
Convolutional Rank Filters in Deep Learning

by

Jonathan Blanchette

*A dissertation submitted to the University of Ottawa
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Electrical and Computer Engineering*



uOttawa

UNIVERSITY OF OTTAWA
FACULTY OF ENGINEERING
SCHOOL OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE

31 January 2020

© Jonathan Blanchette, Ottawa, Canada, 2020

Contents

1	Introduction	1
2	Literature Review	7
2.1	Rank Filtering instances present in the Literature	9
2.1.1	Historical Note	9
2.1.2	Pooling layers as Rank filter instances	11
2.1.3	Single-Element Patch Rank Filtering	14
2.2	Other Data-Dependent Non-Linear Filters	14
2.3	Capsule Networks property overview	15
2.4	Transposed Filters & Deconvolution Blocks	16
2.5	Unpooling Layers are Constant Weight Transposed Rank Filters	17
3	Questions to be answered & Hypothesis	20
3.1	General Behavior of Rank Filters & Transposed Rank Filters	20
3.2	Problem Statement	22
4	Proposed layers	23
4.1	Rank Filters	24
4.1.1	Broad Rank filters	30
4.1.2	Cross-Channel Rank Filter	30
4.1.3	Fully connected rank filters as a Gaussian process	31
4.2	Joint Linear-Rank Filters	33
4.2.1	Stochastic Joint Linear-Rank Filters	35
4.3	Rank+Linear Convolutions	39
4.4	Transposed Rank Filters	41
4.4.1	Avoiding artefacts	42
4.4.2	Block channel slice sorting inverse	45
4.5	Transposed Rank+Linear Filters	47
5	Histogram Filters (amplitude space)	49
5.1	HOG Pipeline Analysis	49
5.2	Histogram Equalization Pipeline Analysis	51
5.3	Amplitude bank filters and link with ReLU activation	52

6 Experiments & Methodology	54
6.1 Introduction	54
6.1.1 Cyclic Learning Rate Algorithm	54
6.1.2 Datasets & Other Experiment Setup	56
6.1.3 Batch Normalization Initialization	56
6.1.4 Filters Initialization	57
6.1.5 Notation	57
6.2 Denoising Autoencoders Experiments	58
6.2.1 Experiment 1: Using rank filters, find which coder-decoder pair is best to reduce image corruption with a wide gaussian noise for shallow autoencoder.	58
6.2.2 Experiment 2: Using rank filters, find which coder-decoder pair is best to reduce gaussian mixture corruption for an autoencoder of total depth of 4.	62
6.2.3 Experiment 3: Search an improved architecture inspired from the best combinations in experiment 1 and 2.	69
6.2.4 Experiment 4: Effects of reducing filter FoV or resolution when image is corrupted under spiked type noise for a shallow autoencoder.	75
6.2.5 Experiment 5: Dense Linear and Rank Filtering with 1D shuffling.	76
6.3 Classification Experiments	83
6.3.1 Experiment 6: Comparing classification capacity of rank weights versus linear weights while keeping number of parameters constant.	83
6.3.2 Experiment 7: Deterministic and Stochastic Joint Linear-Rank.	87
7 Discussion	94
8 Conclusion & Outlook	97
Appendices	106
A Kernel Derivation for the Rank Filtering Process	107
A.1 Rank filters of two elements	110
A.2 Rank filter kernels with $N > 2$	113
B A Curious Link Between Prime Numbers, the Maundy Cake Problem and Parallel Sorting.	116
B.1 Introduction	116
B.2 Converting a sorting problem into a sum problem.	117
B.2.1 Binary partial ranks	117
B.2.2 Smallest divisor partitioning of partial ranks	120
B.2.3 Prime partitioning of partial ranks	123
C Facts	126

Abstract

Deep neural nets mainly rely on convolutions to generate feature maps and transposed convolutions to create images. Rank filters are already critical components of neural nets under the disguise of max-pooling, rank-pooling, and max-Unpooling layers. We propose a framework that generalizes them, and we apply the novel layers successfully in convolution and deconvolution while combining them with linear convolutional feature maps. We call this class of layers rank filters. We explore the robustness, training, and testing performance under different types of noise. We provide analysis for their proper weight initialization, and we explore different architectures to discover where and when the rank filters could be advantageous [1]. We also designed transposed versions of the non-linear filter that doesn't generate artifacts. We propose the use of stochastic algorithms to sample sparse random real weights using the Gumbel max-trick. We compare the novel architectures with the baseline linear architectures. We recommend viewing histogram-based methods as potential deep learning models.

Acknowledgements

I want to thank my wife, Judith, for having supported that I have spent endless hours programming a GPU in isolation. I regret having chosen to work on such a hard topic and raising my two babies Ophélie and Léandre, at the same time. I feel I could've spent more time with my family. I want to thank my brother Philippe and my friend Micha for pulling me out of the abyss where I've been stuck in for a year. I can't thank my wife enough for the sacrifice she's made for our family. I've learned a lot during my Ph.D., the most important thing I learned was that life is better spent caring and providing for the ones we love.

Chapter 1

Introduction

The successes of AI were triggered by convolutional networks. These are based on an old concept: convolution i.e., applying a filter on an image so extract specific information from an image. The idea exploited in machine learning is that it is possible to automatically learn which filters are most likely to perform a given task, for example recognizing a human face.

Deep learning has demonstrated that it is possible to create a vast network of such filters which cascaded together allow achieved performance levels that may exceed what a human can achieve. This high performance is possible, provided sufficient examples to train this network, and earn the millions of parameters of all these filters. But convolution is a linear operation, and so that applying a cascade of filters is more than just one filter, between each convolution operation, use a non-linearity such as the Rectified Linear Units (ReLU)[24] activation.

The convolutional network, therefore, includes different convolutional layers followed by non-linearity. But to make the network more robust and more invariant to the spatial variations of pooling layers are also added. The purpose of this operation is to integrate local information to bring out the relevant features in a specific neighborhood. Besides, pooling is generally also used to reduce the amount of data going from one layer to another. By far, the most popular pooling operation used is max-pooling. This Max-pooling is a specific case of a more general type of pooling: Rank pooling[38], which in turn is an instance of Rank filters, as shown in Section 2.1.2.

CNN's, unfortunately, had the consequence of yielding results that a human would never have done. Miss-classifying a scrambled face¹ as a face (see Figure2.9). Anyone with experience in false-negative data mining from face detection will recognize this. Neural nets learn non-local patterns and textures and can have inevitable high probability classification errors. But what was the cause of these specific adversarial examples?

The culprit identified by Geoffrey Hinton and his team was partly Max-Pooling [33] (The paper that introduced Capsule-Nets). Max-pooling isn't the cause of all adversarial examples. It just caused a specific type of "scrambled face" type false-positives. Let's not use it as a scapegoat for all neural network problems. However, note that already in 2015, Springenberg et al. [43] proposed simplicity, i.e., neural networks without max-pooling. The idea is simple; indeed, we can achieve downsampling with strided convolutions and *voilà* you've achieved

¹ An image with eyes, mouth, and nose anywhere on skin texture.

downsampling without max-pooling, and you have a neural network that performs better. What about average pooling? Average pooling can be reached with a convolutional layer. Hence you can always replace an average pooling with a strided convolutional layer with a FoV equal to the stride. If the optimal weights for this layer average activations together, then effectively, the convolutional layer became an average pooling layer. But again, with or without pooling layers, you will still get classification errors that a human would never make.

Capsule-Nets [33] was designed as an object rendering black box. It comprises two main innovations. Firstly, it has “capsules”, which are vector features instead of a feature pixel. Secondly, it has a robustness property analogous to max-pooling. The robustness stems from the Routing-by-Agreement procedure that attenuates the information of capsules that didn’t have the maximal agreement probability. In other words, since information is discarded (“softly” because it is not fully ignored as in max-pooling), we necessarily will have robustness.

What are the causes of the mistakes caused by max-pooling? To answer this question, we need to take a closer look at the main disadvantage of max-pooling. It **discards too much information**. The robustness advantage is directly caused by throwing away information from activations that weren’t maximal.

What if we designed a layer that didn’t discard all information but learned to use, in addition to the maxima, the second maximum, the third-largest, etc.? This is effectively what a rank filter is designed to process. Rank filters are much more potent than max-pooling layers to encode information in the same way that convolutional layers are more powerful than average pooling. Max-Pooling layers are an instance or a realization of the Adaptive Rank filter. The novel filter presented in this thesis is adaptive because it can learn to combine ranks optimally. We name them Rank Filters for short.

This thesis aims to explore the advantages and disadvantages of using rank filters **systematically**. The **main goal** of this study is to learn the properties of this family of layers that **include** Max-Pooling layers as a realization in order to understand it better. This property analysis will provide valuable insight enabling future architectures with more informed use of rank filters as a critical component of neural network design.

In addition to max-Pooling layers, you have their “reversed” version named Max-Unpooling [50] layers typically used in deconvolutional neural networks. Deconvolutional Neural Networks usually have a bottleneck structure. They have a wide range of applications, such as pixel labeling and denoising. Transposed Rank Filters generalize max-unpooling layers, and we will study them in this thesis since they are intimately linked to Rank filters.


Specific instances of Rank filters were used before, and hence we already know some of their properties. For example, max-pooling leads to a region distortion where high-intensity pixel regions inflate and replace low-intensity pixels (see figure 2.2). This can potentially create artifacts since it is not robust to high-intensity erroneous values². Median filters are also an instance of rank filters. We know that they are robust to outliers and that they preserve edges [10]. Perhaps they could be helpful if used in image reconstruction architectures such as convolutional autoencoders[39], deconvolutional networks[28], or pixel labelling[6, 30]? Although both median and max filters are referred to as robust statistics, the term robust has different properties depending on which a single **specific** rank is kept.

In [28], the authors completely dropped pooling layers because, in the decoding process,

²This is the same class over-exaggeration problem that motivated the use of Capsule Networks.

there is information loss. This reconstruction problem follows from having pooling derivatives (Unpooling layers) that are too sparse i.e., the “only one switch per patch” Unpooling layer loses too much information for proper image reconstruction. Rank-Pooling was introduced in [38], and the pooling layer can learn the optimal rank by using a soft rank selection parametrization. Rank-Pooling [38] is for pooling only and not for general filtering and are not implemented in transposed versions for deconvolution. Rank-Pooling is not sparse³ and it follows that transposed rank pooling shouldn’t be sparse, and yet they aren’t investigated. Now since they belong to the much more general class of transposed rank filters proposed, we have to explore their properties.

Even though median filtering is more expensive than max-pooling, it can be very quickly computed on a GPU since we sort many **small** arrays. For a general rank filter we need a sorting algorithm which is very expensive. However, it is fine to use provided the number of elements to sort is not too large. Typically, Field of Views (FoV) of convolutional layers are small (smaller than or equal to 3×3) and hence can quickly be executed on a GPU.

Please follow the non-refereed “publication” [rank filter video link](#) on . This is my effort in disseminating the research ideas. This video is very helpful to follow the thesis.

Thesis Organization

The thesis is organized in the following chapters:

1. Chapter 2 is the literature review.
 - (a) Rank filters are presented as a family generalizing multiple layers and filters in Sections 2.1 and 2.1.1.
 - (b) We show in Section 2.2 that Rank filters belong to a broader family of data-dependent filters that is related to Histogram based methods. This is shown for theoretical purposes to show where rank filters belong.
 - (c) In Section 2.3, we highlight that Capsule-networks solve the “scrambled face” problem. **We highlight the conceptual similarities between max-pooling and the routing by agreement procedure.** Therefore, it explains why robustness properties are inherent to Capsule networks. We argue that it’s because the underlying intuition for picking the highest agreement probability is a biased⁴ assumption. The term “highest” automatically refers to a rank and it’s relevant to study the general rank properties in order to understand the effects of such design components.
 - (d) Deconvolution layers are presented in Section 2.4.
 - (e) We show that Max-Unpooling can be generalized as a transposed rank filters in Section 2.5.

³This means that the weights in the dot product in order to produce the output are not 0 as shown in Table 2.1.

⁴Biased because we are constrained to discard information. Why not using linear layers? What about using other agreement probabilities(second highest, third highest etc...)?

2. We discuss Questions & Hypothesis that motivate further the use of rank filters in Chapter 3.
3. The proposed layers are described In chapter 4.
 - (a) Rank filters & variants are presented in Section 4.1, we also include derivation of optimal weights when no activation is used in Section 4.1.3.
 - (b) Joint Linear-Rank Filters are presented in Section 4.2 & their stochastic variants are presented in Section 4.2.1.
 - (c) We present Rank+Linear convolutions in Section 4.3.
 - (d) We propose Transposed Rank Filters in Section 4.4 & we explain how we designed our algorithm in order to avoid upsampling artefacts in Section 4.4.1.
 - (e) We very briefly aboard the subject of Transposed Rank+Linear Filters in Section 4.5 for symmetry⁵ purposes.
4. We show how other non-linear data dependent filters could be potentially used in deep learning in Chapter 5. More precisely, we show how Histogram filters can be modelled as layers for a graphical model, and we show the resemblances of the structures to existing neural network architectures.
 - (a) We relate Histogram Of Gradients(HOG) features with a deep learning perspective in Section 5.1.
 - (b) We relate Histogram Equalization features to autoencoders in Section 5.2.
 - (c) We show that ReLU and Concatenated ReLU (CReLU)[37] are linked⁶ to histogram filters and we show that it is a convenient way of backpropagating through the layers derivatives that are histogram filters in Section 5.3.
5. We present the general **guidelines** of our methodology in Chapter 6. Note that since we explore, we included a methodology section for every experiment since every methodology differs from experiment to experiment.
 - (a) We briefly show what initialization we use in our experiments for Batch Normalisation and Filters(linear/transposed or not) in Section 6.1.4.
 - (b) We recapitulate on the notation used for the experiments in section 6.1. Note that in some experiments we further abbreviate the notation for “ease” and concision of reading so that deeper layers architecture descriptions fit in a single short table cell.
6. In Chapter 6, we conduct 7 experiments each containing various case studies⁷.

⁵ For every layer there is a transposed version of it (It may happen that the transposed layer is the same as the original layer, for example the ReLU layer).

⁶ The histogram filters derivatives yield Dirac delta functions which conveniently vanish. Thus we do not need to back-propagate though these.

⁷ In other words, per experiment section, we test various architectures and various Losses depending on the task assigned(A task could be image restoration or classification)

- (a) In Section 6.2, we conduct 5 experiments related to image restoration.
 - (b) In Section 6.3, we conduct 2 more experiments for image classification. The first experiment is to show the “value” of the capacity of rank filters compared to the capacity of convolutional filters(geometry related features). The second experiment is to test joint Linear-Rank filters, and other “weak”⁸ stochastic variants.
7. We discuss our results in Chapter 7.
 8. We conclude our thesis and open up the subject proposing further work on other data-dependent non-linear filters that were shown in Chapter 5.
 9. In the appendices we put in “hardcore” derivations, formulas and other “slightly off topic” work that we think is superfluous and helps in the readability of the thesis.
 - (a) In appendix A, we show the derivation of the Kernel for the rank filtering process. We show that it yields kernel functions (elements of the Kernel matrix) that are similar to the ReLU activation when we sort 2 values⁹ and we show that for more than 2 elements, the Kernel matrix elements becomes very difficult to express in a closed form. We use the results of this appendix to determine optimal weights in Section 4.1.3.
 - (b) In appendix B, we include a new theoretical parallel algorithm for sorting(with little to no implementation value) that has interesting theoretical value in mathematics.
 - (c) In appendix C, we list formulas used in various derivations. The formulas are related to multivariate continuous probability distributions.

Novel Contributions

The main contributions of this thesis are:

1. **Generalize rank filters** and to use them analogously to linear convolutions.
 - (a) Use of CUDA code¹⁰ to **implement novel APIs for forward and back-propagation**. More specifically we implemented on the GPU the channel-slice block sorting (forward propagation in Section 4.1) and un-sorting (for the back-propagation).
 - (b) **Derive the optimal initialization of the rank weights** within a network with no element-wise activation in Section 4.1.3.

⁸ By “weak” we mean a stochastic sampling of weights that is identical for batch samples. For a “strong” sparse stochastic variant to be possible, we would need to uniquely sample random weights for every batch sample. But this scenario is not easily implemented with current GPU hardware.

⁹ This occurs when FoV for Rank filtering of 2×1 or 1×2 .

¹⁰ As a side project while working on parallel sorting algorithms, we investigated a novel ranking algorithm based on partitioning an input sequence of length N into sub-sequences of length equal to the largest and smallest divisor of N . See Appendix B.2.2 for more info. We then apply the divide and conquer method to devise a second novel ranking algorithm based on subdividing an input sequence of length N into sub-sequences of length equal to the prime factors of N , in Appendix B.2.2.

- In this investigation, we derived many properties of the **Kernel matrix** when we view the rank filter as a process. Many derivations are included in Appendix [B.2.3](#)
2. We present **transposed rank filters** in analogy to deconvolution (transposed convolutions), in Section [4.4](#).
 - (a) Use of CUDA code to **implement novel API** for forward and back-propagation of the **transposed rank filters**.
 - (b) A **compensation method is designed** in order to **avoid artefacts** generation after the rank transposed filter, in Section [4.4.1](#).
 3. **Discover empirically the novel layers best possible architectures**.
 4. **Propose a stochastic algorithm** in order to generate sparse random weights. The sampling method is based on the Gumbel max-trick, as shown in Section [4.2.1.1](#).
 - The technique can be used to generate weights that don't necessarily encode probabilities (sum to 1 and positive). **We present an algorithm to generate sparse magnitudes**, in Section [4.2.1.2](#).

Other Minor Contributions

Those are not really novel since we are simply explicitly showing a perspective on viewing histogram based methods as potential building blocks in deep learning. Other contributions of this thesis include:

1. **Propose a systematic view of histogram based methods** (as non-linear data dependent filters) in order to view them as potential **building blocks for architectures in deep neural nets**, in Chapter [5](#).
 - (a) We also show examples that transposed histogram filters have been used in decoder type layers (in Histogram Equalization).
 - (b) We relate the non-linear filters with ReLU activation and CReLU activations.
 - (c) We show how HOG features have building blocks that are very similar to neural nets building blocks.

Chapter 2

Literature Review

The success in neural nets in machine vision is partly attributable to the convolution layer [23]. To be brief, it combines dot products between an input patch \mathbf{x} with weights \mathbf{w}_l and a bias b is added to form an output y i.e. $y = \mathbf{w}^T \mathbf{x} + b$ as depicted in Figure 2.1.

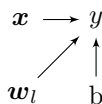


Figure 2.1: Rank filter feed-forward connection graph.

The convolutions generate feature maps that are used by subsequent layers, and the weights have many parameters. In contrast, other layers can take very few parameters and have a non-linear function, for example, the Rank-Pooling[38] combine ranks to generate a pooling layer. When you think about it, there isn't much of a difference in purpose between Convolutional layers and Rank-Pooling. One generates feature maps, and the second is a strided rank filter where the filter itself has very few parameters. Ranks are a type of feature map (see Figure 2.2), and their combination yield even more, but how much more? The feature maps generated by them were studied in [19], there it was shown that rank filters have edge preserving and shifting properties depending on the rank selected in addition to edge detection, low-pass, band-pass and high-pass filtering that preserves edges. In the new capsule network [33] article the authors show that some transformations learned by capsules were dilations. Rank filtering can do dilations very well as in Figure 2.2, maybe rank filters could have an application there? We may argue that the feature maps may not generate as much as the linear ones because the ranks are highly correlated more so than the input. This high correlation is especially true for large rank arrays¹, and it is a well-known result for order statistics[29].

Robustness to noise is an ability of a method to be utterly impervious to a type of noise. For example, median filters are robust to salt and pepper noise if the noise content isn't too high. Convolutional filters can also be robust to noise if the filter is sparse, and the noise is

¹ The more we increase the number of samples, the more the variance of the middle values increases. The minimum and maximum usually exhibit a higher variance than any other rank.



Figure 2.2: At the bottom of the figure are Max, Median and Min (rank) features respectively on the “cameraman”. We see on the bottom left that the **max filter dilates high values** and contracts low intensity values if they are next to a high one. **Median filters** acts similarly and **dilates median values** over others locally but to a lesser degree than the extreme value filters. Note that edges can be found if we use for example a range filter, i.e. Max-Min. Note that we scaled the images below to exaggerate the intensities for display purposes.

always located at the same zero filter parameter (The noise is located at every pixel with the same stride as the filter). So sparsity can yield pure robustness if we can find a dimension where we can separate the noise from the signal². Since sparsity may be desirable, we use the ℓ_1 constraint in the gradient descent algorithm. Note that it would be better to use the LASSO algorithm[46]; however, the algorithm is not implemented in Convolutional deep neural nets. The ℓ_1 criterion helps in keeping multiple parameters very small³ and a few large. The use rank filters may protect against adversarial attacks, or on the contrary, they may add flexibility to adversarial attacks and make them more successful.

When we use filters at the reconstruction in a decoder, we can get chequerboard artefacts[2]. The artifacts are remedied in the article [45]. We would like to see if an analogous type of

² Separating the noise from the signal is analogous from separating useless features from useful ones.

³ By “Very small” we mean it should be 0 but, in practice, when we use the ℓ_1 constraint in SGD they are not.

artifact can arise from a decoder using inverse filter ranks. What would these filters look like?

2.1 Rank Filtering instances present in the Literature

2.1.1 Historical Note

Skipped connections for rank filters is an old idea, historically when rank filters were proposed and studied in the seminal paper [19], some equations could with modelled as in figure 2.3. Additionally, the older Batchelor’s streak and spot detector [7] can be represented as a double skip connection in Figure 2.4. In the figures, rk is a *Sort* operation followed by filtering (see Equation 4.8).

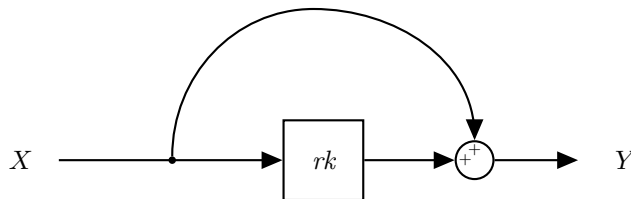


Figure 2.3

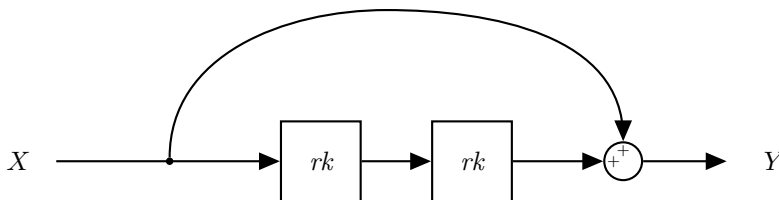


Figure 2.4

Both of these older ideas are valid in neural net language as the rank filter itself is non-linear. However, we could instead use a block that uses an activation or a block that is a shallow autoencoder. We will explore this in experiment 5 in section 6.2.5.

Theory Behind Rank Filters & Links with Existing Layers

Overfitting reduction techniques include Dropout[44], Spatial Dropout [47]. Many more stochastic techniques are used for regularisation, including the stochastic pooling algorithms, in particular the Fractional pooling layer [15]. There are striking similarities in these algorithms that permits to group them into a family. This will provide cues to derive the generalization

of the layers. Let an input $\mathbf{x} \in \mathbb{R}^{N \times 1}$ have a sorted sequence \mathbf{s} where $s_0 \leq \dots \leq s_{N-1}$. Then, if we filter \mathbf{s} with \mathbf{w} , then the output is defined by the dot product⁴:

$$y = \mathbf{w}^T \mathbf{s}_x \quad (2.1)$$

The matrix that sorts the sequence is a permutation matrix \mathbf{P}_x that depends on the data itself, i.e.

$$\mathbf{s}_x = \mathbf{P}_x^T \mathbf{x} \quad (2.2)$$

Sorting a sequence is thus non-linear process. We will drop the data dependence subscript for brevity so $\mathbf{P}_x = \mathbf{P}$ and $\mathbf{s}_x = \mathbf{s}$, unless stated otherwise in potentially ambiguous formulas. Furthermore,

$$\mathbf{P} = [\boldsymbol{\delta}_{\pi_0} \quad \dots \quad \boldsymbol{\delta}_{\pi_{N-1}}] \quad (2.3)$$

where $\boldsymbol{\delta}_i$ is a zero $N \times 1$ vector except the i th entry equals 1. Hence if we define $\mathbf{n}_{N \times 1}$ to be:

$$\mathbf{n}_{N \times 1} = [0 \quad 1 \quad \dots \quad N-1]^T \quad (2.4)$$

then we can rewrite the permutation order vector(rank vector) $\boldsymbol{\pi} \in \mathbb{N}_0^{N \times 1}$ as

$$\boldsymbol{\pi} = \mathbf{P}^T \mathbf{n} \quad (2.5)$$

where $\boldsymbol{\pi} = [\pi_0 \quad \dots \quad \pi_{N-1}]^T$. In MATLAB notation, equation 2.2 is equivalent to

$$\mathbf{s} = \mathbf{x}(\boldsymbol{\pi}) \quad (2.6)$$

Table 2.1: Pooling layers under a rank filtering perspective

Pooling Functions	Filter Parametrization	Output	refs
Average ⁵	$\mathbf{w} = \mathbf{1}/N$	$y = \mathbf{w}^T \mathbf{x}$ $y = \mathbf{w}^T \mathbf{s}$	[9]
Max	$\mathbf{w} = \boldsymbol{\delta}_N$	$y = \mathbf{w}^T \mathbf{s}$	[22, 9]
Fractional pooling	$\mathbf{w} = \boldsymbol{\delta}_{\tilde{N}}; \mathbf{w} \in \mathbb{R}^{\tilde{N} \times 1}$	$y = \mathbf{w}^T \mathbf{s}$	[15]
Stochastic pooling ⁵	$\tilde{\mathbf{w}}; \mathbf{w} = \mathbf{x}/\mathbf{1}^T \mathbf{x}; \mathbf{x} \in \mathbb{R}_{\geq 0}^{N+1}$ $\tilde{\mathbf{w}}; \mathbf{w} = \mathbf{s}/\mathbf{1}^T \mathbf{s}$	$y = \tilde{\mathbf{w}}^T \mathbf{x}$ $y = \tilde{\mathbf{w}}^T \mathbf{s}$	[49]
Rank-based average pooling	$\mathbf{w} = [u(\pi_i - t)]_i / (N - t),$ $t \in [0, N - 1], u(x) = 1; x \geq 0$	$y = \mathbf{w}^T \mathbf{s}$	[38]
Rank-based stochastic pooling	$\tilde{\mathbf{w}}; \mathbf{w} \propto \alpha \cdot (\mathbf{1} - \alpha \mathbf{1})^{\text{cn}_{N \times 1}}$	$y = \tilde{\mathbf{w}}^T \mathbf{s}$	[38]

We have made a small list of layers in table 2.1 trying to shed light with a rank filter perspective. It is by no means exhaustive. The pooling filters in the table, other than the

⁴This is a parametrized L-estimator[16]. Other interesting estimators belonging to this group include the Harrell-Davis estimator [48]

⁵In Average and Stochastic pooling, the use of the input or the sorted sequence are equivalent.

average and stochastic ones, have non-linearities. They are activations on a region larger than a single element of the input tensor.

The fractional pooling layer is interesting because the regularisation is done by averaging out multiple models at test time. The randomness of that layer is not in the filter itself. Rather, it is the field of view and the length of the filter \tilde{N} that is “random”. Since its output is $\mathbf{w} = \delta_{\tilde{N}}$, it is essentially a maximum rank filter. Stochastic layers have generalization benefits, and it would be interesting if we could combine linear and rank information simultaneously.

2.1.2 Pooling layers as Rank filter instances

In this section we will show various features of pooling layers and show how they are instances of a broader rank filtering family. Pooling layers all have the following characteristics in common:

- **For all pooling filters, the window FoV is equal to the stride** and there is **no overlap** between windows.
- **The number of output channels is always equal to the input channels.**
- Pooling layers are designed to downsample. The stride is always greater than 1.

In the following table, we show an analogy table to highlight resemblances and differences of three pooling filters.

Table 2.2: Pooling layers features

Pooling Layers	Parameters	Intuition & Function	Advantages	Disadvantages	refs
Max-Pooling	►No learning possible	Pass highest activation	Robust	►Discards information	[22, 9]
Rank-Pooling	►“Soft” rank selection learning	A rank bandpass	►Can learn robustness ►dilation & contraction features	►Partially discards information ►Constrained parametrization	[38]

Rank filters are similar to Rank-Pooling (Rank-Pooling with linear weights instead of a positive band-pass filter), however the number of output feature maps (output channel) can be whatever the designer wants it to be. It doesn’t have to be equal to the number of input feature maps. A general rank filter shares the following characteristics with convolutional layers:

- The window FoV is **not necessarily equal** to the stride. There could be an **overlap** between windows.
- **The number of output channels are output feature maps.** This number is chosen by the designer.

- Downsampling is an option. The stride is can take on any value.

In Figure 2.5, we show how the methods fall within the scope of rank filters. Each highlighted area correspond to a block. Values highlighted in orange are the input blocks. In the middle, the values are sorted within the block highlighted in red. Each element is then multiplied by weights in blue to form an element in the output (in green). In the following figure, the weights have a 1 at the last position. This means that only the maxima is considered. Note that this figure is for illustrative purposes only, there is no learning occuring, in other words, the weights are static.

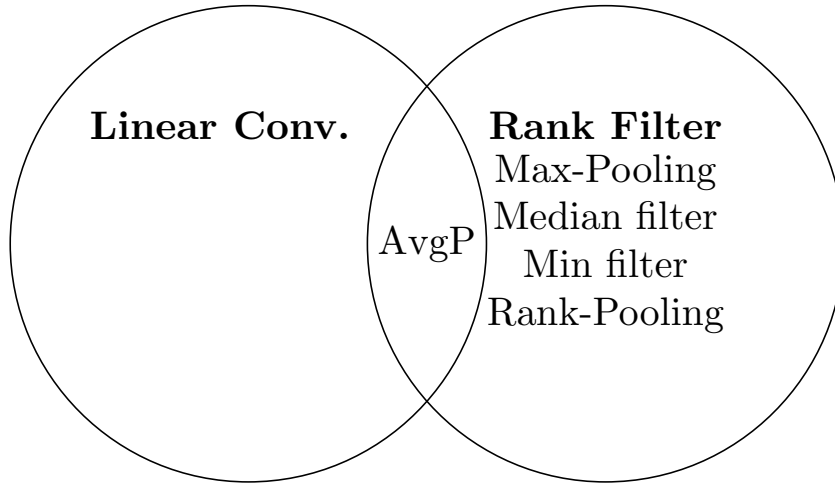


Figure 2.5: Venn diagram showing which method can be reached by rank filters and linear filters. Average pooling can both be reached by convolutional layers and Rank layers if all weights equal to $1/d_{in}$, where d_{in} is the total size of the weight. Average pooling and Max-pooling are static and hence can't learn.

In Figure 2.6, we show how Max-Pooling the methods fall within the scope of rank filters.

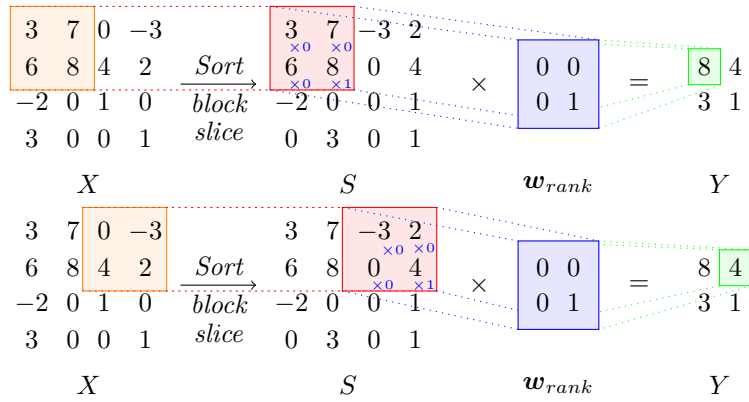


Figure 2.6: Max-Pooling with downsampling of 2 viewed under rank filtering perspective. The Field of View(FoV) is 2×2 so the is stride always equals to the FoV for pooling layers. The weights in blue, can't be learned and are fixed to the matrix with values 0,0,0,1. The last 1 in the weights mean that the max value is kept. The values highlighted in red in the middle show the ranks, i.e. the values are sorted from smallest to largest in column order. The input is on the left and the output is on the right.

In Figures 2.7 and 2.8, we show how Average-Pooling fall within the scope of rank filters but also that it can be reached by the subspace of convolutional layers which explains why we placed it in the overlap in the Venn diagram in Figure 2.5. In the Figures Figures 2.7 and 2.8, the weights are equal to the normalizing factor. This equal weight initialization means that the average is computed. Since the average of ranks is equal to the average of input values, both figures are equivalent.

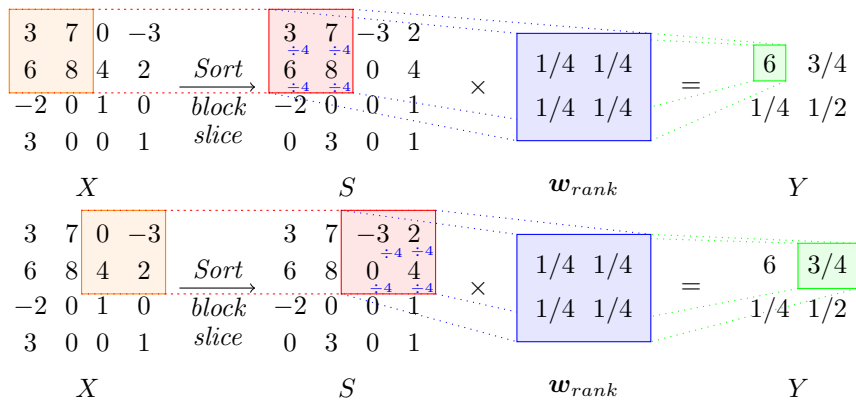


Figure 2.7: Average-Pooling with downsampling of 2 viewed under rank filtering perspective. The Field of View(FoV) is 2×2 and the stride always equals to the FoV for pooling layers. The weights (in blue) can't be learned and are fixed to $1/4$.

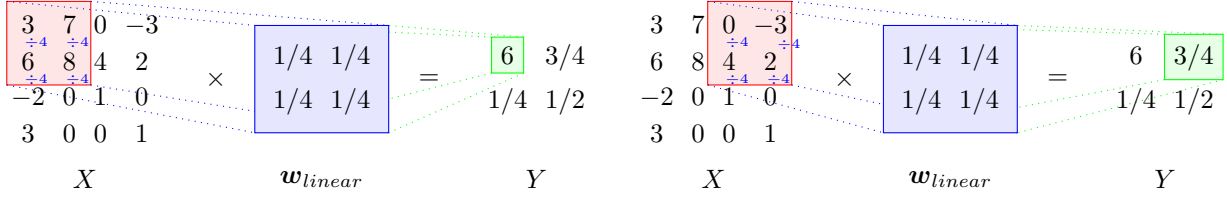


Figure 2.8: Average-Pooling with downsampling of 2 viewed as a fixed Convolutional layer of stride 2. The weights (in blue) can't be learned and are fixed to 1/4.

2.1.3 Single-Element Patch Rank Filtering

Define the zero augmented data vector $\mathbf{x}^z = \begin{bmatrix} 0 & \mathbf{x}^T \end{bmatrix}^T$, and its corresponding sorted sequence $\mathbf{s}^z = \mathbf{P}^T \mathbf{x}^z$. In the layers in table 2.3, the activation and the linear layers are applied to a single tensor element.

Table 2.3: Dropout/ReLU type layers under a rank filtering perspective

Dropout/ReLU Functions	Filter Parametrization	Output	Layer type	refs
ReLU	$\mathbf{w}_{2 \times 1} = \delta_2$	$y = \mathbf{w}^T \mathbf{s}^z$	Activation	[24]
Dropout	$\tilde{\mathbf{w}}; \mathbf{w}_{2 \times 1} = [p - 1 \ p]^T$	$y_{test} = \mathbf{w}^T \mathbf{x}^z,$ $y_{train} = \tilde{\mathbf{w}}^T \mathbf{x}^z$	Stochastic and linear	[44]
PReLU	$\mathbf{w}_{2 \times 1} = [a \ 1]^T$	$y = \mathbf{w}^T \mathbf{s}^z$	Activation	[18]

Looking at the above table, we see that the activations non-linearities are essentially caused by the term $\mathbf{P}_{\mathbf{x}^z}^T$ in the expansion of \mathbf{s}^z . The Dropout layer is not a rank filter, it is stochastic and linear in \mathbf{x}^z , it was shown in the table for comparison purposes. There are degenerate cases where there is no difference between using \mathbf{s} or \mathbf{x} because of the parametrization of the filter. For example, when PReLU with $a = 1$, the nonlinearity vanishes. This means that using a dot product on \mathbf{s} doesn't guarantee that there will be a non-linearity in terms of ranks.

2.2 Other Data-Dependent Non-Linear Filters

Recall equation 2.2, i.e. $\mathbf{s} = P_{\mathbf{x}}^T \mathbf{x}$. The permutations needed to sort \mathbf{x} depend on the data itself. Hence, P^T is a data dependent-transformation. It is non-linear even though there is some form of linearity. It is a special matrix product and hence we can view P as a type of selection “filter”. For a general data-dependent matrix that multiplies an input you have:

$$\mathbf{y} = F_{\mathbf{x}}^T \mathbf{x} \tag{2.7}$$

We thus call it a non-linear filter because it is a “linear” transformation with a matrix whose elements depend on input vector \mathbf{x} . Histogram based methods can be modelled with the

presence of such a data-dependent matrix F_x^T from equation 2.7. Even ReLU layers can be modelled as such.

Note that rank filters are one of them and we will only implement them in this thesis. We will explicitly show the details in how to model Histogram based algorithms using a graphical model perspective for neural networks in Chapter 5. This is for theoretical purposes. In the Conclusion in Chapter 8, we propose to continue the research on this subject.

2.3 Capsule Networks property overview

This section is a very brief reference to the work of Geoffrey Hinton’s team in [33]. It aimed in solving the classification problematic that his team identified as a consequence of using max-pooling layers. The problematic is that neural nets seem to encode features related to a class non-locally as depicted in the following figure:

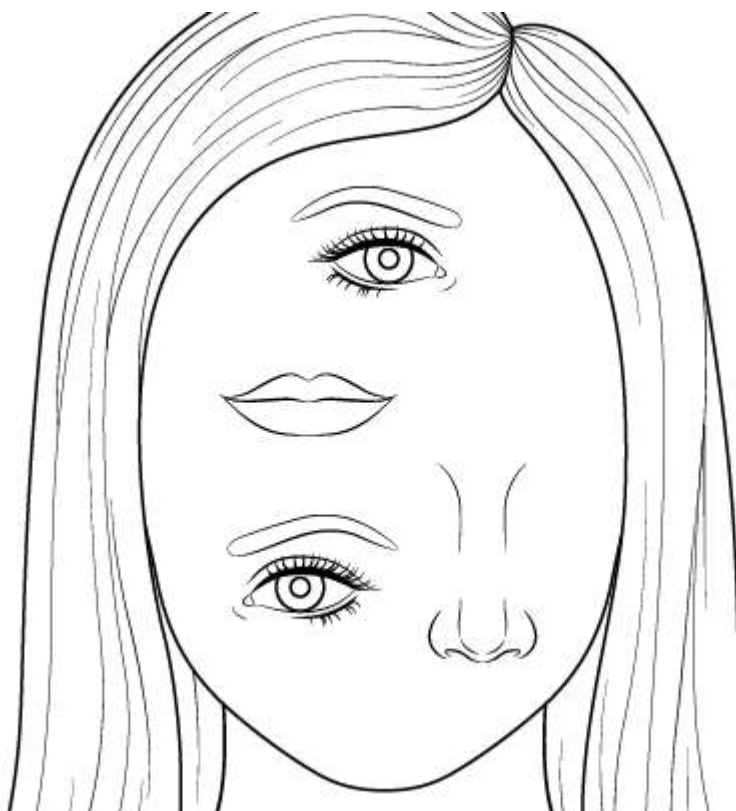


Figure 2.9: A convolutional neural net will generally classify this image as a face with very high probability. Typical CNNs do not encode features locally(geometrically). Capsule net are designed as an inverse object rendering function.

The team proposes a routing procedure as an alternative to max-pooling. This routing procedure is similar to max-pooling in the sense that it selects⁶ the capsule (vector feature)

⁶ Not a pure selection *per se* of the vector feature that agree with other capsules with high probability, it

that agrees with other capsules with high probability. This idea is a great leap forward. However, there are CNNs without max-pooling that have the same non-local classification problem. Surely max-pooling can't have the whole blame. Then what makes a neural net encode feature structures?

This is what the team working on Pure-Capsule networks [12] are trying to elucidate. They are trying to show that it's a generalized higher-order convolution that is present in Capsule nets that solve the scrambled face problem. Remember the article [43] that proposes to use all-convolutional networks instead of using CNNs with max-pooling nets? Well, you can see the Pure-Capsule networks as an extension of this article for networks with a generalized higher dimensional convolution.

The routing procedure possibly has the same **weakness**⁷ as max-pooling layers although **it is less apparent because it is hidden in an extra dimension**. It **discards information** much like max-pooling does and thus has **similar robustness** properties. Why not use all the information? In this thesis, we are going to show the properties of rank filters and systematically compare them with all-convolutional nets. We can then deduce if intuition based on the highest activation or, by extrapolation, the highest inter-capsule agreement probability will have similar advantages and disadvantages. The highest value is rank. And it follows that it falls within the scope of properties of rank filters(albeit on an extra dimension).

2.4 Transposed Filters & Deconvolution Blocks

Deconvolution layers have nothing to do with the deconvolution method used to unblur with Fourier analysis and the Wiener filter. In deep neural net community “deconvolution” is synonym to a transposed convolution. It is mainly used in decoders in autoencoders and it is meant to upsample so as to reflect the coder's convolution downsampling.

The authors of the paper [39] have generated saliency maps using neural nets with decoders that used gradient-based reconstruction and compared them with deconvolutional networks that were presented in the article [50] that introduced Max-Unpooling layers(which is a gradient-based method as pointed to in [39]). Additionally, the authors of [6] also have worked with decoders and made significant observations in the use of pooling window sizes, thus complementing the work of [50].

The authors in [39] pointed out that gradient-based reconstruction is slightly different from the deconvolution net used in [50], primarily because the ReLU layer is applied differently. Adequately, the authors of [50] did consider gradient-based reconstruction⁸ since they wrote “*We also tried rectifying using the binary mask imposed by the feed-forward ReLU operation, but the resulting visualizations were significantly less clear.*”. What defines clearness in this context? What qualities are desirable for a reconstruction? The observation of [50] is analogous to one from the authors of [6] i.e. small pooling windows reduces spacial context but preserve thin structures as opposed to large pooling windows. Perhaps the “less clear” reconstruction mentioned in [50](Explicitly shown in quotes above) when using gradient based

is rather an attenuation of other vector features that don't agree together with the highest probability. This routing step is iterated three times. The more we iterate, the more we discard outlying information.

⁷ Weakness that follows from discarding information that could potentially be useful for subsequent layers.

⁸ Gradient based reconstruction for a ReLU layer is the DeReLU operation.

reconstruction is because ReLU layer reduce spatial context(as mentioned earlier in [6]), since from Table 2.3 a ReLU layer can be modelled as a small max-pooling type layer but with a 1×1 stride applied on a zero-augmented patch \mathbf{x}^{z9} .

Since gradient-based reconstruction for a ReLU layer is the DeReLU operation, from the observations consensus previously discussed, we can state that DeReLU is not desirable for deep decoders. But does this apply for all types of layers? What about gradient-based reconstruction quality for pooling layers (Unpooling layers)? We answer a more general question involving transposed rank filters in Chapter 7.

2.5 Unpooling Layers are Constant Weight (no learning) Transposed Rank Filters

The authors that have implemented the max-Unpooling layers [50] use “switches” as depicted in Figure 2.10 taken from the article [11](We used this article simply because they have a figure that we can use by courtesy). These correspond to an index from where the maximum came from. In a decoder theses switches are used to reverse the process of pooling layers placed in coders. In the following figure, the “switches” are the indices from where the maximum intensity value came from. The Unpooling layer “undoes” the pooling operation by putting the pixel of the Unpooling layer’s input to the position from where the maximum value was picked in the pooling layer at the encoder level.

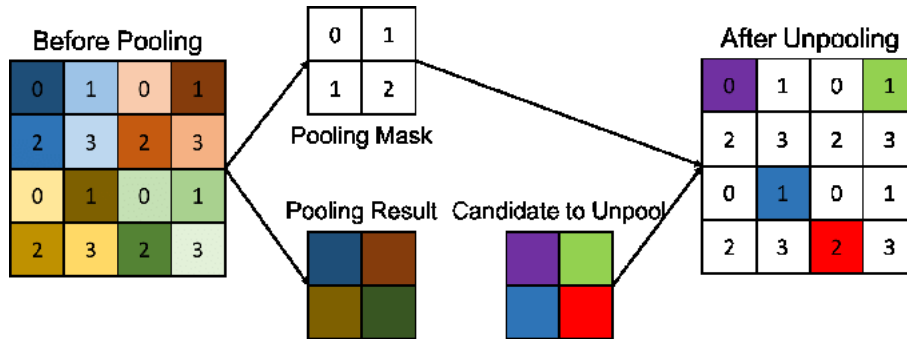


Figure 2.10: This image is taken from [11] under Creative Commons Attribution 4.0 International. It depicts the pooling layer and the corresponding Unpooling layers at the decoder level.

