A Gradient-Based Approach to Adaptive Neural Network Pruning

Guilherme Manuel Cruz Pereira
Master in Mathematical Engineering
Department of Mathematics
Faculty of Sciences of the University of Porto
2025

A Gradient-Based Approach to Adaptive Neural Network Pruning

Guilherme Manuel Cruz Pereira

Dissertation carried out as part of the Master
in Mathematical Engineering
Department of Mathematics
2025

Supervisor
João Pedro Pedroso, Associate Professor, Faculty of Sciences of University of Porto

Co-supervisor
Sónia Gouveia, Assistant Researcher, University of Aveiro

“The more you know, the more you realize you don’t know.”

— Aristotle

Gostaria de dedicar este trabalho à Minha Família, à minha Mãe, ao meu Pai e ao meu Irmão pelo apoio incondicional e amor constante. Sem o vosso apoio nada disto seria possível.

Seguidamente, dedico à minha namorada, aos meus colegas de casa e a todos os meus amigos que me acompanharam neste percurso e que o tornaram mais fácil. Obrigado pelo vossa presença e motivação.

Acknowledgements

Gostaria de reconhecer toda a ajuda e orientação científica por parte dos meus orientadores, a Professora Sónia Gouveia e o Professor João Pedro Pedroso.

Resumo

A Inteligência Artificial tornou-se uma força transformadora em diversas áreas do conhecimento. No entanto, as redes neuronais modernas exigem frequentemente recursos computacionais e de memória substanciais, limitando a sua utilização em dispositivos com restrições de hardware. Este trabalho aborda o problema do consumo de memória e redução do tempo de inferência, com o objetivo de compressão de modelos. A intuição subjacente é que a maioria das redes neuronais sofre, em certa medida, de sobreajuste, o que implica que muitos parâmetros são supérfluos. Neste contexto, a técnica de *pruning* (redução de pesos, conhecida como poda) surge como uma abordagem promissora para reduzir o tamanho do modelo e o custo de inferência, mantendo simultaneamente um bom desempenho preditivo.

Para avaliar o potencial e as limitações da poda, é conduzida uma experiência controlada com uma rede neuronal *feedforward* simples, composta por três camadas escondidas e funções de ativação ReLU. O conjunto de dados é gerado sinteticamente a partir de funções seno e cosseno, moduladas por um hiperparâmetro de frequência que permite controlar a complexidade da tarefa——ainda que esta relação seja somente aproximada, uma vez que a complexidade é difícil de quantificar com precisão. Ao variar essa frequência, a experiência permite observar como o desempenho da rede evolui com a dificuldade da tarefa, com a poda a ser avaliada no processo, identificando o seu potencial e limitações.

Com base nesta fundação, é proposto um método de poda diferenciável como resposta as limitações de poda simples, inspirado em [1, 2], no qual tanto o rácio de poda como o parâmetro de temperatura são tratados como variáveis treináveis. Esta abordagem permite otimizar a esparsidade num espaço contínuo, eliminando a necessidade de pesquisa em grelha (*grid search*) e integrando o controlo da esparsidade diretamente no processo de aprendizagem. Para além da avaliação da eficácia da poda, o método investiga também se o nível de esparsidade pode ser tratado como um parâmetro treinável, substituindo assim a afinação manual por uma quantidade aprendida automaticamente durante o treino. O método é validado no conjunto de dados MNIST, fornecendo indícios da sua capacidade de ajustar automaticamente a esparsidade do modelo, mantendo um desempenho competitivo.

Palavras-chave: Matemática, Matemática Aplicada, Aprendizagem Computacional, *Deep Learning*, Redes Neuronais, Compressão de Modelos, Poda, Poda Diferenciável, *Soft Pruning*

Abstract

Artificial Intelligence has become a transformative force across a wide range of disciplines. However, modern neural networks often demand substantial computational and memory resources, which limits their deployment on resource-constrained devices. This work addresses the problem of memory consumption, with the goal of model compression and reducing inference time. The underlying intuition is that most neural networks suffer from some degree of overfitting, implying that many parameters are superfluous. In this context, pruning emerges as a promising technique to reduce model size and inference cost while preserving predictive performance.

To evaluate the potential and limitations of pruning, a controlled experiment is conducted using a simple feedforward neural network with three hidden layers and ReLU activations. The dataset is synthetically generated using sine and cosine functions, modulated by a frequency hyperparameter to control the complexity of the task, even though complexity is difficult to quantify precisely and frequency serves only as an approximate proxy. By varying this frequency, the experiment explores how network performance evolves with task difficulty, while benchmarking simple pruning in the process, assessing its potential and limitations.

Building on this foundation, a differentiable pruning method is proposed as a response to the limitations of simple pruning, inspired by [1, 2], in which both the pruning ratio and the temperature parameter are treated as trainable variables. This enables sparsity to be optimized in a continuous space, removing the need for discrete grid search and integrating pruning into the learning process itself. Beyond evaluating pruning effectiveness, the method also investigates whether the sparsity level can be treated as a trainable parameter, thereby replacing manual tuning with a learnable quantity integrated into the training process. The approach is validated on the MNIST dataset, providing insights into its ability to automatically adjust model sparsity while maintaining competitive performance.

Keywords: Mathematics, Applied Mathematics, Machine Learning, Deep Learning, Neural Networks, Model Compression, Pruning, Differentiable Pruning, Soft Pruning

Table of Contents

List of Figures

Symbols

$\mathcal{L}$	Loss function
Θ	Parameter set
$\mathbf{W}^{\ell}$	Weight matrix for layer ℓ
$\mathbf{b}^{\ell}$	Bias vector for layer ℓ
w_{kj}^{ℓ}	Weight that connects j-th neuron from $(\ell - 1)$-th layer to k-th neuron from ℓ-th layer
b_k^{ℓ}	Bias term for k-th neuron of ℓ-th layer
a_k^{ℓ}	Activation for k-th neuron of ℓ-th layer
$f(\cdot)$	Activation function
η	Learning rate
$m(w)$	Mask function applied to weight w (e.g., binary or soft mask)
t	Pruning threshold
r	Pruning ratio
τ	Temperature hyperparameter
$\sigma(\cdot)$	Sigmoid function (used for soft mask)
Δx	Optimizer variation step for variable x
$F(r)$	Cumulative distribution function
$f_W(w)$	Probability density function
$K(\cdot)$	Kernel function for KDE
h	Bandwidth for KDE
$\delta(\cdot)$	Delta Dirac function
H_{cont}	Histogram with infinite bins
$H_{discrete}$	Histogram with finite bins (normal histogram)
λ	Regularization hyperparameter
R^2	Coefficient of determination
y	Ground truth regression target
$\hat{y}$	Prediction of regression target

$h(x;\alpha)$	Synthetic target function parameterized by α
α	Frequency hyperparameter

1. Introduction

This chapter begins with Section 1.1 by outlining the motivation behind the work, clarifying what is intended with this dissertation. Namely, it is intended to make clear the utility of pruning and why it is necessary in the first place.

Promptly after, some needed theoretical background is established in Section 1.2 to make sure there are no loose ends in our logical development throughout our work. More specifically, a detailed overview of neural networks is provided, followed by an in-depth discussion of pruning methods and their taxonomy, setting the stage for explaining the rationale behind the decisions made throughout this work.

To conclude this chapter, the goals and structure of the Thesis will be discussed in Section 1.3, as a way to take a breather on the theoretical material, while also laying the groundwork for what is to come, all the while making the overall direction clearer. Ultimately, the aim is to clearly state the objectives of this thesis and how they will be addressed throughout.

1.1. Motivation For This Work - Neural Networks and Computational Limitations

As previously mentioned, the size and computational demands of neural networks, while often justified by their performance, can be difficult to manage. Pruning emerges as a technique to eliminate potentially unnecessary components, thereby reducing computational requirements. However, in most pruning methods, the pruning ratio—or compression rate—is typically fixed prior to training. While this may seem reasonable at first glance, it carries a significant caveat.

how does one find out how much to prune? If a pruning ratio appears to work, is it possible to prune even more? And how much?

The short answer is that it is not possible to know the optimal pruning ratio before training. This obligates the use of a grid search or, in other words, trying a whole lot of different pruning rates in order to get an idea of how much can be pruned. Two main drawbacks arise from this requirement. First, conducting a grid search involves training the model n times, where n is the number of pruning ratios under evaluation, as opposed to training it only once. If k-fold cross-validation is also employed, the total number of training procedures increases to $k \times n$. Considering the additional optimization of other hyperparameters, the overall training time can increase drastically. The second drawback is that only a finite set of pruning ratios is tested, which restricts the search to a discrete space. As a result, the optimal pruning ratio likely lies between two of the tested values, but its exact value—and even its existence, cannot be confirmed with certainty.

With this in mind, our method arises as a response to these limitations. What it promises to do, is make it so only one training procedure is needed, and introduces the ratio into a continuous space, allowing the method's final guess to be, in its own estimation, the most optimal.

1.2. Theoretical Background

Before delving into specific methodologies, it is essential to establish the foundational concepts, as they will be addressed in detail later in this work. To that end, we begin by outlining the principles of neural networks and pruning techniques.

1.2.1 Neural Network - An Overview

In order to discuss pruning, it is first necessary to establish a clear understanding of neural networks. These are machine learning models that have evolved into a distinct subfield known as deep learning [3].

Just like any machine learning process, it is nothing more than an optimization problem, in which a loss function $\mathcal{L}$ is introduced as the goal function to be minimized [3],

$$\min_{\Theta} \mathcal{L}(\boldsymbol{x};\Theta) \tag{1.1}$$

Where the set Θ represents the parameters of the model, which are precisely what gets tweaked during the learning process. The parameters are called the weights and the biases, which will be more thoroughly introduced as the structure of neural networks is further developed. More specifically, the set of parameters can be defined as,

$$\Theta = \left\{ \left(W^{(\ell)}, b^{(\ell)} \right) | \ell = 1, 2, ... L \right\} \tag{1.2}$$

To further understand neural networks, it is first necessary to grasp their structure. A neural network, like any machine learning process, starts with inputs and ends with outputs. Everything in between consists of hidden layers. The number of inputs and outputs depends on the specific problem.

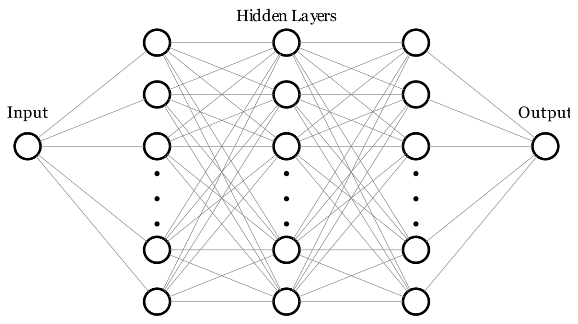


Figure 1.1: Generic Neural Network

These layers consist of units known as neurons, which has led to frequent comparisons with the human brain. However, this analogy is a significant simplification, as each artificial neuron represents only a single numerical value. The neurons are interconnected through elements known as weights and are associated with bias terms that allow for additional adjustments, these constitute the parameters referenced earlier. Fundamentally, the entire structure operates on the principles of linear algebra.

To calculate the output of a neuron, or the activation of a neuron as it is sometimes called, it is necessary to first interpret the diagram in mathematical terms and what that means for the output [3]. Take a simple example, where there is only one layer and only one neuron in the layer.

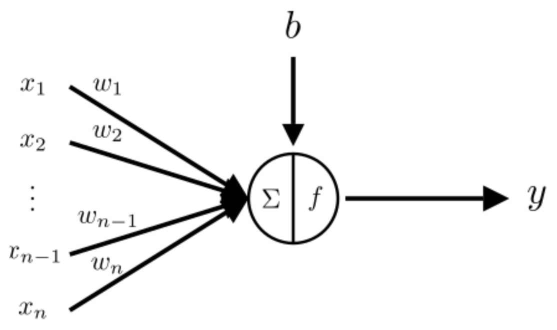


Figure 1.2: Single Layer and Single Neuron Example

This yields the following formula for y,

$$y = f\left(\sum_{k=1}^{n} x_k w_k + b\right) \tag{1.3}$$

where,

- w_k is the k-th weight;
- b is the bias term of the neuron;
- f is the activation function.

Going back to the topic of the parameters, they can now be defined more clearly. The aforementioned matrices in 1.2 can be defined like so,

$$W^{(\ell)} = \begin{bmatrix} w_{11}^{(\ell)} & w_{12}^{(\ell)} & \cdots & w_{1n}^{(\ell)} \\ w_{21}^{(\ell)} & w_{22}^{(\ell)} & \cdots & w_{2n}^{(\ell)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1}^{(\ell)} & w_{m2}^{(\ell)} & \cdots & w_{mn}^{(\ell)} \end{bmatrix} \tag{1.4}$$

Where $W^{[\ell]}$ is the weight matrix relative to the ℓ-th layer. This represents the weight matrix mapping from a layer with n neurons to a layer with m neurons, i.e. $W^{(\ell)} \in \mathbb{R}^{m\times n}$.

$b^{(\ell)}$ is a vector of the form,

$$\mathbf{b}^{(l)} = \begin{bmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_m^{(l)} \end{bmatrix} \tag{1.5}$$

This is because the next layer has m neurons, therefore the bias vector has shape $\mathbb{R}^{m\times 1}$, aligning with the shape of the product of the weight matrix and the output of the neurons in the previous layer $\ell - 1$, which can be defined as the activations of the neurons, $\mathbf{a}^{(\ell-1)} \in \mathbb{R}^{n\times 1}$.

The need for an activation function is more essential than it seems at first [3]. Without it, what is being done is just a really sophisticated linear regression. With this in mind, this detail serves as a welcome way of introducing the needed non-linearity that makes neural networks so special and so widely applicable across virtually any type of problem.

Going a step further, this equation can be generalized for an arbitrary neuron in an arbitrary neural network. Suppose the neuron is the k-th neuron of the ℓ-th layer, where $i \in \{1, \dots, N_\ell\}$ and $\ell \in \{1, \dots, L\}$,

$$a_k^{(\ell)} = f\left(\sum_{j=1}^{N_{\ell-1}} w_{kj}^{(\ell)} a_j^{(\ell-1)} + b_k^{(\ell)}\right) \tag{1.6}$$

where,

- $a_k^{(\ell)}$ is the activation of the k-th neuron in layer ℓ;
- f is the activation function;
- $w_{kj}^{(\ell)}$ is the weight from neuron j in layer $\ell - 1$ to neuron k in layer ℓ;
- $a_j^{(\ell-1)}$ is the activation of the j-th neuron in the previous layer;
- $b_k^{(\ell)}$ is the bias of the k-th neuron in layer ℓ;
- $N_{\ell-1}$ is the number of neurons in the previous layer.

Since the output layer also consists of neurons, the computational workflow of a neuron is now understood. However, this architecture introduces considerable complexity in interpreting what occurs within the network, thereby making the overall problem more difficult to solve. Unlike simpler machine learning methods—such as linear regression—there is no analytical, closed-form solution in this case [4].

With this in mind, the alternative is to instead focus on finding the solution via gradient descent [3]. Making our optimization follow a process similar to the one described below,

$$\Theta^{(n+1)} = \Theta^{(n)} - \eta \nabla_\Theta \mathcal{L} \tag{1.7}$$

Where η is the learning rate, and a hyperparameter of the problem.

This means that the solution to the problem is obtained in an iterative manner. Unlike a closed form solution, which provides the solution straight away, here the solution must be searched for within the parameter space using the gradient. What is implied here, is that this process is not bulletproof, and the solution is at the mercy of getting trapped in local minima. Moreover, unless completely fixed, this is not a deterministic process, in the sense that starting our training process at different starting points might yield significantly different results. In other words, instead of calculating the solution with a formula, this method searches for the most optimal one, with the guidance of the gradient.

