

École polytechnique de Louvain

Investigating Deep Neural Network Internal Clustering and Generalization Properties

Author: **Gilles PEIFFER**

Supervisor: **Christophe DE VLEESCHOUWER**

Readers: **Simon CARBONNELLE, John LEE**

Academic year 2020–2021

Master [120] in Computer Science and Engineering

Abstract

The topic of generalization in deep neural networks has long been one that garners a lot of interest from research groups all around the world [1, 2, 3, 4]. Despite all this interest, however, current knowledge about this topic remains limited. In this thesis, we present a novel way of looking at generalization, through the lens of how internal representations at the neuron level relate to the generalization performance of a deep neural network.

We present three main contributions on this subject. (i) First, we extend some previous results in the literature about the statistical distribution of weights [5], and introduce a novel concept, the EW product, whose distribution properties have an even stronger relation to generalization performance. (ii) Second, we build on results obtained in [6] that show that neurons act as binary classifiers inside the network, and show that the quality of this binary partition is related to generalization performance as well. (iii) Finally, we show that neurons not only act as binary classifiers, but also as pattern discriminators, and that the quality of this discrimination has strong ties to generalization performance.

Acknowledgments

These acknowledgments are surely incomplete, and I apologize in advance to those ignored by these brief words.

To Christophe De Vleeschouwer. Thank you for the time and effort you put into allowing me to work on this topic I find so interesting, and for getting me up to speed as early as you did. I am consistently amazed by the frequency with which you are able to come up with good advice and comments, seemingly at a moment's notice. Repeatedly, I found myself only realizing the insightfulness of your words long after you had said them. I have learned immensely from our near-daily interactions, and hope to continue our fruitful collaboration during my PhD.

To Simon Carbonnelle. Without your preliminary work, this thesis would never have been possible. I am very grateful for the discussions we have had, both related and unrelated to the topic of this thesis, as well as your enthusiasm in sharing your intuitions with me. I would like to thank you as well for the models and data you provided, which allowed me to start working immediately without wasting time on training.

To my thesis committee. To Simon Carbonnelle, for participating in the supervision of my thesis, and for agreeing to read it. To John Lee, for agreeing to read my thesis.

To Anne-Sophie Collin, Brigitte Dupont, François Hubin, and Antoine Vander-schueren, for helping me access the laboratory's remote server, Cassiopee.

To Martin Braquet, for letting me use his template for a UCLouvain master thesis.

To Benoît Legat and Sascha Timme, for introducing me to L^AT_EX and Jupyter notebooks, and for mentoring me in my first experience with applied research.

To the other recipients of the International Lhoist Berghmans Innovation Chair 2019. To Louis Beauloye, Quentin Delhay, Louis Navarre, and Mathieu Simon, for the interesting conversations on our respective thesis topics. One day, SARS-CoV-2 measures will allow us to finally complete our research internships at MIT.

To Gilles Houben, for being a great friend, and for encouraging me to pursue excellence in my studies, but also to take the time to partake in extracurricular activities.

To Liliya Semerikova, for helping me stay disciplined throughout this thesis, and for agreeing to listen to my many rants whenever things did not go as planned. Thank you for the time you spent with me.

To my family. To my dad, for nurturing my interest in mathematics, science, and engineering for as long as I can remember, and for not allowing me to get away

with half-baked ideas and arguments when thinking about these subjects. To my mother, for making sure I did not succumb to the eternal pressure of academia, and for reminding me that there is more to life than academic achievement. To my brother, for joining me on my runs and for always being willing to kick a ball into a net with me. To my sister, for helping me revisit some fundamental concepts in mathematics and computer science. To my grandmother, for her interest in my work and studies.

Thanks to all of you. This work would not have been possible without you.

Contents

1	Introduction	12
1.1	Thesis Goals	12
1.2	General Organization	12
2	Background	14
2.1	General Overview	14
2.2	Gradient Descent with Back-Propagation	16
2.2.1	Cost Function	17
2.2.2	Back-Propagation	18
2.2.3	Gradient Descent	21
2.3	Additional Considerations	23
2.3.1	Parameter Initialization	24
2.3.2	Parameter Norm Penalties	25
2.3.3	Data Augmentation	27
2.3.4	Batch Normalization	27
2.3.5	Dropout	28
2.3.6	Early Stopping	31
2.4	Convolutional Networks	33
2.4.1	The Convolution Operation	33
2.4.2	Intuition Behind Convolutional Networks	34
2.4.3	Pooling Functions	35
2.4.4	Convolution and Pooling as an Infinitely Strong Prior	35
2.4.5	Modifying Convolution for Neural Networks	36
2.4.6	Efficient Convolution Algorithms	39
2.5	Conclusion	39
3	State of the Art and Related Work	40
3.1	Generalization	40
3.1.1	Layer Rotation: a Surprisingly Powerful Indicator of Generalization in Deep Networks?	40

3.1.2	Experimental Study of the Neuron-Level Mechanisms Emerging from Backpropagation	41
3.1.3	Intraclass Clustering: an Implicit Learning Ability that Regularizes DNNs	41
3.1.4	Exploring Generalization in Deep Learning	41
3.1.5	Understanding Deep Learning Requires Rethinking Generalization	42
3.1.6	Asymmetric Valleys: Beyond Sharp and Flat Local Minima .	42
3.1.7	An Investigation into Neural Net Optimization via Hessian Eigenvalue Density	43
3.1.8	The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks	43
3.2	Visual Pattern Identification by Neurons	44
3.2.1	Visualizing and Understanding Convolutional Networks . . .	44
3.2.2	Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps	44
3.2.3	Understanding Neural Networks Through Deep Visualization	44
3.2.4	Compositional Explanations of Neurons	45
3.3	Conclusion	45
4	Methodology and Experimental Setup	46
4.1	Methodology	46
4.2	Experimental Setup	48
4.2.1	Dataset	48
4.2.2	Models	48
4.2.3	Hardware and Software Specifications	52
4.3	Conclusion	52
5	Statistical Analysis of Neuron-Level Representations	54
5.1	Weights	54
5.1.1	Terminology and Notation	55
5.1.2	Relation with Generalization	56
5.1.3	Influence of Hyperparameters	58
5.2	Excitation-Weight Products	58
5.2.1	Motivation	60
5.2.2	Relation with Generalization	61
5.2.3	Influence of Hyperparameters	61
5.3	Conclusion	65

6	Neurons as Binary Classifiers	66
6.1	Terminology and Notation	66
6.1.1	Terminology	66
6.1.2	Metrics to Assess the Binary Partition	67
6.2	Experimental Study of Binary Partition Characteristics and Generalization	69
6.2.1	As Measured Using Silhouette	70
6.2.2	As Measured Using Information Theory	75
6.2.3	As Measured Using Cluster Size	76
6.3	Influence of Number of Significant Components	78
6.4	Conclusion	80
7	Neurons as Pattern Discriminators	81
7.1	Terminology and Notation	81
7.2	Investigation on Synthetic Data	82
7.2.1	Synthetic Data Generation Process and Clustering	82
7.2.2	Synthetic Data Parameter Influence	84
7.3	Empirical Evidence for the Presence of Neuron-Level Patterns	90
7.4	Conclusion	92
8	Future Work	94
8.1	Leveraging Neuron Behavior to Predict Generalization Performance	94
8.2	Pattern-Based Confidence Measures for Classification and Out-of-Distribution Detection	94
8.3	Class-Specific Study of Clustering and Pattern Detection	95
8.4	Learning Dynamics of Good Models	96
8.5	Conclusion	96
9	Conclusion	97
A	Omitted Figures for Chapter 5	99
A.1	Weights	99
A.1.1	Influence of Hyperparameters	99
A.2	Excitation-Weight Products	103
A.2.1	Influence of Hyperparameters	103
B	Omitted Figures for Chapter 6	107
B.1	Experimental Study of Binary Partition Characteristics and Generalization	107
B.1.1	As Measured Using Silhouette	107
B.1.2	As Measured Using Information Theory	124

B.1.3	As Measured Using Cluster Size	140
-------	--	-----

List of Figures

2.1	Example neural network with one hidden layer	15
2.2	Comparison of activation functions	17
2.3	Example of the binary cross-entropy	18
4.1	Experimental methodology of chapter 5	47
4.2	Experimental methodology of chapters 6 and 7	47
4.3	Example VGG-16 variant	50
5.1	Comparison of layer weight distributions across convolutional layers of a randomly chosen network	55
5.2	Relation between peakedness of weight distribution and generaliza- tion performance	57
5.3	Influence of noise regularization on kurtosis of weight distribution .	59
5.4	Influence of effective learning rate on kurtosis of weight distribution	59
5.5	Influence of optimizer on kurtosis of weight distribution	60
5.6	Relation between peakedness of EW product distribution and gener- alization performance	62
5.7	Influence of noise regularization on kurtosis of EW product distribution	63
5.8	Influence of effective learning rate on kurtosis of EW product distri- bution	64
5.9	Influence of optimizer on kurtosis of EW product distribution . . .	64
6.1	Comparison of mean silhouette of activation partition and 2-means clustering with top-1 % magnitude weight components	70
6.2	Influence of noise regularization on mean silhouette for top-1 % magnitude weight components	71
6.3	Influence of architecture on mean silhouette for top-1 % magnitude weight components	72
6.4	Influence of effective learning rate on mean silhouette for top-1 % magnitude weight components	73
6.5	Influence of optimizer on mean silhouette for top-1 % magnitude weight components	74

6.6	Variation of information of activation partition and 2-means clustering with top-1 % magnitude weight components	75
6.7	Influence of effective learning rate on variation of information for top-1 % magnitude weight components	76
6.8	Influence of optimizer on variation of information for top-1 % magnitude weight components	77
6.9	Influence of batch size on variation of information for top-1 % magnitude weight components	77
6.10	Comparison of cluster size of activation partition and 2-means clustering with top-1 % magnitude weight components	78
6.11	Influence of number of significant components on 2-means clusterability	79
7.1	Influence of number of dimensions on pattern detectability	85
7.2	Influence of pattern length on pattern detectability	86
7.3	Influence of number of patterns on pattern detectability	87
7.4	Influence of number of samples on pattern detectability	88
7.5	Influence of pattern separation on pattern detectability	89
7.6	Relation between pattern clusterability and generalization	91
A.1	Influence of architecture on kurtosis of weight distribution	99
A.2	Influence of batch size on kurtosis of weight distribution	100
A.3	Influence of architecture on L^∞ -normalized variance of weight distribution	100
A.4	Influence of noise regularization on L^∞ -normalized variance of weight distribution	101
A.5	Influence of effective learning rate on L^∞ -normalized variance of weight distribution	101
A.6	Influence of batch size on L^∞ -normalized variance of weight distribution	102
A.7	Influence of optimizer on L^∞ -normalized variance of weight distribution	102
A.8	Influence of architecture on kurtosis of EW product distribution . . .	103
A.9	Influence of batch size on kurtosis of EW product distribution . . .	104
A.10	Influence of architecture on L^∞ -normalized variance of EW product distribution	104
A.11	Influence of noise regularization on L^∞ -normalized variance of EW product distribution	105
A.12	Influence of effective learning rate on L^∞ -normalized variance of EW product distribution	105
A.13	Influence of batch size on L^∞ -normalized variance of EW product distribution	106
A.14	Influence of optimizer on L^∞ -normalized variance of EW product distribution	106

B.1	Comparison of mean silhouette of activation partition and 2-means clustering with all weight components	107
B.2	Comparison of mean silhouette of activation partition and 2-means clustering with top-10 % magnitude weight components	108
B.3	Comparison of mean silhouette of activation partition and 2-means clustering with top-5 % magnitude weight components	108
B.4	Comparison of mean silhouette of activation partition and 2-means clustering with top-2 % magnitude weight components	109
B.5	Comparison of mean silhouette of activation partition and 2-means clustering with top-0.5 % magnitude weight components	109
B.6	Influence of batch size on mean silhouette for top-1 % magnitude weight components	110
B.7	Influence of noise regularization on mean silhouette for all weight components	111
B.8	Influence of architecture on mean silhouette for all weight components	111
B.9	Influence of effective learning rate on mean silhouette for all weight components	112
B.10	Influence of optimizer on mean silhouette for all weight components	112
B.11	Influence of batch size on mean silhouette for all weight components	113
B.12	Influence of noise regularization on mean silhouette for top-10 % magnitude weight components	113
B.13	Influence of architecture on mean silhouette for top-10 % magnitude weight components	114
B.14	Influence of effective learning rate on mean silhouette for top-10 % magnitude weight components	114
B.15	Influence of optimizer on mean silhouette for top-10 % magnitude weight components	115
B.16	Influence of batch size on mean silhouette for top-10 % magnitude weight components	115
B.17	Influence of noise regularization on mean silhouette for top-5 % magnitude weight components	116
B.18	Influence of architecture on mean silhouette for top-5 % magnitude weight components	116
B.19	Influence of effective learning rate on mean silhouette for top-5 % magnitude weight components	117
B.20	Influence of optimizer on mean silhouette for top-5 % magnitude weight components	117
B.21	Influence of batch size on mean silhouette for top-5 % magnitude weight components	118

B.22	Influence of noise regularization on mean silhouette for top-2 % magnitude weight components	118
B.23	Influence of architecture on mean silhouette for top-2 % magnitude weight components	119
B.24	Influence of effective learning rate on mean silhouette for top-2 % magnitude weight components	119
B.25	Influence of optimizer on mean silhouette for top-2 % magnitude weight components	120
B.26	Influence of batch size on mean silhouette for top-2 % magnitude weight components	120
B.27	Influence of noise regularization on mean silhouette for top-0.5 % magnitude weight components	121
B.28	Influence of architecture on mean silhouette for top-0.5 % magnitude weight components	121
B.29	Influence of effective learning rate on mean silhouette for top-0.5 % magnitude weight components	122
B.30	Influence of optimizer on mean silhouette for top-0.5 % magnitude weight components	122
B.31	Influence of batch size on mean silhouette for top-0.5 % magnitude weight components	123
B.32	Variation of information of activation partition and 2-means clustering with all weight components	124
B.33	Variation of information of activation partition and 2-means clustering with top-10 % magnitude weight components	125
B.34	Variation of information of activation partition and 2-means clustering with top-5 % magnitude weight components	125
B.35	Variation of information of activation partition and 2-means clustering with top-2 % magnitude weight components	126
B.36	Variation of information of activation partition and 2-means clustering with top-0.5 % magnitude weight components	126
B.37	Influence of noise regularization on variation of information for top-1 % magnitude weight components	127
B.38	Influence of architecture on variation of information for top-1 % magnitude weight components	127
B.39	Influence of noise regularization on variation of information for all weight components	128
B.40	Influence of architecture on variation of information for all weight components	128
B.41	Influence of effective learning rate on variation of information for all weight components	129

B.42	Influence of optimizer on variation of information for all weight components	129
B.43	Influence of batch size on variation of information for all weight components	130
B.44	Influence of noise regularization on variation of information for top-10 % magnitude weight components	130
B.45	Influence of architecture on variation of information for top-10 % magnitude weight components	131
B.46	Influence of effective learning rate on variation of information for top-10 % magnitude weight components	131
B.47	Influence of optimizer on variation of information for top-10 % magnitude weight components	132
B.48	Influence of batch size on variation of information for top-10 % magnitude weight components	132
B.49	Influence of noise regularization on variation of information for top-5 % magnitude weight components	133
B.50	Influence of architecture on variation of information for top-5 % magnitude weight components	133
B.51	Influence of effective learning rate on variation of information for top-5 % magnitude weight components	134
B.52	Influence of optimizer on variation of information for top-5 % magnitude weight components	134
B.53	Influence of batch size on variation of information for top-5 % magnitude weight components	135
B.54	Influence of noise regularization on variation of information for top-2 % magnitude weight components	135
B.55	Influence of architecture on variation of information for top-2 % magnitude weight components	136
B.56	Influence of effective learning rate on variation of information for top-2 % magnitude weight components	136
B.57	Influence of optimizer on variation of information for top-2 % magnitude weight components	137
B.58	Influence of batch size on variation of information for top-2 % magnitude weight components	137
B.59	Influence of noise regularization on variation of information for top-0.5 % magnitude weight components	138
B.60	Influence of architecture on variation of information for top-0.5 % magnitude weight components	138
B.61	Influence of effective learning rate on variation of information for top-0.5 % magnitude weight components	139

B.62	Influence of optimizer on variation of information for top-0.5 % magnitude weight components	139
B.63	Influence of batch size on variation of information for top-0.5 % magnitude weight components	140
B.64	Comparison of cluster size of activation partition and 2-means clus- tering with all weight components	140
B.65	Comparison of cluster size of activation partition and 2-means clus- tering with top-10 % magnitude weight components	141
B.66	Comparison of cluster size of activation partition and 2-means clus- tering with top-5 % magnitude weight components	141
B.67	Comparison of cluster size of activation partition and 2-means clus- tering with top-2 % magnitude weight components	142
B.68	Comparison of cluster size of activation partition and 2-means clus- tering with top-0.5 % magnitude weight components	142

List of Tables

4.1	VGG model configurations	49
4.2	Model hyperparameter choices	51
7.1	Parameters for synthetic data	83

Chapter 1

Introduction

In this chapter, we detail the global goals and organization of this thesis, as well as giving some insight into what motivated each investigation.