Unpooling are in fact transposed rank filters with constant weights $\mathbf{W}_{2 \times 2 \times C_{in} \times C_{in}}$ as shown in Figure 2.11 and illustrated with an example in Figure 2.12. In Figure 2.12, you have Max-Pooling & Max-Unpooling layers with downsampling of 2 Viewed under rank & transposed rank filtering perspective. The Field of View(FoV) and stride is 2×2 . The weights of both the Max-Pooling and Max-Unpooling layers can’t be learned and are fixed to the matrix with weights 0, 0, 0, 1. The last 1 in the weights mean that the max value is

⁹ Refer to Table 2.3 where we explicitly show that a ReLU layer is an element-wise operation $ReLU(x) = \max(x, 0)$.

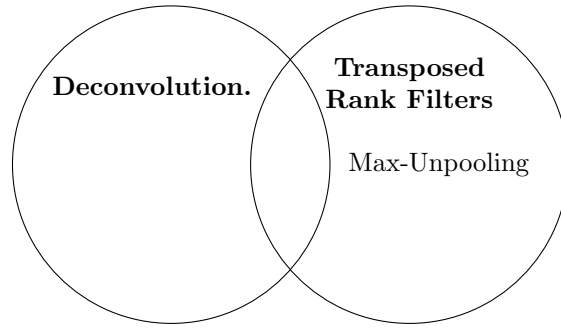


Figure 2.11: Venn diagram showing how Max-Unpooling falls in the space reached by transposed rank filters. The transposed convolution (deconvolution) circle is on the left by analogy.

kept. **The input to the decoder or Max-Unpooling layer is ϕ** in the middle (with the active patch highlighted in orange). In this context, ϕ denotes a random operation (it may or not be an activation). The input to Max-Unpooling layer is upsampled by the weights $0, 0, 0, 1$ arranged as a matrix. Then, the values are permuted back from where they came from. This is done by using the permutations highlighted in blue at the top left that were saved from the coder (in this case the Max-Pooling layer). Note that this is overkill, since the weights are fixed and sparse you really do not need to remember all permutations, this is just for illustrative purposes. The useful permutations is really those associated to the max value. This is what Matthew Zeiler and his team [50] refer to as “Switches”. The output is on the top right highlighted in green. In Pooling layers only a single 1 is present per output

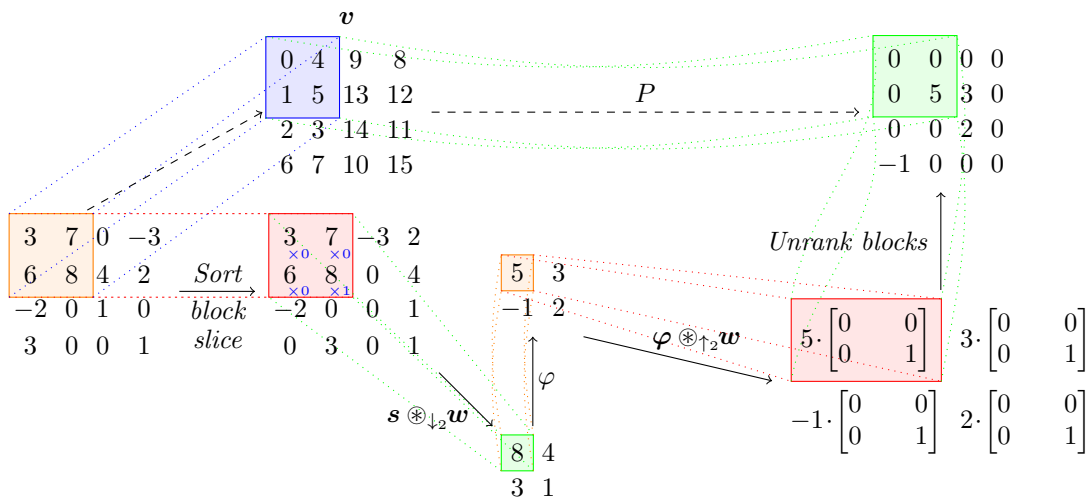


Figure 2.12: Max-Pooling & Max-Unpooling layers with downsampling of 2 Viewed under rank & transposed rank filtering perspective.

channel dimension per input channel slice(at the maximum rank position). The weights are constant, and so, they can't learn to combine ranks optimally. This learning deficiency also

applies to Max-Unpooling layers in the same manner.

How do the switches affect the reconstruction? If we generalize to rank filters, how remembering all permutations affect the reconstruction “clearness”? If we ignore the permutations, how will it affect the reconstruction quality? If gradient-based reconstruction is desirable in the case of the Unpooling layer then shouldn’t it be the same for the generic transposed rank filters?

Chapter 3

Questions to be answered & Hypothesis

3.1 General Behavior of Rank Filters & Transposed Rank Filters

In figure 3.1, we see that saving switches or permutations (in the context of rank filters) correspond to gradient-based reconstruction. However, since the activations used in the decoder are not gradient-based, then why should we bother saving the permutations? In other words, since it is no longer gradient-based, why should we continue the trend with the different permutations? The deeper the coder, the deeper is the decoder, and consequently, the decoder block activations make the total reconstruction even less gradient-based. Accordingly, the permutations shouldn't be able to yield any benefits for the reconstruction using a deep decoder. The following hypothesis is not ours, it follows from the observations previously discussed in Section 2.5.

Hypothesis 1 (H1). *Unpooling layers with large Field of View (FoV) should not be useful for deep decoders using permutations from the coder but should be suitable for shallow decoders. The reconstruction using small FoV should be less of a problem in deep decoders since permuted pixels are (usually) more correlated in a small window.*

Under another perspective if we relax 1 by simply allowing all ranks to serve in the decoder then an analogous Hypothesis would be:

Hypothesis 2 (H2). *Transposed rank layers with large Field of View (FoV) should not be useful for deep decoders using permutations from the coder but should be less problematic for shallow decoders since they use a smaller neighborhood (more correlation).*

We are going to verify this Hypothesis 2 with the full rank filter transpose generalization proposed. Recall that in [6], it is mentioned that small pooling windows reduce spacial context but preserve thin structures as opposed to large pooling windows. It is true that large pooling windows reduce spacial context because it is a maximally sparse rank filter (a lot of information is thrown away). Suppose now that we use all ranks in an optimal way

and that Hypothesis 1 does not cause any issues, then we are no longer throwing everything out, and so we should be able to preserve thin structures especially for large window sizes¹. We must verify if the thin structures reconstruction potential of the decoder depends on the depth. We must also check if some activations are better suited than others in this case. If we use shallow or deep decoders that use the reverse-sorting process, would it still be able to preserve thin structures? If so, would it be able to discern which thin structures are spurious? It is tough to disprove or prove 1 or 2 simply because we can't conduct an experience that would show without any doubt that the large permutations would cause reconstruction problems for a vast decoder. If indeed rank filtering is a bad idea for large decoders, maybe this is simply because the optimization algorithm didn't find a solution. In other words, perhaps rank filtering makes the learning curve more rugged, but that doesn't mean that the minimums didn't sink deeper in the loss function. Or on the contrary, maybe the feature maps generated by deep rank coders aren't good because they become highly redundant. Maybe it's both, the learning curve becomes jagged or flattened without good minima (meaning that the deep rank feature maps are useless). True or not, we'd better verify the hypothesis for shallow rank autoencoders only. In this case, there shouldn't be any filter redundancy issue, and permutations should be correlated for the coder is right next to the decoder. In other words, transposed rank filters can efficiently use locations where the pixels came from in order to fill the corrupted ones by a learned weight part of a feature map only if the permutations came from an adjacent coder. Hence we can state our hypothesis that:

Hypothesis 3 (H3). *We may expect that shallow rank autoencoders perform well in preserving fine structures.*

We can add that the window size has an effect on the fine structure. The extent of it's effects are a subject of investigation.

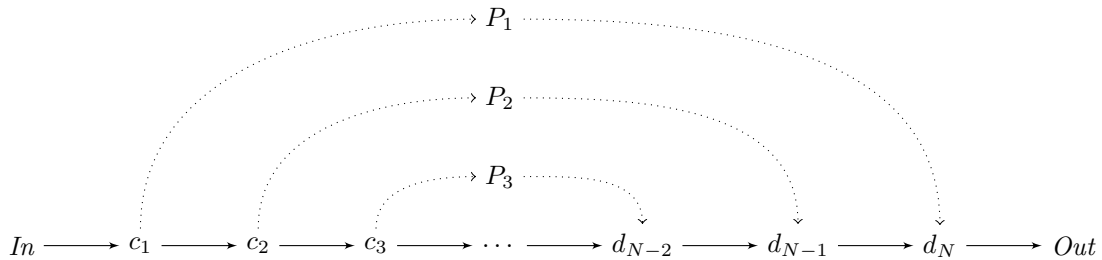


Figure 3.1: Each coder block c_i save the permutations needed to sort a channel slice that will be used for their corresponding decoder block. This is the same for max-Unpooling where switches are saved from the coder to be subsequently used in the decoder. This is considered part of a gradient based reconstruction.

¹ It is reasonable to guess that thin structure preservation is a problem akin to edge preservation in median filters. The edge preservation capacity of a median filter depends on the window size(the more significant, the better). So we are inclined to believe that a similar phenomenon might incur in a reconstruction.

3.2 Problem Statement

This thesis details how rank filters are used in a deep neural net. With success, it has been applied in a wide variety of architectures. We also show how rank filters include max-pooling layers as a particular case. We develop a framework that generalizes max-pooling layers, and it follows that rank-filters are nothing more than non-linear filters generating interesting feature maps. We will test if a neural net can learn max-pooling as an optimal choice. We will show when we can combine additively linear convolutions and rank filters to achieve superior performance compared to if we used either one of the layers alone. This combination will be studied too. Additionally, we will present a novel architecture that can be used to regularise nets combining rank filters and linear ones multiplicatively.

We also aim to answer the following questions:

1. In classification, is a rank parameter worth the same as a linear parameter?
2. When is it advantageous of using rank filters in classification ?
3. What are the advantages and disadvantages of using rank filters in autoencoders ?
4. How remembering all permutations affect a decoder's reconstruction "clearness"?
 - (a) If we simply ignore the permutations how will it affect the reconstruction quality?
5. If we use shallow or deep decoders that use the reverse-sorting process, would it still be able to preserve thin structures?
 - (a) If so, would it be able to discern which thin structures are spurious?
 - (b) Does thin structures reconstruction better or worse with large versus small FoV in shallow decoders?
 - (c) Does thin structures reconstruction better or worse with large versus small FoV in deep decoders?
6. What forms do the rank filters and transposed filters take? What do the weights look like?
7. Do transposed rank filters generate artefacts analogous to transposed convolutions?
8. Is a stochastic version of rank filters combined with linear filters beneficial?

Chapter 4

Proposed layers

Here we will present adaptive rank filters and adaptive transposed filters & other variants including other types of filters that mixes linear and rank features. This chapter is organized as following:

1. Rank filters & variants are presented in Section 4.1.
 - (a) “Spatial” Rank filters are presented in Section 4.1. We explicitly show the forward propagation algorithm.
 - (b) In Section 4.1.1, we present a complex algorithm that uses many local rank statistics.
 - (c) In Section 4.1.1, we present how we can use rank statistics inside the channel dimension with a FoV of 1. This filter¹ is used in experiment 5 (see Section 6.2.5), as a component within a larger network. It has no resolution loss that are caused by the high correlation of the filters. However it has resolution loss within the input feature map dimension.
 - (d) We derive optimal variance initialization when weights are sampled from a random distribution. This derivation assumes no activation is used for rank filters in Section 4.1.3. It relies on results from Appendix A.
2. Joint Linear-Rank Filters are presented in Section 4.2. We propose this filter to generate a superfamily that includes linear layers and rank layers. In the last experiment, this will serve to test if the joint-filter can serve better as a neural net component than a convolutional layer². We provide examples, and forward propagation algorithm. It will also serve as a mathematical framework in order to derive stochastic sampling algorithms that can be equally useful for linear layers and rank layers.
 - (a) Stochastic Joint Linear-Rank Filters are presented in Section 4.2.1. We present the mathematical framework that can serve as a basis for stochastic training of a neural network, using either the rank or linear space.

¹ The filter is superset that includes the Max-Out layer.

² If not, we can conclude that rank weights compete with linear weights.

- (b) In Section 4.2.1.1, we present how the Gumbel-Max trick can be used to sample from a discrete distribution. We provide a technique to sample a probabilistic weight vector(positive weights that sum to 1) with a desired degree of sparsity. In Section 4.2.1.2, we make an important leap that has great potential where we generate weights with a specified average degree of sparsity where the weights can take any value(the weights magnitude no longer sum to 1 and they can take negative values). In Section 4.2.1.3, we show how we can apply the sampling techniques for Joint Linear-Rank Filters.
- 3. We present Rank+Linear convolutions in Section 4.3. This is a family that combine rank features and linear features additively. We test it in experiment 6 for the Classification task. It serves to compare information capacity of rank features versus linear features.
- 4. We propose Transposed Rank Filters in Section 4.4, .i.e. the transposed version of the filter in Section 4.1.
 - (a) We explain the upsampling artefacts problem inherent to transposed layers in Section 4.4.1, and we propose a solution.
 - (b) In Section 4.4.2, the clockwork of the transposed rank filter is explained. The transposed filter includes the modification in order to prevent upsampling artefacts. (This creates an asymmetry between the forward propagation algorithm of the transposed filter and the back-propagation algorithms of the Rank Filter in Section 4.1.
- 5. We very briefly aboard the subject of Transposed Rank+Linear Filters in Section 4.5 for completion sake.

4.1 Rank Filters

A rank filter is simply the dot product of weights \mathbf{w}_r with sorted values \mathbf{x} of an input patch \mathbf{x} added with a bias b as in the following equation:

$$y = \mathbf{w}^T \mathbf{s} + b. \tag{4.1}$$

It's feed-forward connectivity graph is shown in Figure 4.1. In the same figure, we have permutations P that sort the values of the input i.e. $\mathbf{s} = P^T \mathbf{x}$. The dotted arrows in Figures 4.1 and 4.8 indicate that no derivatives propagate through P 's path.

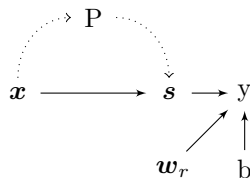


Figure 4.1: Rank filter feed-forward connection graph.

In Figure 4.2, an example is shown on an input(left) and output(right). The filtering is done with a stride of 1 with a 3×3 window with weights given by the blue matrix. The top left numbers inside active zone (show which numbers are to be sorted in red), these are sorted and stored into S into the highlighted active zone in orange. Since the input is 3×3 we have:

$$h_S = 3 \left(\left\lfloor \frac{4-3}{1} \right\rfloor + 1 \right) = 3 \cdot 2 = 6.$$

In the example, there are exactly 2×2 blocks size 3×3 in S hence S is 6×6 . Finally, we do a convolution of stride $(3, 3)$ on S which yields the 2×2 final output (active zone highlighted in green).

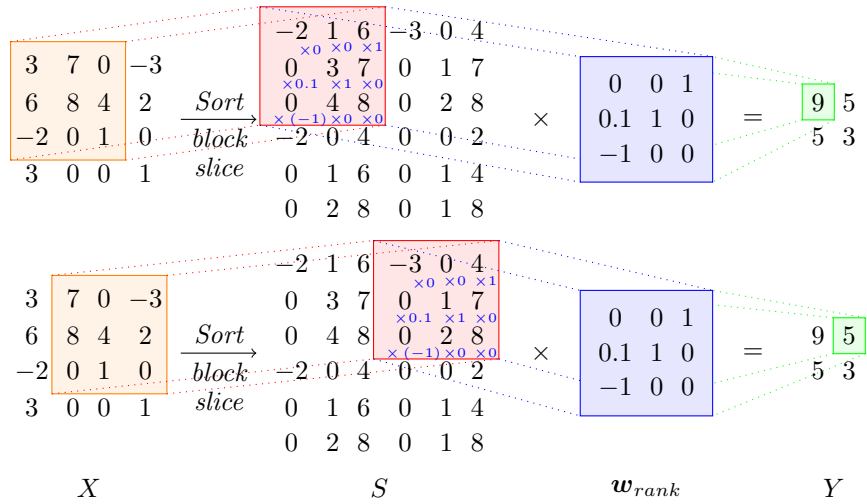


Figure 4.2: Rank filter example with a 3×3 field of view of stride 1. Each channel slice has to be sorted within the window range. Once every channel slice is sorted we store the values into S . The filter (in blue) is then applied exactly with a convolution of stride equal to the window size(3 in this case). We do not rank all values in the channel dimensions together, rather independently. Otherwise, they would be highly correlated.



Figure 4.3: Example of an input X to a rank filter. The input is a 33×33 RGB image $\mathbf{X}_{33 \times 33 \times 3}$.

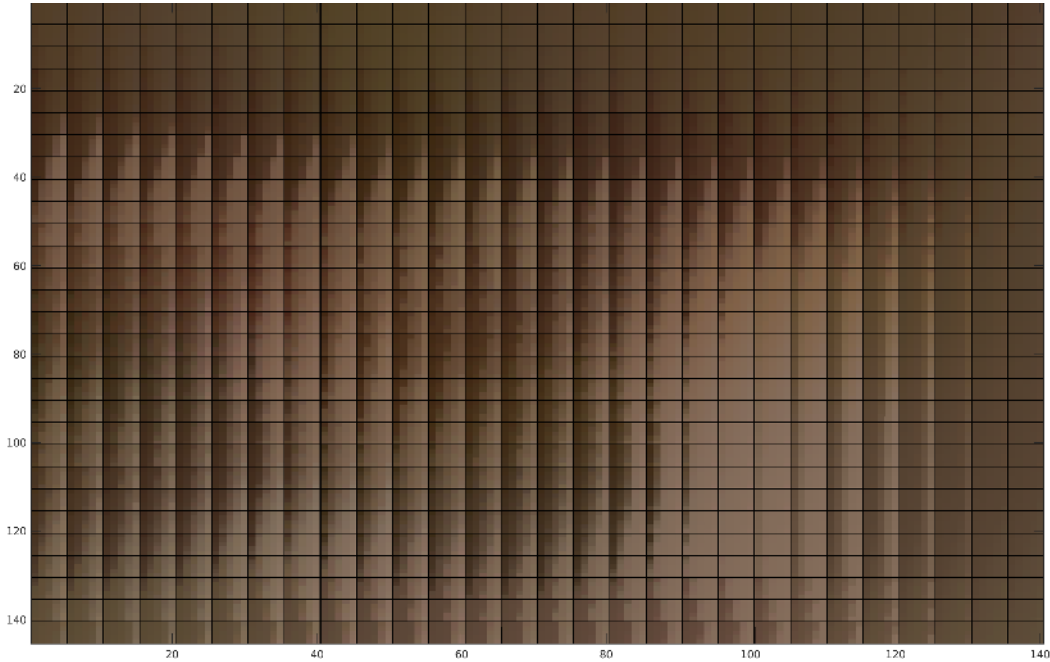


Figure 4.4: Example of rank augmentation by using an FoV= 5×5 with stride $\Delta = 1$. The rank augmented output pictured $\mathbf{S}_{h_S \times w_S \times C_{in}}$ has height and width dimensions $w_S = h_S = 5 \left(\lfloor \frac{33-5}{1} \rfloor + 1 \right) = 145$. The output is thus $\mathbf{S}_{145 \times 145 \times 3}$. The following convolution will stride on the boxes only ($\Delta = 5$, with $FoV = 5$) to yield an output (not depicted) with $h_{out} = w_{out} = \lfloor \frac{145-5}{5} \rfloor + 1 = \lfloor \frac{33-5}{1} \rfloor + 1 = 29$ or $\mathbf{Y}_{29 \times 29 \times C_{out}}$.

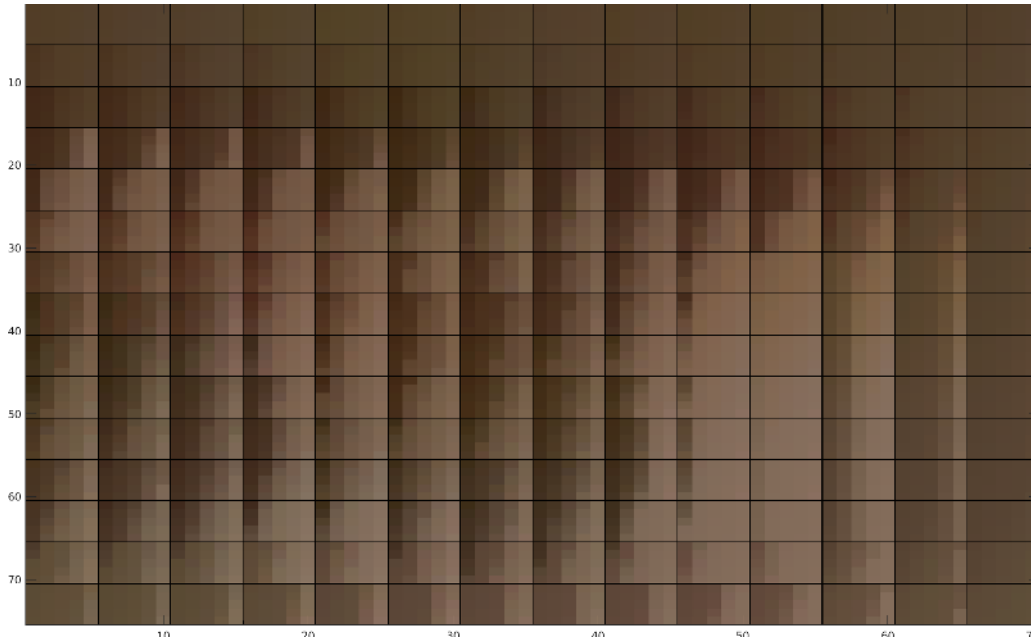


Figure 4.5: Example of rank augmentation by using an FoV= 5×5 with stride $\Delta = 2$. The rank augmented output has height and width dimensions $w_S = h_S = 5 \left(\lfloor \frac{33-5}{2} \rfloor + 1 \right) = 75$. The output is thus $\mathbf{S}_{75 \times 75 \times 3}$. The ranks are stored in 5×5 patches, the lowest intensity pixel is stored at the top left and the highest intensity pixel is stored at the bottom right. The final stage of the rank filter includes the convolution that will stride inside the boxes to yield an output (not depicted) with $h_{out} = w_{out} = \lfloor \frac{75-5}{5} \rfloor + 1 = \lfloor \frac{33-5}{2} \rfloor + 1 = 15$ or $\mathbf{Y}_{14 \times 14 \times C_{out}}$.

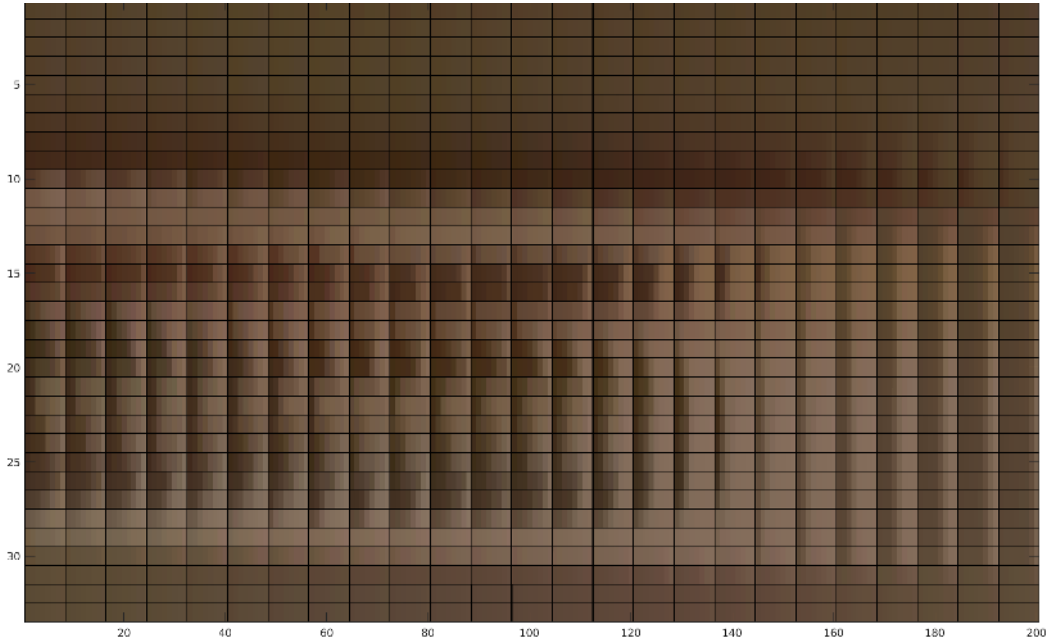


Figure 4.6: Example of rank augmentation by using an FoV= 1×8 with stride $\Delta = 1$. The rank augmented output has height dimension $h_S = 1 \left(\lfloor \frac{33-1}{1} \rfloor + 1 \right) = 33$ and $w_S = 8 \left(\lfloor \frac{33-8}{1} \rfloor + 1 \right) = 208$. The output is thus $\mathbf{S}_{33 \times 208 \times 3}$. The filter is then applied on the boxes only to learn local optimal ranks ($\Delta = 1$, with $FoV = 1 \times 8$) to yield $h_{out} = 33$ and $w_{out} = \lfloor \frac{208-8}{8} \rfloor + 1 = \lfloor \frac{33-8}{1} \rfloor + 1 = 25$ or $\mathbf{Y}_{33 \times 25 \times C_{out}}$.

If we would have used a strided rank filter of say 2, then the orange highlighted zones in the left of Figure 4.2 would have moved by 2 instead of 1 (given that the input was padded for this to be actually possible).

More generally, for a rank filtering with a filter $\mathbf{W}_{h_{win} \times w_{win} \times C_{in} \times C_{out}}$ with input $\mathbf{X}_{H \times W \times C_{in}}$ with a stride of (Δ_H, Δ_W) and double sided padding of (pad_H, pad_W) . We first compute a collection of sorted blocks arranged into a tensor $\mathbf{S}_{h_S \times w_S \times C_{in}}$ where:

$$h_S = h_{out} \cdot h_{win}, \quad (4.2)$$

$$w_S = w_{out} \cdot w_{win}, \quad (4.3)$$

$$h_{out} = \left\lfloor \frac{H - h_{win} + 2pad_H}{\Delta_H} \right\rfloor + 1, \quad (4.4)$$

and

$$w_{out} = \left\lfloor \frac{W - w_{win} + 2pad_W}{\Delta_W} \right\rfloor + 1. \quad (4.5)$$

The sorting is only done in every channel dimension, we do not sort across input channels. Finally, we compute the linear convolution with $\mathbf{S}_{h_S \times w_S \times C_{in}}$ with the filter $\mathbf{W}_{h_{win} \times w_{win} \times C_{in} \times C_{out}}$ with a stride of (h_{win}, w_{win}) . Hence, the output sides are, upon substituting in equations 4.2 and 4.3:

$$\left\lfloor \frac{h_S - h_{win}}{h_{win}} \right\rfloor + 1 = \left\lfloor \frac{h_{out} \cdot h_{win} - h_{win}}{h_{win}} \right\rfloor + 1 = h_{out}, \quad (4.6)$$

$$\left\lfloor \frac{w_S - w_{win}}{w_{win}} \right\rfloor + 1 = \left\lfloor \frac{w_{out} \cdot w_{win} - w_{win}}{w_{win}} \right\rfloor + 1 = w_{out}. \quad (4.7)$$

We shall abbreviate the rank filter with the sublayer models of block sorting (a non-linear upsampling process where the output was labelled with $\mathbf{S}_{h_{out}h_{win} \times w_{out}w_{win} \times C_{in}}$) and a subsequent linear filter. In Section 6.1.5.1 we explain the convolution notation, where $C_{w_{win} \times h_{win}, \downarrow(\Delta_W, \Delta_H), (PadW, PadH), C_{in} \rightarrow C_{out}}$ for a general filter size. Hence, a general rank filter is described by:

$$rk_{FoV, \downarrow \Delta, Pad, C_{in} \rightarrow C_{out}} \equiv Pad \rightarrow BCS_{\downarrow \Delta \uparrow FoV} \rightarrow C_{FoV, \downarrow (w_{win} \times h_{win}), 0, C_{in} \rightarrow C_{out}}. \quad (4.8)$$

In the preceding equation, the BCS layer stands for Block Channel-slice Sort (see Section 4.1) and it outputs $\mathbf{S}_{h_{out}h_{win} \times w_{out}w_{win} \times C_{in}}$. Note that **for back-propagation**, we **need the permutations indices** to permute the derivatives, so we need to save them (or recompute them in the GPU) from a forward pass as in Figure 4.1. The convolution operation (ignoring the bias and padding) is described by the following equation (correlation mode):

$$y[a][b][k][n] = \sum_{i,j,c} \omega[i][j][c][k] \cdot x[i + (a - 1) \cdot \Delta_1][j + (b - 1) \cdot \Delta_2][c][n], \quad (4.9)$$

where the sum indices ranges are $i \in [h_{win}]$, $j \in [w_{win}]$, $c \in [C_{in}]$. Our proposed filter is thus described by the following steps:

$$S_{h_{win} \times w_{win}}[a][b][c][n] = vec_{h_{win} \times w_{win}}^{-1} \left(P^T \left(vec \left(X^{slice}[a][b][c][n] \right) \right) \right), \quad (4.10)$$

where vec is the vectorisation operator, vec^{-1} is the reshaping operation and the input block channel slice $X^{slice}[a][b][c][n] \in \mathbb{R}_{h_{win} \times w_{win}}$ is

$$X^{slice} = x[1 + (a - 1) \Delta_1 : h_{win} + (a - 1) \Delta_1][1 + (b - 1) \Delta_2 : w_{win} + (b - 1) \Delta_2][c][n]. \quad (4.11)$$

The tensor of sorted slices is done by concatenating the matrices $S_{h_{win} \times w_{win}}[a][b][c][n]$ to form the Tensor $\mathbf{S}_{h_S \times w_S \times C_{in} \times N}$. Then (omitting the bias),

$$y[a][b][k][n] = \sum_{i,j,c} \omega[i][j][c][k] \cdot s[i + (a - 1) \cdot h_{win}][j + (b - 1) \cdot w_{win}][c][n]. \quad (4.12)$$

4.1.1 Broad Rank filters

We explored the possibility of using a rank filter that used local rank statistics over a broader region to replace the downsampling that occurs at the input (that may remove useful information). By using information in the height and width dimension by using k_h blocks and k_w blocks and then downsampling after the convolution. We will designate by k the pair (k_w, k_h) , $k \circ FoV$ will mean $(k_w w_{win} \times k_h h_{win})$ and $k \circ \Delta$ will designate $(k_w \Delta_W \times k_h \Delta_H)$. With this notation in hand, our broad version of rank filter is:

$$rk_{FoV, \downarrow \Delta, Pad, C_{in} \rightarrow C_{out}}^{Broad} brkk \equiv Pad \rightarrow BCS_{\downarrow 1 \uparrow FoV} \rightarrow C_{k \circ FoV, \downarrow k \circ \Delta, 0, C_{in} \rightarrow C_{out}}. \quad (4.13)$$

This filter suffers from using too much memory. Effectively, the downsampling is done in the convolution, and thus we have to compute all permutations of all possible blocks. This limits it's applicability for only very small FoVs (though this can be an advantage in low noise cases). Another problem with this design is that the filter is larger $\mathbf{W}_{k_h h_{win} \times k_w w_{win} \times C_{in} \times C_{out}}$. We saw no great improvement, after experimenting with it (although it may help if some regions are less corrupted than others). We will not carry on analysis nor continue experiments on this filter since it is not very fast and doesn't seem particularly advantageous.

4.1.2 Cross-Channel Rank Filter

What about comparing ranks inside the channel? This inter-channel comparison is an appealing option because when we use local statistics, we lose locality information within the block even if we use a shallow autoencoder (see Experiment 4 in Section 6.2.4) with the transposed rank filter of Section 4.4. For example, when we use a FoV that spans only in height, the output filter distorts the input to look unnatural. To solve this problem, we would need to compare this output with its input at a pixel level. Fortunately, the authors of [4] introduced a 2D shuffling idea³ that will be useful. We will use a more straightforward 1D shuffling operation. The main reason why we used the shuffling process was not to reimplement the CUDA code (except for the 1D shuffling operation). The 1D shuffling operates in the following manner: say we have K inputs X^1, X^2, \dots, X^K with dimensions

³This was also studied in [45] and compared with other upsampling methods in the context of reducing checkerboard artifacts.

$H \times W \times C_{in}$. Then we shuffle them across the width within the same channel and height dimension, giving an output of Y of dimensions $H \times K \cdot W \times C_{in}$ giving:

$$Y_{:, :, k_i} = \left[X_{:, 1, k_i}^1 \cdots X_{:, 1, k_i}^K \mid X_{:, 2, k_i}^1 \cdots X_{:, 2, k_i}^K \mid \cdots \mid X_{:, W, k_i}^1 \cdots X_{:, W, k_i}^K \right]; \quad (4.14)$$

given $k_i \in \llbracket 1, C_{in} \rrbracket$. For example, if $K = 3$ and say our three inputs are labelled $X^R, X^G, \dots X^B$ with dimensions $H \times W \times C_{in}$, then in the channel c_i the output is:

$$Y_{:, :, k_i} = \left[X_{:, 1, k_i}^R \quad X_{:, 1, k_i}^G \quad X_{:, 1, k_i}^B \mid X_{:, 2, k_i}^R \quad X_{:, 2, k_i}^G \quad X_{:, 2, k_i}^B \mid \cdots \mid X_{:, W, k_i}^R \quad X_{:, W, k_i}^G \quad X_{:, W, k_i}^B \right].$$

After the shuffle operation, we perform rank filtering with a filter W of size $h_{win} \times K w_{win} \times C_{in} \times C_{out}$.

The number of comparisons within a channel slice is of K . Additionally, since we do not want to augment our signal, as opposed to [4], we use a stride $\Delta = (1, K)$. In our experiments in section 6.2.5, this net was very beneficial in preserving local information given that we used skip-dense connections carrying the input throughout different rank FoV resolutions. Additionally, the 1D-shuffle was also beneficial for the baseline all-linear filtering. Overall, 1D-shuffling combined with dense skip connections is very helpful in many tasks both for rank filtering and the classical convolutions.

4.1.3 Fully connected rank filters as a Gaussian process

In this section, we investigate a new family of kernels to compute the similarity of vector inputs $\mathbf{x}, \mathbf{x}' \in \mathbb{R}_{M \times 1}$ after activation. Comparing the ReLU layer to rank filtering is interesting because it shows that the ReLU layer is a max operation of every single element in a tensor compared to 0. This superficial similarity hints that when comparing two elements, the ranks should have similar features to that of a ReLU activation. But, in the two elements ranks, there are two different non-linearities i.e., the min and max operation. Each rank is a distinct non-linear function. This means that the rank layer itself does not follow the same paradigm as neural nets that usually use a single identical activation function on all elements. The rank function uses different ones for each element, and hence we should not compute a single function, but a matrix of kernel functions. The output of a linear combination of ranks is for the two inputs $Sort(W^T \mathbf{x}) = \phi(W^T \mathbf{x})$ and $Sort(W^T \mathbf{x}') = \phi(W^T \mathbf{x}')$. The weights are independent and random $col_j(W) \in \mathcal{N}(\mathbf{0}, I_{M \times M})$, and $W \in \mathbb{R}_{M \times N}$. Here we used the notation ϕ to stress the fact that the sorting process is a multiple output function i.e. $\phi : \mathbb{R}_{N \times 1} \rightarrow \mathbb{R}_{N \times 1}$. Then,

$$K(\mathbf{x}, \mathbf{x}') = \mathcal{E}_W(\phi(W^T \mathbf{x}') \phi^T(W^T \mathbf{x})). \quad (4.15)$$

Or, equivalently:

$$K(\mathbf{x}, \mathbf{x}') = \frac{1}{(2\pi)^{N^2/2}} \int_{\mathbb{R}^2} \phi(W^T \mathbf{x}') \phi^T(W^T \mathbf{x}) e^{-tr(W^T W)/2} (dW)^\wedge. \quad (4.16)$$

Remarkably, the integral depends only on the angle between the inputs \mathbf{x} and \mathbf{x}' regardless of the input dimension M i.e.

$$\theta = \cos^{-1} \frac{\mathbf{x}^T \mathbf{x}'}{\|\mathbf{x}\| \|\mathbf{x}'\|}. \quad (4.17)$$

For $N = 2$, Equation 4.16 reduces to:

$$K(\mathbf{x}, \mathbf{x}') = \frac{\|\mathbf{x}\| \|\mathbf{x}'\|}{\pi} \begin{bmatrix} \pi \cos \theta + \sin |\theta| - |\theta| \cos \theta & |\theta| \cos \theta - \sin |\theta| \\ |\theta| \cos \theta - \sin |\theta| & \pi \cos \theta + \sin |\theta| - |\theta| \cos \theta \end{bmatrix}. \quad (4.18)$$

Refer to Equation A.40 in the Appendix for the proof. Following the notation in [26], the kernel matrix at the layer ℓ is $K^\ell(\mathbf{x}, \mathbf{x}')$. Given that the first layer is an activated input we have for first output:

$$\mathbf{z}_{d_1 \times 1}^{\ell=1}(\mathbf{x}) = W_\ell^T \Phi^\ell(\mathbf{x}) + \mathbf{b}_{d_1 \times 1}; d_0 = L_0 N_0, \quad (4.19)$$

where the activation vector $\Phi^\ell(\mathbf{z}^{\ell-1}) \in \mathbb{R}_{d_{\ell-1} \times 1}$ consists of $L_{\ell-1}$ vector activations of size $N_{\ell-1}$, and $\mathbf{z}^{\ell-1} = \mathbf{x}$. In other terms, the activation vector in Equation 4.19 is:

$$\Phi_{d_0 \times 1}^1(\mathbf{x}) = \begin{bmatrix} \phi(\mathbf{x}_{1:N_0}) \\ \phi(\mathbf{x}_{N_0+1:2N_0}) \\ \vdots \\ \phi(\mathbf{x}_{L_0 N_0 - N_0 + 1 : L_0 N_0}) \end{bmatrix}, \quad (4.20)$$

and more generally,

$$\Phi_{d_{\ell-1} \times 1}^\ell(\mathbf{z}^{\ell-1}) = \begin{bmatrix} \phi(\mathbf{z}_{1:N_{\ell-1}}^{\ell-1}) \\ \phi(\mathbf{z}_{N_{\ell-1}+1:2N_{\ell-1}}^{\ell-1}) \\ \vdots \\ \phi(\mathbf{z}_{L_{\ell-1} N_{\ell-1} - N_{\ell-1} + 1 : L_{\ell-1} N_{\ell-1}}^{\ell-1}) \end{bmatrix}; d_{\ell-1} = L_{\ell-1} N_{\ell-1}. \quad (4.21)$$

Then the output at layer ℓ is:

$$\mathbf{z}_{d_\ell \times 1}^\ell = W_\ell^T \Phi_{d_{\ell-1} \times 1}^\ell(\mathbf{z}^{\ell-1}) + \mathbf{b}_{d_\ell \times 1}. \quad (4.22)$$

We further suppose that the inputs are all i.i.d. $\mathbf{x}, \mathbf{x}' \sim \mathcal{N}(\mathbf{0}, I)$. We will denote the dependence of the output of the ℓ^{th} layer on the input \mathbf{x} and test point \mathbf{x}' by $\mathbf{z}^\ell(\mathbf{x})$ and $\mathbf{z}^\ell(\mathbf{x}')$ respectively. Then by using the Central Limit Theorem (CLT), we will have the output $\mathbf{z}^\ell \sim \mathcal{GP}(\mathbf{0}, K^\ell)$ provided $d_\ell \rightarrow \infty$. We can thus use Equation C.6 to compute:

$$\mathcal{E} \left(\mathbf{z}_{d_\ell \times 1}^\ell(\mathbf{x}) \left(\mathbf{z}_{d_\ell \times 1}^\ell(\mathbf{x}') \right)^T \right) = K^\ell(\mathbf{x}, \mathbf{x}'). \quad (4.23)$$

$$K^\ell(\mathbf{x}, \mathbf{x}') = \sigma_b^2 I_{d_\ell} + \sigma_w^2 I_{d_\ell} \text{tr} \left(\mathcal{E} \left(\Phi(\mathbf{z}^{\ell-1}(\mathbf{x})) \Phi^T(\mathbf{z}^{\ell-1}(\mathbf{x}')) \right) \right). \quad (4.24)$$

The trace equals on the right hand side becomes

$$\sum_{k=0}^{L_{\ell-1}-1} \text{tr} \left(\mathcal{E} \left(\phi \left(\mathbf{z}_{k \cdot N_{\ell-1} + 1 : (k+1) N_{\ell-1}}^{\ell-1}(\mathbf{x}) \right) \phi^T \left(\mathbf{z}_{k \cdot N_{\ell-1} + 1 : (k+1) N_{\ell-1}}^{\ell-1}(\mathbf{x}') \right) \right) \right).$$

If we assume that the angle between the same subsections of the vectors $\mathbf{z}^{\ell-1}(\mathbf{x})$ and $\mathbf{z}^{\ell-1}(\mathbf{x}')$ to be constant on average and substituting in equation A.19, then the above trace becomes

$$\begin{aligned} & L_{\ell-1} \left(N_{\ell-1} \frac{\sqrt{q_{\mathbf{x}}^{\ell-1}}}{\sqrt{L_{\ell-1}}} \frac{\sqrt{q_{\mathbf{x}'}^{\ell-1}}}{\sqrt{L_{\ell-1}}} (\cos \theta_{\ell-1} + \mathcal{O}(\theta_{\ell-1}^3)) \right) \\ &= N_{\ell-1} \sqrt{q_{\mathbf{x}}^{\ell-1} q_{\mathbf{x}'}^{\ell-1}} (\cos \theta_{\ell-1} + \mathcal{O}(\theta_{\ell-1}^3)), \end{aligned}$$

where $q_{\mathbf{x}}^{\ell-1}$ and $\cos \theta_{\ell-1}$ are the variance and correlation in the $(\ell - 1)^{\text{th}}$ layer. It then follows that

$$K^{\ell}(\mathbf{x}, \mathbf{x}') = \sigma_b^2 I_{d_{\ell}} + \sigma_{w, \ell-1}^2 N_{\ell-1} \sqrt{q_{\mathbf{x}}^{\ell-1} q_{\mathbf{x}'}^{\ell-1}} (\cos \theta_{\ell-1} + \mathcal{O}(\theta_{\ell-1}^3)) I_{d_{\ell}}. \quad (4.25)$$

This kernel recursion has a trivial fixed point when $\theta = 0$ and by if the inputs have the same variance $q_{\mathbf{x}'}^{\ell-1} = q_{\mathbf{x}}^{\ell-1}$, the recursion becomes:

$$K^{\ell}(\mathbf{x}, \mathbf{x}') = q_{\mathbf{x}}^{\ell} = \sigma_b^2 I_{d_{\ell}} + \sigma_{w, \ell-1}^2 N_{\ell-1} q_{\mathbf{x}}^{\ell-1} I_{d_{\ell}}. \quad (4.26)$$

The variance $q_{\mathbf{x}}^{\ell-1} = q_{\mathbf{x}}^{\ell} = q$ has a fixed point when the following conditions are satisfied:

$$q = \frac{\sigma_b^2}{1 - N_{\ell-1} \sigma_{w, \ell-1}^2}, \quad (4.27)$$

or at the point

$$(\sigma_b^2, \sigma_{w, \ell}^2) = \left(0, \frac{1}{N_{\ell}} \right). \quad (4.28)$$

The latter condition makes the weight and bias variance independent of the layer variance and it corresponds to the Edge Of Chaos (EOC) [17] point.

4.2 Joint Linear-Rank Filters

By combing Rank filters additively with Linear filters we get a more flexible space that is the union of both inner circles in Figure 4.7.

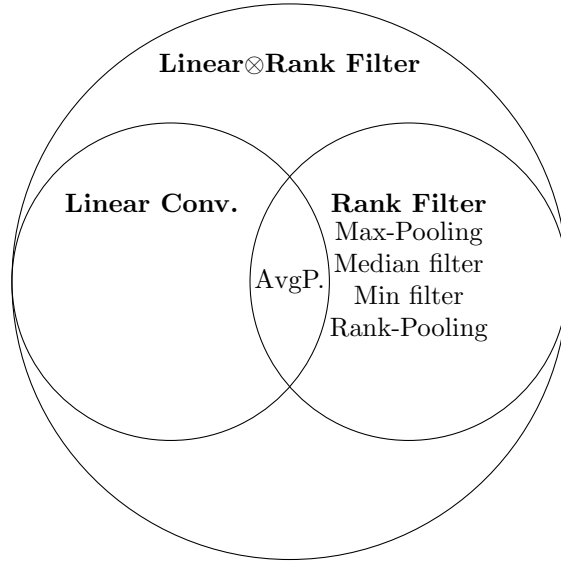


Figure 4.7: Venn diagram showing which method can be reached by rank filters, convolutional layers and Rank+Linear Layers. Average pooling can both be reached by convolutional layers and Rank layers if all weights equal to $1/d_{in}$, where d_{in} is the total size of the weight. Average pooling and Max-pooling are static and hence can't learn.

