \tag{5.5}$$

Here, $dmap(s)$ and $spars(s)$ are the accuracy degradation and sparsity in state s, c_{dmap} and c_{spars} are weighting coefficients for them, T_{dmap} and T_{spars} denote the desired final *dmap* and *spars*, while β corresponds to a fixed scaling factor of 5. In this study, producing

a model with lower performance degradation is prioritized above higher sparsity, hence these parameters are calibrated as summarized in Table 5.3.

Table 5.3: Parameters for the reward function.

T_{dmap} [%]	T_{spars} [%]	c_{dmap}	c_{spars}	β
20	60	1.1	1.0	5

5.6.4 Training The RL Agent

Using the SPN, only the model's state has to be fed to the agent instead of the entire model. This solution requires significantly less amount of GPU memory, allowing multiple agents to operate at the same time. The widely popular A2C algorithm is employed for multi-agent training with a medium-sized architecture: 3 linear hidden layers in actor with 512, 1024 and 256 neurons and two linear hidden layers in critic with 256 and 512 neurons, with ReLU activations in both networks.

Algorithm 2 Advanced Actor-Critic algorithm for finding the best α sequence.

```
bS ← batch_size as the number of agents
aS ← size of the action space
nL ← number of layers that can be pruned
α_sequence[bS, nL] ← []
episode ← 0

while episode ≤ max_episodes do
    model_arch ← load model architecture
    state ← load initial model state
    for each layer_i in model_arch do
        if layer_i is prunable then
            distribution[bS, aS] = actorNet(state)
            q_value[bS, 1] = criticNet(state)
            action[bS] = sample from distribution

            dmap[bS], spars[bS] = SPN(action, state)
            state ← environment(dmap, spars)

            reward[bS] = reward_function(dmap, spars)
            α_sequence.append(action)

▷ check if the predicted dmap and spars are close enough to the real values
    if validation episode then
        model ← load trained model
        pruned_model ← prune model by chosen α_sequence
        real_dmap, real_spars ← validate pruned_model on validartion dataset
        if real_dmap, real_spars are close enough to dmap, spars then
            continue training

return best α_sequence[bS, nL]
```

The proposed RL-based automated pruning algorithm (2) loads the pre-trained YOLOv4 model's state and makes a copy for every agent. Then it goes through the model layer by layer, and chooses an α action for each prunable layer based on the probability distribution of the possible actions predicted by the actor. Receiving the chosen action and the model's state, the SPN predicts the *dmap* and *spars* environmental variables, determines the new state and calculates the error based on them. Meanwhile, the critic seeks to find the optimal value function. To verify the proper behavior of the SPN, in certain episodes, the algorithm randomly takes ten agents and prunes the initial model using their policy. The

pruned models are then validated on a small dataset, and if the obtained environmental variables are close enough to those predicted by SPN, the agents are trained further.

At the end of the training, the algorithm returns bS number of n-sized α sequences, where bS is the batch size - or the number of agents - and n denotes the number of prunable layers. The best policy - or α sequence - has to be chosen by pruning and validating the initial model using them.

5.6.5 Model-Based Learning

There are several theories regarding how to distinguish between model-based and model-free learning. One of them is derived from the agent's relationship with the environment: in the case of model-free learning, the agent interacts with the environment directly, whereas, in the case of model-based learning, their connection is indirect as the agent interacts with a simulated environment. Another theory is based on the way the agent selects actions [44]. Generally, the agent's purpose is to find actions that will result in the highest possible reward. In model-free learning, however, the agent is not directly linked to the reward; it only attempts to enhance the probability of the chosen actions in a given step. After a considerable amount of time, this will lead to the improvement of the reward as well, but solely because of the large number of samples. Therefore, the reward is quite noisy at the beginning of the training, and it converges only after the agent gains significant amount of experience. In model-based learning, on the other hand, the agent can plan directly based on the reward and is able to choose an action that yields a high reward without requiring extensive experience.

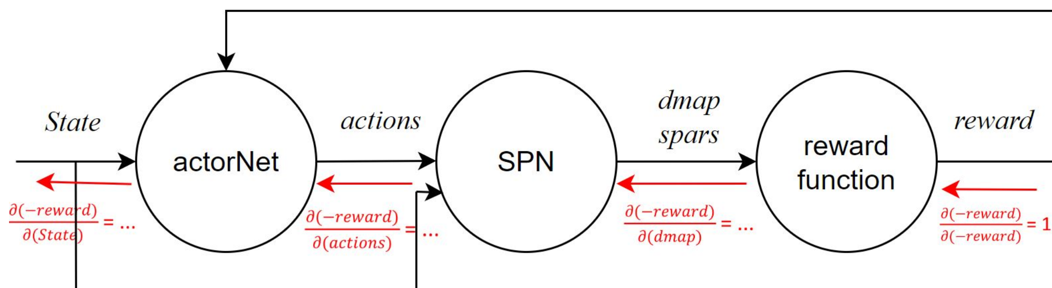

Figure 5.5: Computation graph between the State Predictor Network and *reward*.

The SPN enables to implement a model-based approach in the proposed pruning system. Firstly, its primary purpose is to simulate the environment, which is one crucial aspect of model-based learning. Secondly, the SPN links the actor and the reward's computation graph (Figure 5.5); therefore, by calculating the derivative of the reward, we directly get the gradient that indicates how the weights in the actor network have to change in order to make the actor choose actions yielding higher reward. As a result, the gradient becomes less noisy, which leads to more stable and faster learning.

5.7 Generalization Of The Automatic Pruning System

5.7.1 Generalization Across Datasets

As outlined in Section 5.6.2, the selection of the channels to be pruned depends on two factors: the α pruning coefficient and the standard deviation of the norms of channels in the given layer. This implies that using the same α value may lead to different number of removed channels if the weights have different values in different models. If the same YOLOv4 model is trained for different tasks on different datasets, the resulting models will have different weight configurations. Consequently, the most beneficial α sequence obtained for one dataset may not be the most beneficial for others. In the context of this study, it means that the most effective α sequence for the YOLOv4 model trained on the KITTI dataset may not yield the best results when pruning the YOLOv4 model trained on a different dataset.

To increase the performance of the automatic pruning system applied to the YOLOv4 model trained on a new dataset, the first stage of the process undergoes modification. With the YOLOv4 model trained on a new dataset, a small collection of samples is generated, employing the same methodology discussed in Section 5.5.2. This newly created dataset is significantly smaller than the original dataset used for training the SPN model, consisting of 5120 samples instead of the previous 57000. The trained SPN model is subsequently fine-tuned through transfer learning on the newly generated dataset. This approach leverages the existing knowledge of the SPN, yet ensures its better suitability for the new task. Following this step, the updated SPN model is reintegrated to the pruning pipeline, and the RL agent is re-trained using it, seeking for an α sequence that yields better performance in the given context.

For the assessment of the system's generalization ability across datasets, the YOLOv4 object detector was trained on both the COCO and the Pascal VOC datasets. Unfortunately, no favorable result was achieved with the Pascal VOC dataset; not only the final mAP was poor, but the model demonstrated a high degree of overfitting. On the other hand, the COCO dataset yielded a trained model with a mAP of 53.8 %. Pruning a model that originally shows poor performance is likely to lead to a serious performance drop. For this reason, the model trained on the Pascal VOC is abandoned and solely the model trained on the COCO dataset is used for further analysis in this study. The training configurations for training on COCO are summarized in Table 5.4, and the learning curves of the best runs are depicted in Figure 5.6.

Table 5.4: Training hyperparameters for YOLOv4 object detector on COCO dataset.

epochs	batch size	img size	optimizer	lr scheduler	initial lr	final lr	weight decay	device
300	8	640	Adam	LambdaLR	1e-2	0.1	5e-4	Tesla V100

5.7.2 Generalization Across DNN Architectures

When striving to push the boundaries of the proposed automatic pruning system and expand the range of its features to enable generalization across different DNN architectures, numerous challenges abound. The first limitation comes from the first stage of the pipeline which is not entirely automated; the algorithm that performs the architecture modification during the pruning process is designed manually, specifically for the YOLOv4 architec-

Figure 5.6: Validation curves during the training of YOLOv4 on COCO.

ture. To overcome this obstacle, an algorithm, that is able to automatically detect layer dependencies in an architecture is required. Furthermore, the varying number of prunable layers across different architectures must be taken into consideration. The number of prunable layers determines the dimensions of the model state matrix, which serves as the input to the SPN. This calls for the necessity of an SPN architecture that is able to handle dynamic input sizes. The following subsections discuss the ideas implemented to address the aforementioned challenges.

5.7.3 Torch Pruning Library

Recently, a notable contribution has been made in the field of automatic structural pruning, with the introduction of the Torch Pruning library. This library deploys the DepGraph algorithm [45] which is capable of detecting dependencies between the layers in an arbitrary neural network architecture. Unlike the standard PyTorch pruning function that only zeroises parameters through masking, the Torch Pruning library removes parameters and channels physically from models implemented with the PyTorch framework. This approach ensures that when pruning a specific layer, all dependent channels throughout the architecture are also removed according to the dependency graph. Consequently, it guarantees that the resulting network structure remains intact. Incorporating this library into the proposed pruning system paves the way for generalizing the solution across different architectures.

