Université de Montréal

Optimizing ANN Architecture using Mixed-Integer Programming

par

Mostafa ElAraby

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Informatique

Août 31, 2020

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

Optimizing ANN Architecture using Mixed-Integer Programming

présenté par

Mostafa ElAraby

a été évalué par un jury composé des personnes suivantes :

Fabian Bastin

(président-rapporteur)

Margarida Carvalho

(directrice de recherche)

Guy Wolf

(codirecteur)

Michel Gendreau

(membre du jury)

Résumé

Les réseaux sur-paramétrés, où le nombre de paramètres dépasse le nombre de données, se généralisent bien sur diverses tâches. Cependant, les grands réseaux sont coûteux en termes d'entraînement et de temps d'inférence. De plus, l'hypothèse du billet de loterie indique qu'un sous-réseau d'un réseau initialisé de façon aléatoire peut atteindre une perte marginale après l'entrainement sur une tâche spécifique par rapport au réseau de référence. Par conséquent, il est nécessaire d'optimiser le temps d'inférence et d'entrainement, ce qui est possible pour des architectures neurales plus compactes.

Nous introduisons une nouvelle approche "Optimizing ANN Architectures using Mixed-Integer Programming" (OAMIP) pour trouver ces sous-réseaux en identifiant les neurones importants et en supprimant les neurones non importants, ce qui permet d'accélérer le temps d'inférence. L'approche OAMIP proposée fait appel à un programme mixte en nombres entiers (MIP) pour attribuer des scores d'importance à chaque neurone dans les architectures de modèles profonds. Notre MIP est guidé par l'impact sur la principale tâche d'apprentissage du réseau en élaguant simultanément les neurones. En définissant soigneusement la fonction objective du MIP, le solveur aura une tendance à minimiser le nombre de neurones, à limiter le réseau aux neurones critiques, c'est-à-dire avec un score d'importance élevé, qui doivent être conservés pour maintenir la précision globale du réseau neuronal formé. De plus, la formulation proposée généralise l'hypothèse des billets de loterie récemment envisagée en identifiant de multiples sous-réseaux "chanceux". Cela permet d'obtenir des architectures optimisées qui non seulement fonctionnent bien sur un seul ensemble de données, mais aussi se généralisent sur des différents ensembles de données lors du recyclage des poids des réseaux. Enfin, nous présentons une implémentation évolutive de notre méthode en découplant les scores d'importance entre les couches à l'aide de réseaux auxiliaires et entre les différentes classes. Nous démontrons la capacité de notre formulation à élaguer les réseaux de neurones avec une perte marginale de précision et de généralisabilité sur des ensembles de données et des architectures populaires.

Mots clés : Apprentissage profond, Élagage des réseaux neuronaux, Programmation mixte, Classement des neurones, Optimisation combinatoire, Optimisation de l'architecture.

Abstract

Over-parameterized networks, where the number of parameters surpass the number of training samples, generalize well on various tasks. However, large networks are computationally expensive in terms of the training and inference time. Furthermore, the lottery ticket hypothesis states that a subnetwork of a randomly initialized network can achieve marginal loss after training on a specific task compared to the original network. Therefore, there is a need to optimize the inference and training time, and a potential for more compact neural architectures.

We introduce a novel approach "Optimizing ANN Architectures using Mixed-Integer Programming" (OAMIP) to find these subnetworks by identifying critical neurons and removing non-critical ones, resulting in a faster inference time. The proposed OAMIP utilizes a Mixed-Integer Program (MIP) for assigning importance scores to each neuron in deep neural network architectures. Our MIP is guided by the impact on the main learning task of the network when simultaneously pruning subsets of neurons. In concrete, the optimization of the objective function drives the solver to minimize the number of neurons, to limit the network to critical neurons, i.e., with high importance score, that need to be kept for maintaining the overall accuracy of the trained neural network. Further, the proposed formulation generalizes the recently considered lottery ticket hypothesis by identifying multiple "lucky" subnetworks, resulting in optimized architectures, that not only perform well on a single dataset, but also generalize across multiple ones upon retraining of network weights. Finally, we present a scalable implementation of our method by decoupling the importance scores across layers using auxiliary networks and across different classes. We demonstrate the ability of OAMIP to prune neural networks with marginal loss in accuracy and generalizability on popular datasets and architectures.

Keywords : Deep learning, Pruning Neural Networks, Mixed-Integer Programming, Neurons Ranking, Combinatorial Optimization, Architecture Optimization.

Contents

List of tables

List of figures

List of acronyms and abbreviations

MIP	Mixed-Integer Program
LP	Linear Program
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
SNIP	Single-shot Network Pruning based on Connection Sensitivity
OAMIP	Optimizing ANN Architectures using Mixed-Integer Programming

Acknowledgements

I have received the help and encouragement of numerous people to pursue my master's degree after years of industrial work. They kept encouraging me throughout the admission process and during the course of my studies.

I would like to thank my supervisor Professor Margarida Carvalho and co-supervisor Professor Guy Wolf for their mentorship, encouragement and academic guidance. First and foremost, I am grateful to both for trusting my abilities and giving me the freedom to explore and experiment my ideas. Feeling the trust and confidence from one's advisors would boost any graduate student's creativity and productivity. I have to acknowledge their mentorship not just in the thesis or the choice of my courses, but also concrete and mature advice that guided my future in academia.

I started my first research steps at Microsoft Research Lab where I gained the experience needed to pursue my graduate studies. At Microsoft, I was lucky to work with talented colleagues and friends, improving my research capabilities. I had the honor to work with talented calibers people like my manager and mentor Ahmed Tawfik, Hani Hassan and Mohamed Afifi who gave me guidance and mentorship during my work.

I would like also to acknowledge the support and encouragement of my previous manager Wessam Gad El-Rab at Tensorgraph. He selflessly motivated me to pursue my graduate studies and gave me support and guidance during the first semester of my master's degree. I am forever grateful for his advice and mentorship.

None of this would have been possible without the support and continuous encouragement that my parents, my sister, and all my family have given me throughout the process of pursuing my graduate studies.

I would like to thank IVADO (l'Institut de Valorisation des Données), FRQ-IVADO Research Chair in Data Science for Combinatorial Game Theory, and NSERC grant 2019-04557 for funding this thesis.

Introduction

Deep learning has proven its ability to solve complex tasks and to achieve state-of-the-art results in various domains such as image classification, speech recognition, machine translation, robotics and control (Bengio et al., 2017; LeCun et al., 2015). Over-parameterized deep neural models with more parameters than the training samples can be used to achieve state-of-the-art results on various tasks (Zhang et al., 2016; Neyshabur et al., 2018). However, the large number of parameters comes at the expense of computational cost in terms of memory footprint, training time and inference time on resource-limited Internet of things (IOT) devices (Lane et al., 2015; Li et al., 2018).

In this context, pruning neurons from an over-parameterized neural model has been an active research area (Blalock et al., 2020; Cheng et al., 2017a). This remains a challenging open problem whose solution has the potential to increase the computational efficiency and to uncover potential subnetworks that can be trained effectively. Neural Network pruning techniques (LeCun et al., 1990; Hassibi et al., 1993; Han et al., 2015; Srinivas and Babu, 2015; Dong et al., 2017; Zeng and Urtasun, 2018; Lee et al., 2018; Wang et al., 2019; Salama et al., 2019; Serra et al., 2020) have been introduced to sparsify the models without loss of accuracy.

Nevertheless, most existing work focuses on identifying redundant parameters and non-critical neurons to achieve a lossless sparsification of the neural model. The typical sparsification procedure starts with training a neural model, then computing parameters' importance. After computing the parameters' importance, some existing parameters are pruned using certain criteria, and the neural model is then fine-tuned to regain its lost accuracy. Furthermore, existing pruning and ranking procedures are computationally expensive, requiring iterations of fine-tuning on the sparsified model. Moreover, no experiments have been conducted to check the generalization of sparsified models across different datasets.

We remark that sparse neuron connectivity is often used by modern network architectures, and perhaps, most notably, in convolutional layers used in image processing. Indeed, the limited size of the parameter space in such cases increases the effectiveness of network training and enables the learning of meaningful semantic features from the input images (Goodfellow et al., 2016). Inspired by the benefits of sparsity in such architecture designs, we aim to leverage the neuron sparsity achieved by our framework, Optimizing ANN Architectures using Mixed-Integer Programming (OAMIP), to obtain optimized neural architectures that can generalize well across different datasets. For this purpose, we create a sparse subnetwork by optimizing on one dataset and then training the same architecture, i.e., masked, on another dataset. Our results indicate a promising direction of future research into the utilization of combinatorial optimization for effective automatic architecture tuning to augment handcrafted network architecture design.

Contribution

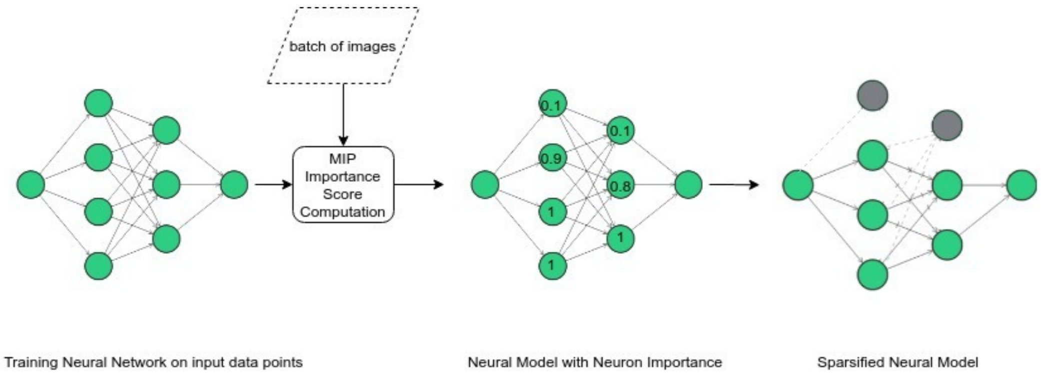

batch of images
MIP Importance Score Computation
0.1
0.9
1
1
0.1
0.8
1
Training Neural Network on input data points
Neural Model with Neuron Importance
Sparsified Neural Model

Fig. I.1. The generic flow of OAMIP used to remove neurons having an importance score less than a specific threshold.

In OAMIP, illustrated in Figure I.1, we formalize the notation of *neuron importance* as a score between 0 and 1 for each neuron in a neural network and the associated dataset. The neuron importance score reflects how much activity decrease can be inflicted on it, while controlling the loss on the neural network model accuracy. Concretely, we propose a Mixed-Integer Programming formulation (MIP) that allows the computation of importance score for each fully connected/convolutional neuron and takes into account the error propagation between different layers. The motivation to use such approach comes from the existence of powerful techniques to solve MIPs efficiently in practice, and consequently, allows the scalability of this procedure to large ReLU[1] activated neural models. In addition, we extend the proposed formulation to support convolutional layers computed as matrices multiplication using Toeplitz format (Gray, 2000), with an importance score associated with each feature map (Molchanov et al., 2016).

Once neuron importance scores have been determined, a threshold is established to enable the identification of non-critical neurons and their consequent removal from the neural

[1]Rectified linear units that applies an activation function between layers allowing only positive logits $f(x) = x^+ = \max(0,x)$.

network. Since it is intractable to compute neuron scores using full datasets, in our experiments, we approximate their values by using subsets. In fact, we provide empirical evidence that our pruning process results in a marginal loss of accuracy (without fine-tuning) when the scores are approximated by a small balanced subset of data points, or even by parallelizing the scores' computation per class and averaging the obtained values. Furthermore, we enhance our approach such that the importance scores computation is also efficient for very deep neural models, like VGG-16 (Simonyan and Zisserman, 2014). This stresses its scalability to datasets with many classes and large deep networks. In addition, we show that the computed neuron importance scores from a specific dataset generalize to other datasets when the pruned subnetwork is retrained using the same initialization.

The evaluation of our OAMIP includes computing the importance score of each neuron in the input of the neural network through a MIP. Afterwards, we create different subnetworks, each with different pruning methods. We compare subnetwork performance between pruning non-critical neurons, randomly pruned neurons and critical neurons with the reference neural model. The number of pruned neurons at each layer is the same as the number of pruned neurons in our framework, so as to perform a fair comparison. Another evaluation concerns the existence of subnetworks with the same initialization (lottery tickets) that generalize on different datasets. First, we compute neurons ranking of a neural model on dataset D_1. Then, we take the same neural model initialization and masked neurons computed on dataset D_1 to retrain on dataset D_2. These experiments show that the calculated importance score is giving the right ranking for the neurons and that it can generalize across different datasets.

Outline

This thesis is organized as follows. Chapter 1 provides an overview on the foundations of deep learning and Mixed-Integer Programming (MIP). In this chapter, we also review relevant literature on neural networks sparsification, and the use of mixed integer programming to model them. Chapter 2 provides background on the formulation of Artificial Neural Networks (ANN) as constraints of a mixed integer program, along with the motivation behind this representation. We also introduce the computation of variable bounds for the MIP and discuss its objective function. Chapter 3 introduces the neuron importance score in an ANN, and its incorporation in the mathematical programming model. In this chapter, we also discuss the objective function that optimizes the sparsification and balances accuracy of the ANN. Chapter 4 provides the experimental settings and computational experiments to validate our proposed approach. Chapter 5 focuses on scaling up our method by decoupling the computation of neuron importance score per layer, useful for large neural networks. Furthermore, in the same chapter we introduce the decoupling of neuron importance score computation per class for datasets with numerous classes. In Chapter 6, we summarize our contribution and possible future research directions, directly benefiting from the work developed.

Working Paper

Chapters 3, 4 and 5 are based on the working paper (ElAraby et al., 2020). I (the first author) contributed in all of its scientific stages, namely, literature review, formulation of the MIP in Chapter 3, design of OAMIP, its implementation, and analysis in Chapter 4, and speed enhancements to tackle large models in Chapter 5, and paper writing. The standard rule for the order of authors in the machine learning field follows the level of contribution. The other co-authors, Margarida Carvalho and Guy Wolf, contributed equally. Our proposed framework was reviewed and accepted for poster presentation at Montreal AI symposium 2020. The code is publicly available at `https://github.com/chair-dsgt/mip-for-ann` to reproduce the major results of this work.

Chapter 1

Background

In this chapter, we provide the necessary background about deep learning models and mixed-integer programming (MIP). For the basics and further details of machine learning, neural networks and backpropagation, we refer the reader to (Goodfellow et al., 2016), and for an introduction on integer programming we refer the reader to (Nemhauser and Wolsey, 1988).

1.1. Deep Learning

The first deep multilayer perceptron was published by (Ivakhnenko and Lapa, 1967). Deep learning was introduced to the machine learning community by (Dechter, 1986). The algorithm behind the training of deep neural networks, the standard backpropagation algorithm, was first proposed by (Werbos, 1982; Rumelhart et al., 1986), and then applied to a deep neural network with the purpose of recognizing handwritten zip codes on mails (LeCun et al., 1989). Next, we will explain the concept of deep learning and different model types that will be represented by our approach using a MIP.