However it would be better to combine them multiplicatively so as to truly unlock the potential of ranks to generate an even richer space (the outer circle in Figure 4.7). Because there is a high correlation between ranks it seems reasonable to use $2 \cdot N$ weights (N is the number of elements in a patch \mathbf{x}) instead of N^2 to model the the joint rank and linear model. To do this we use the following formula:

$$y = \mathbf{w}_r^T P_x^T (\mathbf{x} \circ \mathbf{w}_l) + b, \quad (4.29)$$

where P_x^T are the permutations wrt to \mathbf{x} and not $\mathbf{x} \circ \mathbf{w}_l$. The following feed-forward connectivity graph is made by combining both graphs in Figures 2.1 and 4.1.

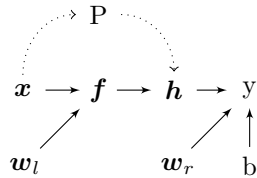


Figure 4.8: Joint Linear-Rank filter feed-forward connection graph. Here, $\mathbf{f} = \mathbf{x} \circ \mathbf{w}_l$, $P = P_x$, $\mathbf{h} = P^T \mathbf{f}$ and $y = \mathbf{w}_r^T \mathbf{h} + b$.

Equation 4.29 is equivalent to

$$y = \mathbf{w}_l^T P (\mathbf{s} \circ \mathbf{w}_r) + b. \quad (4.30)$$

Proof: We start by substituting in $\mathbf{s} = P_x^T \mathbf{x}$ into Equation 4.30. Then

$$y = \mathbf{w}_l^T P_x (P_x^T \mathbf{x} \circ \mathbf{w}_r) + b.$$

Permutation matrices distribute themselves on Hadamard products so

$$y = \mathbf{w}_l^T (P_x P_x^T \mathbf{x} \circ P_x \mathbf{w}_r) + b = \mathbf{w}_l^T (\mathbf{x} \circ P_x \mathbf{w}_r) + b$$

. We can exchange vectors \mathbf{w}_l and $P_x \mathbf{w}_r$ again using a Hadamard product property: $y = (P_x \mathbf{w}_r)^T (\mathbf{x} \circ \mathbf{w}_l) + b$. The result follows when distributing the transposition. \square

This duality property may be exploited during forward and back-propagation and to simplify gradient calculation. To illustrate the joint filter⁴ with $\mathbf{w}_r = [r_i]$ and $\mathbf{w}_l = [l_i]$ we have:

$$\begin{bmatrix} 3 \\ 6 \\ -2 \\ 7 \\ 8 \\ 0 \\ 0 \\ 4 \\ 1 \end{bmatrix} \xrightarrow{P^T} \begin{bmatrix} -2 \\ 0 \\ 0 \\ 1 \\ 3 \\ 4 \\ 6 \\ 7 \\ 8 \end{bmatrix} \xrightarrow{\circ \mathbf{w}_r} \begin{bmatrix} -2 \cdot r_1 \\ 0 \cdot r_2 \\ 0 \cdot r_3 \\ 1 \cdot r_4 \\ 3 \cdot r_5 \\ 4 \cdot r_6 \\ 6 \cdot r_7 \\ 7 \cdot r_8 \\ 8 \cdot r_9 \end{bmatrix} \xrightarrow{P} \begin{bmatrix} 3 \cdot r_5 \\ 6 \cdot r_7 \\ -2 \cdot r_1 \\ 7 \cdot r_8 \\ 8 \cdot r_9 \\ 0 \cdot r_2 \\ 0 \cdot r_3 \\ 4 \cdot r_6 \\ 1 \cdot r_4 \end{bmatrix} \xrightarrow{\circ \mathbf{w}_l} \begin{bmatrix} 3 \cdot r_5 \cdot l_1 \\ 6 \cdot r_7 \cdot l_2 \\ -2 \cdot r_1 \cdot l_3 \\ 7 \cdot r_8 \cdot l_4 \\ 8 \cdot r_9 \cdot l_5 \\ 0 \cdot r_2 \cdot l_6 \\ 0 \cdot r_3 \cdot l_7 \\ 4 \cdot r_6 \cdot l_8 \\ 1 \cdot r_4 \cdot l_9 \end{bmatrix} \xrightarrow{\text{Sum}} y$$

Our proposed joint filter is thus described by the following steps. Let the blocks

$$\mathcal{H}_{h_{win} \times w_{win}}[a][b][c][k][n] = \text{vec}^{-1} \left(P_{\text{vec}(X^{\text{slice}})} \left(\text{vec} \left(\mathbf{w}_{:, :, c, k}^{\text{rank}} \circ S_{h_{win} \times w_{win}}[a][b][c][n] \right) \right) \right), \quad (4.31)$$

where the input block channel slice $X^{\text{slice}}[a][b][c][n] \in \mathbb{R}_{h_{win} \times w_{win}}$ is defined in Equation 4.11, and the sorted slices $S_{h_{win} \times w_{win}}[a][b][c][n]$ are defined in Equation 4.10. Then each output element is (omitting the bias):

$$y[a][b][k][n] = \underset{\text{Dims: } H, W, C_{in}}{\text{Reduce}} \mathbf{w}_{:, :, :, k}^{\text{lin}} \mathcal{H}_{h_{win} \times w_{win} \times C_i}[a][b][k][n]. \quad (4.32)$$

4.2.1 Stochastic Joint Linear-Rank Filters

To ensure a mean of 1 for one of the weight types (linear or ranks), we could pass either one of the weights through a softmax layer \mathcal{S} before the dot product. Adding the two possibilities, we end up with:

$$y = (\mathbf{w}_r^{(1)})^T P^T \left(\mathbf{x} \circ \mathcal{S}(\mathbf{w}_l^{(1)}) \right) + (\mathbf{w}_l^{(2)})^T P \left(\mathbf{s} \circ \mathcal{S}(\mathbf{w}_r^{(2)}) \right) + b. \quad (4.33)$$

The above formula is very interesting since it gives the possibility of using the Gumbel-Max trick[27] to sample from a discrete distribution with probabilities $\mathcal{S}(\mathbf{w})$. Intuitively this random sampling may lead to good generalization as other random algorithms did. Examples

⁴This example refers to the second FoV in Figure 4.2

of random and pseudo-random algorithms include Dropout[44], and Fractional-MaxPooling [15]. We are however only going to explore layers based on the following formulas:

$$y = (\mathbf{w}_r)^T P^T (\mathbf{x} \circ \mathcal{S}(\mathbf{w}_l)), \quad (4.34)$$

$$y = (\mathbf{w}_l)^T P^T (\mathbf{x} \circ \mathcal{S}(\mathbf{w}_r)), \quad (4.35)$$

$$y = (\mathcal{S}(\mathbf{w}_l))^T P^T (\mathbf{x} \circ \mathcal{S}(\mathbf{w}_r)). \quad (4.36)$$

The expected value of the gradient wrt the weights are:

$$\nabla_{\mathbf{w}_l} y = P (\mathbf{s} \circ \mathbf{w}_r) = \mathbf{x} \circ (P_{\mathbf{x}} \mathbf{w}_r). \quad (4.37)$$

$$\nabla_{\mathbf{w}_r} y = P_{\mathbf{x}}^T (\mathbf{x} \circ \mathbf{w}_l) = \mathbf{s} \circ (P_{\mathbf{x}}^T \mathbf{w}_l). \quad (4.38)$$

4.2.1.1 The Gumbel Max-Trick

Let $\mathbf{u} \in \mathbb{R}_{A \times 1}$ and $\mathbf{u} \sim \mathcal{U}(0, 1)$. Then the discrete distribution sample $\mathbf{g}_{A \times 1}$ with probabilities $\boldsymbol{\pi}_{A \times 1}$ can be sampled from the maximum index of a sample a Gumbel distribution $\mathbf{z}_{A \times 1} \sim \mathcal{G}(\boldsymbol{\pi}_{A \times 1})$, where

$$\mathbf{z} = \boldsymbol{\pi}_{A \times 1} - \ln(-\ln(\mathbf{u})).$$

A sample point $\mathbf{g}_{A \times 1}$ has a 1 at the k^{th} element and zeros for all other entries i.e. $\mathbf{g}_{A \times 1} = \boldsymbol{\delta}_k$. The value k is equal to the index of the maximum of \mathbf{z} , in other words,

$$k = \arg \max_{i \in [1, A]} z_i.$$

The vector follows a Multinomial distribution, $\mathbf{g} \sim \mathcal{M}(\boldsymbol{\pi})$. We define $\tilde{\boldsymbol{\pi}}_K$ a sample made from the average of K independent discrete samples $\mathbf{g}^{(j)} = \boldsymbol{\delta}_{k_j}$ with average $\boldsymbol{\pi}$. Hence,

$$\tilde{\boldsymbol{\pi}}_K = \frac{1}{K} \sum_{i=1}^K \boldsymbol{\delta}_{k_i}, \quad (4.39)$$

where $k_j = \arg \max_{k \in [1, A]} z_k^{(j)}$, and $j \in [1, K]$. In the limit we have $\boldsymbol{\pi}_{\infty} = \boldsymbol{\pi}$. We have now all we need to describe our layers. Since a softmax layer can be interpreted as discrete probability vector $\boldsymbol{\pi}$, we set $\mathcal{S}(\mathbf{w}) = \boldsymbol{\pi}$ and approximate the softmax function with :

$$\mathcal{S}(\mathbf{w}) \approx \tilde{\mathcal{S}}_K(\mathbf{w}) = \tilde{\boldsymbol{\pi}}_K. \quad (4.40)$$

The covariance matrix of $\tilde{\boldsymbol{\pi}}_K$ is:

$$\mathcal{E} \left((\tilde{\boldsymbol{\pi}}_K - \boldsymbol{\pi}) (\tilde{\boldsymbol{\pi}}_K - \boldsymbol{\pi})^T \right) = \frac{\text{Diag}(\boldsymbol{\pi}) - \boldsymbol{\pi} \boldsymbol{\pi}^T}{K}. \quad (4.41)$$

Why are we using averages of discrete samples, why not a Gaussian variable with an average $\boldsymbol{\pi}$? The reason is that there is a good chance that one of the elements of $\mathcal{S}(\mathbf{w})$ will have 0 components provided K is not too large. If K is large, then effectively, we should use a

Gaussian. Zero components are desirable to make sparse random derivative updates⁵, similar to the Dropout concept. Random sparsity may be a good thing for training neural nets. But how to choose K ? To answer this question, we need to measure the average sparsity given that the probabilities of activity (an input that is not equal to 0) are $\boldsymbol{\pi}$. Carrying the expectations, we get a function of the average sparsity wrt to K , and we can modify K to obtain the desired sparsity. This method may suffer from having a probability that is too low. Then activations may never go through effectively, making the neuron connection disappear. This situation is analogous to the multi-armed bandit problem that has a learned probability too small for the player to explore. Let us model an activity neuron as a realization of a multinomial distribution $\mathbf{g} \sim \mathcal{M}(\boldsymbol{\pi})$. Since, $\tilde{\boldsymbol{\pi}}_K = \frac{1}{K} \sum_{i=1}^K \boldsymbol{\delta}_{k_i}$, the probability of $\tilde{\boldsymbol{\pi}}_K$ having exactly $N - 1$ zeros (if N is the length of the vector) is:

$$\mathcal{P}(Z = N - 1) = \mathbf{1}^T \boldsymbol{\pi}^{\circ K} = \|\boldsymbol{\pi}\|_K^K, \text{ for } K \geq 1, \quad (4.42)$$

where Z equals the numbers of zeros in $\tilde{\boldsymbol{\pi}}_K$. if we denote $\mathcal{P}(Z = z)$, then

$$\mathcal{P}(Z = N - 2) = K \sum_{m=1}^{K-2} \sum_{1 \leq i < j \leq N} \pi_i^{K-m} \pi_j^m. \quad (4.43)$$

Generally, if $\mathcal{P}(Z = N - a)$, where a is the number of non-zero activations then

$$\mathcal{P}(Z = N - a) = \sum_{\substack{k_{i_1} + k_{i_2} + \dots + k_{i_a} = K \\ 1 \leq i_1 < i_2 < \dots < i_a \leq N \\ k_i \geq 1}} \binom{K}{k_{i_1}, k_{i_2}, \dots, k_{i_a}} \pi_{i_1}^{k_{i_1}} \pi_{i_2}^{k_{i_2}} \dots \pi_{i_a}^{k_{i_a}}. \quad (4.44)$$

The average activation ratio is thus:

$$\mathcal{E}_Z \left(\frac{N - Z}{N} \right) = \sum_{n=0}^{N-1} \frac{N - n}{N} \cdot \mathcal{P}(Z = n) = 1 - \sum_{n=1}^{N-1} \frac{n}{N} \cdot \mathcal{P}(Z = n). \quad (4.45)$$

The average activation ratio is the complement of the average sparsity ratio $\mathcal{E}_Z \left(\frac{Z}{N} \right)$:

$$\mathcal{E}_Z \left(\frac{Z}{N} \right) = \sum_{n=1}^{N-1} \frac{n}{N} \cdot \mathcal{P}(Z = n). \quad (4.46)$$

The expression above is rather complicated whenever $N > 2$, so we will focus rather on the expected activity of an element independently from the others. Then the probabilities that the i^{th} element of $\tilde{\boldsymbol{\pi}}_K$ is activated (larger than 0) is:

$$\mathcal{P} \left(\tilde{\pi}_K^{(i)} \right) = 1 - (1 - \pi_i)^K. \quad (4.47)$$

Thus if we want the probability of a neuron of being activated with a constant α , then

$$K = \frac{\ln(1 - \alpha)}{\ln(1 - \pi_i)} = \mathcal{O} \left(\frac{\alpha}{\pi_i} \right). \quad (4.48)$$

⁵ This idea came after reading the articles recommended by Professor Yongyi Mao.

This means that for small π_i we may be forced to use a large K to ensure gradients update its value,

$$K \approx \frac{-\ln(1 - \alpha)}{\pi_i}. \quad (4.49)$$

Furthermore, the realizations for the weights need to be independent for each batch element. This is intensive for the GPU, and hence we only implemented identical random weights for each batch element⁶. This inefficiency is quite problematic since the average across the batch dimension would be quite high, and thus would degrade the learning significantly. To counteract this degradation, we would have to choose a large K , but this would guarantee that there is no sparsity, and hence there would be no point in using the Gumbel max-trick since we could use samples from a Gaussian distribution instead.

4.2.1.2 Application of the Gumbel Max-Trick for non-probabilistic vectors

The trick can also be applied in normalization contexts such as BN for example. Let a normalization layer output \mathbf{y} from an input \mathbf{w} :

$$\mathbf{y} = \frac{\mathbf{w}}{\mathbf{1}^T (\text{abs} \circ \mathbf{w})} = \frac{\mathbf{w}}{\|\mathbf{w}\|_1}. \quad (4.50)$$

Note the striking similarities with the Softmax operation:

$$\mathcal{S}(\mathbf{w}) = \frac{(\exp \circ \mathbf{w})}{\mathbf{1}^T (\exp \circ \mathbf{w})}. \quad (4.51)$$

Hence, we can reparametrize the absolute value of \mathbf{y} with a Softmax operator:

$$\mathbf{y} = (\text{sign} \circ \mathbf{w}) \circ \mathcal{S}(\ln \circ \text{abs} \circ \mathbf{w}), \quad (4.52)$$

or to alleviate the notation:

$$\mathbf{y} = (\text{sign}(\mathbf{w})) \circ \mathcal{S}(\ln |\mathbf{w}|). \quad (4.53)$$

Since we already defined a way to generate sparse random positive outputs depending on K (see Eq. 4.40), we have a formula to generate sparse random outputs for the training steps:

$$\tilde{\mathbf{y}} = (\text{sign}(\mathbf{w})) \circ \tilde{\mathcal{S}}_K(\ln |\mathbf{w}|). \quad (4.54)$$

This method can be extrapolated to any layer without normalization since

$$\mathbf{w} = \|\mathbf{w}\|_1 (\text{sign}(\mathbf{w})) \circ \mathcal{S}(\ln |\mathbf{w}|).$$

Then, at training time we can use anywhere in a neural net⁷:

$$\tilde{\mathbf{w}} = \|\mathbf{w}\|_1 (\text{sign}(\mathbf{w})) \circ \tilde{\mathcal{S}}_K(\ln |\mathbf{w}|). \quad (4.55)$$

⁶ We could test the idea by generating the samples per batch element on the CPU. But this would be highly inefficient and would increase the bandwidth used to process each batch element individually.

⁷ This equation is highlighted because it is very elegant and has a great potential. However, this is only a theoretical formula for now, and is not tested experimentally as it would require unique initializations per batch.

Note that $\tilde{\mathcal{S}}_K(\ln |\mathbf{w}|)$ is associated to the sampled from max indices of the realizations of the Gumbel distribution $\mathbf{z} \sim \mathcal{G}(\ln |\mathbf{w}|)$, where

$$\mathbf{z} = \ln(\ln |\mathbf{w}| \oslash \ln(\mathbf{u}^{\circ(-1)})).$$

Equations 4.54 and 4.55 do not generate outputs constrained be positive or sum to 1. Unfortunately, it will not be studied in-depth as it requires preferably different instantiations for every batch index, and thus, it considerably increases computation time on the GPU. We **did implement** this idea in the next section, although in **a simpler form** where we didn't use (rather arbitrarily) the logarithm parametrization. It is interesting to note that the ℓ_1 norm occurred here naturally in the context of generating sparse random vectors. There is probably a link to the fact that an ℓ_1 normalization is related to the ℓ_1 regularisation commonly used in LASSO, a technique aimed at generating a sparse regression.

4.2.1.3 Random magnitudes for Joint Linear-Rank filters

Now let the filter FoV area $A = h_{win} \cdot w_{win}$. To make the softmax function have a mean of 1, we can scale it by a factor of A . Let us define the block channel slice Softmax operation as being:

$$\mathcal{S}^s(\mathbf{w}) = [\mathcal{S}(\mathbf{w}_{:, :, c_i, c_o})]_{c_i \in [1, Ch_{in}], c_o \in [1, Ch_{in}]}, \quad (4.56)$$

then, in the prediction step, we can replace \mathcal{S} in Equations 4.34, 4.35 and 4.36. To compute the forward pass at the training step, we use the approximation on \mathcal{S}^s , note that an approximation is preferably unique for every batch index n .

$$y[n] = (\mathbf{w}_{rank})^T P^T \left(\mathbf{x} \circ \tilde{\mathcal{S}}_K^s(\mathbf{w}_{linear})[n] \right). \quad (4.57)$$

$$y[n] = (\mathbf{w}_{linear})^T P^T \left(\mathbf{x} \circ \tilde{\mathcal{S}}_K^s(\mathbf{w}_{rank})[n] \right). \quad (4.58)$$

$$y[n] = \left(\tilde{\mathcal{S}}_K^s(\mathbf{w}_{linear})[n] \right)^T P^T \left(\mathbf{x} \circ \tilde{\mathcal{S}}_K^s(\mathbf{w}_{rank})[n] \right). \quad (4.59)$$

In this section we use only the random approximation of $\mathcal{S}(\mathbf{w})$, so the normalizing factor is $\mathbf{1}^T (\exp \circ \mathbf{w}) = \|\exp \circ \mathbf{w}\|_1$. If the gradients pushes the Softmax activation towards 0 and negative values, then this may cause weights to grow to very large negative numbers. This is an argument to use the stochastic generator of 4.55 instead.

4.3 Rank+Linear Convolutions

The joint linear-rank filter is complicated (at least in implementation) because of the multiplicative double filtering. The easier alternative is to additively combine Equation 4.1 with the normal dot product. We call this Rank+Linear filters. In essence, the output is given by

$$y = \mathbf{w}_r^T \mathbf{s} + \mathbf{w}_l^T \mathbf{x} + b. \quad (4.60)$$

This is already used in [31] because the net evolved use an addition of max-pooling and average pooling.

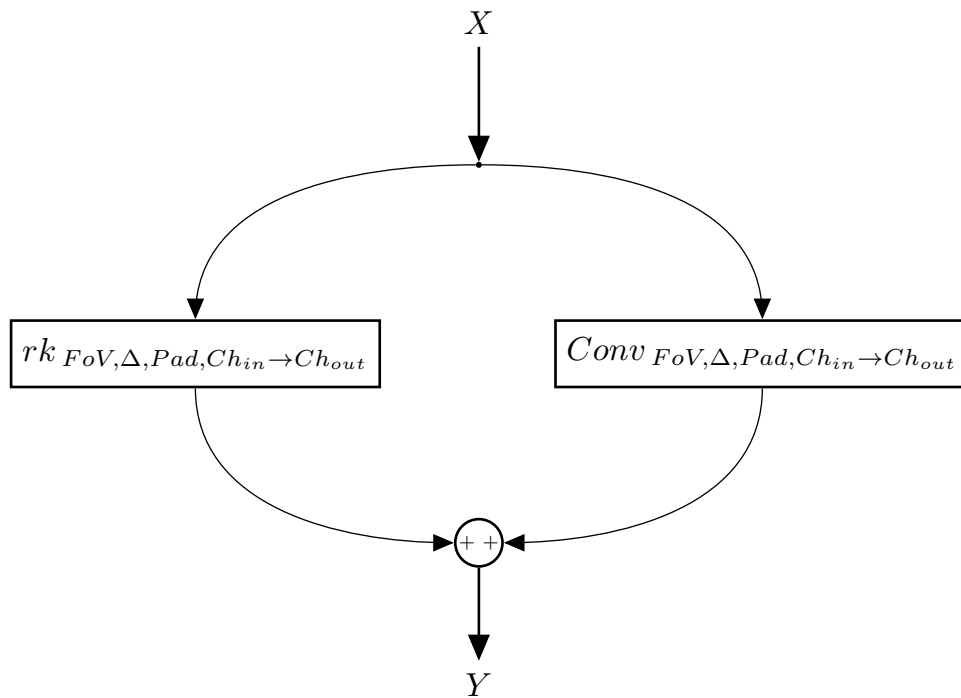


Figure 4.9: The Channel output parameters for both the rank filtering and the convolutional filtering may be different. In this case where we use an addition as the junction, the output channel equals the maximum of the output channel parameters. If we concatenate, the output channel is the sum of both channel output parameters.

4.4 Transposed Rank Filters

We know that the authors in [50] have implemented the max-Unpooling layers where the “switches” correspond to an index from where the maximum came from. We are going to go through the same inverse process and use all of the permutation indices that were recorded to compute the code (y).

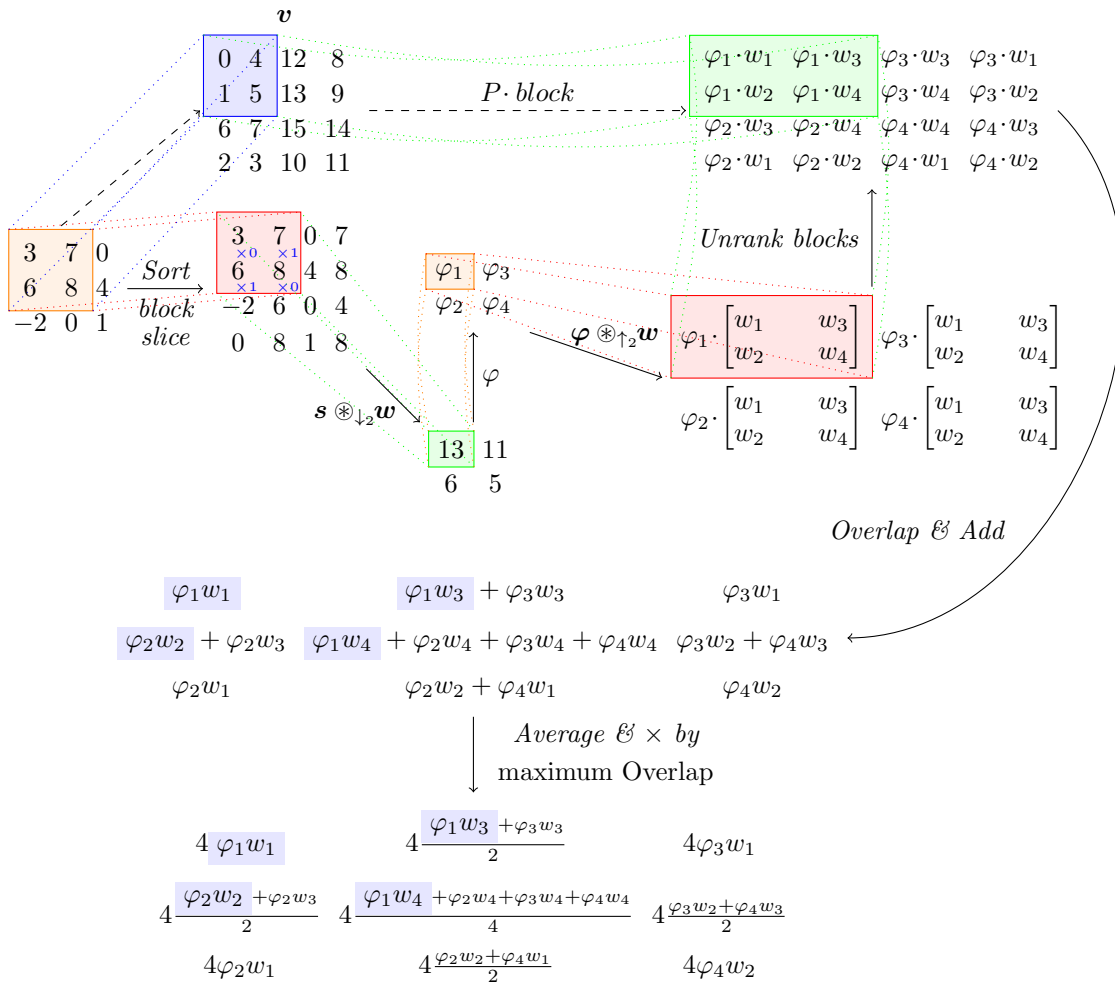


Figure 4.10: Activated Rank filter example followed by a transposed rank filter with a 2×2 field of view. Each channel slice permutations indices “ \mathbf{v} ” are saved into memory. We perform a transposed convolution of the activation followed by a permutation. Finally, to compute the output, we overlap & add the window values followed by rescaling. This particular example doesn’t show the cross-channel computations (it show’s effectively the exact output if there is a single input and output channel.), only a single term is computed.

4.4.1 Avoiding artefacts

Note that in Figure 4.10, we used an example without padding the input. If we were to pad the input, we would need to crop the extra pixels out of the output at the very last step. Reflective padding is desirable in the coder because, if we do not use padding, the rank-based decoder will generate artifacts along the edges of the reconstruction. This edge artifact is caused by the difference in the number of terms that arise by using the overlap & add technique (it has nothing to do with zero padding). Inside the region, we may get checkerboard artifacts depending on the stride and window size, if any of the filter sides is coprime with the stride in its own dimension[45]. To mitigate this, following the overlap and add method, we average by the amount of overlap and scale by the maximum overlap. A solution to remove edge effects is to pad the input enough at the coder and then crop out at the decoder. Other solutions are possible. For example, in the U-net[32], a semantic labeling net, padding can be avoided, and artifacts never seem to appear possibly for two of the net characteristics. First convolutions following the deconvolution in the decoder permit to output a smaller size image removing the need of using cropping. Secondly, the cropped skip-connections between the coder blocks and decoder blocks also yield a smaller dimension size to match the decoder block’s input. We do not use convolutions to reduce our output size in the decoders. It is important to note that we found that using reflective padding in the coder side was much better than zero padding for rank filters. Hence this configuration is used by default.

So we need an overlap normalization to :

1. Avoid edge artifacts when we pad with values that have the same mean as the edge locally. Zero-padding is a bad choice and reflection-padding is a good one.
2. Avoid inner periodical artefacts (checkerboard artefacts) whenever the filter’s stride is coprime with it’s length.

Note that the only rank filter that is not affected by these two types of artifacts are filters that have strides equal to the filter size. Obviously, since there is no overlap, we do not normalize by the overlaps. The types of rank filters that have the same sides as the stride include rank-pooling and max-pooling layers.

The scaling and overlap normalizing help only for the transposed rank filter because they can’t code static locations (The P’s are almost always different.). Hence, the weights of the transposed rank filter are always mixed in an output block. This will create an uneven distribution of power across the different overlapping regions, thus generating a checkerboard artifact. In the case of all-linear nets, the permutations are static (the weights always fall at the same location), so even if the distribution of weights isn’t equally distributed, the net can learn to adapt the power of these weights. Consequently, no checkerboard artifacts appear. To illustrate this, I will give a simple 1D example where the window size is 2, and the overlap is 1. In this example, we are minimizing the MSE. We try to estimate the output $\mathbf{y}_{3 \times 1}$ with the previous layer output $\mathbf{x}_{2 \times 1}$. Note that the artifact here is an edge artifact. Hence, for the linear case:

$$\begin{bmatrix} x_1 \\ x_1 + x_2 \\ x_2 \end{bmatrix} = A\mathbf{x} = \mathbf{y}_{3 \times 1}.$$

We know that $A = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}$. A is a tall matrix of rank-2⁸ Hence the solution to the MSE problem is:

$$A^\dagger A \mathbf{x}_{opt} = \mathbf{x}_{opt} = A^\dagger \mathbf{y}_{3 \times 1} = \frac{1}{3} \begin{bmatrix} 2y_1 + y_2 - y_3 \\ -y_1 + y_2 + 2y_3 \end{bmatrix}.$$

If

$$\mathbf{y} \sim \mathcal{N}(\mu_y \mathbf{1}_{3 \times 1}, \sigma_y^2 \mathbf{I}_{3 \times 3}),$$

then

$$\mathbf{w} \sim \mathcal{N}\left(\mu_y \mathbf{1}_{2 \times 1}, \frac{\sigma_y^2}{3} \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}\right).$$

The estimated output is:

$$A \mathbf{w}_{opt} = \hat{\mathbf{y}}$$

Then

$$\mathcal{E}(\hat{\mathbf{y}}) = \frac{2}{3} \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}^T \mu_y$$

The average output variance is given by:

$$\mathcal{V}(\hat{\mathbf{y}}) = \frac{2}{3} \mathbf{1}_{3 \times 1} \sigma_y^2.$$

The variance result above shows that indeed the power is equally distributed among its indices. Now consider the rank filtering case. In this case, there are 4 outcomes for the matrix A corresponding to the uncontrollable mixing of the permutations. Under the expected value, the active transformation amounts to an average of all the outcomes: We know that

$$\bar{A} = \frac{1}{4} \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} + \frac{1}{4} \begin{bmatrix} 0 & 1 \\ 2 & 0 \\ 0 & 1 \end{bmatrix} + \frac{1}{4} \begin{bmatrix} 1 & 0 \\ 0 & 2 \\ 1 & 0 \end{bmatrix} + \frac{1}{4} \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} \\ 1 \\ \frac{1}{2} \end{bmatrix} \begin{bmatrix} 1 & 1 \end{bmatrix}.$$

Immediately we see that \bar{A} is a rank-1 deficient matrix with a pseudoinverse of

$$\bar{A}^\dagger = \frac{1}{6} \begin{bmatrix} 1 & 2 & 1 \\ 1 & 2 & 1 \end{bmatrix}.$$

Thus, we have an infinity of solutions provided that for both weights:

$$x_1^{opt} = x_2^{opt} = x^{opt} = \frac{1}{2} (x_1 + x_2) = \frac{1}{6} (y_1 + 2y_2 + y_3).$$

The mean and variance of w^{opt} are $\mu = \frac{2}{3} \mu_y$ and $\sigma^2 = \frac{1}{6} \sigma_y^2$. It follows then that the estimated output is:

$$\hat{\mathbf{y}} = \bar{A} \mathbf{x}_{opt} = \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}^T x^{opt}.$$

⁸ Here the term “rank” is the dimension spanned by the matrix. It isn’t used in the sense of order statistics.

Then

$$\mathcal{E}(\hat{\mathbf{y}}) = \frac{2}{3} \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}^T \mu_y$$

The average output variance are not equally distributed as shown by:

$$\mathcal{V}(\hat{\mathbf{y}}) = \frac{1}{6} \begin{bmatrix} 1 & 4 & 1 \end{bmatrix}^T \sigma_y^2.$$

Let's see what happens when we adjust the power, then the average transfer matrix is:

$$\bar{A}_a = \mathbf{1}_{3 \times 2}.$$

Yet again, this is a deficient matrix and a linear combination of the weights solves the problem:

$$w_1^{opt} = w_2^{opt} = (w_1 + w_2) = \frac{1}{3} (y_1 + y_2 + y_3)$$

The mean and variance of w^{opt} are $\mu = \mu_y$ and $\sigma^2 = \frac{1}{3}\sigma_y^2$. The estimated output is

$$\hat{\mathbf{y}}_a = \bar{A}_a \mathbf{w}_{opt} = \frac{2}{3} \mathbf{1}_{3 \times 1} w_{opt}.$$

Finally, the mean and output variance are:

$$\mathcal{E}(\hat{\mathbf{y}}) = \frac{2}{3} \mathbf{1}_{3 \times 1} \mu_y.$$

and

$$\mathcal{V}(\hat{\mathbf{y}}) = \frac{4}{9} \mathbf{1}_{3 \times 1} \sigma_y.$$

Thus by adjusting the power in this situation with the overlap, we set the output variance to be identical for all locations. This simple example generalizes for any transposed filtering. Since usually, the bias in the average is not a problem for learning (assumed to be 0), then we are only left to address the bias of the power. To illustrate the necessity of the overlap scaling, we included [Figure 4.11](#).



Figure 4.11: Shallow denoising rank transpose autoencoder without overlap scaling. The output had a rank transpose with FoV= 3×3 with a stride $\Delta = 2$. Since 3 and 2 are coprime, there is a checkerboard artifact.

4.4.2 Block channel slice sorting inverse

As we can see in the example in Figure 4.10, the permutations are saved so that they can be reused later in the decoder. For stacked autoencoders, we also collect the permutations so that the decoder block can use them to permute its weights accordingly, as in Figure 3.1. Since it may be difficult to see what exactly is happening in Figure 4.10, we made a numerical example in Figure 4.12 with a ReLU activation to clarify the transposed rank filter.

In Figures 4.10 and 4.12, we save indices in a tensor \mathbf{v} that are later used to permute values in reverse order. We will clarify how to get these indices. If we consider the example in Figure 4.10, what happens is that we assign column-wise linear indexing associated with each block. Hence we augment the input⁹ with respect to the blocks:

$$\begin{bmatrix} 3 & 7 & 0 \\ 6 & 8 & 4 \\ -2 & 0 & 1 \end{bmatrix} \xrightarrow{\text{Augmentation}} \left[\begin{array}{cc|cc} 3 & 7 & 7 & 0 \\ 6 & 8 & 8 & 4 \\ \hline 6 & 8 & 8 & 4 \\ -2 & 0 & 0 & 1 \end{array} \right].$$

Then we sort the blocks and log the transformation:

⁹This is done in the GPU, so no actual augmentation is carried away. The augmentation is only done for illustrative purposes.

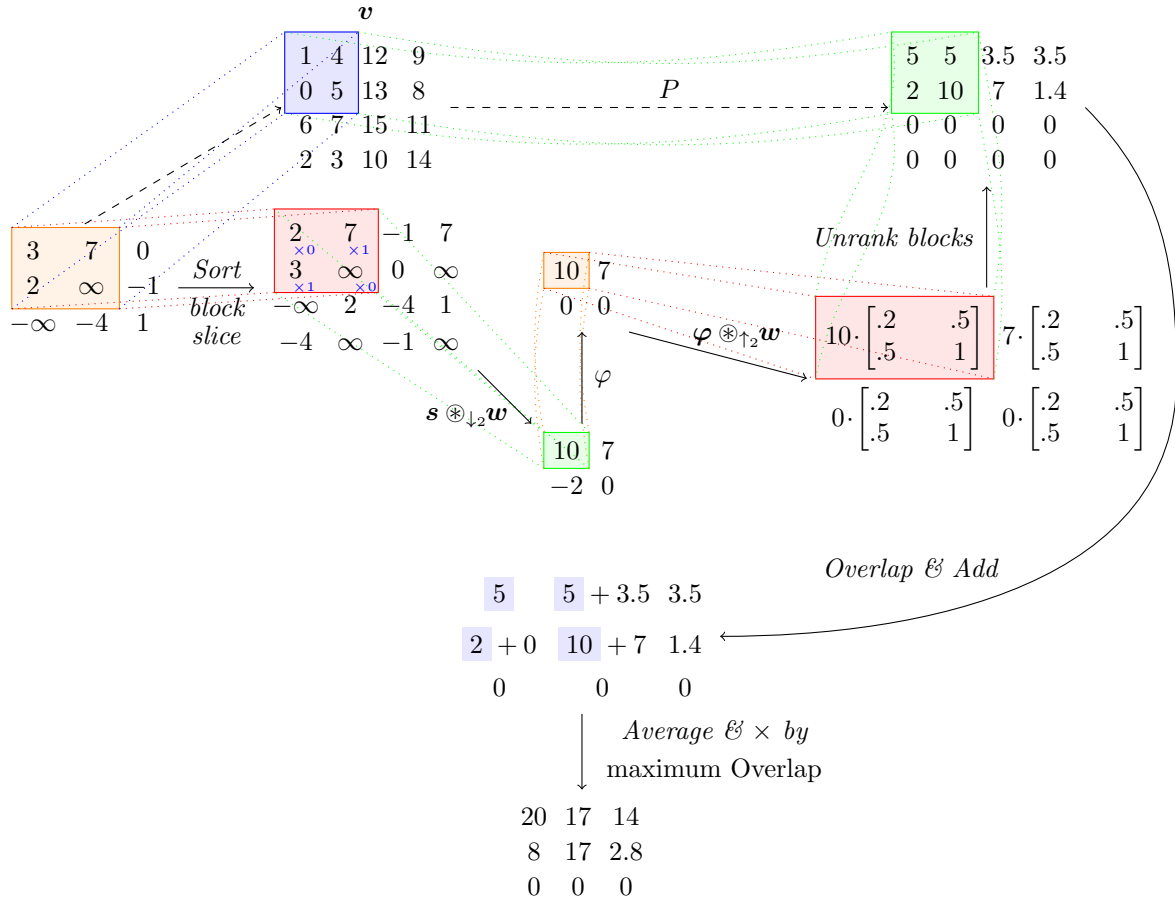
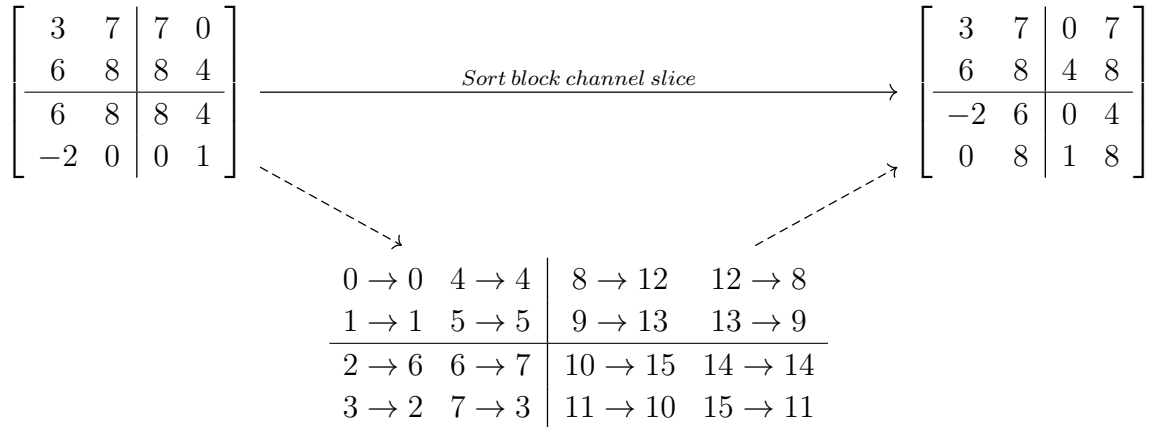


Figure 4.12: Deconvolutional rank filter numerical example.



At the bottom of the diagram lies the permutation change table where each entry $a \rightarrow b$ means “Move index a to index b .” The numbers on the right (the b ’s) correspond to our permutation indices stored in the matrix v in Figures 4.10 and 4.12. This corresponds to the sorting operation $s = P^T x$. When we execute the block unranking operation, we replace the elements from where they came from. Similarly, this corresponds to a matrix multiplication

with P since $Ps = \mathbf{x}$. Hence to sort we executed “move index a to index b ”, and to unsort we “move index b from where it came from” or $a \leftarrow b$. In our example Figure 4.10 the permutations are given by:

$$\begin{array}{cc|cc} 0 \rightarrow 0 & 4 \rightarrow 4 & 12 \rightarrow 8 & 8 \rightarrow 12 \\ 1 \rightarrow 1 & 5 \rightarrow 5 & 13 \rightarrow 9 & 9 \rightarrow 13 \\ \hline 6 \rightarrow 2 & 7 \rightarrow 6 & 15 \rightarrow 10 & 14 \rightarrow 14 \\ 2 \rightarrow 3 & 3 \rightarrow 7 & 10 \rightarrow 11 & 11 \rightarrow 15 \end{array} \equiv \begin{array}{cc|cc} 0 \rightarrow 0 & 4 \rightarrow 4 & 8 \rightarrow 12 & 12 \rightarrow 8 \\ 1 \rightarrow 1 & 5 \rightarrow 5 & 9 \rightarrow 13 & 13 \rightarrow 9 \\ \hline 2 \rightarrow 3 & 6 \rightarrow 2 & 10 \rightarrow 11 & 14 \rightarrow 14 \\ 3 \rightarrow 7 & 7 \rightarrow 6 & 11 \rightarrow 15 & 15 \rightarrow 10 \end{array}.$$

The target indices in the rhs of last equation are the inverse permutation indices π , in this case:

$$\pi = \begin{array}{cc|cc} 0 & 4 & 12 & 8 \\ 1 & 5 & 13 & 9 \\ \hline 3 & 2 & 11 & 14 \\ 7 & 6 & 15 & 10 \end{array}.$$

Up until this point, we’ve abused notation of *Sort* to mean an augmentation operation followed by a block channel-slice sorting operation. The reverse operation will also be abbreviated from now on and use the term $Sort^{-1}$ to designate a block channel-slice unsorting operation followed by an overlap & add method (where the amount of overlap equals to the rank filter stride) finishing with an adjustment of the output intensities. We compress notation by labelling the rank filtering as $rk = (Sort \rightarrow C)$ (see 4.8) and it’s transpose by $rk^T = (C^T \rightarrow Sort^{-1})$.

4.5 Transposed Rank+Linear Filters

These naturally exist and are implemented with transposed ranks and transposed linear filters. In Figure 4.13, we present how the filters can be implemented with parallel paths. This will be particularly useful to determine if rank weights feature maps can be combined additively with linear feature maps. It was designed to see if the learned feature maps can be helpful for the training process.

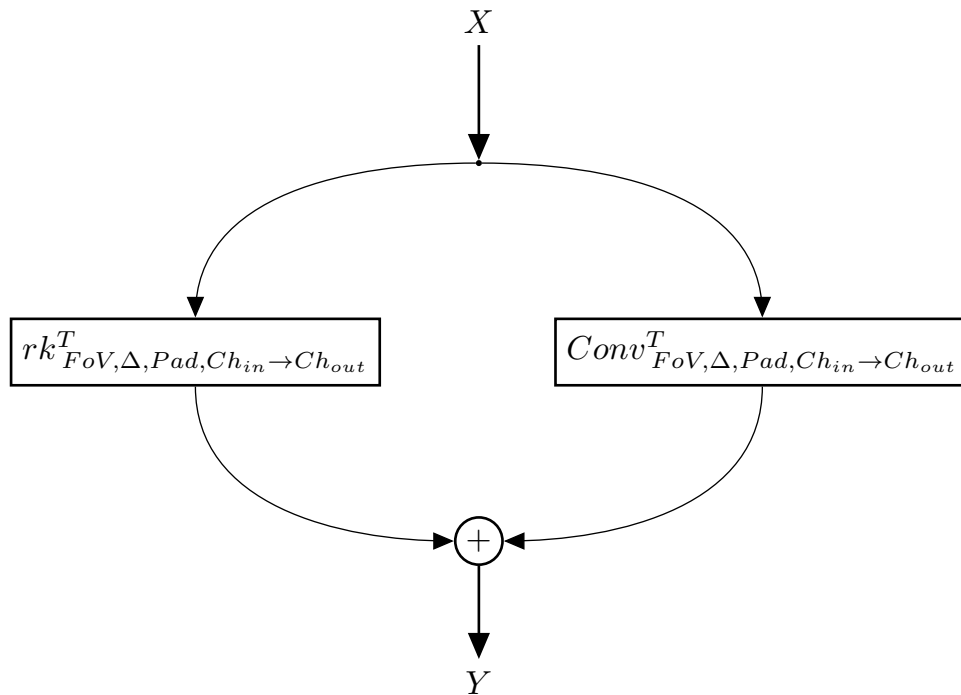


Figure 4.13: The splitting point selects channels for the transposed filters. In the junction where there is a + symbol, the rank filters elements are added to the convolutional filters additively.

Chapter 5

Histogram Filters (amplitude space)

Histogram filters are closely related to rank-filters in the sense that both are a linear combination of permuted input elements. This similarity and the fact that many algorithms are based on histograms is what motivated the inclusion of this chapter within this thesis.

This chapter is simply a graphical framework perspective on histogram-based methods in order to view them as potential architectures or inspiration for new layers for deep learning. We are only systematically using algebra in order to show the similarities with convolutional neural net architectures explicitly.

We implemented a forward pass algorithm for the amplitude space because ordered data(ranks) are just un-binned data for histograms. They are very similar in implementation and required some minor modifications of the rank filtering code. However, for the back-propagation, the equations require more alterations for the GPU implementation.

We based the histogram filter kernel on the Dirac delta function (the most uncomplicated option possible). As explained in Section 5.1, note that implementing a kernel with continuous support to allow gradients to flow through the histogram filter weights node would complicate the forward and gradient computation extensively.