1.1 Thesis Goals

Our overarching goal with this thesis is to understand more about how internal (i.e. at the neuron level) representations of data can be used to understand and predict good generalization performance (i.e. accuracy on an unseen dataset, defined more formally in chapter 2) in deep neural networks, which are some of the most promising tools machine learning and artificial intelligence research has produced to date, but whose true behavior remains widely misunderstood. More precisely, our work tries to answer two questions related to internal representations and generalization performance:

1. Do neurons of models with good generalization performance partition/binarize the internal representation space better than those of worse-generalizing models?
2. Do neurons discriminate between specific patterns in the internal representation of input samples, and is this pattern detection ability related to generalization performance?

1.2 General Organization

This document is organized as follows.

Chapter 2 contains some fundamentals of deep learning and covers some more advanced topics we believe are important to guarantee the reader’s full comprehension of the material discussed in this thesis.

In chapter 3, we look at the state of the art in the deep learning space, and discuss some related work that helped us build our intuition about the topic at hand. We show that understanding generalization is arguably the hottest topic in current deep learning research, as well as motivating our decision to come at it from the until now fairly understudied angle of neuron-level behavior.

Chapter 4 introduces the methodology and experimental setup we use throughout this thesis, and we argue why this choice is sufficient to produce compelling evidence about the link between neuron-level representations and generalization performance.

In chapter 5, we study how the distributions of layer weights and excitation-weight products (two notions we define more formally) are shaped, and through statistical analysis of these distributions, show that only a subset of them is relevant for our further investigations. This subset, which we call the “significant components,” is central to the work done in later chapters, and we consistently refer back to it.

Chapter 6 builds on an observation first made in [6] that neurons in deep neural networks act as binary classifiers as a consequence of the way in which neural networks are trained. We study this emergent classification through the use of several clustering metrics, and see how these metrics are related to model generalization performance. This chapter thus answers question 1, which we asked earlier.

Chapter 7 follows the same methodology as the previous chapter and considers the same clustering tools and metrics to a large extent, but studies whether models with good generalization discriminate between specific patterns in the internal representation, thus answering question 2 we asked earlier.

Chapter 8 considers possible avenues (theoretical and practical) for future research based on the results we obtained in previous chapters. We also bring up some remaining questions we would like to answer on this topic.

Chapter 9 contains a conclusion and recapitulates on the work we did, underlining our main results and assessing whether questions 1 and 2 have been answered satisfactorily.

Finally, appendices A and B contain some figures that were omitted from the main body of the text for reasons of brevity.

Chapter 2

Background

This chapter gives an overview of the theory of artificial neural networks, in order to bring the reader up to speed and to ensure the rest of this thesis is comprehensible. We start by looking at deep learning and neural networks in a general sense in section 2.1. Next, we review how deep networks learn, in section 2.2. We give some additional considerations and talk about regularization in section 2.3, and take a closer look at convolutional networks, which are the type of network we use in this thesis, in section 2.4. Throughout this chapter and the rest of this manuscript, we use the notation introduced in [7].

2.1 General Overview

An *artificial neural network* (NN) is a machine learning model that can be used to approximate a function f^* . For example, a network used for classification defines a mapping $\mathbf{y} = f(\mathbf{x}; \boldsymbol{\theta})$, and tries to learn the value of the parameters $\boldsymbol{\theta}$ such that $f(\cdot; \boldsymbol{\theta})$ is the best function approximation of f^* . The *generalization performance* of this network is then a measure of its quality on a dataset it had not seen during training. In the case of classification, generalization performance usually refers to the percentage of correctly classified test samples. While a theoretical result known as the “no free lunch” (NFL) theorem [8, 9] exists stating that no learning algorithm (such as neural networks) is better than random guessing over all possible functions to be learned, neural networks tend to perform extremely well in real-world tasks.

These models are called *networks* because they can be thought of as a composition of functions: $f(\mathbf{x}) = f^{(l)}(\dots f^{(2)}(f^{(1)}(\mathbf{x})) \dots)$; each function $f^{(i)}$ of this composition then corresponds to a *layer* of the network, and the *depth* of the model refers to the total length of this chain, l . As their name implies, *deep neural networks* (DNN) are thus networks with a comparatively high number of layers. An illustration of an artificial neural network is given in figure 2.1.

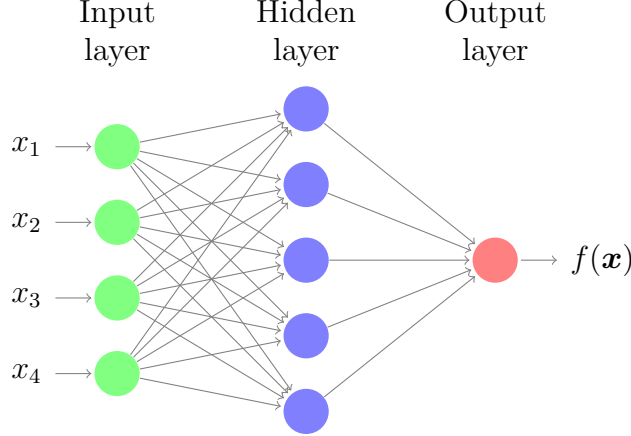


Figure 2.1: Example neural network with one hidden layer.

To understand these networks intuitively, we use the approach of [7]. Consider a linear model. While linear models are easy to work with, they are also limited to modeling linear functions. To allow linear models to represent nonlinear functions of $\mathbf{x}$, we can map the input $\mathbf{x} \mapsto \phi(\mathbf{x})$ before applying the linear model to this mapped representation. ϕ , which is a *nonlinear transformation*, then provides a set of *features* describing $\mathbf{x}$.

Deep learning tries to learn the optimal mapping ϕ . In fact, this is one of the places where it differs from traditional machine learning: the latter usually relies on manually chosen features, whereas deep learning finds them on its own. In practice, we often parametrize the representation as $\phi(\mathbf{x}; \boldsymbol{\theta})$. The optimal parameters $\boldsymbol{\theta}$ can then be learned by an optimization algorithm. The majority of neural nets use a parametrized affine transformation, followed by a fixed nonlinear *activation function*, as mapping in each layer. The outputs $\mathbf{a}$ of a hidden layer with inputs $\mathbf{x}$ are then given by

$$\mathbf{a} = \sigma(\underbrace{\mathbf{W}^T \mathbf{x} + \mathbf{b}}_z), \quad (2.1)$$

where σ is the activation function (applied element-wise; we use the vector and scalar versions interchangeably), $\mathbf{W}$ are the learned weights, and $\mathbf{b}$ are the learned biases.

Several activation function choices are possible, though in practice hidden layers almost always use the *rectified linear unit* (ReLU) [10, 11, 12], $\sigma: \mathbb{R} \rightarrow \mathbb{R}_{\geq 0}$, defined by the formula

$$\sigma(z) = \max\{0, z\} = \begin{cases} z, & \text{if } z > 0, \\ 0, & \text{otherwise.} \end{cases} \quad (2.2)$$

This also means that (where σ is differentiable)

$$\frac{d\sigma(z)}{dz} = \sigma'(z) = \begin{cases} 1, & \text{if } z > 0, \\ 0, & \text{otherwise.} \end{cases} \quad (2.3)$$

Another activation function worth mentioning is the *sigmoid* function $\sigma: \mathbb{R} \rightarrow [0, 1]$, defined by [13, 14] as

$$\sigma(z) = \frac{1}{1 + \exp(-z)}. \quad (2.4)$$

Its derivative is then given by

$$\frac{d\sigma(z)}{dz} = \sigma'(z) = \sigma(z)(1 - \sigma(z)). \quad (2.5)$$

Finally, we also mention the *softmax* function $\sigma: \mathbb{R}^K \rightarrow [0, 1]^K$, which is often used in the last layer of a neural network-based classifier [15, 16], and is given by the formula

$$\sigma(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^K \exp(z_j)}, \quad i = 1, \dots, K. \quad (2.6)$$

The reason for using this activation function in particular in the last layer is that it normalizes its input into what can be interpreted as a probability distribution consisting of K probabilities proportional to the exponential of the input numbers (and in classification, can be used to represent the probability of assigning each label to a given sample) [17]. From a deep learning perspective, this interpretation can be considered valid because of the universal approximation theorems [18, 19, 20, 21, 22, 23]. However, it has been argued in [24] that certain improvements such as those discussed in section 2.3 can affect whether the softmax output can truly be interpreted as a probability distribution.

These various activation function choices are shown in figure 2.2.

2.2 Gradient Descent with Back-Propagation

In order for our neural network to learn a set of parameters that produces good results, we must equip it with a *cost function* J that, in combination with *gradient descent*, will improve the performance of the network on its intended task.¹ In practice, this cost function is usually made up of a *loss function* L , which depends on the output $\hat{\mathbf{y}}$ and the target $\mathbf{y}$, and a *regularizer* $\Omega(\boldsymbol{\theta})$, designed to penalize overly complex models, which usually result in worse generalization performance. Note that $\boldsymbol{\theta}$ contains all model parameters (weights and biases).

¹Intuitively, one would want to take e.g. classification error as loss function, but this “obvious” choice cannot always be optimized efficiently [25].

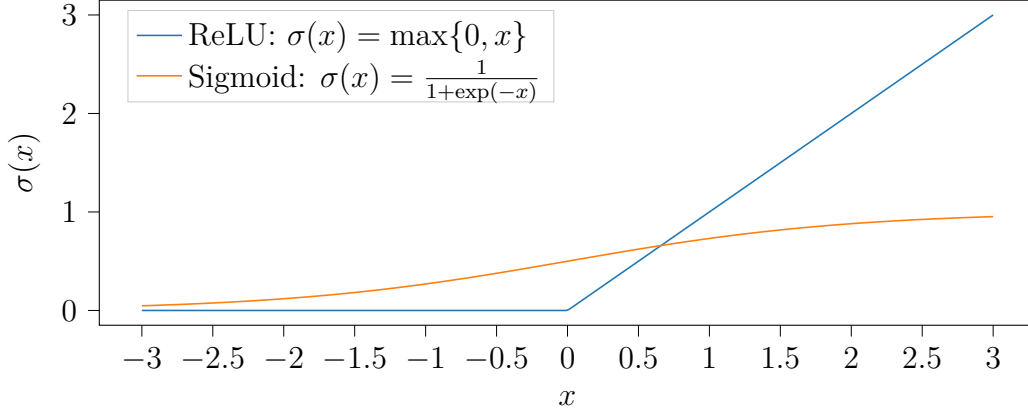


Figure 2.2: Comparison of activation functions.

2.2.1 Cost Function

To intuitively arrive at the cost functions that are used in practice, let us start by considering a single-input neuron, with a sigmoid activation function [26]. We also forget about regularization for now. In other words, the output of our model is given by

$$\hat{y} = f(x; w, b) = \sigma(wx + b) = \frac{1}{1 + \exp(-(wx + b))}. \quad (2.7)$$

Reasoning about the desired behavior of the loss function, one arrives at the conclusion that we want its gradient to be large when the prediction is bad. Indeed, this is a key principle of loss function design. Thinking back to our example single-input sigmoid neuron in (2.7), and concentrating only on the bias term for simplicity, this means that we want that

$$\frac{\partial L(\hat{y}, y)}{\partial b} \propto \hat{y} - y. \quad (2.8)$$

For simplicity, we can consider this proportionality to be equality. We also know that

$$\frac{\partial L(\hat{y}, y)}{\partial b} = \frac{\partial L(\hat{y}, y)}{\partial \hat{y}} \sigma'(wx + b) = \frac{\partial L(\hat{y}, y)}{\partial \hat{y}} \sigma(wx + b) (1 - \sigma(wx + b)) \quad (2.9)$$

$$\iff \frac{\partial L(\hat{y}, y)}{\partial \hat{y}} = \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})} \quad (2.10)$$

$$\iff L(\hat{y}, y) = \int \frac{v - y}{v(1 - v)} dv = -(y \ln \hat{y} + (1 - y) \ln(1 - \hat{y})) + c. \quad (2.11)$$

This function is called the *cross-entropy*, and is a very popular choice in the literature [27]. Its shape is shown in figure 2.3. In the general case of n samples $\mathbf{x}^{(i)}$ and K output neurons, the cross-entropy is given by

$$L(\hat{\mathbf{Y}}, \mathbf{Y}) = -\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^K \left(y_j^{(i)} \ln \hat{y}_j^{(i)} + (1 - y_j^{(i)}) \ln(1 - \hat{y}_j^{(i)}) \right), \quad (2.12)$$

for $i = 1, \dots, n$. For brevity of notation, we write $\hat{\mathbf{Y}}$ and $\mathbf{Y}$ to denote matrices containing the model outputs and targets for each sample.

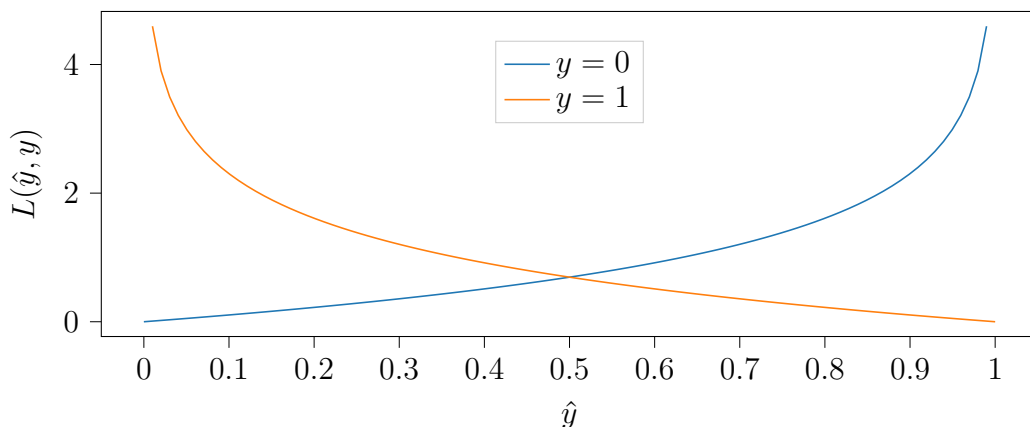


Figure 2.3: Example of the binary cross-entropy.

2.2.2 Back-Propagation

In order to compute the gradient of the cost function, we apply a procedure called *back-propagation* (BP, or backprop), based on repeated application of the chain rule of derivation [7, 26, 28]. Again, we consider only the loss function for now. Let us start by establishing some definitions:

- $z_i^{(k)}$ is the *pre-activation* of neuron i in the layer k ($k \in \{1, \dots, l\}$);
- $a_i^{(k)} = \sigma(z_i^{(k)})$ is the *activation* of neuron i in layer k ($k \in \{1, \dots, l\}$);
- $W_{i,j}^{(k)}$ is the *weight* of the link between neuron i in layer $k - 1$ and neuron j in layer k ($k \in \{1, \dots, l\}$); and
- $b_i^{(k)}$ is the *bias* of neuron i in layer k ($k \in \{1, \dots, l\}$).

Note that an activation coming out of layer $k - 1$ is also sometimes called an *excitation* entering layer k .

We also use the following notation:

- $\mathbf{z}^{(k)}$, $\mathbf{a}^{(k)}$, and $\mathbf{b}^{(k)}$ denote vectors whose i th components are $z_i^{(k)}$, $a_i^{(k)}$, and $b_i^{(k)}$ respectively;
- $\mathbf{W}^{(k)}$ denotes a matrix whose entry on row i , column j is $W_{i,j}^{(k)}$: $\mathbf{z}^{(k)} = \mathbf{W}^{(k)}\mathbf{a}^{(k-1)} + \mathbf{b}^{(k)}$;
- $\odot$ denotes the Hadamard (or element-wise) product;
- $(\nabla_v \sigma)_i = \frac{\partial \sigma}{\partial v_i}$;
- Functions of vectors are applied element-wise.