With this said, it is not intended to discredit or label Gradient Descent a flawed method. Rather, it serves to highlight the need for care when training a neural network, particularly in the choice of hyperparameters such as the learning rate η.

Gradient Descent, while conceptually simple and effective, can become computationally expensive when applied to large datasets, as it requires calculating the gradient of the loss function across all training samples before each parameter update. This is because the loss function is typically defined as the average loss over the entire dataset, meaning the gradient must reflect the contribution of every individual sample to accurately point in the direction of steepest descent. This full pass over the data slows down training and makes it less responsive to new patterns in the data. To address this, Stochastic Gradient Descent (SGD), in its practical form, introduces the idea of batches. Instead of computing the gradient over the entire dataset, the model uses a small, randomly selected subset of samples (a batch) to estimate the gradient at each step. This drastically reduces computation time per update while still providing a reasonably accurate and unbiased estimate of the true gradient. Though noisier than full-batch updates, SGD strikes a useful balance between speed, memory usage, and learning stability [3, 5].

For the purpose of the practical part of this work and implementation, just plain Stochastic Gradient Descent (SGD) will not be used. Instead, the Adam Optimizer [6] will be the one implemented, since it has become one of the most widely adopted optimizers due to its empirical performance across a broad range of tasks.

Adam, short for Adaptive Moment Estimation, extends SGD by incorporating ideas from both momentum and adaptive learning rate methods [6]. It maintains exponentially decaying averages of past gradients (the “momentum” term) and past squared gradients (to adjust the learning rate for each parameter individually), allowing it to adaptively scale the updates during training. This often leads to faster convergence and makes it less sensitive to the initial learning rate compared to vanilla SGD. For these reasons, Adam is frequently preferred in practice, particularly when training deep or complex networks without requiring extensive tuning of the optimizer’s hyperparameters, since it is more forgiving when picking a value that is slightly off.

Now that the groundwork for neural networks has been laid, the discussion about pruning can begin.

1.2.2 Pruning - Concept and Taxonomy

First and foremost, a strong grasp on the concepts of overfitting and underfitting needs to be established. In machine learning, *overfitting* occurs when a model learns the training data too precisely, including noise or irrelevant patterns, leading to poor generalization on unseen data. This is a particularly common issue in deep learning, as most neural networks tend to suffer from overfitting to some extent due to their high capacity and flexibility. Conversely, *underfitting* happens when the model fails to capture the underlying structure of the data, resulting in poor performance even on the training set. Striking the right balance between these two extremes is critical for building models that perform reliably in practice. This balance is often guided by evaluating training and validation errors, adjusting model complexity, and applying appropriate regularization techniques.

One such technique is *pruning*, which reduces the number of active parameters in a model by removing weights that contribute least to its output. When applied carefully, pruning can act as an implicit form of regularization, helping to prevent overfitting by reducing model capacity while preserving essential structure. However, if overapplied, it may push the model toward underfitting—highlighting the importance of calibrated sparsity in achieving generalization.

As deep neural networks grow in scale and complexity, their computational and memory demands increase significantly. This has prompted a surge of interest in model compression techniques, among which pruning stands out as a particularly promising strategy. Pruning aims to reduce model size and inference cost by removing redundant or unimportant parameters, typically weights or entire structures such as neurons, while trying to preserve the model's performance.

With this potential in mind, it becomes possible to enhance architectures that would otherwise be infeasible due to memory and optimization constraints, all while maintaining network performance. Furthermore, when deploying a network locally on a device with limited memory, such as a smartphone or an embedded system, this appears to be a viable and increasingly explored approach to addressing such limitations [7].

Before introducing the present work, it is necessary to examine possible approaches to pruning. Specifically, it is important to first define the taxonomy of pruning as a foundation for what follows.

Firstly, a definition of the taxonomic axis, which are just the categories that are used to split pruning methodologies, needs to be made in order to facilitate the process. According to [8], pruning can be split like so,

- Specific Speedup
- When to prune
- Pruning Criteria
- Niche Methods

Before going in depth, a visual representation of the taxonomic axis via a tree diagram,

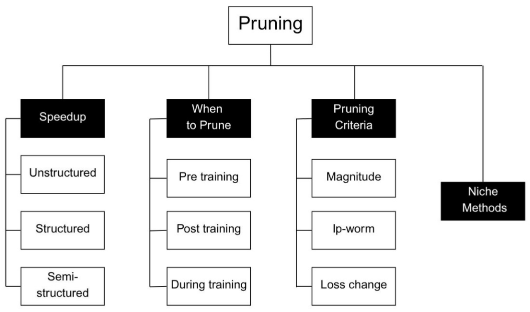


Figure 1.3: Pruning Tree Diagram

For the specific Speedup there is,

- **Structured Pruning**, where whole structures of the network are pruned, like neurons or layers [9].
- **Unstructured Pruning**, is the pruning that is the most fine-grained, in the sense that only weights are pruned from the network [10].

The tradeoff between the two is relatively clear. Unstructured pruning offers greater flexibility, structure wise, it is also more efficient at maintaining performance, whereas structured pruning is more optimizable, since the structure of the pruned weight matrix is easier to tinker with.

Take for example, neuron-level structured pruning, if a neuron is removed, that is the same as zeroing out a row and a column. This predictable sparsity pattern can be easily exploited during computation. On the other hand, the information that is obtained with unstructured pruning is just that the weight matrix is sparse, so optimization is far less straightforward.

A middle ground also exists, though not as prolific, it is called semi-structured pruning. This approach imposes limited, regular sparsity patterns (such as 2:4 or 4:8 sparsity), where only a fixed number of weights are allowed to remain non-zero within small local blocks [11, 12]. This gives a compromise between some flexibility and some needed structure for optimization, offering more granularity than structured pruning, while still enabling some degree of hardware optimization. However, this does not necessarily mean that this method is objectively better than the former ones.

As for when to prune, there are also a few options,

- pruning **Pre Training** refers to techniques that aims to remove weights or structures from the network prior to any learning. The *Lottery Ticket Hypothesis* [13] being one of the most influential works in this category.
- pruning **During Training** is a strategy that prunes the weights throughout the training process. Common regularization techniques, like l_p-norm regularization [3, 14], often constitute the criteria to be considered during training procedures.
- pruning **Post Training** as the name suggests, is just setting the pruning procedure after the training has concluded. Some extra training, or fine tuning, can be implemented after the training is done to recover some performance lost during pruning [15].

Just like the previous category, there are trade-offs between different types of methods. Pruning before training aims to make the model lighter straight away by reducing the number of trainable parameters, resulting in a faster training process and a model that is overall more lightweight. However, the inherent drawback to this approach is that pruning occurs before any learning has taken place—that is, the weights are initialized randomly.

Say there are two exactly identical networks, but they are used for two completely different problems: pre-training pruning would apply the exact same pruning to both of them, with no regard for the actual problem at hand. So, an informed decision is not being made, rather, the capacity of the model is being reduced. This is valid for simple pruning before training. Methodologies such as [13] employ more sophisticated approaches that are more problem-specific.

On the other end of the training spectrum, pruning after training enables the most informed decision possible, as the entire training process has already been completed. This allows for the application of a criterion to the weights in order to identify those that are unnecessary, making use of the maximum amount of available information. A notable drawback, however, is that if the network is large and contains many ultimately redundant weights, it must still be trained in its fully dense form, with every parameter involved. This increases the cost of training and, if fine-tuning is required, introduces an additional training phase.

Meeting both of these methods halfway, there is pruning during training, which analogously to the situation before, is the tradeoff between the strengths and weaknesses of both of the other methods. However, reenforcing the detail, one method is not clearly more optimal than the other, and some methods thrive in situations that others do not.

As for the pruning criteria, there are a plethora of possible options, but the main ones are the following,

- **Magnitude**, the weight is removed based on its absolute value. The intuition is that small weights leverage less contribution to the output, and can therefore be safely removed [15].
- l_p**-norm**, is a generalization of magnitude-based pruning, commonly used to assess the importance of individual weights or groups of weights in a network. As the name implies, one can choose any l_p norm (e.g., l_1, l_2) to compute the norm across a set of weights and use its value as a pruning criterion [14].
- **Loss Change**, here the importance of the weight is based on loss change of the network with and without it. To do so, the first order Taylor expansion is commonly used [16].

The most common and the one that will be talked across this paper is the magnitude based criterion. Basically, as stated, a weight's importance is based on its absolute value. However, in practice, what is needed is a threshold that defines when the weights are small enough to be pruned or big enough to be kept. To do this, a quantile is picked to be more intuitive.

For example, if the network is intended to have a sparsity rate of $r\%$, the r-percentile of the weight magnitudes is taken as the threshold t. After that, a mask can be assigned to each weight to determine whether it is prunable or not, as follows:

$$m(w) = \begin{cases} 1, & \text{if } |w| \geq t \\ 0, & \text{if} |w| < t \end{cases} \tag{1.8}$$

This is called hard pruning, this will be discussed further in 2.1.

1.3. Objectives of this Work and structure of this dissertation

The main aim of this work is to develop a new pruning technique that is able to achieve high sparsity and is also time efficient. On the side, a practical example will be performed, to understand more clearly normal pruning. Let us delve a bit deeper, though.

To carry out the exploratory part of this work, a fixed neural network architecture is defined, with regression performed on an arbitrary function that includes trigonometric components. Within these functions, frequency is introduced as a mutable hyperparameter. The intuition behind this is that the hyperparameter can be increased to a point at which the neural network can no longer approximate the function satisfactorily. This setup helps to better understand the capacity of the models and enables observation of the potential of plain pruning.

As for the transformative part, in which a custom pruning method is developed, this work builds upon an existing paper on soft mask pruning [1]. Following that, the concept of turning hyperparameters—such as the pruning ratio—into trainable parameters is introduced, along with other design considerations. This is not the first attempt at such an approach, as there exists prior work that treats the pruning threshold itself as a trainable parameter [2]. While the method proposed here is not identical, some parallels can be drawn.

The goal, on a conceptual level, is a method that naturally and organically maximizes the sparsity of its own architecture. It is intended that the starting position of the sparsity r does not affect the final value, maybe just making the convergence not as fast. This in turn, would make it so only one training procedure is needed, all the while making the sparsity something that arises from the problem itself and training choices, relieving us of the burden of choosing the sparsity ourselves, and consequently, mitigate the need for a trial and error typically required to find an appropriate value.

There is also a need to emphasize that the method incorporated a during training pruning approach. As previously stated, pruning during training can serve as a tradeoff between pre training and post training, therefore, it was adopted as the pruning strategy employed in this work.

Regarding the structure of this dissertation, the introduction has been presented, and it will be followed by a methods section, in which finer details will be provided, including aspects of implementation and optimization. Specifically, the methods that served as inspiration for the proposed approach will be briefly outlined, followed by the setup of a practical example used to benchmark basic pruning. Only then will the full details of the proposed method be disclosed, including training-related considerations and implementation subtleties. Subsequently, the datasets used for evaluation will be described. Finally, the metrics employed to assess the performance of all networks will be defined and discussed.

Following that, the results will be presented, either supporting or challenging the initial hypotheses. These findings will be summarized and discussed, leading into the conclusion and future work section, where key takeaways will be outlined and potential directions for further research will be proposed.

2. Methods

For this section, what is intended is giving all the necessary details to grasp what goes on under the hood. Namely, the methods from which heavy inspiration was taken will be briefly described, in order to make clear what was used from which one. Immediately after, our practical example to benchmark pruning will be described, followed by the actual method in display in this work. Of course that for the method that was created, some extra detail will be given, like some limitations found in the process and how they were solved.

For the last two subsections, the focus will be more on technical details for testing. The Dataset section was written for the sole purpose of enunciating the datasets that were used throughout the making of this work. As for the last section, the process of finding the best benchmark for the neural networks used will be documented, as well as the final decision for the best performance evaluator in this context.

2.1. PDP - Parameter-Free Differentiable Pruning

The method proposed in [1] laid the groundwork for the concept of soft pruning. The term soft pruning in this context denotes a more flexible mechanism for determining whether a parameter should be pruned. Instead of assigning a binary mask—as is done in hard pruning—this approach applies a sigmoid function, allowing values to lie between the strict 0 and 1 of hard pruning. This creates a continuous spectrum of pruning, enabling more nuanced weight suppression during training.

To illustrate this point, let us consider the simple case of unstructured, magnitude-based pruning. In hard pruning, if a weight w is below the threshold t, it is assigned a mask value of 0, effectively pruning it. If the weight exceeds the threshold, it receives a value of 1, preserving the connection. However, this is a rigid approach: once a weight is pruned, it cannot be recovered during training—even if it would later prove beneficial.

On the other hand, in soft pruning, even if a weight is smaller than the threshold, it is not automatically set to zero. Only when $w \ll t$ does it become approximately zero, and likewise, only when $w \gg t$ does it approach one. This creates a smoother transition between pruned and retained weights. Importantly, this formulation preserves differentiability, which is essential for gradient-based optimization during training. The differences between these pruning techniques can be summarized by the following equations:

$$m(w) = \begin{cases} 1 & \text{if } w \geq t \\ 0 & \text{if } w < t \end{cases} \qquad \text{(2.1, Hard Pruning)}$$

$$m(w) = \sigma\left(\frac{w^2 - t^2}{\tau}\right) = \frac{1}{1 + e^{-\frac{w^2-t^2}{\tau}}} \qquad \text{(2.2, Soft Pruning)}$$

Where the new variable τ is a newly introduced hyperparameter to the training process, it is called the temperature parameter, and it basically controls how steep the transition from 0 to 1. if $\tau \to 0$ that means our temperature freezes and an asymptotic behavior is reached, since we are dividing by τ. In practice, this means that the hard mask profile is recovered and we get a step function. On the other hand, if τ is large relative to the weights, or in the more extreme version, $\tau \to \infty$, this means that we would have a constant function at 0.5. This would, in turn, mean that our function would never know what to do, remaining in a perpetual state of doubt about whether to prune or not.

If we were to plot these functions, it would look something like this

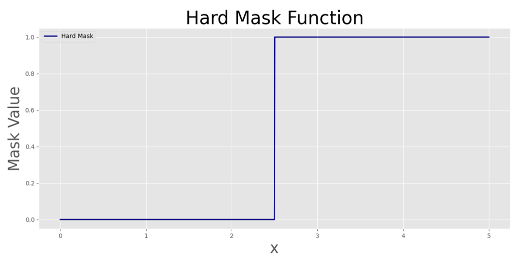


Figure 2.1: Hard mask function

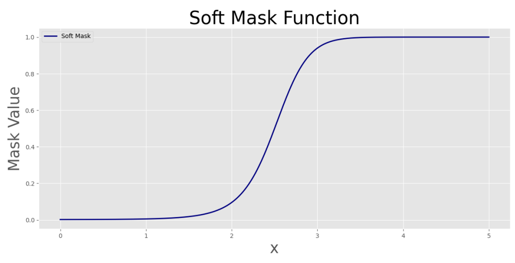


Figure 2.2: Soft mask function

This further illustrates that, whereas hard pruning restricts the decision to two discrete outcomes, soft pruning allows for a continuous range of values that more precisely reflect the relative importance of each weight.

Another important consideration is that the use of a sigmoid function results in a smooth and well-behaved mask, which is easily differentiable. In contrast, employing a hard step function would yield an unstable and ill-conditioned gradient during training. The importance of this differentiability will be made increasingly clear in this and the following sections.