The range of values of the ranks is data-dependent (meaning that they could, in theory, span the whole real line). In contrast, a histogram bin range is amplitude dependent¹ and can be fixed. Another motivation is that local histogram statistics achieve crisper and more appealing transformations [20]. Other histogram methods include histogram gradients (HOG). Thus we will show how it can be viewed as a graphical model for a potential neural net having a non-linear(histogram) filter.

5.1 HOG Pipeline Analysis

HOG features are generated from multiple operations. First, we need to compute two features i.e., gradient in x and gradients in y. Then, we generate two non-linearities features since we calculate the angle θ and the magnitude m of the gradient. Then, we use a transfer matrix

$$H_{\#Bins \times Area}^T(\theta) \mathbf{1}_{Area \times 1} = \mathbf{h}_{\#Bins \times 1}(\theta), \quad (5.1)$$

¹ Ordered data may yield highly variable minima and maxima statistics and low variance highly predictable centered values. If we bin these values, we move to the amplitude space, which is possibly less correlated than the ranks.

where $\mathbf{h}_{\#Bins \times 1}(\boldsymbol{\theta})$ is the histogram of $\boldsymbol{\theta}$. Since usually interpolation is used, the transfer matrix $H^T(\boldsymbol{\theta})$ will generally have more than one non-zero entry per column, and it obeys $H_{Area \times \#Bins}(\boldsymbol{\theta}) \mathbf{1}_{\#Bins \times 1} = \mathbf{1}_{Area \times 1}$. The fact that the matrix H has argument $\boldsymbol{\theta}$ means that the matrix is non-linear with respect to $\boldsymbol{\theta}$ and bin boundaries $a_1 < a_2 < \dots < a_{\#Bins+1}$. Note that extreme values of the bin limits can be unbounded for example, it can divide the whole real line $\mathbf{a} = [-\infty, a_2, \dots, a_{\#Bins}, \infty]$. Let us define a bin histogram made from a kernel $K(x)$ taking a real argument:

$$hist(\mathbf{a}, \boldsymbol{\theta}, x) = \sum_{j=1}^{Area} \sum_{i=1}^{\#Bins} \int_{a_i}^{a_{i+1}} K(t - \theta_j) dt \cdot \delta[a_i < x < a_{i+1}]. \quad (5.2)$$

If we define the indicator function vector to be $\boldsymbol{\delta}_{\#Bins \times 1}(x) = [\delta[a_i < x < a_{i+1}]]_{i \in [1: \#Bins]}$, then if we instead of binning into the real line, we bin into a vector $\mathbf{h}_{\#Bins \times 1}(\boldsymbol{\theta})$ then

$$\mathbf{h}_{\#Bins \times 1}(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\delta}(x)} hist(\mathbf{a}, \boldsymbol{\theta}, x) = \sum_{j=1}^{Area} \int_{a_i}^{a_{i+1}} K(t - \theta_j) dt; i \in [1 : \#Bins]. \quad (5.3)$$

By substituting in eq.5.1 we get :

$$H_{\#Bins \times Area}^T(\boldsymbol{\theta}) = \left[\int_{a_i}^{a_{i+1}} K(t - \theta_j) dt \right]_{i,j} \quad \text{for } i \in [1, \#Bins] \text{ and } j \in [1, Area]. \quad (5.4)$$

Then we apply the transformation matrix to \mathbf{m} : $\mathbf{z} = H_{\#Bins \times Area}^T(\boldsymbol{\theta}) \mathbf{m}_{Area \times 1}$. The final HOG feature is thus $\mathbf{y} = norm(\mathbf{z})$, where norm could be any normalization technique (for example using ℓ_p , L2-Hys or L1-sqrt).

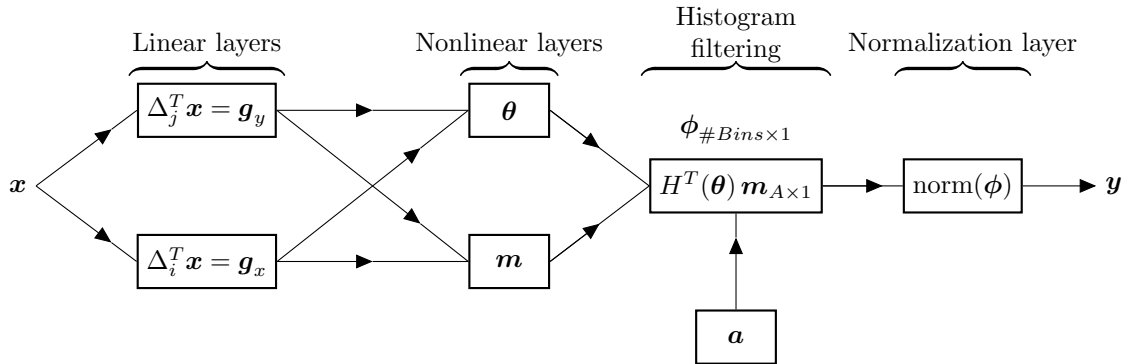


Figure 5.1: HOG pipeline analysis. The non-linearity after the non-linear filter is a vector function as in Capsule net [33]. It is not an elementwise function such as the ReLU activation.

Doesn't this pipeline look familiar? We have a linear layer followed by a non-linear function, then a non-linear filter, after a normalization. If we lump the two non-linear operations, we get the usual block commonly used in deep learning a linearity \rightarrow activation \rightarrow normalization. Additionally, gradients can back-propagate through the pipeline except

possibly through $H_{i,j}(\boldsymbol{\theta})$. To see when it can propagate through, let us compute the gradient of a single element eq.5.4 with respect to θ_k :

$$\frac{\partial}{\partial \theta_k} H_{(j,i)} = -\delta[k = j] \int_{a_i}^{a_{i+1}} K'(t - \theta_j) dt = \delta[k = j] (K(a_i - \theta_j) - K(a_{i+1} - \theta_j)). \quad (5.5)$$

If the Kernel K is a Dirac delta function $\delta(x)$, then

$$\frac{\partial}{\partial \theta_j} H_{(j,i)} = \delta(a_i - \theta_j) - \delta(a_{i+1} - \theta_j) = 0; a_i \neq \theta_j \forall (i, j). \quad (5.6)$$

Hence, the gradients will not propagate through H in the $\theta \neq a$ case, but they will explode in the opposite case. However, that may not be a problem if the gradient gets multiplied (during back-propagation) by a vanishing function at the singularities. Usually, HOG features are computed using a boxcar kernel with equidistant bins; it may also be possible to use a Gaussian kernel. These examples do yield a derivative by using eq.5.4. To let eq.5.4 produce positive values on a range, and the Kernel can't have discrete support on the real line. In other words, it can't contain any Dirac delta functions. It is always possible to back-propagate through a HOG layer if it's inside a neural net. Still, more research should be made on this subject to show what would be the impact of having vanishing gradients for H .

Since histogram filter weights depend on bin limits \mathbf{a} , it may also be possible to back-propagate through and update the bin limits. If we compute the gradients of the histogram filter with a_k :

$$\frac{\partial}{\partial a_k} H_{(j,i)} = \frac{\partial}{\partial a_k} \int_{a_i}^{a_{i+1}} K(t - \theta_j) dt = \delta[k = i + 1] K(a_{i+1} - \theta_j) - \delta[k = i] K(a_i - \theta_j). \quad (5.7)$$

The above equation shows that to update the bin limits, we need the kernel to be bounded and have continuous support. Note that the kernel functions may be different, and the difference may even depend on the limits \mathbf{a} in order, for example, the sum adds up to 1 between the bin range when we overlap and add the kernels. In this case, more complicated derivatives will incur. The kernels may also depend on more parameters such as bandwidth, for example.

5.2 Histogram Equalization Pipeline Analysis

In this special case, the bins are centered on the set of pixel values $\{0, \dots, B - 1\}$. Then, we have assigned the equalization of a histogram to their corresponding pixel. Since we already know how to compute a discrete histogram by using eq.5.3, we can easily compute the cdf of $\mathbf{x}_{Area \times 1}$:

$$\mathbf{cdf}(\mathbf{x}) = L_{B \times B} \mathbf{h}_{B \times 1}(\mathbf{x}), \quad (5.8)$$

where L is a left triangular $B \times B$ matrix where all of the lower diagonal terms (including the diagonal) are 1. We stretch the cdf values in $[0, B - 1]$ by using:

$$\mathbf{q}(\mathbf{x}) = (B - 1) \frac{\mathbf{cdf} - \mathbf{cdf}[1]}{Area - \mathbf{cdf}[1]}, \quad (5.9)$$

where $cdf[1] = \min \mathbf{cdf}$. the output is:

$$\mathbf{y} = H\mathbf{q}(\mathbf{x}). \quad (5.10)$$

Note that usually, the equalized histogram \mathbf{q} are rounded to integers, and so gradients can't be computed. We need to ignore the rounding operation to permit non-zero gradients. For histogram equalization, the matrix H is made from the dirac Kernel; hence it should have only a single "1" per row. This choice for K is inconvenient for gradient propagation. The pipeline is equivalent to a non-linear filter (to compute histogram) → a normalization function → the transposed non-linear filter of the first layer (permute back values to original place).

When seen as a deep learning architecture, we see that the first layer is a coder structure followed by code processing subsequently passed to a decoder. In order to fit the equalizer

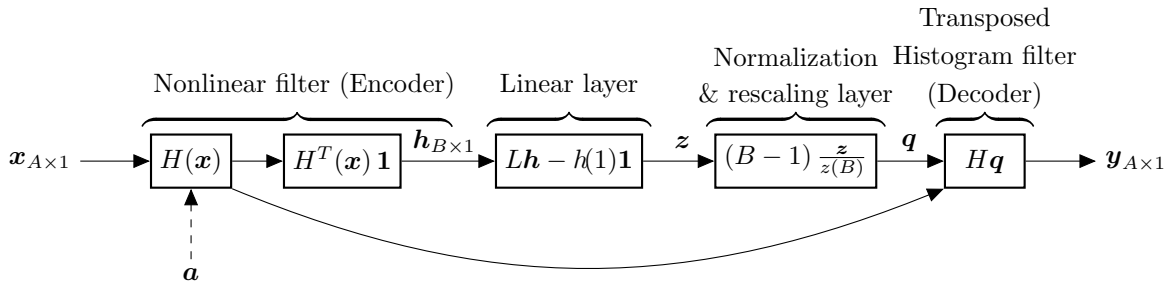


Figure 5.2: Histogram Equalization pipeline analysis. The non-linear filter is the encoder, but the transformation is made on the $\mathbf{1}$ rather than the input \mathbf{x} such as rank filters. Following the encoder layer, there is a linear transformation to compute the cdf followed by a scaled normalization. The normalised (equalized) histogram \mathbf{q} is then passed to the decoder which is the transposed histogram filter linked to the encoder.

into a neural net, we would have to use a Kernel with a continuous support on the real line.

5.3 Amplitude bank filters and link with ReLU activation

Implementing amplitude banks rectifiers needs us to define a histogram of say $\#Bins = B$ bins bounded by \mathbf{a} . We can set the output amplitude bank output to be:

$$Y_{:,:(1:Ch_{in})+(b-1)Ch_{in},:} = X_{:,,:,:,} \circ \delta [a_b < X_{:,,:,:,} < a_{b+1}]; b \in [1, B] \quad (5.11)$$

If we used the simplified vector notation as in equation 5.1, we get:

$$Y = H_{Area \times B}(\mathbf{x}) \circ (\mathbf{x}_{Area \times 1} \mathbf{1}_{1 \times B}^T) \quad (5.12)$$

This is not exactly a histogram, because, we would need to use the term $\delta [a_b < X_{:,,:,:,} < a_{b+1}]$ and sum-reduce in a dimension in order to get a count frequency as in eq.5.1. However if

we do that, we will inevitably encounter problems since $\delta [a_b < X_{:, :, :, :} < a_{b+1}]$ has vanishing gradients and infinite gradients at some points². To make it differentiable, we would need to smooth it with a kernel K that is not a dirac function. However, we will not use this approach. Instead, we use the differentiable version in Eq. 5.11. If backprop gradients are related to the derivative of $Y_{:, :, (1:Ch_{in})+(b-1)Ch_{in}, :}$ by X :

$$\frac{\partial \mathcal{L}}{\partial X} = \sum_{b=1}^B \frac{\partial \mathcal{L}}{\partial Y_{:, :, (1:Ch_{in})+(b-1)Ch_{in}, :}} \circ \delta [a_b < X < a_{b+1}] \quad (5.13)$$

This function can be easily implemented by using ReLUs and unit step functions. Moreover, we want to add that this has been explored when the bin boundaries are $\mathbf{a} = [y, 0, \infty]$ for the ReLU case and $\mathbf{a} = [-\infty, 0, \infty]$ in the equivalent³ Concatenated ReLU (CReLU) [37]. Note that the authors in [37] used the notation $y = [[x]_+, [-x]_+]$, which yields all positive values in contrast with our version $y = [[x]_+, [x]_-]$. This sign difference doesn't matter since the weights can learn the sign, and they usually have 0 mean. In the CReLU article [37], the authors report that it is advantageous to use CReLU for multiple reasons. Among the reasons is that there is no information loss compared to ReLU because the bins divide the whole real line in the CReLU case.

In [37] (and also as in the ReLU case), a dot product follows the bank. Thus, the authors effectively have implemented the following equation⁴:

$$\mathbf{y} = \mathbf{1}^T \left(H_{Area \times B}(\mathbf{x}) \circ (\mathbf{x}_{Area \times 1} \mathbf{1}_{1 \times B}^T) \circ W_{Area \times B} \right) \mathbf{1}. \quad (5.14)$$

That being said, the weights are modulated by Histogram filters and thus the updates on W depend on the histogram or distribution of \mathbf{x} algebraically since:

$$\nabla_W \mathbf{y} = H_{Area \times B}(\mathbf{x}) \circ (\mathbf{x}_{Area \times 1} \mathbf{1}_{1 \times B}^T). \quad (5.15)$$

ReLU is linked to one of simplest⁵ histogram filters that has only a single bin.

²The reason being is because H generated from the Kronecker delta function $\delta [a_b < X_{:, :, :, :} < a_{b+1}]$ is modelled from dirac delta Kernels.

³The methods are equivalent if the output of the CReLU is followed by a dot product where the weights are permitted to be negative.

⁴In convolutional neural nets, the Area would correspond to the patch volume or the area times the input channels. In the ReLU case $B = 1$, and in the CReLU case $B = 2$.

⁵“Simplest” because the histogram transformation is based on delta functions, and so has vanishing gradients (except at bin locations).

Chapter 6

Experiments & Methodology

6.1 Introduction

The primitives for the novel functions are all implemented using CUDA with MATLAB's mexcuda compiler. We do not use MATLAB's inbuilt neural net library but have implemented it using cuDNN. We've built a library from scratch to precisely know how layers are implemented and control all aspects that we wish to investigate, particularly for the training algorithm.

6.1.1 Cyclic Learning Rate Algorithm

The hyper-parameters for Stochastic Gradient Descent (SGD) are no longer considered black-magic guessing. In the articles presenting the Cyclic Learning Rate (CLR) [41], [40] algorithms, the authors find a range of optimal learning rates that permit to obtain state-of-the-art performance using only SGD with costless periodic updates of hyper-parameters (mom and μ). We use this technique to train our nets unless otherwise stated. The Learning rate $\mu \in [\mu_{opt}; \mu_{max}]$, as seen in Figure 6.1. Additionally, we use a cyclical momentum range $mom \in [0.85; 0.95]$ with an opposite phase to that of the learning rate's. When μ is a minimum, mom is 0.95, and when μ is minimal mom=0.85. Only the learning rate range (and not the momentum) is determined via a preliminary test where the LR is incremented, during training, from 0 to a large number. We plot the loss function with respect to the learning rate. The learning rate is an approximate range where the loss converged quickest as shown in Figure 6.2.

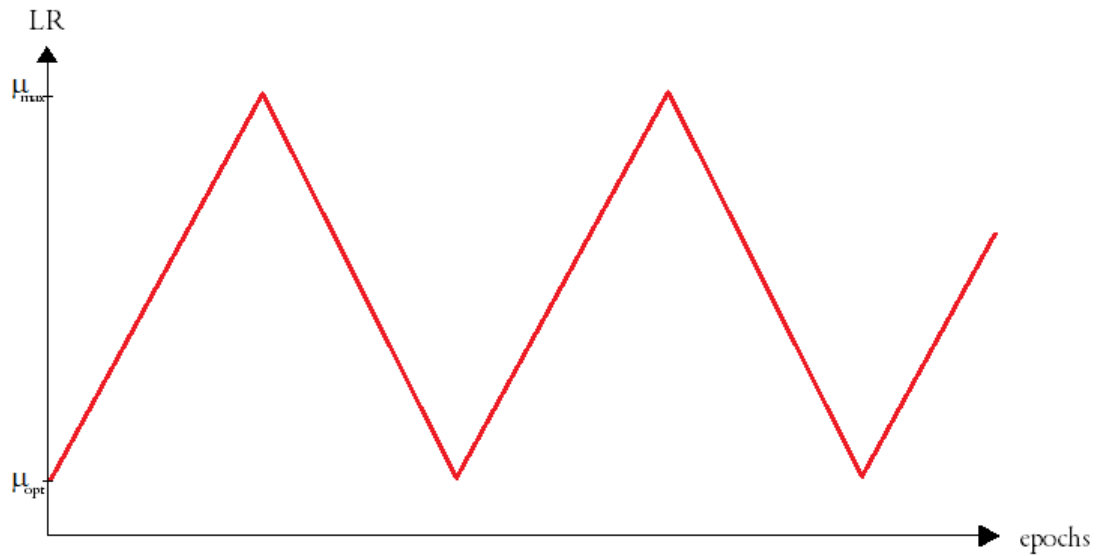


Figure 6.1: Cyclic Learning Rate technique. The learning rate varies with respect to the number of epochs. In this case, the learning rate follows a triangular wave. The lowest LR is μ_{opt} , and the highest is μ_{max} .

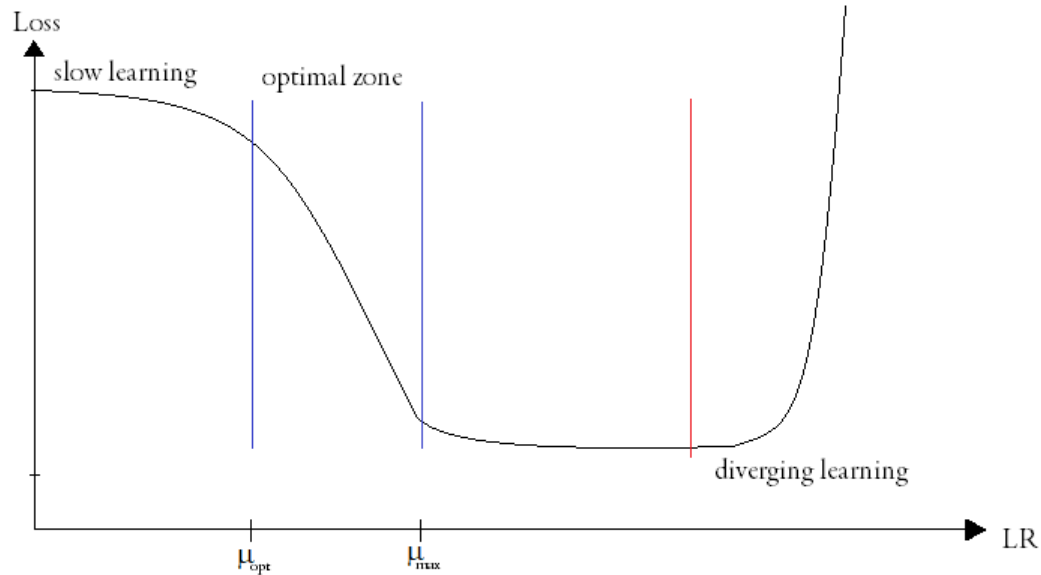


Figure 6.2: Cyclic Learning Rate initialization. The learning rate is initially set to a ramp that grows linearly from 0. It is stopped when the Loss function diverges. The optimal range to set the Learning rate amplitudes are given by the limits in blue. The first zone on the left has very slow learning.

6.1.2 Datasets & Other Experiment Setup

Depending on the experiment, we use various datasets for training, for example, MNIST, SVHN, and CIFAR-10. . The other methodology specifics, for example, the mini-batch size, the noise type, the loss function, and the specifics of the architecture, please refer to the experiment themselves as they are highly variable i.e., go to section 6.2 and Chapter 6. The architectures are highly variable for an excellent reason. Since the layers tested are new, we need to find the architecture that works with it. We use a hands-on approach for the architecture design because we need to test as many different types of architectures possible that would optimally work with the proposed rank filters. Our baselines architectures are the ones where we replace rank layers with the linear convolution layer unless otherwise stated.

We will refrain in bloating the experiments with unnecessary graphics unless otherwise stated for training or generalization purposes. It is better to make comparative tables of multiple experiments than a confusing profusion of graphics of lesser experiments. We opt for many experiments so that we can see patterns in architectures that are helpful to fulfill the main objective, i.e., to understand how rank filters behave.

6.1.3 Batch Normalization Initialization

We initialize our Batch normalization layer scales sampled from a uniform distribution $\gamma \sim \mathcal{U}$ and the biases are set to 0. We found that initializing the bias to small random numbers had a small¹ positive effect for the linear convolution layer, however, it was very detrimental for rank-based autoencoders especially in decoders². We tried different initializations i.e.:

1. Small random bias with $\gamma = \mathbf{1}$
2. Small random bias with $\gamma \sim \mathcal{U}(0, 1)$
3. Small random bias with $\gamma = \gamma_1/2 + \gamma_2/2; (\gamma_1, \gamma_2) \sim \mathcal{U}(0, 1)$
4. Bias $\mathbf{b} = \mathbf{0}$ with $\gamma = \mathcal{U}(0, 1)$
5. Small random bias with $\gamma \sim \mathcal{U}(0, 1)$

The best option was the fourth. The initialization of BN had a positive effect on the values for the CLR test. The LR range wasn't the same for different initializations of the scale.

In decoders, we place BN explicitly after the ReLU layers. Since in preliminary experiments, BN best location depends on the problem's nature, we thus somewhat disapprove of observations³ discussed in [3]. We found consistently that BN is best located before the ReLU layer in a coder, and after the ReLU layer in the decoder. We could exchange the

¹This positive effect occurred when we used blocks that ended in Batch normalization. It exhibited a better score at the beginning. Still, when the net converged no difference could be noted from a net that used non-zero initial bias (which is reasonable because the bias is allowed to grow).

²We found that when we removed the biases altogether, we had better results than when we didn't. This effect didn't depend on the number of decoder blocks. The results only depended if BN was the last layer within a decoder block having a transposed rank filter.

³If Batch normalization biases are initialized to small random values, then we can get problems especially in decoders that use transpose rank filters.

activation and normalization only(as recommended in [3]) if the bias was set to be 0 in the normalization and filtering layer.

6.1.4 Filters Initialization

We use an initialization to set $\sigma = \sqrt{\frac{1}{d_{in}}}$ when using the Tanh activation with the BN initialization using $\gamma \sim \mathcal{U}(0, 1)$ for transposed filters. An exception to this rule applies when a skipped connection is present in which case we use the normal initialization of He (Kaiming)[18] with $\sigma = \sqrt{\frac{2}{d_{in}}}$. Additionally, note that the weights are chosen to be orthogonal, hence we generate them using the single value decomposition from a random matrix $R_{d_{in} \times d_{out}}$. The orthogonal initialization of weights consistently yields better training speeds, much better than when sample from a gaussian distribution.

The input receptive field size is named d_{in} and the output receptive field is d_{out} (named fan_{out} in [25]).

In the case of convolutions we use always He (Kaiming)[18] initialization with zero bias.

6.1.5 Notation

6.1.5.1 Convolutional layers

For the convolutional layer we use the notation of $C_{FoV, \downarrow \Delta, Pad, C_{in} \rightarrow C_{out}}$ to designate a convolution with square filters than span into the whole input channel depth, i.e. $W \in \mathbb{R}_{FoV \times FoV \times C_{in} \times C_{out}}$. Δ is the stride the filter takes (or the downsampling rate) and Pad is the zero-padding added to one side of the image boarder. Moreover, we also have the general case $C_{w_{win} \times h_{win}, \downarrow (\Delta_W, \Delta_H), (PadW, PadH), C_{in} \rightarrow C_{out}}$ for general window size filters strides and padding. We will not distinguish between reflective and zero padding in this case. Zero-padding is used in linear convolutions unless it is an integral component of a rank filter, where reflective-padding is used. We will explicitly mention whenever the linear convolution uses reflective padding.

6.1.5.2 Transposed Convolutional layers

The convolutional transpose is annotated in a similar manner $C_{FoV, \uparrow \Delta, UnPad, C_{out} \rightarrow C_{in}}$, here the signal is upsampled by a factor of Δ , we crop the edges of an image by a length of $Unpad$ pixels.

6.1.5.3 Rank filter layers

For the convolutional layer we use the notation of $rk_{FoV, \downarrow \Delta, Pad, C_{in} \rightarrow C_{out}}$ and the general case $rk_{w_{win} \times h_{win}, \downarrow (\Delta_W, \Delta_H), (PadW, PadH), C_{in} \rightarrow C_{out}}$ to designate a rank convolutions. Note that in the case of rank filtering, the padding is done by padding the input with reflections of itself for example, we have $x_{1 \times 5} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \end{pmatrix}$, then a left and right padding will be $x_{1 \times (5+4)}^{pad} = \begin{pmatrix} 2 & 1 & 1 & 2 & 3 & 4 & 5 & 5 & 4 \end{pmatrix}$. The reason why we chose reflective padding is that

for rank filters it makes more sense to compare with local order statistics around edges instead of the mean. Empirically this decision showed the best results.

6.1.5.4 Transposed Convolutional layers

The rank transpose is annotated with $rk_{Fov,\uparrow\Delta,Unpad,Cout\rightarrow Cin}^T$, here the signal is upsampled by a factor of Δ , we crop the edges of an image by a length of $Unpad$ pixels. It is important to note that this layer needs another input, i.e. the coder’s permutation indices.

6.1.5.5 Joint Linear-Rank Filters

The joint Linear-Rank filter is annotated with $J_{Fov,\uparrow\Delta,Unpad,Cout\rightarrow Cin}$.

6.2 Denoising Autoencoders Experiments

6.2.1 Experiment 1: Using rank filters, find which coder-decoder pair is best to reduce image corruption with a wide gaussian noise for shallow autoencoder.

We will start by testing the properties of rank filters in a pure single-layer coder and single-layer decoder.

6.2.1.1 Methodology

The database used is SVHN. We include the case where the coder has no activations because the rank filter is non-linear itself. We will study the interactions when BN is placed after activation in the coder. We will also include the interactions of ReLU with different activation functions to determine which one yields the best performance. We test the architectures under the ℓ_2 pixel-wise error and compare them with the all-linear counterpart. We conduct the tests where images are to be denoised, given that the pixels of the input have a probability of having each channel value being corrupted by 50% with Gaussian noise $\mathcal{N}(\mu = 0, \sigma = 2)$. We should only see uncorrupted pixels with a chance of $1 - p^3 = 1 - \frac{1}{2^3} = 87.5\%$. We chose this type of noise because it is very light. It is ideal for a small network, and we also want to generate images to show qualitative results.

The mini-batch size is set to 512. This large mini-batch size was chosen because the CLR method works best for large Batch sizes. The total number of batches used for training the baseline linear net is 64 epochs. The number of epochs for all other nets is 32⁴. We use a large mini-batch size because in [41] it is recommended to use the largest possible. We use a cycle period of 8 epochs.

We test the Leaky ReLU activation to see what could be the amount of linearity required to train the network. The LReLU net is tested in the fourth row of Table 6.1. The LReLU “a” parameter was left to grow large. The only constraint used was $|a| < 1$. We just used Leaky-ReLU’s parameter as a pressure gauge to measure optimal linearity. At the end of the

⁴ Usually they converged at 16 epochs, as opposed to deeper nets.

training, “a” was larger for the coder than the decoder (though that might have changed later on). This suggests that a more linear activation may be needed in the coder compared to the second. Either way, the Leaky ReLU was worse than the ReLU method. Because of this, we explored the possibility of using Tanh as a further experiment in the table. The fact that there was an asymmetry in a ’s might suggest that we could get even better results if we used different activations in the coder and the decoder. This activation choice difference within an autoencoder will be investigated further in later tests in experiment 3. Additionally, while at it, we added more models to test that used alternate activations.

All coder and decoder blocks tested have a large receptive field of $4 \times 4 \times C_{in}$, and no downsampling is performed. We use a relatively large(32) output channel for the coder.

Our initial plan was to use only the ReLU activation. However, we discovered that when we initialized the bias of BN, the ReLU nets were significantly affected except for all linear ones. The Tanh activation served as an alternative that worked better than ReLU when BN biases were initialized, and because LReLU indicated some linearity (which Tanh approximately has when it’s argument is small.).

We use a simple labels to designate the 2 layers after the filtering. The two last layers can be either “B” for BN, “R” for ReLU, “L” for Leaky-ReLU and “T” for Tanh. For example the coder block $C_{4,\downarrow 1,3,3 \rightarrow 32} \rightarrow BN \rightarrow ReLU$ is equivalent to $C_{4,\downarrow 1,3,3 \rightarrow 32} \rightarrow BR$

6.2.1.2 Experiment results & Discussion

In Table 6.1, we trained 13 different denoising autoencoders. The best one was the one has the following architecture:

$$rk_{4,\downarrow 1,3,3 \rightarrow 32} \rightarrow BN \rightarrow Tanh \rightarrow C_{4,\uparrow 1,3,32 \rightarrow 3}^T \rightarrow ReLU \rightarrow BN.$$

So indeed alternating between activations may be beneficial. We suspect that Tanh may have helped in attenuating high corruption values (unrealistically unbounded in this case). Therefore, in this particular scenario, Tanh may have had an advantage over ReLU. This advantage is why, in subsequent experiments, we will use possible values for an image. The three best nets were the ones that finished with a convolution decoder. This observation may be because we used a large FoV filter with a high overlap.

To compare the best result with it’s all-linear counterpart, we convert the shallow autoencoder rank layer of $rk_{4,\downarrow 1,3,3 \rightarrow 32} \rightarrow BT \rightarrow C_{4,\uparrow 1,3,32 \rightarrow 3}^T \rightarrow RB$ to a linear one to directly compare the methods. The new one is $C_{4,\downarrow 1,3,3 \rightarrow 32} \rightarrow BT \rightarrow C_{4,\uparrow 1,3,32 \rightarrow 3}^T \rightarrow RB$ and the comparison is in Table 6.2. It suggests that the rank layer is better than the linear one in the experiment setup.

Autoencoder and it's corresponding coder-decoder pair with learning rate range of CLR and results (ℓ_2).

Id	Coder	Decoder	μ_{opt}	μ_{max}	ℓ_2	Fig.
1	$C_{4,\downarrow 1,3,3 \rightarrow 32} \rightarrow BR$	$C_{4,\uparrow 1,3,32 \rightarrow 3}^T \rightarrow RB$	$1 \cdot 10^{-7}$	$1 \cdot 10^{-4}$.0738	6.3c
2	$rk_{4,\downarrow 1,3,3 \rightarrow 32} \rightarrow BR$	$rk_{4,\uparrow 1,3,32 \rightarrow 3}^T \rightarrow RB$	$3 \cdot 10^{-7}$	$1 \cdot 10^{-4}$.0420	6.3d
3	$rk_{4,\downarrow 1,3,3 \rightarrow 32} \rightarrow RB$	$rk_{4,\uparrow 1,3,32 \rightarrow 3}^T \rightarrow RB$	$6 \cdot 10^{-7}$	$8 \cdot 10^{-6}$.0502	6.3e
4	$rk_{4,\downarrow 1,3,3 \rightarrow 32} \rightarrow BL$	$rk_{4,\uparrow 1,3,32 \rightarrow 3}^T \rightarrow LB$	$3 \cdot 10^{-7}$	10^{-5}	.0496	6.3f
5	$rk_{4,\downarrow 1,3,3 \rightarrow 32} \rightarrow B$	$rk_{4,\uparrow 1,3,32 \rightarrow 3}^T \rightarrow B$	$5 \cdot 10^{-8}$	10^{-6}	.0774	6.3g
6	$rk_{4,\downarrow 1,3,3 \rightarrow 32} \rightarrow BT$	$rk_{4,\uparrow 1,3,32 \rightarrow 3}^T \rightarrow TB$	$2 \cdot 10^{-8}$	10^{-6}	.0463	6.3h
7	$rk_{4,\downarrow 1,3,3 \rightarrow 32} \rightarrow BT$	$rk_{4,\uparrow 1,3,32 \rightarrow 3}^T \rightarrow RB$	$2 \cdot 10^{-8}$	10^{-6}	.0422	6.3i
8	$rk_{4,\downarrow 1,3,3 \rightarrow 32} \rightarrow BT$	$C_{4,\uparrow 1,3,32 \rightarrow 3}^T \rightarrow RB$	$2 \cdot 10^{-8}$	10^{-6}	.0310	6.3j
9	$rk_{4,\downarrow 1,3,3 \rightarrow 32} \rightarrow BT$	$C_{4,\uparrow 1,3,32 \rightarrow 3}^T \rightarrow TB$	$2 \cdot 10^{-8}$	10^{-6}	.0342	6.3k
10	$rk_{4,\downarrow 1,3,3 \rightarrow 32} \rightarrow BT$	$C_{4,\uparrow 1,3,32 \rightarrow 3}^T$	$8 \cdot 10^{-6}$	$1.5 \cdot 10^{-5}$.0373	6.3l
11	$rk_{4,\downarrow 1,3,3 \rightarrow 32} \rightarrow BT$	$rk_{4,\uparrow 1,3,32 \rightarrow 3}^T$	$4 \cdot 10^{-8}$	$1.2 \cdot 10^{-7}$.0657	6.3m
12	$rk_{4,\downarrow 1,3,3 \rightarrow 32} \rightarrow BR$	$C_{4,\uparrow 1,3,32 \rightarrow 3}^T$	10^{-6}	10^{-5}	.0393	6.3n
13	$rk_{4,\downarrow 1,3,3 \rightarrow 32} \rightarrow BR$	$rk_{4,\uparrow 1,3,32 \rightarrow 3}^T$	$2 \cdot 10^{-7}$	$1.2 \cdot 10^{-6}$.0577	6.3o
14	$rk_{4,\downarrow 1,3,3 \rightarrow 32}$	$rk_{4,\uparrow 1,3,32 \rightarrow 3}^T$	$3 \cdot 10^{-7}$	$2 \cdot 10^{-6}$.0768	6.3p

Table 6.1: Experiment 1 results

Best coder-decoder pair compared to it's linear counterpart and learning rate range of CLR and ℓ_2 costs.

Id	Coder	Decoder	μ_{opt}	μ_{max}	ℓ_2	Fig.
8	$rk_{4,\downarrow 1,3,3 \rightarrow 32} \rightarrow BT$	$C_{4,\uparrow 1,3,32 \rightarrow 3}^T \rightarrow RB$	$2 \cdot 10^{-8}$	10^{-6}	.0310	6.3j
-	$C_{4,\downarrow 1,3,3 \rightarrow 32} \rightarrow BT$	$C_{4,\uparrow 1,3,32 \rightarrow 3}^T \rightarrow RB$	$2 \cdot 10^{-6}$	10^{-5}	.0765	

Table 6.2: Comparing best result of experiment 1 with its linear counterpart.

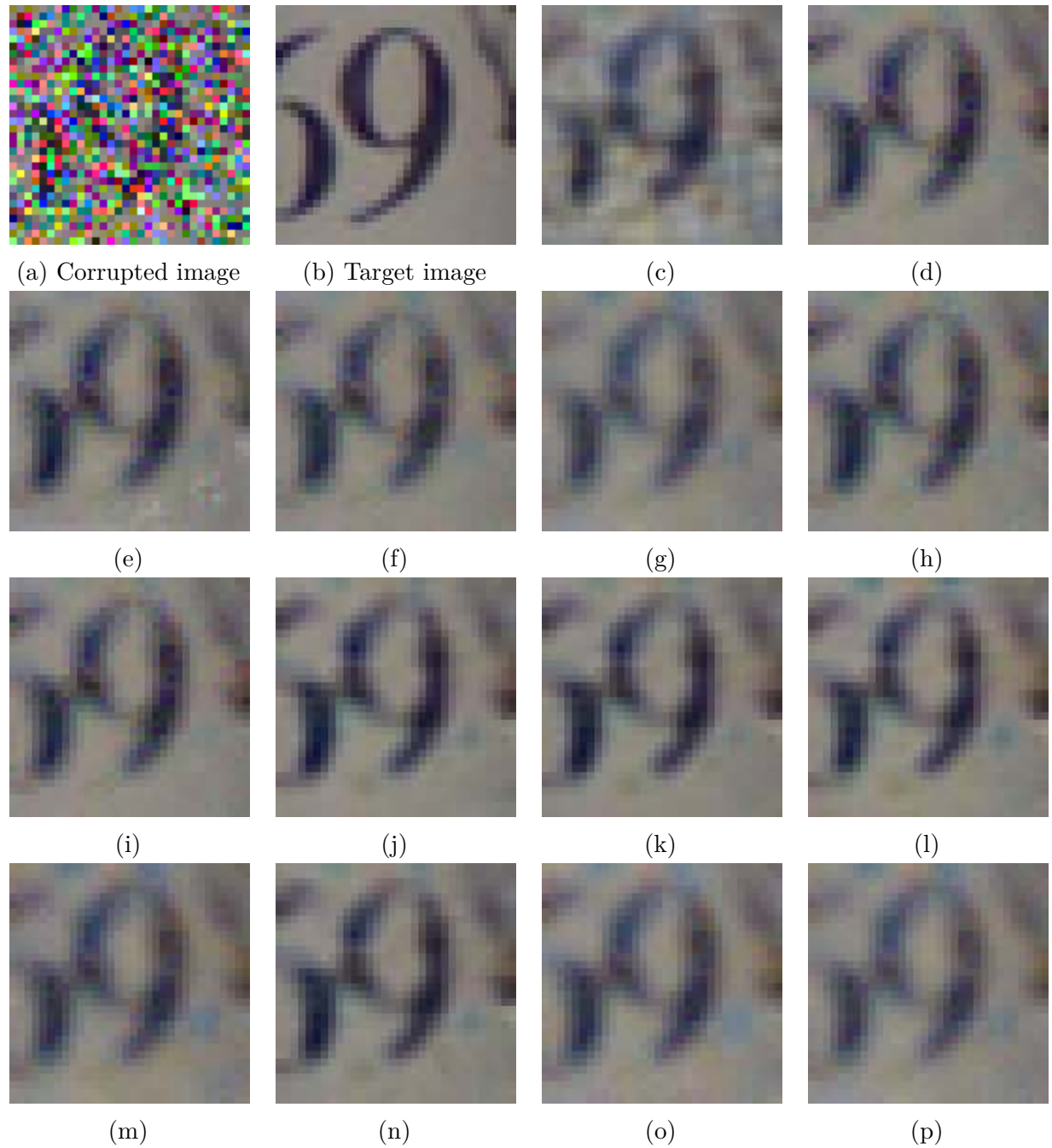


Figure 6.3: Denoising net qualitative results for experiments 1 in Table 6.1.

6.2.2 Experiment 2: Using rank filters, find which coder-decoder pair is best to reduce gaussian mixture corruption for an autoencoder of total depth of 4.

6.2.2.1 Methodology

In experiment 6.2.1, we investigated the shallowest possible autoencoder. Since it has a coder and a decoder, we say it has a total depth of 2. Since in this experiment, we look at the next largest autoencoder, it necessarily implies that we will use two coders and two decoders summing to a total of 4 layers. We will see what the general trend that happens when we increase the depth of autoencoders is.

6.2.2.2 Methodology

The CIFAR-10 database is used in this case. At each epoch, we estimate the test set loss with only a single batch of 256 randomly picked from the test set. The images pixel channels have a 1/3 probability of being corrupted by a Gaussian mixture noise that has three modes. One of the means of the gaussian spikes is located at 0. The other two are located at two-thirds of the minimum and maximum possible pixel value. The standard deviation of the peaks is 1/20, and the three noise modes occur at the same rate. This means that chance of seeing a pixel with all RGB channels uncorrupted is of $(1 - \frac{1}{3})^3 \approx 29.6\%$. We chose this noise because it is a moderate type of noise suitable for depth four networks. Other types of noise yield the same conclusions; however, for display purposes, we chose a moderate one. Very aggressive types of noise didn't produce "nice" pictures. We chose a mild kind of noise simply because the reader can see and judge by himself the preservation of the fine-structure capacities for tested architectures in Figures 6.4 and 6.5. Note that the noise saturates at possible values corresponding to the pixel amplitude range of $[0; 255]$, therefore the noise is clipped at the minimum and maximal values for a normalized image. A total of 64 epochs are used for training.

To emulate a skip connection with an average pooling all the weights are added 1/FoV if the weight channel input index equals its output channel index. That initialization will quickly be forgotten once the net is converged. This initialization is applied only to all coders including linear and rank types. In the following subsections, I will list ten architectures used for image reconstruction under the ℓ_1 and ℓ_2 loss. Then, you will see the CLR search test results of the experiment in Table 6.3 for both losses.

6.2.2.3 The linear convolution autoencoder architecture.

The coder architecture is formed of two convolutional blocks as follows:

$$(C_{4,\downarrow 2,3,3 \rightarrow 16} \rightarrow BN \rightarrow ReLU) \rightarrow (C_{2,\downarrow 1,1,16 \rightarrow 16} \rightarrow BN \rightarrow ReLU).$$

The decoder is formed of two blocks:

$$(C_{2,\uparrow 1,1,16 \rightarrow 16}^T \rightarrow ReLU \rightarrow BN) \rightarrow (C_{4,\uparrow 2,3,16 \rightarrow 3}^T \rightarrow ReLU \rightarrow BN).$$

6.2.2.4 The rank autoencoder with ReLU architecture.

The coder architecture is formed of blocks as follows:

$$(rk_{4,\downarrow 2,3,3 \rightarrow 16} \rightarrow BN \rightarrow ReLU) \rightarrow (rk_{2,\downarrow 1,1,16 \rightarrow 16} \rightarrow BN \rightarrow ReLU).$$

The decoder is formed of two blocks:

$$(rk_{2,\uparrow 1,1,16 \rightarrow 16} \rightarrow ReLU \rightarrow BN) \rightarrow (rk_{4,\uparrow 2,3,16 \rightarrow 3} \rightarrow ReLU \rightarrow BN).$$

6.2.2.5 The rank autoencoder without ReLU (BN only).

The coder architecture is formed of blocks as follows:

$$(rk_{4,\downarrow 2,3,3 \rightarrow 16} \rightarrow BN) \rightarrow (rk_{2,\downarrow 1,1,16 \rightarrow 16} \rightarrow BN).$$

The decoder is formed of two blocks:

$$(rk_{2,\uparrow 1,1,16 \rightarrow 16} \rightarrow BN) \rightarrow (rk_{4,\uparrow 2,3,16 \rightarrow 3} \rightarrow BN).$$

6.2.2.6 The rank autoencoder with Tanh activation.

The coder architecture is formed with the following blocks:

$$(rk_{4,\downarrow 2,3,3 \rightarrow 16} \rightarrow BN \rightarrow Tanh) \rightarrow (rk_{2,\downarrow 1,1,16 \rightarrow 16} \rightarrow BN \rightarrow Tanh).$$

The decoder is formed of two blocks:

$$(rk_{2,\uparrow 1,1,16 \rightarrow 16} \rightarrow Tanh \rightarrow BN) \rightarrow (rk_{4,\uparrow 2,3,16 \rightarrow 3} \rightarrow Tanh \rightarrow BN).$$

6.2.2.7 The double shallow rank autoencoder with Tanh activation.

The overall architecture is formed with two shallow autoencoders connected sequentially:

The first is:

$$(rk_{4,\downarrow 2,3,3 \rightarrow 16} \rightarrow BN \rightarrow Tanh) \rightarrow (rk_{4,\uparrow 2,3,16 \rightarrow 3} \rightarrow Tanh \rightarrow BN)$$

The second code consists of:

$$(rk_{2,\downarrow 1,1,16 \rightarrow 16} \rightarrow BN \rightarrow Tanh) \rightarrow (rk_{2,\uparrow 1,1,16 \rightarrow 16} \rightarrow Tanh \rightarrow BN).$$

6.2.2.8 The shallow rank autoencoder with Tanh activation with $\downarrow 2$.

The overall architecture is exactly the first shallow autoencoder used in the previous section :

$$(rk_{4,\downarrow 2,3,3 \rightarrow 16} \rightarrow Tanh \rightarrow BN) \rightarrow (rk_{4,\uparrow 2,3,16 \rightarrow 3}^T \rightarrow BN \rightarrow Tanh)$$

6.2.2.9 The shallow rank autoencoder with Tanh activation with FoV=2 × 2.

The overall architecture is exactly the second block of the double shallow rank autoencoder :

$$(rk_{2,\downarrow 1,1,3 \rightarrow 16} \rightarrow Tanh \rightarrow BN) \rightarrow (rk_{2,\uparrow 1,1,16 \rightarrow 3}^T \rightarrow BN \rightarrow Tanh)$$

6.2.2.10 The double shallow linear autoencoder.

This is the linear version of the rank double shallow autoencoder. The overall architecture is formed with two shallow linear autoencoders connected sequentially: The first is:

$$(C_{4,\downarrow 2,3,3 \rightarrow 16} \rightarrow BN \rightarrow ReLU) \rightarrow (C_{4,\uparrow 2,3,16 \rightarrow 3}^T \rightarrow ReLU \rightarrow BN)$$

And the second is:

$$(C_{2,\downarrow 1,1,16 \rightarrow 16} \rightarrow BN \rightarrow ReLU) \rightarrow (C_{2,\uparrow 1,1,16 \rightarrow 16}^T \rightarrow ReLU \rightarrow BN).$$

6.2.2.11 Autoencoder with Rank outer layers and Tanh activation in linear layers.

The net is similar to the baseline linear autoencoder on the experiment list (the first autoencoder architecture presented in this section). The difference is that we replace the outer layers with rank layers, and we left the inner ones intact. The coder is:

$$(rk_{4,\downarrow 2,3,3 \rightarrow 16} \rightarrow BN \rightarrow Tanh) \rightarrow (C_{2,\downarrow 1,1,16 \rightarrow 16} \rightarrow BN \rightarrow Tanh).$$

The decoder is formed of two blocks:

$$(C_{2,\uparrow 1,1,16 \rightarrow 16}^T \rightarrow Tanh \rightarrow BN) \rightarrow (rk_{4,\uparrow 2,3,16 \rightarrow 3}^T \rightarrow Tanh \rightarrow BN).$$

6.2.2.12 Autoencoder with Rank inner layers and Tanh activation in linear layers.

The net is similar to the baseline autoencoder except that this time we replaced the inner layers with rank ones. The coder is:

$$(C_{4,\downarrow 2,3,3 \rightarrow 16} \rightarrow BN \rightarrow Tanh) \rightarrow (rk_{2,\downarrow 1,1,16 \rightarrow 16} \rightarrow BN \rightarrow Tanh).$$

The decoder is formed of two blocks:

$$(rk_{2,\uparrow 1,1,16 \rightarrow 16}^T \rightarrow Tanh \rightarrow BN) \rightarrow (C_{4,\uparrow 2,3,16 \rightarrow 3}^T \rightarrow Tanh \rightarrow BN).$$

6.2.2.13 CLR range search test.

Learning rate range $\mu \in [\mu_{opt}; \mu_{max}]$.