The Torch Pruning library fulfils the same role in the system as the self-designed architecture modifier algorithm. Once the channel indices to be pruned are determined for each layer using the method described in Section 5.6.2, it takes these indices and prunes the current and all dependent layers. Nevertheless, the DepGraph algorithm discovers some additional dependencies that the self-designed algorithm is unable to identify. As a result, when using the same α sequence, the Torch Pruning library might remove slightly more parameters from the model.

To demonstrate this phenomenon, a comparison was conducted between the amount of parameters removed from the same model when using the two algorithms with the same α sequence. The model used for the comparison is the YOLOv4 model trained on the COCO dataset. The two handcrafted α sequences with a length of 44 are as follows:

s1 = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0.3,0,0,0,0,0,0,0,2.0,0,0,2,0,2,0,1.0,0,2,0,1.8,0,0,0,0,0,2.2,2.2,2.2]

s2 = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2.2,0,0,0,0,0,0,2.2,0,0.1,0,0,0,0,0,0.2,0,2.2,2.2,2.2]

According to Table 5.5, when pruning the model with α sequence s1, the self-designed algorithm removed 52.07 % of the model's parameters, while the Torch Pruning library removed 64.07 %. It is important to note, that the difference ratio in the number of

Table 5.5: Amount of removed parameters from the YOLOv4 model trained on the COCO dataset with two different α sequences, using the self-designed architecture modifier algorithm and the Tosrch Pruning library.

	Removed parameters with α sequence	
	s1	s2
Self-designed	52.07 %	40.25 %
Torch Pruning	64.07 %	40.25 %

removed parameters cannot be defined universally, as it may vary depending on the α sequence chosen. For example, in the case of s2, the number of removed parameters are the same for both algorithms. This implies that in this case, the self-designed algorithm identified all dependencies that were detected by DepGraph.

Additionally, the Torch Pruning library not only discovers more dependencies, it increases the number of prunable layers. When a layer's input channels are pruned due to the dependencies from the previous layer, the self-designed algorithm does not remove additional channels from its output channels. However, the Torch Pruning library takes a different approach and further prunes the the output channels. As a result, it identifies 107 prunable layers in the YOLOv4 architecture, compared to the previous 44, meaning that a new α value can be chosen for 107 layers. This change significantly increases the RL agent's action space, therefore makes the RL task more complex.

5.7.4 The Tranformer SPN Architecture

Although transformer NNs are commonly associated with natural language processing tasks, they are perfectly suitable for improving the proposed pruning system's generalization ability across architectures. While their attention mechanism is not the primary motivation in this particular task, their ability to handle varying input sizes can be leveraged effectively. Changing the original SPN architecture to a transformer NN enables the application of the proposed pruning system on different DNN architectures that have varying number of prunable layers.

In this concept, the input sequence that can have varying size is the model state with the corresponding α sequence. The embeddings are the feature vectors in the model state, that represent a given layer, concatenated with the corresponding α value. Consequently, the length of the input sequence is determined by the number of prunable layers in the DNN architecture to be pruned.

The transformer-based SPN architecture is illustrated in Figure 5.7, along with the dimensions of its elements. The input sequence is first fed to a linear layer to increase the feature vector's dimension. This is followed by the positional encoding module, which is responsible for injecting information about the layer positions in the architecture. For this task, sine and cosine functions with varying frequencies are employed. The positional encoded sequence serves as the input for the encoder module, which is implemented using the built-in PyTorch TransormerEncoderLayer and TransformerEncoder modules. As for the output, there is no need for generating sequences with varying sizes, the decoder module is abandoned. Instead, the first output sequence from the encoder is passed to a final linear layer, which produces the two environmental variables, namely the *dmap* and $spars$.

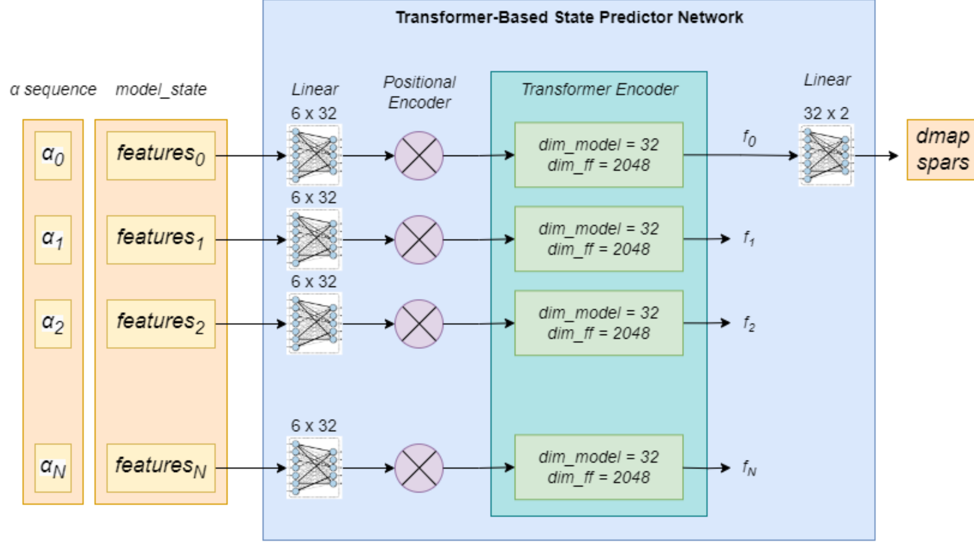

Figure 5.7: Transformer-Based State Predictor Network. N denotes the number of prunable layers in the input DNN architecture.

The model state is represented by a 5-element feature vector, containing the following information: $(in_chs, out_chs, k, stride, pad)$. The experiments conducted in order to find the optimal architecture setting will be discussed in detail in the results section. However, the finally used settings are summarized in table Table 5.6. Here, N_heads denote the number of heads, N_layers is the number of encoder layers, dim_model is the embedding dimension (also the converted feature vector dimension), dim_ff is the number of channels in the encoder layers and dropout is the dropout applied on the encoder layer.

Table 5.6: Model hyperparameters for the transformer-based SPN.

N_heads	N_layers	dim_ff	dim_model	dropout
2	2	2048	32	0.05

Chapter 6

Results

6.1 Towards The Optimal Solution

Before diving into the discussion of the results obtained with the proposed pruning system, this section provides a summary of the experimental results aimed to address the key challenges arising during the development.

6.1.1 Model State Size

The model state is represented by a 2D matrix with dimensions $[N_{players} \times N_{features}]$, where $N_{players}$ denotes the number of prunable layers in the architecture (which is 44 as discussed earlier), while $N_{features}$ refers to the number of observed features. This state is used in two parts of the system: it serves as input for the SPN and for the agent's actor and critic networks. As outlined earlier, the feature vectors have different sizes in these cases.

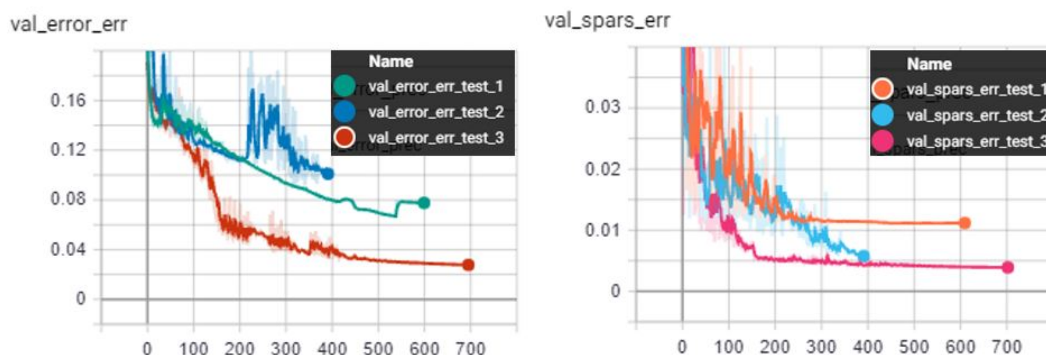

Figure 6.1: Mean average error for *dmap* (left) and *spars* (right) in test cases *test*_1, *test*_2 and *test*_3.

When serving the input for the SPN, it only contains the sparsity level achieved until the given layer, which helps to reduce training imbalances arising from the already imbalanced labels. Figure 6.1 depicts the mean absolute error of the *dmap* and *spars* labels predicted by the SPN in three different test cases, in which different feature vectors were used. In the first case *test*_1, the observed 6 features are $(in_channels, out_channels, k, stride, pad, spars)$, and the corresponding values in the state array are updated from their initial -1 values as the pruning progresses. In *test*_2,

the same 6 features are observed, however, the layer-specific values are loaded into the state array before the pruning starts, which significantly reduces the number of -1 values in the matrix. Evidently, the achieved sparsity level and the chosen α values for each layer are updated after pruning each layer. Finally, in *test*_3, the feature vector only comprises the *spars* value.

By observing the validation results in these different test scenarios, it can be deduced that a 1-element feature vector proves to be the most beneficial for training the SPN. The curves also indicate that learning to predict the *spars* is way less challenging than to predict the *dmap* metric. Nevertheless, in *test*_3, the trained SPN model is able to predict both metrics with less than 2 % mean absolute error. The training hyperparameters are summarized in Table 6.1.