Before proceeding, it is important to clarify where this method fits within the proposed taxonomy. It is a during-training method, and its pruning criterion is based on weight magnitude. Regarding granularity, it can operate in either a structured or unstructured manner. It is also a differentiable method, meaning that additional differentiable elements are introduced into the gradient computation. However, unlike many similar approaches, it does not include additional trainable parameters. The method developed in this work, by contrast, will——this will be discussed in detail in Section 2.4.

Because the method is magnitude based, a threshold t is calculated using the quantile function. As previously stated, every weight is passed through the mask function $m(w)$(2.2, Soft Pruning). After that, the value of the mask tells us how worthy the weight is of being kept. This ultimately changes the output, and ends up affecting the gradient in the following way. Take a random weight w,

$$\Delta w = m(w)\Delta\hat{w} + \frac{2w}{\tau}m(w)(1-m(w))\Delta\hat{w} \tag{2.3}$$

Here, $\hat{w} = m(w) \cdot w$ represents the weight after applying soft pruning, and $\Delta\hat{w}$ denotes the gradient of the loss with respect to $\hat{w}$, i.e., $\frac{\partial\mathcal{L}}{\partial\hat{w}}$, as computed by PyTorch's automatic differentiation system. Similarly, Δw refers to the effective gradient with respect to the original parameter w, which is the quantity used in the optimizer update.

This notation is a deliberate simplification that will be used consistently throughout this work, where Δx denotes both the gradient of the loss with respect to parameter x and, by extension, the parameter's variation in the optimizer step.

To reach this expression, the chain rule is required to add on the extra partial derivatives to the conversation. Since $\hat{w} = m(w)w$, only one extra derivative appears for the computation of the gradient of each weight,

$$\frac{\partial\mathcal{L}}{\partial w} = \frac{\partial\hat{w}}{\partial w}\frac{\partial\mathcal{L}}{\partial\hat{w}} \tag{2.4}$$

all there is left to do is calculate $\frac{\partial\hat{w}}{\partial w}$ and the gradient expression 2.3 is recovered, with the detail that, to facilitate calculations, the property of the sigmoid function, $\frac{d}{dx}\sigma(x) = \sigma(x)(1-\sigma(x))$, can be used.

Also, looking at the gradient, one interesting detail arises, if the weight is undecided and is 0.5, then $m(w)(1-m(w))$ will be maximized. However, if the weight has already made up its mind, being 1 or 0, then that expression will be zero. This shines light on the fact that this method, though soft, incentivizes a choice by default.

2.2. LTP - Learned Threshold Pruning

The Learned Threshold method [2], like PDP, adopts a soft mask pruning approach. A key distinction is that this method closely aligns with the objective of the present work by introducing learnable parameters—specifically, a pruning threshold for each layer. This makes it a magnitude-based, layer-wise pruning approach, where each layer of the Neural Network (NN) has its own threshold.

The threshold can be denoted as t_l, where l refers to the l-th layer. In other words, our set of parameters is now changed to the following,

$$\Theta = \left\{ \left(W^{(\ell)}, b^{(\ell)}, t^{(\ell)} \right) | \ell = 1, 2, ...L \right\} \tag{2.5}$$

Now, if there are any ambitions to add this to the gradient calculations, the actual gradient with respect to $t^{(\ell)}$ must be derived. By following the same reasoning used in the PDP 2.1 method, the following formula is reached,

$$\frac{\partial \mathcal{L}}{\partial t^{(\ell)}} = \sum_k \frac{\partial \hat{w}_{kl}}{\partial t^{(\ell)}} \frac{\partial \mathcal{L}}{\partial \hat{w}_{kl}} \tag{2.6}$$

Here, the summation over k accounts for the contribution of all weights in the ℓ-th layer to the gradient. The formula for the gradient with respect to the weights is identical to that used in the PDP method, with the key distinction that each layer has its own threshold $t^{(\ell)}$, which is treated as a trainable parameter throughout training.

It is also worth noting the use of two indices to label the weights. This reflects the layer-wise formulation, since the gradient is computed over all weights within a given layer, both the position of the weight within the layer (k) and the layer index (ℓ) must be explicitly included.

If the partial derivatives are worked out in 2.6, the following expression is extracted,

$$\Delta t^{(\ell)} = -\sum_k \frac{w_{kl}}{\tau} m(w_{kl})(1 - m(w_{kl}))\Delta \hat{w}_{kl} \tag{2.7}$$

And thus, the set of parameters $\{t^{(\ell)} | \ell = 1, ..., L\}$ can be implemented into our optimizer.

2.3. Practical Example

This part of the work will be more of the revision type, all the while also having an experimental element to it.

The objective here is, essentially, to simulate, in as much of a controlled environment as possible, a problem that can be solved using a neural network. For this, we will be doing regression on a function of our choosing, where a frequency parameter controls how complex the function is. This function is an expression that was tinkered until an appropriate profile was obtained. The function will be further described in section 3.1.

As for the actual NN used here, a somewhat simple architecture was used, as a way to have a powerful model, but also have a model small enough that would allow grid searches in a manageable time. For the computer power available throughout this work, this translates to a neural network with 1 input and 1 output, and 3 hidden layers with 32 neurons each.

One crucial detail worth highlighting is that, since this experiment is meant to be conducted in a controlled setting, training must not be left to uncontrolled randomness. By default, training deep models, especially on GPUs, is a non-deterministic process. This means that even with the same exact model and the exact same hyperparameters, different runs can yield noticeably different results. This variability stems not only from weight initialization and data shuffling but also from GPU-level optimizations, which often employ nondeterministic algorithms for performance.

To ensure reproducibility and consistency across runs, a fixed random seed was used throughout the experiments. While enabling determinism can slightly reduce training efficiency in some cases, the benefits far outweigh the drawbacks in a controlled study. This setup also simplified the experimental process, as only the random seed needed to be changed between runs, allowing us to assess training variability while keeping the data split fixed.

An important detail is the choice to fix the data split, even though k-fold cross-validation is more commonly used. Since the dataset used in this work is synthetically generated from a known deterministic function, it contains no measurement noise or sampling uncertainty. As a result, the typical motivation for performing k-fold cross-validation, assessing generalization across different data splits, does not apply in the same way. Every point in the dataset is a direct and noiseless evaluation of the same underlying function. Consequently, a single fixed train/validation split is sufficient to evaluate the model's ability to approximate the function.

Instead of assessing variation across data splits, our focus shifts toward evaluating the training process itself — specifically, its sensitivity to stochastic factors such as random weight initialization, data shuffling, and the inherent randomness of certain GPU operations. These factors can lead to different optimization trajectories and model outcomes, even when the training data and hyperparameters remain fixed.

To quantify this sensitivity, we kept the data split constant and ran multiple training iterations, each with a different random seed. This allowed us to measure the robustness and stability of the learning process under controlled sources of randomness.

The thing to benchmark will be simple, unstructured magnitude-based pruning, as it represents the most intuitive starting point. The main emphasis will be on post-training pruning, where the network is first trained fully and then pruned. However, for comparison, pre-training pruning will also be evaluated. This will serve to confirm what intuition already suggests, that plain pruning before training tends to yield worse results, due to the lack of information available at that early stage.

2.4. PDP + LTP - Learned Ratio Pruning (LRP)

The core contribution of this work is the development of a novel pruning method designed to address the limitations of plain pruning while improving upon several existing differentiable pruning approaches. The proposed formulation is grounded in two principal lines of prior work [1, 2].

To add on to this, the already discussed, soft mask approach will be used, which makes the pruning criteria more flexible. Also, the pruning ratio r and the temperature parameter τ, will be introduced into our optimizer. As of now, all the extra parameters are global, meaning they aren't layer wise and are applied to the neural network as a whole.

One important detail about the soft pruning approach is, even though it is not as harsh as hard pruning and weights can be brought back from the pruned, such flexibility is no longer desirable once training is complete. To fully realize the memory and computational benefits of pruning, the soft mask formulation cannot be retained after training. While soft masks scale weights to values between 0 and 1, all weights remain technically present in the network. Therefore, upon completion of training, the soft pruning approach is discarded: all weights with mask values below the pruning threshold are permanently set to zero, and those above the threshold are retained. This is equivalent to setting each mask value below 0.5 to 0 and each mask value above 0.5 to 1.

With this approach, the goal is to develop a method that eliminates the need for grid search over pruning-related hyperparameters, thereby reducing the time required to identify an effective training configuration. Furthermore, the aim is to transform the pruning ratio from an arbitrary, manually selected value into a problem-dependent quantity that emerges naturally from the training process.

2.4.1 Calculating the Extra Gradients

Just like it was done for the previous work, the extra gradients for the extra parameters need to be calculated. Namely, $\frac{\partial \mathcal{L}}{\partial r}$ and $\frac{\partial \mathcal{L}}{\partial \tau}$ need to be extracted along with the new gradients for the weights, $\frac{\partial \mathcal{L}}{\partial w_{kl}}$. Doing so yields the following equations,

$$\frac{\partial \mathcal{L}}{\partial \tau} = \sum_k \frac{\partial \hat{w}_k}{\partial \tau} \frac{\partial \mathcal{L}}{\partial \hat{w}_k} \tag{2.8}$$

and,

$$\frac{\partial \mathcal{L}}{\partial r} = \sum_k \frac{\partial t}{\partial r} \frac{\partial \hat{w}_k}{\partial t} \frac{\partial \mathcal{L}}{\partial \hat{w}_k} \tag{2.9}$$

The chain rule for the gradient with respect to the weights is identical to the one in PDP 2.4.

Just like before, the extra partial derivatives need to be calculated. Performing the derivations, the following result is obtained,

$$\Delta \tau = -\sum_k w_k \frac{w_k^2 - t^2}{\tau^2} m(w_k)(1 - m(w_k)) \Delta \hat{w}_k \tag{2.10}$$

and for r,

$$\Delta r = -\frac{\partial t}{\partial r} \sum_k \frac{2 w_k t}{\tau} m(w_k)(1 - m(w_k)) \Delta \hat{w}_k \tag{2.11}$$

Looking at this, there is clearly an elephant in the room - Why was the gradient $\frac{\partial t}{\partial r}$ not calculated?

The analytical answer is not immediately obvious, but it is attainable. However, it soon became evident during the course of the calculation that, several hidden complexities were inadvertently introduced, requiring a more careful and extended analysis than initially expected.

2.4.2 Estimation of the Pruning Threshold Derivative

The problem with $\frac{\partial t}{\partial r}$ is that $t(r)$ is the quantile function for a given distribution of weights, which at first glance does not seem to have an analytical solution, but with some manipulation, a closed-form expression can be derived through appropriate manipulation.

If we define the Cumulative Distribution Function (CDF) as $F(w)$, then the quantile function can be written as,

$$t(r) = F^{-1}(r) \tag{2.12}$$

Since they're inverse functions, it follows that,

$$F(t(r)) = r \tag{2.13}$$

Deriving both sides with respect to r,

$$\frac{d}{dr}F(t(r)) = 1 \tag{2.14}$$

Applying the chain rule,

$$\frac{dt}{dr}\frac{d}{dt}F(t) = 1 \tag{2.15}$$

By definition, the cumulative distribution function is given by

$$F(t) = \int_{-\infty}^{t} f_W(x)\,dx,$$

where $f_W(x)$ is the probability density function (PDF) of the random variable W. By the Fundamental Theorem of Calculus, the derivative of F with respect to t is

$$\frac{dF}{dt} = f_W(t).$$

Substituting this into the previous expression yields:

$$\frac{dt}{dr} = \frac{1}{f_W(t(r))} \tag{2.16}$$

Note that previously, f was used to denote the activation function, however, here it is merely the Probability Density Function (PDF).

Also, in the context of this derivation, total derivatives are used rather than partial derivatives. This is because the cumulative distribution function F and the quantile function t are treated here as univariate functions with a single input variable r. As such, their derivatives with respect to r or t are total by definition. In contrast, within the broader context of the proposed method—where gradients are computed over a high-dimensional parameter space involving the entire neural network—partial derivatives are used to reflect the dependence of the loss function on multiple parameters simultaneously. Thus, the choice of derivative notation reflects the difference in dimensionality and functional dependence in each setting.

This provides an analytical expression for the previously problematic derivative. However, in a realistic NN training scenario, the distribution of the weights is not trivial, let alone having an explicit form for its PDF.

The philosophy of scientific programming is that, at times, it is preferable to obtain an estimate of a value in exchange for improved computational efficiency. While finding an exact analytical form for the PDF may be desirable in theory, it is not necessarily required in practice. With this in mind, the method chosen to estimate $f_W(w)$ was a sufficiently accurate approximation for the purpose of this work, namely Kernel Density Estimation (KDE).

KDE is a method to estimate density functions [17], it has the following form,

$$\hat{f}_W(w) = \frac{1}{nh}\sum_{i=1}^{n} K\left(\frac{w - W_i}{h}\right) \tag{2.17}$$

Where,

- $K(\cdot)$ is the kernel, a non-negative function to be chosen, usually a Gaussian;
- W_i is an observation of the weight dataset, with the sum, the whole dataset is being iterated through;
- h is the bandwidth, basically, it dictates how narrow our kernel is.

Both the kernel and bandwidth, are arbitrary choices of the user, however, the choice of the bandwidth affects the precision far more than the kernel [17].

Essentially, this involves summing multiple kernels, each centered on the corresponding observation. This summation across the dataset gives a new distribution - which is normalized by the nh factor - that, if everything goes as intended, provides an estimation, $\hat{f}_W(\cdot)$, that is worth our while.

The real challenge here is, in order to get a reliable estimation, picking the adequate bandwidth, since too small a bandwidth and the function will look more like a histogram, meaning the distribution is overfitted and too spiky, and too big of a bandwidth will oversmooth the data and lose structure, yielding something uninteresting, with no value to the estimation.

Since the distribution of the weights changes throughout the training process, guessing for the optimal bandwidth is not ideal. So, to combat this, a heuristic was picked to estimate the best bandwidth. There are two main candidates for this, Scott's rule [18] and Silverman's rule [17].

For Silverman's the rule is the following,

$$h = 0.9 \cdot \min\left(\sigma, \frac{\text{IQR}}{1.34}\right) \cdot n^{-1/5} \tag{2.18}$$

- σ: Standard deviation of the sample.
- IQR: Interquartile range of the sample (i.e., $Q_3 - Q_1$, the difference between the 75th and 25th percentiles).
- n: Number of data points in the sample.

The one chosen, though, was Scott's rule, which has the following expression,

$$h_{Scott} = \sigma \cdot n^{-1/(d+4)} \tag{2.19}$$

Where the only new variable is d, the dimensionality. For the purpose of this work, the dimensionality is always 1, as our weights are always scalars, not vectors.

However, advocating for a trade-off between accuracy and computational efficiency while simultaneously relying on KDE may appear contradictory. As is well known to practitioners familiar with KDE, this method has a time complexity of $\mathcal{O}(nm)$ [18], where n is the number of data points and m the number of evaluation points. As a result, this approach to estimating the PDF——and, by extension, computing the derivative——is not inherently scalable. Since the number of weights grows rapidly with model size, and this computation is repeated during every backward pass, the overall computational cost becomes significant. Therefore, KDE offers little computational advantage in this setting.

2.4.3 Optimization via Kernel Density Estimation and FFT

Given the aforementioned critiques of KDE, one might reasonably ask: why is it described in such detail in this work? Could there be a viable alternative?

Using the Fast Fourier Transform (FFT), a much better time complexity can be achieved by taking advantage of the algorithm [18, 19].