Architecture name	$\mu_{opt} \ell_1$	$\mu_{max} \ell_1$	$\mu_{opt} \ell_2$	$\mu_{max} \ell_2$
Linear of depth 4	$1.6 \cdot 10^{-6}$	$4 \cdot 10^{-5}$	10^{-6}	$4 \cdot 10^{-5}$
Rank ReLU of depth 4	$3 \cdot 10^{-6}$	$6 \cdot 10^{-5}$	10^{-6}	$5 \cdot 10^{-5}$
Rank BN only of depth 4	10^{-6}	10^{-5}	10^{-6}	$3 \cdot 10^{-6}$
Rank Tanh of depth 4	$1.6 \cdot 10^{-6}$	$3 \cdot 10^{-5}$	$1.4 \cdot 10^{-6}$	$3 \cdot 10^{-5}$
Double shallow Rank Tanh	$2 \cdot 10^{-6}$	$2 \cdot 10^{-5}$	$2 \cdot 10^{-6}$	$2 \cdot 10^{-5}$
Shallow Rank Tanh with $\downarrow 2$	$1.6 \cdot 10^{-6}$	$4 \cdot 10^{-5}$	10^{-7}	$2.5 \cdot 10^{-6}$
Shallow Rank Tanh with FoV=2	$2 \cdot 10^{-6}$	$6 \cdot 10^{-6}$	$2 \cdot 10^{-6}$	$2 \cdot 10^{-5}$
Double shallow Linear	$1 \cdot 10^{-6}$	$2 \cdot 10^{-5}$	$2 \cdot 10^{-6}$	$2 \cdot 10^{-4}$
Autoencoder with Rank outer layers	$1.6 \cdot 10^{-6}$	$4 \cdot 10^{-5}$	$2 \cdot 10^{-6}$	$5 \cdot 10^{-5}$
Autoencoder with inner Rank layer all Tanh	$2 \cdot 10^{-6}$	$5 \cdot 10^{-5}$	$2 \cdot 10^{-6}$	$5 \cdot 10^{-5}$

Table 6.3: LR search test results for the interactions of rank with linear layers in autoencoders given that the cost function is either the ℓ_1 or ℓ_2 cost. When the μ_{max} was conducted under the ℓ_2 cost, the ones that yielded the highest learning rate have shown a relatively small plateau and a rapid decrease of performance when $\mu > 2 \times \mu_{max}$. The plateau was very large under the ℓ_1 cost and difficult to pinpoint from what value of μ it started. On this plateau there was no clear minimum while on the ℓ_2 cost a minima could sometimes be observed.

6.2.2.14 Experiment results.

Quantitative results for experiment 2.

Id	Architecture name	ℓ_1	ℓ_2	ℓ_1 Figure	ℓ_2 Figure
1	Linear of depth 4	0.164	0.0586	6.4f	6.4b
2	Rank ReLU of depth 4	0.173	.0811	6.4g	6.4c
3	Rank BN only of depth 4	0.176	0.0917	6.4h	6.4d
4	Rank Tanh of depth 4	0.168	0.0784	6.4m	6.4i
5	Double shallow Rank Tanh	0.146	0.0447	6.4n	6.4j
6	Shallow Rank Tanh with $\downarrow 2$	0.181	0.0903	6.4o	6.4k
7	Shallow Rank Tanh with FoV=2	0.166	0.0658	6.4p	6.4l
8	Double shallow Linear	0.188	0.0797	6.5d	6.5a
9	Autoencoder with Rank outer layers Tanh	0.168	0.0983	6.5e	6.5b
10	Autoencoder with Rank inner layers Tanh	0.178	0.0591	6.5f	6.5c

Table 6.4: Quantitative results for the rank autoencoder depth test. Remarkably, the best method is the double shallow rank autoencoder. It beats the linear method by a significant margin. If we separate the best method shallow autoencoders, we get the nets of the two rows below the best one in the table. Neither one of them beat the linear autoencoder, but put together they do. It is curious to see that the double shallow linear codec does not beat the deep linear autoencoder, that is the opposite behaviour for the rank filter where the double shallow one beat the full one by a large margin.

The results in Table 6.4 show that the Rank autoencoders of depth 4 using the ReLU activation had similar results compared to the one that used the Tanh activation. We only tested the double shallow net with the Tanh activation under the bias that the ReLU activation might be problematic for three possible reasons :

1. ReLU placed shortly after the ranking operation waters down the potential of the rank filter. ReLU after a unranking or transposed rank filter is fine.
2. The rank filter is already a non-linear function so activating it isn't as crucial compared to the all linear convolution.
3. It might not attenuate large values enough.

We shall test this in experiment 3 in Section 6.2.3.

If we compare the two double shallow autoencoders, we notice that the rank one is much better than the linear. Thus, we can confidently assert that rank filters are better to be under the form of shallow autoencoders. But maybe the linear nets are good at generating data, versus the rank filters that could be just better at replacing data(losing bad data in this case). The fact that the double rank filter has a worse performance compared to the double shallow rank net may be evidence for Hypothesis 3 where we mentioned that rank filters work best as shallow filters. A clear advantage of this architecture is that we can design it is that we can design them to use less weights than linear coders. For example in this experiment, the total number of weights of the depth 4 Linear net is of $2 \cdot (3 \cdot 16 \cdot 4 \cdot 4 + 16 \cdot 16 \cdot 2 \cdot 2) = 3584$, compared to $2 \cdot (3 \cdot 16 \cdot 4 \cdot 4 + 3 \cdot 16 \cdot 2 \cdot 2) = 1920$. Furthermore, the use of shallow rank codecs has outstanding potential in a skip-connections since they can “clean” the input on the way and refining the output without losing too much initial information. Additionally, we saw that the shallow rank codecs could be placed in the center of a larger linear filter autoencoder without any clear advantage over the all linear net.

Looking at Figures 6.4f with 6.4n we can see that the double shallow rank autoencoder reconstructs differently from the linear net. The linear net appears smoother, and the rank-net reconstruction appears crisper.

Note that the autoencoder of depth four that had two rank middle layers was slightly worse than the linear one. Additionally, the codec that had outer rank layers wasn't particularly useful. This observation may suggest that deep rank features or deep linear features from rank ones aren't particularly useful, but that a shallow rank autoencoder can be dealt with by an outer linear net. We also noticed that the rank filters generally performed better under the ℓ_2 cost function compared to the ℓ_1 cost. This is probably because the ℓ_2 penalizes more large errors and leading to a back-propagation of derivatives through values that were classed as outlying ranks. However, the smaller errors in the ℓ_1 cost function weren't as easily transmitted through by backprop since they came from ambiguous ranks (because ranks are highly correlated).



Figure 6.4: Denoising net qualitative results (part 1)

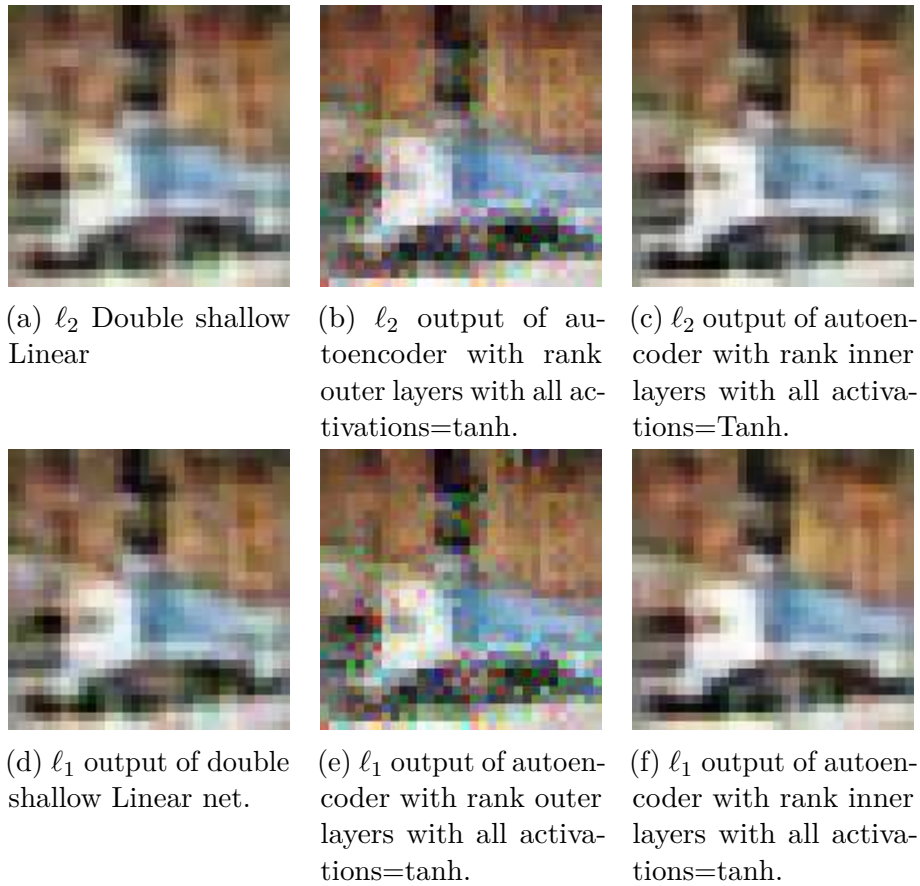


Figure 6.5: Denoising net qualitative results (part 2)

6.2.3 Experiment 3: Search an improved architecture inspired from the best combinations in experiment 1 and 2.

This experiment aims to search an “improved” architecture in the sense that we continue the search for architectures that yield better image reconstruction results directly inspired by designs that best performed in the previous experiment.

6.2.3.1 Methodology

We will use the CIFAR-10 dataset with the same setup as in Section 6.2.2 except that this time, we use a mini-batch size of 512, and we initialize weights to simulate average pooling skip-connection fully. This simulation is forced onto the weights by using the following initialization:

$$W_{i,j,k_i,k_o} = \begin{cases} \frac{1}{FoV} & k_i = k_o \\ 0 & k_i \neq k_o \end{cases}; (k_i, k_o) < \min(C_{in}, C_{out}) \quad (6.1)$$

This initialization is applied only to all coders, including linear and rank types, and permitted us to use higher learning rates. A notable example is for the “Double Codec 2,” where we found that $\mu_{max} = 10^{-5}$ without the initialization and much slower convergence to a high value. In contrast, with the initialization, we could obtain $\mu_{max} = 3 \cdot 10^{-4}$, faster convergence, and a much better final performance. However, we noticed in subsequent experiments that it was better to add the average pooling weights to the random initialization. The mean of the weights is around the average pooling initialization, like in experiment 2 in Section 6.2.2. This convergence was found after the investigation was conducted (the difference between both methods isn’t huge). A reason for this may be because we set off-diagonal weights to 0 if $(k_i, k_o) < \min(C_{in}, C_{out})$ resulting weights that have a variance too small compared to the rest resulting in slower convergence. We did consider the initialization of parts of the weights to max-pooling but with worse results. This is probably because the max-pooling weights are much larger than the weights, thus taking much longer to train by possibly putting the Tanh activation into the dead-gradient zone. Maybe biasing the weights with max-pooling fares better in other architectures.

In experiment 1, recall that we used the Leaky ReLU parameter as a linearity measure gauge. We found that the coder got a lower a than the decoder, this might suggest that the decoder activation should be better off as a ReLU and the first activation something linear like (at least close to 0) for example the Tanh function. Hence in our tested architectures, we are going to alternate between ReLU and Tanh in the decoder.

In experiment 1, we used high resolution, FoV, and capacity shallow decoders. In this experiment, we are going to use a low-resolution, low-capacity, and high-FoV shallow autoencoder for the first block. Then, following the first block, we have a high-resolution, low-capacity, and low FoV block. We will test on a lot of different architectures, and we will sort them concerning their ℓ_2 score to see which layer combination is best. We will also look at the images generated to give us a good intuition on why some nets are better than others.

6.2.3.2 Double Codec 1

The net is:

$$(rk_{4,\downarrow 2,3,3\rightarrow 16}\rightarrow BN\rightarrow Tanh)\rightarrow (rk_{4,\uparrow 2,3,16\rightarrow 3}^T\rightarrow Tanh\rightarrow BN)\rightarrow (rk_{2,\downarrow 1,1,16\rightarrow 16}\rightarrow BN\rightarrow Tanh)\rightarrow (rk_{2,\uparrow 1,1,16\rightarrow 16}^T\rightarrow Tanh\rightarrow BN).$$

6.2.3.3 Double Codec 2

The net is:

$$(rk_{4,\downarrow 2,3,3\rightarrow 16}\rightarrow BN\rightarrow Tanh)\rightarrow (rk_{4,\uparrow 2,3,16\rightarrow 3}^T\rightarrow ReLU\rightarrow BN)\rightarrow (rk_{2,\downarrow 1,1,16\rightarrow 16}\rightarrow BN\rightarrow Tanh)\rightarrow (rk_{2,\uparrow 1,1,16\rightarrow 16}^T\rightarrow ReLU\rightarrow BN).$$

6.2.3.4 Double Codec 3

The net is:

$$(rk_{4,\downarrow 2,3,3\rightarrow 16}\rightarrow BN\rightarrow Tanh)\rightarrow (rk_{4,\uparrow 2,3,16\rightarrow 3}^T\rightarrow Tanh\rightarrow BN)\rightarrow (rk_{2,\downarrow 1,1,16\rightarrow 16}\rightarrow BN\rightarrow Tanh)\rightarrow (rk_{2,\uparrow 1,1,16\rightarrow 16}^T\rightarrow ReLU\rightarrow BN).$$

6.2.3.5 Double Codec 4

The net is:

$$(rk_{4,\downarrow 2,3,3\rightarrow 16}\rightarrow BN\rightarrow Tanh)\rightarrow (rk_{4,\uparrow 2,3,16\rightarrow 3}^T\rightarrow ReLU\rightarrow BN)\rightarrow (rk_{2,\downarrow 1,1,16\rightarrow 16}\rightarrow BN\rightarrow Tanh)\rightarrow (rk_{2,\uparrow 1,1,16\rightarrow 16}^T\rightarrow Tanh\rightarrow BN).$$

6.2.3.6 Double Codec 5

The net is:

$$(rk_{4,\downarrow 2,3,3\rightarrow 16}\rightarrow BN\rightarrow Tanh)\rightarrow (C_{4,\uparrow 2,3,16\rightarrow 3}^T\rightarrow ReLU\rightarrow BN)\rightarrow (rk_{2,\downarrow 1,1,16\rightarrow 16}\rightarrow BN\rightarrow Tanh)\rightarrow (rk_{2,\uparrow 1,1,16\rightarrow 16}^T\rightarrow ReLU\rightarrow BN).$$

6.2.3.7 Double Codec 6

The net is:

$$(rk_{4,\downarrow 2,3,3\rightarrow 16}\rightarrow BN\rightarrow Tanh)\rightarrow (C_{4,\uparrow 2,3,16\rightarrow 3}^T\rightarrow ReLU\rightarrow BN)\rightarrow (rk_{2,\downarrow 1,1,16\rightarrow 16}\rightarrow BN\rightarrow Tanh)\rightarrow (C_{2,\uparrow 1,1,16\rightarrow 16}^T\rightarrow ReLU\rightarrow BN).$$

6.2.3.8 Double Codec 7

The net is:

$$(rk_{4,\downarrow 2,3,3\rightarrow 16}\rightarrow BN\rightarrow Tanh)\rightarrow (C_{4,\uparrow 2,3,16\rightarrow 3}^T\rightarrow ReLU\rightarrow BN)\rightarrow (C_{2,\downarrow 1,1,16\rightarrow 16}\rightarrow BN\rightarrow Tanh)\rightarrow (C_{2,\uparrow 1,1,16\rightarrow 16}^T\rightarrow ReLU\rightarrow BN).$$

6.2.3.9 Double Codec 8

The net is:

$$(rk_{4,\downarrow 2,3,3 \rightarrow 16} \rightarrow BN \rightarrow Tanh) \rightarrow (C_{4,\uparrow 2,3,16 \rightarrow 3}^T \rightarrow ReLU \rightarrow BN) \rightarrow (C_{2,\downarrow 1,1,16 \rightarrow 16} \rightarrow BN \rightarrow ReLU) \rightarrow (C_{2,\uparrow 1,1,16 \rightarrow 16}^T \rightarrow ReLU \rightarrow BN).$$

6.2.3.10 Double Codec 9

The net is:

$$(rk_{4,\downarrow 2,3,3 \rightarrow 16} \rightarrow BN \rightarrow Tanh) \rightarrow (rk_{4,\uparrow 2,3,16 \rightarrow 3}^T \rightarrow ReLU \rightarrow BN) \rightarrow (C_{2,\downarrow 1,1,16 \rightarrow 16} \rightarrow BN \rightarrow Tanh) \rightarrow (C_{2,\uparrow 1,1,16 \rightarrow 16}^T \rightarrow ReLU \rightarrow BN).$$

6.2.3.11 Double Codec 10

The net is:

$$(rk_{4,\downarrow 2,3,3 \rightarrow 16} \rightarrow BN \rightarrow Tanh) \rightarrow (rk_{4,\uparrow 2,3,16 \rightarrow 3}^T \rightarrow Tanh \rightarrow BN) \rightarrow (rk_{2,\downarrow 1,1,16 \rightarrow 16} \rightarrow BN \rightarrow Tanh) \rightarrow (C_{2,\uparrow 1,1,16 \rightarrow 16}^T \rightarrow Tanh \rightarrow BN).$$

6.2.3.12 Double Codec 11

The net is:

$$(rk_{4,\downarrow 2,3,3 \rightarrow 16} \rightarrow BN \rightarrow Tanh) \rightarrow (C_{4,\uparrow 2,3,16 \rightarrow 3}^T \rightarrow Tanh \rightarrow BN) \rightarrow (rk_{2,\downarrow 1,1,16 \rightarrow 16} \rightarrow BN \rightarrow Tanh) \rightarrow (C_{2,\uparrow 1,1,16 \rightarrow 16}^T \rightarrow Tanh \rightarrow BN).$$

6.2.3.13 Double Codec 12

The net is:

$$(rk_{4,\downarrow 2,3,3 \rightarrow 16} \rightarrow BN \rightarrow Tanh) \rightarrow (rk_{4,\uparrow 2,3,16 \rightarrow 3}^T \rightarrow Tanh \rightarrow BN) \rightarrow (C_{2,\downarrow 1,1,16 \rightarrow 16} \rightarrow BN \rightarrow Tanh) \rightarrow (C_{2,\uparrow 1,1,16 \rightarrow 16}^T \rightarrow Tanh \rightarrow BN).$$

6.2.3.14 Double Codec 13

The net is:

$$(rk_{4,\downarrow 2,3,3 \rightarrow 16} \rightarrow BN \rightarrow Tanh) \rightarrow (C_{4,\uparrow 2,3,16 \rightarrow 3}^T \rightarrow Tanh \rightarrow BN) \rightarrow (C_{2,\downarrow 1,1,16 \rightarrow 16} \rightarrow BN \rightarrow Tanh) \rightarrow (C_{2,\uparrow 1,1,16 \rightarrow 16}^T \rightarrow Tanh \rightarrow BN).$$

6.2.3.15 Double Codec 14

The net is:

$$(rk_{4,\downarrow 2,3,3 \rightarrow 16} \rightarrow BN \rightarrow Tanh) \rightarrow (rk_{4,\uparrow 2,3,16 \rightarrow 3}^T \rightarrow Tanh \rightarrow BN) \rightarrow (rk_{2,\downarrow 1,1,16 \rightarrow 16} \rightarrow BN \rightarrow Tanh) \rightarrow (C_{2,\uparrow 1,1,16 \rightarrow 16}^T \rightarrow ReLU \rightarrow BN).$$

6.2.3.16 Experiment results.

We will use a simple code to name the codec’s blocks in Tables 6.5 and 6.6 because this way we can easily keep track of the architecture without giving them proper naming. A coder or decoder is labelled with three symbols $[aBC]$. Each symbol corresponds to it’s component layers labelled in order. The first one is a small caption letter either “c” or “r” for linear convolution and rank filtering respectively and it will have the transpose subscript “T” to label if it belongs to a decoder. The second symbol corresponds to the subsequent layer and it could be “B” for BN, “R” for ReLU, and “T” for Tanh.

Quantitative ℓ_2 performance results for double codecs with their CLR test range and the figure output references.

Id	Codec layer labels	μ_{opt}	μ_{max}	ℓ_2	Figure
1	$rBT r^T TB rBT r^T TB$	$1 \cdot 10^{-6}$	$2 \cdot 10^{-5}$.0459	6.6a
2	$rBT r^T RB rBT r^T RB$	$1 \cdot 10^{-6}$	$3 \cdot 10^{-4}$.0464	6.6b
3	$rBT r^T TB rBT r^T RB$	$1 \cdot 10^{-6}$	$5 \cdot 10^{-5}$.0468	6.6c
4	$rBT r^T RB rBT r^T TB$	$1 \cdot 10^{-6}$	$4 \cdot 10^{-5}$.0452	6.6d
5	$rBT c^T RB rBT r^T RB$	$1 \cdot 10^{-6}$	$6 \cdot 10^{-4}$.0706	6.6e
6	$rBT c^T RB rBT c^T RB$	$1 \cdot 10^{-6}$	$4 \cdot 10^{-4}$.0757	6.6f
7	$rBT c^T RB cBT c^T RB$	$2 \cdot 10^{-6}$	$4 \cdot 10^{-4}$.0739	6.6g
8	$rBT c^T RB cBR c^T RB$	$2 \cdot 10^{-6}$	$4 \cdot 10^{-4}$.0737	6.6h
9	$rBT r^T RB cBT c^T RB$	$5 \cdot 10^{-6}$	$4 \cdot 10^{-4}$.0444	6.6i
10	$rBT r^T TB rBT c^T TB$	$1 \cdot 10^{-6}$	$1.5 \cdot 10^{-4}$.0451	6.6j
11	$rBT c^T TB rBT c^T TB$	$2 \cdot 10^{-7}$	$2 \cdot 10^{-5}$.0749	6.6k
12	$rBT r^T RB cBT c^T TB$	$1 \cdot 10^{-6}$	$4 \cdot 10^{-5}$.0456	6.6l
13	$rBT c^T TB cBT c^T TB$	$1 \cdot 10^{-6}$	$7 \cdot 10^{-5}$.0784	6.6m
14	$rBT r^T TB rBT c^T RB$	$1 \cdot 10^{-6}$	$4 \cdot 10^{-5}$.0462	6.6n

Table 6.5: Quantitative results for the double shallow codecs.

Note that whenever the final layer was a Tanh activation, it was difficult to pinpoint μ_{max} . The test curve was atypical, showing sudden large spikes for the train set score and even more extensive and earlier for the test score. If we ignored the spikes, there was a broad plateau with a slight declination (useful usually if there is no sudden drop in performance). Typical test scores do not have large spikes, although, in the CLR articles [42][41], authors test their method with ReLU activations only. If we use a learning rate corresponding to the lowest point before definite divergence, we get disproportionate regularisation at high μ values during training, and the net doesn’t recover from it. Conversely, if we pick the lowest (best) ℓ_2 point before the first spike, we get a much slower convergence⁵.

If we reorder the rows from the best score to worse, we get Table 6.6. It should help in seeing what could be the common denominator between the best methods. We indeed

⁵ Ironically, there is still somewhat of a guessing game going on for the CLR test, but this technique requires some trials and is a much better alternative than pure guessing. Thanks to this technique, we were able to save time and perform many experiments.

observe that activation types are somewhat essential, and they can potentially be different depending on where they are located. In general, it seems that the best activation is Tanh for rank-blocks. We also found nets with ReLU activations at the very end of an all-rank chain were slightly less performant. However, the exact opposite is found for the linear layers. We, unfortunately, had to rerun this experiment because rank decoders were sensitive to bias initialization. The training was terrible for these and, we found that initializing the bias to 0 helped (although this initialization lowered the score slightly for the linear ones and Tanh-based rank decoders). We see from Table 6.6 that the best nets had approximately the same score. What the worse layers show in Figure 6.6 is that a linear decoder in the second block yielded too much smoothing for the second layer to manage. Nets that had a first shallow fully rank-based autoencoder block were the best.

Although similar in score, the best net (net 9) consistently fell in first place even after trying many μ ranges and different initializations. The best net architecture is $rBT|r^T RB|cBT|c^T RB$. Probably a better option would have been $rBT|r^T RB|cBR|c^T RB$ since ReLU layers work well with linear blocks.

Quantitative ℓ_2 performance results for double codecs.

Id	Codec layer labels	ℓ_2	Figure
9	$rBT r^T RB cBT c^T RB$.0444	6.6i
10	$rBT r^T TB rBT c^T TB$.0451	6.6j
4	$rBT r^T RB rBT r^T TB$.0452	6.6d
12	$rBT r^T RB cBT c^T TB$.0456	6.6l
1	$rBT r^T TB rBT r^T TB$.0459	6.6a
14	$rBT r^T TB rBT c^T RB$.0462	6.6n
2	$rBT r^T RB rBT r^T RB$.0464	6.6b
3	$rBT r^T TB rBT r^T RB$.0468	6.6c
5	$rBT c^T RB rBT r^T RB$.0706	6.6e
8	$rBT c^T RB cBR c^T RB$.0737	6.6h
7	$rBT c^T RB cBT c^T RB$.0739	6.6g
11	$rBT c^T TB rBT c^T TB$.0749	6.6k
6	$rBT c^T RB rBT c^T RB$.0757	6.6f
13	$rBT c^T TB cBT c^T TB$.0784	6.6m

Table 6.6: Reordered quantitative results for the double shallow codecs.

Although subjective, we may say that Figures 6.6i and 6.6l(i.e. 9 and 12) yielded crisper qualitative results, although this may be for that specific figure and could be a coincidence. Moreover, the two pictures belong to the best net and the net that came in fourth place. This qualitative result may suggest that shallow rank decoders are suitable for removing false information, but aren't useful for generating rich details.

To conclude, given that it was hard to say which net was the best, except for net 9, since the other nets ℓ_2 score were similar, we will say that nets 9 and 12 were the best, given the qualitative data.

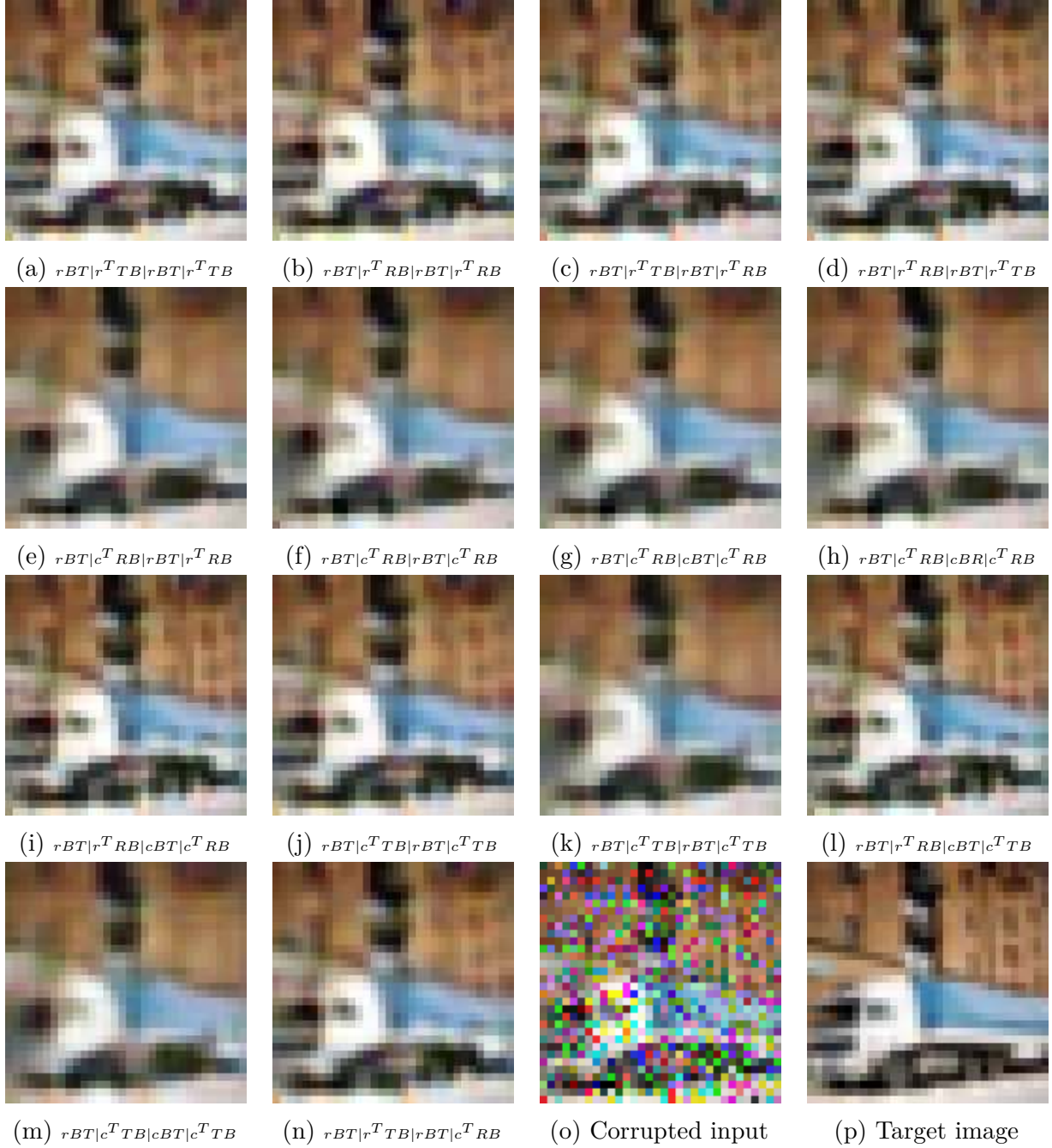


Figure 6.6: Experiment 3 qualitative results. Output of nets compared with input and target.

6.2.4 Experiment 4: Effects of reducing filter FoV or resolution when image is corrupted under spiked type noise for a shallow autoencoder.

We designed this experiment to gather evidence for Hypothesis 2.

6.2.4.1 Methodology

We will present three tables in this section having each row specifying which architecture was used. The noise and methodology are identical to the one used in the previous experiment. Here we are going to compare both rank filtering and linear filtering when we vary the FoV and strides of the simplest codec. In table 6.7, we see that the methods are about the same

Autoencoder and it's corresponding coder-decoder pair with learning rate range of CLR and results (ℓ_2).

Coder	Decoder	μ_{opt}	μ_{max}	ℓ_2
$rk_{4,\downarrow 1,3,3\rightarrow 32}\rightarrow BN\rightarrow Th$	$rk_{4,\uparrow 1,3,32\rightarrow 3}\rightarrow R\rightarrow BN$	$1.5\cdot 10^{-5}$	$1\cdot 10^{-4}$.0662
$rk_{4,\downarrow 1,3,3\rightarrow 32}\rightarrow BN\rightarrow Th$	$C_{4,\uparrow 1,3,32\rightarrow 3}^T\rightarrow R\rightarrow BN$	$1.5\cdot 10^{-5}$	$1\cdot 10^{-4}$.0722
$rk_{3,\downarrow 1,2,3\rightarrow 32}\rightarrow BN\rightarrow Th$	$rk_{3,\uparrow 1,2,32\rightarrow 3}\rightarrow R\rightarrow BN$	$1.5\cdot 10^{-5}$	$1\cdot 10^{-4}$.0628
$rk_{3,\downarrow 1,2,3\rightarrow 32}\rightarrow BN\rightarrow Th$	$C_{3,\uparrow 1,2,32\rightarrow 3}^T\rightarrow R\rightarrow BN$	$1.5\cdot 10^{-5}$	$1\cdot 10^{-4}$.0600
$rk_{2,\downarrow 1,1,3\rightarrow 32}\rightarrow BN\rightarrow Th$	$rk_{2,\uparrow 1,1,32\rightarrow 3}\rightarrow R\rightarrow BN$	$1.5\cdot 10^{-5}$	$1\cdot 10^{-4}$.0624
$rk_{2,\downarrow 1,1,3\rightarrow 32}\rightarrow BN\rightarrow Th$	$C_{2,\uparrow 1,1,32\rightarrow 3}^T\rightarrow R\rightarrow BN$	$1.5\cdot 10^{-5}$	$1\cdot 10^{-4}$.0652
$rk_{(3\times 1),\downarrow 1,(2,1),3\rightarrow 32}\rightarrow BN\rightarrow Th$	$rk_{(3\times 1),\uparrow 1,(2,1),32\rightarrow 3}\rightarrow R\rightarrow BN$	10^{-6}	$5\cdot 10^{-4}$.0989
$rk_{(3\times 1),\downarrow 1,(2,0),3\rightarrow 32}\rightarrow BN\rightarrow Th$	$C_{(3\times 1),\uparrow 1,(2,0),32\rightarrow 3}^T\rightarrow R\rightarrow BN$	10^{-6}	$5\cdot 10^{-4}$.108

Table 6.7: Experiment 4 results with stride 1.

when the overlap between frames is largest. When we increase the stride, using a rank decoder gives an advantage. However, this advantage is lost when the stride equals the FoV.

Autoencoder and it's corresponding coder-decoder pair ℓ_2 result and μ range for a stride of 2.

Coder	Decoder	μ_{opt}	μ_{max}	ℓ_2
$rk_{4,\downarrow 2,3,3\rightarrow 32}\rightarrow BN\rightarrow Th$	$rk_{4,\uparrow 2,3,32\rightarrow 3}\rightarrow R\rightarrow BN$	$2\cdot 10^{-5}$	$1\cdot 10^{-3}$.0862
$rk_{4,\downarrow 2,3,3\rightarrow 32}\rightarrow BN\rightarrow Th$	$C_{4,\uparrow 2,3,32\rightarrow 3}^T\rightarrow R\rightarrow BN$	$2\cdot 10^{-5}$	$1\cdot 10^{-3}$.0940
$rk_{3,\downarrow 2,2,3\rightarrow 32}\rightarrow BN\rightarrow Th$	$rk_{3,\uparrow 2,2,32\rightarrow 3}\rightarrow R\rightarrow BN$	$2\cdot 10^{-5}$	$4\cdot 10^{-4}$.0693
$rk_{3,\downarrow 2,2,3\rightarrow 32}\rightarrow BN\rightarrow Th$	$C_{3,\uparrow 2,2,32\rightarrow 3}^T\rightarrow R\rightarrow BN$	$2\cdot 10^{-5}$	$4\cdot 10^{-4}$.0829
$rk_{2,\downarrow 2,1,3\rightarrow 32}\rightarrow BN\rightarrow Th$	$rk_{2,\uparrow 2,1,32\rightarrow 3}\rightarrow R\rightarrow BN$	10^{-5}	$4\cdot 10^{-4}$.137
$rk_{2,\downarrow 2,1,3\rightarrow 32}\rightarrow BN\rightarrow Th$	$C_{2,\uparrow 2,1,32\rightarrow 3}^T\rightarrow R\rightarrow BN$	10^{-5}	$4\cdot 10^{-4}$.1490
$rk_{(3\times 1),\downarrow (2,1),(2,1),3\rightarrow 32}\rightarrow BN\rightarrow Th$	$rk_{(3\times 1),\uparrow (2,1),(2,1),32\rightarrow 3}\rightarrow R\rightarrow BN$	10^{-5}	$1\cdot 10^{-3}$.123
$rk_{(3\times 1),\downarrow (2,1),(2,0),3\rightarrow 32}\rightarrow BN\rightarrow Th$	$C_{(3\times 1),\uparrow (2,1),(2,0),32\rightarrow 3}^T\rightarrow R\rightarrow BN$	10^{-5}	$3\cdot 10^{-4}$.122

Table 6.8: Experiment 4 results with stride 2.

Autoencoder and it’s corresponding coder-decoder pair ℓ_2 result and μ range for a stride of 3 and 4.

Coder	Decoder	μ_{opt}	μ_{max}	ℓ_2
$rk_{4,\downarrow 3,3,3 \rightarrow 32} \rightarrow BN \rightarrow Th$	$rk_{4,\uparrow 3,3,32 \rightarrow 3}^T \rightarrow R \rightarrow BN$	10^{-6}	$4 \cdot 10^{-4}$.0927
$rk_{4,\downarrow 3,3,3 \rightarrow 32} \rightarrow BN \rightarrow Th$	$C_{4,\uparrow 3,3,32 \rightarrow 3}^T \rightarrow R \rightarrow BN$	10^{-6}	$4 \cdot 10^{-4}$.122
$rk_{4,\downarrow 4,2,3 \rightarrow 32} \rightarrow BN \rightarrow Th$	$rk_{4,\uparrow 4,2,32 \rightarrow 3}^T \rightarrow R \rightarrow BN$	10^{-6}	$4 \cdot 10^{-4}$.133
$rk_{4,\downarrow 4,2,3 \rightarrow 32} \rightarrow BN \rightarrow Th$	$C_{4,\uparrow 4,2,32 \rightarrow 3}^T \rightarrow R \rightarrow BN$	10^{-6}	$4 \cdot 10^{-4}$.122
$rk_{3,\downarrow 3,2,3 \rightarrow 32} \rightarrow BN \rightarrow Th$	$rk_{3,\uparrow 3,2,32 \rightarrow 3}^T \rightarrow R \rightarrow BN$	$1.6 \cdot 10^{-5}$	10^{-4}	.129
$rk_{3,\downarrow 3,2,3 \rightarrow 32} \rightarrow BN \rightarrow Th$	$C_{3,\uparrow 3,2,32 \rightarrow 3}^T \rightarrow R \rightarrow BN$	$1.6 \cdot 10^{-5}$	10^{-4}	.128
$rk_{(3 \times 1),\downarrow (3,1),(2,1),3 \rightarrow 32} \rightarrow BN \rightarrow Th$	$rk_{(3 \times 1),\uparrow (3,1),(2,1),32 \rightarrow 3}^T \rightarrow R \rightarrow BN$	10^{-5}	$1 \cdot 10^{-3}$.160
$rk_{(3 \times 1),\downarrow (3,1),(2,0),3 \rightarrow 32} \rightarrow BN \rightarrow Th$	$C_{(3 \times 1),\uparrow (3,1),(2,0),32 \rightarrow 3}^T \rightarrow R \rightarrow BN$	10^{-5}	$3 \cdot 10^{-4}$.169

Table 6.9: Experiment 4 results with stride 3.

From the tables, we see that the best results are achieved whenever the FoV of rank filters are not greater than 3×3 . Additionally, we see that the best results coincide with small strides. Hence we can conclude that Hypothesis 2 is true: *Transposed rank layers with large Field of View (FoV) should not be useful for deep decoders using permutations from the coder, but should be less problematic for shallow decoders since they use a smaller neighbourhood (more correlation).*

6.2.5 Experiment 5: Dense Linear and Rank Filtering with 1D shuffling.

To test the robustness of deep rank networks, we conduct this experiment. Additionally, we check if the shuffle operation to combine features from different resolutions (FoVs) of filters linear or not. The nets used in this experiment are described in Figure 6.7, with the exception of the net with 2 rank based codec blocks. As shown in Figure 6.7, the networks are deep. We designed the nets this way to see how rank filters behave in deep nets. The neural networks have dense skip-connections, and each convolution outputs an image with the same size as the input image X . The dense input is then reorganized with the shuffle-1D operation in order to have a single pixel resolution (best resolution possible) processing achieved by the final convolutional layer.

6.2.5.1 Methodology

The CIFAR-10 database is used in this experiment. At each epoch, we estimate the test set ℓ_1 loss with only a single batch of 256 randomly picked from the test set. The images pixel channels have a 1/2 probability of being corrupted by a Gaussian mixture noise that has three modes. One of the means of the Gaussian spikes is located at 0. The other two are located at two-thirds of the minimum and maximum possible pixel value. The standard deviation of the spikes is $\sigma = 1$, and the three noise modes occur at the same rate. This means that the chance of seeing a pixel with all RGB channels uncorrupted is of $(1 - \frac{1}{2})^3 \approx 12.5\%$. Note that the noise saturates at possible values corresponding to the pixel amplitude range of

[0; 255], therefore the noise is clipped at the minimum and maximal values for a normalized image. A total of 64 epochs are used for training. We test the net in Figure 6.7. The baseline net has linear blocks with its first block being activated with ReLU instead of Tanh. The activation choice was made because, in previous experiments using a Tanh activation in the coder of a shallow autoencoder was beneficial. As shown in Figure 6.8, we tested the nets once they were done training with various different image corruption rates where we varied the corruption all the way to 100%.

The reason why we Gaussian mixture noise that has three modes is because it is a severe type of noise compared to a single spike Gaussian noise. The neural net needs to identify the modes and “fix” them. It is easier to fix a single-mode than three peaks. **The noise corruption is very high**, having the effect rendering the image unrecognizable for a human observer. We chose this type of noise because it is a difficult task easily handled by deep nets. In this experiment, we use very deep networks in order to handle this difficult task. This experiment was conducted multiple times with less “aggressive” noise, but the conclusions were the same.

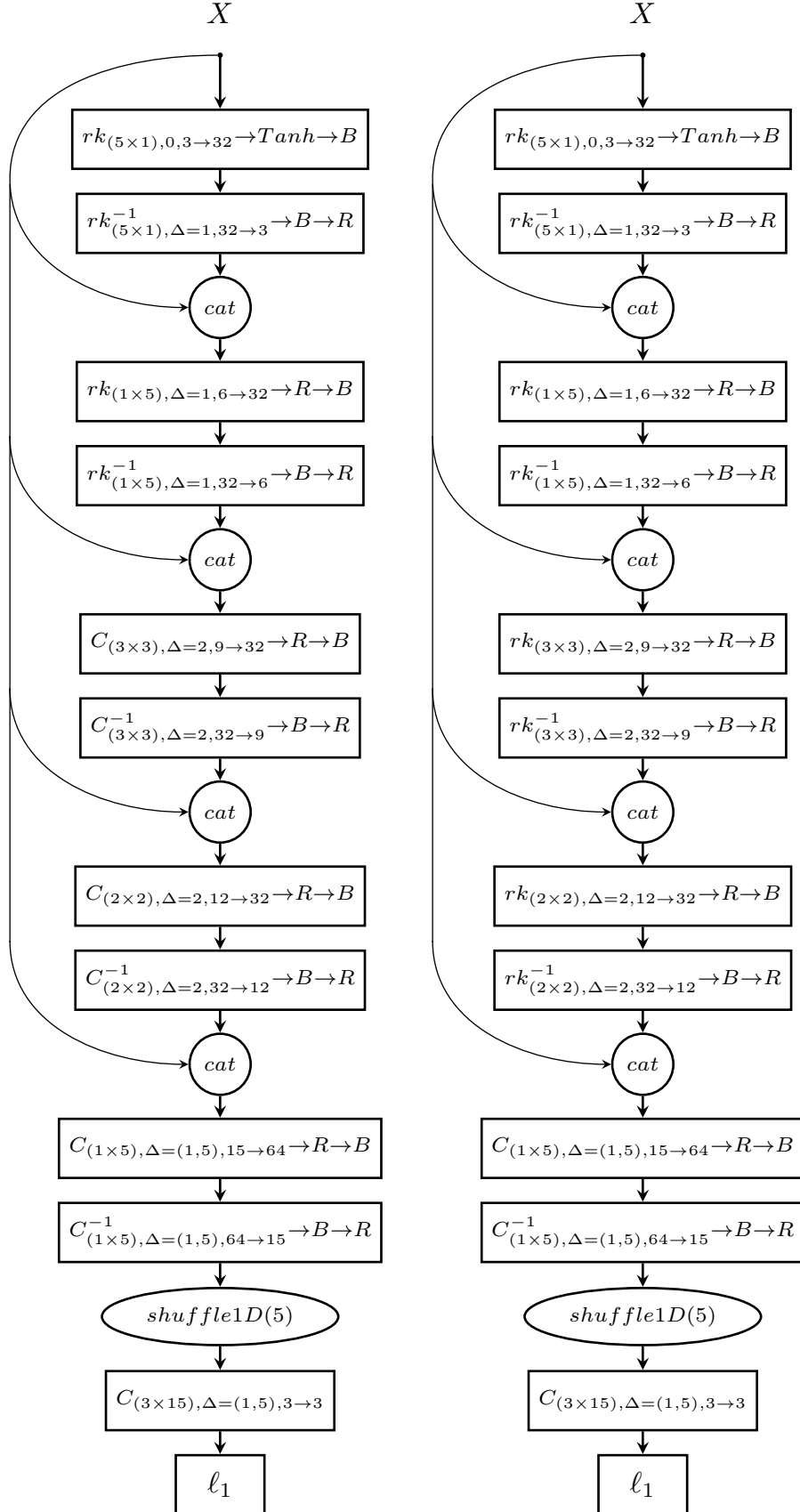


Figure 6.7: Rank nets for denoising at 5. The order of the left net rank codes as first blocks. The right hand side net has all rank layers before the last concatenation. For both nets there are skip connections where the input is concatenated with codecs outputs.