Table 6.1: Training hyperparameters for the SPN.

epochs	batch size	loss function	optimizer	lr scheduler	initial lr.	final lr.	weight decay	device
2400	2048	LogCoshLoss	Adam	CosineAnne-alingLR	1e-3	1e-05	1e-5	Titan X

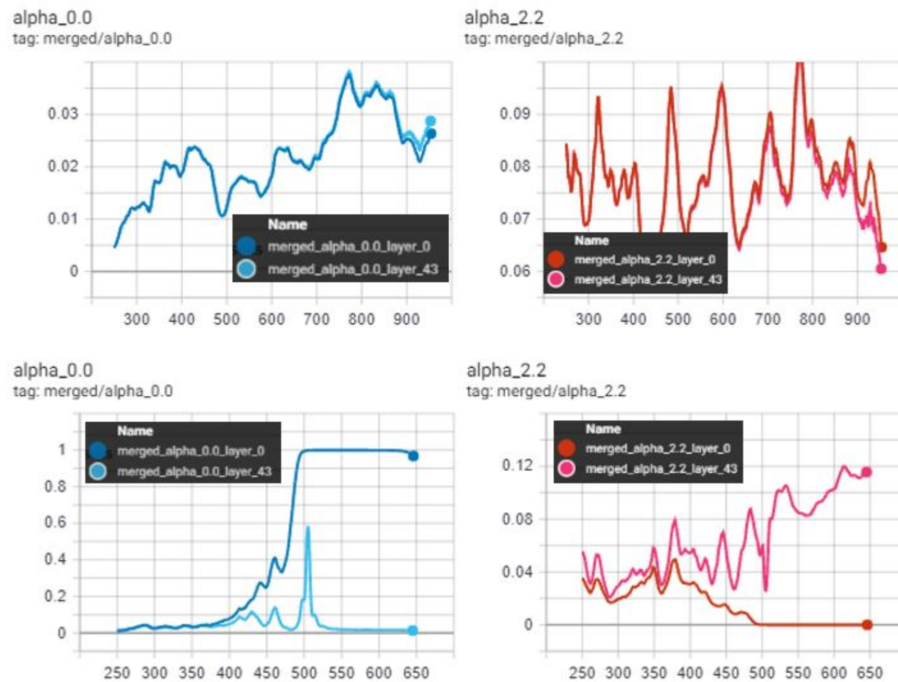

Figure 6.2: Probabilities of choosing $\alpha = 0.0$ and $\alpha = 2.2$ actions for the first and the last prunable layers, using a 1-element (top) and a 6-element feature vector (bottom).

On the other hand, in the case of the RL agent, the feature vector consists of 6 different features. The explanation to this lies in the agent's behavior. Figure 6.2 demonstrates that if a 1-element feature vector is used, the agent cannot distinguish between the first and the last layers: the predicted probability distribution is independent of the layers' features. This phenomenon is demonstrated by the equal probability of selecting a specific α action for both the first and last prunable layers. Ideally, the agent should exhibit a higher probability of choosing the $\alpha = 0$ action for the first layer and a lower probability

for the last prunable layer, and vice versa for the $\alpha = 2.2$ action. However, the agent fails to achieve this favorable behavior.

In contrast, when using a 6-element feature vector, the predicted distributions differ more and more in the case of the first and last layers as we move forward with the training. This suggests that the layer-specific features, that may seem to lack additional information, are crucial for the agent to effectively distinguish different layers.

6.1.2 Overconfident Policy

When deploying the A2C algorithm as the RL agent, the overconfident policy phenomenon soon becomes apparent. This phenomenon can most easily be illustrated by observing the probability of choosing the $\alpha = 0$ and $\alpha = 2.2$ actions for the first and last prunable layers. As expected, the agent quickly learns that it is beneficial to leave the initial layers in the architecture unmodified; it selects $\alpha = 0$ actions for them. However, after approximately 500 episodes, instead of exploring alternative options, the probability of choosing this action suddenly increases for every layer, and the agent fails to recover from this overconfident policy (Figure 6.3).

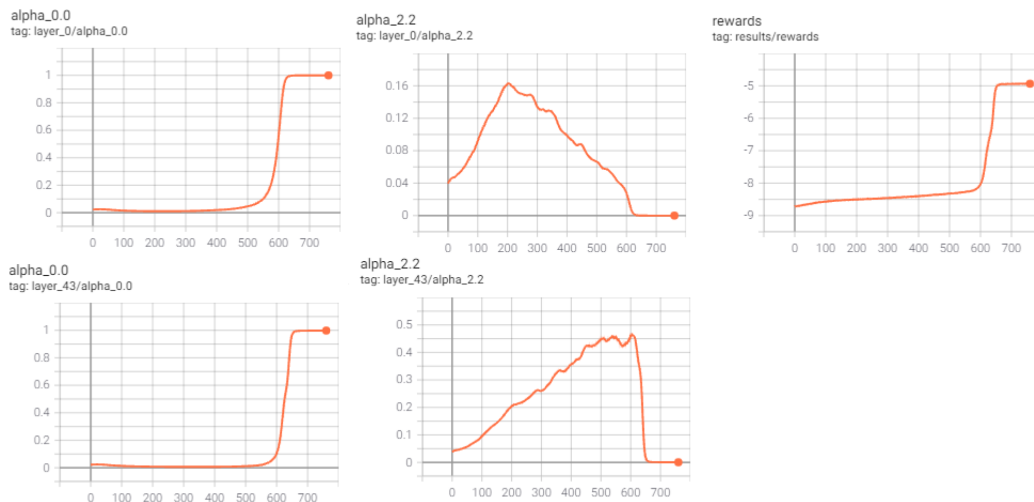

Figure 6.3: Reward and the probabilities of choosing $\alpha = 0.0$ and $\alpha = 2.2$ actions for the first and the last prunable layers with the A2C algorithm.

The SOTA PPO algorithm addresses this issue by implementing a special actor loss function as described in Section 2.4.2. However, while attempting to reduce the confidence of the policy, the truncation of the actor loss function leads to excessively small steps. Although the agent seems to move towards the optimum, it takes such small steps, that even after 14k episodes, the critical point is not even reached (Figure 6.4). This clearly demonstrates that in this particular case, the PPO algorithm brings more disadvantages than benefits.

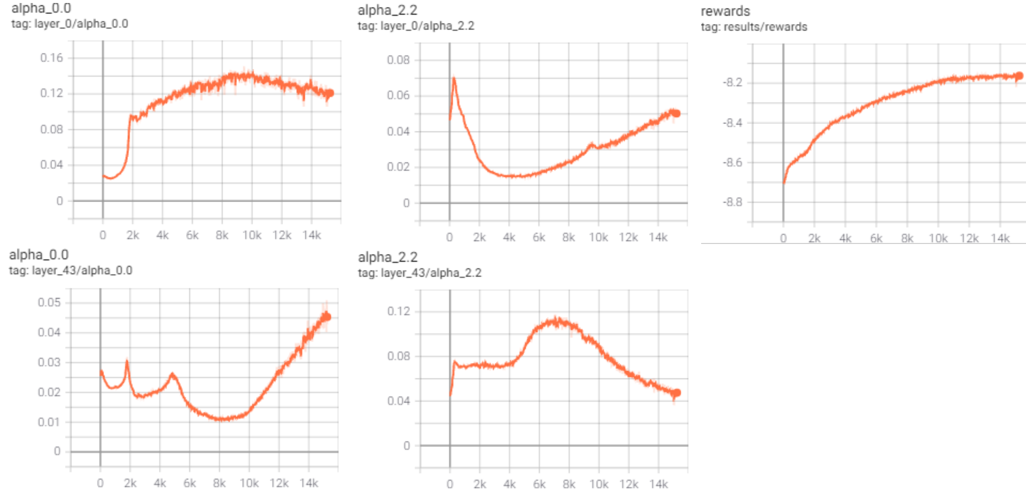

Figure 6.4: Reward and the probabilities of choosing $\alpha = 0.0$ and $\alpha = 2.2$ actions for the first and the last prunable layers with the PPO algorithm.

6.1.3 Careful Hyperparameter Selection

After numerous attempts, a careful selection of hyperparameters helps the A2C algorithm to overcome the critical point. By observing the different test cases with different hyperparameters, it became apparent that it is beneficial to chose high learning rate for the actor in the beginning of the training. This way, it quickly learns the advantage of leaving the initial layers unmodified. However, right before the policy would become overconfident around the 250th episode, the learning rate is decreased significantly, while the entropy is increased to allow the agent to explore the environment thoroughly. As a result, the reward curve is no longer stuck at the critical point, but converges to a higher value, as shown in Figure 6.5. The hyperparameter configurations for the actor and critic networks are summarized in table Table 6.2.

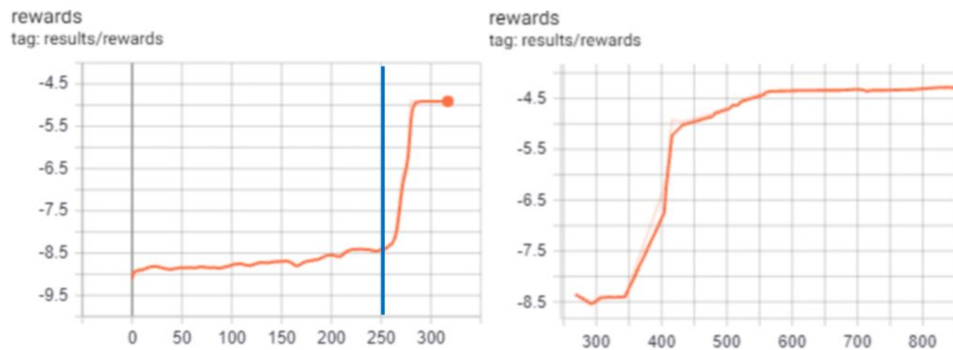

Figure 6.5: The received reward after pruning the last layer, averaged for the whole batch. From the 250th episode, the training is continued with a smaller learning rate and higher entropy.