Theorem 2.1. *Let $\{x_i\}_{i=1}^n \subset \mathbb{R}$ be a dataset and let $K_h(x) = \frac{1}{h}K\left(\frac{x}{h}\right)$ be a kernel function and bandwidth $h > 0$. Suppose the evaluation domain is discretized on a uniform grid, like in a histogram. Then the kernel density estimate*

$$\hat{f}_X(x) = \frac{1}{n}\sum_{i=1}^{n} K_h(x - x_i)$$

can be rewritten as a discrete convolution and efficiently computed via the Fast Fourier Transform (FFT):

$$\hat{f}_X \approx \mathit{FFT}^{-1}\left(\mathit{FFT}[\mathbf{K}] \cdot \mathit{FFT}[\mathbf{H}]\right),$$

where $\mathbf{K}$ is the discretized kernel, $\mathbf{H}$ is the binned histogram of the data, and $\cdot$ denotes pointwise multiplication.

Proof. we can define our dataset in the following way, it can be though of as a histogram that has infinite bins, where $\Delta x \to 0$,

$$H_{cont}(x) = \frac{1}{n}\sum_{i=1}^{n}\delta(x - x_i)$$

Using the property of the delta Dirac [20], $\int f(x)\delta(x - x_0)dx = f(x_0)$, we can define the KDE expression as follows,

$$\hat{f}(x) = (H_{cont} * K_h)(x) = \frac{1}{n}\sum_{i=1}^{n} K_n(x - x_i)$$

But performing an infinite histogram is not at all feasible, so we must resort to a discrete approximation. Let's define a uniform grid over a finite interval $[a, b]$

Let $\Delta x = \frac{b-a}{m}$ be the bin width, and m the size of the grid space.

Then, the set $\mathcal{G} = \{x_j = a + j\Delta x | j = 1, ..., m\}$ is our set of grid points. This makes it so each bin is of the form $B_j = [x_j - \frac{\Delta x}{2}, x_j + \frac{\Delta x}{2})$.

With this said, the discrete histogram is defined as,

$$H_{discrete}(x_j) = \frac{1}{n}\sum_{j=1}^{m} \mathbb{1}_{B_j}(x_j)$$

Where,

$$\mathbb{1}_{B_j}(x) = \begin{cases} 1 & \text{if } x \in B_j \\ 0 & \text{if } x \notin B_j \end{cases}$$

Using this information, and putting our datapoints into the bins we created, an approximation of the KDE can be made,

$$\hat{f}_X(x) = \frac{1}{n}\sum_{i=1}^{n} K_h(x - x_i) \approx \frac{1}{n}\sum_{j=1}^{m} H_{discrete}(x_j)K_h(x - x_j)$$

This expression is, by definition, a discrete Fourier Transform. This is exactly what is intended, and can be written in the form,

$$\hat{f}(x) \approx (\mathbf{K} * \mathbf{H})(x).$$

As long as the number of bins is chosen to be compatible with the FFT algorithm—specifically, $m = 2^k$, with $k \in \mathbb{N}$—performance is improved, since the FFT operates most efficiently on input sizes that are powers of two [21].

$\square$

This approach reduces the time complexity to $\mathcal{O}(n + m \log m)$ [19], since binning n data points into a histogram takes $\mathcal{O}(n)$ time [18], and computing the FFT over m bins requires $\mathcal{O}(m \log m)$ time [21]. This represents a significant improvement over directly computing the KDE, which has a time complexity of $\mathcal{O}(nm)$, as stated before.

Another relevant detail is that m is picked by the user and not something that naturally scales with size like n, this meaning that the size of the NN can increase, but the resolution of the FFT can remain the same.

This is not to say that the resolution of the grid should not increase with the number of weights, because if the approximation is to remain truthful to the reality, then it must do so. However, it grows at a much smaller rate. In our experiments, for a NN with approximately 700,000 weights, a resolution of 1024 is more than enough, which is a much smaller order of magnitude than the weights.

This paired up with the fact that it also has a much more favorable time complexity, FFT-based estimation of PDF is a highly efficient alternative.

With all of this, the algorithm for the method can be summed up with the following pseudocode,

Algorithm 1 Learned Ratio Pruning (LRP)

```
1: **Input:** Dataset $\mathcal{D}$, initial ratio $r_0$, temperature $\tau_0$
2: Initialize parameters: $\mathbf{r} \leftarrow r_0$, $\tau \leftarrow \tau_0$
3: sort initial weights and compute threshold $t$
4: **while** not converged **do**
5:     **for all** minibatch $(x, y) \in \mathcal{D}$ **do**
6:         Compute soft masks: $m_{ij} \leftarrow \frac{e^{w_{ij}^2/\tau}}{e^{w_{ij}^2/\tau} + e^{t^2/\tau}}$
7:         Forward pass with masked weights
8:         Compute loss $\mathcal{L}$
9:         **if** Using Regularization **then**
10:            $\mathcal{L} = \mathcal{L} + \lambda(r - 1)^2$
11:        **end if**
12:        Compute gradients using the automatic differentiation graph
13:        Apply the FFT KDE algorithm to estimate PDF
14:        Compute new gradients for the weights, $\tau$ and $r$
15:        Update parameters using optimizers
16:        Update sorted weights and threshold $t$
17:    **end for**
18: **end while**
19: Apply hard masks: $m_{ij} \leftarrow \mathbb{1}[m_{ij} \geq 0.5]$
```

2.5. Training tuning - Learning Rates and Regularization

With all the setup said and done, training optimization also needs to be covered, as the method will not work properly if the hyperparameters are not carefully picked.

For the method under discussion, Learned Ratio Pruning (LRP), there are more hyperparameters to consider compared to standard model training. In addition to the usual learning rate for weights and biases, separate learning rates must be defined for the ratio r and the temperature τ. Although r and τ are updated simultaneously during training, they likely operate on different scales, which justifies assigning them distinct learning rates. Furthermore, like all other parameters, r and τ require initial values. Therefore, the initial values r_0 and τ_0 must be selected at the start of training.

For τ, a certain behavior is expected. What is intended, is for τ to start off as a relatively high value, in order for the sigmoid function, σ, to resemble a more constant function around 0.5, this allows the weights to freely explore their place in the beginning of training, without the repercussion of being pruned, since the mask value will hardly be anything close to 0. This warrants a more stable early stage of training. As for the progression, it is expected for the temperature to go down and “freeze” the weights in place, in terms of if they are to be pruned or not. This is because if $\tau \to 0$, the sigmoid function will be more like a step function, and that will make it so there is a smoother transition during training between the stable and permissive soft pruning to the decisive and better for optimization hard pruning. This is not to say, that this needs to happen for the method to work, however, it is a good indicator that everything is going according to plan.

To ensure that the initial value of τ_0 is appropriately scaled, it is not assigned directly. Instead, the user-specified value is interpreted as a scaling factor that is multiplied by the standard deviation of the corresponding weight set. Since this method is, as of now, a global method and not layer wise, this weight set corresponds to all the weights of the network. This will make it a lot easier to place τ into a favorable scale.

One final detail regarding τ is that, due to its proximity to zero, even minor instabilities in the gradient can cause it to become negative, an undesirable outcome, as this would invert the sigmoid function and corrupt the mask values, flipping them in the process. So to combat this and to promote training stability, τ is clamped to a minimum value of 1e-6. This works fine, because at that point the softmax already resembles a hard mask binary function, so there is no need for τ to be smaller anyway.

In contrast, the initial value of r_0 does not require any adaptive scaling. Since r is defined over a fixed and known interval $[0, 1]$, its scale is inherently interpretable and consistent across different settings. This makes it possible to assign a meaningful value to r_0 directly, without needing to normalize it based on the weight distribution.

Another important detail, is that, since a lot of sigmoid functions are being calculated for the masks (2.2, Soft Pruning), a lot of exponents are being computed, which ends up being fine, since it always yields a value between 0 and 1. However, if one of the exponents amidst the calculations explodes to a value that goes beyond the computer precision, then the mask will just be imprecise, or worse, it diverges and there is no value at all. To combat this, the argument of the exponents can be shifted.

to make it more intuitive, the soft max function is written like so,

$$m(w) = \frac{e^{\frac{w^2}{\tau}}}{e^{\frac{w^2}{\tau}} + e^{\frac{t^2}{\tau}}} \tag{2.20}$$

If training with this equation or (2.2, Soft Pruning), it is easy for something to overflow, not because of w^2 or t^2 or their difference, but because of τ, since the temperature can never be zero, but it can definitely approach it. To prevent such a thing, the scaling factor is introduced,

$$\theta = \max\left(\frac{w^2}{\tau}, \frac{t^2}{\tau}\right) \tag{2.21}$$

Multiplying $\frac{e^{-\theta}}{e^{-\theta}}$, which is just 1, to 2.20

$$m(w) = \frac{e^{\frac{w^2}{\tau}-\theta}}{e^{\frac{w^2}{\tau}-\theta} + e^{\frac{t^2}{\tau}-\theta}} \tag{2.22}$$

This ensures no exponent explodes and yields the exact same result, but with much more stable numbers.

The parameter r is not particularly problematic from a computational standpoint, but it does require careful consideration during training. When optimizing solely with respect to the primary loss function $\mathcal{L}$, the value of r may remain low and stable—yet this may not lead to the desired level of sparsity. In other words, if training begins with a conservative (low) value for r, there is no motivation for the model to increase it, as long as a low loss is achieved. After all, why would the model risk a higher value for r if it does not have a motive for it? This occurs because there is no inherent incentive within $\mathcal{L}$ for r to increase, so long as the model achieves a low loss, the optimization process has no reason to alter r.

To account for this, inspired by the general principles of regularization in neural networks [3], a custom loss term was devised with the following expression,

$$\mathcal{L}_R = \lambda(1-r)^2 \tag{2.23}$$

Say $\mathcal{L}$ is the original loss function for the problem, then the total loss to be minimized by the parameters will be,

$$\mathcal{L}_T = \mathcal{L} + \mathcal{L}_R \tag{2.24}$$

Here, λ is a scaling factor applied to the regularization term and serves as another hyperparameter in the optimization problem. It dictates the relative importance of the regularization objective. If λ is too large, the optimization process may focus disproportionately on adjusting r; if too small, r may be effectively ignored. Ideally, λ should be on the same order of magnitude as the original loss function. However, that is dependent on the problem and there really is no clear heuristic for that effect.

As for the expression, it is zero when $r = 1$ and maximum when $r = 0$. This is like telling the model that the target for r is 1. Though, this is clearly not feasible, because if $r = 1$ then all the weights will be pruned and there will be no model, and the loss will skyrocket. However, this makes it so the training process tries to find a tradeoff between high sparsity and good model performance, which is exactly what is intended with this methodology.

As a result, this formulation ensures that r, even if initialized at a conservative low value, will increase during training toward the maximum feasible value dictated by the task and the chosen hyperparameters. This balancing act between sparsity and performance is precisely the behavior the method aims to induce.

As with most deep learning methods, selecting the right hyperparameters for this approach requires some trial and error. This reinforces that the method is not a fully automatic solution that determines the optimal sparsity level on its own, rather, it follows the standard practice of requiring tuning to achieve good performance. With appropriate tuning, the method is expected to produce sparse models that can approach the performance of dense models, even under high sparsity conditions.

2.6. Performance Evaluators and Benchmarks

Model benchmarking was approached with careful consideration. Given that the task involves both regression and classification problems, distinct evaluation metrics were employed to appropriately assess performance in each case. For the classification task, evaluation was more straightforward, as Top-1 accuracy was used. This metric simply reflects the percentage of times the model's most confident prediction matches the true label, making it both simple to compute and easy to interpret.

In the case of regression, the evaluation process was more nuanced, as there is no widely accepted universal metric analogous to accuracy in classification tasks. Initially, only the loss function itself was considered—specifically, the Mean Square Error (MSE). However, this quickly proved insufficient, as MSE is highly scale-dependent. In other words, it provides an absolute measure of error, making comparisons across models or datasets difficult without normalization.

The objective, then, was to identify a metric that would provide both a useful and meaningful assessment of model performance in the regression setting. The first metric considered was the Root Mean Square Error (RMSE). Although it is often preferred over MSE due to being expressed in the same units as the target variable, in this context, where the data is synthetic and unitless, this advantage is irrelevant. More importantly, RMSE, like MSE, remains a scale-dependent, absolute error metric. This limits its usefulness for general comparison or evaluation. Consequently, an alternative approach was adopted using the decibel (dB) scale, which offers a relative measure and is computed as follows [22]:

$$\mathrm{dB} = 10 \cdot \log_{10}\left(\frac{\mathcal{L}}{\mathcal{L}_{ref}}\right) \tag{2.25}$$

Where $\mathcal{L}_{ref}$ is a reference value for the loss. However, in the absence of a meaningful reference, as is the case here, the resulting scale lacks interpretability. For testing purposes, the reference was set to 1, but since this value has no meaningful value, it defeats the purpose of using a relative scale. As such, the transformed values do not tell the whole story of model performance.

Finally, the third metric considered was the coefficient of determination, commonly referred to as the R^2 score.In the case of simple linear regression, it corresponds to the squared Pearson correlation coefficient between predicted and true values. It is defined as follows [23, 24]:

$$R^2 = 1 - \frac{\text{MSE}}{Var(y)} \tag{2.26}$$

Where,

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.27}$$

and

$$\text{Var}(y) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \bar{y})^2 \tag{2.28}$$

Note that this is the **biased estimator** of variance (dividing by n rather than $n-1$), which is the convention adopted by [24] and was consequently used in this work for computing the R^2 score.

As for y_i and $\hat{y}_i$, they are the real value and the predicted one, respectively.

This expression is great, because it is relative, which as stated, makes the metric not scale dependent. What this means in this context is that an R^2 of 1 is the absolute maximum, and would mean the best possible outcome that a model can achieve for this problem, and an R^2 of 0 would be just as good as a random guess.

These results stem from the fact that this R^2 expression is for linear regressions. And how does that tie with the regression of the problem?

The regression that is implicitly being assumed here is $y(\hat{y}) = \hat{y}$. This is just stating that they are equal, and means that if a regression were to be done, then the slope would be 1 and the y-intercept 0.

With all this said and done, the metric that will be used for regression will merely be R^2.

Also, as stated earlier in Sec. 2.3, the experiment will be repeated multiple times. That is, the results of all runs will not be individually plotted. Instead, the mean performance with a standard deviation window will be shown, along with the best and worst models from the experiments.

3. Datasets

3.1. Simulated Datasets - Practical Example

For the practical example, as stated, the dataset was simulated, rather than taken from somewhere else. A custom function was designed, and its construction involved iterative tuning to achieve a desirable balance between localized and global structure in the output, suitable for evaluating pruning effects.

$$h(x;\alpha) = \sin(\alpha x) \cdot \left[\frac{1}{1+e^{-\frac{x+0.2}{\tau}}} - \frac{1}{1+e^{-\frac{x-0.2}{\tau}}} \right] \cdot e^{-x^2} + x^3 \cdot \cos(\alpha x) \tag{3.1}$$

In this expression, x is the input variable, and $h(x;\alpha)$ is the resulting output. The parameter τ controls the smoothness of the sigmoidal window centered at the origin. This τ is unrelated to the parameter τ used in LRP. The parameter α, on the other hand, determines the frequency of the oscillatory components and, as intended by design, controls the complexity of the problem. Although complexity is not something that can be quantified, adjusting α effectively increases the number of local variations.

The intention behind this function is not to introduce multiple mathematical components for the sake of complexity, but rather to construct a regression target that is inherently rich and whose difficulty can be systematically increased. The trigonometric functions serve to introduce and control the frequency content of the signal, thereby allowing for the modulation of task complexity via the parameter α.

The function comprises two main terms. The first term is concentrated near the origin, this because of e^{-x^2}, and the sigmoidal window. This term captures localized high-frequency oscillations and allows for precise control over their spatial extent. The second term, $x^3 \cdot \cos(\alpha x)$, introduces a form of polynomial drift that ensures the overall function is not strictly periodic. This design choice intentionally breaks regularity in the signal, making it less predictable and thus more challenging to approximate.