Deep learning refers to the process of training deep neural models to learn a set of complex hierarchical features. We start by introducing deep learning from the simplest statistical tool, the linear regression (Gauss, 1809; Plackett, 1972). In linear regression, we are given a training set consisting of n input data points $\{x_1, \dots, x_n\}$ having a target to be predicted $\{y_1, \dots, y_n\}$. We assume the existence of a linear function, mapping input data point $x_i \in \mathbb{R}^Q$ to the target $y_i \in \mathbb{R}^D$ with some possible noise in the training data. In that case, the linear regression fits a set of parameters to the observed data namely, the weight matrix $\boldsymbol{W}$ with dimension Q by D and bias vector $b \in \mathbb{R}^D$, as shown for a single dimension in Figure 1.1. In this way, the linear function mapping each input observation to the output targets is $f(x) = \boldsymbol{W}x + b$. The goal is to learn these parameters such that the mean squared error over the input training data points is minimized:

$$\min_{\boldsymbol{W},b} \frac{1}{n} \sum_i (y_i - (\boldsymbol{W}x_i + b))^2. \tag{1.1.1}$$

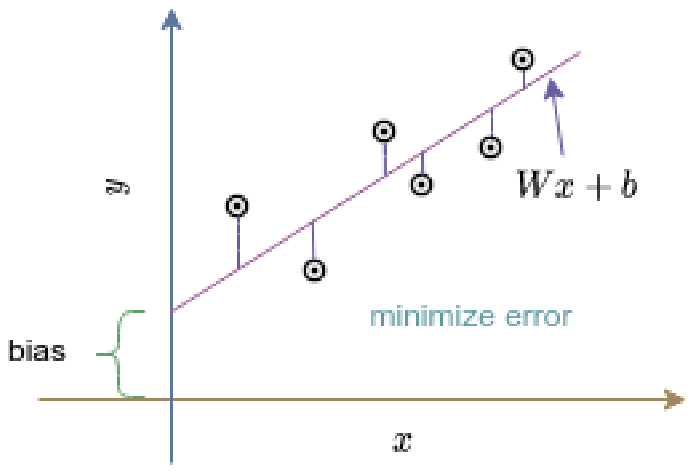


Fig. 1.1. Linear regression in a 1D data points, showing the process of fitting a line.

The relation between input data points and labels is restricted to be a linear relation in Equation (1.1.1). However, in most of the practical cases, the relation is non-linear. We can represent this non-linearity (Bishop, 2006) by feeding input data to a fixed scalar transformation function and composing it with a feature vector having a linear relation with the target. Finally, we fit the linear regression parameters on the created feature vector.

The simplest deep learning architecture consists of a set of hierarchical linear layers. Each linear layer is a linear regression. However, to represent non-linearity, instead of using a transformation function on the input data points, a non-linear transformation function is added between layers to represent non-linear complex features.

In what follows, we introduce the formal notation for a set of simple neural network models. Since our experiments are conducted on images, we will show the extension of these formulations to process image data using hand crafted features representing domain knowledge of connections' design.

Feed-forward Artificial Neural Networks (ANNs). Also called multi-layer perceptrons (MLPs) (Rumelhart et al., 1985), are the first and simplest form of neural networks devised with directed acyclic connections between its nodes as shown in Figure 1.2. For simplicity, we will introduce the notation for a single ANN layer whose objective is to approximate some function f^*. A feed forward network defines a mapping $f(x, \theta)$ to approximate f^* and learns the parameters θ through the process (Goodfellow et al., 2016). The parameters of the linear transformation of the first hidden layer is given by the matrix $\boldsymbol{W^{(1)}}$, referred to as the weight matrix, and $b^{(1)}$ the translation, referred to as the bias. These parameters are used to linearly transform input data points to an intermediate set of features, and then an element wise non-linear function $\sigma(\cdot)$ is applied. Rectified Linear Units (ReLU) are widely used as non-linear function: $ReLU(x) = max(x, 0)$. We then apply a linear transformation to map the hidden's layer output to our target y using weight matrix $\boldsymbol{W^{(2)}}$ and bias $b^{(2)}$. In that case, $\theta = (\boldsymbol{W^{(1)}}, b^{(1)}, \boldsymbol{W^{(2)}}, b^{(2)})$. The network's output is denoted as $\hat{y}$,

$$\hat{y} = \sigma(\boldsymbol{W^{(1)}}x + b^{(1)})\,\boldsymbol{W^{(2)}} + b^{(2)}. \tag{1.1.2}$$

$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(\hat{y}_i, y_i). \tag{1.1.3}$$

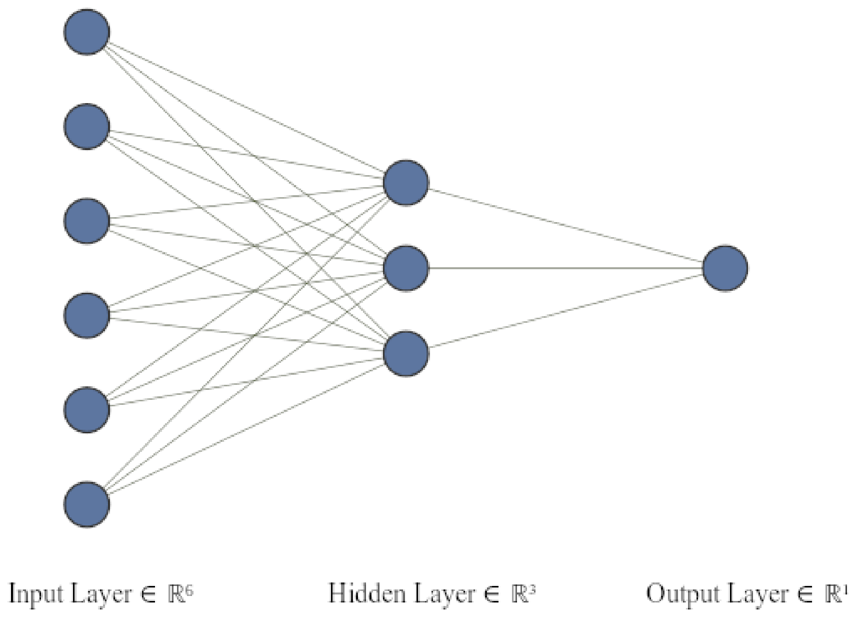


Fig. 1.2. Simple fully connected artificial neural network.

Equation (1.1.3) represents the loss function for a dataset having n training examples. An objective function l is used to determine the distance between the predicted and true labels. In this setting, learning is performed using empirical risk minimization which involves finding the optimal parameters $\theta^* \in \arg\min_\theta L(\theta)$. The optimal parameters would hopefully result in a model that can generalize its knowledge to unseen test data.

Theorem 1.1.1. *Universal approximation theorem (Funahashi, 1989). A single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units.*

Hence, in order to increase the capacity[1] of the network, we can extend this simple network to have multiple layers and more hidden units per layer as shown in Theorem 1.1.1.

[1]It can be defined as the complexity of the functions that the network can represent.

For classification tasks, the output of the network needs to be normalized to the range $[0, 1]$, representing the predicted class probability. Consequently, a softmax activation function is applied on the output logit of the network.

$$softmax(a) = \left[\frac{\exp(a_1)}{\sum_c \exp(a_c)}, \ldots, \frac{\exp(a_c)}{\sum_c \exp(a_c)} \right]^T. \tag{1.1.4}$$

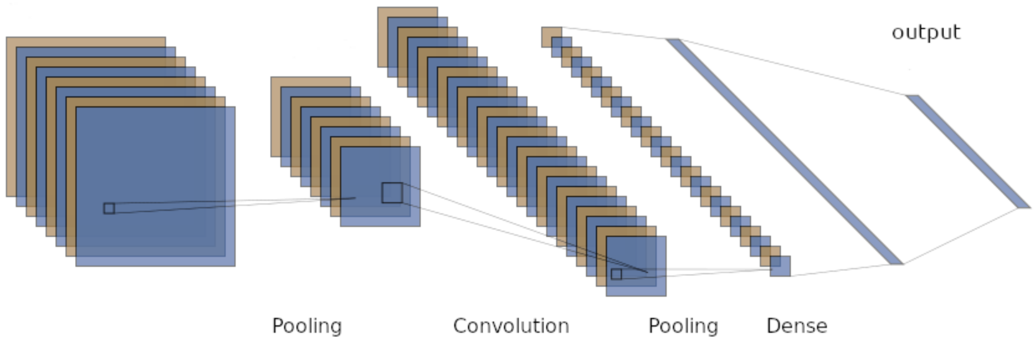


Fig. 1.3. Simple LeNet-5 convolutional neural network architecture consisting of convolutional transformation, pooling and fully connected layers.

Convolutional neural networks (CNNs). CNNs (LeCun et al., 1989) are popular neural networks for image processing, which make use of the 2D topology of images to process input images with a few shared parameters. The typical CNN consists of recursive application of convolutional layers with a non-linear activation function, followed by a pooling operation to reduce its spatial dimensionality, and then a feed-forward layer to convert transformed features to the target as in Figure 1.3.

Convolutional layers are the core building block of convolutional model architectures. A convolutional layer consists of a set of kernels exploiting the 2D topology of images. Each kernel is a tensor having a height, a width and a depth. This latter is the number of input channels. During a forward pass, we slide (more specifically, convolve) each kernel across the width and height of the input image and compute the dot product between the kernel and the input image at any position. The output of each kernel is the sum of the convolution at each input channel dimension (depth) with its corresponding dimension in that kernel. Each neuron is locally connected to a patch of the input image, which is called the receptive field (kernel dimension), see Figure 1.4. The kernel is also known as a feature map, since it is a mapping of where a specific feature occurs in an image.

We can control the size of the output data from the convolution using a set of hyperparameters:

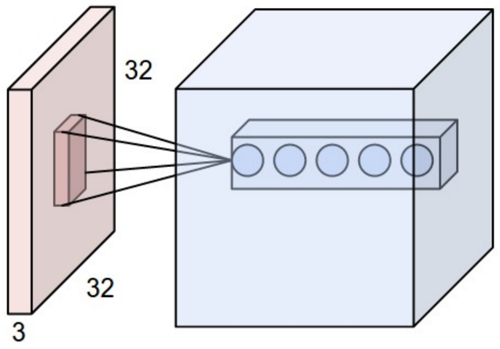


Fig. 1.4. An example of an input image in red, and an example of a set of neurons in a kernel in blue. This illustration is showing the local connectivity between neurons of a convolutional layer. This local connection allows dealing with high dimensional input images.

(1) The depth, which is the number of kernels used in this layer. After convolving an input image, the number of channels is the same as the number of kernels used in this layer.
(2) The stride, which is the amount of pixels used to slide the kernel. If we set the stride to 1, we slide the kernel by 1 pixel at a time. When the stride is 2, the kernel would slide with 2 pixels at a time resulting in a spatially smaller output.
(3) The padding, which is the amount of zeros padded to each side of the input image, most commonly used to preserve the size of the input image after applying the convolutions.

The convolutional operation can be equivalently implemented as a large simple matrix multiplication with the downside of taking more memory.

Pooling layers on multi-input units are used to reduce spatial representation of input images by applying an arithmetic operation on the output of each kernel of the previous convolutional layer. Average Pooling layer applies the average operation on the output of each kernel. Max Pooling takes the maximum value of each kernel of the previous layer.

Accordingly, hand crafted neural connections that use domain knowledge are shown to give good results while reducing the computational costs, as shown in convolutional layers (LeCun et al., 1989).

1.2. Mixed-Integer Programming

Linear-programming (LP) was first initiated by the work of (Dantzig, 1948). Dantzig (1948) introduced the simplex algorithm broadly used nowadays to solve linear programs. Later, the branch-and-bound algorithm was introduced by (Land and Doig, 1960), and used in LP/90/94 commercial IBM solver released 1963 on IBM 7094 machine with the ability to handle up to 1024 constraints. Furthermore, LP/90/94 has the ability to handle mixed-integer problems initiated by (Beale and Small, 1965) using the algorithm from (Dakin, 1965) along with dichotomous branching (Land, 1960). We refer the reader to (Bixby, 2012) for a detailed history of LPs and MIPs.

Integer programs are combinatorial optimization problems restricted to discrete decision variables, linear constraints and linear objective function. Integer programs can be expressed using the canonical form

$$\begin{aligned} \text{Maximize} \quad & \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \text{subject to} \quad & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \\ & \mathbf{x} \in \mathbb{Z}^n. \end{aligned} \tag{1.2.1}$$

These problems are NP-hard, even when the variables are restricted to binary values (Garey and Johnson, 1979). The difficulty comes from ensuring integer solutions, and thus, the impossibility of using gradient methods. When continuous variables are included, they are designated by mixed-integer programming problems.

Mixed-integer programs are solved using linear programming algorithms embedded in a branch-and-bound scheme. Basic branch-and-bound starts by removing the integrality restrictions in the MIP variables, resulting in the so-called relaxed linear program LP. We then solve the relaxed LP. If the solution satisfies the integrality constraints then it is optimal for the MIP. If not, the method branches on a variable that did not satisfy the integrality constraint, i.e., two new linear programs are obtained by imposing that the variable is greater than or equal the ceiling of its fractional value, and another one where it must be less or equal to its floor value. We then proceed by applying the same branching procedure. Note that branching on a variable value, creates a search tree and the nodes of the tree are the LPs generated by the search procedure as in Figure 1.5. The solver iterates over the nodes of the search tree until the solution of the relaxed LP in a node is integer, or the relaxed LP is infeasible, or the optimal value of the relaxed LP is inferior to the best feasible solution encountered so far. We call these nodes fathomed. The leaves of the search tree under exploration are the LPs that are not yet solved, infeasible or with a feasible solution to the MIP.

During the search process at the leaf nodes, the node having a MIP feasible solution with the best objective function value is called the best bound. The difference between the objective's value of any node and the best bound's objective value is called the *gap*. We can demonstrate optimality by having a gap with a value of zero.

Before applying the branch-and-bound algorithm, usually, the MIP solvers run a presolve step to reduce the problem. These reductions aim to reduce the size of the problem and to tighten the formulation for faster solving time. Presolve step in any MIP solver is a critical step that may considerably reduce the linear relaxation feasible search space.