The most important milestones of the RL agent's training are summarized in Table 6.3. Around the 200th episode, the reward received after pruning the last layer is notably poor. The *dmap* and *spars* values averaged across the batch indicate that the agents follow such policies, that although result in high sparsity, the accuracy degradation is 100 %.

Table 6.2: Hyperparameters for the actor and critic networks. Final values are applied from episode 250.

	ep.	batch size	opt.	initial lr	final lr	initial ent. coeff.	final ent. coeff.	device
actor	700	4096	Adam	1e-3	5e-4	5e-3	1e-2	Titan X
critic	700	4096	Adam	1e-2	1e-2	1e-2	1e-2	Titan X

The results obtained from real pruning and validation prove the correctness of the SPN's predictions. At episode 450, it is more likely for the agents to choose actions that cause less severe performance degradation, but the achieved sparsity is also lower. After 700 episodes of training, the SPN predicts decent results for the average for the batches: -0.8 % *dmap* and 30 % *spars*. Moreover, during the validation of the SPN, the real results turn out to be even better. The best pruned model has 49 % fewer parameters accompanied with 4.1 % decrease in mAP. The corresponding α sequence is the one illustrated in orange in Figure 6.6. As the reward curve already converges, it is unlikely for the model to improve and the final result to be better, therefore the training is terminated.

Table 6.3: Important milestones of the RL agent's training.

episode	reward	dmap [%]	sparsity [%]	best real dmap [%]	best real sparsity [%]
200	-8.551	100	65	99	70
450	-4.8	70	17	0	20
700	-4.32	-0.8	30	4.1	49

6.2 The Pruned YOLOv4 Model

The two most important indicators of the goodness of the proposed pruning system are the performance of the pruned model and the pruning system's speed. Therefore, this section introduces the evaluation of the results based on these objectives.

6.2.1 The Model's Performance

Table 6.4 summarizes the performance of the compressed models pruned with various α sequences. The baseline model is the YOLOv4 object detector trained on the KITTI dataset. The compared compressed models are the ones pruned with the following α sequences: *proposed* is the best performing sequence predicted by the proposed pruning system, while $handcrafted1$ and $handrafted2$ are manually crafted sequences that were selected randomly, following a simple assumption: the larger the layer index is, the larger α value is more beneficial. The different sequences are illustrated in Figure 6.6.

proposed = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,0,2,0,2,0,1,0,0,0,0,0,0,0,0,0,2,2,2]

handcrafted1 = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2.2,0,0,0,0,0,0,2.2,0,0.1,0,0,0,0,0,0.2,0,2.2,2.2,2.2]

handcrafted2 = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0.1,0,0,0,0,0,0,0,0,2.2,0,0,0,0,0,0,2.2,0,0.2,0.2,0,0,0.1,0,0,0.1,2.2,2.2,0]

As demonstrated by the results in Table 6.4, the α sequence predicted by the proposed pruning system outperforms both handcrafted sequences. It leads to a compressed model with 49 % fewer parameters compared to the baseline, accompanied by only 4.1 % accuracy

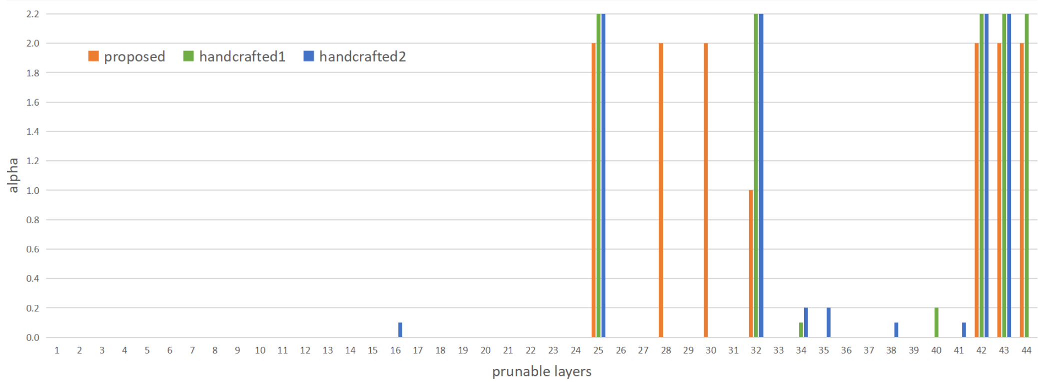

Figure 6.6: Different α sequences compared.

degradation. Meanwhile, the best handcrafted sequence causes 8 % accuracy degradation with 40 % sparsity. When leveraging the Torch Pruning (TP) library - while evidently keeping the prunable layer number at 44 - additional dependencies are discovered, resulting in a sparser model with 61 % sparsity. Nevertheless, there is a slightly higher accuracy degradation of 6.4 %. To verify that the predicted α sequence does not overfit to the validation dataset, it was evaluated on a different test dataset as well. The results does not show significant difference, indicating the pruning system's consistency across the KITTI dataset.

Table 6.4: Performance of compressed YOLOv4 models pruned with different α sequences.

	mAP [%]	#params [M]	dmap [%]	sparsity [%]
YOLOv4	72.6	63.9	-	-
handcrafted1	66.8	38.3	8	40
handcrafted2	45.8	42.2	36.9	34
proposed	**69.6**	32.6	**4.1**	49
proposed + TP	68	**24.9**	6.4	**61**
YOLOv4 + test	73.4	63.9	-	-
proposed + test	70.5	32.6	3.9	48.9

As far as the sparsification of the YOLOv4 object detector is concerned, there are several SOTA methods that achieve impressive results. One of them is the YOLObile framework, that removes 93 % of the parameters from a model trained COCO, with only a 8.3 % performance degradation [36]. Another notable result comes from the YOLO-Tight method, that prunes the YOLOv3 architecture to almost 90 % with only 1 % performance degradation [37]. Unfortunately, the proposed pruning system cannot be directly compared to these methods, as they either use different architectures or the baselines models were trained on different datasets. Although, it is evident that the proportion of removed parameters are much higher in the aforementioned cases, it is important to highlight, that these methods employ unstructured pruning, and prioritize higher sparsity over the low accuracy degradation. In contrast, this study focused on achieving as low accuracy degradation as possible. Accordingly, the reward function is calibrated to assign a higher weight to accuracy degradation compared to sparsity. With different settings, it would likely be

feasible to generate a pruned model with higher sparsity, yet a slightly higher performance degradation as well.

6.2.2 The Proposed Pruning System's Speed

To assess the speed of the proposed pruning system, it is compared to the AMC and PuRL RL-based pruning methods introduced in Section 3.2. However, the comparison is not possible directly here either, as the different approaches are not evaluated based on the same aspects. The proposed pruning system is designed specifically for the YOLOv4 object detector, while the PuRL was evaluated only on classifier networks, and the AMC was tested on both classifier networks and the Faster R-CNN object detector. Furthermore, these methods publish only the number of episodes needed for reaching convergence, but do not provide any information about the corresponding time duration. Since the main difference among these methods lies in the procedure used to determine the environmental variables, a comparable format is achieved by applying the proposed pruning system's implementation in all three cases, however, in in the case of AMC and PuRL, the SPN is replaced with the procedure they use to determine the environmental variables.

In the case of the AMC, the reward is calculated from the pruned model's accuracy, obtained by the validation of the pruned model without any additional fine-tuning steps. The agent receives a reward once in each episode, after pruning the entire model, therefore, the validation is performed once per episode. In the proposed system, the dense reward approach is employed; the reward is calculated after pruning each prunable layer, therefore, there are 44 calculations in each episode. The PuRL method also uses dense rewards, but in that case, prior to the validation step, the model is fine-tuned for one epoch on a small dataset containing 1000 images.

In the proposed pruning system, the RL agent requires 700 episodes of training to converge. To find the optimal training hyperparameters, state space size and to calibrate the reward function, 90 test runs were conducted in total. In each test run, the agent was trained for approximately 300 episodes to assess the progress of the training. In every 50th episode, validation is performed on 10 randomly selected α sequences from the batch to verify the proper functioning of the SPN. The initial automatic data generation is also part of the development, where a total of 57k samples were collected. To generate each sample, one validation step was necessary, taking approximately 14 days to complete. The training of the SPN itself lasted for 14 hours, and to find the optimal hyperparameters, 10 test runs had to be performed. Considering all this information, the total development time can be described as the sum of the following part-times:

$$t_{dev} = t_{data} + t_{SPN} + t_{tRL} + t_{fRL} \tag{6.1}$$

where t_{data} denotes the time needed for the automatic data generation, t_{SPN} is the time needed to train the SPN, t_{tRL} is the time needed to run the tests in order to find the optimal hyperparameters for the agent, and finally, t_{fRL} is the training time of the final model. These part times can be calculated as follows:

$$t_{data} = N_{samples} * t_{val} \tag{6.2}$$

$$t_{SPN} = N_{SPN} * t_{SPNtrain} \tag{6.3}$$

$$t_{tRL} = (test_{RL} - 1) * (N_{avgep} * t_{ep} + 10 * \frac{N_{avgep}}{50} * t_{val}) \tag{6.4}$$

$$t_{fRL} = N_{finep} * t_{ep} + 10 * \frac{N_{finep}}{50} * t_{val} \tag{6.5}$$

where $N_{samples}$ is the number of generated samples, t_{val} is the time needed to run the validation, N_{SPN} is the number of test runs needed for the SPN to converge, $t_{SPNtrain}$ is the time needed to train the SPN, N_{finep} is the number of episodes needed for the RL agent to converge in the final setup, N_{avgep} is the number of episodes needed to assess the progress of the RL agent's training, t_{ep} is the run time of one episode, and $test_{RL}$ represents the number of test runs required to find the best performing pruned model. The calculation of the total development time with the exact values:

t_dev = 5700*21 s + 10*14*3600 s + 89(300*6.32 s + 10*6*21 s) + (700*6.32 s + 10*14*21 s) = 1989248 s = 23.023 days