To avoid numerical instability or overflow during training, a min-max normalization was applied to the output of the function. For good measure, the input domain was rescaled to the interval $[0,1]$, ensuring consistency between input and output ranges. As a result, we have $x \in [0,1]$ and $h(x) \in [0,1]$ throughout the dataset. Just to be clear though, the values that were passed through the function were in the range $[-1,1]$, but for normalization' sake, the points were assigned to a set of inputs with the same size but range $[0,1]$. Finally, in an initial point, the output was at the range $[0,1]$, the loss values for some frequencies were in the order of magnitude of 10^{-5}, which even though there are no guarantees that it could affect the precision of calculations, the output was multiplied by 10 for good measure and turned those losses to the order of magnitude of 10^{-3}.

To simulate the dataset, the parameter $\tau = 0.05$ was fixed throughout all experiments. This value was selected empirically during initial testing, as it provided the wanted smoothness from the sigmoid window.

For simplicity and interpretability, only integer values of the frequency parameter α were considered. When $\alpha = 1$, the function exhibits a relatively smooth and slowly varying pattern:

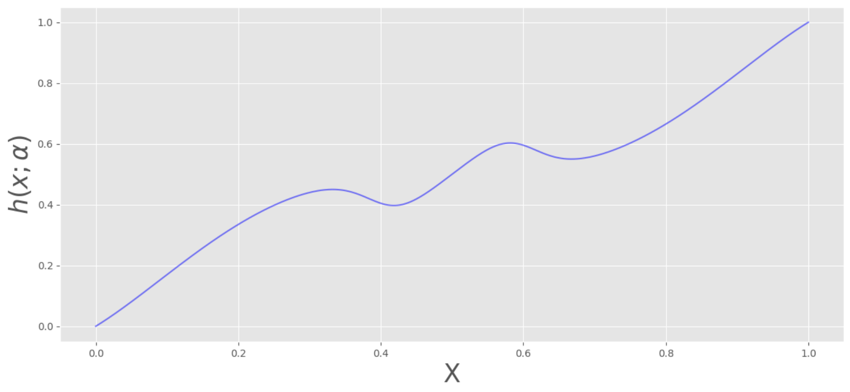


Figure 3.1: Simulated dataset for $\alpha = 1$

Such a function is expected to be well within the capacity of an overparameterized neural network to approximate, even under moderate pruning levels. However, increasing the frequency parameter to $\alpha = 25$ introduces significantly more rapid oscillations,

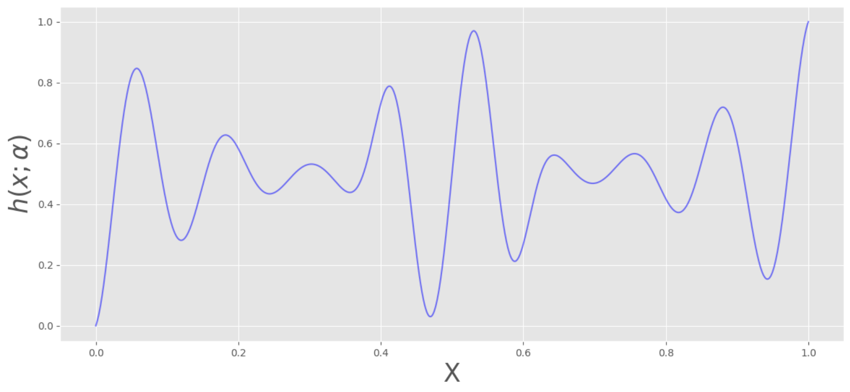


Figure 3.2: Simulated dataset for $\alpha = 25$

Although the function remains continuous and differentiable, its increased frequency content presents a more challenging regression target. This setting is expected to be more sensitive to pruning effects, as it requires the network to retain finer-grained representational capacity to capture the high-frequency variations.

3.2. Existing Datasets - MNIST

The Modified National Institute of Standards and Technology (MNIST) dataset is a widely used benchmark in the field of machine learning and computer vision. It was introduced by Yann LeCun, Corinna Cortes, and Christopher J.C. Burges [25], and has since become a canonical dataset for testing and comparing classification algorithms. The dataset consists of grayscale images of handwritten digits ranging from 0 to 9. Each image is normalized and centered in a 28×28 pixel bounding box, resulting in a total of 784 pixels per image. These pixel values range from 0 (black) to 255 (white), although, as in most practical machine learning pipelines, were normalized to $[0,1]$ range for numerical stability and convergence benefits.

The full dataset contains 70,000 labeled images, divided into two disjoint subsets: 60,000 images are designated for training purposes, and the remaining 10,000 form the standard test set. For this work, a train/validation split of 80/20 was chosen, which results in 48,000 train images, and the remaining 12,000 are used for validation throughout training. The batch size selected for the optimizer was 32. The labels are integers corresponding to the digit present in each image, and the distribution of digits is approximately uniform, avoiding severe class imbalance. This makes MNIST a clean and balanced dataset, albeit a relatively easy one by modern standards.

Despite its simplicity, MNIST remains a widely used benchmark for testing novel algorithms and architectures due to its well-understood structure and low computational cost. Given that the method is in an early stage of development, MNIST provides an ideal environment for rapid prototyping and initial validation. However, any promising results will need to be validated on more challenging datasets to ensure generalizability.

To accommodate the architecture that will be used, each 28×28 image will be reshaped into a 1D vector of size 784. This is necessary because fully connected networks expect flat vector inputs. If a convolutional neural network (CNN) were used instead, this step would not be required, as CNNs can process multidimensional inputs directly, using the discrete convolution operation.

Labels are often converted into one-hot encoded vectors for compatibility with classification loss functions like cross-entropy. More specifically, to put something in one-hot encoding is to represent a categorical variable as a binary vector in which only the index corresponding to the correct class is set to 1, and all other entries are set to 0. The resulting vector has a length equal to the total number of classes.

With these conditions in mind, for the NN architecture used for this portion of the work, there will be an input of 784 neurons for the 1D vector, two hidden layers of 512 neurons, and finally an output composed of 10 neurons due to the one-hot encoding. As for the optimizer, as stated, the Adam optimizer will be used, and for the loss function, cross-entropy was chosen, as it is commonly used for classification tasks [3],

$$\mathcal{L} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{C} y_{n,i}\log(\hat{y}_{n,i}) \tag{3.2}$$

where:

- N is the number of samples in the batch,
- C is the number of classes,
- $y_{n,i}$ is the ground-truth label for sample n and class i ,
- $\hat{y}_{n,i}$ is the predicted probability for sample n and class i, typically obtained via a softmax function [3] .

Despite its utility, MNIST has some limitations. Its computationally flexible ways act as a double edge sword, since, as stated, it has a low computational cost but that also makes it lack complexity. Thus, while it is useful for prototyping and educational purposes, results on MNIST are not indicative of model performance on more challenging or realistic tasks. That said, it is a valid dataset and approach, but, this serves to emphasize, that yielding good results on this dataset is definitely a good indicator, however, further testing on more challenging datasets and contexts is needed to completely validate the method.

Figure 3.3 shows representative samples from the MNIST dataset, illustrating the handwritten digits and the typical variability in writing style among different individuals.

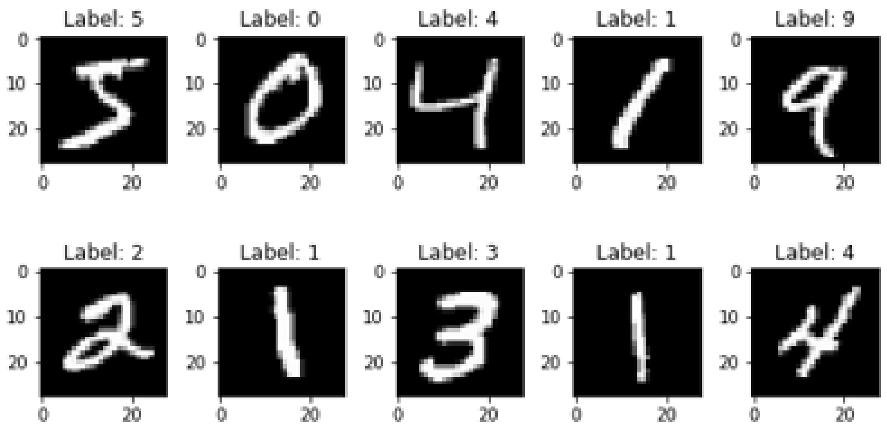


Figure 3.3: Sample images from the MNIST dataset, showing handwritten digits from 0 to 9.

4. Results

Now that the necessary context has been established, this chapter presents the experimental results. Section 4.1 discusses the outcomes of the previously described practical example, providing insight into the potential and limitations of basic pruning techniques. Subsequently, Section 4.2 reports the results of the proposed method evaluated on the MNIST dataset, along with a comparative analysis against other existing pruning approaches applied to the same dataset.

4.1. Practical Example

To begin, rather than testing arbitrary frequency components at random pruning rates, we first investigate how well the network performs for each individual frequency. To do this, rather than working on the entire dataset, it is better to start on just a batch and overfit it for 1000 epochs, this is a standard machine learning practice. The intuition behind it is straightforward, if a model cannot nearly perfectly memorize (i.e., overfit) a small subset of the data, then it likely lacks the capacity to handle the full dataset effectively. With this in mind, the aforementioned method in 2.3 of replicating the experiment on different seeds was employed, and the following graphs 4.1 were obtained. Note that for all graphs involving repetition across different random seeds, including the ones presented here, each experiment was conducted five times.

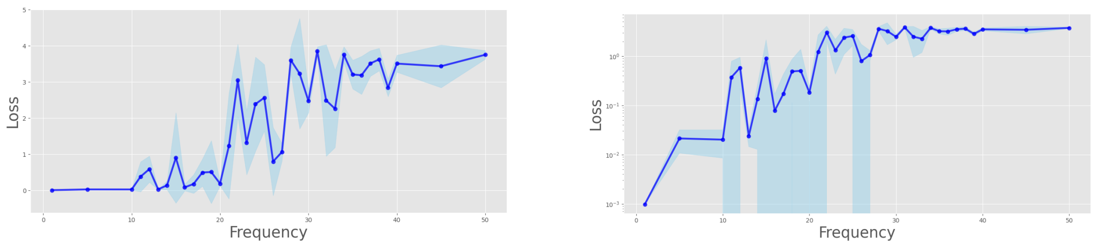


Figure 4.1: Overfitted Batch Loss per Frequency. *Left:* Absolute Scale, *Right:* Log Scale

Using the other benchmark, R^2, the following graph is obtained, fig 4.2,

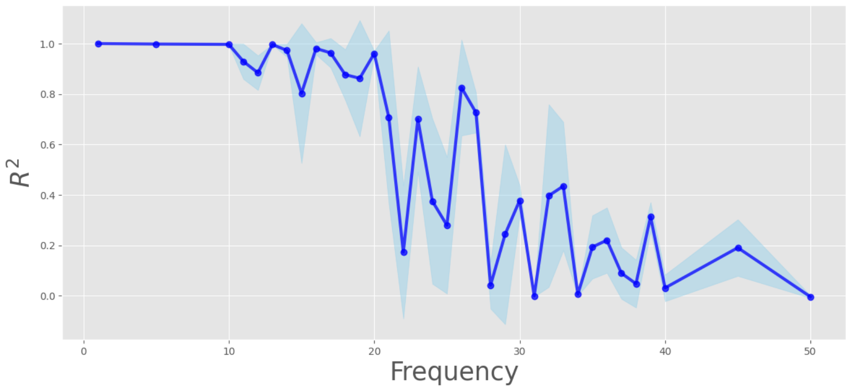


Figure 4.2: Overfitted Batch R^2 per Frequency

As stated before, this warrants the needed confidence that the subjective loss scale could not provide. Even though it also gave a pretty good idea of the profile of the loss on different frequencies, this graph ultimately serves as the decisive piece of evidence to support the conclusions drawn.

It becomes somewhat evident—albeit not entirely objective—that the model's performance begins to deteriorate shortly after $\alpha = 20$. This observation suggests that the transition from $\alpha = 20$ to $\alpha = 21$, while far from linear, may mark a critical threshold beyond which the model no longer possesses sufficient capacity to fit the dataset. The case of $\alpha = 22$ also appears to fall within this transitional range, already exhibiting noticeably degraded performance. In light of this, a more detailed analysis of different pruning rates for these frequencies is now warranted.

Here only R^2 will be used, since it is the more conclusive benchmark.

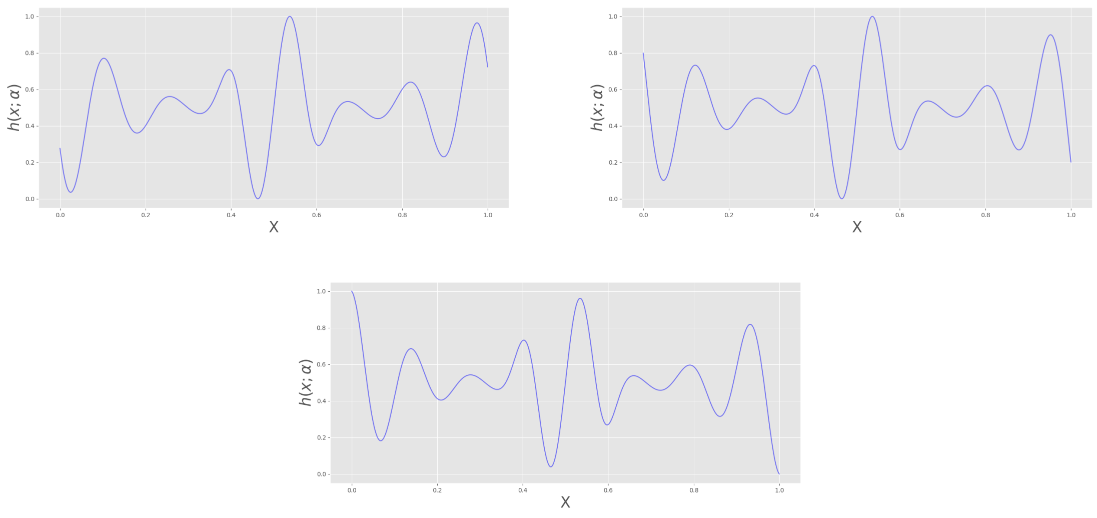


Figure 4.3: Dataset for $\alpha = 20, 21, 21$

At first sight, these seem like seemingly equal problems, in 4.3, with only slight differences, but if we try to prune them over different ratios, doing post training, unstructured empirical pruning 4.4, they do not seem like the same problem anymore, which makes evident that it is hard to have an intuition for the optimal pruning rate.