In order to update the network weights and biases, we need $\frac{\partial L(\hat{\mathbf{y}}, \mathbf{y})}{\partial W_{i,j}^{(k)}}$ and $\frac{\partial L(\hat{\mathbf{y}}, \mathbf{y})}{\partial b_i^{(k)}}$. Applying the chain rule of derivation, we get

$$\frac{\partial L(\hat{\mathbf{y}}, \mathbf{y})}{\partial b_i^{(k)}} = \frac{\partial L(\hat{\mathbf{y}}, \mathbf{y})}{\partial z_i^{(k)}} \triangleq \delta_i^{(k)}, \quad (2.13)$$

$$\frac{\partial L(\hat{\mathbf{y}}, \mathbf{y})}{\partial W_{i,j}^{(k)}} = \frac{\partial L(\hat{\mathbf{y}}, \mathbf{y})}{\partial z_i^{(k)}} a_j^{(k-1)} = \delta_i^{(k)} a_j^{(k-1)}. \quad (2.14)$$

We thus compute

$$\frac{\partial L(\hat{\mathbf{y}}, \mathbf{y})}{\partial z_i^{(k)}} = \sigma'(z_i^{(k)}) \frac{\partial L(\hat{\mathbf{y}}, \mathbf{y})}{\partial a_i^{(k)}} \quad (2.15)$$

$$= \begin{cases} \sigma'(z_i^{(k)}) \sum_{i'} \left(\frac{\partial L(\hat{\mathbf{y}}, \mathbf{y})}{\partial z_{i'}^{(k+1)}} W_{i',j}^{(k+1)} \right), & \text{if } k < l, \\ \sigma'(z_i^{(l)}) \frac{\partial L(\hat{\mathbf{y}}, \mathbf{y})}{\partial a_i^{(l)}}, & \text{if } k = l. \end{cases} \quad (2.16)$$

In vector notation, we thus write

$$\boldsymbol{\delta}^{(k)} = \begin{cases} \sigma'(\mathbf{z}^{(k)}) \odot \left(\mathbf{W}^{(k+1)\top} \boldsymbol{\delta}^{(k+1)} \right), & \text{if } k < l, \\ \sigma'(\mathbf{z}^{(l)}) \odot \nabla_{\hat{\mathbf{y}}} L(\hat{\mathbf{y}}, \mathbf{y}), & \text{if } k = l. \end{cases} \quad (2.17)$$

In summary, the forward and backward passes through a deep neural network can be computed by algorithms 2.1 and 2.2, respectively [7].

Note that these algorithms present a simplified view, and that more information on these simplifications is given in section 2.3; we have nonetheless added the regularization term, to generalize the presentation given above, which considered only the loss function.

Algorithm 2.1 Forward pass through a typical deep neural network.

```

1: procedure FORWARD( $l, \mathbf{W}^{(i)}, \mathbf{b}^{(i)}, \mathbf{x}, \mathbf{y}$ )     $\triangleright$  the network depth, the weight
   matrices and bias parameters at each layer, the input, and the target output
2:    $\mathbf{a}^{(0)} \leftarrow \mathbf{x}$                                  $\triangleright$  define zeroth layer
3:   for  $k \leftarrow 1, \dots, l$  do
4:      $\mathbf{z}^{(k)} \leftarrow \mathbf{W}^{(k)} \mathbf{a}^{(k-1)} + \mathbf{b}^{(k)}$      $\triangleright$  compute pre-activations
5:      $\mathbf{a}^{(k)} \leftarrow \sigma(\mathbf{z}^{(k)})$                  $\triangleright$  apply activation function
6:   end for
7:    $\hat{\mathbf{y}} \leftarrow \mathbf{a}^{(l)}$                              $\triangleright$  compute network output
8:    $J \leftarrow L(\hat{\mathbf{y}}, \mathbf{y}) + \lambda \Omega(\boldsymbol{\theta})$      $\triangleright$  compute cost function value
9: end procedure

```

Algorithm 2.2 Back-propagation through the deep neural network of algorithm 2.1.

```

1: procedure BACKPROP( $l, \mathbf{W}^{(i)}, \mathbf{b}^{(i)}, \mathbf{x}, \mathbf{y}$ )     $\triangleright$  the network depth, the weight
   matrices and bias parameters at each layer, the input, and the target output
2:    $\boldsymbol{\delta} \leftarrow \nabla_{\hat{\mathbf{y}}} J = \nabla_{\hat{\mathbf{y}}} L(\hat{\mathbf{y}}, \mathbf{y})$      $\triangleright$  compute gradient on output layer
3:   for  $k \leftarrow l, l-1, \dots, 1$  do
4:      $\boldsymbol{\delta} \leftarrow \nabla_{\mathbf{a}^{(k)}} J = \boldsymbol{\delta} \odot \sigma'(\mathbf{z}^{(k)})$      $\triangleright$  convert gradient on layer output into
       gradient on pre-activation
5:      $\nabla_{\mathbf{b}^{(k)}} J \leftarrow \boldsymbol{\delta} + \lambda \nabla_{\mathbf{b}^{(k)}} \Omega(\boldsymbol{\theta})$      $\triangleright$  compute gradient on biases
6:      $\nabla_{\mathbf{W}^{(k)}} J \leftarrow \boldsymbol{\delta} \mathbf{a}^{(k-1)\top} + \lambda \nabla_{\mathbf{W}^{(k)}} \Omega(\boldsymbol{\theta})$      $\triangleright$  compute gradient on weights
7:      $\boldsymbol{\delta} \leftarrow \nabla_{\mathbf{a}^{(k-1)}} J = \mathbf{W}^{(k)\top} \boldsymbol{\delta}$      $\triangleright$  Propagate gradients w.r.t. excitations
8:   end for
9: end procedure

```

2.2.3 Gradient Descent

Note that our cross-entropy loss function (and thus the resulting cost function, though for now we restrict ourselves to studying the case without regularization) is nonconvex, and thus has to be brought to a very low value by iterative, gradient-descent based procedures rather than solving the problem with convex optimization algorithms, providing global convergence guarantees. The canonical example in deep learning of such a method is stochastic gradient descent. Indeed, *stochastic gradient descent* (SGD) is sensitive to initialization parameters when applied to nonconvex cost functions, and provides no such convergence guarantee [7].

2.2.3.1 Benefits of Small Batches

SGD is different from traditional gradient descent in that it uses only a subset of the training set at each step. To see why this choice makes sense, consider that minimizing the cross-entropy corresponds to maximizing the likelihood [7, 27]. We can thus write

$$\boldsymbol{\theta}_{\text{ML}} = \arg \max_{\boldsymbol{\theta}} \sum_{i=1}^m \ln p_{\text{model}}(\mathbf{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}). \quad (2.18)$$

Maximizing this sum is equivalent to maximizing the expectation over the empirical distribution defined by the training set [7], hence we could use the following objective function:²

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}} \ln p_{\text{model}}(\mathbf{x}, y, \boldsymbol{\theta}). \quad (2.19)$$

The gradient of the objective function thus has the same property:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x}, y \sim \hat{p}_{\text{data}}} \nabla_{\boldsymbol{\theta}} \ln p_{\text{model}}(\mathbf{x}, y, \boldsymbol{\theta}). \quad (2.20)$$

Computing this expectation is linear in the size of the dataset, n . However, the standard error of the mean, $\sigma/\sqrt{n}$, is inversely proportional to the square root of the size of the dataset. In other words, using a small *batch* to approximate the value of the gradient is much faster than computing the exact gradient using the entire dataset, but is only slightly less accurate. We refer to passes of the entire dataset as *epochs*; after each epoch, samples are usually shuffled to avoid cyclical repetitions. In practice, most optimization algorithms converge faster (in terms of operations) when using small batches [7].

This idea is behind many of the *minibatch* algorithms in the literature, with SGD being arguably the most common one in deep learning. We also note the

²The notation here does not reflect the change in optimization goal (from minimization to maximization) we make for simplicity, since it does not affect the argument. One could trivially maintain the minimization goal by changing signs.

inherent regularization that small batches bring [29]. It has been argued that this is a consequence of the noise they add to the learning process [7, 30]. The influence of batch size on generalization performance is discussed in [31].

2.2.3.2 Stochastic Gradient Descent

Stochastic gradient descent (SGD) is an optimization algorithm used in deep learning. As discussed previously, it achieves considerable speed up over traditional gradient descent by only considering a subset of size m (a minibatch) of the training data to approximate the gradient of the loss function at each step. Because this sampling is a source of noise, the *learning rate* (involved in the step size of the gradient steps) of SGD should be decreased over time [7, 32]. We denote η_k the learning rate at iteration k . A sufficient condition to guarantee convergence of SGD is that [7, 33, 34]

$$\sum_{k=1}^{\infty} \eta_k = \infty, \quad (2.21)$$

$$\sum_{k=1}^{\infty} \eta_k^2 < \infty. \quad (2.22)$$

The minima found by SGD are influenced by three factors: the learning rate, batch size and gradient covariance [35]. In [36], the authors argue that bad global minima of the loss landscape exist, and can be reached by SGD. However, regularization techniques such as those presented in section 2.3 can help prevent this from happening.

Algorithm 2.3 gives the general shape of an SGD update [7].

Algorithm 2.3 Stochastic gradient descent.

```

1: procedure SGD( $\theta, \eta_1, \eta_2, \dots$ )  $\triangleright$  the initial parameters, and a learning rate
   schedule
2:    $k \leftarrow 1$ 
3:   while stopping criterion not met do
4:     sample minibatch  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  with corresponding targets  $\mathbf{y}^{(i)}$ 
5:      $\hat{\mathbf{g}} \leftarrow \frac{1}{m} \sum_{i=1}^m \nabla_{\theta} L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$   $\triangleright$  compute gradient estimate
6:      $\theta \leftarrow \theta - \eta_k \hat{\mathbf{g}}$   $\triangleright$  apply update
7:      $k \leftarrow k + 1$ 
8:   end while
9: end procedure

```

2.2.3.3 Momentum

In order to speed up SGD, the method of *momentum* [37] accumulates an exponentially decaying moving average of past gradients, and continues to move in their direction [7], by introducing a variable $\mathbf{v}$ playing the role of velocity. The rate of decay is given by a hyperparameter $\alpha \in [0, 1)$. The update rule is given by

$$\mathbf{v} \leftarrow \alpha \mathbf{v} - \eta \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^m L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), \mathbf{y}^{(i)}) \right), \quad (2.23)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}. \quad (2.24)$$

Algorithm 2.4 shows a version of algorithm 2.3 updated with the method of momentum [7]. Note that other forms of momentum updates also exist, such as Nesterov’s accelerated gradient method [38, 39, 40].

Algorithm 2.4 Stochastic gradient descent with momentum.

```

1: procedure SGDMOMENTUM( $\boldsymbol{\theta}, \mathbf{v}, \eta, \alpha$ )  $\triangleright$  the initial parameters, the initial
   velocity, the learning rate, and the momentum parameter
2:   while stopping criterion not met do
3:     sample minibatch  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  with corresponding targets  $\mathbf{y}^{(i)}$ 
4:      $\hat{\mathbf{g}} \leftarrow \frac{1}{m} \sum_{i=1}^m \nabla_{\boldsymbol{\theta}} L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), \mathbf{y}^{(i)})$   $\triangleright$  compute gradient estimate
5:      $\mathbf{v} \leftarrow \alpha \mathbf{v} - \eta \hat{\mathbf{g}}$   $\triangleright$  compute velocity update
6:      $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{v}$   $\triangleright$  apply update
7:   end while
8: end procedure

```

2.2.3.4 Adam

Adam [41] is an adaptive learning rate optimization algorithm whose name derives from “Adaptive Moment Estimation.” It is generally regarded as robust to the choice of hyperparameters. Algorithm 2.5 presents a more formal view of the method [7]. All operations are applied element-wise. It has been argued in [42] that adaptive optimization methods such as Adam generalize worse than stochastic gradient descent.

2.3 Additional Considerations

The previous sections only give a brief overview of the fundamentals of neural network training. We also mention some more advanced tools that were particularly

Algorithm 2.5 Adam.

```
1: procedure ADAM( $\theta, \eta, \rho_1, \rho_2, \delta$ )  $\triangleright$  the initial parameters, the step size, the
   exponential decay rates for moment estimates, and a constant for numerical
   stabilization
2:    $\mathbf{s} \leftarrow \mathbf{0}, \mathbf{r} \leftarrow \mathbf{0}$   $\triangleright$  initialize first and second moment variables
3:    $t \leftarrow 0$   $\triangleright$  initialize time step
4:   while stopping criterion not met do
5:     sample minibatch  $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$  with corresponding targets  $\mathbf{y}^{(i)}$ 
6:      $\hat{\mathbf{g}} \leftarrow \frac{1}{m} \sum_{i=1}^m \nabla_{\theta} L(f(\mathbf{x}^{(i)}; \theta), \mathbf{y}^{(i)})$   $\triangleright$  compute gradient estimate
7:      $t \leftarrow t + 1$ 
8:      $\mathbf{s} \leftarrow \rho_1 \mathbf{s} + (1 - \rho_1) \hat{\mathbf{g}}$   $\triangleright$  update biased first moment estimate
9:      $\mathbf{r} \leftarrow \rho_2 \mathbf{r} + (1 - \rho_2) \hat{\mathbf{g}} \odot \hat{\mathbf{g}}$   $\triangleright$  update biased second moment estimate
10:     $\hat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \rho_1^t}$   $\triangleright$  correct bias in first moment estimate
11:     $\hat{\mathbf{r}} \leftarrow \frac{\mathbf{r}}{1 - \rho_2^t}$   $\triangleright$  correct bias in second moment estimate
12:     $\Delta \theta \leftarrow -\eta \frac{\hat{\mathbf{s}}}{\sqrt{\hat{\mathbf{r}} + \delta}}$   $\triangleright$  compute update
13:     $\theta \leftarrow \theta + \Delta \theta$   $\triangleright$  apply update
14:   end while
15: end procedure
```

useful for the work done in this thesis. Indeed, up until now, we have only considered one side of the learning process of a deep neural network, namely the performance on the training set. In practice however, we are also concerned with ensuring good generalization performance on the test set, eventually with some cost to the training performance of the network. Indeed, we want to avoid *over-* or *underfitting* the network. One common definition of *regularization* is “any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error [7].” We thus give a very brief overview of regularization techniques that were used in this thesis: data augmentation, L^2 regularization, batch normalization, dropout, and early stopping. Note that implicit forms of regularization emerge from interactions between the neural network architecture, optimization, and the inherent structure of the data [43].

2.3.1 Parameter Initialization

Parameter initialization refers to the way in which the various parameters θ are fixed at the start of neural network training. This part of training is mostly heuristic in nature.

It has been argued [7] that the only property of parameter initialization that is known with complete certainty is that one needs to “break symmetry” between

different units, i.e., neurons with the same inputs and activation function. In practice, this is achieved by drawing initial weights from a Gaussian or uniform distribution [7], and by setting the biases and other extra parameters to heuristically chosen constants.

In order to choose the scale of the initial weight distribution, several heuristics exist. Larger weights favor optimization by increasing the symmetry-breaking property of the initialization, but can result in exploding values during forward or back-propagation, negatively affecting regularization. Considering initialization of $\boldsymbol{\theta}$ to $\boldsymbol{\theta}_0$ as being similar to imposing a Gaussian prior $p(\boldsymbol{\theta})$ with mean $\boldsymbol{\theta}_0$, it makes sense to choose $\boldsymbol{\theta}_0$ to be near 0. This prior says it is more likely that units do not interact than that they do interact. A larger value would specify which units should interact with each other, and how they should interact [7].

Several more detailed heuristic strategies for initialization exist in the literature, such as [44, 45, 46, 47].

2.3.2 Parameter Norm Penalties

Parameter norm penalty regularization works by adding a parameter norm penalty $\Omega(\boldsymbol{\theta})$ to the objective function J , to obtain a *regularized objective function* $\tilde{J}$ [7]:

$$\tilde{J}(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) = J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y}) + \alpha\Omega(\boldsymbol{\theta}), \quad (2.25)$$

where $\alpha \in [0, \infty)$ is a hyperparameter weighting the relative contribution of Ω to $\tilde{J}$ [7]. It is customary to only penalize the weights of the affine transformation at each layer, and leave the biases unregularized, as they require fewer data to fit correctly, and regularizing them could introduce underfitting [7]. We thus use $\mathbf{w}$ to denote the vector containing all weights affected by a norm penalty in what follows, while $\boldsymbol{\theta}$ refers to the set of all parameters.

Perhaps the most well-known form of parameter norm penalty regularization in deep learning (and the only one we will cover in any depth in this thesis) is *L^2 -norm parameter regularization*, also known as *weight decay*. It works by adding a regularization term $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\mathbf{w}\|_2^2$ to the objective function.

The regularized objective function is then given by

$$\tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \frac{\alpha}{2}\mathbf{w}^\top \mathbf{w} + J(\mathbf{w}; \mathbf{X}, \mathbf{y}), \quad (2.26)$$

with its gradient being

$$\nabla_{\mathbf{w}} \tilde{J}(\mathbf{w}; \mathbf{X}, \mathbf{y}) = \alpha\mathbf{w} + \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y}). \quad (2.27)$$

The weight update is then given by

$$\mathbf{w} \leftarrow (1 - \eta\alpha)\mathbf{w} - \eta\nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{X}, \mathbf{y}). \quad (2.28)$$

This shows that the weight decay term multiplicatively shrinks the weight vector by a constant factor on each step [7].