The results of the experiments conducted in this manner are summarized in Table 6.5. In the case of the AMC and PuRL methods, the run time of one episode was measured, and the final development time is estimated based on this duration and the number of episodes required for convergence as stated in their respective papers. However, due to practical limitations, the actual execution of these tests was not performed, as it would have taken months utilizing the available NVIDIA Titan X GPU. Therefore, the total development times provided are approximations. Assuming that in these cases the number of required test runs was 90, equal to that of the proposed system, the estimated total development time for AMC is 385 days, while it is 279.4 days for PuRL. This amount of time is obviously unsuitable for efficient development. According to these estimations, the proposed method needs 16.72 × less amount of time to produce the final best performing pruned model than the PuRL, while 12.14 × less amount of time than the AMC.

Table 6.5: Speed of various RL-based automated pruning methods.

	Run time of one episode (t_{ep}) [s]	Episodes (N_{finep})	Agent's learning time ($t_{ep} * N_{finep}$) [h]	Full development time [days]
AMC	21	400[39]	2.33	**8.74**
AMC with dense reward	924 (21s*44=15,4m)	400	102.67	385
PuRL	4879,6 (110,9s*44=1,35h)	**55**[39]	74.5	279.4
proposed	**6.32**	700	**1.22**	23.023

6.3 Generalization Ability

This section presents the results and findings obtained during the generalization development. Before embarking on the discussion of the achieved results, I find it important to clarify some contextual aspects that will be applied from now on. When talking about the generalization ability of the pruning system across different datasets, the different datasets are actually the ones that were used to train the YOLOv4 object detectors which are then used to generate the training dataset for the SPN. For example, in the case of the KITTI dataset, the YOLOv4 model was trained on the KITTI dataset and was later used to generate the training dataset for the SPN. However, to avoid this opaque description, even though the SPN wasn't directly trained on the KITTI dataset, the aforementioned case will be referred to as "the SPN trained on the KITTI dataset".

6.3.1 Fine-Tuning The SPN On COCO

The first step in the investigation process of the generalization ability across different datasets is to evaluate the performance of the best α sequence predicted for a specific dataset on others datasets. To accomplish this, the YOLOv4 model trained on the COCO dataset is pruned using the α sequence that resulted in the best performing pruned model for the KITTI dataset. The results are summarized in Table 6.6. As expected, the ratio of removed parameters is approximately consistent with that observed in the case of the KITTI dataset. The performance degradation, on the other hand, is higher than in the former case; 24.6 % instead of 4.1 %. This outcome was also foreseen, and considered positive as the result is not complete degradation. In addition, the proposed α sequence outperforms the handcrafted α sequences in this scenario as well, which further proves the efficiency of the proposed solution. The bottom part of the table shows how the Torch Pruning library changes the pruning results. The integration of the library has an influence only on the proposed α sequence; it removes approximately an additional 12 % of parameters, while reducing the accuracy degradation by 5 %.

Table 6.6: Results of pruning the YOLOv4 model trained on the COCO dataset with the α sequence that resulted in the best performing pruned model for the KITTI dataset.

	mAP [%]	#params [M]	dmap [%]	sparsity [%]
YOLOv4	53.8	63.9	-	-
handcrafted1	38.9	38.1	27.6	40.4
handcrafted2	21.7	42.6	59.6	33.4
proposed	**40.5**	**32.3**	**24.7**	**49.4**
handcrafted1+TP	41.6	38.1	22.7	40.4
handcrafted2+TP	25.8	42.5	52	33.5
proposed+TP	**43.2**	**24.5**	**19.7**	**61.6**

Although the α sequence predicted for the KITTI dataset shows promising results on the COCO dataset, the desirable output is a sequence that performs with approximately the same precision. To accomplish this, the fine-tuning of the SPN is performed and the resulting SPN is used for re-training the RL agent on the new task. After generating roughly 5k images using the YOLOv4 model trained on the COCO dataset, the SPN model is fine-tuned on them using transfer learning. To show the importance of transfer

learning, the following cases are compared: when the SPN is trained from scratch on the COCO dataset and when the SPN model trained on the KITTI dataset is fine-tuned on the COCO dataset using transfer learning. Figure 6.7 shows the learning curves: the accuracy of *dmap* and *spars* with a 2 % margin and the mean absolute error for *dmap* and *spars*. When fine-tuning the old SPN model, the accuracy of *dmap* easily reaches 75 %, without overfitting. However, when attempting to train the SPN model from scratch on the COCO dataset, it struggles to converge, and on top of that, a high degree of overfitting is present.

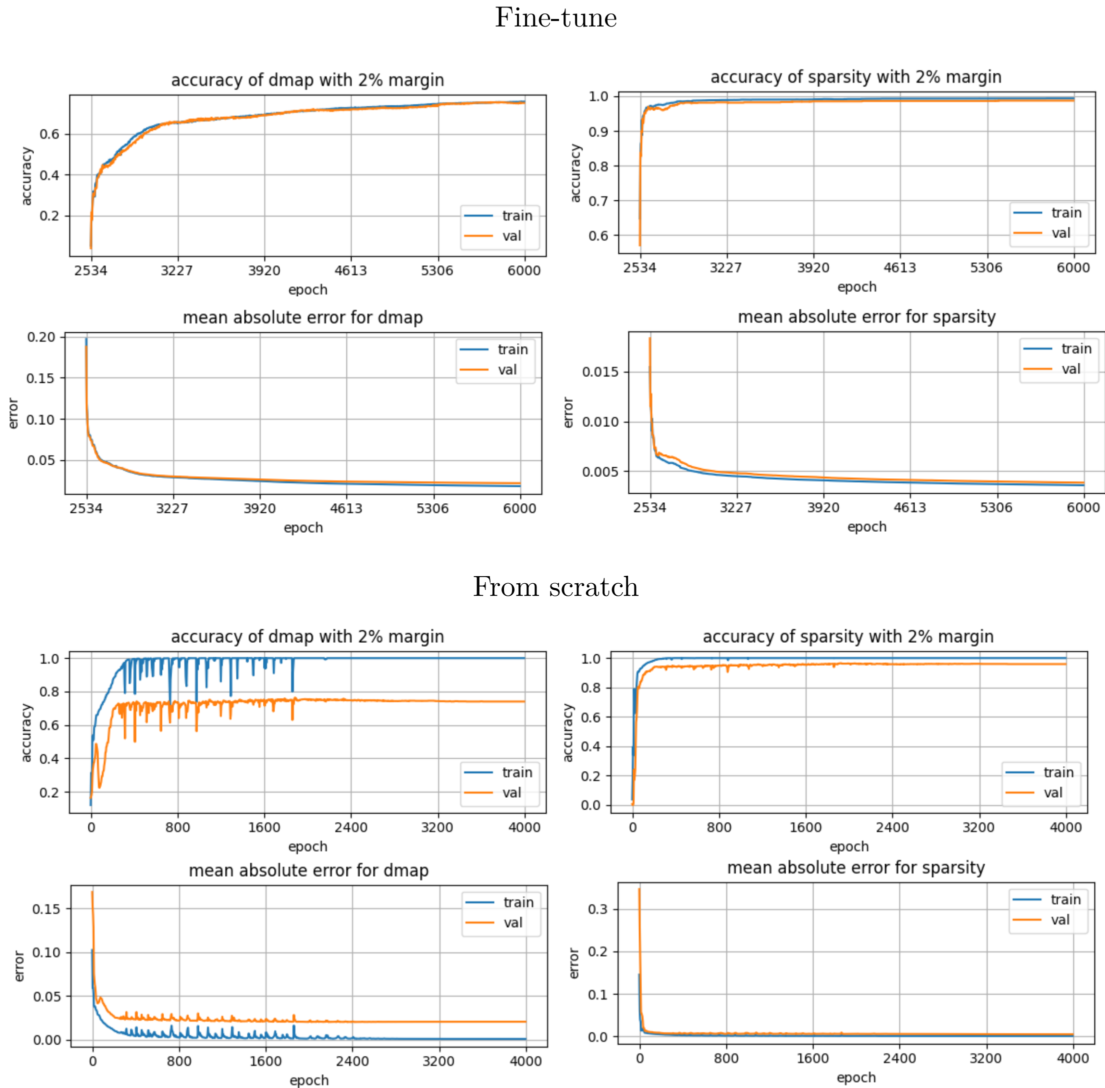

Figure 6.7: Comparison of two cases when training the SPN on the COCO dataset: fine-tuning the model trained on the KITTI dataset and training it from scratch. The graphs illustrate the accuracy of dmap and sparsity with a 2 % margin and the mean absolute error for *dmap* and *spars* for botch cases.