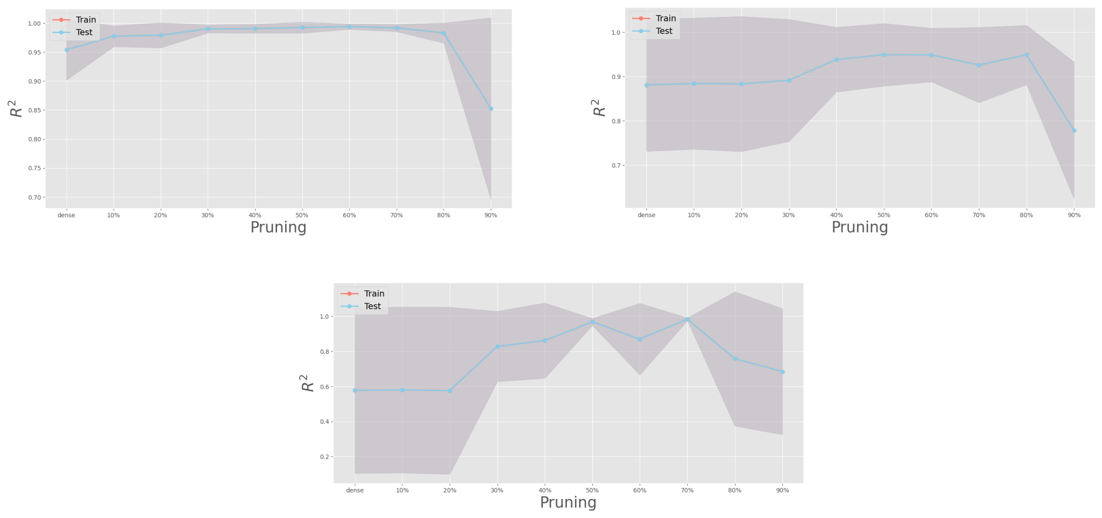


Figure 4.4: R^2 Mean Score per Pruning Ratio, Post Training pruning results. *Left:* $\alpha = 20$; *Right:* $\alpha = 21$; *Bottom:* $\alpha = 22$

The minmax scores can also be plotted 4.5, or in other words, the minimum score of all the 5 experiments can be plotted along with the maximum score. This is not better or worse than the mean score per se, but it is also informative how do the worst and the best model did,

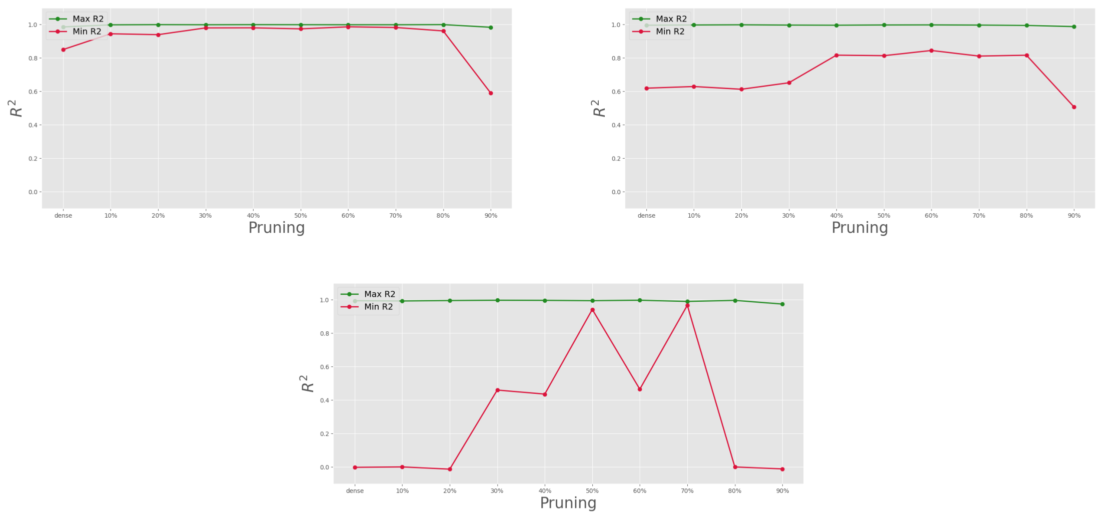


Figure 4.5: R^2 Minmax Score per Pruning Ratio, Post Training pruning results. *Left:* $\alpha = 20$; *Right:* $\alpha = 21$; *Bottom:* $\alpha = 22$

Looking at all the side by side comparisons in fig 4.5 and fig 4.4, it's not that the differences are day and night, but they are definitely noticeable, this means that, very similar problems yield significantly different results. So, even if pruning has a lot of compression potential, it is hard to develop an intuition as to when it is ok to prune. Not to mention that for a relatively simple architecture, to get reliable results (i.e. by repeating the experiment) and to have an idea of what to expect in the ratio space by doing a grid search, the computational power needed is way to high.

Pre-training pruning was also tested using the same visualizations. However, the results for $\alpha = 22$ were found to be very similar to those for $\alpha = 21$, making their inclusion redundant. Therefore, only the results for $\alpha = 21$ and $\alpha = 20$ are presented.

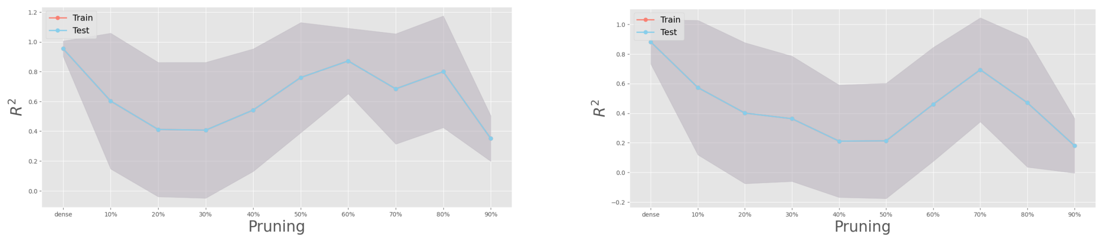


Figure 4.6: R^2 Mean Score per Pruning Ratio, Pre Training pruning results. *Left:* $\alpha = 20$; *Right:* $\alpha = 21$

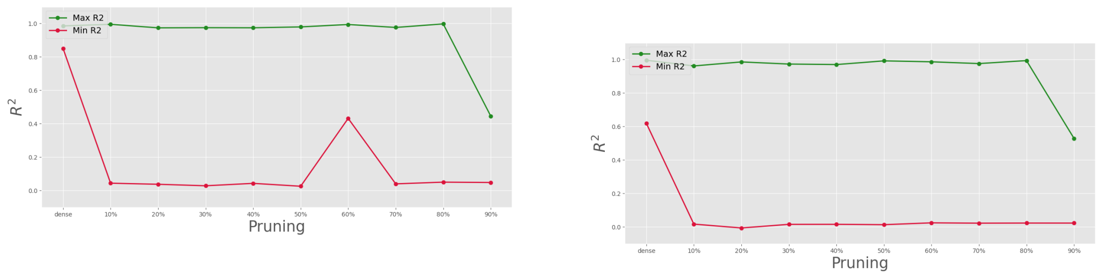


Figure 4.7: R^2 Minmax Score per Pruning Ratio, Pre Training pruning results. *Left:* $\alpha = 20$; *Right:* $\alpha = 21$

These, fig 4.6 and fig 4.7, confirm the suspicions about pre training pruning, as these graphs showcase a much weaker performance overall. But one thing cannot go without saying, this is just plain empirical pruning, and it is not representative of every pre training pruning methodology. Also, it is interesting to see in the minmax plots 4.7, that a model that was pruned before training, can be a great model as it can just as easily be the same as a random guess. This highlights that, in pre-training pruning, the key challenge lies less in defining a pruning criterion and more in identifying a suitable initialization or configuration for the pruned network. This alludes to the ideas discussed in [13].

4.2. LRP on MNIST Dataset

To start off the results for the method, it would be a good practice to see how the model would do on its own without the developed pruning methodology. For all the experiments and training, a batch size of 32 was used,

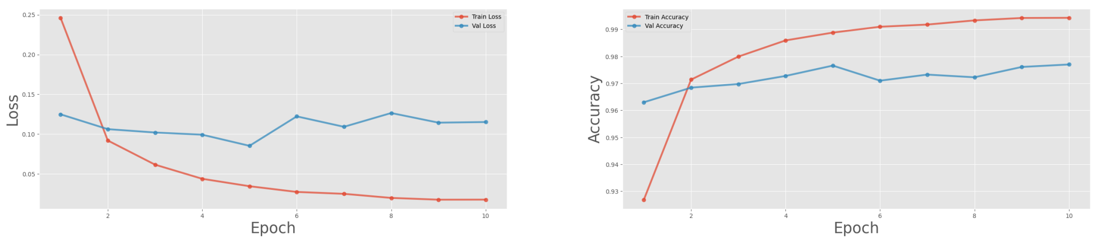


Figure 4.8: Training performance of a dense model. *Left:* Loss per epoch; *Right:* Accuracy per epoch.

For the test set, the model obtained a loss of **0.0927** and an accuracy of **97.9%**, compared to **0.0178** loss and **99.4%** accuracy on the training set and **0.160** loss and **96.8%** accuracy on the validation set. While this reflects strong fitting to the training data, the validation and test performance are slightly below. This discrepancy may indicate a really mild case of overfitting, or simply reflect the limitations of the unregularized dense architecture used. As no regularization methods were applied—such as pruning—this model serves as a baseline to evaluate the effectiveness of the proposed pruning methodology in improving generalization and reducing model size.

What follows is the evaluation of the proposed method. Since the results obtained without regularization were considered relevant, they are also included for comparison. The model was trained for 25 epochs, a configuration that will be maintained in all subsequent experiments. Regarding the choice of hyperparameters, after a process of trial and error, the following values were selected, table 4.1:

r_0	$\tau_0(\times std(W))$	l_r	l_τ
0.5	1e-2	1e-4	1e-6

Table 4.1: Hyperparameters chosen for LRP

Where W is the entire set of all the weights in the whole network.

	Train	**Validation**	**Test**
Loss	0.00752	0.131	0.156
Accuracy	99.8%	97.9%	97.6%

Table 4.2: Performance Metrics for LRP With no Regularization and $r_0 = 0.5$

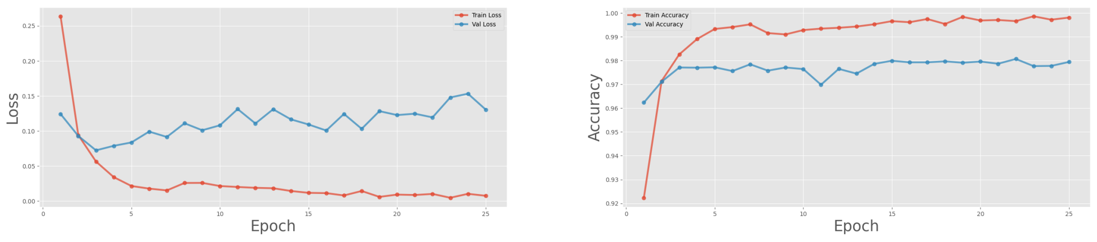


Figure 4.9: LRP With no Regularization and $r_0 = 0.5$. *Left:* Loss; *Right:* Accuracy

It seems not much has changed in fig 4.9 and table 4.2, which becomes more apparent when looking at the progress of the extra parameters 4.10.

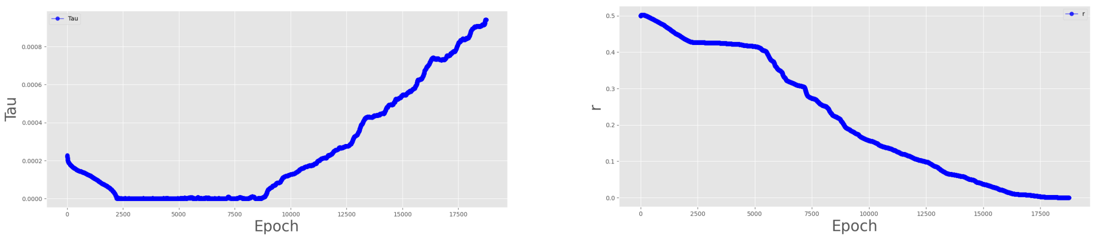


Figure 4.10: LRP With no Regularization and $r_0 = 0.5$. *Left:* τ; *Right:* r

The parameter r not only failed to converge but rapidly approached zero. This behavior is undesirable and supports concerns regarding the absence of regularization. Consequently, the current approach remains unsatisfactory.

As for the parameter τ, it appears to behave contrary to its intended role. While it initially decreases as expected, it eventually begins to increase and continues to do so throughout training. This behavior is detrimental to the pruning process, because with a sufficiently large τ, the sigmoid function becomes too flat, yielding mask values predominantly above 0.5 for weights near zero—resulting in minimal or no pruning. In this particular case, it led to a complete absence of pruning.

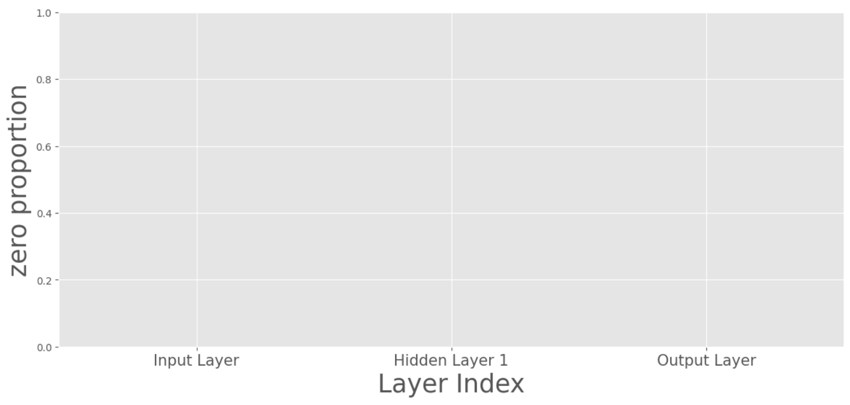


Figure 4.11: Bar plot of sparsity for all layers for LRP With no Regularization and $r_0 = 0.5$

The fig 4.11 translates to a total sparsity of exactly 0%, which is far from the intended outcome. If the model remains fully dense by the end of training, then the pruning method has effectively failed to induce sparsity, adding unnecessary complexity to the training process. Although the final performance matched that of the dense baseline, the training took more than twice as long. As a result, the increased computational cost yielded no practical benefit, rendering the method ineffective in this case.

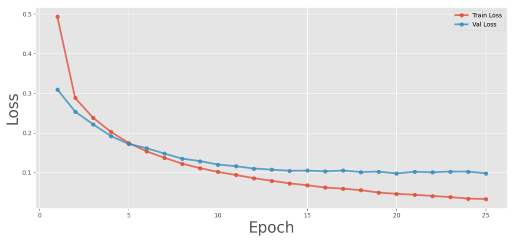
Another important detail to get out of the way, why was the sparsity calculated instead of just looking at r? Although the parameter r is designed to control the target sparsity level, the actual sparsity achieved by the network is not guaranteed to be exactly equal to r. This is because r is used to compute a global threshold t, which indirectly influences the soft masks applied to the weights. The masks themselves are continuous values in the interval $[0, 1]$, and the final pruning decision is made by thresholding these masks at 0.5 to obtain binary masks. As a result, small shifts in weight values or mask activations can lead to sparsity levels that deviate from the intended r, especially during training when the weights and thresholds are still evolving. Therefore, monitoring the realized sparsity separately from r is more precise for understanding the actual compression behavior of the model.

With this said, a comparable performance to the dense model while being also dense is not at all satisfactory. With some testing, $r_0 = 0.95$, with the rest of the hyperparameters remaining unchanged, was the first value found where r did not just plainly go down, fig 4.12(d).

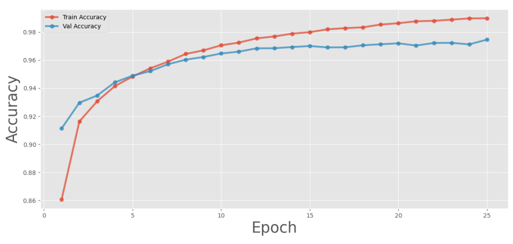
	Train	**Validation**	**Test**
Loss	0.0329	0.0978	0.0964
Accuracy	98.9%	97.5%	97.2%

Table 4.3: Performance Metrics for LRP With no Regularization and $r_0 = 0.95$

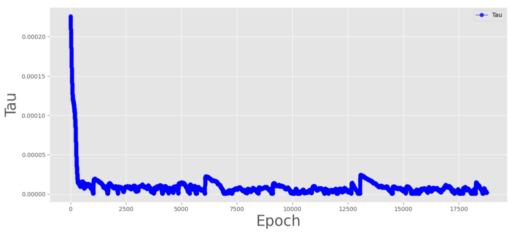
These results are somewhat better, fig 4.12(a,b) table 4.3, the gap between training and validation is smaller, and the sparsity is not 0%, namely one of 86.65%. However, it seems as if r is plummeting down, as can be seen in fig 4.12(d). This suggests that, given the opportunity, and training for even more epochs, either r would go all the way down to zero or it go to a lower value when it could be bigger. Moreover, having to search for a r_0 that does not make its value plummet to zero during the chosen epochs, or until model convergence, kind of defeats the purpose of not wanting to do a grid search, as this motivates the user to do so.