To garner more insight as to how this affects the entire training process, we consider a quadratic approximation to the objective function in the neighborhood of $\mathbf{w}^* \triangleq \arg \min_{\mathbf{w}} J(\mathbf{w})$. This approximation is given by

$$\hat{J}(\boldsymbol{\theta}) = J(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^\top \mathbf{H}(\mathbf{w} - \mathbf{w}^*), \quad (2.29)$$

where $\mathbf{H}$ is the Hessian matrix of J with respect to $\mathbf{w}$ evaluated at $\mathbf{w}^*$. The first-order term is omitted by virtue of $\mathbf{w}^*$ being a minimum. This also allows us to conclude that $\mathbf{H}$ is positive semi-definite: $\mathbf{H} \succeq 0$.

This approximation has a minimum where its gradient

$$\nabla_{\mathbf{w}} \hat{J}(\mathbf{w}) = \mathbf{H}(\mathbf{w} - \mathbf{w}^*) \quad (2.30)$$

is $\mathbf{0}$.

Adding the weight decay gradient, we can solve for the minimum of the regularized version of $\hat{J}$, $\tilde{\mathbf{w}}$.

$$\alpha \tilde{\mathbf{w}} + \mathbf{H}(\tilde{\mathbf{w}} - \mathbf{w}^*) = \mathbf{0} \quad (2.31)$$

$$\iff (\mathbf{H} + \alpha \mathbf{I}) \tilde{\mathbf{w}} = \mathbf{H} \mathbf{w}^* \quad (2.32)$$

$$\iff \tilde{\mathbf{w}} = (\mathbf{H} + \alpha \mathbf{I})^{-1} \mathbf{H} \mathbf{w}^*. \quad (2.33)$$

As α approaches 0, we get closer to the unregularized case, and $\tilde{\mathbf{w}}$ approaches $\mathbf{w}^*$. On the other hand, as α grows, we observe the following. Because $\mathbf{H}$ is real and symmetric, we can compute its eigendecomposition $\mathbf{H} = \mathbf{Q} \boldsymbol{\Lambda} \mathbf{Q}^\top$, with $\boldsymbol{\Lambda} = \text{diag}(\lambda_i)$ a diagonal matrix containing the eigenvalues of $\mathbf{H}$ and $\mathbf{Q}$ an orthonormal matrix containing its eigenvectors. Applying this to (2.31), we get [7]

$$\tilde{\mathbf{w}} = (\mathbf{Q} \boldsymbol{\Lambda} \mathbf{Q}^\top + \alpha \mathbf{I})^{-1} \mathbf{Q} \boldsymbol{\Lambda} \mathbf{Q}^\top \mathbf{w}^* \quad (2.34)$$

$$= (\mathbf{Q}(\boldsymbol{\Lambda} + \alpha \mathbf{I}) \mathbf{Q}^\top)^{-1} \mathbf{Q} \boldsymbol{\Lambda} \mathbf{Q}^\top \mathbf{w}^* \quad (2.35)$$

$$= \mathbf{Q}(\boldsymbol{\Lambda} + \alpha \mathbf{I})^{-1} \boldsymbol{\Lambda} \mathbf{Q}^\top \mathbf{w}^*. \quad (2.36)$$

We thus see that the effect of weight decay is to rescale $\mathbf{w}^*$ along the axes defined by the eigenvectors of $\mathbf{H}$: the component of $\mathbf{w}^*$ that is aligned with the i th eigenvector of $\mathbf{H}$ is rescaled by a factor $\frac{\lambda_i}{\lambda_i + \alpha}$. L^2 -regularization thus causes the learning algorithm to “perceive” the input $\mathbf{X}$ as having higher variance, causing shrinking of the weights on features whose covariance with the output target is low compared to this added variance [7].

Other forms of parameter norm penalties exist, such as L^1 -regularization [7, 48].

2.3.3 Data Augmentation

Deep learning models tend to benefit greatly from a larger amount of training data. One way to achieve this is to manipulate existing data in order to create new, “fake” data with desirable properties. This approach is easiest for classification tasks where a high-dimensional input $\mathbf{x}$ is summarized with a single category identity y [7]. This type of task thus requires the machine learning model to be invariant to a certain set of transformations. New $(\mathbf{x}, y)$ pairs can be generated easily by applying those transformations to existing inputs $\mathbf{x}$ in the training set. Example transformations for the object classification task (which is at the center of the investigations in this thesis) are translations by a few pixels, or scaling the image.

Injection of noise in the input to a neural network can also be considered a form of data augmentation [49], whether it be as part of unsupervised learning algorithms [50], or applied to the hidden units [51]. One should however keep in mind that neural networks prove not to be very robust to noise [52].

2.3.4 Batch Normalization

Batch normalization (BN, or batchnorm) [53] is a method of adaptive reparametrization, motivated by the difficulty of training very deep models. Indeed, the gradient update assumes that other layers do not change, which is not the case in practice. The effect of this inaccuracy is exacerbated in very deep networks, because of the high number of layers.

Batch normalization can be applied to any input or hidden layer in a network, often with the goal of improving optimization, but can have a regularizing effect. The summary we present here is inspired strongly by the one in [7]. Let $\mathbf{H}$ be a minibatch of activations of the layer to normalize, arranged as a design matrix with the activations for each example appearing in a row of the matrix. Its normalized version, which we replace it by, is

$$\mathbf{H}' = \frac{\mathbf{H} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}, \quad (2.37)$$

where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are vectors containing the mean and standard deviation of each unit. The operations are applied element-wise at every row of the matrix $\mathbf{H}$: $H_{i,j}$ is normalized by subtracting μ_j and dividing by σ_j .

At training time, we thus have

$$\boldsymbol{\mu} = \frac{1}{m} \sum_{i=1}^m \mathbf{H}_{i,:} \quad (2.38)$$

and

$$\sigma = \sqrt{\delta + \frac{1}{m} \sum_{i=1}^m (\mathbf{H} - \boldsymbol{\mu})_i^2}, \quad (2.39)$$

where δ is a small positive constant imposed to avoid the undefined gradient of $\sqrt{z}$ at $z = 0$. We back-propagate through these operations for computing the mean and the standard deviation, and for applying them to normalize $\mathbf{H}$. The gradient will never propose an operation that acts simply to increase the standard deviation or mean of $\mathbf{H}_{i,:}$; the normalization operations remove the effect of such an action and zero out its component in the gradient [7].

At test time, $\boldsymbol{\mu}$ and σ may be replaced by running averages that were collected during training time [7].

The approach outlined above can reduce the expressivity of the neural network units to which it is applied. It is common to replace $\mathbf{H}$ by $\gamma \mathbf{H}' + \beta$ instead, where γ and β are learned parameters that allow the new variable to have any mean and standard deviation, thus being able to represent the same family of functions of the input as the simple normalization, while having different learning dynamics. This new parametrization is much easier to learn with gradient descent [7].

Finally, one could ask whether batch normalization should be applied to the input $\mathbf{X}$, or to the pre-activation $\mathbf{X}\mathbf{W} + \mathbf{b}$. The authors of [53] note that it is preferable to apply batch normalization to the latter, and more specifically replace it by a normalized version of $\mathbf{X}\mathbf{W}$, the bias term being omitted by virtue of its redundancy with the second reparametrization parameter β . The reasoning for this choice is that the statistics of the input are less Gaussian than those of the pre-activation, and thus less amenable to standardization by linear operations [7].

In the special case of convolutional neural networks, explained in section 2.4, $\boldsymbol{\mu}$ and σ should be the same at every spatial location within a feature map, so that the statistics of the feature map remain the same regardless of spatial location.

2.3.5 Dropout

Dropout [54] provides a computationally efficient but powerful method of regularizing a broad family of models [7].

In order to understand dropout, one first needs to understand the concept of *bagging* (from “bootstrap aggregating”), a technique combining multiple models to reduce generalization error [55]. When bagging, several models trained separately “vote” together on the predicted output for each test sample. This is an example of *model averaging*, and techniques employing this method are known as *ensemble methods* [7]. Bagging involves constructing k different datasets of the same size as the original dataset, but sampled from it with replacement, for k different models.

Model i is then trained on dataset i . This improves the generalization error as follows: consider k regression models, each making an error ε_i on each sample, with the errors drawn from a zero-mean multivariate normal distribution with variances $\mathbb{E}[\varepsilon_i^2] = v$ and covariances $\mathbb{E}[\varepsilon_i \varepsilon_j] = c$. The error made by the average prediction of the ensemble models is $\frac{1}{k} \sum_i \varepsilon_i$. The expected squared error is

$$\mathbb{E} \left[\left(\frac{1}{k} \sum_i \varepsilon_i \right)^2 \right] = \frac{1}{k^2} \mathbb{E} \left[\sum_i \left(\varepsilon_i^2 + \sum_{j \neq i} \varepsilon_i \varepsilon_j \right) \right] = \frac{1}{k} v + \frac{k-1}{k} c. \quad (2.40)$$

If $c = v$, the mean squared error reduces to v , and if $c = 0$, the expected squared error is equal to $\frac{1}{k} v$. The expected squared error is thus inversely proportional to the ensemble size, and the ensemble model performs at least as well as its individual submodels [7].

Dropout essentially trains the ensemble of all subnetworks that can be formed by removing (i.e. multiplying by zero in our case) nonoutput units from an underlying base network [7]. However, to train with dropout, one needs to use a minibatch-based learning algorithm such as SGD, due to the exponentially growing number of subnetworks. Each time an example is loaded into a minibatch, a randomly and independently sampled binary mask is applied to all the input and hidden units in the network, with the probability of inclusion being a fixed hyperparameter (which can potentially depend on the type of unit) [7].

More formally, let $\boldsymbol{\mu}$ be a mask specifying which units to include, and let $J(\boldsymbol{\theta}, \boldsymbol{\mu})$ define the cost of the model defined by the parameters $\boldsymbol{\theta}$ and mask $\boldsymbol{\mu}$. Dropout training then attempts to minimize $\mathbb{E}_{\boldsymbol{\mu}}[J(\boldsymbol{\theta}, \boldsymbol{\mu})]$, containing exponentially many terms. Through sampling of $\boldsymbol{\mu}$, an unbiased estimate of the gradient of this expectation can be obtained [7].

Dropout differs from traditional bagging in that the models are not independent but share parameters. This makes the problem of representing an exponential number of models have tractable space complexity. For reasons of tractability, we also only train a fraction of the possible subnetwork models for a single step, with parameter sharing allowing the remaining subnetworks to arrive at good settings of the parameters [7].

To make a prediction, a bagged ensemble must accumulate votes from all its members, a process called *inference*. Let us now assume that the role of the model is to output a probability distribution. For bagging, each model i then produces a probability distribution $p^{(i)}(y \mid \mathbf{x})$, and the ensemble prediction is given by the arithmetic mean of the individual distributions

$$\frac{1}{k} \sum_{i=1}^k p^{(i)}(y \mid \mathbf{x}). \quad (2.41)$$

For dropout, each submodel (associated with a mask $\boldsymbol{\mu}$) defines a probability distribution $p(y \mid \mathbf{x}, \boldsymbol{\mu})$. The arithmetic mean is then given by

$$\sum_{\boldsymbol{\mu}} p(\boldsymbol{\mu}) p(y \mid \mathbf{x}, \boldsymbol{\mu}), \quad (2.42)$$

where $p(\boldsymbol{\mu})$ is the probability distribution used to sample $\boldsymbol{\mu}$ at training time [7]. This sum is intractable in general because of the large number of possible masks, but the inference can be approximated by averaging the output from multiple sampled masks.

A better approach that has shown promise both theoretically and empirically is to use the geometric mean instead [56]. Our explanation of this method is inspired by the one in [7]. This approach requires some additional requirements in order to ensure that the result is still a probability distribution. We thus require that every event gets assigned a nonzero probability by each submodel. The unnormalized probability distribution defined directly by the geometric mean is then given by

$$\tilde{p}_{\text{ensemble}}(y \mid \mathbf{x}) = \sqrt[2^d]{\prod_{\boldsymbol{\mu}} p(y \mid \mathbf{x}, \boldsymbol{\mu})}, \quad (2.43)$$

where d is the number of units that may be dropped. We use a uniform distribution over $\boldsymbol{\mu}$ to simplify the presentation. This probability distribution can be normalized as

$$p_{\text{ensemble}}(y \mid \mathbf{x}) = \frac{\tilde{p}_{\text{ensemble}}(y \mid \mathbf{x})}{\sum_{y'} \tilde{p}_{\text{ensemble}}(y' \mid \mathbf{x})}. \quad (2.44)$$

Perhaps surprisingly, p_{ensemble} can be approximated by evaluating $p(y \mid \mathbf{x})$ in the model with all units, but with the weights going out of unit i multiplied by the inclusion probability of unit i [54, 57]. This approach is called the *weight scaling inference rule*, and its performance has been empirically shown to be good in the literature [58, 59].

A key insight of dropout is that training a network with stochastic behavior and making predictions by averaging over multiple stochastic decisions implements a form of bagging with parameter sharing [7]. We described dropout as bagging an ensemble of models formed by including or excluding units earlier. However, these modifications do not need to be limited to modifications involving inclusion and exclusion. Any form of modification parametrized by a vector $\boldsymbol{\mu}$ can be thought of as training an ensemble consisting of $p(y \mid \mathbf{x}, \boldsymbol{\mu})$ for all possible values of $\boldsymbol{\mu}$.

The good performance of dropout in practice can also be explained by the observation that dropout trains not just a bagged ensemble of models, but an ensemble of models sharing hidden parameters [7, 57]. This means that each hidden unit must be able to perform well regardless of which other hidden units are in the model.

Finally, we give two more important aspects of dropout. The first is that the power of dropout resides partially in the fact that the masking noise is applied to the hidden units. In practice, this means that the information content is adaptively “destroyed,” while still making use of the knowledge that the model has acquired so far about the input distribution [7]. Secondly, the noise is multiplicative, and not additive, meaning it does not allow pathological solutions to the noise robustness problem. Indeed, dropout can be seen as a process of constructing new inputs (which is data augmentation, as explained in section 2.3.3) by multiplying existing ones by noise [7].

2.3.6 Early Stopping

The final regularization method we present is *early stopping*. Given the large representational capacity of neural networks, one can often observe that the training error decreases steadily over time but that the validation error begins to rise again after a certain number of epochs. One could thus obtain a model with a better validation set error (and thus hopefully test set error) if one were to return to the parameter setting at the point in time with the lowest validation set error. Algorithm 2.6 gives a more formal view of how this can be achieved [7].

Early stopping can be viewed as a parameter selection algorithm, where the parameter in question is the number of training steps. The optimal number of training steps can then be computed in parallel to the training process, by periodically computing the validation set error. The length of this “period” determines the resolution of the estimate of the optimal training time [7].

Early stopping requires a validation set, hence some training data is not fed to the model. This extra data can be used for a second training process after the first training process with early stopping has completed, this time using the entire training set. This second step can be done in two ways:

- initialize the model again and retrain on all the data, either for the optimal number of training steps, or for the optimal number of parameter updates; or
- continue training the model, but now on the entire training dataset, until the training loss reaches its minimum from the early stopping step again.

Both methods are viable in practice, with the second being faster but less well-behaved [7].

Finally, one could ask how early stopping serves as a form of regularization. It has been argued that early stopping has the effect of restricting the optimization procedure to a small volume of parameter space in the neighborhood of θ_0 [60, 61]. More formally, taking τ optimization steps with learning rate η , we can view $\eta\tau$ as a measure of effective capacity. Assuming the gradient is bounded, restricting

Algorithm 2.6 Early stopping.

```
1: procedure EARLYSTOPPING( $n, p, \theta_0$ ) ▷ number of steps  
   between evaluations, number of times to observe worsening validation set error,  
   and initial parameters  
2:    $\theta \leftarrow \theta_0$   
3:    $i \leftarrow 0$   
4:    $j \leftarrow 0$   
5:    $v \leftarrow \infty$   
6:    $\theta^* \leftarrow \theta$   
7:    $i^* \leftarrow i$   
8:   while  $j < p$  do  
9:     update  $\theta$  by running  $n$  steps of the training algorithm  
10:     $i \leftarrow i + n$   
11:     $v' \leftarrow \text{ValidationSetError}(\theta)$   
12:    if  $v' < v$  then  
13:       $j \leftarrow 0$   
14:       $\theta^* \leftarrow \theta$   
15:       $i^* \leftarrow i$   
16:       $v \leftarrow v'$   
17:    else  
18:       $j \leftarrow j + 1$   
19:    end if  
20:  end while  
21:  best parameters are  $\theta^*$ , best number of training steps is  $i^*$   
22: end procedure
```

both τ and η limits the volume of parameter space reachable from θ_0 . In this sense, $\eta\tau$ behaves as if it were the reciprocal of the coefficient used for weight decay, explained in section 2.3.2 [7].