6.2.5.2 Experiment results & Discussion.

The results show that there is not a big difference between all nets. Additionally, the all-linear net performs slightly better than the rank based networks. Rank-based filtering yields poor high-resolution content compared to linear convolution. The baseline linear net performed better than the other nets. This suggests that rank filtering isn't very good for encoding geometrical information in deep neural networks. It has less capacity than all-linear layers. In spite of this, only the rank based net with four codecs showed better generalization than the all-linear layer.

Autoencoder and it's corresponding coder-decoder pair ℓ_1 result and μ range .

Net	μ_{opt}	μ_{max}	ℓ_1
4 first codecs are rank based	10^{-6}	$3 \cdot 10^{-5}$.121
2 first codecs are rank based	10^{-6}	$2 \cdot 10^{-4}$.119
All linear baseline	10^{-6}	$1 \cdot 10^{-4}$.112

Table 6.10: Experiment 5 results.

We see in Figure 6.8 that the net with four rank codecs generalizes better image reconstruction for low corruption levels compared to the baseline network. The net that has two first rank wasn't included in the figure since it displayed a slightly worse performance than the baseline network on all corruption rates.

Loss versus corruption rate.

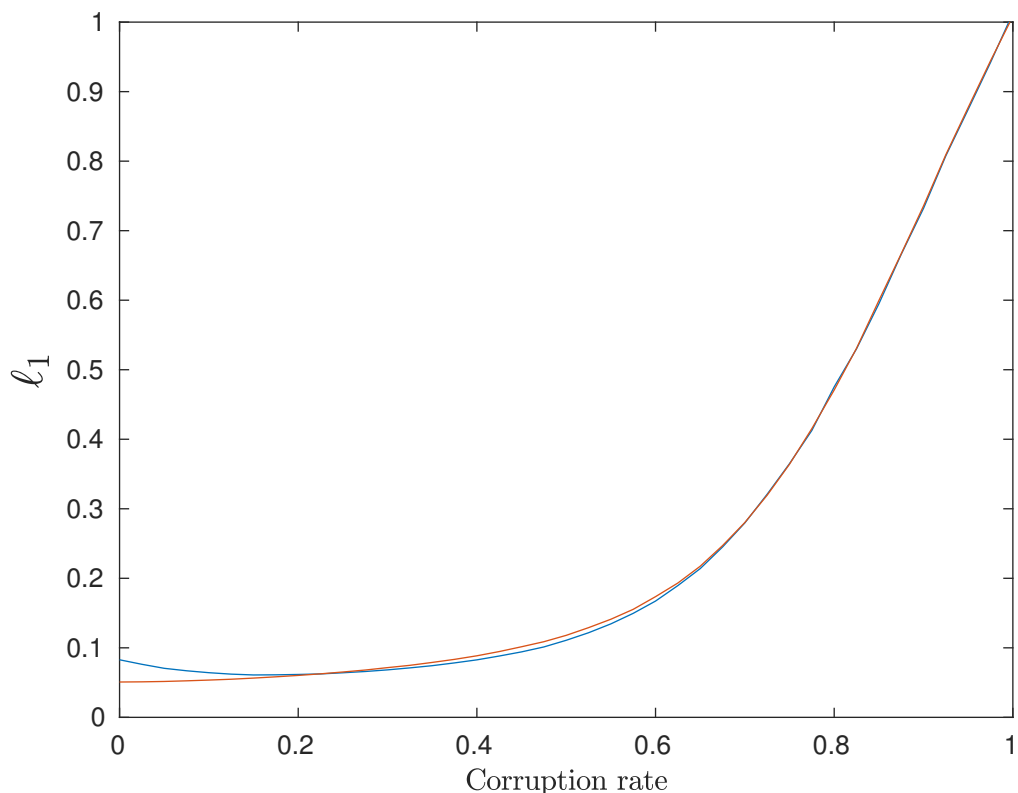


Figure 6.8: We see how the all linear net and the net with four rank codecs perform for different rates of corruption after they were trained on an 50% corruption rate. The blue curve is the all-linear baseline network, and the red curve is the net with four codec blocks. We see that the all-linear net does outperform the rank based net in a region around the corruption rate of 50%. However, the rank-based net shows better robustness for low corruption rates.

6.2.5.3 Rank weights

Here we show the weights visualizations for the net with the 4 rank codec block in Figures 6.9 and 6.10. For Figure 6.9, the j^{th} column represents the weights for the j^{th} output channel. We can always arrange the rank weights in columns to visualize them since there is no 2-dimensional information. The i^{th} pixel in a column shows the weights of the rank. The coloring used is RGB since these are the channel inputs.

It isn't straightforward to conclude from these figures. One possible pattern may that when the coder and the next decoder weights have similar correlations.

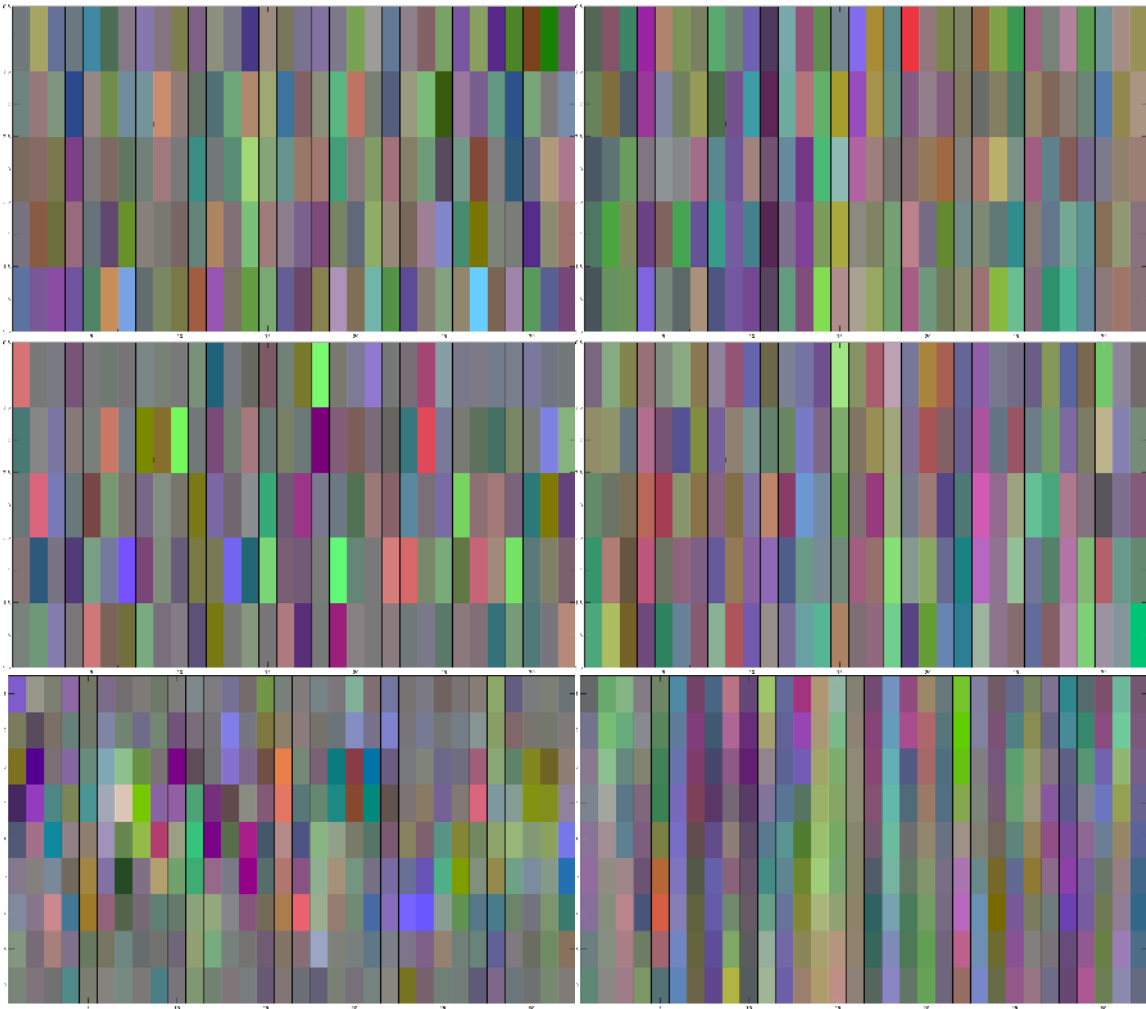


Figure 6.9: Visualization of first block rank coder weights. The left column are the coder weights and on the right we have the decoder weights. From top to bottom you have the first, second and third coder-encoder pair of the net with 4 rank codecs. Notice how the weights of the third codec (with FoV= 3×3) are highly correlated. Every column corresponds to an output channel, and the colors correspond to the RGB channels of the input.

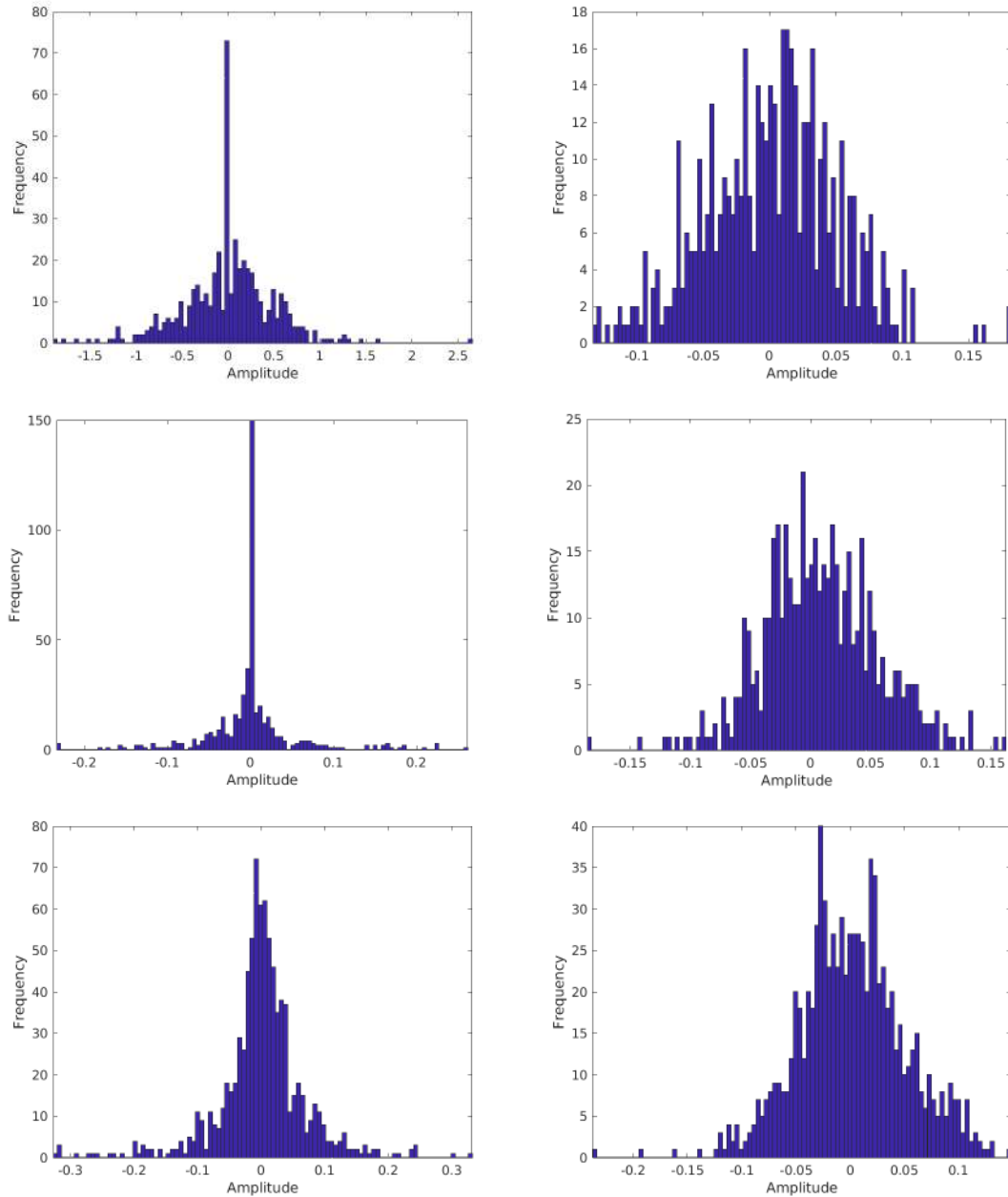


Figure 6.10: Histogram of rank codec weights. The left column is the coder weights histogram, and on the right, we have the decoder weights histogram. From top to bottom rows, you have the first, second, and third coder-encoder pair.

6.3 Classification Experiments

6.3.1 Experiment 6: Comparing classification capacity of rank weights versus linear weights while keeping number of parameters constant.

Here we try to answer the question if the rank parameters are worth it for classification.

6.3.1.1 Methodology

A 5 layer net was trained on the MNIST database. The architecture used a block that had a skip connection over two feed-forward filtering block. The ReLU activation is applied directly after a skip connection junction. Each architecture used five blocks for a total of 10 feed-forward filtering blocks. Every filtering block uses ReLU and BN. A fully Connected (FC) layer with Dropout, ReLU, and BN topped with a final FC for input to the log softmax layer. The first FC layer has 128 outputs⁶. The details for the filtering layers are in table 6.11. To divide the Filtering capacity between Linear filtering and Rank filtering we used the

Filtering layer parameters					
Filtering layer number	FoV	Stride	Pad	Ch_{in}	Ch_{out}
1	2	1	1	3	16
2	2	1	1	16	32
3	3	1	2	32	32
4	3	2	2	32	32
5	2	1	1	32	32
6	2	1	1	32	32
7	3	1	2	32	64
8	3	2	2	64	64
9	2	1	1	64	128
10	2	1	1	128	128

Table 6.11: Experiment 6 filtering parameters. First column indicates the first filtering block to the last filtering blocks. There are a total of 10 feed forward filtering blocks since we have 5 blocks with a residual connection over two blocks.

layer in 4.3 but each branch yields different output channels as in Figure 6.11. For example, if the proportion of rank filtering is one quarter out of 16 output channels, then $q = 1/4$ and the channel outputs are 4 and 12 for the rank and linear filtering blocks.

⁶ We chose this small number simply to make the net execute faster.

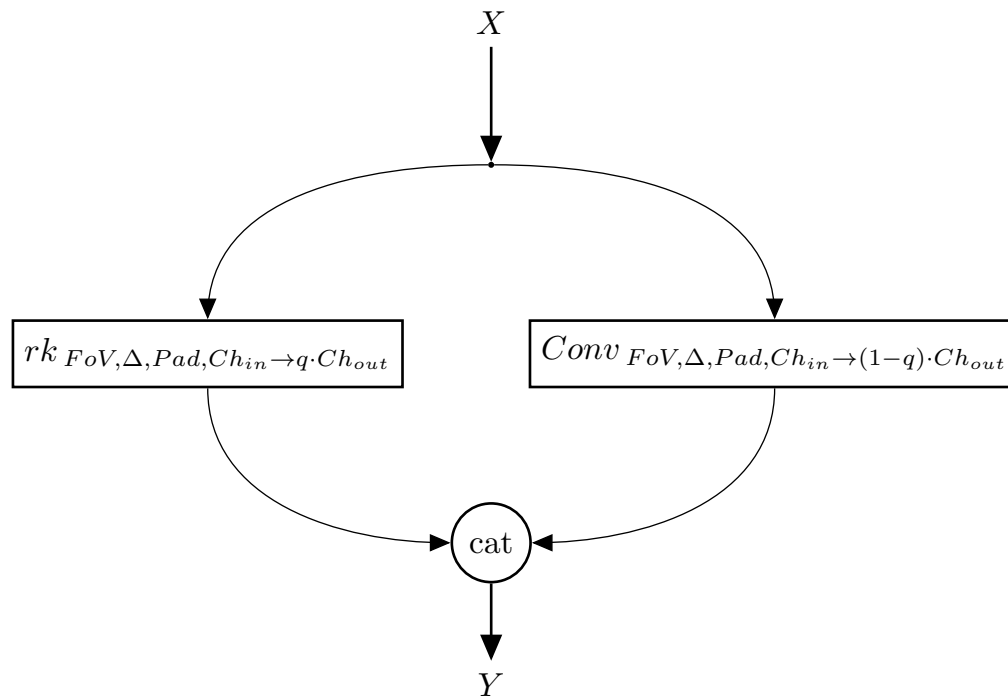


Figure 6.11: The Channel output parameters for both the rank filtering and the convolutional filtering are different. The junction is a concatenation in the channel dimension. The rank filter outputs $q \cdot Ch_{out}$ channels and the linear filter outputs $(1 - q) \cdot Ch_{out}$ channels. After the concatenation, we effectively have Ch_{out} dimensions.

6.3.1.2 Results & Discussion

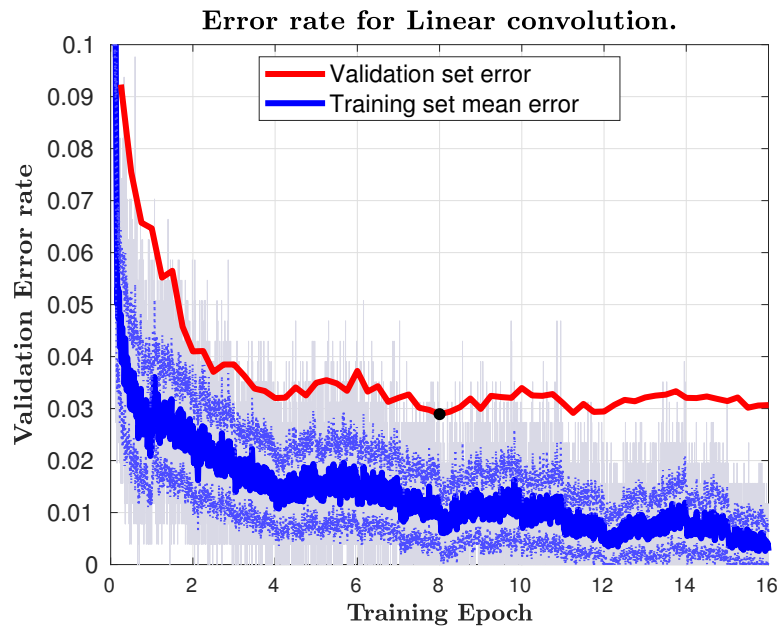


Figure 6.12: Error rate using all linear convolutional net. The minima is attained at epoch 8 for an error of 2.89%.

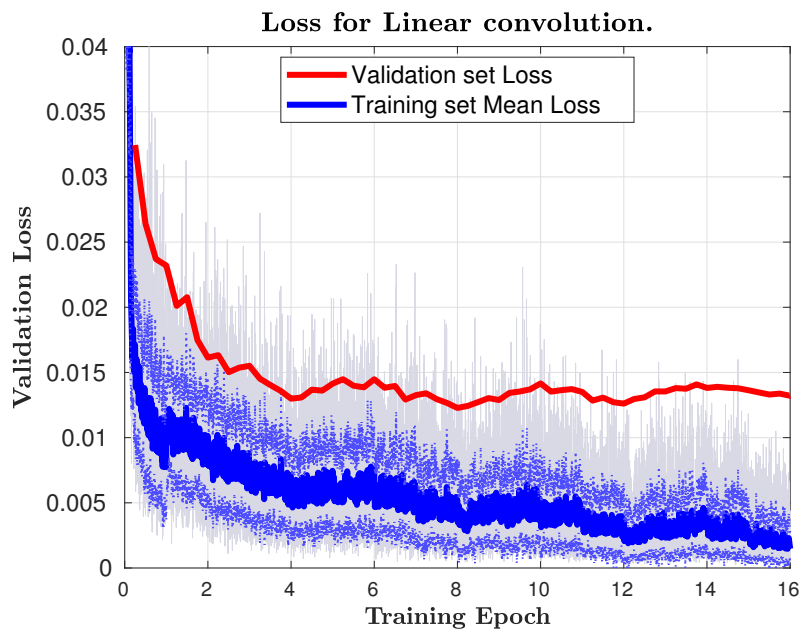


Figure 6.13: As expected no, because rank weights are highly correlated we should not expect a great improvement. This is certainly not the case in autoencoders. There the permutation information is saved and used at the decoder.

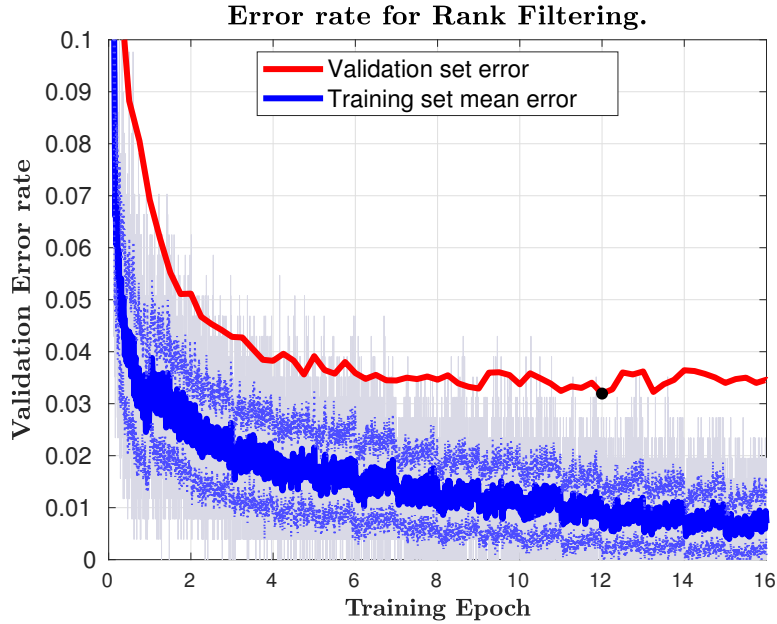


Figure 6.14: Error rate using one fourth capacity (or number of parameters for filtering) reserved for rank filtering. The minima is attained at epoch 12 for an error of 3.19% at the black dot.

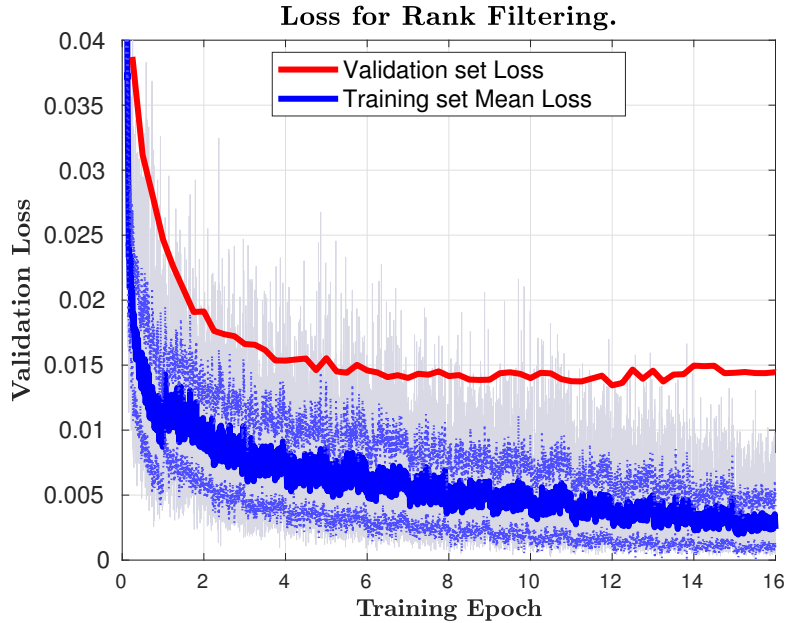


Figure 6.15: Loss using 1/4 capacity for rank filtering.

From the results, we can see that the all-linear filtering yielded better results than the rank filtering one. Additionally, we performed the $\epsilon \times \text{sign}(\frac{\partial \mathcal{L}}{\partial X})$ method recursively once the nets were trained to harness adversarial examples. We observed that rank filtering does yield

adversarial examples that look like noise for a human viewer(not shown here). Qualitatively, the adversarial examples weren't that different from the ones generated by the all-linear net⁷.

6.3.2 Experiment 7: Deterministic and Stochastic Joint Linear-Rank.

In this experiment, we train networks on noiseless data from the SVHN database. We wish to investigate the effect of the deterministic and stochastic joint linear-rank filter compared to the baseline linear network for classification. We use a mini-batch size of 128. In the following table we describe the experiments architectures: We will test the general architecture of the

Layer general parameters.						
Layer	Layer type	Ch_{in}	Ch_{out}	FoV	Stride (Δ)	Padding
1	Filter Block	3	16	(2×2)	1	0
2	Filter Block	16	16	(2×2)	1	0
3	Filter Block	16	32	(3×3)	$\downarrow 2$	0
4	FC	32	10			
5	Log Softmax	10	10			
6	Cross-Entropy	10	10			

Table 6.12: General architecture form for the joint linear-rank experiment. A block's filter can be either linear, joint linear-rank, or a stochastic joint-linear layer. Each block consists of a filter followed by BN and ReLU. The Fully Connected layer is linear.

network of table 6.12 by setting the Block filter to be either linear, deterministic, or stochastic joint linear-rank. We will also test different initializations and if we use a "softmaxed" weights for the prediction step and the random version as in eq.4.40 providing a total of 8 different networks as described in the following table. Since, the symmetry is broken with the gradient update in eq.4.37 and eq.4.38, we are allowed to initialize the weights to a mean of 1 with $\sigma = 0$. We add the last net in the table where we use the stochastic algorithm given that the weight seed \mathbf{v} has $\sigma = 0$ and mean 1.

Since initializing the random weights uniquely for each batch element is too expensive, we set the weights to be identical across batches. This constraint is highly problematic, and we expect the results to be harmful since the variance would not be scaled by $1/(KN)$, but only by $1/(K)$. Hence, we are forced to use a high K in order to reduce the deleterious effects, however to chose a high K is pointless using the Gumbel trick since (when using a large K) it would be the same as using Gaussian sampling.

⁷That's why we didn't include the figures. They convey very little information since they look to produce the same visual gibberish.

⁸The weights is initialized as described in Sec.6.1.4. In other words, zero-mean orthogonal weights with standard deviation $\sigma = \sqrt{2/d_{in}}$.

⁹The weights is initialized by sampling from a uniform distribution where $\mathbf{v} = -\ln(\mathbf{u})$; $\mathbf{u} \sim \mathcal{U}$.

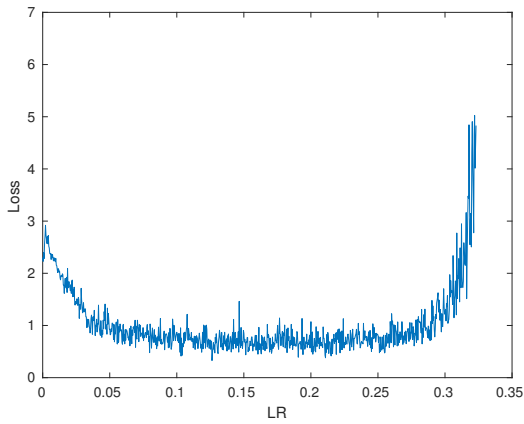
Weight initialization and sampling or activation used while training and testing per network.

Net Id	Block filter	Linear filter		Rank filter	
		Weights	init.	Weights	init.
1	Lin.	\mathbf{w}^{lin}	\mathbf{w}^8	N/A	
2	Lin.-R.	\mathbf{w}^{lin}	\mathbf{w}^8	\mathbf{w}^{rnk}	\mathbf{w}^8
3	Lin.-R.	$\mathbf{w}_{fwd}^{lin} = \tilde{\mathcal{S}}_K^s(\mathbf{v}^{lin})$ $\mathbf{w}_{pred}^{lin} = \mathcal{S}^s(\mathbf{v}^{lin})$	\mathbf{v}^9	\mathbf{w}^{rnk}	\mathbf{w}^8
4	Lin.-R.	\mathbf{w}^{lin}	\mathbf{w}^8	$\mathbf{w}_{fwd}^{rnk} = \tilde{\mathcal{S}}_K^s(\mathbf{v}^{rnk})$ $\mathbf{w}_{pred}^{rnk} = \mathcal{S}^s(\mathbf{v}^{rnk})$	\mathbf{v}^9
5	Lin.-R.	$\mathbf{w}_{fwd}^{lin} = \tilde{\mathcal{S}}_K^s(\mathbf{v}^{lin})$ $\mathbf{w}_{pred}^{lin} = \mathcal{S}^s(\mathbf{v}^{lin})$	\mathbf{v}^9	$\mathbf{w}_{fwd}^{rnk} = \tilde{\mathcal{S}}_K^s(\mathbf{v}^{rnk})$ $\mathbf{w}_{pred}^{rnk} = \mathcal{S}^s(\mathbf{v}^{rnk})$	\mathbf{v}^9
6	Lin.-R.	\mathbf{w}^{lin}	\mathbf{w}^8	$\mathbf{w}^{rnk} = 1$	
7	Lin.-R.	$\mathbf{w}^{lin} = 1$		\mathbf{w}^{rnk}	\mathbf{w}^8
8	Lin.-R.	$\mathbf{w}^{lin} = 1$		$\mathbf{w}^{rnk} = 1$	
9	Lin.-R.	\mathbf{w}^{lin}		$\mathbf{w}_{fwd}^{rnk} = \tilde{\mathcal{S}}_K^s(\mathbf{v}^{rnk})$ $\mathbf{w}_{pred}^{rnk} = \mathcal{S}^s(\mathbf{v}^{rnk})$	$\mathbf{v} = 1$

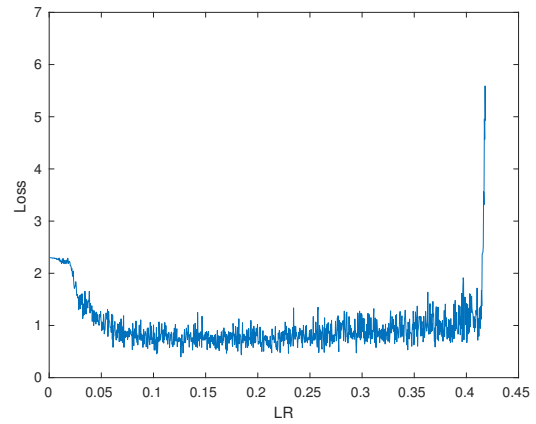
Table 6.13: Architectures tested for joint linear-rank experiment. For all experiments we chose $K = Area$.

6.3.2.1 CLR tests

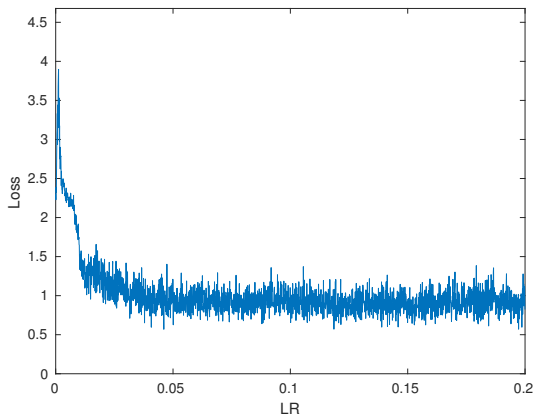
We conduct the CLR test in the following figures:



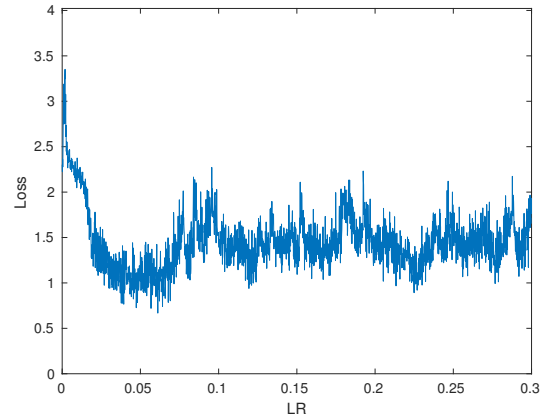
(a) Net 1 (baseline) CLR test



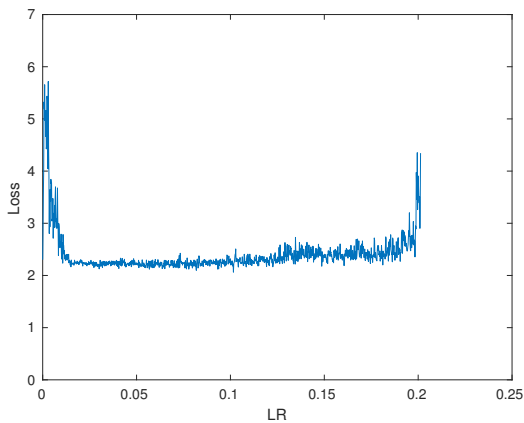
(b) Net 2 CLR test



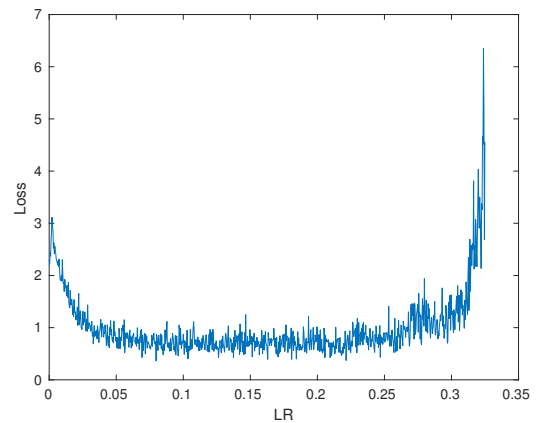
(c) Net 3 CLR test



(d) Net 4 CLR test

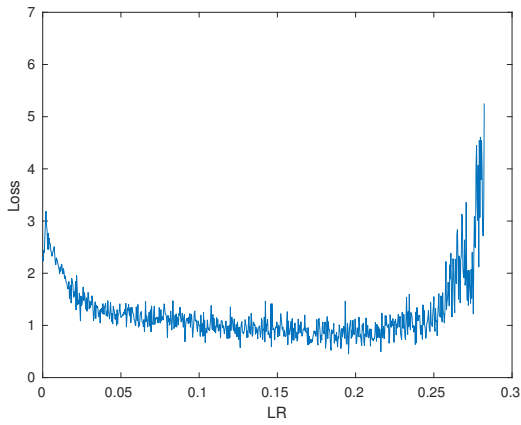


(e) Net 5 CLR test

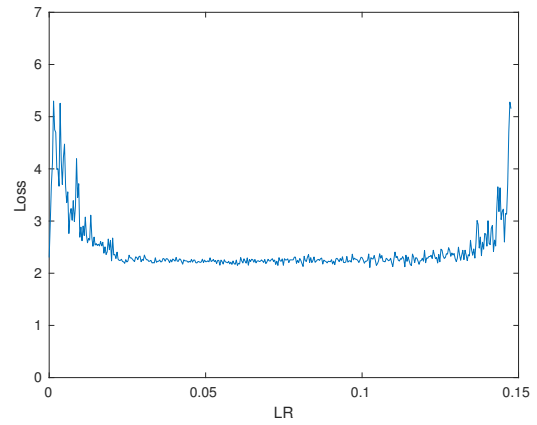


(f) Net 6 CLR test

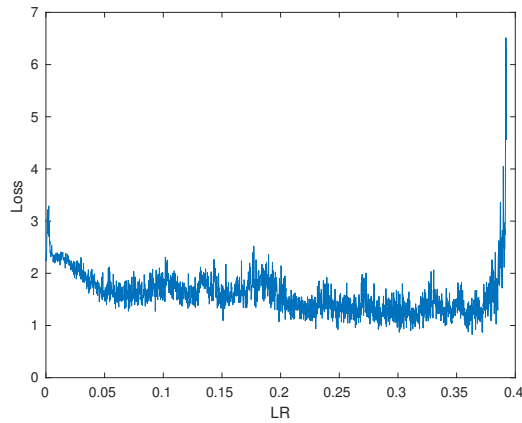
Figure 6.16: Joint Linear-Rank CLR tests (part 1).



(a) Net 7 CLR test



(b) Net 8 CLR test



(c) Net 9 CLR test

Figure 6.17: Joint Linear-Rank CLR tests (part 2).

6.3.2.2 Results & Discussion

In Figures 6.19 and 6.18, we see that the best testing errors, are achieved in 6.18f and 6.18a. The best results are therefore obtained by the baseline all linear net and the joint linear-rank net with all rank weights initialized to 1. The net 1 obtained an error rate of 13.3% and Net 2 obtained an error rate of 13.8%.

We will attempt to explain this observation. The expected value of the gradient wrt the weights are:

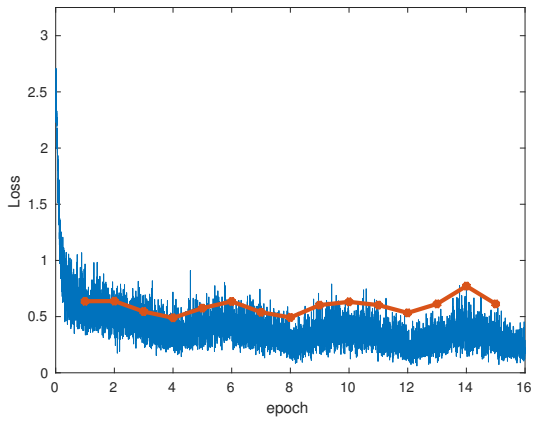
$$\mathcal{E}_x [\nabla_{\mathbf{w}_i} y] = \mathcal{E}_x [P(\mathbf{s} \circ \mathbf{w}_r)] = \mathcal{E}_x [\mathbf{x} \circ P\mathbf{w}_r] = \mathcal{E}_x [\mathbf{x}] \circ (\mathcal{E}_x [P] \mathbf{w}_r) = \bar{\mathbf{x}} \cdot \frac{\mathbf{1}^T \mathbf{w}_r}{n}. \quad (6.2)$$

$$\mathcal{E}_x [\nabla_{\mathbf{w}_r} y] = \mathcal{E}_x [P^T(\mathbf{x} \circ \mathbf{w}_l)] = \mathcal{E}_x [\mathbf{s}] \circ (\mathcal{E}_x [P^T] \mathbf{w}_l) = \bar{\mathbf{s}} \cdot \frac{\mathbf{1}^T \mathbf{w}_l}{n}. \quad (6.3)$$

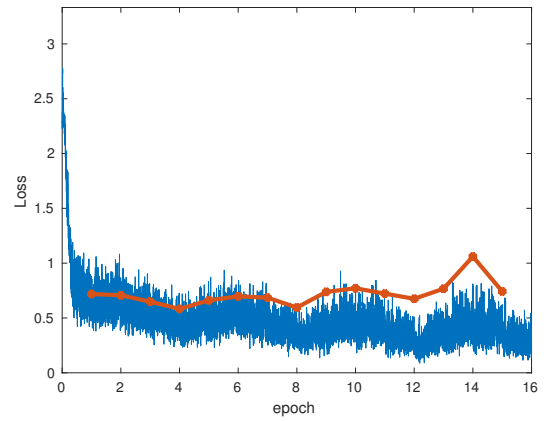
Equations 6.2 and 6.3 imply that we need to be careful in the initialization. We use a scaled version orthogonal weight initialization [35]. This initialization is unit variance and zero-mean. This implies that both Equations 6.2 and 6.3 will tend to 0, effectively slowing the weight update. To prevent this and to ensure quick learning we would need to make either the rank weights or the linear weights have mean of 1. But since the net works best with linear weights we are left with only setting the rank weights to 1 (net 6). In the end, the training rank weights mean of net 6 for layers 1 to 3 are $\mathcal{E}[\mathbf{w}_r] = (1.0369, 0.9977, 0.9988)$, and the weight standard deviations for layers 1 to 3 are $\sigma_{\mathbf{w}_r} = (0.1716, 0.0842, 0.0547)$. This result may be empirical proof that we are better not to use rank filters in many layers or at all.

Furthermore, all stochastic experiments yielded terrible results. We quickly trained a net where K was large, and the results improved. But we will not elaborate more on this since to truly test the potential of the stochastic algorithms, we would need to implement it also in the FC layer, and we would also need to sample the weights for each batch element uniquely. Testing the idea on the FC makes sense since, by analogy, Dropout performs better when placed after FC layers. Perhaps, we could see a similar result? But this is hardware intensive, and hence we will not pursue more experiments.

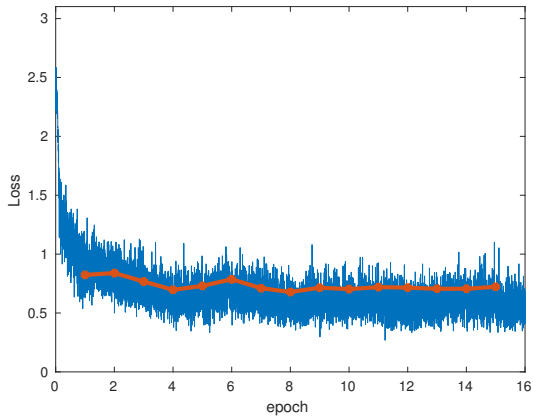
From figures 6.18e and 6.19b, we see that it is a bad idea to set both linear weights and rank weights to an average of 1 since it effectively stalls learning.



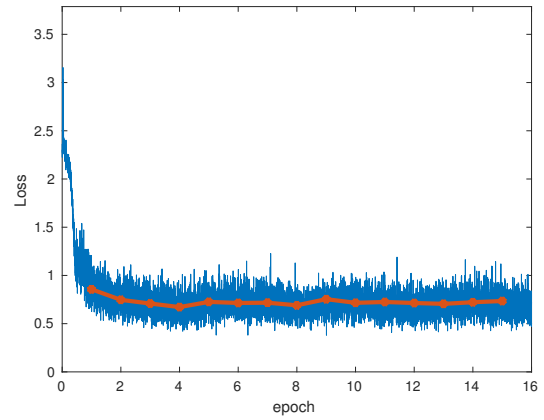
(a) Net 1 (baseline) linear conv. loss.



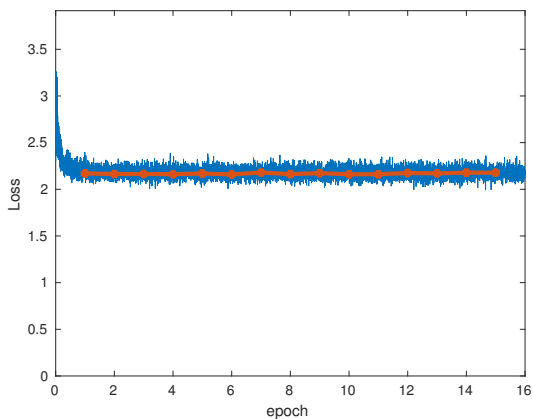
(b) Net 2 loss.



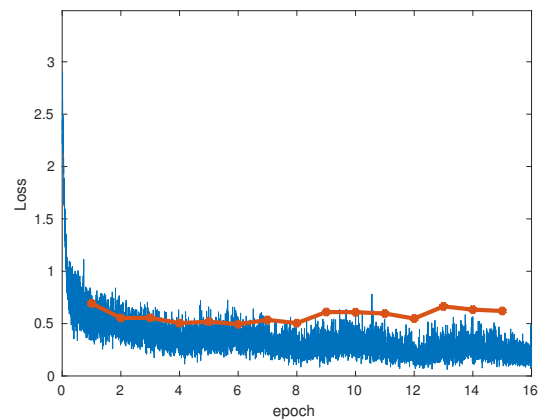
(c) Net 3 loss.



(d) Net 4 loss.

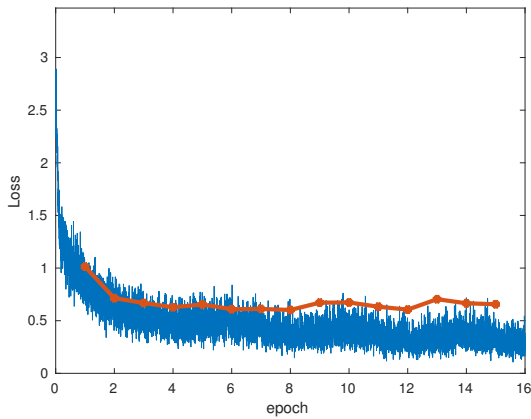


(e) Net 5 loss.