These results yet does not prove that using the fine-tuned SPN model, the RL agent will find the best performing α sequence for the COCO dataset. However, they do show that on the collected data, the fine-tuned SPN performs well, which is a crucial achievement towards the generalization. The results indicate that the SPN trained on the extensive

dataset generated with KITTI, serves as a great general start point for other datasets as well. Without the pre-trained SPN model, the small dataset generated on the COCO dataset would not be sufficient to effectively train a stable SPN model.

Table 6.7: Hyperparameters for the actor and critic networks during the RL agent's re-training with the SPN fine-tuned on COCO.

	episodes	batch size	optimizer	lr	ent. coeff.	device
actor	2000	4096	Adam	5e-3	5e-3	Tesla V100
critic	2000	4096	Adam	1e-2	-	Tesla V100

In spite of the SPN model that is reliable on the collected data, it is still possible that the collected samples are not representative enough. This premise is consolidated by the observed phenomenon when the fine-tuned SPN is placed back into the pipeline and the RL agent is re-trained using it. After a few episodes, it appears to be a progress in the RL agent's search for a suitable α sequence for the COCO dataset. Figure 6.8 depicts the predicted *dmap* and *spars* at the last prunable layer. According to the SPN's predictions, at episode 1600, the average of the *dmap* values is around 20 % with a corresponding 55 % *spars*. However, when the real pruning results are examined by pruning and validating the initial YOLOv4 model (Table 6.8), it turns out, that the SPN's predictions are incorrect, leading the agent into the wrong direction. The hyperparameter setting for the RL agent's training are summarized in Table 6.7, while the reward function is configurated the same way as described in Section 5.6.3.

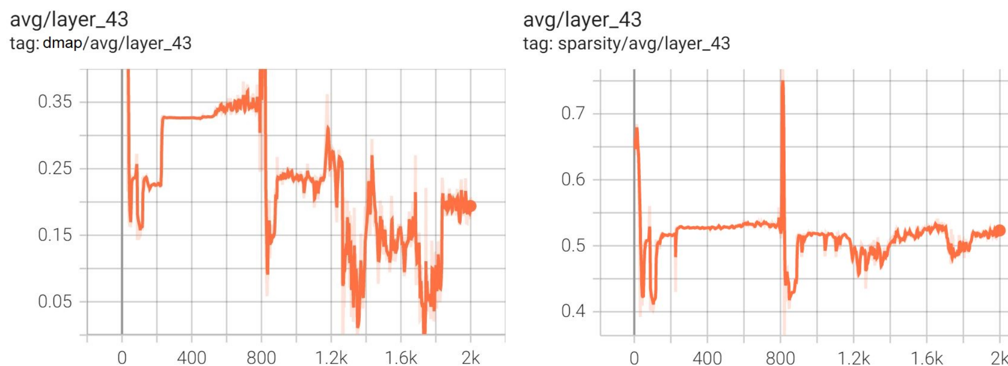

Figure 6.8: Predicted *dmap* and *spars* by the fine-tuned SPN at the last prunable layer, averaged across the batch.

Table 6.8: Comparison of the predicted and ground truth (gt.) values of *dmap* and *spars* for two randomly selected samples from the batch at episode 1600.

sample	pred. dmap [%]	gt. dmap [%]	pred. spars [%]	gt. spars [%]
rand1	32.2	99.9	52.1	62.3
rand2	33.6	99.9	53.9	61.4

6.3.2 Training The Transformer SPN

To generalize the pruning system across various DNN architectures, the initial step involves training the transformer-based SPN on the original task: pruning the YOLOv4 model trained on the KITTI dataset optimally. However, since the Torch Pruning library is employed to detect the number of prunable layers in the architecture, it alters the original task by discovering 107 prunable layers instead of the previous 44. Consequently, a new training dataset is required for the transformer-based SPN. The newly generated dataset consists of approximately 33k samples, following the same data collection process described in Section 5.5.2. The samples contain the model states with dimensions of $[N_{players} \times N_{features}]$, where $N_{players} = 107$, and the α sequences with a length of 107.

One dimension of the model state is given by default by the number of prunable layers, while the number of features observed need to be decided. Considering that the feature vectors serve as input embeddings for the SPN, it is reasonable to assume that that larger and more informative feature vectors would be beneficial. Therefore I choose a feature vector with a size of 5. The reason of choosing 5 instead of 6, as in the case of the actor network, is that the embedding's dimension should be the multiple of the number of heads, and one element is already taken by the corresponding α value. The learning curves in Figure 6.9 demonstrate that the SPN's performance is quite poor with this choice. When increasing the embedding's dimension by incorporating a (6, 32) linear layer, the problem seems to be resolved. Even though the accuracy of *dmap* with 2 % margin is only around 60 %, the mean absolute error of *dmap* is less than 4 %, which is acceptable (Figure 6.10). Similar to previous cases, the network does not encounter difficulties in learning the sparsity. The hyperparameters utilized for training a well-performing SPN are summarized in Table 6.9.

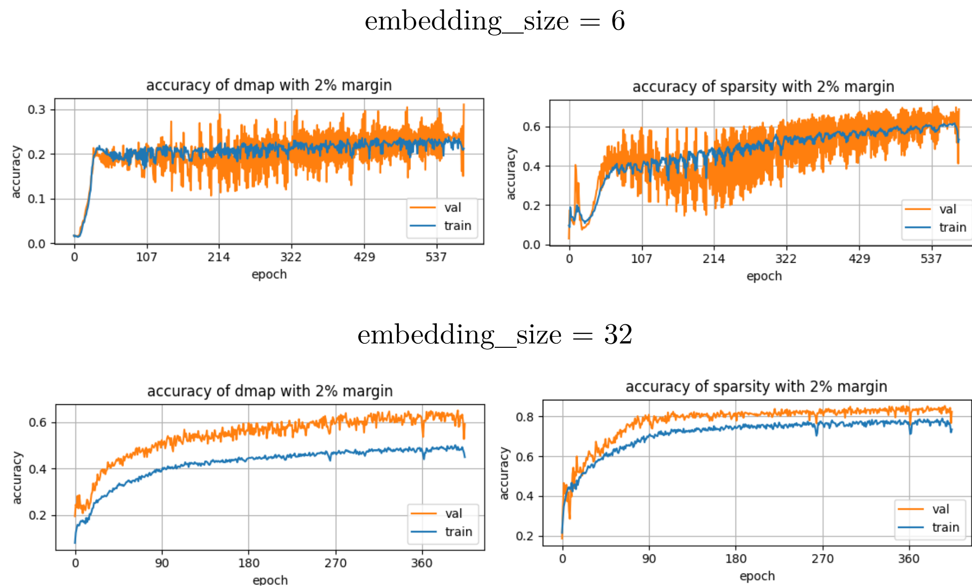

Figure 6.9: Comparison of the accuracy of *dmap* and *spars* with a 2 % margin metric of the SPN when using embeddings with size of 6 and 32.

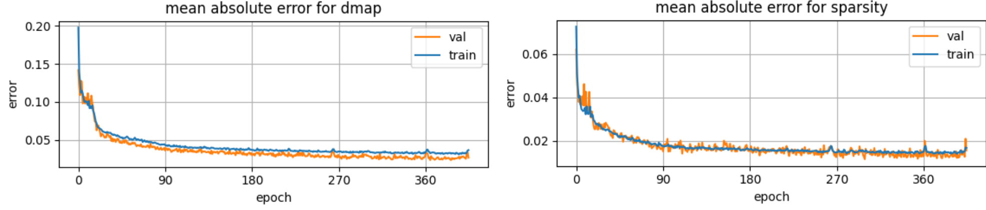

Figure 6.10: The mean absolute error for *dmap* and *spars* when training the SPN using embeddings with size of 32.

Table 6.9: Training hyperparameters for the transformer-based SPN.

epochs	batch size	loss function	optimizer	initial lr.	weight decay	device
2000	64	LogCoshLoss	Adam	8e-4	1e-05	Titan X

6.3.3 Training The RL Agent With The Transformer SPN

Once the transformer SPN is trained, the subsequent step involves training the RL agent to find the best suitable α sequence with length of 107 to acquire the best performing model. Unfortunately, this has not been accomplished successfully. This section presents the encountered obstacles during the development process and discusses the potential solutions to address them.

After extensively experimenting with hyperparameter adjustment without succeeding in achieving convergence towards any possible solution, the hyperparameter settings $hyp1$ in Table 6.10 eventually led the agent to suspect a promising policy. It starts to choose α = 0.1 values for the initial layers, but abruptly jumps to an α = 2.2 value at the 38th prunable layer. To further explore the alternative options at this point, I have retrained the agent form episode 1700 with a higher entropy coefficient of 1e-3. That way the agent selects $\alpha = 0.0$ values for the first few layers, but continues to jump to a high value at the 38th prunable layer (Figure 6.11).

This behavior raises concerns regarding the representativeness of the SPN's training data. Upon analyzing the data, I discovered that there are very few sparse sequences among the collected samples, where the majority of the α values are zero. Additionally, the maximum error metric stays relatively high during the SPN's training, indicating, that there are significant prediction errors regardless the 75 % accuracy with a 2 % margin. It must also be noted that the action space is quite large, and that accompanied with a state space that is more than double its previous size, makes the RL task overly complex. Several datasets have been generated to test each of these premises, summarized in Table 6.11.