2.4 Convolutional Networks

Convolution neural networks (CNNs) [62] are a specialized kind of neural network for processing data that has a known grid-like topology. CNNs are distinct from traditional neural networks in that they use *convolution*, as explained in section 2.4.1, in place of general matrix multiplication in at least one of their layers [7].

2.4.1 The Convolution Operation

Convolution is an operation on two functions of a real-valued argument. Let f and g be two such functions. Their convolution is then given by

$$(f * g)(t) \triangleq \int_{-\infty}^{\infty} f(\tau)g(t - \tau) d\tau. \quad (2.45)$$

We say that f is the *input*, and g is the *kernel*. $f * g$ is the *feature map*.

This definition can also be adapted to the discrete case, when f and g are only defined for integer values of n :

$$(f * g)[n] \triangleq \sum_{m=-\infty}^{\infty} f[m]g[n - m]. \quad (2.46)$$

In convolutional networks, the input and kernel are often tensors, for which a finite set of values are stored. We make the assumption that outside of the domain for which values are stored, the function is zero. The infinite summation in (2.46) can thus be replaced by a finite one [7].

Convolution can also be defined over more than one axis at a time. In general, let f and g be functions on $\mathbb{R}^d$ for the real-valued case, or $\mathbb{Z}^d$ for the integer-valued case. We then have

$$(f * g)(x_1, \dots, x_d) \triangleq \int_{\mathbb{R}^d} f(y_1, \dots, y_d)g(x_1 - y_1, \dots, x_d - y_d) dy_1 \cdots dy_d, \quad (2.47)$$

$$(f * g)[n_1, \dots, n_d] \triangleq \sum_{m \in \mathbb{Z}^d} f[m_1, \dots, m_d]g[n_1 - m_1, \dots, n_d - m_d]. \quad (2.48)$$

Convolution is commutative by virtue of the kernel being *flipped* relative to the input. In practical implementations, the commutative property is not required,

and the *cross-correlation* is used instead:

$$(f \star g)(x_1, \dots, x_d) \triangleq \int_{\mathbb{R}^d} f(y_1, \dots, y_d) g(x_1 + y_1, \dots, x_d + y_d) dy_1 \cdots dy_d, \quad (2.49)$$

$$(f \star g)[n_1, \dots, n_d] \triangleq \sum_{m \in \mathbb{Z}^d} f[m_1, \dots, m_d] g[n_1 + m_1, \dots, n_d + m_d]. \quad (2.50)$$

We do not distinguish between the two and instead refer to both as “convolution,” without specifying whether the kernel is flipped, unless relevant [7].

2.4.2 Intuition Behind Convolutional Networks

According to [7], there are three important ideas leveraged by convolution that can improve a machine learning system, as well as allow one to work with inputs of variable size.

2.4.2.1 Sparse Interactions

Convolutional networks have *sparse interactions*, i.e. input units do not interact with every output unit and vice versa. This is a result of making the kernel smaller than the input, which also allows us to lower memory requirements and improve statistical efficiency, while speeding up the computation of the output: for m inputs and n outputs, matrix multiplication requires $O(mn)$ operations, whereas kernel convolution with a kernel of size k only requires $O(kn)$ [7].

2.4.2.2 Parameter Sharing

Parameter sharing, which we briefly touched on in section 2.3.5, refers to using the same parameter for more than one function in a model. In a convolutional network, each weight of the kernel is used at every position of the input, except potentially at the boundary. This means only one set of parameters is learned. This does not affect the $O(kn)$ complexity of forward propagation, but reduces the memory requirements to just k parameters [7].

2.4.2.3 Equivariant Representations

The particular form of parameter sharing used with convolution causes convolutional networks to have *equivariance* to translation. We say a function f is *equivariant* to a function g if $f(g(x)) = g(f(x))$ [63]. Convolution is then equivariant with any function g that translates the input. Convolution is not equivariant to some other transformations, such as rotation or scaling [7].

2.4.3 Pooling Functions

In convolutional networks, it is customary to use *pooling layers* to modify the output of the activation function further. The goal of pooling is to make the representation approximately *invariant* to small translations of the input, which is a desirable property when the existence of a feature is more important than its exact position [7].

Many different pooling functions can be imagined, with perhaps the most famous example being *max pooling* [64], where the pooled output is the maximum output over a rectangular neighborhood. We also mention *average pooling*, where the mean of the neighborhood is kept instead. Other approaches are explained in [65, 66, 67]. For example, one could pool over the outputs of separately parametrized convolutions, so features can learn which transformations to be invariant to.

Pooling can also reduce the computational complexity by using a *pooling stride* $k > 1$, i.e., by reporting pooling statistics for regions spaced apart k pixels. This reduces the number of units to process in the next layer by a factor of k . Pooling also allows one to handle inputs of varying size [7].

2.4.4 Convolution and Pooling as an Infinitely Strong Prior

This interpretation is adapted from [7].

A prior probability distribution is considered to be stronger when its entropy is lower [68]. An infinitely strong prior is thus a distribution which places zero probability on some events, regardless of the support the data gives to these events. In our case, the events are simply the assignment of a certain value to a certain parameter.

Indeed, a convolutional network behaves the same³ as a fully connected network with the infinitely strong prior that the weights for a hidden unit must be identical to the weights of its neighbor but shifted in space, as well as that the weights outside the spatially contiguous kernel assigned to that unit must be zero. Convolution thus introduces an infinitely strong prior probability distribution over the parameters of a layer, saying that the function the layer should learn contains only local interactions and should be equivariant to translation. Similarly, pooling imposes a prior that each unit should be invariant to small translations.

Two insights this perspective gives are worth pointing out:

- convolution and pooling can cause underfitting when the assumptions made by the prior are not accurate; and
- convolutional models should only be compared to other convolutional models in benchmarks of statistical learning performance; this insight has also led to

³Conceptually, but computationally it is much faster.

the development of benchmarks specifically for *permutation invariant* models, which have to “learn” the concept of topology, as opposed to models in which this knowledge is built-in.

2.4.5 Modifying Convolution for Neural Networks

The convolution operation used in convolutional networks is slightly different from the one in (2.46):

- convolution in neural networks usually refers to an operation that consists of many applications of convolution in parallel; and
- the input is not just a grid of real numbers, but a grid of vector-valued observations; each element of the vector corresponds to a different *channel*.

This means we lose out on the commutative property in general, regardless of flipping [7].

Let $\mathbf{K}$ be a 4-D kernel tensor with element $K_{i,j,k,l}$ giving the connection strength between a unit in channel i of the output and channel j of the input, with an offset of k rows and l columns between the output unit and the input unit, and let $\mathbf{V}$ be the input tensor with element $V_{i,j,k}$ giving the value of the input unit within channel i at row j and column k . Finally, let $\mathbf{Z}$ be the output tensor, with the same format as $\mathbf{V}$. We then have

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n}, \quad (2.51)$$

where the summation bounds are inferred from the values for which the tensor indexing operations are valid [7].

When we introduce a convolution stride s , meaning we sample only every s pixels in every direction, we can skip over some positions of the kernel; this can be thought of as a downsampling of the output of the full convolution. Defining a downsampled convolution function c , we then have [7]

$$Z_{i,j,k} = c(\mathbf{K}, \mathbf{V}, s)_{i,j,k} = \sum_{l,m,n} \left[V_{l,(j-1)s+m,(k-1)s+n} K_{i,l,m,n} \right]. \quad (2.52)$$

Note that, in general, a bias term is also used before the nonlinearity. It is typical to have one bias per channel of the output, which is shared across all locations within each convolution map. With fixed-size inputs, separate biases can also be learned at each location of the output map, allowing the model to correct for differences in image statistics at different locations [7].

2.4.5.1 Zero Padding

Note that in positions where this is required, the input tensor $\mathbf{V}$ can be zero padded, in order to maintain representation width. Indeed, we mention three common types of zero padding:

- No padding at all; the convolution kernel can then only visit positions in which the kernel is entirely contained within the image. With input width m and kernel width k , the output will then have width $m - k + 1$.
- Enough padding to maintain the size; the convolution kernel can then visit every position in the input image, maintaining the representation width.
- Enough padding in order for every pixel in the input image to be visited k times in each direction. With input width m and kernel width k , the output will then have width $m + k - 1$.

The optimal (in terms of generalization performance) amount of zero padding is usually somewhere in between the first two options [7].

2.4.5.2 Locally Connected Layers

In some cases, we know that certain features can only occur in a subset of the input image. Convolution is then usually replaced by *locally connected layers* [62, 69], also called *unshared convolution*. The adjacency matrix of the network graph is then the same, but with a different weight on each connection, specified in a 6-D tensor $\mathbf{W}$. The indices into $\mathbf{W}$ are respectively: i , the output channel; j , the output row; k , the output column; l , the input channel; m , the row offset within the input; and n , the column offset within the input. The linear part of the locally connected layer is then given by [7].

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} W_{i,j,k,l,m,n}. \quad (2.53)$$

It is natural to give each unit its own bias in locally connected layers [7].

2.4.5.3 Tiled Convolution

A compromise between convolutional and locally connected layers is given by *tiled convolutional layers* [70, 71]. Rather than learning separate weights at every spatial location, we learn multiple kernels that we rotate through as we move through space. This maintains the property of locally connected layers that neighboring locations have different filters, while reducing the memory requirements compared to this approach. Let $\mathbf{K}$ be a 6-D tensor where two of the dimensions correspond to

different locations in the output map. Instead of separate indices for every location in the output map, output locations are cycled through a choice of t kernel stacks in each direction. We thus have

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n,(j \bmod t)+1,(k \bmod t)+1}. \quad (2.54)$$

This can be generalized to include different tiling ranges in each dimension. Note that typically, tiled convolution also shares biases, with the same tiling pattern as the kernels [7].

2.4.5.4 Learning in Convolutional Networks

In order to learn in convolutional networks, one needs to be able to compute the gradient with respect to the kernel, given the gradient with respect to the outputs. As convolution is a linear operation, it can be written as a matrix multiplication (after the reshaping the input tensor into a flat vector), with a *Toeplitz matrix* [72]. This allows us to distinguish the three operations that are required to compute the gradients to train a convolutional network [7]:

- Convolution is trivially required.
- Multiplication by the transpose of the matrix defined by convolution is required to back-propagate error derivatives through a convolutional layer.
- Transpose convolution is necessary because, in general, convolution is not sufficient to compute the input gradient.

We now give an example of how the gradient computation is done in the two-dimensional, single example case, inspired from [7, 73]. More formally, we wish to train a convolution network incorporating strided convolution of kernel stack $\mathbf{K}$ applied to a multichannel image $\mathbf{V}$ with stride s as defined by $c(\mathbf{K}, \mathbf{V}, s)$, with c being defined as in (2.52). We thus wish to minimize some loss function $J(\mathbf{V}, \mathbf{K})$. During forward propagation, we will need c to output $\mathbf{Z}$, which is propagated through the network and used to compute J . During back-propagation, we will receive a tensor $\mathbf{G}$ such that $G_{i,j,k} = \frac{\partial}{\partial Z_{i,j,k}} J(\mathbf{V}, \mathbf{K})$.

To compute the derivatives with respect to the weights in the kernel, which are needed to train the network, we use an auxiliary function, g :

$$g(\mathbf{G}, \mathbf{V}, s)_{i,j,k,l} = \frac{\partial}{\partial K_{i,j,k,l}} J(\mathbf{V}, \mathbf{K}) = \sum_{m,n} G_{i,m,n} V_{j,(m-1)s+k,(n-1)s+l}. \quad (2.55)$$

Unless this layer is the bottom layer of the network, the gradient with respect to $\mathbf{V}$ needs to be computed in order to propagate the error further down. This can be done with another auxiliary function, h :

$$h(\mathbf{K}, \mathbf{G}, s)_{i,j,k} = \frac{\partial}{\partial V_{i,j,k}} J(\mathbf{V}, \mathbf{K}) = \sum_{\substack{l,m \\ \text{s.t.} \\ (l-1)s+m=j}} \sum_{\substack{n,p \\ \text{s.t.} \\ (n-1)s+p=k}} \sum_q K_{q,i,m,p} G_{q,l,n}. \quad (2.56)$$

2.4.6 Efficient Convolution Algorithms

Convolution can be computed more efficiently by passing through the frequency domain; indeed, convolution is equivalent to converting both the input and output to the frequency domain using a Fourier transform, performing point-wise multiplication of the two signals, and then converting back to the time domain using an inverse Fourier transform [7].

Another way of improving convolution is through *depthwise separable convolution*. Indeed, when a d -dimensional kernel can be expressed as the outer product of d vectors, it is called *separable*. When this is the case, it is more efficient to compute a d -dimensional convolution as a composition of d instances of one-dimensional convolution. If the kernel has width w in each dimension, the complexity improves from $O(w^d)$ to $O(wd)$, both in runtime and storage space [7].

2.5 Conclusion

In this section, we have introduced the necessary background information about neural networks and deep learning for the rest of this thesis. We have given a general introduction to the topic, an explanation of how neural networks are trained, and some information about relevant modern tools. Finally, we have also introduced the special type of networks known as convolutional networks, which we use throughout this thesis.

In the next chapter, we will use some of these notions in order to motivate our further experimental investigations.

Chapter 3

State of the Art and Related Work

This section gives a brief overview of the main related works that led us to develop the intuitions which guided the investigations conducted during this thesis. It should also allow the reader to infer that the topic of this thesis is of great interest in the literature on deep neural networks. We divide these papers into two classes: those discussing generalization in section 3.1, and those discussing the identification of visual patterns by neurons in section 3.2. Our goal, as described in chapter 1, is then to combine these two notions as well as the relation between the internal (i.e. neuron-level) representations of networks and generalization.

3.1 Generalization

The topic of generalization, which we defined in section 2.1, is one that is discussed extensively in the literature, and the problem of understanding how deep neural networks are able to perform so well on generalization tasks can certainly be considered timely. This section briefly summarizes some papers that helped us build our current intuitions, and led to the experiments we discuss in the rest of this thesis.

3.1.1 Layer Rotation: a Surprisingly Powerful Indicator of Generalization in Deep Networks?

In [5], the authors observe that layer rotation (i.e. the evolution across training of the cosine distance between each layer’s weight vector and its initialization) correlates well with generalization performance. Specifically models with good generalization performance are associated to fast layer rotation during the early

phase of training. They also observe that the addition of weight decay (which we describe in section 2.3.2) to a network’s cost function has precisely the effect of increasing layer rotation.

This paper also shows evidence that networks with high layer rotation and hence good generalization performance tend to have less noisy filters, i.e. that the percentage of zero/negligible weights is larger. This observation led us to perform the experiments in chapter 5, and section 5.1 in particular, and motivated our subsequent definition of the notion of significant component. Fundamentally, the notion of a noiseless filters is comparable to that of a pattern, which we mention in chapter 7. This paper was instrumental in motivating the study of the emergence of such patterns in the network, notably by generalizing its observations to convolutional layers other than the first layer of the network.

3.1.2 Experimental Study of the Neuron-Level Mechanisms Emerging from Backpropagation

In [6], preliminary evidence is presented that during network training using back-propagation, which we discussed in section 2.2, local, neuron-level mechanisms emerge, despite the fact that back-propagation is based on a global, black box optimization procedure.

These observations motivate our investigation into binary partitions at the neuron level, which are discussed in chapter 6.

3.1.3 Intraclass Clustering: an Implicit Learning Ability that Regularizes DNNs

The results of [1] show that intraclass clustering (i.e. the ability to extract meaningful clusters among the samples of a class) is an implicit regularization mechanism in deep neural networks. In order to validate this hypothesis, they design several measures of intraclass clustering, and show that these measures correlate with generalization performance across a large set of hyperparameters.

This paper was instrumental to this thesis, most notably in inspiring the methodology of our work, which we explain in section 4.1, as well as guiding our choice of measures in chapters 6 and 7. The authors also provided us with a large part of the experimental data and models that were used throughout this thesis.

3.1.4 Exploring Generalization in Deep Learning

In [2], the authors discuss two related questions:

- What is the complexity measure that is implicitly enforced to lead to good generalization performance of deep neural networks?
- Given different global minimizers of the empirical risk, can one tell which one generalizes better based on this complexity measure?

As such, this paper presents an empirical analysis of different metrics, such as sharpness, Lipschitz continuity, and various norms on the weight matrices, that can be used to predict the generalization error of a neural network.

This paper and the investigations detailed in it give the reader a good idea of the complexity of understanding generalization in deep neural networks, thus affirming the importance of the work done in this thesis.

3.1.5 Understanding Deep Learning Requires Rethinking Generalization

In [3] and its updated version [74], the authors explore the curious observation that despite their enormous expressivity (as evidenced by the universal approximation theorems of [18, 19, 20, 21, 22, 23], but also by the observation that given enough training time, convolutional neural networks are able to reach a zero loss on a training set with random permutation of labels), deep neural networks rarely overfit thanks to implicit regularization, which is also discussed in [1].