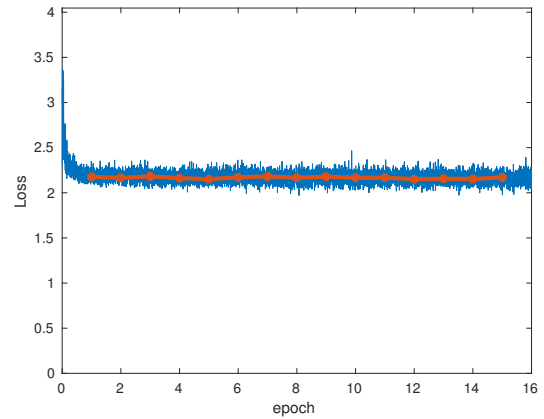


(f) Net 6 loss.

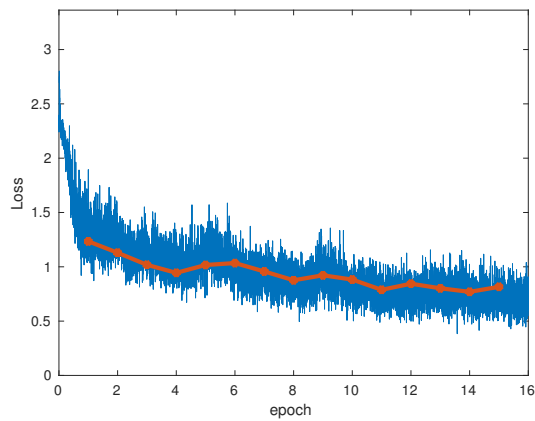
Figure 6.18: Joint Linear-Rank training and test loss (part 1). The red curves are the test set loss, and the training loss is in blue.



(a) Net 7 loss.



(b) Net 8 loss.



(c) Net 9 loss.

Figure 6.19: Joint Linear-Rank training and test loss (part 2). The red curves are the test set loss, and the training loss is in blue.

Chapter 7

Discussion

We've answered the questions on page [22](#):

1. In classification, is a rank parameter worth the same as a linear parameter?
 - In general, no rank filters don't seem to be particularly advantageous for deep or shallow layers.
 2. When is it advantageous of using rank filters in classification ?
 - When we use shallow neural nets that have impulse noise corruption on the data. In experiment 3, we saw that it is better to use a rank filter at the beginning of a network.
 3. What are the advantages and disadvantages of using rank filters in autoencoders ?
 - The advantage is that under impulse noise, rank filters really do seem to be more robust than linear filters. However, it is not spectacularly more robust and in fact deep linear layers perform systematically better than rank neural nets when the test data has the same corruption type when training both nets, as seen in [Figure 6.8](#).
 - It is definitely better to use rank filters when we use shallow neural nets. But, this result isn't too interesting since the advantage seems to be lost quickly when we use deep nets.
 4. How remembering all permutations affect a decoder's reconstruction "clearness"?
 - As seen in [Figure 6.6i](#) and [Tables 6.7, 6.7 and 6.9](#), remembering permutations do seem to be helpful only when the rank transpose filter is immediately after the rank filter. The advantage is more evident when we use strides that aren't too large. When it is used in a shallow autoencoder, from [Figure 6.3j](#), we see that we are better to not use transposed rank permutations.
 - (a) If we simply ignore the permutations how will it affect the reconstruction quality?
 - It depends...
-

- In the Tables 6.7, 6.7 and 6.9 from experiment 4 and the best result from experiment 1, when transposed rank filters are used in shallow autoencoders, we saw that the performance depends on the FoV and stride of the encoder rank filters. Therefore, sometimes it is better to use transposed rank filters, and sometimes not. In experiment 4, we've only seen what was the effect of FoV variations within the context of corrupted images. Hence, if there is no noise, the reconstruction error might have been less punitive for low FoV's. Additionally, in presence of corruption noise, results show that using large FoVs of 4 yields worse results than an FoV of 3, but when the FoV was smaller than 3, we obtained worse performance. The degradation of performance for large FoVs seem to support 2.
 - It would be better to ignore altogether the permutations if the decoder is more than 1 layers apart from it's corresponding coder layer as seen in experiment 2.
 - In the case where we used many small codecs, we saw better visual reconstruction results in experiment 3.
5. If we use shallow or deep decoders that use the reverse-sorting process, would it still be able to preserve thin structures?
- Thin structures are preserved by the reverse-sorting process. Transposed rank filters seem to work very well if used within a block that has a coder-decoder structure (see experiments 1 to 4). They don't work well with deep decoders. Moreover, when we use an autoencoder with 4 or more layers, using a rank codec only at the beginning seems to be beneficial.
- (a) If so, would it be able to discern which thin structures are spurious?
- Yes, but it does seem to introduce a lot more smoothing than regular deconvolution. Although this is probably due to the loss of information in the coder (since ranks are highly correlated) and not the decoder itself.
- (b) Does thin structures reconstruction better or worse with large versus small FoV in shallow decoders?
- In experiment 4, we saw that an Fov of (3, 3) performed best when we used a stride of 1. It was even better than the performance for the net with FoV of (4, 4) with stride 1. So a larger FoV isn't better, and too small isn't better either. The degradation of performance for large FoVs seem to support Hypothesis 2.
- (c) Does thin structures reconstruct better or worse with large versus small FoV in deep decoders?
- It really depends on the architecture of the deep net and the noise level. But from the results in experiment 1, we saw that it was better to avoid deep decoders when the corresponding encoder had rank filters.
6. What forms do the rank filters and transposed filters take?
-

- From Figure 6.9, we see that both the decoder and encoder exhibit high correlation in their weights.
7. Do transposed rank filters generate artefacts analogous to transposed convolutions?
 - If transposed rank filters are used without any Overlap compensation as described in 4.4.1, then we get artefacts as shown in Figure 4.11. However, by using the compensation we avoid artefacts completely. Directly using scale compensation may not be the only solution since the strategy used in [45] was to smoothing the “artifactory” output of the transposed filter.
 8. Is a stochastic version of rank filters combined with linear filters beneficial?
 - This hasn’t been fully resolved as to test this idea we would need to randomly sample different weights for every batch, which requires an efficient algorithm on the GPU. Moreover, we tested a low sparsity and identical random weights across the batch dimension to observe that in Figures 6.18c, 6.18d, 6.18e and 6.19c overall worse testing loss compared to the baseline linear convolutional net (see 6.18a).

Furthermore, we’ve gathered evidence from experiment 1 that seem to confirm Hypothesis 3, that postulated that transposed rank filters performed better as shallow autoencoders. More evidence extends this hypothesis to shallow encoder-decoder building blocks in later experiments.

In experiment 5, we attempted to use Cross-Channel Rank filters in order to mitigate the loss of information inherent to rank filters, and maximise the resolution of the neural net. However, we found that the baseline all-linear net performed better for image reconstruction.

In experiment 6 we found that rank weights do not encode as much information as linear weights.

Chapter 8

Conclusion & Outlook

We've built novel layers based on ranks and framework to describe how they fit within the family of commonly used layers. We analyzed its mathematical properties. By symmetry, we've designed transposed non-linear filters to mimic the inverse operation and analyzed and proposed a solution to the common artifact problem that transposed filters suffer from and applied it to our rank transposed filter.

We have analyzed the rank filters as a non-linear layer, and we've derived the variance of the optimal weights during the initialization step, given that we do not use any other activation. In experiment 1, we saw that it was best to use an activation even though the sorting process is itself non-linear.

We've explored when we should use them and what architectures our new layers best perform in tasks such as image restoration and classification. We found that rank filters work best within a block that has a shallow encoder and decoder. Transposed rank filters are beneficial, particularly if they are immediately placed after it's corresponding coder layer. The transposed filters were also shown to perform best in low FoVs. Max-Pooling layers are rank filters with stride equal to their FoV. If we were to permit max-pooling layers to have a lower downsampling rate¹, then it might be possible to have better performance. We also saw that rank filters or shallow rank encoder-decoder blocks performed better when they were only present at the beginning of a neural network. Deeper into the layer, it is best to avoid rank filters. Pooling layers lose delicate structure information for large FoVs because they are sparse rank filters, we saw in our experiments that rank filters and the transposed ones could preserve fine structures. Still, they do tend to induce a lot of diffusion in image restoration.

Stochastic algorithms

We haven't explored the full potential of stochastic rank filters, but the mixed results obtained may suggest that we might be better to use the classical convolution layer. **Moreover, the sampling technique that was developed in designing the sampling method for the stochastic algorithm may be used for linear filters. This hasn't been investigated thoroughly,** and the main obstacle is that the algorithm would need to

¹ Although this would destroy the purpose of max-pooling, which is to downsample by keeping the highest activation.

generate different random weights for each batch element. The design of the algorithm itself is pretty straightforward since the sampling method is based on the Gumbel max-trick, which is based on a maximal index of a Gumbel distribution. There exist excellent parallel algorithms to get maximal index, and thus it shouldn't be too hard to implement the sampling trick on a GPU (provided that the GPU hardware manages to process random weights that have an extra batch dimension quickly).

In designing a method to generate sparse random weights for training, we found that the ℓ_1 norm appeared naturally in eq. 4.55. It is interesting to note that the ℓ_1 norm also appeared in another sparsity context, such as LASSO. Perhaps there is an interesting probabilistic explanation for this similarity. There might be a link.

Lessons to be Retained for Future Design of Neural Networks

In Capsule networks [33], there is a max-operation in order to make the capsules agree for a probability. The features learned by Capsules were shown to contain dilations [33]. Max-operations are a type of rank filter that generates high dilation features. Hence we may think that capsule nets will inherently have dilation features. In this thesis, we saw that rank features fare almost always worse than linear features. Both the max-pooling layer and the capsule network architecture do not use activations that aren't maximal. Thus we may need to reconsider the capsule architecture and see what the effect of removing the routing by agreement method is. Surely better performance will be achieved if we used more information e.g. if we also used the second-highest probability capsule and so on. But as shown in the thesis, all-linear nets generally performed better in very deep neural networks. We could make an analogous prediction that the overall performance would be much better for very deep capsule nets if we simply abandon altogether the routing procedure based on the maxima (abandon all ranking procedures) and use all-linear capsule nets. Indeed, in [12], the authors show that it is not necessary to implement routing procedures. Essentially, they show that the benefits of Capsule Networks lie in the abstraction of a general convolution in yet another dimension and not on the routing. This is analogous to why the authors of the paper [43] show that it's better to use only convolutional layers and downsample using strided convolutions (or average pooling) instead of using max-pooling layers.

Outlook on Other Non-Linear Filters

We've briefly explored histogram filters in Section 5. Histogram filters can be smooth or not, and they are akin to rank filters in the sense that they can be modelled as a non-linear filter. We've derived the gradients of the Histogram filters in order to permit back-propagation through the filters. We've also related the Histogram filters and interpreted the following

techniques in the context of neural networks HOG², Histogram equalization³ and amplitude banks⁴. Naturally, neural nets that use amplitude banks are directly related to histograms, and thus they are algebraically linked to a non-linear operation. Analysis of the gradients permits the use of HOG features in neural networks. There is so much work done in the literature that belongs to this wide⁵ class of filters that it is hard to believe that we can't find a better design that would greatly contribute to the emergent field of deep learning.

² We've shown that HOG features have striking resemblances with the now commonly used blocks that comprise of linear→activation→normalization.

³ We've shown that Histogram equalization has an autoencoder structure with a non-linear filter→normalization→transposed non-linear filter.

⁴ As demonstrated in the thesis, CReLU and ReLU are amplitude banks.

⁵ I've avoided to work on this topic because the idea is too general. Programming on a GPU is difficult and takes a lot of time. The speed of the algorithm depends on the code design and hardware. Complicated Histogram filters may not be the best choice for today's technological limitations in hardware.

References

- [1] Adaptive rank filters and transposed rank filters in deep learning. <https://www.youtube.com/watch?v=pCONU8Rxqto/>. Accessed: 2019-12-06.
- [2] Deconvolution and Checkerboard Artifacts. <https://distill.pub/2016/deconv-checkerboard/>. Accessed: 2019-06-21.
- [3] Intro to optimization in deep learning: Busting the myth about batch normalization. <https://blog.paperspace.com/busting-the-myths-about-batch-normalization/>. Accessed: 2019-06-27.
- [4] Andrew Aitken, Christian Ledig, Lucas Theis, Jose Caballero, Zehan Wang, and Wenzhe Shi. Checkerboard artifact free sub-pixel convolution: A note on sub-pixel convolution, resize convolution and convolution resize. *arXiv preprint arXiv:1707.02937*, 2017.
- [5] Kazuhiko Aomoto. Analytic structure of schläfli function. *Nagoya Mathematical Journal*, 68:1–16, 1977.
- [6] Vijay Badrinarayanan, Ankur Handa, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for robust semantic pixel-wise labelling. *arXiv preprint arXiv:1505.07293*, 2015.
- [7] BG Batchelor. Streak and spot detection in digital pictures. *Electronics Letters*, 15(12):352–353, 1979.
- [8] Elwyn R Berlekamp, John H Conway, and Richard K Guy. Winning ways for your mathematical plays.
- [9] Y-Lan Boureau, Jean Ponce, and Yann LeCun. A theoretical analysis of feature pooling in visual recognition. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 111–118, 2010.
- [10] Raymond H Chan, Chung-Wa Ho, and Mila Nikolova. Salt-and-pepper noise removal by median-type noise detectors and detail-preserving regularization. *IEEE Transactions on image processing*, 14(10):1479–1485, 2005.
- [11] Liang Chen, Paul Bentley, and Daniel Rueckert. Fully automatic acute ischemic lesion segmentation in dwi using convolutional neural networks. *NeuroImage: Clinical*, 15, 06 2017.

-
- [12] Zhenhua Chen, Xiwen Li, Chuhua Wang, and David Crandall. Capsule networks without routing procedures. 2019.
- [13] MC Cheng. The orthant probabilities of four gaussian variates. *The Annals of Mathematical Statistics*, 40(1):152–161, 1969.
- [14] Youngmin Cho and Lawrence K Saul. Kernel methods for deep learning. In *Advances in neural information processing systems*, pages 342–350, 2009.
- [15] Benjamin Graham. Fractional max-pooling. *arXiv preprint arXiv:1412.6071*, 2014.
- [16] Frank R Hampel, Elvezio M Ronchetti, Peter J Rousseeuw, and Werner A Stahel. *Robust statistics: the approach based on influence functions*, volume 114. John Wiley & Sons, 2011.
- [17] Soufiane Hayou, Arnaud Doucet, and Judith Rousseau. On the impact of the activation function on deep neural networks training. *arXiv preprint arXiv:1902.06853*, 2019.
- [18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1026–1034, 2015.
- [19] Robert M Hodgson, Donald G Bailey, MJ Naylor, ALM Ng, and SJ McNeill. Properties, implementations and applications of rank filters. *Image and Vision Computing*, 3(1):3–14, 1985.
- [20] Michael Kass and Justin Solomon. Smoothed local histogram filters. *ACM Transactions on Graphics (TOG)*, 29(4):100, 2010.
- [21] Steven G Krantz. *The proof is in the pudding: The changing nature of mathematical proof*. Springer Science & Business Media, 2011.
- [22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- [23] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.
- [24] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.
- [25] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.
- [26] Jaehoon Lee, Yasaman Bahri, Roman Novak, Samuel S Schoenholz, Jeffrey Pennington, and Jascha Sohl-Dickstein. Deep neural networks as gaussian processes. *arXiv preprint arXiv:1711.00165*, 2017.

-
- [27] Chris J Maddison, Daniel Tarlow, and Tom Minka. A* sampling. In *Advances in Neural Information Processing Systems*, pages 3086–3094, 2014.
- [28] Xiao-Jiao Mao, Chunhua Shen, and Yu-Bin Yang. Image restoration using convolutional auto-encoders with symmetric skip connections. *arXiv preprint arXiv:1606.08921*, 2016.
- [29] Frederick Mosteller. On some useful inefficient statistics. In *Selected Papers of Frederick Mosteller*, pages 69–100. Springer, 2006.
- [30] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. Learning deconvolution network for semantic segmentation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1520–1528, 2015.
- [31] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548*, 2018.
- [32] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.
- [33] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. Dynamic routing between capsules. In *Advances in neural information processing systems*, pages 3856–3866, 2017.
- [34] Kenzi Satô. Spherical simplices and their polars. *Quarterly journal of mathematics*, 58(1):107–126, 2007.
- [35] Andrew M Saxe, James L McClelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*, 2013.
- [36] L Schläfli. On the multiple integral whose limits are $p_1 = a_1x + b_1y + \dots + h_1z > 0, \dots, p_n > 0, x^2 + y^2 + \dots + z^2 < 1$. *Quart. J. Math. Pure Appl.*, 2:261–301, 1858.
- [37] Wenling Shang, Kihyuk Sohn, Diogo Almeida, and Honglak Lee. Understanding and improving convolutional neural networks via concatenated rectified linear units. In *international conference on machine learning*, pages 2217–2225, 2016.
- [38] Zenglin Shi, Yangdong Ye, and Yunpeng Wu. Rank-based pooling for deep convolutional neural networks. *Neural Networks*, 83:21–31, 2016.
- [39] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034*, 2013.
- [40] Leslie N Smith. No more pesky learning rate guessing games. *CoRR*, abs/1506.01186, 2015.
- [41] Leslie N Smith. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472. IEEE, 2017.
-

- [42] Leslie N Smith and Nicholay Topin. Super-convergence: Very fast training of residual networks using large learning rates. 2018.
- [43] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.
- [44] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.
- [45] Yusuke Sugawara, Sayaka Shiota, and Hitoshi Kiya. Convolutional neural networks without any checkerboard artifacts. In *2018 26th European Signal Processing Conference (EUSIPCO)*, pages 1317–1321. IEEE, 2018.
- [46] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.
- [47] Jonathan Tompson, Ross Goroshin, Arjun Jain, Yann LeCun, and Christoph Bregler. Efficient object localization using convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 648–656, 2015.
- [48] Rand R Wilcox. *Introduction to robust estimation and hypothesis testing*. Academic Press, 2005.
- [49] Matthew D Zeiler and Rob Fergus. Stochastic pooling for regularization of deep convolutional neural networks. *arXiv preprint arXiv:1301.3557*, 2013.
- [50] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

List of Figures

2.1	Convolution connection graph.	7
2.2	Rank filter Grayscale feature map.	8
2.3	Rank skip connection	9
2.4	Rank double skip connection	9
2.5	Venn diagram showing which layer can be reached by subspaces of rank filters and convolutional layers.	12
2.6	Max-Pooling with downsampling of 2 Viewed under rank filtering perspective.	13
2.7	Average-Pooling with downsampling of 2 Viewed under rank filtering perspective.	13
2.8	Average-Pooling with downsampling of 2 viewed as a fixed Convolutional layer of stride 2.	14
2.9	Scrambled face.	15
2.10	Unpooling layers	17
2.11	Venn diagram showing how Max-Unpooling falls in transposed rank filters.	18
2.12	Max-Pooling & Max-Unpooling layers with downsampling of 2 Viewed under rank & transposed rank filtering perspective.	18
3.1	Permutations saved in coder for the decoder	21
4.1	Rank filter connection graph.	24
4.2	Rank Filter example	25
4.3	Input example	26
4.4	Augmentation example with FoV=5x5 with stride=1	26
4.5	Augmentation example with FoV=5x5 with stride=2	27
4.6	Augmentation example with FoV=1x8 with stride=1	28
4.7	Venn diagram showing various subspaces reach.	34
4.8	Joint Linear-Rank Filter graph.	34
4.9	Rank plus Linear filter.	40
4.10	Deconvolutional Rank Filter example	41
4.11	Shallow denoising rank transpose autoencoder without overlap scaling.	45
4.12	Deconvolutional rank filter numerical example	46
4.13	Transposed Rank plus Linear filter.	48
5.1	HOG pileline analysis.	50
5.2	Histogram Equalization pileline analysis.	52
6.1	Cyclic Learning Rate.	55

6.2	Cyclic Learning Rate.	55
6.3	Denoising net qualitative results for experiments 1.	61
6.4	Denoising net qualitative results (part 1)	67
6.5	Denoising net qualitative results (part 2)	68
6.6	Experiment 3 qualitative results	74
6.7	Networks for experiment 5	78
6.8	Robustness test for experiment 5	80
6.9	Visualization of rank codec weights.	81
6.10	Histogram of rank codec weights.	82
6.11	Rank plus Linear filter.	84
6.12	Error rate for a 5 layer net classifier with linear filtering.	85
6.13	Loss rate for a 5 layer net classifier with linear filtering.	85
6.14	Error rate for a 5 layer net classifier with 25% rank filtering.	86
6.15	Loss rate for a 5 layer net classifier with 25% rank filtering.	86
6.16	Joint Linear-Rank CLR tests (part 1)	89
6.17	Joint Linear-Rank CLR tests (part 2)	90
6.18	Joint Linear-Rank training and test loss (part 1).	92
6.19	Joint Linear-Rank training and test loss (part 2).	93
B.1	Network that yields permutation indices of 4 elements.	120
B.2	Network that yields permutation indices of 6 elements.	121

Appendices

Appendix A

Kernel Derivation for the Rank Filtering Process

Since the integral depends only on the angle between the input vectors \mathbf{a} and \mathbf{b} (see Equation 4.17), without loss of generality, we can pick $M = 2$. The mean

$$\boldsymbol{\mu} = \mathcal{E}_W[\boldsymbol{\phi}(W^T \mathbf{a})]. \quad (\text{A.1})$$

The mean vector is skew-symmetric since when we flip it, we change it's sign:

$$J\boldsymbol{\mu} = -\boldsymbol{\mu}, \quad (\text{A.2})$$

where J is the cross-diagonal matrix. Analogously, the kernel matrix $\mathcal{E}_W[\boldsymbol{\phi}(W^T \mathbf{a}) \boldsymbol{\phi}^T(W^T \mathbf{b})]$ is bisymmetric and thus contains only $\left\lfloor \frac{(N+1)^2}{4} \right\rfloor$ unique functions. The permutation matrix P_x element (i, j) is given by:

$$P_{i,j} = \sum_{k=1: \binom{N}{j}} \prod_{\substack{n=1 \\ n \neq i}}^N u(\sigma_n(x_i - x_n)), \quad (\text{A.3})$$

where $\boldsymbol{\sigma}_{k,j}$ is a unique $N - 1 \times 1$ vector that has $N - i$ (1)'s and $i - 1$ (-1)'s. In other terms,

$$\boldsymbol{\sigma}_k = P^{(k)} \begin{bmatrix} \mathbf{1}_{N-i \times 1} \\ -\mathbf{1}_{i-1 \times 1} \end{bmatrix}; \boldsymbol{\sigma}_{m,j} \neq \boldsymbol{\sigma}_{n,j} \forall (m, n),$$

where $V^{(k)}$ is a permutation matrix. The kernel's derivative at $\theta = 0$ is for all N :

Lemma A.0.1.

$$\left. \partial_\theta K(\mathbf{a}, \mathbf{b}) \right|_{\theta=0} = 0_{N \times N}. \quad (\text{A.4})$$

Proof: Since the correlation between two vectors of the same magnitude is maximal when the vectors are colinear, then the correlation between the sorted vectors is also maximal when the vectors are co-linear. \square

Lemma A.0.2. For any N , the cross-correlation matrix $K(\mathbf{a}, \mathbf{b}) = K(\theta)$ obeys:

$$\frac{K(\theta)}{\|\mathbf{a}\| \|\mathbf{b}\|} = \cos \theta \cdot I_N + \frac{K(0)}{\|\mathbf{a}\| \|\mathbf{b}\|} - I + V(\theta); V_{i,j}(\theta) = \mathcal{O}(\theta^3), \theta > 0. \quad (\text{A.5})$$

is equivalent in stating that

$$\lim_{\theta \rightarrow 0^+} \frac{-1}{\sin \theta} \frac{\partial}{\partial \theta} \frac{K(\mathbf{a}, \mathbf{b})}{\|\mathbf{a}\| \|\mathbf{b}\|} = I. \quad (\text{A.6})$$

Proof: The matrix $V(\theta) = \mathcal{O}(\theta^1)$ is easy to verify since by Equation A.5:

$$\lim_{\theta \rightarrow 0} \frac{K(\theta)}{\|\mathbf{a}\| \|\mathbf{b}\|} = \lim_{\theta \rightarrow 0} \frac{K(0)}{\|\mathbf{a}\| \|\mathbf{b}\|} + V(\theta);$$

and by cancelation we have

$$V(0) = 0.$$

Then, we know that $K(\theta)$ is an even function and so is $V(\theta)$, hence:

$$V(\theta) = |\theta| A(\theta) + B(\theta), \quad (\text{A.7})$$

where $B(0) = 0$ and both $A(\theta)$ and $B(\theta)$ are even. By using A.0.1, we have

$$\dot{V}(0) = 0 = A(0), \quad (\text{A.8})$$

For small θ , and by using A.6

$$\lim_{\theta \rightarrow 0^+} \frac{-1}{\sin \theta} \frac{\partial}{\partial \theta} \frac{K(\mathbf{a}, \mathbf{b})}{\|\mathbf{a}\| \|\mathbf{b}\|} \approx I + \lim_{\theta \rightarrow 0^+} \frac{\dot{V}(\theta)}{\theta} = I. \quad (\text{A.9})$$

Hence, $\lim_{\theta \rightarrow 0^+} \frac{\partial^2}{\partial \theta^2} B(\theta) = 0$ and thus, $V_{i,j}(\theta) = |\theta| \mathcal{O}(\theta^2) + \mathcal{O}(\theta^4) = \mathcal{O}(\theta^3)$. \square

Note that the above hypothesis is supported by experimental data using Monte-Carlo simulations. From Lemma A.0.3, we can deduce that $V(\theta) \mathbf{1} = 0$.

Lemma A.0.3.

$$\frac{K(\mathbf{a}, \mathbf{b})}{\|\mathbf{a}\| \|\mathbf{b}\|} \mathbf{1}_{N \times 1} = \cos \theta \mathbf{1}_{N \times 1} \quad (\text{A.10})$$

This means that $\mathbf{1}_{N \times 1}$ is in the null space of the matrix terms (except for the identity) that compose $K(\mathbf{a}, \mathbf{b})$. In other words,

$$\left(\frac{K(\mathbf{a}, \mathbf{b})}{\|\mathbf{a}\| \|\mathbf{b}\|} - \cos \theta I \right) \mathbf{1}_{N \times 1} = \mathbf{0}, \quad (\text{A.11})$$

independently from θ . Equation A.11 shows that $\cos \theta$ and $\mathbf{1}$ is an eigenpair of $\frac{K(\mathbf{a}, \mathbf{b})}{\|\mathbf{a}\| \|\mathbf{b}\|}$.

Proof: To prove Equations A.10 and A.11, we can use $\phi(\mathbf{x}) = P_x^T \mathbf{x}$. Although P_x and $P_{\cos \theta \mathbf{x} + \sin \theta \mathbf{y}}$ are dependent, it won't matter since $P_{\cos \theta \mathbf{x} + \sin \theta \mathbf{y}}$ will vanish from the expectation because $P^T \mathbf{1} = \mathbf{1}$ leaving a single permutation matrix that will permit us to use the law of total expectation. This is justified because the sets are disjoint since $P^T \mathbf{1} = P \mathbf{1} = \mathbf{1}$ because one element is active per row. Hence,

$$\mathcal{E}_{\mathbf{x}, \mathbf{y}} \left(\phi(\mathbf{x}) \phi^T(\cos \theta \mathbf{x} + \sin \theta \mathbf{y}) \mathbf{1} \right) = \mathcal{E}_{\mathbf{x}, \mathbf{y}} \left(P_x^T \mathbf{x} (\cos \theta \mathbf{x} + \sin \theta \mathbf{y})^T P_{\cos \theta \mathbf{x} + \sin \theta \mathbf{y}} \mathbf{1} \right) \quad (\text{A.12})$$

$$= \mathcal{E}_{\mathbf{x}, \mathbf{y}} \left(P_x^T \mathbf{x} (\cos \theta \mathbf{x} + \sin \theta \mathbf{y})^T \mathbf{1} \right) = \cos \theta \mathcal{E}_{\mathbf{x}} \left(P_x^T \mathbf{x} \mathbf{x}^T \mathbf{1} \right) \quad (\text{A.13})$$

$$= \cos \theta \sum_n \mathcal{E}_{\mathbf{x}} \left(x_n \mathbf{x}^T \mathbf{1} | \pi_{n,i} = 1 \right) \Pr(\pi_{n,i} = 1); i \in [1, N] \quad (\text{A.14})$$

$$= \frac{\cos \theta}{N} \sum_n \mathcal{E}_{x_n} \left(x_n^2 \right) \mathbf{1} = \cos \theta \mathbf{1} \quad (\text{A.15})$$

□

This reduces the kernel functions to solved by integration from $\left\lfloor \frac{(N+1)^2}{4} \right\rfloor$ to $\left\lfloor \frac{N^2}{4} \right\rfloor$. For general $N > 2$ and θ , the last (comparing max with max) kernel function to solve has many terms since P_x and $P_{\cos \theta \mathbf{x} + \sin \theta \mathbf{y}}$ are dependent. There is some redundancy that makes it possible to at least reduce the number terms to compute since many of them are linearly dependent. This number c depends on N and varies depending on the kernel function computed. c is at most $\mathcal{O}(N^2)$ and hence the number of integrals to solve is at most $|I|_0 = c \left\lfloor \frac{N^2}{4} \right\rfloor = \mathcal{O}(N^4)$. This is why the Kernel functions are not shown for $N > 2$ and a general θ . Fortunately, when $\theta = 0$, $c = 1$ for the extreme values Kernel.

For the cases $N > 2$, we couldn't find a closed form solution¹. When $\theta = 0$, the integrals become considerably easier to solve, however they do become difficult when $N \geq 5$.

Lemma A.0.4. *Yet another property includes:*

$$\text{tr} \left(\frac{K(\mathbf{a}, \mathbf{a})}{\|\mathbf{a}\|^2} \right) = N. \quad (\text{A.16})$$

Proof:

$$\text{tr} \left(\mathcal{E}_{\mathbf{x}} \left(\phi(\mathbf{x}) \phi^T(\mathbf{x}) \right) \right) = \mathcal{E}_{\mathbf{x}} \left(\text{tr} \left(P_x^T \mathbf{x} \mathbf{x}^T P_x \right) \right). \quad (\text{A.17})$$

Then,

$$= \mathcal{E}_{\mathbf{x}} \left(\text{tr} \left(P_x P_x^T \mathbf{x} \mathbf{x}^T \right) \right) = \mathcal{E}_{\mathbf{x}} \left(\text{tr} \left(\mathbf{x} \mathbf{x}^T \right) \right) = \text{tr} \left(I_{N \times N} \right) = N. \quad (\text{A.18})$$

□

Property A.0.4 further reduces the number of unknowns to $\left\lfloor \frac{N^2}{4} \right\rfloor - 1$ for $\theta = 0$.

¹ There definitely is a closed form solution in terms of Owen's Generalized T function. A good start on solving the extreme value kernels, in the case of $N = 3$, is to reduce a term with 6 variable gaussians to finding P_4 that depends on a variance of the form of $C \otimes \begin{bmatrix} 1 & -\cos \theta \\ -\cos \theta & 1 \end{bmatrix}$ [13].

Theorem A.0.5. *The trace of the Kernel matrix is :*

$$\text{tr} \left(\frac{K(\theta)}{\|\mathbf{a}\| \|\mathbf{b}\|} \right) = N \cdot \cos \theta + \mathcal{O}(\theta^3) \quad (\text{A.19})$$

Proof: *We start from Equation A.5:*

$$\text{tr} \left(\frac{K(\theta)}{\|\mathbf{a}\| \|\mathbf{b}\|} \right) = N \cdot \cos \theta + \text{tr} \left(\frac{K(0)}{\|\mathbf{a}\| \|\mathbf{b}\|} \right) - N + \text{tr}(V(\theta)). \quad (\text{A.20})$$

We then use substitute in property A.0.4:

$$\text{tr} \left(\frac{K(\theta)}{\|\mathbf{a}\| \|\mathbf{b}\|} \right) = N \cdot \cos \theta + \text{tr}(V(\theta)), \quad (\text{A.21})$$

Then, by A.0.2, the result follows. □

A.1 Rank filters of two elements

In this section, we show how to evaluate the matrix integral in Equation 4.16 for $N = (2)$ only. The non-linearity is, when $N=2$:

$$\phi(\mathbf{x}) = \begin{bmatrix} x_1 u(x_2 - x_1) + x_2 u(x_1 - x_2) \\ x_1 u(x_1 - x_2) + x_2 u(x_2 - x_1) \end{bmatrix}. \quad (\text{A.22})$$

In the above equation, u is the Heaviside unit step function. For $N=2$ and for the first element, the integral in 4.16 is:

$$K_{2,2}(\mathbf{a}, \mathbf{b}) = \mathcal{E}_W \left[(\mathbf{w}_1^T \mathbf{a} u(\Delta \mathbf{w}_{1,2}^T \mathbf{a}) + \mathbf{w}_2^T \mathbf{a} u(-\Delta \mathbf{w}_{1,2}^T \mathbf{a})) (\mathbf{w}_1^T \mathbf{b} u(\Delta \mathbf{w}_{1,2}^T \mathbf{b}) + \mathbf{w}_2^T \mathbf{b} u(-\Delta \mathbf{w}_{1,2}^T \mathbf{b})) \right] \quad (\text{A.23})$$

We used the shorthand notation of $\Delta \mathbf{w}_{i,j} = \mathbf{w}_i - \mathbf{w}_j$; $\mathbf{w}_i = \text{col}_i W$. There are four terms after expanding the product, but we really need to compute the integral for two of them since the other two are linearly dependent. The first being:

$$J_{1,1}^{max} = \mathcal{E}_W \left[\mathbf{w}_1^T \mathbf{a} u(\Delta \mathbf{w}_{1,2}^T \mathbf{a}) \mathbf{w}_1^T \mathbf{b} u(\Delta \mathbf{w}_{1,2}^T \mathbf{b}) \right]. \quad (\text{A.24})$$

Changing the variables $\mathbf{w}_2 = \mathbf{w} - \mathbf{z}$ and $\mathbf{w}_1 = \mathbf{w}$ yields:

$$J_{1,1}^{max} = \frac{1}{(2\pi)^2} \int_{\mathbb{R}^4} \mathbf{w}^T \mathbf{a} u(\mathbf{z}^T \mathbf{a}) \mathbf{w}^T \mathbf{b} u(\mathbf{z}^T \mathbf{b}) e^{-\frac{\|\mathbf{w}\|^2 + \|\mathbf{w} - \mathbf{z}\|^2}{2}} (d\mathbf{w})^\wedge (d\mathbf{z})^\wedge. \quad (\text{A.25})$$

Since in the exponential, $-\frac{\|\mathbf{w}\|^2 + \|\mathbf{w} - \mathbf{z}\|^2}{2} = -\|\mathbf{w}\|^2 + \mathbf{w}^T \mathbf{z} - \|\mathbf{z}\|^2 / 2$:

$$J_{1,1}^{max} = \frac{1}{(2\pi)^2} \int_{\mathbb{R}^4} \mathbf{b}^T \mathbf{w} \mathbf{w}^T \mathbf{a} \cdot u(\mathbf{z}^T \mathbf{a}) \cdot u(\mathbf{z}^T \mathbf{b}) e^{-\|\mathbf{w}\|^2 + \mathbf{w}^T \mathbf{z} - \|\mathbf{z}\|^2 / 2} (d\mathbf{w})^\wedge (d\mathbf{z})^\wedge. \quad (\text{A.26})$$

By using the identity in Equation C.1 we can carry out the integration in \mathbf{w} :

$$J_{1,1}^{max} = \frac{1}{(2\pi)^2} \int_{\mathbb{R}^2} \mathbf{b}^T \left(\frac{e^{\frac{\|\mathbf{z}\|^2}{4}}}{2} \pi \left(I + \frac{\mathbf{z}\mathbf{z}^T}{2} \right) \right) \mathbf{a} \cdot u(\mathbf{z}^T \mathbf{a}) \cdot u(\mathbf{z}^T \mathbf{b}) e^{-\|\mathbf{z}\|^2/2} (d\mathbf{z})^\wedge. \quad (\text{A.27})$$

Rescaling the vector $\mathbf{z} \leftarrow \sqrt{2}\mathbf{z}$ we get:

$$J_{1,1}^{max} = \frac{1}{4\pi} \int_{\mathbb{R}^2} \mathbf{b}^T (I + \mathbf{z}\mathbf{z}^T) \mathbf{a} \cdot u(\mathbf{z}^T \mathbf{a}) \cdot u(\mathbf{z}^T \mathbf{b}) e^{-\|\mathbf{z}\|^2/2} (d\mathbf{z})^\wedge. \quad (\text{A.28})$$

We can recognise arc-cosine kernels and use substitute in Equation C.8 and C.9 to get:

$$J_{1,1}^{max} = \frac{1}{2} (\mathbf{a}^T \mathbf{b} \cdot k^{(0)}(\mathbf{a}, \mathbf{b}) + k^{(1)}(\mathbf{a}, \mathbf{b})) = \frac{\|\mathbf{a}\| \|\mathbf{b}\|}{4\pi} (\cos \theta \cdot J_0(\theta) + J_1(\theta)). \quad (\text{A.29})$$

By using Equations C.12 and C.13, into the expression above:

$$J_{1,1}^{max} = \frac{\|\mathbf{a}\| \|\mathbf{b}\|}{4\pi} (\sin |\theta| + 2(\pi - |\theta|) \cos \theta). \quad (\text{A.30})$$

The second integral:

$$J_{1,2}^{max} = \mathcal{E}_W [\mathbf{w}_1^T \mathbf{a} u(\Delta \mathbf{w}_{1,2}^T \mathbf{a}) \mathbf{w}_2^T \mathbf{b} u(\Delta \mathbf{w}_{2,1}^T \mathbf{b})]. \quad (\text{A.31})$$

Substituting in $\mathbf{w}_2 = \mathbf{w} - \mathbf{z}$ and $\mathbf{w}_1 = \mathbf{w}$ yields and changing the angle of \mathbf{b} to $\pi - |\theta|$ to absorb a negative ($\mathbf{b} = -\mathbf{v}$) yields:

$$J_{1,2}^{max} = -\frac{1}{(2\pi)^2} \int_{\mathbb{R}^4} \mathbf{a}^T (\mathbf{w}\mathbf{w}^T + \mathbf{w}\mathbf{z}^T) \mathbf{v} \cdot u(\mathbf{z}^T \mathbf{a}) \cdot u(\mathbf{z}^T \mathbf{v}) e^{-\|\mathbf{w}\|^2 - \mathbf{w}^T \mathbf{z} - \|\mathbf{z}\|^2/2} (d\mathbf{w})^\wedge (d\mathbf{z})^\wedge. \quad (\text{A.32})$$

By Eq.C.1 and C.3:

$$J_{1,2}^{max} = -\frac{1}{(2\pi)^2} \int_{\mathbb{R}^2} \mathbf{a}^T \left(\frac{\pi}{2} e^{\|\mathbf{z}\|^2/4} \left(I - \frac{\mathbf{z}\mathbf{z}^T}{2} \right) \right) \mathbf{v} \cdot u(\mathbf{z}^T \mathbf{a}) \cdot u(\mathbf{z}^T \mathbf{v}) e^{-\|\mathbf{z}\|^2/2} (d\mathbf{z})^\wedge. \quad (\text{A.33})$$

Then we scale \mathbf{z} by $\sqrt{2}$ and use the arc-cosine kernel definitions:

$$J_{1,2}^{max} = -\frac{1}{(4\pi)} \int_{\mathbb{R}^2} \mathbf{a}^T (I - \mathbf{z}\mathbf{z}^T) \mathbf{v} \cdot u(\mathbf{z}^T \mathbf{a}) \cdot u(\mathbf{z}^T \mathbf{v}) e^{-\|\mathbf{z}\|^2/2} (d\mathbf{z})^\wedge, \quad (\text{A.34})$$

$$J_{1,2}^{max} = \frac{\|\mathbf{a}\| \|\mathbf{b}\|}{4\pi} (\cos(|\theta|) \cdot J_0(\pi - |\theta|) + J_1(\pi - |\theta|)). \quad (\text{A.35})$$

$$J_{1,2}^{max} = \frac{\|\mathbf{a}\| \|\mathbf{b}\|}{4\pi} \sin |\theta|. \quad (\text{A.36})$$

The kernel for the maximum is :

$$K_{2,2} = 2 \|\mathbf{a}\| \|\mathbf{b}\| (J_{1,2}^{max} + J_{1,1}^{max}). \quad (\text{A.37})$$

$$K_{2,2} = \|\mathbf{a}\| \|\mathbf{b}\| \left(\cos \theta + \frac{\sin |\theta| - |\theta| \cos \theta}{\pi} \right). \quad (\text{A.38})$$

The kernel for the minimum is equal to the maximum kernel. This can be deduced from the fact that the kernel matrix is bisymmetric. Proceeding similarly as shown above, we find that the kernel function $K_{1,2}$ is:

$$K_{1,2} = \frac{\|\mathbf{a}\| \|\mathbf{b}\|}{\pi} (|\theta| \cos \theta - \sin |\theta|). \quad (\text{A.39})$$

Thus the final kernel matrix is:

$$K(\mathbf{a}, \mathbf{b}) = \frac{\|\mathbf{a}\| \|\mathbf{b}\|}{\pi} \begin{bmatrix} \pi \cos \theta + \sin |\theta| - |\theta| \cos \theta & |\theta| \cos \theta - \sin |\theta| \\ |\theta| \cos \theta - \sin |\theta| & \pi \cos \theta + \sin |\theta| - |\theta| \cos \theta \end{bmatrix}. \quad (\text{A.40})$$

The mean for the maximum is:

$$\mu_{max} = \mathcal{E}_W [(\mathbf{w}_1^T \mathbf{a} u(\Delta \mathbf{w}_{1,2}^T \mathbf{a}) + \mathbf{w}_2^T \mathbf{a} u(-\Delta \mathbf{w}_{1,2}^T \mathbf{a}))],$$

$$\mu_{max}(\mathbf{a}) = \frac{1}{(2\pi)^2} \int_{\mathbb{R}^4} \left((\mathbf{w} - \mathbf{z})^T \mathbf{a} + \mathbf{z}^T \mathbf{a} u(\mathbf{z}^T \mathbf{a}) \right) e^{-\|\mathbf{w}\|^2 + \mathbf{w}^T \mathbf{z} - \frac{\|\mathbf{z}\|^2}{2}} (d\mathbf{w})^\wedge (d\mathbf{z})^\wedge. \quad (\text{A.41})$$

$$= \frac{\|\mathbf{a}\|}{(2\pi)} \int_{\mathbb{R}^4} ((w_1 - z_1) + z_1 u(z_1)) e^{-w_1^2 + w_1 z_1 - \frac{z_1^2}{2}} dw_1 dz_1. \quad (\text{A.42})$$

$$= \frac{\|\mathbf{a}\|}{(2\pi)} \int_{\mathbb{R}^2} z_1 u(z_1) e^{-w_1^2 + w_1 z_1 - \frac{z_1^2}{2}} dw_1 dz_1. \quad (\text{A.43})$$

$$= \frac{\|\mathbf{a}\| \sqrt{\pi}}{(2\pi)} \int_{\mathbb{R}} z_1 u(z_1) e^{-\frac{z_1^2}{4}} dz_1. \quad (\text{A.44})$$

$$= \frac{\|\mathbf{a}\|}{\sqrt{\pi}} = \mu_{max}(\mathbf{a}). \quad (\text{A.45})$$

Note that, $\boldsymbol{\mu}(\mathbf{b})$ does not depend on θ , it depends only on it's norm. The covariance matrix is:

$$C(\mathbf{a}, \mathbf{b}) = K(\mathbf{a}, \mathbf{b}) - \boldsymbol{\mu}(\mathbf{a}) \boldsymbol{\mu}^T(\mathbf{b}), \quad (\text{A.46})$$

$$C(\mathbf{a}, \mathbf{b}) = \frac{\|\mathbf{a}\| \|\mathbf{b}\|}{\pi} \begin{bmatrix} \pi \cos \theta + \sin |\theta| - |\theta| \cos \theta - 1 & |\theta| \cos \theta - \sin |\theta| + 1 \\ |\theta| \cos \theta - \sin |\theta| + 1 & \pi \cos \theta + \sin |\theta| - |\theta| \cos \theta - 1 \end{bmatrix}. \quad (\text{A.47})$$

When $\mathbf{a} \parallel \mathbf{b}$ at $\theta = 0$, then the Kernel from Equation A.40 becomes

$$K(\mathbf{a}, \mathbf{b}) \Big|_{\theta=0} = \|\mathbf{a}\| \|\mathbf{b}\| I_{2 \times 2}. \quad (\text{A.48})$$

The covariance becomes:

$$C(\mathbf{a}, \mathbf{b}) \Big|_{\theta=0} = \|\mathbf{a}\| \|\mathbf{b}\| J_{2 \times 2}. \quad (\text{A.49})$$

The derivative of the non-linearity is:

$$\boldsymbol{\phi}(\mathbf{x}) = \begin{bmatrix} u(x_2 - x_1) + u(x_1 - x_2) \\ u(x_1 - x_2) + u(x_2 - x_1) \end{bmatrix}. \quad (\text{A.50})$$

A.2 Rank filter kernels with $N > 2$

The second kernel's derivative relation at $\theta = 0$ is for all N :

$$-\frac{1}{\sin \theta} \frac{\partial}{\partial \theta} K(\mathbf{a}, \mathbf{b}) \Big|_{\theta=0} = I_{N \times N}. \quad (\text{A.51})$$