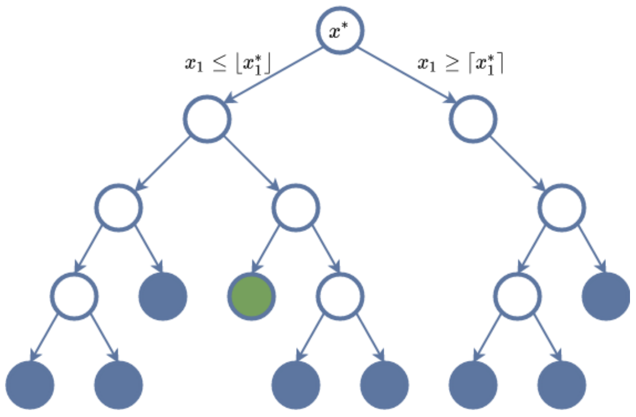

x^*
$x_1 \leq \lfloor x_1^* \rfloor$
$x_1 \geq \lceil x_1^* \rceil$

Fig. 1.5. Branch-and-bound algorithm used in solving mixed-integer programs.

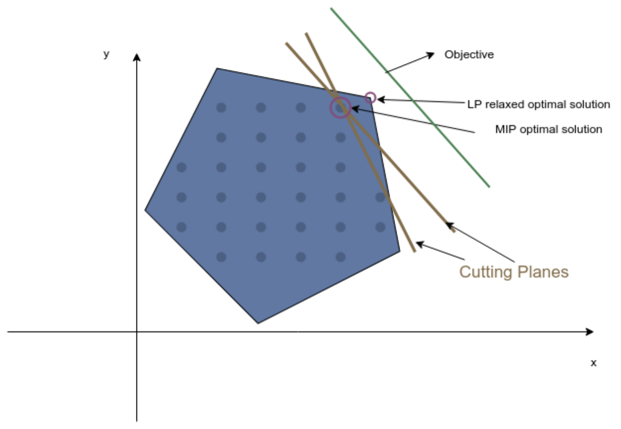


Fig. 1.6. Cutting plane method used to refine a feasible set by valid cuts.

Gomory (1969) introduced cutting planes used to tighten the relaxations by removing the undesired fractional solutions during the solution process without branching, see Figure 1.6.

Heuristic methods are a part of modern solvers with the aim of finding rapidly a good feasible solution, although with no guarantee of optimality. Examples of heuristic methods include simple rounding heuristics, feasibility pump (Achterberg, 2009; Fischetti et al., 2005), meta-heuristics (Glover and Kochenberger, 2006), relaxation induced neighborhood search (Danna et al., 2005) and local branching (Fischetti and Lodi, 2003).

Advances in combinatorial optimization such as branching techniques, bounds tightening, valid inequalities, decomposition and heuristics, to name few, have resulted in powerful solvers that can in practice solve MIPs of large size in seconds.

1.3. Related Work

Pruning neurons from an over-parameterized neural model has been an active research area and an open challenging problem. Identifying which neurons to remove from a model requires computing an importance score for each neuron. The importance score can be used to prune, interpret and identify the critical features learned by the neural model. Different objectives have been proposed to decide which neurons are important. Next, we will review the related literature in the neural architecture search, compression, the lottery ticket hypothesis and use of combinatorial optimization on trained ANNs.

Classical weight pruning methods. LeCun et al. (1990) proposed the optimal brain damage that theoretically prunes weights having a small "saliency", i.e., those having minor effect on the training error. The saliency is predicted by computing the weights' second derivatives with respect to the training objective. Hassibi and Stork (1993) introduced the optimal brain surgeon that aims at removing "non-critical" weights; these are weights that when pruned, result in a marginal loss of the network's predictive capacity. The non-critical weights are determined by approximating the inverse of the Hessian of the network. Another approach is presented by (Chauvin, 1989; Weigend et al., 1991), where a penalizing term is added to the loss function during the model training (e.g., L0 or L1 norm) as a regularizer. The model is sparsified during backpropagation of the loss function. Since these classical methods depend *i)* on the scale of the weights, *ii)* are incorporated during the learning process, and, *iii)* some of them; rely on computing the Hessian with respect to some objective, they turn out to be slow, requiring iterations of pruning and fine-tuning to avoid loss of accuracy. OAMIP identifies a set of non-critical neurons that when pruned simultaneously results in a marginal loss of accuracy without the need of fine-tuning or re-training.

General weight pruning methods. These methods generally prune the network without following any constraint, which is simply pruning entire neurons or specific connections. Molchanov et al. (2016) devised a greedy criteria-based pruning with fine-tuning by backpropagation. The criterion devised is given by the absolute difference between dense and sparse neural model loss (ranker). This cost function ensures that the model will not significantly decrease its predictive capacity. The drawback of this approach is in requiring a retraining after each pruning step. Shrikumar et al. (2017) developed a framework that computes the neurons' importance at each layer through a single backward pass with respect to each output target. This technique compares the activation values among the neurons and assigns a contribution score to each of them based on the predicted output of each input data point. Other related techniques, using different objectives and interpretations of neurons importance, have been presented (Berglund et al., 2015; Barros and Weber, 2018; Liu et al., 2018a; Yu et al., 2018; Hooker et al., 2019). They all demand intensive computation, while OAMIP aims at efficiently determining the neurons' importance.

Lee et al. (2018) investigated the pruning of connections, instead of entire neurons. The connections' sensitivity is studied through the model's initialization and a batch of input data. Connections sensitivities lower than a certain threshold are removed. This proposed technique is the current state-of-the-art in deep networks' compression. The generalization across different datasets was not studied in their proposed approach and the connections' sensitivity are computed at the initialization. On the other hand, we show the generalization of OAMIP and its ability to compute the neuron importance on different convergence levels of the trained neural network.

Structured weight filter pruning methods. These methods prune the network in a structured way like placing sparsified parameters at well-defined locations enabling GPUs to exploit computational savings. Srinivas and Babu (2015) proposed a data free way of pruning neural models by computing correlation between neurons. This correlation is computed using a saliency of two weight sets for all possible sets in the model. The saliency set computation is computationally intensive. Li et al. (2016); Molchanov et al. (2016); Jordao et al. (2018) focused on methods to prune entire filters in convolutional neural networks which include pruning and retraining to regain lost accuracy. He et al. (2018) proposed an approach to rank entire filters in convolutional networks. The ranking method uses the L1 norm of the weights for each filter. This approach is effective to detect non-critical filters. However, it involves alternating between pruning the lowest ranking filters and retraining, in order to avoid a drop in accuracy. Our work focuses on structured weight compression and it has the ability of being applied to convolutional filters in Toeplitz[2] fully connected format.

AutoML. Zoph and Le (2016) were the first to draw attention to automatic architecture search by using a reinforcement learning technique to search for the best performing architecture of a recurrent neural network. Since then, a set of techniques were proposed specifically for neural architecture search. Cortes et al. (2017) focused on an adaptive model that learns both the architecture of the model and its weight using a training policy. This policy works on minimizing the generalization bound and makes the choice of whether, or not to expand a subnetwork as the current. This proposed policy is a convex function of w, which makes it easy to optimize. Optimizing this proposed policy gives a good model achieving high results on the provided dataset. Weill et al. (2019) proposed a technique based on AdaNet (Cortes et al., 2017) that learns the structure of a neural network as an ensemble of weak learners. In the neural architecture search techniques, we do not just learn the weights but also the features associated to each dataset depending on how important the connection is to that specific dataset. We refer the reader to (He et al., 2019; Zöller and Huber, 2019) for more details about AutoML.

[2]Converting convolutional layers to Toeplitz blocks is explained in depth in Section 2.3

Lottery ticket. Frankle and Carbin (2018) introduced the lottery ticket conjecture and empirically showed the existence of a lucky pruned subnetwork, a *winning ticket.* This winning ticket can be trained effectively with fewer parameters without loss in accuracy as formally explained in Conjecture 1.3.1.

***Conjecture* 1.3.1.** The Lottery ticket hypothesis: A randomly-initialized, dense neural network contains a subnetwork that is initialized such that when trained in isolation it can match the test accuracy of the original network after training for at most the same number of iterations.

Morcos et al. (2019) proposed a technique for sparsifying n over-parameterized trained neural models based on the lottery hypothesis. Their technique involves pruning the model and disabling some of its subnetworks. The pruned model can be fine-tuned on a different dataset achieving good results. The lucky subnetwork is found by iteratively pruning the lowest magnitude weights and retraining. Another phenomenon discovered by (Ramanujan et al., 2020; Wang et al., 2020a), was the existence of smaller high-accuracy models that resides within larger random networks. This phenomenon is called strong lottery ticket hypothesis and was proved by (Malach et al., 2020) on ReLU fully connected layers. Furthermore, Wang et al. (2020b) proposed a technique of selecting the winning ticket at initialization before training the ANN by computing an importance score, based on the gradient flow in each unit.

Mixed-integer programming. Fischetti and Jo (2018) and Anderson et al. (2019) represent a ReLU ANN using a MIP. Fischetti and Jo (2018) presented a big-M formulation to represent trained ReLU neural networks. Later, Anderson et al. (2019) introduced the strongest possible tightening to the big-M formulation by adding strengthening separation constraints when needed[3], which reduced the solving time by orders of magnitude. All the proposed formulations, are designed to represent trained ReLU ANNs with fixed parameters. In OAMIP, we used the formulation from (Fischetti and Jo, 2018) because its performance was good due to our tight local variable bounds, and its polynomial number of constraints. On the other hand, Anderson et al. (2019)'s model has an exponential number of constraints). Representing an ANN as a MIP can be used to evaluate robustness, compress networks and create adversarial examples for the trained neural network. Tjeng et al. (2019) used a big-M formulation to evaluate the robustness of neural models against adversarial attacks. In their proposed technique, they assessed the ANN's sensitivity to perturbations in input images. The MIP solver tries to find a perturbed image (adversarial attack) that would get misclassified by the ANN. Serra et al. (2020) also used a MIP to maximize the compression of an existing neural network without any loss of accuracy. Different ways of compressing (removing neurons, folding layers, etc) are presented. However, the reported computational

[3]The cut callbacks in Gurobi were used to inject separated inequalities into the cut loop.

experiments lead only to the removal of inactive neurons. Our method has the capability to identify such neurons, as well as to identify other units that would not significantly compromise accuracy.

Huang et al. (2019) used also mathematical programming models to check neural models' robustness in the domain of natural language processing. In their proposed technique, the bounds computed for each layer would get shifted by an epsilon value for each input data point for the MIP. This epsilon is the amount of expected perturbation in the input adversarial data.

Chapter 2

ANN as a MIP

In this chapter, we explain how to formulate a neural network as a mixed-integer program. As seen in Section 1.2, a neural network's architecture consists of a set of affine linear transformations with non-linear activation functions. Recall that a commonly used non-linearity is the Rectified Linear Unit (ReLU), whose output is the maximum between the input and zero. However, we start by introducing the linearly connected layers' formulations as a base for the upcoming ReLU formulations. Furthermore, given that convolutional and pooling layers are crucial for learning tasks involving images, we also introduce them. Mainly, we model fully connected layers as a mixed-integer program, and then add each of the aforementioned's elements to it: ReLU activations, convolutional layers, pooling and batch norm. Finally, we discuss the computation of bounds for the variables of the MIP which significantly impacts solving times.

The motivation behind representing neural networks using a MIP results from their successful application in *1)* deep reinforcement learning with high dimensional action spaces where the state transition cost is learned by a neural network and the policy is determined by solving a MIP (Ryu et al., 2019; Mladenov et al., 2017), *2)* neural network's robustness verification against adversarial examples (Tjeng et al., 2019; Huang et al., 2019), and *3)* compression of neural networks (Serra et al., 2020).

2.1. Linear Fully Connected Layers

We will recover the neural network notation from Chapter 1. In what follows. We assume that the neural network was trained, i.e., the parameters are known. This is because our goal is to prune a trained neural network without having to fine-tune or to re-train. For each input data point x, let h^l be a decision vector denoting the output value of layer l, $w_i^{(l)}$ row i of $\boldsymbol{W}^{(l)}$, and $b_i^{(l)}$ ith element of bias vector $b^{(l)}$, i.e., $h^l = \boldsymbol{W}^{(l)} h^{l-1} + b^{(l)}$ for $l > 0$ and $h^0 = x$.

For the sake of simplicity, we describe the formulation for one layer l at neuron i of the model and one input data point x[1].

We start by presenting the constraints used in a MIP formulation for linear fully connected layers without activation functions. To this end, the following set of constraints completely mimics the linear fully connected layers:

$$h_i^l = x_i \quad \text{if } l = 0, \text{ otherwise} \tag{2.1.1a}$$

$$h_{i+1}^l = w_i^{(l)} h_i^l + b_i^{(l)} \quad \forall l = 1, \ldots, n \quad \forall i = 1, \ldots, N_l \tag{2.1.1b}$$

where the output of layer i is denoted by the decision vector h_{i+1}^l, and its value is based on the parameters $w_i^{(l)}$, and $b_i^{(l)}$. Without a fixed set of parameters (a trained network), along with an objective defined by the user depending on their end goal, the problem is extremely hard due to its non-linearity.

2.2. ReLU Activated Layers

Next, we provide the representation of ReLU neural networks of (Fischetti and Jo, 2018) called big-M formulation. Although, Anderson et al. (2019) proposed an ideal MIP formulation, where there is an exponential number of facets defining constraints that can be separated efficiently, we used only the big-M formulation which performed well since we can compute tight local bounds.

The ReLU activation functions are used to increase the model's capacity and to introduce non-linearity between its layers. The ReLU is a piecewise non-linear activation function that enables positive logits only. For this purpose, we use a gating binary decision variable z_i^l in addition to the previous variables used in the linear fully connected layers.

Let $h^l = ReLU(\boldsymbol{W^{(l)}} h^{l-1} + b^{(l)})$, and an additional decision variable z_i^l that takes value 1 if the unit i is active, i.e., $w_i^{(l)} h^{l-1} + b_i^{(l)} \geq 0$, and 0 otherwise. Finally, let L_i^l and U_i^l be constants indicating a valid lower and upper bound for the input of each neuron i in layer l. We discuss the computation of these bounds in Section 2.6. For now, we assume that L_i^l and U_i^l are sufficiently small and large numbers, respectively, i.e., the so-called big-M values. The constraints for a ReLU activation at layer l on neuron i are:

[1]These constraints must be repeated for each layer and each input data point.

$$h_i^l \geq 0, \tag{2.2.1a}$$

$$h_i^l + (1 - z_i^l)L_i^l \leq w_i^l h^{l-1} + b_i^l, \tag{2.2.1b}$$

$$h_i^l \leq z_i^l U_i^l, \tag{2.2.1c}$$

$$h_i^l \geq w_i^l h^{l-1} + b_i^l, \tag{2.2.1d}$$

$$z_i^l \in \{0, 1\}. \tag{2.2.1e}$$

We additionally make $z_i^0 = 1$, so that constraint (2.1.1a) forces the initial decision vector h^0 to be equal to the input x of the first layer. When z_i^l is 0, constraints (2.2.1a) and (2.2.1c) force h_i^l to be zero, reflecting a non-active neuron. If an entry of z_i^l is 1, then constraints (2.2.1b) and (2.2.1d) enforce h_i^l to be equal to $w_i^l h^{l-1} + b_i^l$. See (Fischetti and Jo, 2018; Anderson et al., 2019) for details.