However, while regularization improves performance, it is not the end-all-be-all of understanding generalization, and it is still not clear what in CNNs may allow one to predict their generalization performance; our hope with this thesis is to work towards a theory of generalization in deep learning.

3.1.6 Asymmetric Valleys: Beyond Sharp and Flat Local Minima

In [75], the authors show that the loss landscapes of modern neural networks present local minima which, rather than being flat or sharp, present many asymmetric directions. This type of loss landscape favors robustness, as SGD is favorably biased towards flat sides, and is thus more robust to empirical shift.

These observations helped us choose and study the optimizer and effective learning rate hyperparameters in this thesis.

3.1.7 An Investigation into Neural Net Optimization via Hessian Eigenvalue Density

In [76], the authors uncover more about the learning and optimization dynamics of deep neural networks, using properties of the Hessian eigenvalue spectrum. The first contribution of the paper is the development of a tool allowing one to study the evolution of the Hessian eigenvalue spectrum throughout the optimization process of a neural network. This tool is then used to study several hypotheses in the existing machine learning literature about the geometry of the loss surface. The role of outlier eigenvalues in particular is explored further, and the authors show that it is closely related to the method of batch normalization, and explains how this innovation in training deep neural nets speeds up optimization.

These observations, together with many of the other papers in this section, motivate our research into what causes good generalization performance in modern neural networks.

3.1.8 The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks

In [77], it is shown that “a randomly-initialized, dense neural network contains a subnetwork that is initialized such that –when trained in isolation– it can match the test accuracy of the original network after training for at most the same number of iterations.” Finding this subnetwork is referred to as finding the “winning ticket,” and the paper describes an algorithm to do so:

1. randomly initialize a network $f(\mathbf{x}; \boldsymbol{\theta}_0)$ (where $\boldsymbol{\theta}_0 \sim \mathcal{D}_{\boldsymbol{\theta}}$);
2. train it for j iterations, arriving at parameters $\boldsymbol{\theta}_j$;
3. prune $p\%$ of the parameters in $\boldsymbol{\theta}_j$, creating a mask $\mathbf{m}$;
4. reset the remaining parameters to their initial values in $\boldsymbol{\theta}_j$, creating the winning ticket $f(\mathbf{x}; \mathbf{m} \odot \boldsymbol{\theta}_0)$.

They show that the network obtained by pruning with reset outperforms the network without pruning, but also strongly outperforms the network obtained through pruning and reinitialization.

This paper thus partially motivated our investigations in chapter 5, about the existence of significant weight components that are particularly important for classification.

3.2 Visual Pattern Identification by Neurons

In this section, we discuss some related work concerning the presence of visual patterns and their identification by neurons. This observation has been extensively discussed in the literature [78, 79, 80, 81].

3.2.1 Visualizing and Understanding Convolutional Networks

In [82], the authors show that features learned by convolutional networks have many desirable properties, and are far from being random. They thus agree with the intuition one would have of the type of learning done by each layer: the first few layers of the convolutional network learn basic characteristics of the input image, while later layers compose these basic characteristics to form more abstract objects.

These basic characteristics and abstract objects lead us to a particular definition of patterns, and motivate our study of these patterns in chapter 7.

3.2.2 Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps

In [83], the authors present two visualization techniques of image classification models learned using deep convolutional networks, based on computing the gradient of the class score with respect to the input image. The first generates an image which maximizes the class score, thus embodying the notion the convolutional network has of a given class. The second computes a saliency map specific to a given image and class.

This paper contributed to our desire to study the presence patterns at the neuron level in chapter 7.

3.2.3 Understanding Neural Networks Through Deep Visualization

The authors of [84] introduce two tools that aid in understanding how convolutional networks work (specifically, what computations they perform in their intermediate layers). Their first tool enables one to look at how activations are produced in each layer of a trained convolutional network as it processes an image or video, thus helping build one's intuition as to how convolutional networks work. The second tool helps to visualize features at each layer of a deep neural network via regularized optimization in the image space.

This work clearly shows the importance of studying what is happening inside the network at the neuron and layer level when trying to understand how convolutional networks generalize.

3.2.4 Compositional Explanations of Neurons

[85] proposes a way to explain deep neural network models as compositions of atomic concepts based on observed neuron activations. This paper considerably extends previous work by allowing concepts to be composed by defining logical operators, and applying a beam search to identify a logical form that closely matches the neuron behavior. They apply their procedure to study models trained on an image classification (scene classification) task and a natural language inference (NLI) task.

This paper partially motivates our decision to mainly consider neuron-level behavior throughout this thesis, and helped us build our intuition about what drives deep convolutional networks to perform so well.

3.3 Conclusion

Through the related work in this chapter, we hope to have convinced the reader of a couple of things: (i) understanding generalization in deep neural networks is a hot topic for theoretical machine learning research, and (ii) there are strong indications that pattern detection plays an important role in a network’s ability to generalize well.

These notions are studied in more detail throughout this thesis, with the global objective being to better understand how internal representations of the network are related to its generalization performance. In chapter 4, we explain our general methodology (which was also inspired from related work), and lay out the experimental setup and tools that were used to arrive at our results.

Chapter 4

Methodology and Experimental Setup

This section gives an overview of the methodology (section 4.1) and experimental setup (section 4.2) that were used to evaluate whether a specific characteristic of the trained models correlates with generalization.

4.1 Methodology

In chapter 1, we explained several hypotheses that we study as part of this thesis. Generally speaking, our goal is to better understand the link between the generalization performance of a model and a given metric of interest. In our case, the metrics are functions of the characteristics of the filter or of the internal representation of samples at the neuron level, for a given model, as motivated in chapter 3.

With results mainly based on experimental investigation, it is important to make our experiments as broad and all-encompassing as possible. Our experimental setup is a consequence of this insight. It is explained in more depth in section 4.2, but the general idea is simple to grasp. As depicted in figures 4.1 and 4.2, through the use of a large family of trained models $\{\mathcal{M}_i\}$, each with its own parameter choices θ_i , we obtain a set of vastly different generalization performances. We can then plot the generalization as a function of some metrics of interest, each of those metrics reflecting the state of neuron filters or internal activations. Through careful examination of the resulting figures, we then hope to highlight some correlation between generalization and some characteristic of the model captured by the metric of interest, thereby validating the hypothesis that led to the metric definition. Our methodology is thus similar to the one employed in [1, 86].

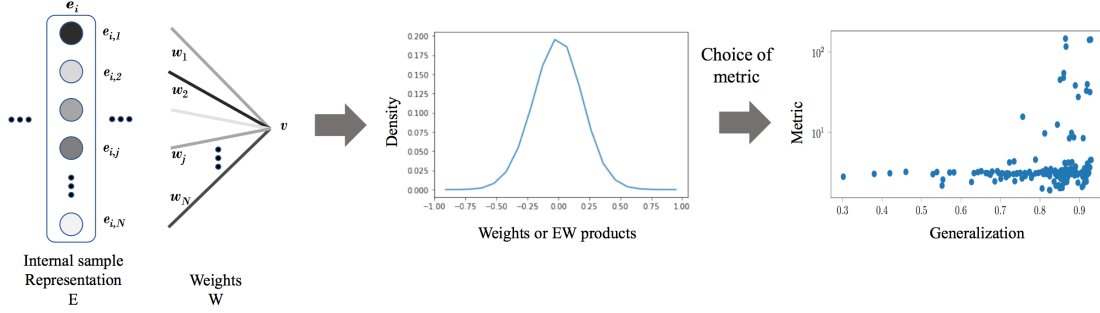


Figure 4.1: The experimental methodology used in chapter 5. We look at how the statistical properties of the distribution of layer weights (section 5.1) and of excitation-weight products (section 5.2) relate to generalization performance through studying them for a variety of models.

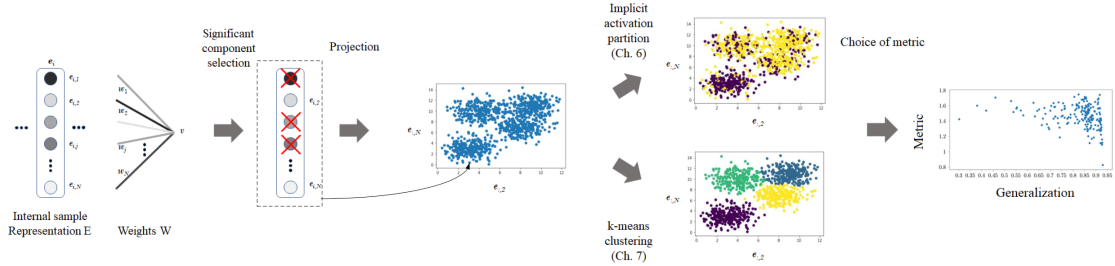


Figure 4.2: The experimental methodology used throughout chapters 6 and 7. There are three steps to our approach: (i) Select the excitations corresponding to significant weight components, and project input samples onto this representation space. These components are typically the ones corresponding to large weight magnitude for the neuron of interest. (ii) Assign a *label* to each sample, based on a certain aspect of the internal representation of the input vector associated to that sample. These labels are represented by the various colors in the clusters. In chapter 6, these labels are obtained through the implicit partition of samples as activated or not activated by the neuron, whereas in chapter 7, they are obtained through *k*-means clustering in the internal representation space. (iii) Compute a *metric* on the *clusters* resulting from applying this labeling strategy to each sample, and see how this metric correlates with model generalization performance.

4.2 Experimental Setup

4.2.1 Dataset

The dataset we use for the experiments in this thesis is the CIFAR-10 dataset [87]. It consists of 60 000 32×32 colour images in 10 classes, with 6000 images per class. There are 50 000 training images, and 10 000 test images. We thus constrain ourself to the study of deep neural networks for the image classification task.

The arguments for using CIFAR-10 are as follows:

- the dataset is representative of real image contents; and
- the small size and relative simplicity of the dataset allow us to reduce the complexity and thus the training and evaluation time of the networks, which is useful when one wants to perform a wide array of vastly different simulations, as is the case in this thesis.

Our choice of dataset was also influenced by the choice of [1], one of the many papers that helped shape our current intuition and understanding of the field.

4.2.2 Models

As described in section 4.1, our methodology revolves around using various models $\mathcal{M}_i$, each with its own parameter set θ_i . For reasons not dissimilar to those explained in section 4.2.1, we use a network architecture based on the family of VGG models [88] for our experiments.

Indeed, the VGG model architecture is fairly simple from a theoretical point of view compared to many of the current state-of-the-art networks, and networks implementing it can be trained in a reasonable amount of time. Despite this simplicity, they perform reasonably well on image classification tasks such as CIFAR-10, with minimal adaptations. The various original VGG configurations are presented in table 4.1.

In order to make VGG work on images from the CIFAR-10 dataset, our architectures are slightly modified. Figure 4.3 shows a schematic of one of the model architectures that was actually used; other adapted models are easily visualized through the explanations in this section.

As explained in section 4.1, we train a varied set of models $\{\mathcal{M}_i\}$, each with its parameter set θ_i . In order to obtain different models, we densely explore a grid of hyperparameter values, and train the resulting models to convergence. We consider the following hyperparameters:

- The *architecture* refers to the shape of the layers and network, with the depth and width factors being multiplicative with respect to a base value.

VGG Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
Input (224×224 RGB image)					
conv3-64	conv3-64 LRN	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
Max Pooling					
conv3-128	conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
Max Pooling					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 conv1-256	conv3-256 conv3-256 conv3-256	conv3-256 conv3-256 conv3-256 conv3-256
Max Pooling					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
Max Pooling					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 conv1-512	conv3-512 conv3-512 conv3-512	conv3-512 conv3-512 conv3-512 conv3-512
Max Pooling					
fc-4096					
fc-4096					
fc-1000					
Softmax					

Table 4.1: VGG model configurations. The depth increases from left (A) to right (E), as more layers are added (added layers are in bold). Convolutional layer parameters are denoted as “conv⟨receptive field size⟩-⟨number of channels⟩;” fully connected layer parameters are denoted as “fc-⟨layer width⟩.” The ReLU activation functions are not shown. This table is heavily inspired from table 1 in [88].

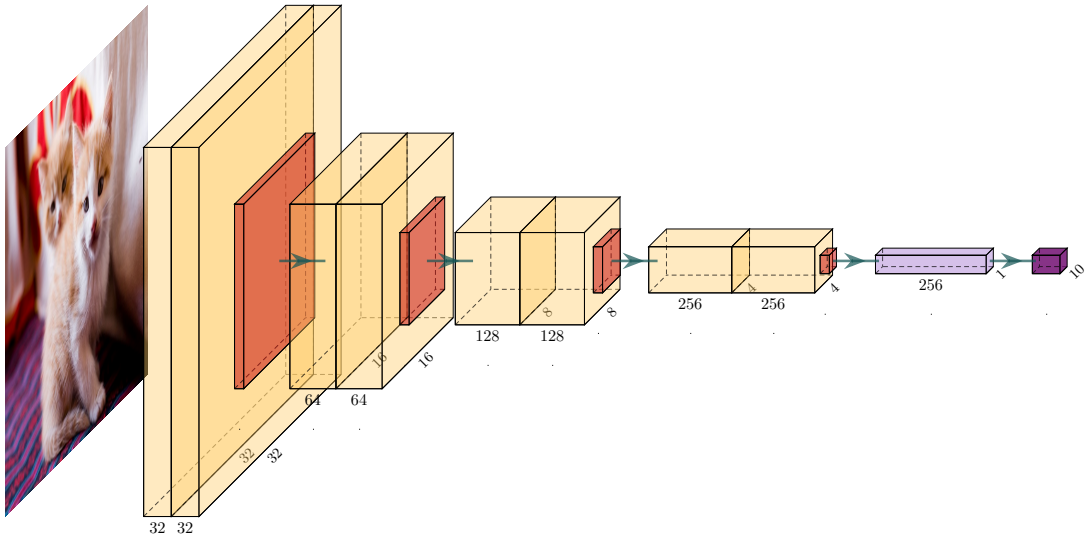


Figure 4.3: Example architecture of one of the models that was used. Convolutional layers are in yellow; (average) pooling layers are in red; fully connected layers are in light purple; and softmax layers are in dark purple. Inputs are assumed to be of size 32×32 , over 3 RGB channels, as is the case with CIFAR-10 images (section 4.2.1). Regularization techniques implemented as layers (e.g. dropout/batch normalization/etc. (section 2.3) are omitted), and the convolutional and fully connected layers implicitly assume ReLU activation (section 2.1). Image generated using software based on [89].

- The *batch size* is the number of samples per minibatch (section 2.2.3), i.e. m in algorithms 2.3 through 2.5.
- The *effective learning rate* is a combination of the learning rate (LR) and eventual L^2 -regularization/weight decay. Note that the learning rate is not constant, but is scheduled to decrease by a constant multiplicative factor at predefined numbers of epochs. The weight decay value simply refers to α in (2.25).
- *Noise regularization* combines the dropout rate when dropout is used (section 2.3.5) and whether data augmentation is used (section 2.3.3).
- The *optimizer* is the algorithm used to perform gradient descent. It is either SGD with momentum (algorithm 2.4, though it is simply written as “SGD” in figures) or Adam (algorithm 2.5).

Table 4.2 gives an overview of the values each hyperparameter takes.

Hyperparameter	Values
Architecture (depth factor, width factor)	$\{(1, 1), (1, 1.5), (1.5, 1)\}$
Batch size	$\{100, 300\}$
Effective learning rate (learning rate, weight decay)	$\{(0.01, 0), (0.032, 0), (0.1, 0), (0.1, 4 \cdot 10^{-5})\}$
Noise regularization (dropout, data augmentation)	$\{(0, \text{no}), (0.2, \text{no}), (0.4, \text{no}), (0, \text{yes})\}$
Optimizer	SGDMomentum, Adam

Table 4.2: Model hyperparameter choices. The meaning of each hyperparameter is explained in the main body of the text.

Additionally, all models use batch normalization, described in section 2.3.4, and implement some form of early stopping (section 2.3.6). Through the use of early stopping, we have thus implicitly taken into account part of the learning dynamics of the network. This leaves us with a total of 192 different models, which are used for the experiments throughout the rest of this thesis.

4.2.3 Hardware and Software Specifications

Due to the largely experimental nature of this thesis, this section gives a brief overview of the technical environment in which experiments were run.