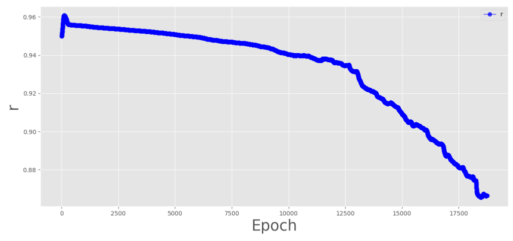
(a) Loss per epoch



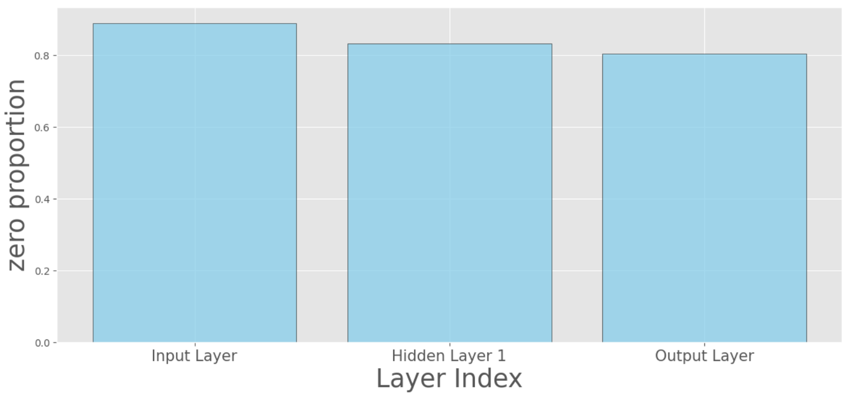
(b) Accuracy per epoch



(c) τ per backward pass



(d) r per backward pass



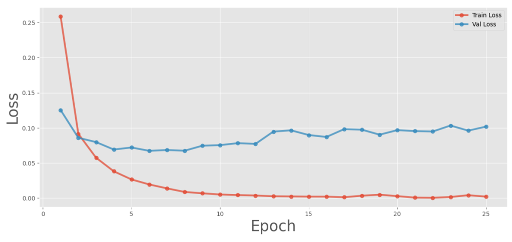
(e) Final sparsity per layer

Figure 4.12: Results for LRP with no regularization and $r_0 = 0.95$.

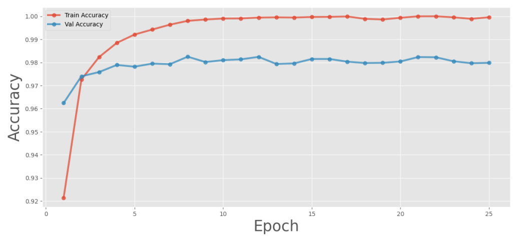
As for τ, is stays low and stabilized as intended, fig 4.12(c). However, the effect of the clamping threshold is also noticeable, suggesting it plays a non-negligible role in stabilizing its final value. This raises the question of whether the final value of τ results from natural convergence or is predominantly enforced by the clamp, which could warrant further investigation.

These goes hand in hand with what was said earlier, a regularization function could be what is needed in order for r to grow as intended. With this in mind, the following models were trained using the exact hyperparameters as in table 4.1 along with the regularization mentioned in 2.5 where $\lambda = 1\text{e}1$.

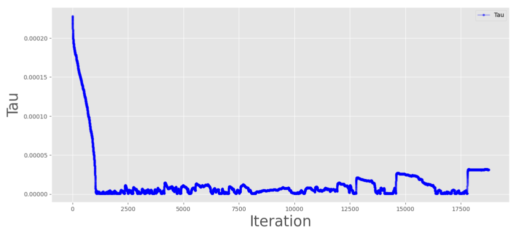
For assuring convergence, the models with regularization were trained for 25 epochs just like for the models with no regularization, to make sure the pruning ratio had time to



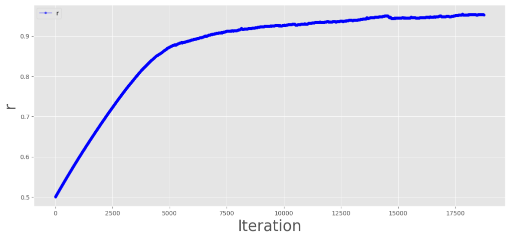
(a) Loss per epoch



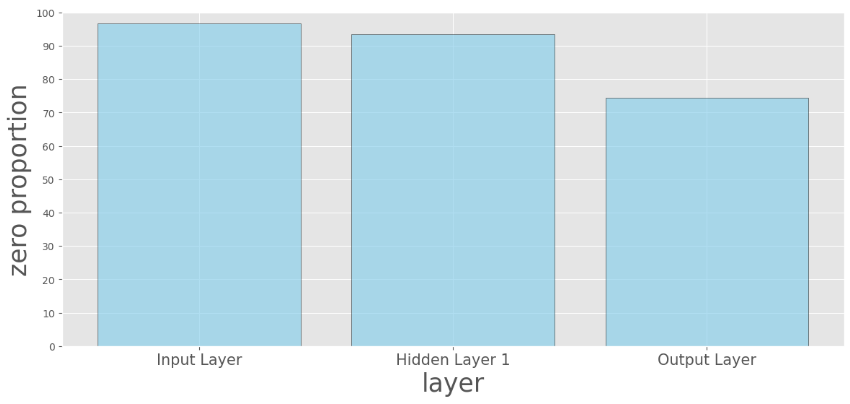
(b) Accuracy per epoch



(c) τ per backward pass



(d) r per backward pass



(e) Final sparsity per layer

Figure 4.13: Results for LRP with regularization and $r_0 = 0.5$.

	Train	Validation	Test
Loss	0.0237	0.101	0.0936
Accuracy	99.96%	98.0%	98.1%

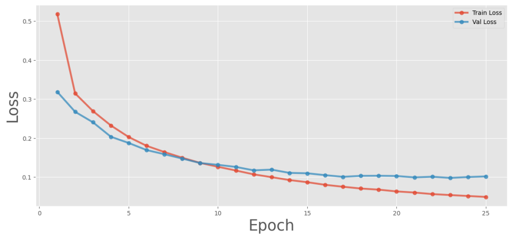
Table 4.4: Performance Metrics for LRP With Regularization and $r_0 = 0.5$

converge. What was observed, is that if starting in a smaller value of r, the model might achieve the same final value for r but it might need more epochs to do.

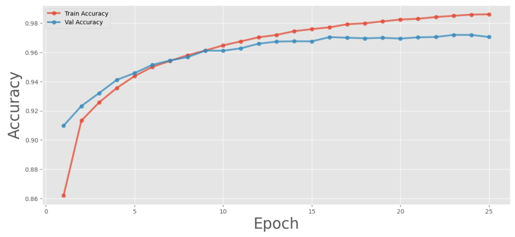
Although the results and the accuracy is better for all the sets, table 4.4, the slight gap between train and validation has returned, fig 4.13(a,b), which is not to say as a bad thing, as it is not that big and still constitutes a model with good generalization, however, it is something to be mindful when working with harder and larger datasets.

τ's behavior remains unchanged, fig 4.13(c), however, r does not go down, fig 4.13(d), quite the contrary actually, r goes to a value close to 1 and the model actually seems to feel the need to be sparse. The sparsity achieved here is one of 95.3%, which is the same as saying that the model ended up on a 21.28$\times$ compression rate.

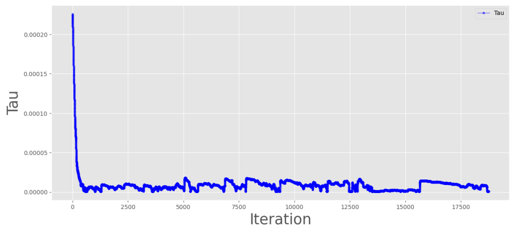
Giving the same treatment to the method with regularization as we did for without regularization, a starting position of $r_0 = 0.95$ was also tested.



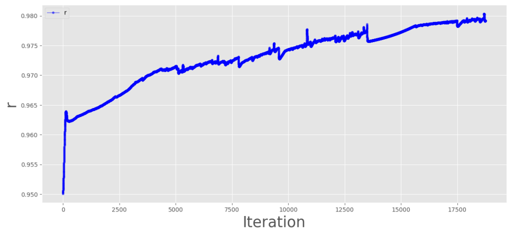
(a) Loss per epoch



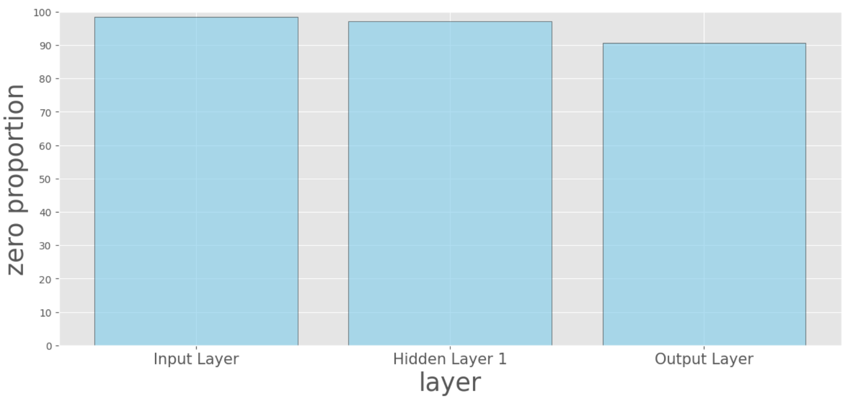
(b) Accuracy per epoch



(c) τ per backward pass



(d) r per backward pass



(e) Final sparsity per layer

Figure 4.14: Results for LRP with regularization and $r_0 = 0.95$.

	Train	Validation	Test
Loss	0.0493	0.102	0.0967
Accuracy	98.6%	97.0%	97.2%

Table 4.5: Performance Metrics for LRP With Regularization and $r_0 = 0.95$

As in the unregularized case, the gap between training and validation performance is more pronounced for $r_0 = 0.5$ compared to $r_0 = 0.95$, fig 4.14(a, b). However, unlike what might be expected from this larger gap, both training and validation accuracies are in fact slightly higher in the $r_0 = 0.5$ case. For instance, while the $r_0 = 0.95$ setting may yield accuracies of 98.6% and 97.0%, table 4.5, the $r_0 = 0.5$ configuration reaches 99.6% and 98%, respectively. This suggests that the slower convergence of r can allow the model to optimize more effectively, despite introducing a larger separation between training and validation curves. It is possible that this effect becomes more evident on larger or more complex datasets, as the observed difference on MNIST is relatively small.

For the model compression, an even higher sparsity is achieved, one of 97.9%, which equates to a staggering 47.62$\times$ compression rate. However, even though r follows the intended trend, fig 4.14(d), one cannot help to notice that this specific profile seems more irregular than the previous one.

Also, it would also be interesting to see how the structure of the weights matrix stays after pruning. For this, a binary heatmap was represented for the masks, giving an idea of what was pruned and what was not. For the layer between the input and the first hidden layer, for the model with $r_0 = 0.95$ and with regularization, fig 4.15.

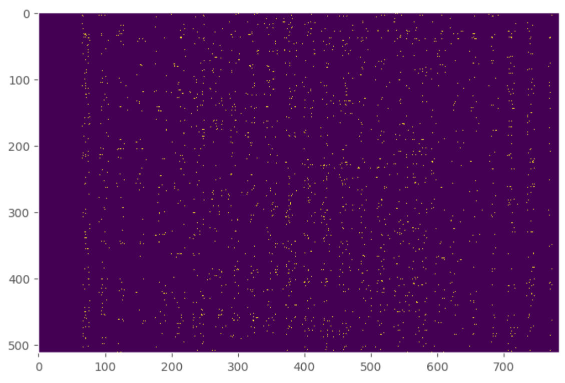


Figure 4.15: Mask Heat Map for 1st Hidden Layer for LRP with regularization and $r_0 = 0.95$

Knowing the dark purple indicates pruned weights and the bright yellow kept weights, the sparsity of the model can somewhat be visualized. In both binary mask heatmaps, each **row** corresponds to an output neuron and each **column** to an input feature (e.g., a pixel in the case of the first layer). A fully dark **row** indicates that a neuron has been entirely pruned, receiving no input connections, while a fully dark **column** implies that the corresponding input feature is unused by any neuron in the layer.

The binary pruning mask of 4.15 reveals a high degree of structured sparsity, even though the pruning is unstructured. Notably, many columns in the mask are entirely zero, indicating that a substantial portion of the input pixels are completely ignored by the network, probably in the peripheral are of the photo, since in that are there is less pixel variability across samples. The remaining active connections are not randomly scattered but appear clustered, suggesting that the network has converged to a localized and efficient representation of the input space. This form of emergent structure is particularly significant, as it implies that the pruning mechanism is not only effective at reducing redundancy but also implicitly encourages interpretability and feature selection.

For the version with no regularization, fig 4.16, since it is not as sparse, this clustering situation is even more apparent,

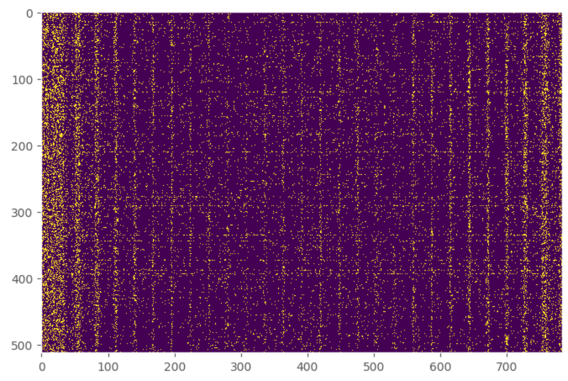


Figure 4.16: Mask Heat Map for 1st Hidden Layer for LRP with no regularization and $r_0 = 0.95$

As for the other mask heatmaps, some traces of this behavior can be seen, though they are not as apparent and thus not as interesting. They can be seen in chapter 7

Putting all the performed methods summed up in table 4.6

	Train Acc	**Test Acc**	**Prun.ratio r**	**Comp. rate**
LRP, $r_0 = 0.5$, no reg.	99.8%	97.6%	0%	1×
LRP, $r_0 = 0.95$, no reg.	98.9%	97.2%	86.6%	7.46×
LRP, $r_0 = 0.5$, with reg.	99.96%	98.1%	95.3	21.28×
LRP, $r_0 = 0.95$, with reg.	98.6%	97.2%	97.9%	47.62×

Table 4.6: Overall Performance Metrics for LRP

Since the validation accuracy and the test accuracy were so similar across all cases, only the test accuracy was shown. As for the results themselves, the best performing one across all metrics was the last one, with regularization and a high initial value for $r_0 = 0.95$.

Finally, for good measure, some extra experiments can be made to confirm the suspicions on r's behavior. Essentially, the starting positions from $r_0 = 0.1$ to $r_0 = 0.9$, with increments of 0.1, were all tested 5 times each, with and without regularization. Due to computing time, only 15 epochs were carried out for each and every experiment.