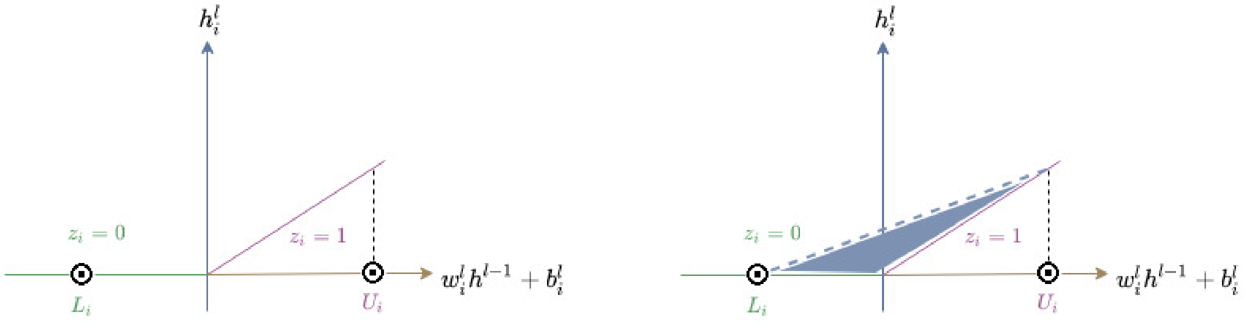


Fig. 2.1. Effect of relaxing the gating binary decision variable in the ReLU formulation, and the effect on the search space of feasible solutions.

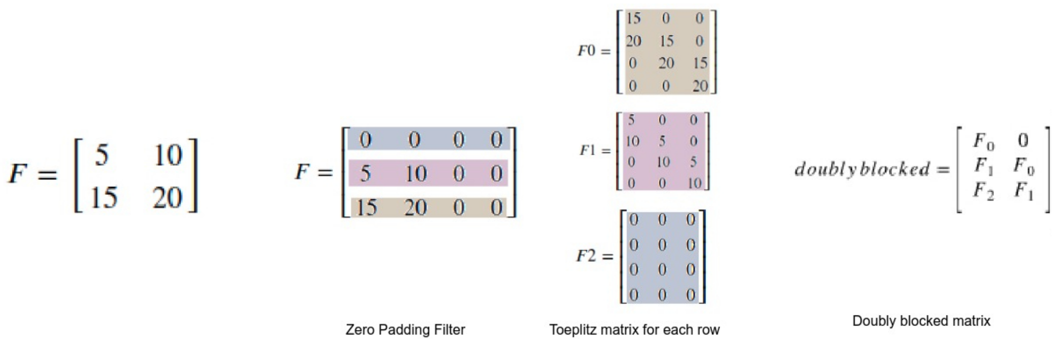
After formulating the ReLU, if we relax the binary constraint (2.2.1e) on z_i^l to $[0, 1]$, we obtain a linear programming problem which is easier and faster to solve. This relaxation would convert the ReLU function from Figure 2.1 into the convex set in blue. This allows to see the role of the binary requirement on z_i^l, and the trade-off between an exact MIP representation of a ReLU activation function and its approximation by relaxing the MIP in one unit. Furthermore, the quality (tightness) of such relaxation highly depends on the choice of tight upper and lower bounds, U_i, L_i. In fact, the determination of tight bounds reduces the search space and hence, the solving time (recall from Chapter 1 that MIP solvers rely on LP relaxations).

2.3. Convolutional Layers

We convert the convolutional feature map to a Toeplitz matrix, and the input image to a vector. Hence, allowing us to use simple matrix multiplication that can be efficiently computed, but with high memory cost. Through this procedure, we can represent the convolutional layer using the same formulation of linear or ReLU activated fully connected layers presented in Section 2.1 and Section 2.2.

Toeplitz Matrix is a matrix in which each value is along the main diagonal and sub diagonals are constant. So given a sequence a_n, we can create a Toeplitz matrix by putting the sequence in the first column of the matrix and then shifting it by one entry in the following columns. Figure 2.2 shows the steps of creating a doubly blocked Toeplitz matrix from an input filter.

$$\begin{pmatrix} a_0 & a_{-1} & a_{-2} & \cdots & \cdots & \cdots & \cdots & a_{-(N-1)} \\ a_1 & a_0 & a_{-1} & a_{-2} & & & & \vdots \\ a_2 & a_1 & a_0 & a_{-1} & \ddots & & & \vdots \\ \vdots & a_2 & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & a_{-2} & \vdots \\ \vdots & & & \ddots & a_1 & a_0 & a_{-1} & a_{-2} \\ \vdots & & & & a_2 & a_1 & a_0 & a_{-1} \\ a_{(N-1)} & \cdots & \cdots & \cdots & \cdots & a_2 & a_1 & a_0 \end{pmatrix}. \qquad (2.3.1)$$



F = [5 10; 15 20]
F = [0 0 0 0; 5 10 0 0; 15 20 0 0]
F0 = [15 0 0; 20 15 0; 0 20 15; 0 0 20]
F1 = [5 0 0; 10 5 0; 0 10 5; 0 0 10]
F2 = [0 0 0; 0 0 0; 0 0 0; 0 0 0]
doublyblocked = [F0 0; F1 F0; F2 F1]
Zero Padding Filter
Toeplitz matrix for each row
Doubly blocked matrix

Fig. 2.2. Steps for converting a filter to doubly blocked Toeplitz matrix.

Feature maps or kernels, as discussed in Chapter 1.1, at each input channel are flipped, and then converted to a matrix. The computed matrix when multiplied by the vectorized input image will provide the fully convolutional output. For padded convolution, we use only parts of the output of the full convolution, and for the strided convolutions we use sum of 1 strided convolutions as proposed by (Brosch and Tam, 2015). First, we pad zeros to the top and right of the input feature map to have same size as the output of the full convolution. Then, we create a Toeplitz matrix for each row of the zero padded feature map. Finally, we arrange these small Toeplitz matrices in a large doubly blocked Toeplitz matrix. Each small

Toeplitz matrix is arranged in the doubly Toeplitz matrix in the same way a Toeplitz matrix is created from the input sequence, with each small matrix as an element of the sequence, as shown in Figure 2.2.

The formulation will become $\sum_{d=1}^{N^l} \boldsymbol{W_d^{(l)}} h^{l-1} + b^{(l)}$ at each feature map. We then, incorporate the same constraints used for the fully connected layer repeated for each feature map.

2.4. Pooling Layers

Pooling layers are used to reduce the spatial representation of an input image by applying an arithmetic operation on each feature map of the previous layer, as shown in Figure 2.3. We model both average and max pooling on multi-input units as constraints of a MIP formulation with kernel dimensions *ph* and *pw*.

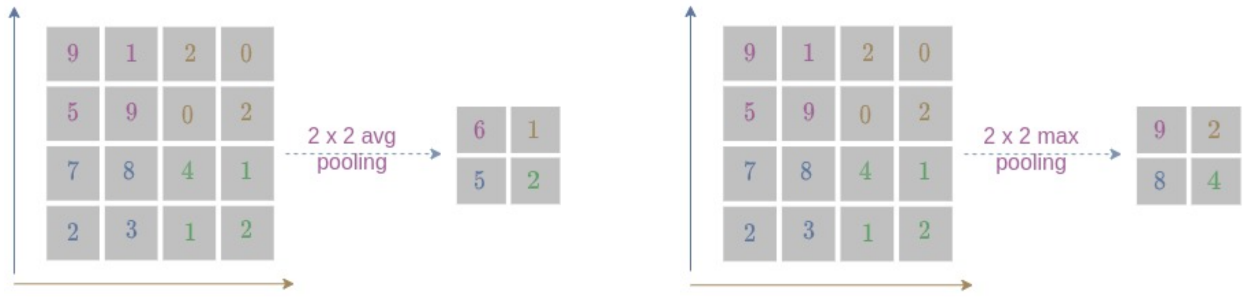


Fig. 2.3. Diagram of average pooling in the left, and max pooling in the right.

Avg Pooling layer applies the average operation on each feature map of the previous layer. This operation is just linear and it can be easily incorporated into our MIP formulation:

$$x = \text{AvgPool}(y_1, \ldots, y_{ph*pw}) = \frac{1}{ph * pw} \sum_{i=1}^{ph*pw} y_i. \tag{2.4.1a}$$

Max Pooling takes the maximum of each feature map of the previous layer.

$$x = \text{MaxPool}(y_1, \ldots, y_{ph*pw}) = \max\{y_1, \ldots, y_{ph*pw}\}. \tag{2.4.2a}$$

This operation can be expressed by introducing a set of binary variables $m_1, \ldots, m_{ph*pw}$.

$$\sum_{i=1}^{ph*pw} m_i = 1 \tag{2.4.3a}$$

$$\left.\begin{aligned} x &\geq y_i, \\ x &\leq y_i m_i + U_i(1 - m_i) \\ m_i &\in \{0,1\} \end{aligned}\right\} i = 1, \ldots, ph * pw. \tag{2.4.3b}$$

Fischetti and Jo (2018) devised the max and avg pooling representation using a MIP. The max pooling representation contains a set of gating binary variables enabled only for the maximum value y_i in the set $\{y_1, \ldots, y_{ph*pw}\}$.

2.5. Batch Norm

Ioffe and Szegedy (2015) introduced a technique to reduce internal covariate shift between layers using batch normalization. Internal covariate shift can be defined as the change in the distribution of the network architecture due to the parameters update during the training. The reduction in internal covariate shift would make the landscape of the corresponding optimization problem significantly smoother. The smoothness of the optimization space would allow us to use a larger range of learning rates with faster convergence rate. These features made the batch norm layers popular; they are thus used in most deep learning architectures.

The batch norm operation normalizes the input by subtracting the mini batch mean and dividing it by the variance, then scale and shift are applied using learnable parameters γ and β. Thus, this operation is linear and can be easily represented in a MIP formulation. Given the fixed parameters γ and β, along with the running mean and variance computed during the training on mini-batches μ and σ^2, we have a constant ϵ added during the computation for numerical stability.

$$h^{l+1} = \gamma \frac{h^l - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta. \tag{2.5.1}$$

Equation (2.5.1) shows the constraint for a batch norm at layer $l+1$ based on the output of the previous layer h^l. Its value is saved into the decision variable h^{l+1}.

2.6. Computing Tight Bounds

In the described MIP constraints, we assumed to have a large upper bound U_i^l and a small lower bound L_i^l. However, using large bounds increases the feasible space of the MIP relaxation, as shown in Figure 2.1, causing long computational times. In order to overcome this issue, we compute tight bounds on the input that are tailored to our particular use of the MIP models. Before describing our bounds computation procedure, we review different techniques of bounds tightening and the differences between them.

The standard approach computes the bounds for each unit by simply solving its MIP formulation for two different objective functions. The first, maximizes the decision variable value, leading to an upper bound. The second, minimizes the decision variable value, leading to a lower bound. Moreover, we can halt the solver before termination, and yet get useful bounds, as shown by (Cheng et al., 2017b). Since the ANN is an acyclic graph, the tightened bounds computed in one unit can be used for subsequent computation of bounds to the remaining layers. However, this approach is computationally expensive, it leads to global tight bounds that do not depend on the input, and can be saved and used for the same ANN on different input data points. Cheng et al. (2017b); Fischetti and Jo (2018) used this approach to compute tight bounds. Alternatively, the MIPs used by this bound computation method can be relaxed (LP bounds tightening), leading to a more efficient way to compute the bounds, at the expense of weaker bounds.

Interval arithmetic is an efficient technique to compute tight local bounds depending on input data points x fed to the MIP in Equation (2.1.1a). We tailor these bounds accordingly with their respective input point x by considering small perturbations on its value:

$$L^0 = x - \epsilon \tag{2.6.1a}$$

$$U^0 = x + \epsilon \tag{2.6.1b}$$

$$L^l = \boldsymbol{W}^{(l-)} U^{l-1} + \boldsymbol{W}^{(l+)} L^{l-1} \tag{2.6.1c}$$

$$U^l = \boldsymbol{W}^{(l+)} U^{l-1} + \boldsymbol{W}^{(l-)} L^{l-1} \tag{2.6.1d}$$

$$\boldsymbol{W}^{(l-)} \triangleq \min(\boldsymbol{W}^{(l)}, 0) \tag{2.6.1e}$$

$$\boldsymbol{W}^{(l+)} \triangleq \max(\boldsymbol{W}^{(l)}, 0). \tag{2.6.1f}$$

Propagating the initial bounds of the input data points throughout the trained model will create the desired bound using the arithmetic interval (Moore et al., 2009). The obtained bounds are tight, narrowing the space of feasible solutions.

Moreover, this procedure is extremely efficient, particularly when compared with the standard approach which requires solving multiple optimization problems. Therefore, in our proposed work, we used the interval arithmetic which complied with our usage for input data points.

2.7. Objective Function

After defining the constraints and bounds, for a complete MIP description, the user needs to define the problem's objective to be optimized.

For example, if we want to generate adversarial examples for a classification task of an ANN function denoted as h_θ with perturbation δ upper bounded by ϵ, our objective would be to maximize the target class logit y_{targ}, and minimize all other logits denoted by y':

$$\underset{\|\delta\| \leq \epsilon}{\text{Maximize}} \left(h_\theta(x+\delta)_{y_{\text{targ}}} - \sum_{y' \neq y_{\text{targ}}} h_\theta(x+\delta)_{y'} \right). \tag{2.7.1}$$

We have shown the constraints representing an ANN using a MIP, the computation of bounds and an example of an objective function. Next, we describe how we build OAMIP for identifying critical neurons in a trained ANN based on these MIP representations.

Chapter 3

Neuron Importance Score

In this chapter, we will describe the mathematical programming formulation devised by us to model the computation of ANN layers with ReLU activation functions, while being able to evaluate the effect on the predictive task of each neuron. To that end, we introduce a neuron importance score variable (ElAraby et al., 2020) to the previously introduced MIP constraints for ANN in Section 2.1, denoting how critical is each neuron for the trained neural network.

Our main focus is to compute an importance score between 0 and 1 for each neuron in an ANN. The importance score represents the contribution and the activation of each neuron to the learning task at hand. Neurons with a score close to 1 are extremely critical to the ANN, while neurons with a score close 0 are non-critical. With the computed neuron importance score, we can prune neurons having a low importance score (non-critical) without losing the model's predictive capacity. In this way, we can obtain sparse architectures.