The following hardware and software were involved in getting the material for this thesis:

- Networks were trained using the Cassiopee remote server containing four Nvidia GeForce GTX 1080 Ti graphical processing units, with just under 12 GB of RAM.
- Experiments using these networks were run on one of two computers:
 - An Early 2015 MacBook Pro, with a 2.9 GHz Intel Core i5 processor, 8 GB of 1867 MHz DDR3 RAM, an Intel Iris Graphics 6100 GPU with 1536 MB of RAM, and 500 GB of flash storage. This machine runs on macOS Sierra, version 10.12.6.
 - A 2019 MacBook Pro, with a 2.3 GHz 8-Core Intel Core i9 processor, 16 GB of 2667 MHz DDR4 RAM, an AMD Radeon Pro 5500M GPU with 4 GB of RAM, and 1 TB of flash storage. This machine runs on macOS Big Sur, version 11.2.3.
- Models and experiments are implemented in Python (version 3.6.8) [90], on the Anaconda distribution platform.
- Jupyter notebooks (version 6.0.1) were used for rapid prototyping of code.
- The machine learning library Keras (version 2.4.3) [91] was used for tasks related to the neural networks, as well as NumPy (version 1.18.5) [92], SciPy (version 1.3.1) [93] and Scikit-learn (version 0.23.2) [94] for other numerical problems.
- Matplotlib (version 3.1.1) [95], PlotNeuralNet (version 1.0.0, adapted) [89], and tikzplotlib (version 0.9.8) [96] were used to generate the various plots in this thesis.
- This thesis was typeset using L^AT_EX 2_ε (T_EX Live 2018), using the EPL thesis template.

4.3 Conclusion

In this chapter, we have given a fairly exhaustive overview of both the methodology and experimental setup of this thesis. This information should help the

reader understand our experimental procedure, as well as allow a high level of reproducibility.

In the next chapter, we will follow this methodology as we described it in figure 4.1 in order to learn more about the statistical properties of the neuron-level representations of samples.

Chapter 5

Statistical Analysis of Neuron-Level Representations

As explained in section 1.1, our thesis aims at studying whether the hidden neurons perceive the internal representation of samples as a set of clusters, whether binary or otherwise (the latter case being referred to as “patterns” in this manuscript). Before considering how the input vectors of each neuron are clustered (which is the purpose of chapters 6 and 7), this chapter investigates whether it is relevant, when considering a specific neuron, to restrict the dimension of the input vector to a small subset of neuron-dependent significant components. Our investigation is motivated by preliminary results from the literature, explained in chapter 3, showing that models offering better generalization are characterized by filters with many close-to-zero coefficients in the first layer of the network [5]. Demonstrating that this observation remains valid in other layers of the network would suggest that only the layer-input components (the excitations) that correspond to large absolute values of the weights of a neuron have the opportunity to significantly impact the pre-activation of a sample for this neuron, i.e. are relevant for this neuron. Hence, in section 5.1, we study whether the peakedness of the distribution of weights in a layer is correlated with generalization (as hypothesized in section 3.1.1), and confirm experimentally that this is the case. In section 5.2, we give theoretical arguments based on network structure for why the distribution of “excitation-weight” products (which we define formally) is also worth studying, and show that it behaves similarly.

5.1 Weights

This section looks at the intuition mentioned in section 3.1.1, based on results of [5], that the empirical distribution of layer weights tends to be more peaked in

models that generalize well.

5.1.1 Terminology and Notation

In order formally study this notion, we must first define what is meant by the distribution of layer weights, as well as its “peakedness.” Recall from section 2.4.5 that in convolutional networks such as the ones we use (figure 4.3 gives an example of such a network), the weights in a given layer are given by a 4-D kernel tensor $\mathbf{K}$, with $K_{i,j,k,l}$ giving the connection strength between a unit in channel i of the output and channel j of the input, with an offset of k rows and l columns between the output unit and the input unit. By “distribution of layer weights,” we simply mean the distribution of all the values in $\mathbf{K}$. For simplicity of computation, we assume that this tensor is rearranged into a vector $\mathbf{k}$ of length n .

We only consider in this section the weights of the last convolutional layer of the network, because this layer contains the largest number of weights (as can be seen in table 4.1), and should thus be the most robust to noise. Figure 5.1 shows that this assumption holds reasonably well in practice on a randomly picked model.

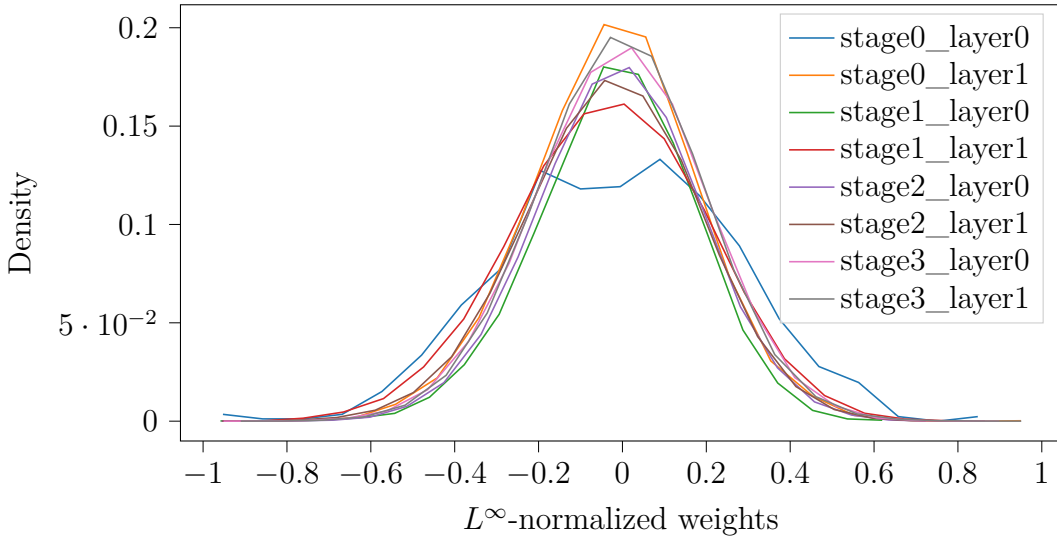


Figure 5.1: Comparison of layer weight distributions across convolutional layers of a randomly chosen network. The weights of the last layer appear to be a robust approximation to the behavior of the other layers, and we restrict our study to them in what follows.

In contrast, the definition of “peakedness” is less straightforward. The intuition given in [5] on a toy example is that networks that have a high layer rotation and

that generalize well had learned less noisy filters, i.e. filters containing a larger percentage of close-to-zero weights, for the same features, when compared to other, worse-generalizing models. To define this formally, we look at two metrics: the variance of L^∞ -normalized weights, and the kurtosis¹. More formally, the variance of normalized weights is defined as

$$\mathbb{V}_\infty[\mathbf{k}] = \mathbb{V}[\|\mathbf{k}\|_\infty] = \frac{1}{n-1} \sum_{i=1}^n \left(\frac{k_i - \bar{\mathbf{k}}}{\max_{1 \leq j \leq n} |k_j|} \right)^2, \quad (5.1)$$

where $\bar{\mathbf{k}}$ refers to the mean of $\mathbf{k}$. The kurtosis of weights is defined as [98]

$$\mathbb{K}[\mathbf{k}] = \frac{m_4}{m_2^2} = \frac{\frac{1}{n} \sum_{i=1}^n (k_i - \bar{\mathbf{k}})^4}{\left[\frac{1}{n} \sum_{i=1}^n (k_i - \bar{\mathbf{k}})^2 \right]^2}, \quad (5.2)$$

where m_2 is the second sample moment around the mean, and m_4 is the fourth sample moment around the mean. L^∞ -normalization has no effect on kurtosis; when looking at this metric, we thus do not mention explicitly whether weights are normalized or not.

Note that for the variance-based metric, a low value indicates a high “peakedness,” whereas for the kurtosis, higher values are indicative of a more peaked distribution.

5.1.2 Relation with Generalization

Figure 5.2 shows the link between the variance of the L^∞ -normalized weights as defined in (5.1) and the generalization performance of the models (note the use of the logarithmic scale on the vertical axis). It also shows the link between the kurtosis of the weight distribution, as defined in (5.2). Interestingly, a **low variance (or a high kurtosis) appears to be a sufficient condition for good generalization**, but it is not necessary for this to be the case.

This observation is particularly interesting, as it allows us to define a *significant weight component* as one whose weight strongly differs from zero. This notion is central in this thesis, and will especially be useful when analyzing results related to clustering in chapters 6 and 7. A more precise definition is given in section 6.1. Indeed, it hints at the presence of *patterns* in the neuron-level representation, a notion we will define in more detail in section 5.2 and chapter 7.

¹The erroneous notion that kurtosis measures peakedness in general has been discussed in the literature before [97]. However, in our simple, well-behaved case where the distribution is approximately normal, we expect that kurtosis still provides a reasonable measure of peakedness, or at least captures what we wish to convey. Additionally, our true use of the peakedness metric is its impact on the large magnitude weights, i.e. those in the tails of the distribution. The choice of the kurtosis as metric is then perfectly reasonable.

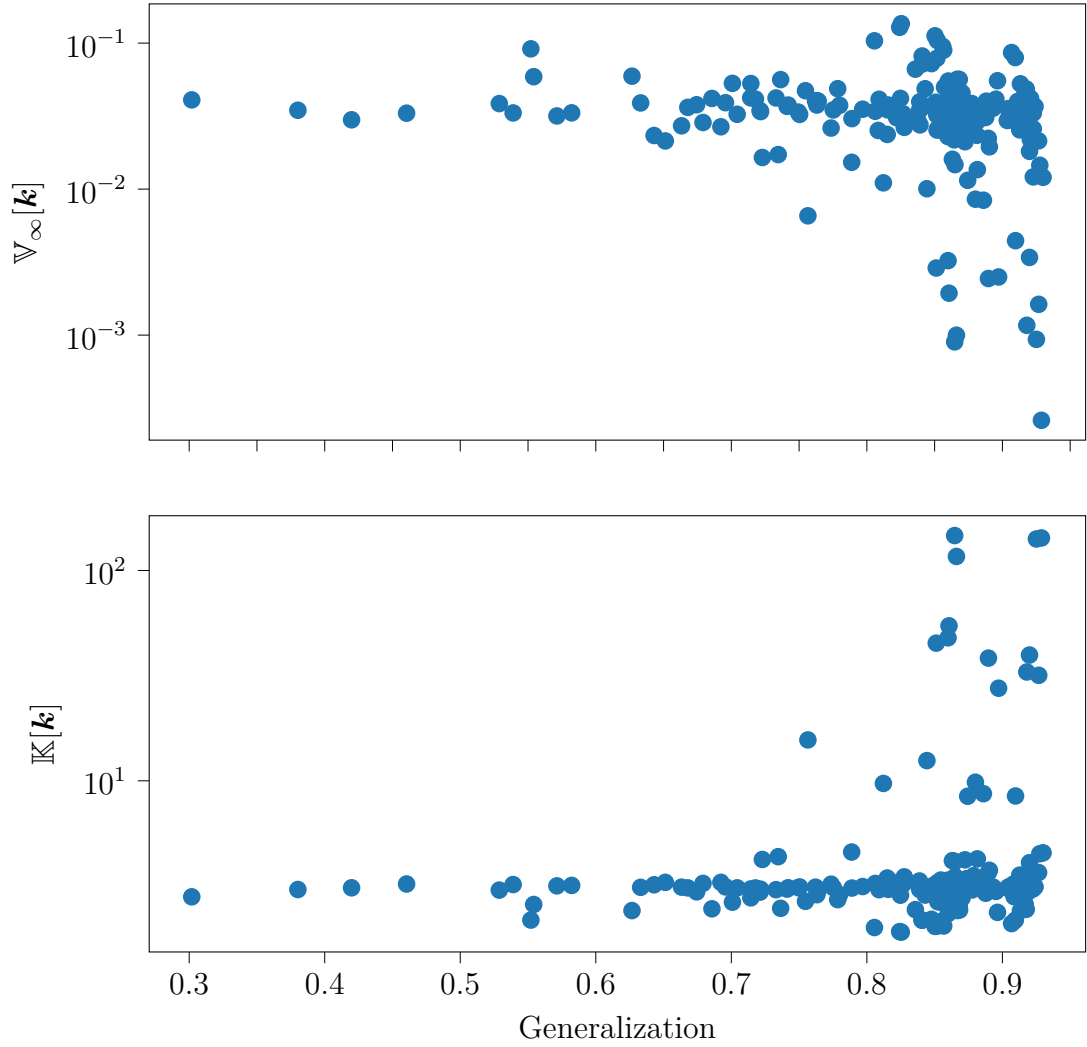


Figure 5.2: Relation between peakedness of weight distribution and generalization performance. Peakedness is measured using either variance (top) or kurtosis (bottom) of L^{∞} -normalized weights. Lower variance or higher kurtosis indicates a more “peaked” distribution. The vertical axis is logarithmically scaled. We note that low variance or high kurtosis appear to be sufficient conditions for good generalization performance, but are not necessary.

Due to the similarity of the results with both metrics, we only consider the kurtosis in the main body of the text that follows. Appendix A.1.1.2 contains the full results when using the variance of L^∞ -normalized weights as metric instead.

5.1.3 Influence of Hyperparameters

In order to better understand which hyperparameters are influencing these results most strongly, figures 5.3 through 5.5 separate the models according to their values of one given hyperparameter “family,” as presented in table 4.2. We only mention here the hyperparameters for which there is a strong link to the weight distribution kurtosis. Results for other hyperparameters and metrics can be found in appendix A.1.1.

We observe on figure 5.3 that while noise regularization has no direct effect on the kurtosis of the weight distribution, it could potentially help explain why a high kurtosis appeared to be a sufficient condition for good generalization performance in figure 5.2. Indeed, the noise regularization hyperparameter puts some models at a strong generalization disadvantage compared to others, which obscures the fact that for a given choice of noise regularization, models with high weight distribution kurtosis have good generalization performance, and vice versa.

Note that data augmentation tremendously improves generalization performance, and that the optimal dropout rate without data augmentation appears to lie between 0 and 0.4.

Figure 5.4 shows that the effective learning rate (and in particular, its weight decay component) has a very strong impact on kurtosis. This intuitively makes sense when one considers that weight decay tends to penalize models with large weights, as discussed in section 2.3.2.

Finally, figure 5.5 shows that models trained with Adam (algorithm 2.5) tend to have a much higher weight distribution kurtosis than their counterparts trained using stochastic gradient descent with momentum (algorithm 2.4).

5.2 Excitation-Weight Products

The observation that a high weight distribution peakedness is not always a necessary condition for good generalization performance can be explained theoretically by thinking about how information is propagated through the network layers. Indeed, the pre-activation is computed as a sum of products between weights and excitations; we call these *excitation-weight products*, or EW products. This section uses much of the notation and terminology that was described in section 5.1.1. Note that rather than using the vectorized weights in (5.1) and (5.2), we use the vector $\mathbf{p}$ of

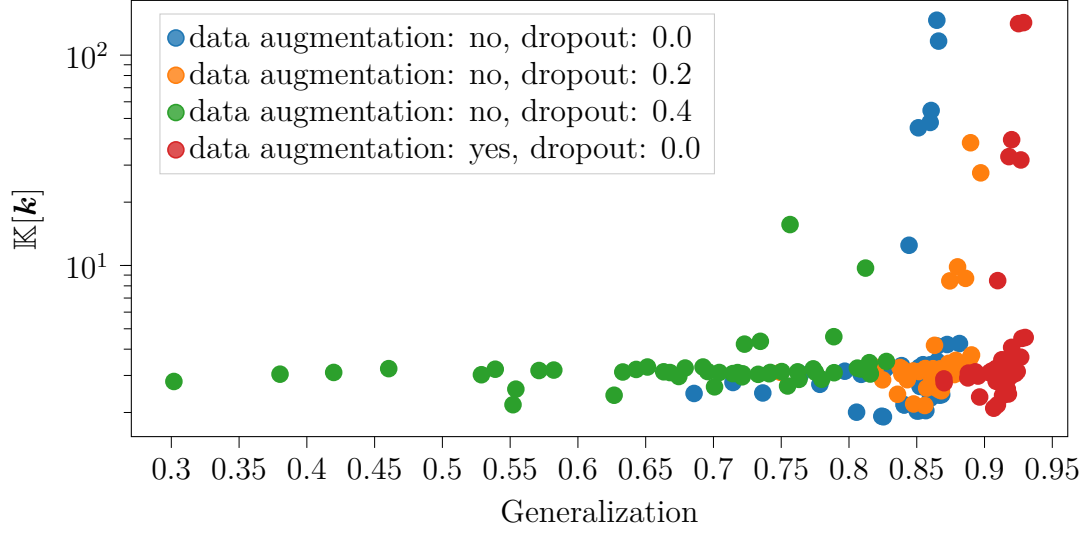


Figure 5.3: Influence of noise regularization hyperparameter on kurtosis of weight distribution. Each color corresponds to a different value of the noise regularization hyperparameter tuple. The vertical axis is logarithmically scaled. The noise regularization strongly impacts generalization performance, while barely affecting kurtosis. This is partly why the results in figure 5.2 were not conclusive.