Table 6.10: Hyperparameter settings used to lead the agent towards a promising policy with a larger state space (107 prunable layers).

	ep.	batch size	critic lr	actor lr	actor ent. coeff.	T_{dmap}	T_{spars}	coeff *dmap*	coeff *spars*
hyp1	5000	4096	1e-2	1e-3	1e-4	0.1	0.6	1.5	1
hyp2	5000	512	1e-3	1e-3	1e-2	0.2	0.6	1.5	1

The attempt to guide the agent towards the same promising policy observed before, using the *dset*2 dataset that includes additional sparse samples, was unsuccessful with *hyp*1

Table 6.11: Different datasets for training the SPN with a larger state space (107 prunable layers).

	#samples	description
dset1	32768	Original dataset.
dset2	43008	dset1 with additional sparse samples.
dset1_augm	40268	dset1 with augmentation.
dset3	23066	Dataset collected with a smaller action space ($\alpha \in [0.0, 0.5, 1.0, 1.5, 2.2]$)

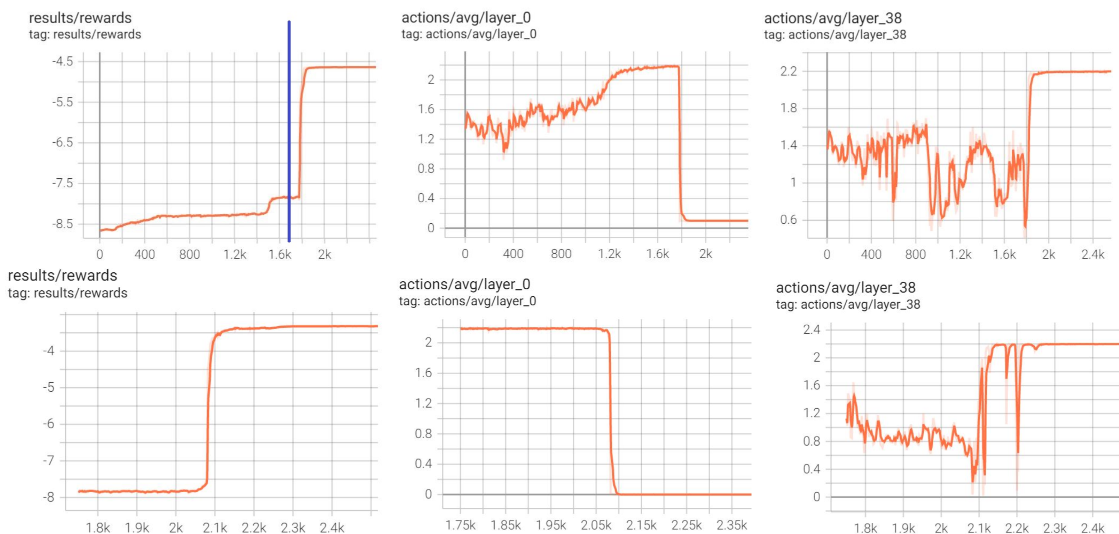

Figure 6.11: Mean reward and mean selected α actions for prunable layers 0 and 38 using the SPN trained on *dset*1 (top figures). Comparison after re-training the RL agent from episode 1700 with a higher entropy coefficient (bottom figures).

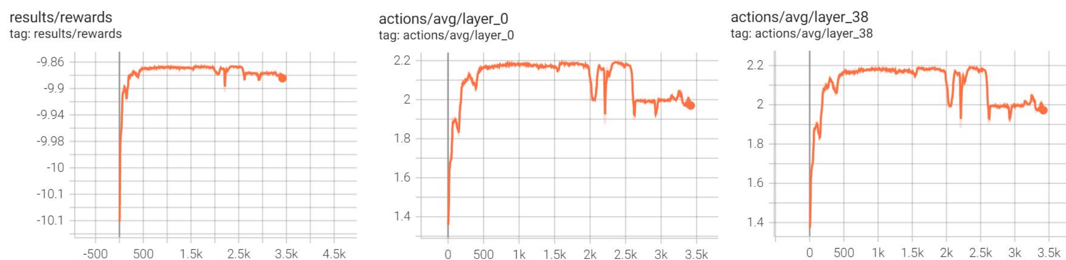

Figure 6.12: Mean reward and mean selected α actions for prunable layers 0 and 38 using the SPN trained on *dset*2.

and several other hyperparameter settings (Figure 6.12). A similar conclusion can be drawn from the *dset*1*_augm* dataset, which comprises augmented versions of samples from the original dataset *dset*1 predicted with significant error. A total of 7500 samples were generated by adding Gaussian noise to the problematic samples, with a mean of 0 and a standard deviation of 0.001.

To tackle the action and state space complexity issue, a new *dset*3 dataset was generated with a smaller action space; the α variable could only take on 5 values instead of the

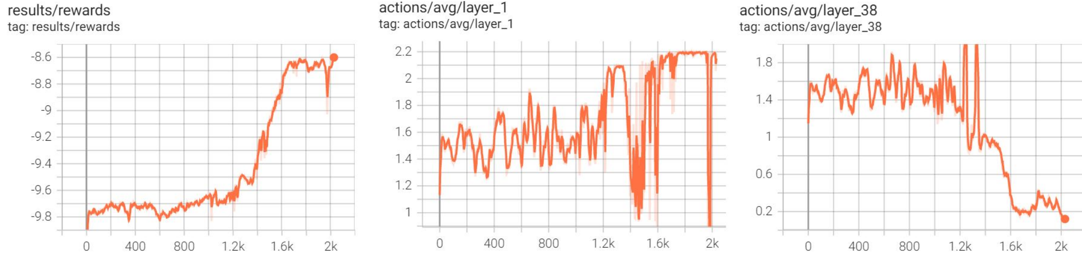

Figure 6.13: Mean reward and mean selected α actions for prunable layers 1 and 38 using the SPN trained on dset1_augm.

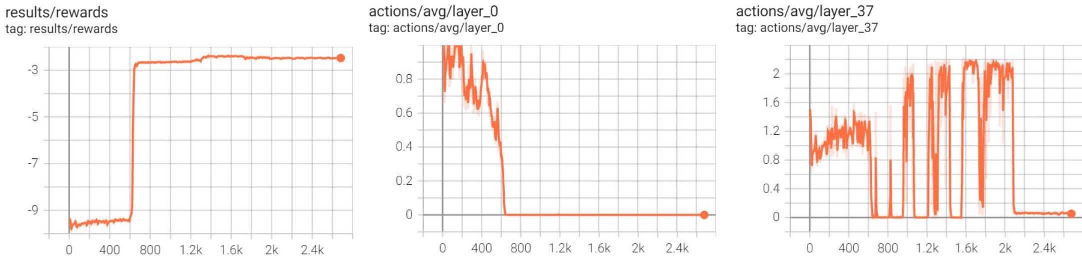

Figure 6.14: Mean reward and mean selected α actions for prunable layers 0 and 37 using the SPN trained on $dset3$.

previous 23: $\alpha \in [0.0,\ 0.5,\ 1.0,\ 1.5,\ 2.2]$. With this dataset, the hyperparameters again needed some adjustment to guide the agent towards a favorable policy. As shown in Figure 6.14, the agent starts to choose $\alpha = 0.0$ values for the initial layers, but the hesitation for jumping to a higher values around the 37th prunable layer appears here as well. While the smaller action space did not solve the problem entirely, it is important to note, that the agent discovered the favorable direction in a significantly shorter time; after 600 episodes instead of the previous 1700.

To conclude the lessons learned here, the larger state space definitely made the agent's training more challenging. Choosing a smaller action space can mitigate this for a certain extent. However, when it comes to finding the best solution, the SPN's incorrect predictions are the prominent obstacles.

6.4 Discussion

The core advantage of the proposed pruning system is reflected by the total development time required to find the best performing pruned model. The most time-consuming part of the development is the automated data generation for training the SPN. However, it only has to be done once, at the beginning of the development. After that, the agent's training can be performed rapidly, and valuable time can be devoted to hyperparameter-optimization, reward function design and further essential tasks. On the contrary, in the case of other SOTA approaches, the agent's training time also includes long validation, and in some cases, even fine-tuning steps, which severely limits the number of experiments. Another advantage of the proposed pruning system is that none of its parts requires high computation capacity and GPU memory, unlike, methods where fine-tuning is performed. Therefore, the field of RL-based automated pruning becomes accessible for researchers who do not own expensive, high-performance GPUs. Furthermore, the produced pruned

YOLOv4 model exhibits outstanding results, having 49 % less parameters than the initial model while experiencing only a minimal 4.1 % accuracy degradation. All things considered, the study has been highly successful in terms of effectively pruning the YOLOv4 object detector.

On the other hand, the development of a comprehensive general automated pruning system, that is able to perform effective pruning across various datasets and DNN architectures, remains an ongoing challenge. Nonetheless, some major achievements have been accomplished in this regard. Firstly, it was proven that the α sequence that works best on the KITTI dataset shows promising results on the COCO dataset as well; when pruning the YOLOv4 model trained on COCO with it, it caused only a moderate accuracy degradation. In addition, results indicate that fine-tuning the existing SPN is more effective compared to training it from scratch on the new task. Unfortunately, attempts to generate a more suitable sequence proved unsuccessful in the study. Secondly, notable achievements towards generalizing the pruning system across DNN architectures involve the integration of the Torch Pruning library, and the successful design of a transformer-based SPN. The former allows the automatic modification of several architectures, while the latter enables dynamic inputs, therefore varying number of prunable layers. Despite these accomplishments, the search for the ideal α sequence remains unresolved here as well.