3.1. Constraints

We modelled ReLU activated layers with the ability to evaluate the effect of decreasing the neurons' activation. We keep the previously introduced binary variables z_i^l, and continuous variables h_i^l defined for each input data point x. Additionally, we create the continuous decision variables $s_i^l \in [0,1]$ representing neuron i importance score in layer l. In this way, we modified the ReLU constraints (2.2.1) by adding the neuron importance decision variable s_i^l to constraints (2.2.1b) and (2.2.1d). Remark that s_i^l is the same for all input data points.

$$h_i^l + (1 - z_i^l)L_i^l \leq w_i^l h^{l-1} + b_i^l - (1 - s_i^l)\max(U_i^l, 0) \tag{3.1.1a}$$

$$h_i^l \geq w_i^l h^{l-1} + b_i^l - (1 - s_i^l)\max(U_i^l, 0). \tag{3.1.1b}$$

In (3.1.1), when neuron i is activated due to the input h^{l-1}, i.e., $z_i^l = 1$, h_i^l is equal to the right-hand-side of those constraints. This value can be directly decreased by reducing the neuron importance s_i^l. When neuron i is non-active, i.e., $z_i^l = 0$, constraint (3.1.1b) becomes irrelevant as its right-hand-side is negative. This fact together with constraints (2.2.1a) and (2.2.1c), imply that h_i^l is zero. Now, we claim that constraint (3.1.1a) allows s_i^l to be zero if that neuron is indeed non-important, i.e., for all possible input data points, neuron i is not activated. This claim can be shown through the following observations. Note that decisions h and z must be replicated for each input data point x as they present the propagation of x over the neural network. On the other hand, s evaluates the importance of each neuron for the main learning task and thus, it must be the same for all data input points. Thus, the key ingredients are the local bounds L_i^l and U_i^l that are computed for each input data point, as explained in Section 2.6. In this way, if U_i^l is non-positive for all considered data points, s_i^l can be zero without interfering with the constraints (3.1.1). The latter is enforced by the objective function derived in Section 3.2 which minimizes the neuron importance scores. If U_i^l is positive for some input data point, the decrease on s_i^l will be limited by the fact that h_i^l must be non-negative for that data point. We remark that the decrease in s_i^l relies heavily on the coefficient of $(1 - s_i^l)$ in (3.1.1). While our constraints correctly propagate the computation of some data point x, while allowing some neurons' activation decrease, the choice of the coefficient $\max(U_i^l, 0)$ may be resulting in an over estimation of neurons' importance score. We apply an element-wise ReLU on the upper bound $\max(U_i^l, 0)$ in order to zero out the neurons' importance score for dead neurons.

In Figure 3.1, we show the new bounds when we add the neuron importance score to the constraints of the ReLU compared to the previous bounds. Moreover, the correctness of the computed neuron importance scores relies heavily on the tightness of the bounds. In brief, the bounds need to be computed locally per each input data point, which explains the choice of interval arithmetic discussed in Section 2.6.

We note that this MIP formulation can naturally be extended to convolutional layers converted to matrix multiplication using Toeplitz matrix (Gray, 2000), as explained in Section 2.3, and with an importance score associated with each feature map.

3.2. MIP Objective

The aim for the proposed framework is to sparsify the ANN by removing non-critical neurons without reducing the predictive accuracy of the pruned ANN.

We consider a neuron critical, if once pruned, the ANN's predictive capacity drops dramatically, and non-critical, if once pruned, it causes only a marginal decrease in the ANN's predictive capacity. To this end, we combine two optimization objectives.

Our first objective is to maximize the set of neurons sparsified from the trained ANN. Let n be the number of layers, N^l the number of neurons at layer l, and $I^l = \sum_{i=1}^{N^l}(s_i^l - 2)$ be the sum of neuron importance scores at layer l with s_i^l scaled down to the range $[-2, -1]$.

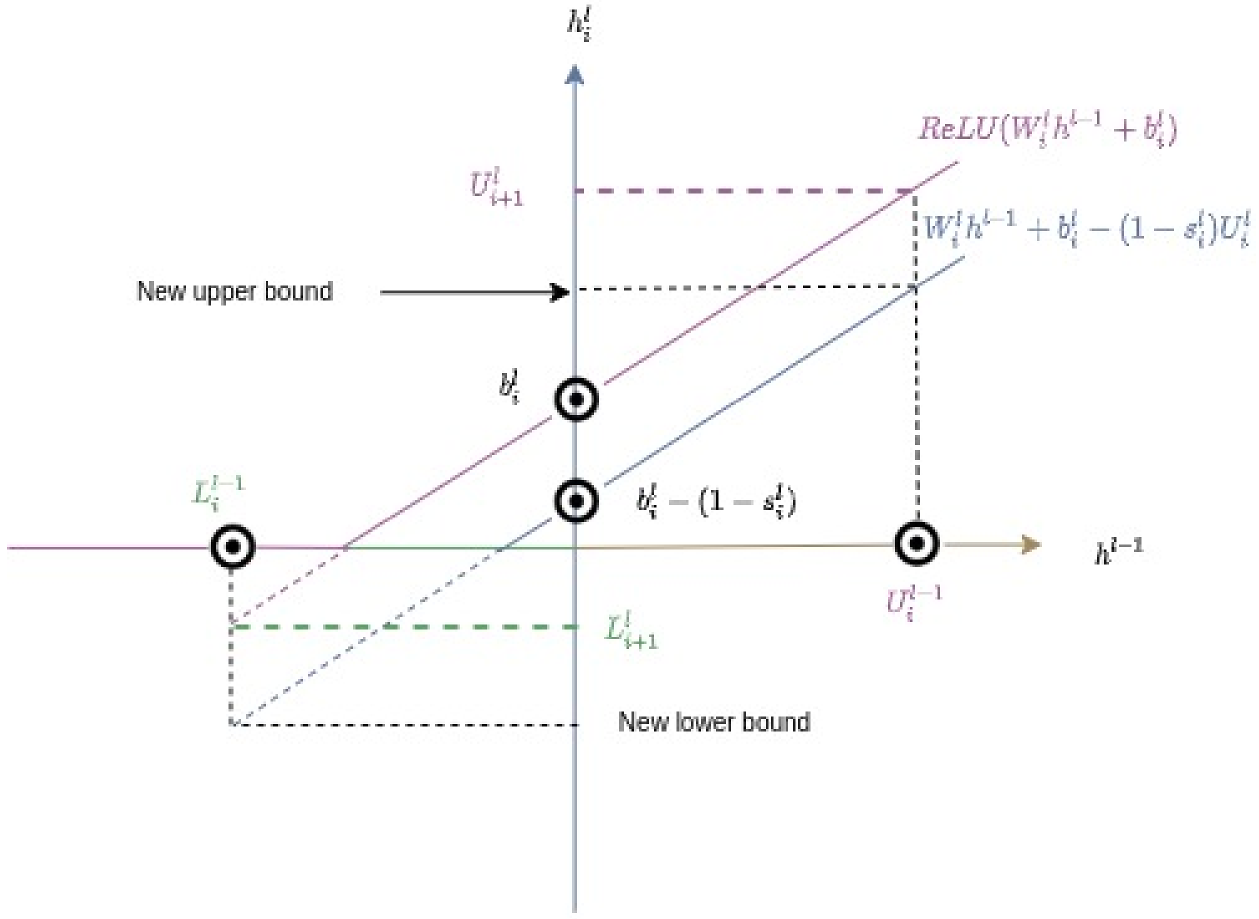


Fig. 3.1. The bounds changing with the introduced decision variable s_i^l.

Table 3.1. Importance of re-scaling sparsification objective to prune more neurons shown empirically on LeNet-5 model using threshold 0.05, by comparing accuracy on test set between reference model (Ref.), and pruned model (Masked).

Dataset	Objective	Ref. Acc.	Masked Acc.	Pruning Percentage (%)
MNIST	$s_i^l - 2$	98.9% ± 0.1	**98.7% ± 0.1**	**13.2% ± 2.9**
	$s_i^l - 1$		98.8% ± 0.1	9.6% ± 1.1
	s_i^l		98.9% ± 0.2	8% ± 1.6
Fashion-MNIST	$s_i^l - 2$	89.9% ± 0.2	**89.1% ± 0.3**	**17.1% ± 1.2**
	$s_i^l - 1$		89.2% ± 0.1	17% ± 3.4
	s_i^l		89% ± 0.4	10.8% ± 2.1

In order to create a relation between neurons' importance score in different layers, our objective becomes the maximization on the amount of neurons sparsified from the $n-1$ layers with higher score I^l, which with the re-scaling incorporates the layer size. Hence, we denote $A = \{I^l : l = 1, \dots, n\}$ and formulate the sparsity loss as

$$\text{sparsity} = \frac{\max\limits_{A' \subset A, |A'| = (n-1)} \sum\limits_{I \in A'} I}{\sum_{l=1}^{n} N^l}. \tag{3.2.1}$$

Here, the objective is to maximize the number of non-critical neurons (small importance scores) at each layer compared to other layers in the trained neural model. Note that only the $n-1$ layers with the largest importance score will weight in the objective, allowing to reduce the pruning effort on some layer that will naturally have low scores. The sparsity quantification is then normalized by the total number of neurons.

In Table 3.1, we compare re-scaling the neuron importance score in the objective function to $[-2, -1]$, to $[-1, 0]$ and no re-scaling $[0, 1]$ using LeNet-5 (LeCun et al., 1998) trained on MNIST (LeCun et al., 2010) and Fashion-MNIST (Xiao et al., 2017). This comparison shows empirically the importance of re-scaling the neuron importance score to optimize sparsification through neuron pruning.

Our second objective is to minimize the loss of *important* information due to the sparsification of the trained neural model. Additionally, we aim for this minimization to be done without relying on the values of the logits, which are closely correlated with neurons pruned at each layer. Otherwise, this would drive the MIP to simply give a full score of 1 to all neurons in order to keep the same output logit value. Instead, we formulate this optimization objective using the marginal softmax as proposed in (Gimpel and Smith, 2010). Using marginal softmax allows the solver to focus on minimizing the misclassification error without relying on logit values. Marginal softmax loss avoids putting a large weight on logits coming from the trained neural network and predicted logits from decision vector h^n computed by the MIP. On the other hand, in the proposed marginal softmax loss, the label having the highest logit value is the one optimized regardless its value. Formally, we write the objective

$$\text{softmax} = \sum_{i=1}^{N^n} \log\left[\sum_c \exp(h^n_{i,c})\right] - \sum_{i=1}^{N^n} \sum_c Y_{i,c} h^n_{i,c}, \tag{3.2.2}$$

where index c stands for the class label. The used marginal softmax objective keeps the correct predictions of the trained model for the input batch of images x having one hot encoded labels Y without considering the logit value.

Finally, we combine the two objectives to formulate the multi-objective loss

$$\text{loss} = \text{sparsity} + \lambda \cdot \text{softmax}, \tag{3.2.3}$$

as a weighted sum of sparsification regularizer and marginal softmax, as proposed by (Ehrgott, 2005). Our experiments revealed that $\lambda = 5$ generally provides the right trade-off between our two objectives.

3.3. Choice of λ

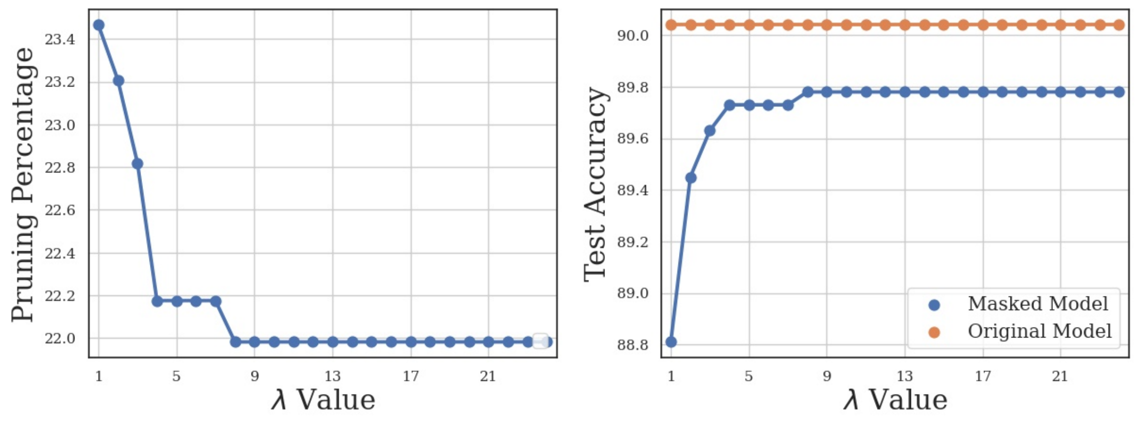


Fig. 3.2. Effect of changing value of λ when pruning LeNet-5 model trained on Fashion-MNIST.

Note that our objective function (3.2.3) is implicitly using a Lagrangian relaxation, where $\lambda \geq 0$ is the Lagrange multiplier. In fact, the loss on accuracy versus pruning percentage needs to be controlled using Equation (3.2.2) by imposing the constraint softmax$(h) \leq \epsilon$ for a very small ϵ, or even to avoid any loss via $\epsilon = 0$. However, this would introduce a nonlinear constraint, which would be hard to handle. Thus, for tractability purposes we follow a Lagrangian relaxation on this constraint, and penalize the objective whenever softmax(h) is positive. Accordingly, with the weak (Lagrangian) duality theorem, the objective (3.2.3) is always a lower bound to the problem where we minimize sparsity. A bound on the accuracy loss is imposed. Furthermore, the problem of finding the best λ for the Lagrangian relaxation, formulated as

$$\max_{\lambda \geq 0} \min\{\text{sparsity} + \lambda \cdot \text{softmax}\}, \tag{3.3.1}$$

has the well-known property of being concave, which in our experiments revealed to be empirically determined[1]. We note that the value of λ introduces a trade-off between pruning more neurons and the predictive capacity of the model. For example, increasing the value of λ would result in pruning fewer neurons, as shown in Figure 3.2, while the accuracy on the test set would increase.

[1]We remark that if the trained model contains misclassifications, there is no guarantee that problem (3.3.1) is concave.

Chapter 4

Network Pruning Experiments

In this chapter, we demonstrate empirically the robustness and the validity of OAMIP in identifying non-critical neurons. We start by describing the experimental setting in Section 4.1, namely, the neural network architectures to be analyzed, and the full description of OAMIP, i.e., the usage of neuron importance scores to prune the ANNs. We then show in Section 4.2 the robustness of our neuron pruning policy to different input data points used in our MIP formulation and different convergence levels of a neural network. Next, in Section 4.3, we validate empirically that the computed neuron importance scores are meaningful, i.e., it is crucial to guide the pruning accordingly with the determined scores. In Section 4.4, we proceed with experiments to show that subnetworks generated by OAMIP on a specific initialization can be transferred to another dataset with marginal loss in accuracy (lottery hypothesis). Finally, in Section 4.5, we compare our pruning methodology to SNIP (Lee et al., 2018), a framework used to compute connections sensitivity and to create a sparsified subnetwork based on the input dataset and model initialization. The code can be found here: `https://github.com/chair-dsgt/mip-for-ann`.