For $N=3$, at $\theta = 0$ the kernel matrix is given in eq.A.52:

$$K(\mathbf{a}, \mathbf{b}) \Big|_{\theta=0} = \|\mathbf{a}\| \|\mathbf{b}\| \left(I_{3 \times 3} + \frac{\sqrt{3}}{2\pi} \begin{bmatrix} 1 & 1 & -2 \\ 1 & -2 & 1 \\ -2 & 1 & 1 \end{bmatrix} \right). \quad (\text{A.52})$$

and the mean of of the maximum:

$$\mu_{max} = 3\mathcal{E}_W [(w_1^T \mathbf{a} u(\Delta w_{1,2}^T \mathbf{a}) u(\Delta w_{1,3}^T \mathbf{a}))], \quad (\text{A.53})$$

Upon changing variables, $\mathbf{w}_1 \leftarrow \mathbf{w}$, $\mathbf{w}_2 \leftarrow \mathbf{w} - \mathbf{y}$, $\mathbf{w}_3 \leftarrow \mathbf{w} - \mathbf{z}$ and carrying (or ignoring) the integration in z_2, y_2, z_2 :

$$\mu_{max}(\mathbf{a}) = \frac{3 \|\mathbf{a}\|}{(2\pi)^{3/2}} \int_{\mathbb{R}^3} \mathbf{w} \cdot u(y) u(z) e^{-3w^2/2 + w(z+y) - \frac{z^2}{2} - \frac{y^2}{2}} dx dy dz. \quad (\text{A.54})$$

$$\mu_{max}(\mathbf{a}) = \frac{3 \|\mathbf{a}\|}{3^{3/2} (2\pi)} \int_{\mathbb{R}^2} (x+y) \cdot u(y) u(z) e^{-\frac{y^2 - zy + z^2}{3}} dy dz. \quad (\text{A.55})$$

$$\mu_{max}(\mathbf{a}) = \frac{3 \|\mathbf{a}\|}{(2\pi)} \int_{\mathbb{R}^2} (y+z) \cdot u(y) u(z) e^{-y^2 + zy - z^2} dy dz. \quad (\text{A.56})$$

$$\mu_{max}(\mathbf{a}) = \frac{3 \|\mathbf{a}\|}{(\pi)} \int_{\mathbb{R}^2} (y) \cdot u(y) u(z) e^{-y^2 + zy - z^2} dy dz. \quad (\text{A.57})$$

Now we substitute in polar coordinates and integrate out the radius:

$$\mu_{max}(\mathbf{a}) = \frac{3 \|\mathbf{a}\|}{4\sqrt{\pi}} \int_0^{\pi/2} \frac{\cos \phi}{\sqrt{(1 - \frac{1}{2} \sin 2\phi)^3}} d\phi, \quad (\text{A.58})$$

$$\mu_{max}(\mathbf{a}) = \frac{3 \|\mathbf{a}\|}{2\sqrt{\pi}}. \quad (\text{A.59})$$

Hence,

$$\boldsymbol{\mu}(\mathbf{a}) = \frac{3 \|\mathbf{a}\|}{2\sqrt{\pi}} \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} \quad (\text{A.60})$$

For $N=4$, at $\theta = 0$ the kernel is conjectured² to be:

$$\frac{K(\mathbf{a}, \mathbf{b})|_{\theta=0}}{\|\mathbf{a}\| \|\mathbf{b}\|} = I_{4 \times 4} + \frac{\sqrt{3}}{\pi} \begin{bmatrix} 1 & 1 & -2 & 0 \\ 1 & -1 & 2 & -2 \\ -2 & 2 & -1 & 1 \\ 0 & -2 & 1 & 1 \end{bmatrix} + \frac{3}{\pi} \begin{bmatrix} 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 1 \\ 1 & -1 & 0 & 0 \\ -1 & 1 & 0 & 0 \end{bmatrix}. \quad (\text{A.61})$$

The last kernel function is for general N at $\theta = 0$ is:

$$k_{N,N} = \frac{N}{\sqrt{(2\pi)^N}} \cdot \int_x \int_{\mathbf{x}\mathbf{1} - \mathbf{x}_{N-1} \geq \mathbf{0}} x^2 e^{-\frac{x^2}{2} - \frac{\mathbf{x}^T \mathbf{x}}{2}} dx (d\mathbf{x})^\wedge. \quad (\text{A.62})$$

Substituting in $\mathbf{x} \leftarrow \mathbf{x}\mathbf{1} - \mathbf{x}_{N-1}$ we obtain

$$k_{N,N} = \frac{N}{\sqrt{(2\pi)^N}} \cdot \int_x \int_{\mathbf{x}_{N-1} \geq \mathbf{0}} x^2 e^{-N\frac{x^2}{2} + \mathbf{x}^T \mathbf{1} x - \frac{\mathbf{x}^T \mathbf{x}}{2}} dx (d\mathbf{x})^\wedge. \quad (\text{A.63})$$

$$k_{N,N} = \frac{1}{\sqrt{N} (2\pi)^N} \cdot \int_x \int_{\mathbf{x}_{N-1} \geq \mathbf{0}} x^2 e^{-\frac{x^2}{2} + \frac{\mathbf{x}^T \mathbf{1}}{\sqrt{N}} x - \frac{\mathbf{x}^T \mathbf{x}}{2}} dx (d\mathbf{x})^\wedge. \quad (\text{A.64})$$

$$k_{N,N} = \frac{1}{\sqrt{N} (2\pi)^N} \cdot \int_x \int_{\mathbf{x}_{N-1} \geq \mathbf{0}} x^2 e^{-\frac{1}{2} \left(x - \frac{\mathbf{x}^T \mathbf{1}}{\sqrt{N}}\right)^2 - \frac{1}{2} \mathbf{x}^T \left(I - \frac{\mathbf{1}\mathbf{1}^T}{N}\right) \mathbf{x}} dx (d\mathbf{x})^\wedge. \quad (\text{A.65})$$

$$k_{N,N} = \frac{1}{\sqrt{N} (2\pi)^{N-1}} \cdot \int_{\mathbf{x}_{N-1} \geq \mathbf{0}} \left(1 + \frac{(\mathbf{x}^T \mathbf{1})^2}{N}\right) e^{-\frac{1}{2} \mathbf{x}^T \left(I - \frac{\mathbf{1}\mathbf{1}^T}{N}\right) \mathbf{x}} (d\mathbf{x})^\wedge. \quad (\text{A.66})$$

$$k_{N,N} = \frac{1}{\sqrt{N} (2\pi)^{N-1}} \cdot \int_{\mathbf{x}_{N-1} \geq \mathbf{0}} \left(1 + \frac{(\mathbf{x}^T \mathbf{1})^2}{N}\right) e^{-\frac{1}{2} \mathbf{x}^T (I + \mathbf{1}\mathbf{1}^T)^{-1} \mathbf{x}} (d\mathbf{x})^\wedge. \quad (\text{A.67})$$

Then we substitute $\mathbf{x} \leftarrow \sqrt{2}\mathbf{x}$ to make the covariance matrix have 1's as diagonal entries, and set $c_N = \frac{1}{\sqrt{N(\pi)^{N-1}}}$:

$$k_{N,N} = c_N \cdot \int_{\mathbf{x}_{N-1} \geq \mathbf{0}} \left(1 + 2\frac{(\mathbf{x}^T \mathbf{1})^2}{N}\right) e^{-\frac{1}{2} \mathbf{x}^T (I + \frac{1}{2}(\mathbf{1}\mathbf{1}^T - I))^{-1} \mathbf{x}} (d\mathbf{x})^\wedge. \quad (\text{A.68})$$

$$k_{N,N} = c_N \cdot \left(g_1 + 2\frac{g_2}{N}\right), \quad (\text{A.69})$$

²This was a guess tested with Monte-Carlo simulations and the digits matched up to 4 decimals. The integrals become increasingly tedious to solve. For $N = 4$, we need to integrate over a spherical simplex. However, for $N > 5$ we need to integrate over an n-sphere. We thus solved the integral of the first element of $K(\mathbf{a}, \mathbf{b})$. This result combined with property A.10, we found that two other kernel function matched the integral. Noting that the terms are powers of π and integers mixed with square roots, we deduced A.61

where

$$g_1 = \int_{\mathbf{x}_{N-1} \geq \mathbf{0}} e^{-\frac{1}{2} \mathbf{x}^T (I + \frac{1}{2} (\mathbf{1}\mathbf{1}^T - I))^{-1} \mathbf{x}} (d\mathbf{x})^\wedge, \quad (\text{A.70})$$

and

$$g_2 = \int_{\mathbf{x}_{N-1} \geq \mathbf{0}} (\mathbf{x}^T \mathbf{1})^2 e^{-\frac{1}{2} \mathbf{x}^T (I + \frac{1}{2} (\mathbf{1}\mathbf{1}^T - I))^{-1} \mathbf{x}} (d\mathbf{x})^\wedge. \quad (\text{A.71})$$

Equation A.68 is related to Aomoto's hypergeometric function for an integral (see Equation A.72) over a spherical simplex [5]. Aomoto shows that the gaussian orthant probability equals to an integral over a spherical simplex and he solves a general case.

$$f_N = \int_{\mathbf{x} \geq \mathbf{0}} e^{-\frac{1}{2} \mathbf{x}^T B \mathbf{x}} (d\mathbf{x})^\wedge, \quad (\text{A.72})$$

where B is the precision matrix. Aomoto then express a solution that depend on the correlation matrix terms. His solution is an infinite power series in terms of dihedral angles. When variates are equicorrelated or have other simple structures, it is possible to have closed form solutions in terms of inverse trigonometric and polylogarithm functions. For example, in our case for $N = 2$ and 3 , we have terms that are algebraic numbers scaled by inverse powers of π . Indeed, integrals of this type are well known and solved up to an orthant of order 5 and 4 exactly for various types of correlation matrices [13]. Cheng's equicorrelated case solution's for dimensions 4 and 5 depend on dilogarithms³ and inverse trigonometric functions. Schläfli [36] has shown a recursive formula to solve orthant integrals over an odd number of variables f_{2k+1} provided we know the formulas for all the smaller dimension f_n 's such that $n < 2k + 1$. However, it is not possible to do so for even numbered orthant integration (because the term desired vanishes out). Instead, one must use Schläfli's differential recursive formula [34] for f_{2k} . One must rely on Monte-Carlo simulations to get the variances.

³ When evaluated at $\rho = 1/2$, the dilogarithm functions can be expressed as a sum of dilogarithms of different powers of the inverse golden ratio ϕ^{-1} . Unfortunately, even if there are many simplifications associated to the golden ratio, we couldn't find a closed form solution that eliminates the dilogarithms altogether.

Appendix B

A Curious Link Between Prime Numbers, the Maundy Cake Problem and Parallel Sorting.

B.1 Introduction

Computing is generally done using binary comparators. Perhaps the most intuitive way of sorting is by using brute force. If we count the number of times an element is larger than another, then effectively, we indirectly are implementing Insertion Sort or a similar Selection sort. By counting the number of times an element is larger than the others in the sequence, we effectively ranked the element. By obtaining the rank, we can sort the sequence with a permutation. However, ranking (to count the number of times an element is larger than another) can be assigned to multiple processors simultaneously and is therefore highly parallelizable. In the insertion sort or the selection sort algorithm, the depth is of the order of the sequence length. Since we rank in parallel, we have, in the end, a reduction (or a sum) of $\mathcal{O}(N)$ comparator arrays. Again, using GPU's, this can be reduced in $\mathcal{O}(\log(N))$ time. But in most algorithms, comparators used are binary. In this paper, we study if we can use comparators that take more than two numbers. More specifically, we see what comparator sizes are possible given N , the size of the sequence to be ranked. When we count the number of partial ranks(output of a comparator bank) required to be reduced, we get a direct connection to the Maundy cake problem [8]. Additionally, we analyze the number of comparators that will have to be used, and we provide a new number-theoretical sequence related to them.

The new divide and conquer algorithms are mainly theoretical since it is not scalable for large N . It still requires $\mathcal{O}(N^2)$ processors running in parallel in the worse case, and it requires to factorize a large number, which is a nondeterministic polynomial time (NP) problem [21]. Another possible drawback is that the processors need to output n -ary comparators, hence they could need specialized hardware to implement them.

Sorting a sequence is a non-linear in the sense that the matrix that transforms the sequence is a permutation matrix \mathbf{P}_x that depends on the data itself, i.e.

$$\mathbf{s} = \mathbf{P}_x^T \mathbf{x} \tag{B.1}$$

We will drop the data dependence subscript for brevity so $\mathbf{P}_x = \mathbf{P}$. Furthermore,

$$\mathbf{P} = \begin{bmatrix} \mathbf{e}_{\pi_0} & \cdots & \mathbf{e}_{\pi_{N-1}} \end{bmatrix} \quad (\text{B.2})$$

where \mathbf{e}_i is a zero $N \times 1$ vector except the i th entry equals 1. Hence if we define $\mathbf{n}_{N \times 1}$ to be:

$$\mathbf{n}_{N \times 1} = \begin{bmatrix} 0 & 1 & \cdots & N-1 \end{bmatrix}^T \quad (\text{B.3})$$

then,

$$\mathbf{P}^T \mathbf{n} = \boldsymbol{\pi} \quad (\text{B.4})$$

where $\boldsymbol{\pi} = \begin{bmatrix} \pi_0 & \cdots & \pi_{N-1} \end{bmatrix}^T$.

B.2 Converting a sorting problem into a sum problem.

Note that, unless otherwise stated, Zero-indexing is used in this project. If we consider equation B.1 and B.2 we see that what really is important is the indices $\boldsymbol{\pi}$ to order a sequence. That is if we solve for $\boldsymbol{\pi}$, we indirectly know what our sorted sequence \mathbf{s} is since $\boldsymbol{\pi}$ fully characterizes the permutation matrix. In this section we will use comparator units that output ranks or keys $\boldsymbol{\pi}$ of the corresponding input sequence number \mathbf{x} , instead of using them to output a “swapped in order” sequence \mathbf{s} . If we consider that a K-ary comparator $\mathcal{C}_{K \times 1}$ or simply \mathcal{C}_K , it would output keys $\in [0, 1, \dots, K-1]$:

$$\mathcal{C}_N(\mathbf{x}_{N \times 1}) = \boldsymbol{\pi} \quad (\text{B.5})$$

For example, $\mathcal{C}_3 \left(\begin{bmatrix} 6.4 & -9.3 & 0.1 \end{bmatrix}^T \right) = \begin{bmatrix} 2 & 0 & 1 \end{bmatrix}^T$ and $\mathcal{C}_2 \left(\begin{bmatrix} -40.56 & 10.76 \end{bmatrix}^T \right) = \begin{bmatrix} 0 & 1 \end{bmatrix}^T$.

Consider for now that the sequence has no repeated elements. Then, for the case where we have exactly 2 elements to compare, we can model \mathcal{C}_2 as being a boolean function that outputs 0 and 1 for false and true respectively:

$$\mathcal{C}_2 \left(\begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \right) = \begin{bmatrix} (x_0 > x_1) \\ (x_1 > x_0) \end{bmatrix} = \begin{bmatrix} \delta[x_0 - x_1] \\ \delta[x_1 - x_0] \end{bmatrix} = \boldsymbol{\pi} \quad (\text{B.6})$$

The relation above is important enough to be given a special notation $c_{i,j}$ for output of the comparison $(x_i > x_j) = \delta[x_i - x_j] = c_{i,j}$. We can rewrite eq. B.6 into:

$$\mathcal{C}_2 \left(\begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \right) = \begin{bmatrix} c_{0,1} \\ c_{1,0} \end{bmatrix} \quad (\text{B.7})$$

B.2.1 Binary partial ranks

The first algorithm is very intuitive. To illustrate it with an example, if in a sequence of 16 numbers, I told you that one of the elements is bigger than exactly six elements, then we know that this is the seventh element. We don't need to know the order of the other

elements to infer this. Hence it is possible to do sorting in parallel by finding the permutation index of a sequence element by counting the number of times one element is higher than the other. So we sum together the partial ranks to get the correct index. In vector notation, the permutation vector $\boldsymbol{\pi}_{N \times 1}$ is a sum of elements of partial ranks stored in a matrix $\mathbf{C}_{N \times N}(\mathbf{x})$ or just \mathbf{C}_N .

$$\mathbf{C}_N(\mathbf{x}) = \boldsymbol{\pi} = \mathbf{C}_N \mathbf{1}_{N \times 1} \quad (\text{B.8})$$

The entry (i, j) of matrix \mathbf{C}_N are defined by:

$$c_{i,j} = \begin{cases} \delta[x_i - x_j], & \text{if } i > j \\ 1 - \delta[x_i - x_j], & \text{if } j > i \\ 0 & \text{if } i = j \end{cases}$$

or perhaps more clearly:

$$\mathbf{C}_N = \begin{bmatrix} 0 & c_{0,1} & c_{0,2} & \dots \\ \bar{c}_{0,1} & 0 & c_{1,2} & \dots \\ \bar{c}_{0,2} & \bar{c}_{1,2} & 0 & \dots \\ \vdots & & & \ddots \end{bmatrix} \quad (\text{B.9})$$

Here \bar{c} is the 1-bit complement of c . The diagonal is assigned a 0 by default because it never contributes to the ordering permutation. The lower diagonal are filled with complements of the upper diagonal elements because this way the permutation indices can't repeat themselves. Note that if \mathbf{x} has repeated values there could be multiple possible solution for $\boldsymbol{\pi}$ since the elements π_i could be repeated. Despite that, even if there are equal elements in the vector \mathbf{x} , the permutation indices will correspond to a sorted vector, i.e. the sorting algorithm is stable. The matrix \mathbf{C}_N is skew-symmetric in the 1-bit sense, in other words the negative sign is replaced with the boolean complement. Let us denote the space of the boolean skew-symmetric matrices by $\mathbb{S}^{N \times N}$ and the possible comparison matrix space by $\mathbb{C}^{N \times N}$. In general, not all boolean skew-symmetric matrices correspond to a possible \mathbf{x} , or $\mathbb{C}^{N \times N} \subsetneq \mathbb{S}^{N \times N}$ for $N > 2$. For example, there is no sequence \mathbf{x} that correspond to the following matrix:

$$\mathbf{S}_4 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

The indexing vector $\boldsymbol{\pi}$ will correspond to the indices of a stable sorting algorithm if the permutation indices are unique. The structure imposed on \mathbf{C}_N also ensures that the solution $\boldsymbol{\pi}$ has unique elements.

Proof: Let $v(\mathbf{C}_N)$ denote the $\frac{1}{2}(N-1)N \times 1$ vector obtained from the vectorized matrix $\text{vec}(\mathbf{C}_N)$ by eliminating all the diagonal and lower diagonal elements of \mathbf{C}_N . For example if $N = 3$, $\mathbf{c} = v(\mathbf{C}_N) = [c_{0,1} \ c_{0,2} \ c_{1,2}]^T$. Then the sum of partial indices is:

$$\mathbf{C}_N \mathbf{1}_{N \times 1} = \boldsymbol{\Delta}_{N \times \frac{1}{2}(N-1)N} \mathbf{c} + \mathbf{n} \quad (\text{B.10})$$

The matrix $\Delta_{N \times \frac{1}{2}(N-1)N} = \Delta_N$ has the following recursive structure:

$$\Delta_N = \begin{bmatrix} \Delta_{N-1} & \mathbf{I}_{N-1 \times N-1} \\ \mathbf{0}_{N-1 \times 1}^T & -\mathbf{1}_{N-1 \times 1}^T \end{bmatrix} \quad (\text{B.11})$$

The smallest of these matrices is: $\Delta_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$. We have¹ $\text{rank}(\Delta_N) = N - 1$ and

$$\mathbf{1}^T \Delta_N = \mathbf{0}_{\frac{1}{2}(N-1)N \times 1}^T \quad (\text{B.12})$$

Combining equations B.8, B.10 and B.4 we obtain:

$$\Delta_N \mathbf{c} + \mathbf{n} = \mathbf{P}^T \mathbf{n} \quad (\text{B.13})$$

Assume that indeed π has repeated values, then \mathbf{P}^T has repeated 1's in at least a column and at least a zero column. In other words, assume $\text{rank}(\mathbf{P}) < N$. If we multiply the left of eq. B.13 with $\mathbf{1}^T$ we get: $\mathbf{1}^T \Delta_N \mathbf{c} + \mathbf{1}^T \mathbf{n} = \mathbf{1}^T \mathbf{P}^T \mathbf{n}$. Because of eq. B.12, this reduces to: $\mathbf{1}^T \mathbf{n} = \mathbf{1}^T \mathbf{P}^T \mathbf{n}$. However, for this equation to hold \mathbf{P}^T must be of full rank. This contradicts the assumption. \square

Algorithm 1 Binary algorithm

```

1: for  $i = 0$  to  $N - 2$  do
2:   for  $j = 1$  to  $N - 1$  do
3:     if  $x[i] > x[j]$  then
4:        $c[i][j] = 1$ 
5:        $c[j][i] = 0$ 
6:     else
7:        $c[i][j] = 0$ 
8:        $c[j][i] = 1$ 
9:     end if
10:  end for
11: end for
12: for  $(i \neq j)$  and  $(j = 0$  to  $N)$  do
13:    $\pi[i] = \pi[i] + c[i][j]$ 
14: end for
15: for  $j = 0$  to  $N$  do
16:    $s[\pi[i]] = x[i]$ 
17: end for

```

The cumulative sum is done in parallel too. Since the elements in the sum are binary, it could be possible to use logical circuits to implement this. So it is possible to convert the problem of sorting of N numbers into the problem of $N-1$ binary cumulative sums. For a

¹This is a matrix rank, not to be confused with a sorting order rank.

sequence of length N , the comparison complexity required to sort N numbers (\mathcal{C}_N) in terms of the quantity of binary comparisons (\mathcal{C}_2) is :

$$\mathcal{C}_N \equiv \frac{N^2 - N}{2} \mathcal{C}_2 = \mathcal{O}(N^2) \mathcal{C}_2 \quad (\text{B.14})$$

Note that we can see this directly from equation B.9 since there are $\frac{N^2 - N}{2}$ disjoint binary comparisons to represent \mathcal{C}_N , so the above equation reads as: A N -ary comparison is equivalent to $\frac{N^2 - N}{2} \mathcal{C}_2$ simultaneous binary comparisons. The quadratic term shows that the algorithm is practically useless for large sequences. Recall that only short sequences are of interest here.

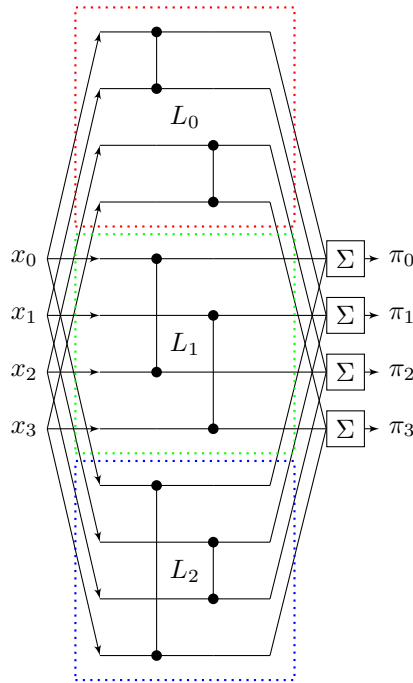


Figure B.1: The comparators only outputs ranks, they don't swap the compared numbers. The ranks from the three parallel levels L_0 , L_1 and L_2 are added together in the end. There are three levels and each level has 2 binary comparators.

The question arises if we can minimize the number of parallel levels labelled L_i using ternary or even K -ary comparisons. In fact, this is possible, for example a sequence of length 9 can be separated into 4 parallel nets using ternary comparisons only. The generalization of this for K -ary comparisons is the aim of the next subsection. Furthermore, we would like the levels to be maximal, i.e. the number of K -ary comparators per level is $\frac{N}{K}$.

B.2.2 Smallest divisor partitioning of partial ranks

Let the sequence length N be factored into a pair of divisors:

$$N = d \cdot D \quad (\text{B.15})$$

In the above equation, d is the smallest prime divisor of N , and D is the largest divisor of N or the conjugate divisor of d since $D = N/d$. The topology of the connections of such networks is what determines if the network design sorts correctly. Before stating the general formula, we will enunciate some definitions.

The MATLAB convenient notation for a vector permutation is used, i.e.:

$$\mathbf{y}_{N \times 1} = \mathbf{x}_{N \times 1} (\boldsymbol{\pi}_{N \times 1}) \implies y_i = x_{\pi_i}; i \in [0, N - 1] \quad (\text{B.16})$$

The $M \times L$ matrix $\mathbf{E}_{i,j,M \times L}$, or simply $\mathbf{E}_{i,j}$, has 1 in its (i, j) entry and 0's elsewhere. Thus,

$$\mathbf{E}_{i,j} = \boldsymbol{\delta}_i \boldsymbol{\delta}_j^T \quad (\text{B.17})$$

Let an index vector $\mathbf{v}_{j,k} \in \mathbb{R}^{d \times 1}$ be constructed with it's i th element defined as:

$$v_{i,j,k} = (j + k \cdot i) \bmod D + D \cdot i \quad (\text{B.18})$$

where $i \in [0, d - 1]$, $j \in [0, D - 1]$, $k \in [0, D - 1]$. Let a second index vector $\mathbf{w}_j \in \mathbb{R}^{D \times 1}$ with it's i th element defined as $w_{i,j} = (i) \bmod D + D \cdot j$:

$$\mathbf{w}_j = \mathbf{n}_{D \times 1} + i \cdot D \cdot \mathbf{1}_{D \times 1} \quad (\text{B.19})$$

Then we have:

$$\boldsymbol{\pi}_{N \times 1} = \sum_{j=0}^{d-1} \boldsymbol{\delta}_{j,d \times 1} \otimes \mathcal{C}_{D \times 1}(\mathbf{x}(\mathbf{w}_j)) + \sum_{k=0}^{D-1} \sum_{j=0}^{D-1} \left(\sum_{i=0}^{d-1} \mathbf{E}_{v_{i,j,k,i,N \times d}} \right) \mathcal{C}_{d \times 1}(\mathbf{x}(\mathbf{v}_{j,k})) \quad (\text{B.20})$$

Here \otimes denotes the Kronecker product. The pattern for the connections are illustrated with an example figure with $N = 6$ network that looks like:

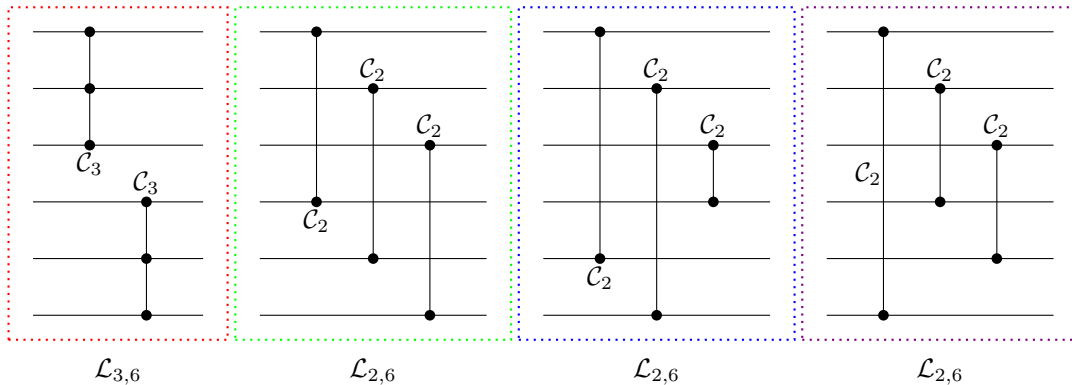


Figure B.2: At the left $\mathcal{N}_{3,6}$ is the ternary network, the others are binary networks. The binary and ternary comparators are labeled with \mathcal{C}_2 and \mathcal{C}_3 respectively. There are 1 level of ternary comparators and there are 3 levels of binary comparators hence $\mathcal{L}_{6,6} \equiv \mathcal{L}_{3,6} + 3\mathcal{L}_{2,6}$. There is 2 ternary comparators in the ternary level and 3 binary ones per binary level, i.e. $\mathcal{C}_6 \equiv 2\mathcal{C}_3 + 9\mathcal{C}_2$.

We can decompose a sorting network of N numbers into one D -ary network and D d -ary nets:

$$\mathcal{L}_{N,N} \equiv 1 \cdot \mathcal{L}_{D,N} + D \cdot \mathcal{L}_{d,N} \quad (\text{B.21})$$

Let \mathcal{C}_D denote a D -ary comparison. Then each network $\mathcal{L}_{d,N}$ contains D d -ary comparisons:

$$\mathcal{L}_{d,N} \equiv \frac{N}{d} \cdot \mathcal{C}_d = D \cdot \mathcal{C}_d \quad (\text{B.22})$$

Each comparator unit \mathcal{C}_{p_i} can be modelled by any algorithm, provided it saves the permutation indices.

		Partial rank from levels					
i	x_i	$\mathcal{L}_{8,4}$	$\mathcal{L}_{8,2}$	$\mathcal{L}_{8,2}$	$\mathcal{L}_{8,2}$	$\mathcal{L}_{8,2}$	π_i
0	5	2	0	0	0	0	2
1	12	3	1	1	1	1	7
2	2	0	0	0	0	0	0
3	3	1	0	0	0	0	1
4	5	0	1	1	1	0	3
5	7	0	1	1	1	0	5
6	8	3	1	0	1	1	6
7	6	1	1	1	0	1	4

Partial ranks, or output of \mathcal{L}_{N,d_j} and their sum.

Table B.1: Example of summing partial ranks with divisor partitioning (eq.B.20) with $N = 8$. Here, $D = 4$ and the first column of outputs corresponds to 2 quaternary comparator ranks placed one above the other, hence it's partial ranks go from 0 to 3. The last column are the ranks, or the sum of all the partial ranks.

Substituting eq.B.22 into B.21 we get the mixed comparisons complexity:

$$\mathcal{C}_N \equiv d \cdot \mathcal{C}_{N/d} + \left(\frac{N}{d}\right)^2 \cdot \mathcal{C}_d \quad (\text{B.23})$$

If we take take eq. B.23, and substitute in eq.B.14, we show that the binary complexity are the same for both algorithms:

$$\mathcal{C}_N \equiv d \cdot \frac{1}{2} \left(\left(\frac{N}{d}\right)^2 - \frac{N}{d} \right) \mathcal{C}_2 + \left(\frac{N}{d}\right)^2 \cdot \frac{1}{2} (d^2 - d) \mathcal{C}_2 = \frac{1}{2} (N - 1) N \mathcal{C}_2 \quad (\text{B.24})$$

The above relation is a hint that both algorithms are equivalent. If we use algorithm 1, to model all comparisons, the whole network reduces to binary comparisons. This effectively reduces the conjugate divisor algorithm into alg. 1. Although this substitution wouldn't be interesting because the goal of this section was to see if it was possible to use non binary comparators, this reduction means that the two algorithms are equivalent.

Proof:

If we take eq.B.20 and substitute all comparisons with eq.B.8 without summing in the columns together we get:

$$\mathbf{C}_N = \sum_{j=0}^{d-1} \mathbf{E}_{j,j,d \times d} \otimes \mathbf{C}_{D \times D}(\mathbf{x}(\mathbf{w}_j)) + \sum_{k=0}^{D-1} \sum_{j=0}^{D-1} \mathbf{C}_{d \times d}(\mathbf{x}(\mathbf{v}_{j,k})) \otimes \mathbf{E}_{k,j,D \times D} \quad (\text{B.25})$$

The above equation shows that the binary matrix \mathbf{C}_N can be represented as an addition of small discontinuous tiles i.e. \mathbf{C}_D 's and \mathbf{C}_d 's. \square

B.2.3 Prime partitioning of partial ranks

Let

$$N = \prod_i^m f_i = \prod_j^n p_j^{k_j} \quad (\text{B.26})$$

Where f_i are prime factors of N and $f_1 \leq \dots \leq f_m$. The p_j 's are the unique prime factors and k_j is their corresponding power and $p_1 < \dots < p_n$. If D can be further decomposed into prime factors then we can reapply the algorithm iteratively on equation B.21 until no further decompositions are possible. Using this iterative scheme, we can sort N numbers using minimal sized partitions consisting of the prime factors of N . We start the steps by rewriting equation B.21:

$$\mathcal{L}_{N,N} \equiv 1 \cdot \mathcal{L}_{\frac{N}{f_1},N} + \frac{N}{f_1} \cdot \mathcal{L}_{f_1,N} \quad (\text{B.27})$$

We then repeat the recursion until we can't factor any more:

$$\mathcal{L}_{N,N} \equiv \mathcal{L}_{\frac{N}{f_1},N} + \frac{N}{f_1} \cdot \mathcal{L}_{f_1,N} \equiv \left(\mathcal{L}_{\frac{N}{f_1 f_2},N} + \frac{N}{f_1 f_2} \cdot \mathcal{L}_{f_2,N} \right) + \frac{N}{f_1} \cdot \mathcal{L}_{f_1,N} \equiv \dots$$

We then end up with the sum:

$$\mathcal{L}_{N,N} \equiv \sum_{i=1}^m \frac{N}{\prod_{j=1}^i f_j} \mathcal{L}_{f_i,N} = \sum_{i=1}^m \left(\prod_{j=i}^m f_j \right) \frac{\mathcal{L}_{f_i,N}}{f_i} \quad (\text{B.28})$$

This sequence can be then be organised wrt to it's unique prime factors:

$$\mathcal{L}_{N,N} \equiv \sum_{i=1}^n \frac{N}{\prod_{j=1}^i p_j^{k_j}} \frac{p_i^{k_i} - 1}{p_i - 1} \mathcal{L}_{p_i,N} \quad (\text{B.29})$$

Applying equation B.22 to B.28, we get a formula for the different types of comparisons:

$$\mathcal{C}_N \equiv \sum_{i=1}^m \frac{N}{\prod_{j=1}^i f_j} \binom{N}{f_i} \mathcal{C}_{f_i} = N \sum_{i=1}^m \left(\prod_{j=i}^m f_j \right) \frac{\mathcal{C}_{f_i}}{f_i^2} \quad (\text{B.30})$$

$$\mathcal{C}_N \equiv \sum_{i=1}^n \frac{N}{\prod_{j=1}^i p_j^{k_j}} \frac{p_i^{k_i} - 1}{p_i - 1} \frac{N}{p_i} \mathcal{C}_{p_i} \quad (\text{B.31})$$

The proof for eq. B.31 follows the same lines as the proof for equation B.33. If we substitute each $\mathcal{L}_{p_i, N}$ in equation B.28 with 1, we get the number of column vectors of keys to be added in the final adder step:

$$|\mathcal{L}_N| = \sum_{i=1}^m \frac{N}{\prod_{j=1}^i f_j} \quad (\text{B.32})$$

Consequently, the addition complexity $\mathcal{C}_N^\oplus = |\mathcal{L}_N| - 1$. A special case of eq. B.28 when $N = p^k$ include:

$$\mathcal{L}_{N, N} \equiv \frac{N-1}{p-1} \mathcal{L}_{p, N} \quad (\text{B.33})$$

Proof: We start with equation B.28 and set all primes $p_i = p$. We get a geometrical series $\mathcal{L}_{N, N} \equiv \sum_{i=1}^m p^{m-i} \mathcal{L}_{p, N} = p^{m-1} \frac{1-p^{-m}}{1-p^{-1}} \mathcal{L}_{p, N}$ and B.33 follows. \square

The comparison complexity for prime powers is:

$$\mathcal{C}_N \equiv \frac{N-1}{p-1} \cdot \frac{N}{p} \mathcal{C}_p \quad (\text{B.34})$$

A second case of eq. B.28 when $N = p_1^{k_1} p_2^{k_2}$ include:

$$\mathcal{L}_{N, N} \equiv p_2^{k_2} \frac{p_1^{k_1} - 1}{p_1 - 1} \mathcal{L}_{p_1, N} + \frac{p_2^{k_2} - 1}{p_2 - 1} \mathcal{L}_{p_2, N}. \quad (\text{B.35})$$

Now upon inspecting the formula for $|\mathcal{L}_N|$ in equation B.32 we see that the upper bound is obtained when the number of prime factors m is maximal and the denominator is minimal. This happens only when $N = 2^m$. Then $|\mathcal{L}_{2^m}| = N - 1$. The minimum is when N is prime and hence $|\mathcal{L}_p| = 1$ and so we have:

$$1 \leq |\mathcal{L}_N| \leq N - 1; N \geq 2 \quad (\text{B.36})$$

The formula provided in the sequence of OEIS **A006022**, (related to the Maundy cake problem [8]) is:

$$a(N) = \max \left([d \cdot a(N/d) + 1]_{\forall d|N} \right) \quad (\text{B.37})$$

applying the iteration until we cover all the values we end up with the formula for $|\mathcal{L}_N|$. In other words, $a(N) = |\mathcal{L}_N|$ implies that $|\mathcal{L}_N|$ are Nim numbers.

If we are interested in the total number of comparisons regardless of the number of input the comparators take we get by setting all \mathcal{C}_{f_i} to 1:

$$|\mathcal{C}_N| \equiv \sum_{i=1}^m \frac{N^2}{f_i \prod_{j=1}^i f_j} \quad (\text{B.38})$$

It can be shown using the same arguments to show the limits in B.36 that:

$$1 \leq |\mathcal{C}_N| \leq \frac{N^2 - N}{2}; N \geq 2 \quad (\text{B.39})$$

The first 50 terms are:

0, 1, 1, 6, 1, 11, 1, 28, 12, 27, 1, 58, 1, 51, 28, 120, 1, 105, 1, 154, 52, 123, 1, 260, 30, 171, 117, 298,
1, 281, 1, 496, 124, 291, 54, 534, 1, 363, 172, 708, 1, 545, 1, 730, 309, 531, 1, 1096, 56, 685.

N	p							$ \mathcal{L}_N $
	2	3	5	7	11	13	...	
2	1							1
3		1						1
4	3							3
5			1					1
6	3	1						4
7				1				1
8	7							7
9		4						4
10	5		1					6
11					1			1
12	9	1						10
13						1		1
14	7			1				8
15		5	1					6
16	15							15

Table B.2: Coefficients in $\mathcal{L}_{p,N}$ of eq.B.29 and $|\mathcal{L}_{N,N}|$.

N	p							$ \mathcal{C}_N $
	2	3	5	7	11	13	...	
2	1							1
3		1						1
4	6							6
5			1					1
6	9	2						11
7				1				1
8	28							28
9		12						12
10	25		2					27
11					1			1
12	54	4						58
13						1		1
14	49			2				51
15		25	3					28
16	120							120

Table B.3: Coefficients in \mathcal{C}_{p_i} of eq.B.31 and $|\mathcal{C}_N|$.

Appendix C

Facts

$$\int_{\mathbb{R}^n} \mathbf{w} \mathbf{w}^T e^{-\mathbf{w}^T A \mathbf{w} \pm \mathbf{b}^T \mathbf{w}} (d\mathbf{w})^\wedge = \frac{e^{\frac{\mathbf{b}^T A^{-1} \mathbf{b}}{4}}}{2} \sqrt{\frac{\pi^n}{|A|}} \left(A^{-1} + \frac{A^{-1} \mathbf{b} \mathbf{b}^T A^{-1}}{2} \right). \quad (\text{C.1})$$

$$\int_{\mathbb{R}^n} \mathbf{w} \mathbf{w}^T e^{-\mathbf{w}^T \Sigma^{-1} \mathbf{w} / 2} (d\mathbf{w})^\wedge = \Sigma \cdot 2^{n/2} \pi^{n/2} \sqrt{|\Sigma|}. \quad (\text{C.2})$$

$$\int_{\mathbb{R}^n} \mathbf{w} e^{-\mathbf{w}^T A \mathbf{w} - \mathbf{b}^T \mathbf{w}} (d\mathbf{w})^\wedge = -\frac{A^{-1} \mathbf{b}}{2} \sqrt{\frac{\pi^n}{|A|}} e^{\frac{\mathbf{b}^T A^{-1} \mathbf{b}}{4}}. \quad (\text{C.3})$$

If $W_{n \times m} = [\mathbf{w}_1, \dots, \mathbf{w}_m]$, and $\text{vec}(W) = \mathcal{W} \sim \mathcal{N}(\mathbf{0}_{mn \times 1}, \Sigma_{mn \times mn})$, with

$$\Sigma = \begin{bmatrix} \Sigma_{11} & 0 & \cdots & 0 \\ 0 & \Sigma_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \Sigma_{mm} \end{bmatrix},$$

and $\Sigma_{ii} \in \mathbb{R}_{n \times n}$.

$$\begin{aligned} \int_{\mathbb{R}^{nm}} W_{m \times n}^T B_{n \times n} W_{n \times m} e^{-\text{tr}(\mathcal{W}^T \Sigma^{-1} \mathcal{W}) / 2} (dW)^\wedge &= \int_{\mathbb{R}^{nm}} [\mathbf{w}_i^T B_{n \times n} \mathbf{w}_j]_{i,j} e^{-\text{tr}(\mathcal{W}^T \Sigma^{-1} \mathcal{W}) / 2} (dW)^\wedge \\ &= \int_{\mathbb{R}^{nm}} [\mathbf{1}^T (B_{n \times n} \circ \mathbf{w}_i \mathbf{w}_j^T) \mathbf{1}]_{i,j} e^{-\text{tr}(\mathcal{W}^T \Sigma^{-1} \mathcal{W}) / 2} (dW)^\wedge \\ &= [\mathbf{1}^T (B_{n \times n} \circ \Sigma_{ij}) \mathbf{1}]_{m \times m} \sqrt{(2\pi)^{mn} |\Sigma|} = [\text{tr}(B_{n \times n} \Sigma_{ij}^T)]_{m \times m} \sqrt{(2\pi)^{mn} |\Sigma|} \end{aligned} \quad (\text{C.4})$$

$$= [\text{tr}(B_{n \times n} \Sigma_{ij}^T)]_{m \times m} \sqrt{(2\pi)^{mn} |\Sigma_{11}|^{\frac{1}{2}} |\Sigma_{22}|^{\frac{1}{2}} \cdots |\Sigma_{mm}|^{\frac{1}{2}}}. \quad (\text{C.5})$$

In the simple case where $\Sigma_{ii} = \Sigma_{n \times n}$,

$$\int_{\mathbb{R}^{nm}} W_{m \times n}^T B_{n \times n} W_{n \times m} e^{-\text{tr}(\mathcal{W}^T \Sigma^{-1} \mathcal{W}) / 2} (dW)^\wedge = \text{tr}(B_{n \times n} \Sigma_{n \times n}^T) \cdot I_{m \times m} \sqrt{(2\pi)^{mn} |\Sigma_{n \times n}|^{\frac{m}{2}}},$$

and

$$\int_{\mathbb{R}^{nm}} W_{m \times n}^T W_{n \times m} e^{-\text{tr}(\mathcal{W}^T \Sigma^{-1} \mathcal{W}) / 2} (dW)^\wedge = \text{tr}(\Sigma_{n \times n}) \cdot I_{m \times m} \sqrt{(2\pi)^{mn} |\Sigma_{n \times n}|^{\frac{m}{2}}}.$$

This includes the special case $\Sigma_{ii} = \sigma^2 I_{n \times n}$,

$$\int_{\mathbb{R}^{nm}} W_{m \times n}^T B_{n \times n} W_{n \times m} e^{-tr(\mathcal{W}^T \Sigma^{-1} \mathcal{W})/2} (dW)^\wedge = \sigma^2 tr(B_{n \times n}) \cdot I_{m \times m} \sqrt{(2\pi)^{mn}} \sigma^{mn}. \quad (\text{C.6})$$

It then follows that

$$\int_{\mathbb{R}^{nm}} W_{m \times n}^T W_{n \times m} e^{-tr(\mathcal{W}^T \Sigma^{-1} \mathcal{W})/2} (dW)^\wedge = \sigma^2 n \cdot I_{m \times m} \sqrt{(2\pi)^{mn}} \sigma^{mn}. \quad (\text{C.7})$$

Additionally,

The following Equations C.8, C.9 and C.10 are taken from [14]. Equation C.9 is named the arc-cosine kernel [14], [26].

$$k^{(m)}(\mathbf{a}, \mathbf{b}) = \frac{1}{(2\pi)^{n/2}} \int_{\mathbb{R}^n} (\mathbf{a}^T \mathbf{w} \mathbf{w}^T \mathbf{b})^m \cdot u(\mathbf{w}^T \mathbf{a}) \cdot u(\mathbf{w}^T \mathbf{b}) e^{-\frac{\mathbf{w}^T \mathbf{w}}{2}} (d\mathbf{w})^\wedge, \quad (\text{C.8})$$

or

$$k^{(m)}(\mathbf{a}, \mathbf{b}) = \frac{1}{2\pi} \|\mathbf{a}\|^m \|\mathbf{b}\|^m J_m(\theta). \quad (\text{C.9})$$

The angle

$$\theta = \arccos\left(\frac{\mathbf{a}^T \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|}\right).$$

$$J_m(\theta) = (-1)^m (\sin \theta)^m \left(\frac{1}{\sin \theta} \frac{\partial}{\partial \theta}\right)^m \left(\frac{\pi - \theta}{\sin \theta}\right); \theta \in [0, \pi] \quad (\text{C.10})$$

$$J_m(-\theta) = J_m(\theta); \theta \in [0, \pi] \quad (\text{C.11})$$

$$J_0(\theta) = \pi - |\theta| \quad (\text{C.12})$$

$$J_1(\theta) = \sin |\theta| + (\pi - |\theta|) \cos \theta \quad (\text{C.13})$$

$$J_2(\theta) = 3 \sin |\theta| \cos \theta + (\pi - |\theta|) (1 + 2 \cos^2 \theta) \quad (\text{C.14})$$