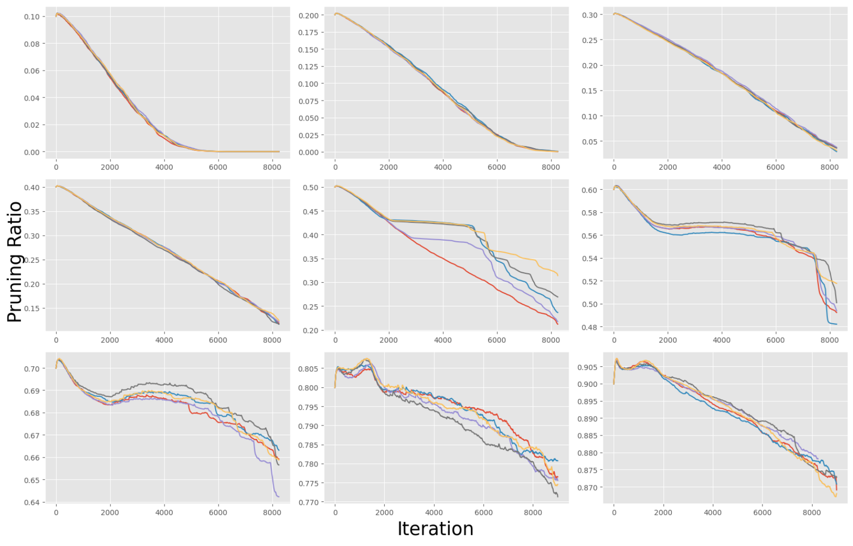


Figure 4.17: Multiple Starting Positions for r for LRP With no Regularization

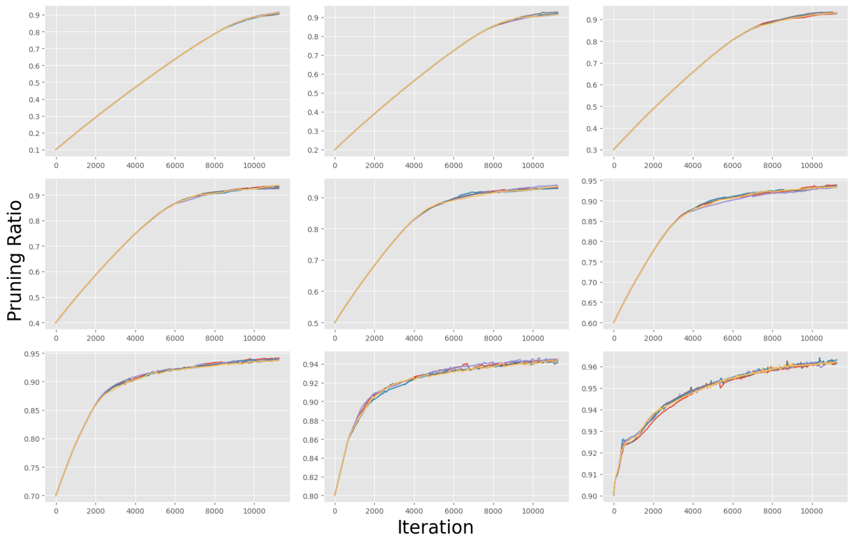


Figure 4.18: Multiple Starting Positions for r for LRP With Regularization

As suspected, in the absence of regularization, the model tends to avoid sparsity unless it is explicitly initialized close to an optimal pruning ratio, fig 4.17. In contrast, with regularization, fig 4.18, the final pruning ratio consistently converges to a similar value, regardless of the initial setting. That said, it is worth noting that when using regularization with a high initial value of r_0 (e.g., greater than 0.9), the progression of r becomes noticeably more erratic.

5. Conclusion and Future Work

5.1. Summary of Findings

This work demonstrated the potential of pruning while also highlighting its limitations when applied in its simplest form.

Namely, This preliminary study revealed that this more naive approach lacks meaningful information about the problem, and that forces the user to perform a time-consuming grid search. These observations not only demonstrated the limitations of simplistic pruning but also provided motivation for developing a more principled and adaptive approach.

Our proposed method showed promising results, particularly when combined with regularization. While it did not mitigate overfitting as effectively as techniques like dropout, it performed competitively with other pruning approaches in terms of compression rate.

Importantly, the study reinforced the idea that no method, including ours, eliminates the need for careful hyperparameter tuning. Rather than removing this burden, pruning-based approaches may shift it toward different, potentially more tractable, hyperparameters. However, selecting an optimal pruning ratio still remains a non-trivial task.

5.2. Limitations

A critical evaluation of the method's limitations is essential, not only to contextualize the results but also to motivate directions for future work.

Although the method reduces the need for grid search, it increases training time significantly. For the tests throughout this work, training with the method took roughly double the time as training without it. This raises the critique, that though the method eliminates the need for more runs than it needs, it also raises the time for the training itself. Although grid search typically requires many more training runs, this is still something to be mindful for larger architectures and datasets.

The method also makes the promise that, regardless of r_0, it will converge to a high sparsity. Despite this being somewhat truthful, fig 4.18 makes it evident that the higher the initial value for r the higher the value it will converge to, though even at $r_0 = 0.1$, a relatively high value of $r \approx 0.9$ is also achieved. Moreover, It was also observed, in fig 4.13, that starting at a lower value or r_0 also opens the possibility to lengthen the gap between the validation and train curve.

While this had no noticeable impact on performance in the current setting, it does not imply irrelevance. Rather, the MNIST dataset may be to simple and never really punish the model. This is even more evident in the case with no regularization and $r_0 = 0.5$, where the final sparsity was 0%, fig 4.11, and the performance remain unchanged compared to the version where the method is not applied. This means that this context was rather forgiving, and allowed the model to fail without deteriorating performance.

Another shortcoming lies in the experimental design, apart from the cross-validation plots in figs 4.17 and 4.18, only a single training run was performed for each setup. This limits the ability to assess variance across runs and weakened statistical rigor.

5.3. Future Work

Regarding future work, several optimizations can be made to improve the efficiency of the method. Since the pruning criterion involves repeated quantile calculations, the underlying sorting operations can become computationally expensive, especially with large models, creating a bottleneck for the training process. To address this, one could explore more efficient sorting algorithms tailored to this use case, or approximate methods that provide sufficiently accurate quantile estimates.

To enhance adaptability across different model architectures, it is beneficial to develop a layer-wise variant of the method instead of relying solely on global pruning. This could make the approach more flexible and easier to integrate into heterogeneous networks.

Regarding regularization, exploring alternative forms of regularization could also be beneficial, as they may help mitigate overfitting more effectively. Along with this, convergence criteria might also be helpful to determine when to shut down the method more effectively.

Finally, while the MNIST dataset served as a useful testing ground, its limited complexity, as already stated, means further validation is needed. Future work should include evaluations on more challenging datasets, as well as modern architectures such as Convolutional Neural Networks and Transformers.

References

[1] M. Cho, S. Adya, and D. Naik, “Pdp: Parameter-free differentiable pruning is all you need,” *arXiv preprint arXiv:2305.11203*, 2023. [Cited on pages iv, v, 11, 13, and 18.]

[2] K. Azarian, Y. Bhalgat, J. Lee, and T. Blankevoort, “Learned threshold pruning,” *arXiv preprint arXiv:2003.00075*, 2020. [Cited on pages iv, v, 11, 16, and 18.]

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Cited on pages 2, 3, 4, 5, 6, 9, 28, and 35.]

[4] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006. [Cited on page 5.]

[5] H. Robbins and S. Monro, “A stochastic approximation method,” *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400–407, 1951. [Cited on page 6.]

[6] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014. [Cited on page 6.]

[7] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” in *International Conference on Learning Representations (ICLR)*, 2016, arXiv:1510.00149. [Cited on page 7.]

[8] H. Cheng, M. Zhang, and J. Q. Shi, “A survey on deep neural network pruning: Taxonomy, comparison, analysis, and recommendations,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 46, no. 12, pp. 10 558–10 578, 2024. [Online]. Available: https://doi.org/10.1109/TPAMI.2024.3447085 [Cited on page 8.]

[9] S. Ashkboos, M. L. Croci, M. G. do Nascimento, T. Hoefler, and J. Hensman, “Slicegpt: Compress large language models by deleting rows and columns,” 2024. [Online]. Available: https://arxiv.org/abs/2401.15024 [Cited on page 8.]

[10] M. Zhu and S. Gupta, “To prune, or not to prune: exploring the efficacy of pruning for model compression,” 2017. [Online]. Available: https://arxiv.org/abs/1710.01878 [Cited on page 8.]

[11] F. Meng, H. Cheng, K. Li, H. Luo, X. Guo, G. Lu, and X. Sun, “Pruning filter in filter,” in *NeurIPS Workshop on Energy Efficient Machine Learning and Cognitive Computing*, 2020. [Cited on page 9.]

[12] X. Ma, W. Niu, T. Zhang *et al.*, “An image-enhancing pattern-based sparsity for real-time inference on mobile devices,” in *European Conference on Computer Vision (ECCV)*, 2020. [Cited on page 9.]

[13] J. Frankle and M. Carbin, “The lottery ticket hypothesis: Finding sparse, trainable neural networks,” in *International Conference on Learning Representations (ICLR)*, 2019. [Online]. Available: https://openreview.net/forum?id=rJl-b3RcF7 [Cited on pages 9, 10, and 40.]

[14] R. Tibshirani, “Regression shrinkage and selection via the lasso,” *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996. [Cited on pages 9 and 10.]

[15] S. Han, J. Pool, J. Tran, and W. Dally, “Learning both weights and connections for efficient neural networks,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2015, pp. 1135–1143. [Cited on pages 9 and 10.]

[16] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, “Pruning convolutional neural networks for resource efficient inference,” in *International Conference on Learning Representations (ICLR)*, 2017. [Cited on page 10.]

[17] B. W. Silverman, *Density Estimation for Statistics and Data Analysis*. Chapman and Hall, 1986. [Cited on pages 21 and 22.]

[18] D. W. Scott, *Multivariate Density Estimation: Theory, Practice, and Visualization*, 2nd ed. John Wiley & Sons, 2015. [Cited on pages 22, 23, and 25.]

[19] B. W. Silverman, “Algorithm as 176: Kernel density estimation using the fast fourier transform,” *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 31, no. 1, pp. 93–99, 1982. [Cited on pages 23 and 25.]

[20] G. B. Arfken, H. J. Weber, and F. E. Harris, *Mathematical Methods for Physicists*, 7th ed. Academic Press, 2013. [Cited on page 24.]

[21] J. W. Cooley and J. W. Tukey, “An algorithm for the machine calculation of complex fourier series,” *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965. [Cited on page 25.]

[22] S. M. Kay, *Fundamentals of Statistical Signal Processing, Volume I: Estimation Theory*. Prentice Hall, 1993. [Cited on page 30.]

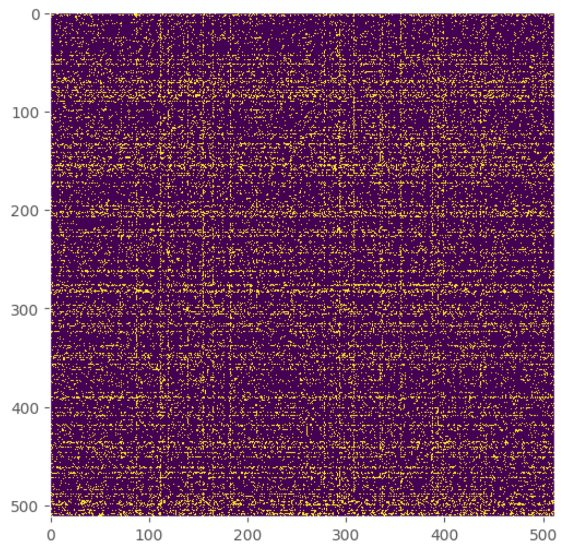
[23] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning*, 1st ed., ser. Springer Texts in Statistics. Springer, 2013. [Online]. Available: https://www.statlearning.com/ [Cited on page 30.]

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and Duchesnay, “scikit-learn: Machine learning in python,” pp. 2825–2830, 2011. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html [Cited on page 30.]

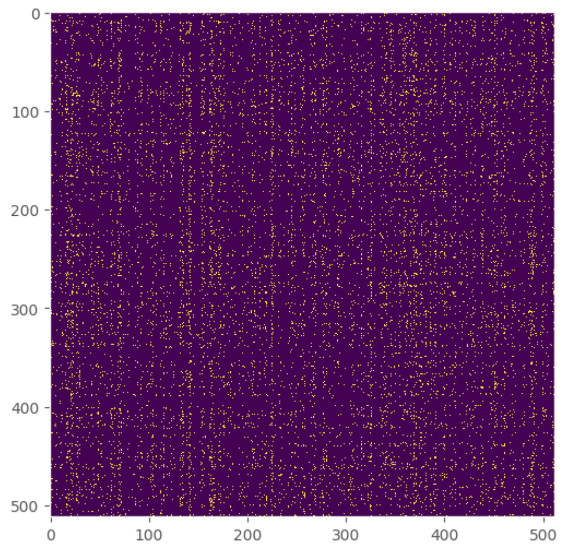
[25] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. [Cited on page 34.]

Appendix A: Additional LRP mask heat maps

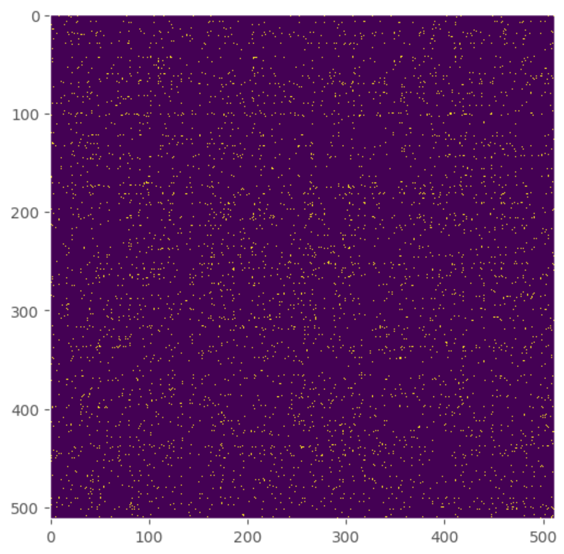
This appendix contains additional visualizations for different layers and configurations of the model. Figures include both regularized and unregularized settings for various values of r_0.



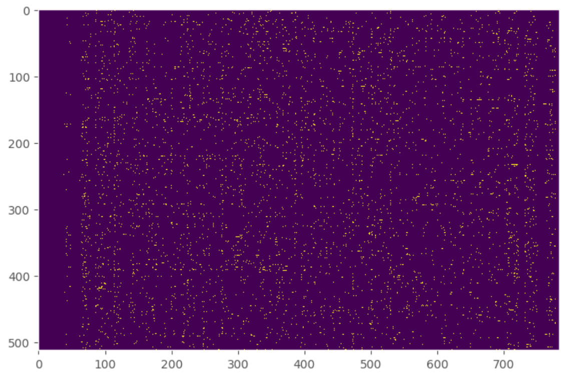
2nd Hidden layer (no regularization), $r_0 = 0.95$



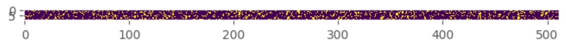
2nd Hidden layer (regularized), $r_0 = 0.50$



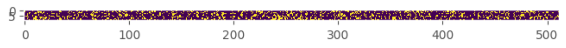
2nd Hidden layer (regularized), $r_0 = 0.95$



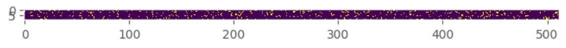
1st Hidden layer (regularized), $r_0 = 0.50$



Output layer (no regularization), $r_0 = 0.95$



Output layer (regularized), $r_0 = 0.50$



Output layer (regularized), $r_0 = 0.95$