4.1. Experimental Setting

Architectures and Training. We used a simple fully connected 3-layer ANN (FC-3) model, with 300+100 hidden units, from (LeCun et al., 1998), and another simple fully connected 4-layer ANN (FC-4) model, with 200+100+100 hidden units. In addition, we used convolutional LeNet-5 (LeCun et al., 1998) consisting of two sets of convolutional and average pooling layers, followed by a flattening convolutional layer, then two fully-connected layers. The largest architecture investigated was VGG-16 (Simonyan and Zisserman, 2014) consisting of a stack of convolutional (conv.) layers with a very small receptive field: 3×3. The VGG-16 was adapted for CIFAR-10 (Krizhevsky, 2009) having 2 fully connected layers of size 512 and average pooling instead of max pooling. Each of these models was trained 3 times with different initialization.

All models were trained for 30 epochs using the RMSprop optimizer (Tieleman and Hinton, 2012) with 1e-3 learning rate for MNIST (LeCun et al., 2010) and Fashion-MNIST (Xiao et al., 2017). LeNet-5 on CIFAR-10 (Krizhevsky, 2009) was trained using the SGD optimizer with learning rate 1e-2 and 256 epochs[1]. VGG-16 on CIFAR-10 was trained using the Adam optimizer (Kingma and Ba, 2015) with 1e-2 learning rate for 30 epochs. Decoupled greedy learning (Belilovsky et al., 2019) was applied for training each layer of the VGG-16 network through a small auxiliary network. The motivation behind this training procedure is the scalability of our framework (OAMIP), which will be later clarified in Section 5.2. For the training of all models, the hyper parameters were tuned on the validation set's accuracy. All images were resized to 32 by 32 and converted to 3 channels to allow the generalization verification of the pruned network across different datasets.

Algorithm 4.1.1. OAMIP: Optimizing ANN using a MIP.

```
Input: Trained ANN, dataset $D$, threshold.
Output: subnetwork selected from the trained ANN.
Step 1: Select subset of images $D' \subset D$ to be fed into the MIP.
Step 2: Solve MIP restricted to $D'$ and save neuron importance score $s$.
Step 3: Remove every neuron $i$ from layer $l$ with $s_i^l \leq$ threshold from ANN.
Step 4: Return pruned ANN (sub-network).
```

MIP and Pruning Policy. Using all the training set as input to the MIP solver is intractable. Hence, we only use a subset of the data points to approximate the neuron importance score. Representing classes with a subset of the data points would give us an under estimation of the score, i.e., neurons will look less critical than they really are. To that extent, the selected subset of data points must be carefully chosen. Whenever we computed neuron importance scores for a trained model, we had to feed the MIP with a balanced set of images[2], each representing a class of the classification task (step 1). The aim was to avoid that the determined importance scores lead to pruning neurons (features) critical to a class represented by fewer images as input to the MIP. In decoupled layer wise, the neuron importance score was computed independently on each auxiliary network; then we fine-tuned the generated masks for 1 epoch to propagate the error across them. Decoupled training of each layer allowed us to represent deep models using the MIP formulation and to parallelize the computation per layer. We used $\lambda = 5$ in the MIP objective function (3.2.3) as discussed in Section 3.2. The proposed framework, recall Figure I.1 and Algorithm 4.1.1,

[1]CIFAR-10 dataset is harder than MNIST and Fashion-MNIST, thus requiring different hyperparameters.
[2]Recall that in all our experiments, the input data points are images.

computes an estimation of the importance score of each neuron (step 2), and with a small tuned threshold based on the network's architecture, we mask (prune) non-critical neurons with a score lower than the threshold (step 3).

Computational Environment. The experiments were performed in an Intel(R) Xeon(R) CPU @ 2.30GHz with 12 GB RAM and Tesla k80 using Mosek 9.1.11 (Mosek, 2010) solver on top of CVXPY (Agrawal et al., 2018; Diamond and Boyd, 2016) and PyTorch 1.3.1 (Paszke et al., 2019).

4.2. Robustness Verification

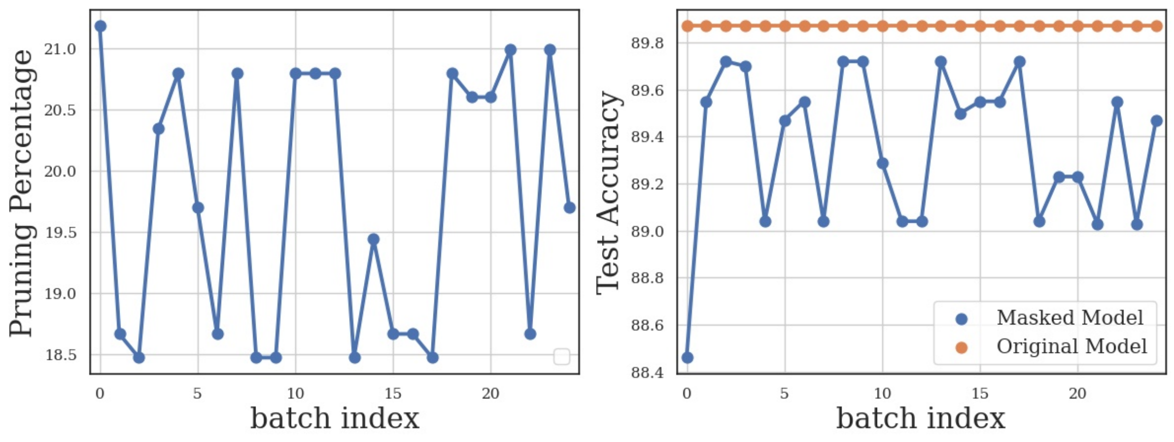


Fig. 4.1. Effect of changing validation set of input images.

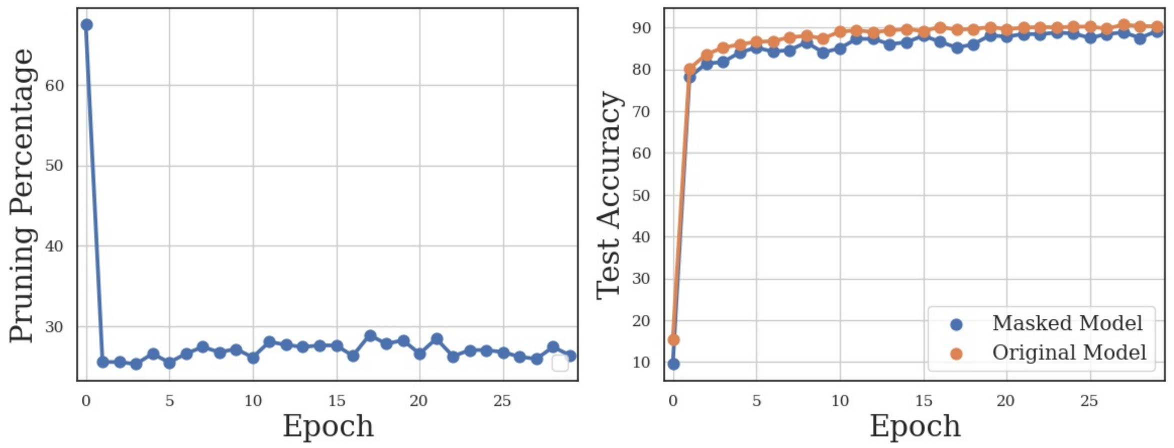


Fig. 4.2. Evolution of the computed masked subnetwork during model training.

We examine the robustness of OAMIP against different batches of input images fed into the MIP, on the implementation of step 2 of OAMIP. Namely, we used 25 randomly sampled balanced images from the validation set. Figure 4.1 shows that changing the input images used by the MIP to compute neuron importance scores in step 2 resulted in marginal changes in the test accuracy between different batches. We remark that the input batches may contain images that were misclassified by the neural network. In this case, the MIP tries to use the score s to obtain the true label, which explains the variations in the pruning percentage. Indeed, as discussed in Section 3.3 for the choice of λ, the marginal fluctuations of these results depend on the accuracy of the input batch used in the MIP. Furthermore, we show empirically that OAMIP is robust on different convergence levels of the trained neural network as shown in Figure 4.2. Hence, we do not need to wait for the ANN to be trained to identify the target subnetwork (strong lottery ticket hypothesis in Conjecture 1.3.1).

Additionally, we experiment parallelizing per class neuron importance score computation using a balanced and imbalanced set of images per class. For those experiments, we sampled a random number of images per class (IMIDP), then we took the average of the computed neuron importance scores from solving the MIP on each class. The obtained subnetworks were compared to solving the MIP with 1 image per class (IDP) and to solving the MIP with balanced images representing all classes (SIM). We achieved comparable results in terms of test accuracy and pruning percentage.

Table 4.1. Comparing test accuracy of Lenet-5 on imbalanced independent class by class (IMIDP.), balanced independent (IDP.) and simultaneously all classes (SIM) with 0.01 threshold, and $\lambda = 1$.

	MNIST	Fashion-MNIST
Ref.	$98.8\% \pm 0.09$	$89.5\% \pm 0.3$
IDP.	$98.6\% \pm 0.15$	$87.3\% \pm 0.3$
Prune (%)	$19.8\% \pm 0.18$	$21.8\% \pm 0.5$
IMIDP.	$98.6\% \pm 0.1$	$88\% \pm 0.1$
Prune (%)	$15\% \pm 0.1$	$18.1\% \pm 0.3$
SIM.	$98.4\% \pm 0.3$	$87.9\% \pm 0.1$
Prune (%)	$13.2\% \pm 0.42$	$18.8\% \pm 1.3$

To conclude on the robustness of the scores computed based on the input points used in the MIP, we empirically show in Table 4.1 that our method is scalable, and that class contribution can be decoupled without deteriorating the approximation of neuron scores and thus, the performance of our methodology. Moreover, we show that OAMIP is robust even when an imbalanced number of data points per class (IMIDP) is used in the MIP.

4.3. Comparison to Random and Critical Pruning

We started by training a reference model (REF.) using the training parameters in Section 4.1. After training and evaluating the reference model on the test set, we ran Algorithm 4.1.1. In particular, in step 2, we used as a selection strategy for D', one data point per class, and the tuned thresholds of Table 4.2 selected based on the maximum number of pruned neurons with minimum loss in the ANN's capacity.

Table 4.2. Pruning results on fully connected (FC-3, FC-4) and convolutional (LeNet-5, VGG-16) network architectures using three different datasets. We compare the test accuracy between the reference network (REF.), randomly pruned model (RP.), model pruned based on critical neurons selected by the MIP (CP.) and our non-critical pruning approach with (OAMIP + FT) and without (OAMIP) fine-tuning for 1 epoch. The selected thresholds were tuned to prune the largest number of neurons with marginal loss in accuracy.

		Ref.	RP.	CP.	OAMIP	OAMIP + ft	Prune (%)	threshold
MNIST	FC-3	98.1% ± 0.1	83.6% ± 4.6	44.5% ± 7.2	**95.9% ± 0.87**	**97.8 ± 0.2**	44.5% ± 7.2	0.1
	FC-4	97.9% ± 0.1	77.1% ± 4.8	50% ± 15.8	**96.6% ± 0.4**	**97.6% ± 0.01**	42.9% ± 4.5	0.1
	LeNet-5	98.9% ± 0.1	56.9% ± 36.2	38.6% ± 40.8	**98.7% ± 0.1**	**98.9% ± 0.04**	17.2% ± 2.4	0.2
Fashion-MNIST	FC-3	87.7% ± 0.6	35.3% ± 6.9	11.7% ± 1.2	**80% ± 2.7**	**88.1% ± 0.2**	68% ± 1.4	0.1
	FC-4	88.9% ± 0.1	38.3% ± 4.7	16.6% ± 4.1	**86.9% ± 0.7**	**88% ± 0.03**	60.8% ± 3.2	0.1
	LeNet-5	89.7% ± 0.2	33% ± 24.3	28.6% ± 26.3	**87.7% ± 2.2**	**89.8% ± 0.4**	17.8% ± 2.1	0.2
CIFAR-10	LeNet-5	72.2% ± 0.2	50.1% ± 5.6	27.5% ± 1.7	**67.7% ± 2.2**	**68.6% ± 1.4**	9.9% ± 1.4	0.3
	VGG-16	83.9% ± 0.4	85% ± 0.4	83.3% ± 0.3	N/A	**85.3% ± 0.2**	36% ± 1.1	0.3

In order to validate our pruning policy guided by the computed importance scores (Algorithm 4.1.1), we created different subnetworks of the reference model, where the same number of neurons is removed in each layer, therefore allowing a fair comparison among them. These subnetworks were obtained through different procedures: non-critical (our methodology), critical and randomly pruned neurons. For the VGG-16 experiments, an extra fine-tuning step for 1 epoch is performed on all generated subnetworks. Although we pruned the same number of neurons, which according to (Liu et al., 2018b) should result in similar performances, Table 4.2 shows that pruning non-critical neurons results in marginal loss and gives a better performance.

What happens if we mask critical neurons instead of non-critical neurons? Would the model's accuracy on test set drop after masking critical neurons? We test OAMIP by pruning neurons having top score from each layer, and with the same number of non-critical neurons selected by our Algorithm 4.1.1. In that case, our experiments of Table 4.2 show a significant increase in error from the reference model. The significant increase in the error shows that the selected neurons are critical for the predictive capacity of the model.

What will happen if we fine-tune our pruned model for just 1 epoch? We used this experiment to show that in some cases after fine-tuning for just 1 epoch the model's accuracy can surpass the reference model. Since the MIP is solving its marginal softmax (3.2.2) on true labels, the generated subnetwork, after fine-tuning, can outperform the reference model.

Computing neuron scores on the trained FC-3, FC-4 and LeNet5 generally takes around 10 seconds, and VGG16 with our decoupled approach takes 4 hours if we compute the neurons' importance score sequentially without parallelizing the per layer computation[3].

4.4. Generalization

In this experiment, we train the ANN on a dataset d_1, and we create a subnetwork using OAMIP, i.e., Algorithm 4.1.1, with the thresholds from Table 4.2, and step 2 with 1 image per class. After creating the masked model, we restart it to its original initialization. Finally, the new masked model is re-trained on another dataset d_2, and its generalization is analyzed.

Table 4.3. Cross-dataset generalization: subnetwork masking is computed on source dataset (d_1) and then applied to target dataset (d_2) by retraining with the same early initialization. Test accuracies are presented for masked and unmasked (REF.) networks on d_2, as well as pruning percentage.