Based on the experimental results, it seems like in both cases the problem originated from the SPN, more precisely, the lack of proper representation of its training data. The easiest solution to eschew this undesired behaviour is to generate more training data. However, aiming to improve the generalization ability, a more robust solution is preferred. One possible idea to address this issue is to introduce a novelty factor. It would measure the uniqueness of a sample, indicating the frequency of the similar samples available in the collected dataset. Based on its value, the pruning system would occasionally perform a real pruning and validation. The obtained results would be added to the dataset and the SPN would be eventually fine-tuned on it. This experiment, however, falls outside the scope of this study.

Chapter 7

Conclusion

During this study, according to the initial plans, I have designed and implemented a RL-based automated pruning system for the YOLOv4 object detector trained on the KITTI dataset. After a comprehensive literature research, I have found that the common drawback of the existing RL-based pruning solutions lies in the method used to determine the environmental variables; they are obtained by validating the pruned model during the RL agent's training, which significantly prolongs the training, therefore the overall development time. To address this issue, I introduce a novel component to the pruning system, called State Predictor Network. This network is responsible for simulating the environment and was trained before the RL agent, on automatically generated data. To the best of my knowledge, no such approach exists in the literature. The remaining components of the system, including the RL agent, the state and action space and the reward function, were designed based on ideas inspired by relevant publications.

The proposed pruning system is evaluated by the performance of the pruned YOLOv4 model and the total development time required to achieve this outcome. The valuation demonstrates exceptional results for the pruned model: it contains 49 % fewer parameters while experiencing only 4.1 % mAP degradation. When using the Torch Pruning library, the resulting pruned model contains 61 % fewer parameters with 6.4 % mAP loss. These results outperform the best handcrafted pruning method examined in the study, with 9 % more sparsity and 3.9 % less accuracy degradation. The pruned model's performance results cannot be directly compared to SOTA pruning results for YOLO-type detectors, as they were either trained on different datasets or were designed to different versions of the detector. Regarding the full development time, the proposed method surpasses both the AMC and PuRL SOTA LR-based pruning method's estimated overall development time. The estimation was performed by using their methods for determining the environmental variables in the proposed system, tested on a medium-performance Nvidia Titan X GPU. The proposed pruning system proves to be 16.72 $\times$ faster than the PuRL and 12.14 $\times$ faster than the AMC. This highlights the core advantage of the proposed pruning system.

After achieving outstanding results for the pruning of the YOLOv4 object detector, an extensive experimentation has been conducted to push the boundaries of the pruning system and generalize it across various datasets and DNN architectures. The α sequence predicted for the KITTI dataset shows promising results on the YOLOv4 model trained on the COCO dataset as well. When re-training the SPN for the new task, it proved to be more effective to fine-tune the existing SPN than training it from scratch. This conveys that the trained SPN serves as a solid basis for pruning various datasets. Significant achievements have also been made in generalizing the pruning system across DNN

architectures. Firstly, the integration of the Torch Pruning library allows the automatic modification of several architectures. Secondly, a transformer-based SPN has been designed and trained successfully, which enables dynamic inputs, therefore varying number of prunable layers. Despite the extensive efforts, finding the optimal results was not accomplished in any of the generalization tasks. Therefore, while substantial achievements have been made, the overall generalization remains a future challenge.

Future plans involve advancing the development towards the overall generalization. Since the prominent issue appears to stem from the SPN, one possible idea is to introduce a novelty factor, that measures the uniqueness of a sample. It would indicate the frequency of the similar samples present in the collected dataset. Based on its value, the pruning system would occasionally perform a real pruning and validation; the dataset would be supplemented by the obtained results, and the SPN would be eventually fine-tuned on it. Another future objective is to test the proposed system on widely used datasets and architectures, making it directly comparable to SOTA methods. Finally, I would like to experiment with varying desired sparsity-accuracy degradation ratio, and produce a solution with higher sparsity, even if it comes at the cost of higher performance loss.

Bibliography

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. http://www.deeplearningbook.org.

[2] Nielsen Michael A. *Neural Networks and Deep Learning.* Determination Press, 2015. http://neuralnetworksanddeeplearning.com/index.html.

[3] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[4] Yann LeCun, Koray Kavukcuoglu, and Clément Farabet. Convolutional networks and applications in vision. In *ISCAS 2010 - 2010 IEEE International Symposium on Circuits and Systems*, ISCAS 2010 - 2010 IEEE International Symposium on Circuits and Systems: Nano-Bio Circuit Fabrics and Systems, pages 253–256, 2010. 2010 IEEE International Symposium on Circuits and Systems: Nano-Bio Circuit Fabrics and Systems, ISCAS 2010 ; Conference date: 30-05-2010 Through 02-06-2010.

[5] Tom M Mitchell et al. Machine learning. 1997.

[6] Richard S. Sutton. Two problems with backpropagation and other steepest-descent learning procedures for networks. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society.* Hillsdale, NJ: Erlbaum, 1986.

[7] Ross Girshick. Fast r-cnn. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1440–1448, 2015.

[8] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(6):1137–1149, 2017.

[9] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788, 2016.

[10] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6517–6525, 2017.

[11] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *ArXiv*, abs/1804.02767, 2018.

[12] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *ArXiv*, abs/2004.10934, 2020.

[13] W. Liu, Dragomir Anguelov, D. Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. Ssd: Single shot multibox detector. In *European Conference on Computer Vision*, 2015.

[14] Zhanchao Huang and Jianlin Wang. Dc-spp-yolo: Dense connection and spatial pyramid pooling based yolo for object detection. *Inf. Sci.*, 522:241–258, 2019.

[15] Yolov5. `https://ultralytics.com/yolov5`, Last accessed on 2023-05-28.

[16] Chuyi Li, Lulu Li, Hongliang Jiang, Kaiheng Weng, Yifei Geng, Liang Li, Zaidan Ke, Qingyuan Li, Meng Cheng, Weiqiang Nie, Yiduo Li, Bo Zhang, Yufei Liang, Linyuan Zhou, Xiaoming Xu, Xiangxiang Chu, Xiaoming Wei, and Xiaolin Wei. Yolov6: A single-stage object detection framework for industrial applications, 2022.

[17] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors, 2022.

[18] Yolov8. `https://ultralytics.com/yolov8`, Last accessed on 2023-05-28.

[19] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning?, 2020.

[20] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *ArXiv*, abs/1608.08710, 2016.

[21] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. MIT Press, 2nd edition, 2018.

[22] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[23] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, 2016.

[24] Martin A. Riedmiller. Neural fitted q iteration - first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, 2005.

[25] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *ArXiv*, abs/1707.06347, 2017.

[26] John Schulman, Sergey Levine, P. Abbeel, Michael I. Jordan, and Philipp Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, 2015.

[27] Robin M. Schmidt. Recurrent neural networks (rnns): A gentle introduction and overview. *ArXiv*, abs/1912.05911, 2019.

[28] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6:107–116, 04 1998.

[29] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997.

[30] Weijiang Feng, Naiyang Guan, Yuan Li, Xiang Zhang, and Zhigang Luo. Audio visual speech recognition with multimodal recurrent neural networks. pages 681–688, 05 2017.

[31] Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, 2017.

[32] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[33] Guiying Li, Chao Qian, Chunhui Jiang, Xiaofen Lu, and Ke Tang. Optimization based layer-wise magnitude-based pruning for dnn compression. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 2383–2389. International Joint Conferences on Artificial Intelligence Organization, 7 2018.

[34] Xiangyu Zhang, Jianhua Zou, Kaiming He, and Jian Sun. Accelerating very deep convolutional networks for classification and detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(10):1943–1955, 2016.

[35] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, page 1135–1143, Cambridge, MA, USA, 2015. MIT Press.

[36] Yuxuan Cai, Hongjia Li, Geng Yuan, Wei Niu, Yanyu Li, Xulong Tang, Bin Ren, and Yanzhi Wang. Yolobile: Real-time object detection on mobile devices via compression-compilation co-design. In *Association for the Advancement of Artificial Intelligence*, 2021.

[37] Wei Yan, Ting Liu, and Yuzhuo Fu. Yolo-tight: an efficient dynamic compression method for yolo object detection networks. *13th International Conference on Machine Learning and Computing*, 2021.

[38] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *European Conference on Computer Vision*, 2018.

[39] Manas Gupta, Siddharth Aravindan, Aleksandra Kalisz, Vijay Ramaseshan Chandrasekhar, and Lin Jie. Learning to prune deep neural networks via reinforcement learning. *ArXiv*, abs/2007.04756, 2020.

[40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[41] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. Scaled-YOLOv4: Scaling cross stage partial network. In *Proceedings of the IEEE/CVF Conference on*

Computer Vision and Pattern Recognition (CVPR), pages 13029–13038, June 2021. https://github.com/WongKinYiu/ScaledYOLOv4, Last accessed on 2023-05-26.

[42] Darknet yolo files. https://sourceforge.net/projects/darknet-yolo.mirror/files/darknet_yolo_v4_pre/, Last accessed on 2023-05-26.

[43] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[44] Fengji Yi, Wenlong Fu, and Huan Liang. Model-based reinforcement learning: A survey. 2018.

[45] Gongfan Fang, Xinyin Ma, Mingli Song, Michael Bi Mi, and Xinchao Wang. Depgraph: Towards any structural pruning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16091–16101, 2023.