Model	Source dataset d_1	Target dataset d_2	REF. Acc.	Masked Acc.	Pruning (%)
LeNet-5	Mnist	Fashion-MNIST	$89.7\% \pm 0.3$	$89.2\% \pm 0.5$	$16.2\% \pm 0.2$
		CIFAR-10	$72.2\% \pm 0.2$	$68.1\% \pm 2.5$	
VGG-16	CIFAR-10	MNIST	$99.1\% \pm 0.1$	$99.4\% \pm 0.1$	$36\% \pm 1.1$
		Fashion-Mnist	$92.3\% \pm 0.4$	$92.1\% \pm 0.6$	

Table 4.3 displays our experiments and respective results. When we compare generalization results to pruning using OAMIP on Fashion-MNIST and CIFAR-10, we observe that computing the critical subnetwork LeNet-5 architecture on MNIST, is creating a more sparse subnetwork with test accuracy better than zero-shot pruning without fine-tuning using OAMIP, and comparable accuracy with the original ANN. This behavior occurs because the solver is optimizing on a batch of images that are classified correctly with high confidence from the trained model. Furthermore, computing the critical VGG-16 subnetwork architecture on CIFAR-10 using decoupled greedy learning (Belilovsky et al., 2019) with significant pruning percentage generalizes well to Fashion-MNIST and MNIST.

4.5. Comparison to SNIP

OAMIP can be viewed as a compression technique of over-parameterized neural models. In what follows, we compare it to the state-of-the-art framework, Single-shot Network Pruning SNIP (Lee et al., 2018).

[3]Computational time relies heavily on the network's architecture.

SNIP computes connection sensitivities in a data-dependent way before the training. The sensitivity of a connection represents its importance based on the influence of the connection on the loss function. After computing the sensitivity, the connections that are below a predefined threshold are pruned before training (single shot). The sensitivity of the connections is computed using a mini-batch of the data. SNIP computes the magnitude of the derivatives of the mini-batch with respect to the loss function. If the magnitude of the derivative is high, then this connection is having an influence on the model's predictive capacity. Thus, using connection sensitivity, one can identify the important connections to that specific task.

In our methodology, we exclusively identify the importance of neurons and essentially prune all the connections of non-important ones. On the other hand, SNIP only focuses on pruning individual connections. Moreover, we highlight that SNIP can only compute connection sensitivity on the initialization of an ANN. Indeed, for a trained ANN, the magnitude of the derivatives with respect to the loss function was optimized during the training, making SNIP more keen to keep all the parameters. On the other hand, OAMIP can work on different convergence levels as shown in Section 4.2. Furthermore, the connection sensitivity computed by SNIP is only network and dataset specific, thus the computed connection sensitivity for a single connection does not give a meaningful signal about its general importance for a given task, but rather, it needs to be compared to the sensitivity of other connections.

In order to bridge the differences between the two methods, and provide a fair comparison in equivalent settings, we make a slight adjustment to our method. In step 4.1.1 of OAMIP, we compute neuron importance scores on the model's initialization[4]. We note that we used only 10 images as an input to the MIP corresponding to 10 different classes, and 128 images as input to SNIP, as in the associated paper (Lee et al., 2018). Our algorithm was able to prune neurons from fully connected and convolutional layers of LeNet-5. After creating the sparse network using both SNIP and our methodology, we trained them on Fashion-MNIST dataset. The difference between SNIP ($88.8\% \pm 0.6$) and our approach ($88.7\% \pm 0.5$) was marginal in terms of test accuracy. SNIP pruned 55% of the ANN's parameters and our approach 58.4%.

Next, in Table 4.4, we show that OAMIP outperforms SNIP in terms of generalization. Here we adjusted SNIP to prune entire neurons based on aggregated (i.e., summed) sensitivity of incoming connections in each neuron, while still applying OAMIP to the initialization of each ANN before training. We note that when OAMIP is applied to the initialization, more neurons are pruned, since the marginal softmax part of the objective discussed in Section 3.2 is weighting less ($\lambda = 1$), driving the optimization to focus on model sparsification.

Finally, we remark that the adjustments made to SNIP and OAMIP in the previous experiments are solely for the purpose of comparison, while (unlike SNIP) the main purpose

[4]Remark: we used $\lambda = 1$ and pruning threshold 0.2 and kept ratio 0.45 for SNIP. Training procedures as in Section 4.1.

Table 4.4. Cross-dataset generalization comparison between SNIP, with neurons having the lowest sum of connections' sensitivity pruned, and OAMIP, both applied on initialization, see Section 4.4 for the generalization experiment description.

Source dataset d_1	Target dataset d_2	REF. Acc.	Method	Masked Acc.	Pruning (%)
Mnist	Fashion-MNIST	$89.7\% \pm 0.3$	SNIP	$85.8\% \pm 1.1$	$53.5\% \pm 1.8$
			OAMIP	$\mathbf{88.5\% \pm 0.3}$	$\mathbf{59.1\% \pm 0.8}$
	CIFAR-10	$72.2\% \pm 0.2$	SNIP	$53.5\% \pm 3.3$	$53.5\% \pm 1.8$
			OAMIP	$\mathbf{63.6\% \pm 1.4}$	$\mathbf{59.1\% \pm 0.8}$

of our method is to allow optimization at any stage – before, during, or after training. In the specific case of optimizing at initialization and discarding entire neurons based on aggregated connection sensitivity, the SNIP approach may have some advantages, notably in scalability for deep architectures. However, it also has some limitations, as previously discussed.

4.6. Summary

Along Section 4.2, we presented the experiments demonstrating the robustness of OAMIP against different input data points to the MIP, and different convergence levels of the trained ANN. These results also highlighted the scalability of our approach to datasets with large number of classes. In Section 4.3, we showed the validity of OAMIP by comparing different pruning strategies based on the computed neuron importance score. Furthermore, in Section 4.4, we empirically showed the generalization capability of our approach by computing the neuron importance score on one dataset, and applying it on another one. Finally, we conducted a comparison between our approach and SNIP showing comparable results but with our approach having the ability to generalize across different datasets, and with a computed neuron importance score as a metric comparable across different architectures.

In summary, we have demonstrated empirically the robustness, validity, scalability, and generalization of OAMIP on different architectures and datasets.

Chapter 5

Decoupled Computation

The most time sensitive step of OAMIP's Algorithm 4.1.1 is the optimization of the MIP. The number of variables and constraints increases with the number of neurons and input data points. While for the latter, we have already shown that it suffices to consider one image per class, the former is intrinsic to the ANN under analysis. In practice, one can mask a small ANN in a matter of seconds due to the reasonable size of the associated MIP. However, if we try to represent large and more realistic ANNs, the computation time would become very large as observed in the problem tackled in (Fischetti and Jo, 2018). To overcome the computational time issue, we propose in this chapter two techniques to decompose the computation of the neuron scores discussed in Chapter 3 in smaller MIPs. In Section 5.1, we decouple the computation of scores per classification class, and then we take the average of the computed neuron importance scores to use afterwards for pruning non-critical neurons. In Section 5.2, we decouple the computation of scores per layer independently using auxiliary networks (Belilovsky et al., 2019).

5.1. Class-wise Decoupling

In this experiment, we show that the neuron importance scores can be approximated by *1)* solving for each class the MIP with only one data point from it, and then *2)* taking the average of the computed scores for each neuron as the final score estimation. Such procedure would speed-up our methodology for problems with numerous classes. We compare the subnetworks obtained through this balanced independent class by class approach (IDP.) and by feeding at once the same data points from all the classes to the MIP (SIM.) on MNIST and Fashion-MNIST using LeNet-5[1].

Table 5.1 expands the results presented in Section 4.2, where we discussed the comparable results between IDP, and SIM, with a small threshold 0.01. However, we can notice a difference between both of them when we use a threshold of 0.1. This difference comes

[1]Experimental setup in Section 4.1

Table 5.1. Comparing balanced independent class by class (IDP.) and simultaneously all classes (SIM.) with different thresholds using LeNet-5.

	MNIST	Fashion-MNIST	threshold
Ref.	$98.8\% \pm 0.09$	$89.5\% \pm 0.3$	
IDP.	$96.6\% \pm 2.4$	$86.81\% \pm 1.2$	0.1
Prune (%)	$28.4\% \pm 1.5\%$	$29.6\% \pm 1.8$	
SIM.	$98.5\% \pm 0.28$	$88.7\% \pm 0.4$	0.1
Prune (%)	$16.5\% \pm 0.5$	$18.9\% \pm 1.4$	
IDP.	$98.6\% \pm 0.15$	$87.3\% \pm 0.3$	0.01
Prune (%)	$19.8\% \pm 0.18$	$21.8\% \pm 0.5$	
SIM.	$98.4\% \pm 0.3$	$87.9\% \pm 0.1$	0.01
Prune (%)	$13.2\% \pm 0.42$	$18.8\% \pm 1.3$	

from the fact that computing neurons' importance score on each class independently zeros out more neurons' importance score resulting in a computed average with more neurons to be pruned. Additionally, solving independently for each class allows parallelizing the computation per class, reducing the solving time needed for datasets with large number of classes.

5.2. Layer-wise Decoupling

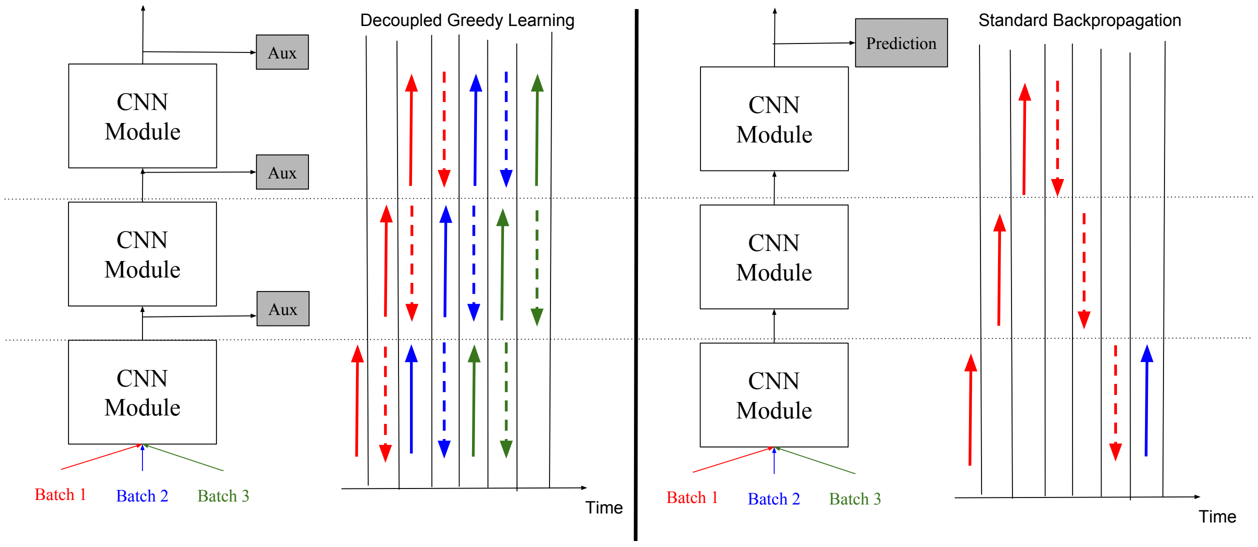


Fig. 5.1. Illustration of the auxiliary network atttached to each sub-module along with the signal backpropagation during training as shown in (Belilovsky et al., 2019)

As mentioned in Section 4.1, we can use decoupled greedy learning (Belilovsky et al., 2019) to parallelize learning of each layer by computing its gradients and using an auxiliary network attached to it as illustrated in Figure 5.1. In decoupled greedy learning, a small auxiliary network is attached to each convolutional layers, and the backpropagation is applied on each layer independently throughout the auxiliary network. By using this procedure, we have auxiliary networks of the deep neural network that represent subsets of layers, thus, allowing us to formulate the MIP for each sub-representation of the neural network. Without decoupling the layers, when we tried to formulate VGG-16 (Simonyan and Zisserman, 2014), the memory footprint was large, and the MIP solver was stuck for hours without giving any initial result. On the other side, when we decoupled the formulation to become a per layer computation using an auxiliary network, the optimization process for VGG-16 took around 4 hours to obtain the optimal solution. Next, we explain the training procedure, the auxiliary network architecture and the MIP's representation.

Training procedure. We start by constructing auxiliary networks for each convolutional layer except for the last convolutional layer that will be attached to the classifier part of the model. During the training, each auxiliary network is optimized with a separate optimizer and the auxiliary network's output is used to predict the backpropagated gradients. Each sub-network's input is the output of the previous subnetwork and the gradients will flow through only the current subnetwork. In order to parallelize this operation, a replay buffer of previous representations should be used to avoid waiting for the output from previous subnetworks during the training.

Auxiliary Network Architecture. We use a spatial averaging operation to construct a scalable auxiliary network applied to the output of the trained layer and to reduce the spatial resolution by a factor of 4, then we apply one 1×1 convolution with batch norm (Ioffe and Szegedy, 2015), followed by a reduction to 2×2, and a one-layer MLP. The architecture used for the auxiliary network is smaller than the one mentioned in the paper (Belilovsky et al., 2019) allowing a speed up in the MIP solving time per layer.

MIP's representation. After training each subnetwork, we create a separate MIP formulation for each auxiliary network using its trained parameters and taking as input the output of the previous subnetwork. This operation can be easily parallelized, and each MIP associated with a subnetwork can be solved independently. Then, we take the computed neuron importance scores per layer and apply them to the main deep neural network. Since these layers were solved independently, we fine-tune the network for one epoch to backpropagate the error across the network. The created subnetwork can be generalized across different datasets and yields marginal loss, as shown in Section 4.4 on VGG-16.

Chapter 6

Conclusion and Future Work

We proposed a mixed-integer linear program to compute the importance score of neurons in an ANN with ReLUs, max/average pooling, batch norm and convolutional layers. The presented MIP formulation is a first step in understanding which components in an ANN are critical for its capacity to perform a given task.

We have shown scalable computation of importance scores by decoupling classes and layers, which allowed us to effectively solve MIP formulations for deep neural architectures. We presented results showing these scores can be effectively used to prune unimportant parts of the network without significantly affecting its main task (e.g., showing small or negligible drop in classification accuracy). Furthermore, our results indicate the automatic construction of efficient subnetworks created using OAMIP, which can be transferred and retrained on different datasets. In conclusion, with OAMIP, we are able to prune subsets of neurons which would make the network more efficient in terms of inference and computation time.

Our pruning technique based on a mixed-integer programming optimization problem can be applied to a wide range of neural networks to create smaller efficient architectures. These architectures can be deployed on resource limited IOT devices to enable a different set of tasks, including, but not limited to, image classification, speech recognition, robotics and control. Furthermore, understanding and quantifying which neuron in a given architecture is contributing to the learning task of the deep neural network will help shed light on internal processing in deep learning, which is typically considered as a black box. Our work dives into the black box of neural networks to reveal which subnetworks are contributing more to the output and to identify the "lottery tickets" that generalize best across different datasets.

Beyond theoretical understanding provided in this thesis, our work may also have an impact on downstream tasks, specifically on the ability to reduce the network's size without losing its predictive accuracy to deploy on IOT devices, as mentioned above. Similar to other studies in the field, misinterpretation and misuse in such downstream applications may in time raise ethical concerns regarding privacy, bias in the model's predictions, etc. However, this work is computational in nature and addresses the foundations of modern deep learning, agnostic of specific downstream tasks. As such, by itself, it is not expected to raise ethical concerns nor to have adverse effects on society.

Several directions on the improvement of OAMIP, and its extensions remain to be explored. Future work should include an analysis to determine the effect of changing the conservative coefficients of the neuron importance score on the number of pruned neurons, shown in Equation (3.1.1). In the current conservative coefficients, we applied a ReLU on the upper bounds, which guarantees the pruning of dead neurons, but might also be restricting the freedom of the neuron importance score, i.e., the power of decreasing neuron activation. Moreover, it would be essential to investigate the sparsity part of the objective function defined in Equation (3.2.1), namely, its simplification (which could impact the MIP solving time), and alternative formulations that could potentially result in pruning more neurons. Finally, devising an objective function that can roughly assess the generalization gap based on the input model and a subset of the training data would be worth investigating.

Besides further analysis of our MIP model, it remains to explore other ways for approximating neurons importance score. Instead of taking the average of per class computation of neurons importance score as discussed in Section 5.1, we could use the Shapley value (Shapley, 1953). In game theory, Shapley value is a method to compute the contribution of each player in a game to achieve a desired outcome. In our case, the game is solving the MIP formulation for each class. The desired outcome is the minimization of the neuron importance scores. The players are the different classes of the dataset we are studying. The Shapley value allows us to distribute a potentially unequal contribution of each class in the outputted neuron importance score as some classes might provide better neuron importance score information than others. The main drawback of using Shapley values is in their computational complexity. Furthermore, instead of computing each neuron importance score, we could study the generalization of our approach to compute the connections' importance score. The connections' importance score would help us to understand in depth which set of features are contributing more to the network's predictive capacity.

In a broader sense, our work contributes for the growing literature on the use of combinatorial optimization within machine learning. In this line, we present two immediate lines of research. One is the use of OAMIP to select which neurons to be trained for newly upcoming task without catastrophically forgetting previous task in a continual learning setting. Another future research direction is to determine whether or not we can assess the network's uncertainty using a MIP representation.

References

Eugene Belilovsky, Michael Eickenberg, and Edouard Oyallon. Decoupled greedy learning of CNNs. *arXiv preprint arXiv:1901.08164*, 2019.

Yoshua Bengio, Ian Goodfellow, and Aaron Courville. *Deep learning*, volume 1. Citeseer, 2017.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.

Behnam Neyshabur, Zhiyuan Li, Srinadh Bhojanapalli, Yann LeCun, and Nathan Srebro. Towards understanding the role of over-parametrization in generalization of neural networks. *arXiv preprint arXiv:1805.12076*, 2018.

Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, and Fahim Kawsar. An early resource characterization of deep learning on wearables, smartphones and internet-of-things devices. In *Proceedings of the 2015 international workshop on internet of things towards applications*, pages 7–12, 2015.

He Li, Kaoru Ota, and Mianxiong Dong. Learning IoT in edge: Deep learning for the internet of things with edge computing. *IEEE network*, 32(1):96–101, 2018.

Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? *arXiv preprint arXiv:2003.03033*, 2020.

Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*, 2017a.

Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990.

Babak Hassibi, David G Stork, and Gregory J Wolff. Optimal brain surgeon and general network pruning. In *IEEE international conference on neural networks*, pages 293–299. IEEE, 1993.

Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages

1135–1143, 2015.

Suraj Srinivas and R Venkatesh Babu. Data-free parameter pruning for deep neural networks. *arXiv preprint arXiv:1507.06149*, 2015.

Xin Dong, Shangyu Chen, and Sinno Pan. Learning to prune deep neural networks via layer-wise optimal brain surgeon. In *Advances in Neural Information Processing Systems*, pages 4857–4867, 2017.

Wenyuan Zeng and Raquel Urtasun. Mlprune: Multi-layer pruning for automated neural network compression. In *International Conference on Learning Representations (ICLR)*, 2018.

Namhoon Lee, Thalaiyasingam Ajanthan, and Philip HS Torr. Snip: Single-shot network pruning based on connection sensitivity. *arXiv preprint arXiv:1810.02340*, 2018.

Chaoqi Wang, Roger Grosse, Sanja Fidler, and Guodong Zhang. Eigendamage: Structured pruning in the kronecker-factored eigenbasis. *arXiv preprint arXiv:1905.05934*, 2019.

Abdullah Salama, Oleksiy Ostapenko, Tassilo Klein, and Moin Nabi. Pruning at a glance: Global neural pruning for model compression. *arXiv preprint arXiv:1912.00200*, 2019.

Thiago Serra, Abhinav Kumar, and Srikumar Ramalingam. Lossless compression of deep neural networks. *arXiv preprint arXiv:2001.00218*, 2020.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

Robert M Gray. Toeplitz and circulant matrices: A review, 2002. *URL http://ee. stanford. edu/˜ gray/toeplitz. pdf*, 2000.

Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Mostafa ElAraby, Guy Wolf, and Margarida Carvalho. Identifying critical neurons in ann architectures using mixed integer programming. *arXiv preprint arXiv:2002.07259*, 2020.

G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. Wiley-Interscience, New York, NY, USA, 1988. ISBN 0-471-82819-X.

Alekse Grigorevich Ivakhnenko and Valentin Grigorevich Lapa. Cybernetics and forecasting techniques. In *North-Holland*, 1967.

Rina Dechter. Learning while searching in constraint-satisfaction problems. In *University of California, Computer Science Department, Cognitive Systems ...*, 1986.

Paul J Werbos. Applications of advances in nonlinear sensitivity analysis. In *System modeling and optimization*, pages 762–770. Springer, 1982.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

Carl Friedrich Gauss. *Theoria motus corporum coelestium in sectionibus conicis solem ambientium*, volume 7. Perthes et Besser, 1809.

Robin L Plackett. Studies in the history of probability and statistics. xxix: The discovery of the method of least squares. *Biometrika*, 59(2):239–251, 1972.

Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

Ken-Ichi Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural networks*, 2(3):183–192, 1989.

George B Dantzig. Programming in a linear structure. In *Bulletin of the American Mathematical Society*, volume 54, pages 1074–1074. AMER MATHEMATICAL SOC 201 CHARLES ST, PROVIDENCE, RI 02940-2213, 1948.

AH Land and AG Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

EML Beale and RE Small. Mixed integer programming by a branch and bound technique. In *Proceedings of the IFIP Congress*, volume 2, pages 450–451, 1965.

Robert J Dakin. A tree-search algorithm for mixed integer programming problems. *The computer journal*, 8(3):250–255, 1965.

AH Land. Doig. ag,"an automatic method for solving discrete programming problems". *Econometrica*, 28(3):497–520, 1960.

Robert E Bixby. A brief history of linear and mixed-integer programming computation. *Documenta Mathematica*, pages 107–121, 2012.

M.l R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0716710447.

Ralph E Gomory. Some polyhedra related to combinatorial problems. *Linear algebra and its applications*, 2(4):451–558, 1969.

Tobias Achterberg. SCIP: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

Matteo Fischetti, Fred Glover, and Andrea Lodi. The feasibility pump. *Mathematical Programming*, 104(1):91–104, 2005.

Fred W Glover and Gary A Kochenberger. *Handbook of metaheuristics*, volume 57. Springer Science & Business Media, 2006.

Emilie Danna, Edward Rothberg, and Claude Le Pape. Exploring relaxation induced neighborhoods to improve MIP solutions. *Mathematical Programming*, 102(1):71–90, 2005.

Matteo Fischetti and Andrea Lodi. Local branching. *Mathematical programming*, 98(1-3): 23–47, 2003.

Babak Hassibi and David G Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in neural information processing systems*, pages 164–171, 1993.

Yves Chauvin. A back-propagation algorithm with optimal use of hidden units. In *Advances in neural information processing systems*, pages 519–526, 1989.

Andreas S Weigend, David E Rumelhart, and Bernardo A Huberman. Generalization by weight-elimination with application to forecasting. In *Advances in neural information processing systems*, pages 875–882, 1991.

Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. Learning important features through propagating activation differences. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3145–3153. JMLR. org, 2017.

Mathias Berglund, Tapani Raiko, and Kyunghyun Cho. Measuring the usefulness of hidden units in boltzmann machines with mutual information. *Neural Networks*, 64:12–18, 2015.

LF Barros and B Weber. Crosstalk proposal: an important astrocyte-to-neuron lactate shuttle couples neuronal activity to glucose utilisation in the brain. *The Journal of physiology*, 596(3):347–350, 2018.

Kairen Liu, Rana Ali Amjad, and Bernhard C Geiger. Understanding individual neuron importance using information theory. *arXiv preprint arXiv:1804.06679*, 2018a.

Ruichi Yu, Ang Li, Chun-Fu Chen, Jui-Hsin Lai, Vlad I Morariu, Xintong Han, Mingfei Gao, Ching-Yung Lin, and Larry S Davis. Nisp: Pruning networks using neuron importance score propagation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9194–9203, 2018.

Sara Hooker, Dumitru Erhan, Pieter-Jan Kindermans, and Been Kim. A benchmark for interpretability methods in deep neural networks. In *Advances in Neural Information Processing Systems*, pages 9734–9745, 2019.

Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

Artur Jordao, Ricardo Kloss, Fernando Yamada, and William Robson Schwartz. Pruning deep neural networks using partial least squares. *arXiv preprint arXiv:1810.07610*, 2018.

Yang He, Guoliang Kang, Xuanyi Dong, Yanwei Fu, and Yi Yang. Soft filter pruning for accelerating deep convolutional neural networks. *arXiv preprint arXiv:1808.06866*, 2018.

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

Corinna Cortes, Xavier Gonzalvo, Vitaly Kuznetsov, Mehryar Mohri, and Scott Yang. Adanet: Adaptive structural learning of artificial neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 874–883. JMLR. org, 2017.

Charles Weill, Javier Gonzalvo, Vitaly Kuznetsov, Scott Yang, Scott Yak, Hanna Mazzawi, Eugen Hotaj, Ghassen Jerfel, Vladimir Macko, Ben Adlam, et al. AdaNet: A scalable and flexible framework for automatically learning ensembles. *arXiv preprint arXiv:1905.00080*, 2019.

Xin He, Kaiyong Zhao, and Xiaowen Chu. AutoML: A survey of the state-of-the-art. *arXiv preprint arXiv:1908.00709*, 2019.

Marc-André Zöller and Marco F Huber. Benchmark and survey of automated machine learning frameworks. *arXiv preprint arXiv:1904.12054*, 2019.

Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

Ari S Morcos, Haonan Yu, Michela Paganini, and Yuandong Tian. One ticket to win them all: generalizing lottery ticket initializations across datasets and optimizers. *arXiv preprint arXiv:1906.02773*, 2019.

Vivek Ramanujan, Mitchell Wortsman, Aniruddha Kembhavi, Ali Farhadi, and Mohammad Rastegari. What's hidden in a randomly weighted neural network? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11893–11902, 2020.

Yulong Wang, Xiaolu Zhang, Lingxi Xie, Jun Zhou, Hang Su, Bo Zhang, and Xiaolin Hu. Pruning from scratch. In *AAAI*, pages 12273–12280, 2020a.

Eran Malach, Gilad Yehudai, Shai Shalev-Shwartz, and Ohad Shamir. Proving the lottery ticket hypothesis: Pruning is all you need. *arXiv preprint arXiv:2002.00585*, 2020.

Chaoqi Wang, Guodong Zhang, and Roger Grosse. Picking winning tickets before training by preserving gradient flow. *arXiv preprint arXiv:2002.07376*, 2020b.

Matteo Fischetti and Jason Jo. Deep neural networks and mixed integer linear optimization. *Constraints*, 23(3):296–309, 2018.

Ross Anderson, Joey Huchette, Christian Tjandraatmadja, and Juan Pablo Vielma. Strong mixed-integer programming formulations for trained neural networks. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 27–42. Springer, 2019.

Vincent Tjeng, Kai Y. Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=HyGIdiRqtm`.

Po-Sen Huang, Robert Stanforth, Johannes Welbl, Chris Dyer, Dani Yogatama, Sven Gowal, Krishnamurthy Dvijotham, and Pushmeet Kohli. Achieving verified robustness to symbol substitutions via interval bound propagation. *arXiv preprint arXiv:1909.01492*, 2019.

Moonkyung Ryu, Yinlam Chow, Ross Anderson, Christian Tjandraatmadja, and Craig Boutilier. CAQL: Continuous action Q-learning. *arXiv preprint arXiv:1909.12397*, 2019.

Martin Mladenov, Craig Boutilier, Dale Schuurmans, Gal Elidan, Ofer Meshi, and Tyler Lu. Approximate linear programming for logistic markov decision processes. In *Proceedings of the Twenty-sixth International Joint Conference on Artificial Intelligence (IJCAI-17)*, pages 2486–2493, Melbourne, Australia, 2017. URL `https://www.ijcai.org/proceedings/2017/346`.

Tom Brosch and Roger Tam. Efficient training of convolutional deep belief networks in the frequency domain for application to high-resolution 2d and 3d images. *Neural computation*, 27(1):211–227, 2015.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. Maximum resilience of artificial neural networks. In *International Symposium on Automated Technology for Verification and Analysis*, pages 251–268. Springer, 2017b.

Ramon E Moore, R Baker Kearfott, and Michael J Cloud. *Introduction to interval analysis*, volume 110. Siam, 2009.

Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2:18, 2010.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

Kevin Gimpel and Noah A Smith. Softmax-margin CRFs: Training log-linear models with cost functions. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 733–736. Association for Computational Linguistics, 2010.

Matthias Ehrgott. *Multicriteria optimization*, volume 491. Springer Science & Business Media, 2005.

Alex Krizhevsky. Learning multiple layers of features from tiny images. Master's thesis, University of Toronto, 2009.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

D Kingma and J Ba. Adam: A method for stochastic optimization in: Proceedings of the 3rd international conference for learning representations (iclr'15). *San Diego*, 2015.

APS Mosek. The mosek optimization software. *Online at http://www. mosek. com*, 54(2-1): 5, 2010.

Akshay Agrawal, Robin Verschueren, Steven Diamond, and Stephen Boyd. A rewriting system for convex optimization problems. *Journal of Control and Decision*, 5(1):42–60, 2018.

Steven Diamond and Stephen Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. In *International Conference on Learning Representations*, 2018b.

Lloyd S Shapley. A value for n-person games. *Contributions to the Theory of Games*, 2(28): 307–317, 1953.