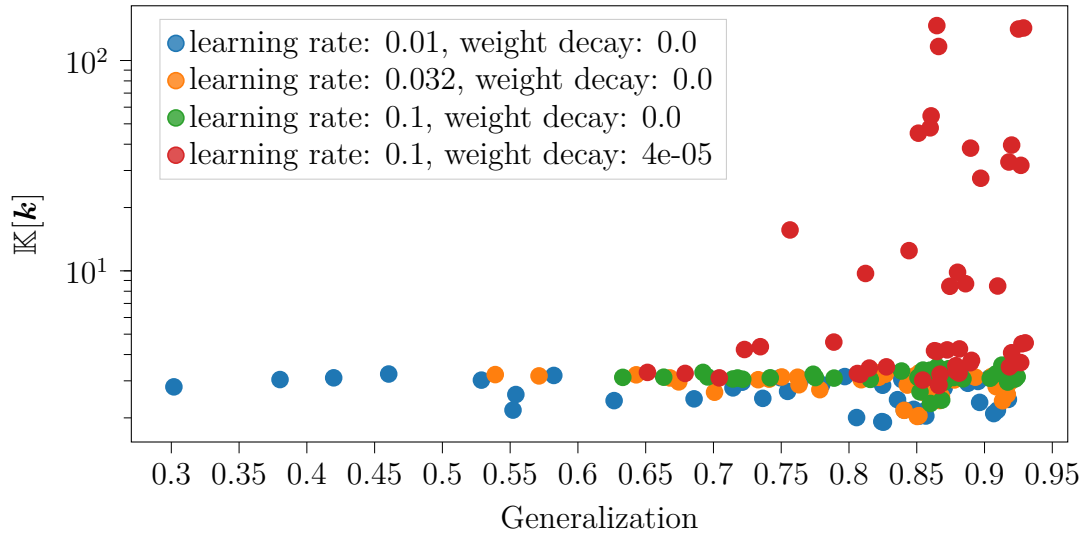


Figure 5.4: Influence of effective learning rate hyperparameter on kurtosis of weight distribution. Each color corresponds to a different value of the effective learning rate hyperparameter tuple. The vertical axis is logarithmically scaled. We observe that weight decay, in accordance with the theory, penalizes large weights and thus leads to a more peaked distribution.

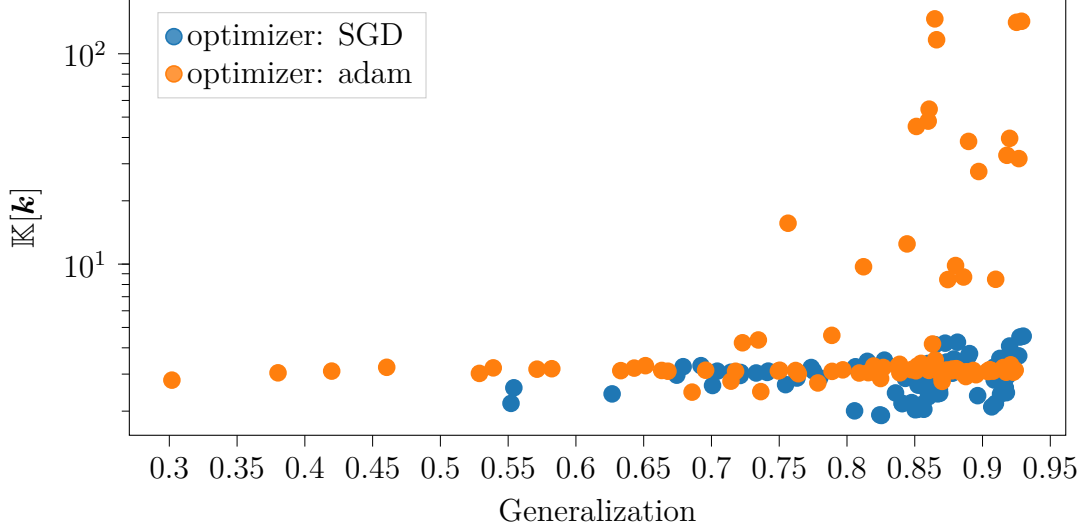


Figure 5.5: Influence of optimizer hyperparameter on kurtosis of weight distribution. Each color corresponds to a different value of the optimizer. The vertical axis is logarithmically scaled. The Adam optimization algorithm clearly leads to an increase in weight distribution peakedness.

all $V_{l,j+m-1,k+n-1}K_{i,l,m,n}$ products in (2.51):

$$\mathbb{V}_{\infty}[\mathbf{p}] = \mathbb{V}[\|\mathbf{p}\|_{\infty}] = \frac{1}{n-1} \sum_{i=1}^n \left(\frac{p_i - \bar{\mathbf{p}}}{\max_{1 \leq j \leq n} |p_j|} \right)^2, \quad (5.3)$$

$$\mathbb{K}[\mathbf{p}] = \frac{m_4}{m_2^2} = \frac{\frac{1}{n} \sum_{i=1}^n (p_i - \bar{\mathbf{p}})^4}{\left[\frac{1}{n} \sum_{i=1}^n (p_i - \bar{\mathbf{p}})^2 \right]^2}. \quad (5.4)$$

Section 5.2.1 discusses why studying EW products is important.

5.2.1 Motivation

Trivially, the EW products are zero when the weights implicated in them are zero. However, zeroing the weights is not the only way the network has of zeroing the EW products; it could also learn to zero excitations; this is a potential explanation for why some models have good generalization despite not having extremely peaked weight distributions. The precise utility of the ability to zero out excitations is detailed in chapter 7, but we give a preview here. In chapter 7, we explain how the significant weight components can be used to define “patterns,” which are simply sets of high-magnitude weight components, forming a mask through which the neuron views the important components of its input. As we will show in section 7.3,

these patterns help with classification. However, one neuron can be equipped with weights allowing the detection of multiple patterns, thus potentially making its weight distribution less peaked when these patterns have little overlap. This means that not all high-magnitude weights should be getting multiplied by large excitations in the computation of the pre-activation.

In the simplest case, consider a neuron able to detect k input patterns $\mathcal{P}_1$ through $\mathcal{P}_k$, with each pattern being a set of indices in $\mathbf{k}$. Ideally, this means that $|k_i| \gg 0$ if $i \in \bigcup_{1 \leq j \leq k} \mathcal{P}_j$, and $k_i \approx 0$ otherwise. When a sample $\mathbf{x}^{(i)}$ is being fed as input to the model, assuming this sample activates pattern $\mathcal{P}_l$ in that neuron, then we want the excitations entering indices in $(\bigcup_{1 \leq j \leq k} \mathcal{P}_j) \setminus \mathcal{P}_l$ to be zeroed out, lest they add noise to the classification process.

We have thus given a theoretical argument for why EW products should be peaked at least as much, and very likely even more than the weight distribution. Section 5.2.2 demonstrates that these hypotheses hold, and shows how the statistical properties of the excitation-weight product distribution relate to generalization performance, while section 5.2.3 details the influence of the various hyperparameters on this relation.

5.2.2 Relation with Generalization

Figure 5.2 shows the link between the variance of the L^∞ -normalized EW products defined in (5.3) and the generalization performance of each model (note the use of the logarithmic scale on the vertical axis). It also shows the link between the kurtosis of the EW product distribution, as defined in (5.4). We see that EW product peakedness is an even stronger sufficient condition for generalization than the weight peakedness, but not a necessary one. Note that as we predicted in section 5.2.1, the distribution of EW products is much more peaked than the corresponding distribution of weights.

Due to the similarity of the results with both metrics, we only consider the kurtosis in the main body of the text that follows. Appendix A.2.1.2 contains the full results when using the variance of L^∞ -normalized EW products as metric instead.

5.2.3 Influence of Hyperparameters

As we did in section 5.1.3, we now look at the influence of the various hyperparameters of the model on the relation between EW product peakedness and generalization performance. We only show hyperparameters for which there is a strong link to the EW product distribution kurtosis; results for other hyperparameters and metrics can be found in appendix A.2.1.

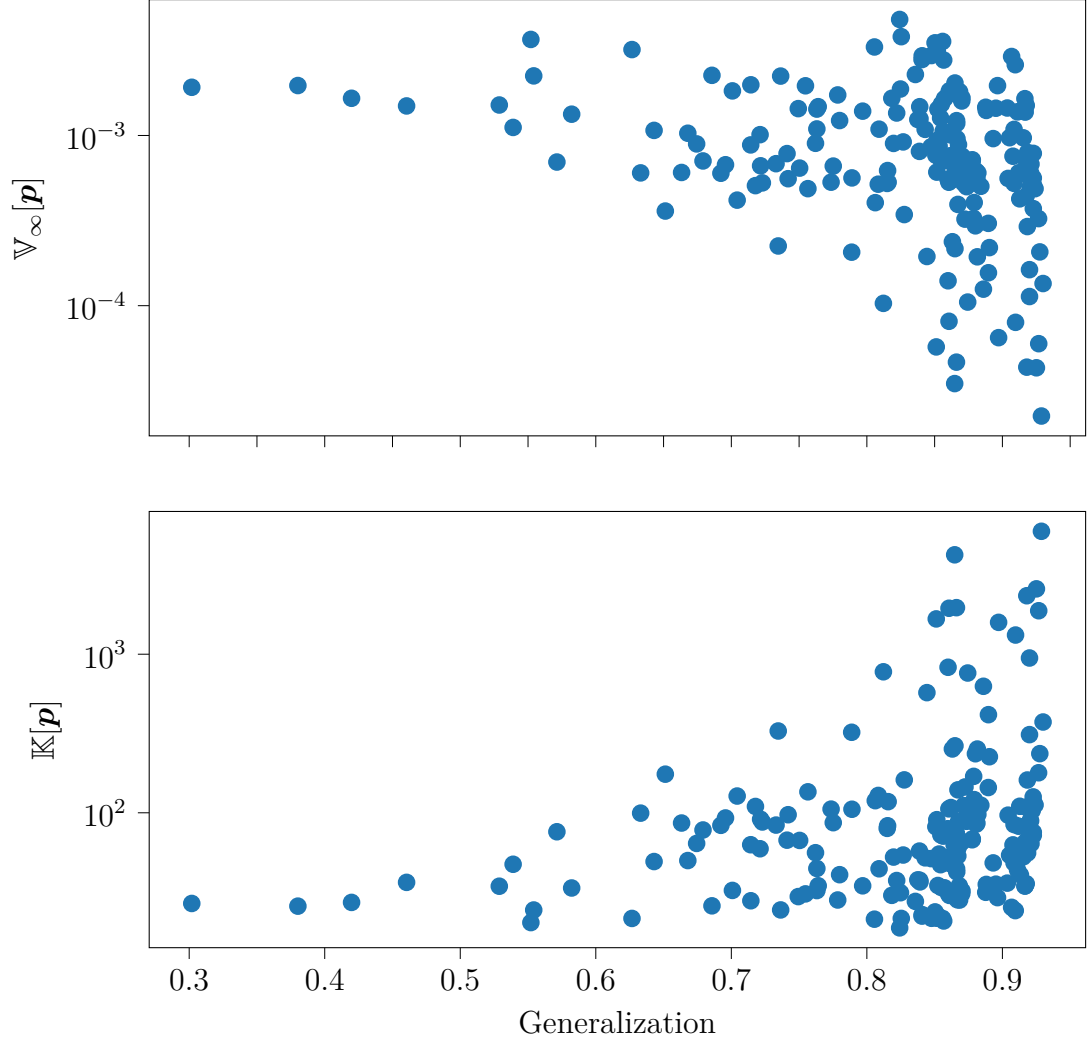


Figure 5.6: Relation between peakedness of EW product distribution and generalization performance. Peakedness is measured using either variance (top) or kurtosis (bottom) of L^∞ -normalized EW products. Lower variance or higher kurtosis indicates a more “peaked” distribution. The vertical axis is logarithmically scaled. We note that low variance or high kurtosis appear to be sufficient conditions for good generalization performance (even more so than when simply considering the weights), but are not necessary.

We observe on figure 5.7 the same thing that we observed on figure 5.3 for the weight distribution, namely that noise regularization explains the large differences in generalization performance between models. Looking only at models within one noise regularization “family,” we observe that a peaked EW product distribution is both sufficient and necessary for good generalization.

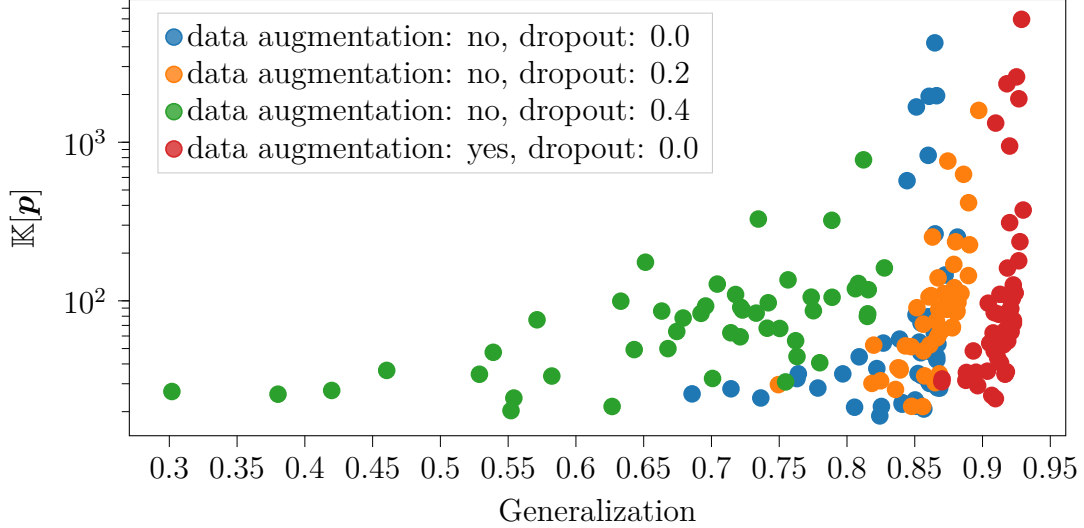


Figure 5.7: Influence of noise regularization hyperparameter on kurtosis of EW product distribution. Each color corresponds to a different value of the noise regularization hyperparameter tuple. The vertical axis is logarithmically scaled. The noise regularization mainly impacts generalization performance. This is partly why in figure 5.6, we concluded that peakedness is not a necessary condition for good generalization, when in fact it appears to be once noise regularization is accounted for.

Figure 5.8 shows that the effective learning rate (and in particular, its weight decay component) has a very strong impact on kurtosis. This intuitively makes sense when one considers that weight decay tends to penalize models with large weights, as discussed in section 2.3.2. This observation is thus in line with what we observed for the weights, although the EW products make it even more apparent.

Finally, figure 5.9 shows that models trained with Adam (algorithm 2.5) tend to have a much higher EW product distribution kurtosis than their counterparts trained using stochastic gradient descent with momentum (algorithm 2.4).

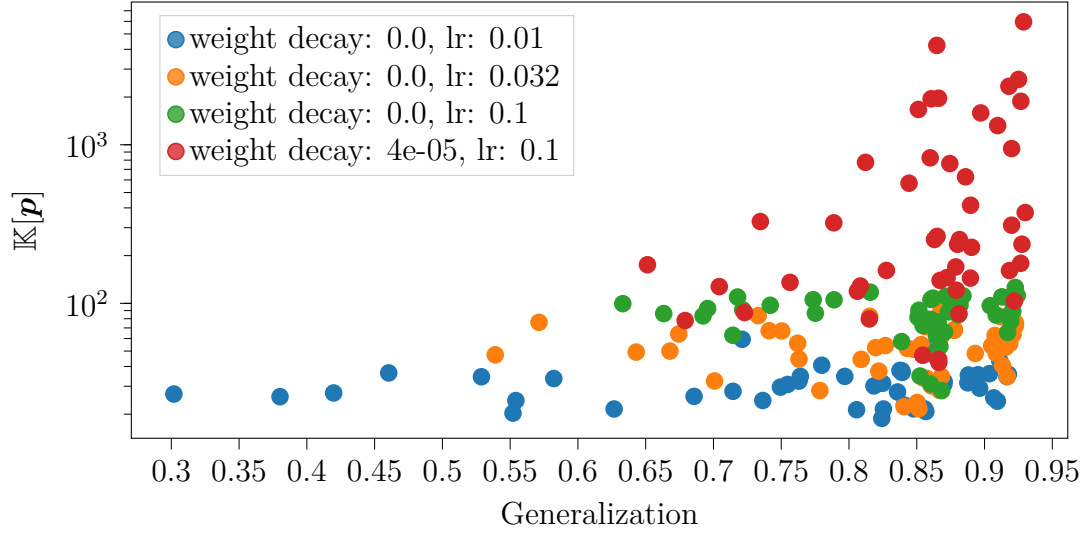


Figure 5.8: Influence of effective learning rate hyperparameter on kurtosis of EW product distribution. Each color corresponds to a different value of the effective learning rate hyperparameter tuple. The vertical axis is logarithmically scaled. We observe that weight decay, in accordance with the theory, penalizes large weights and thus leads to a much more peaked distribution.

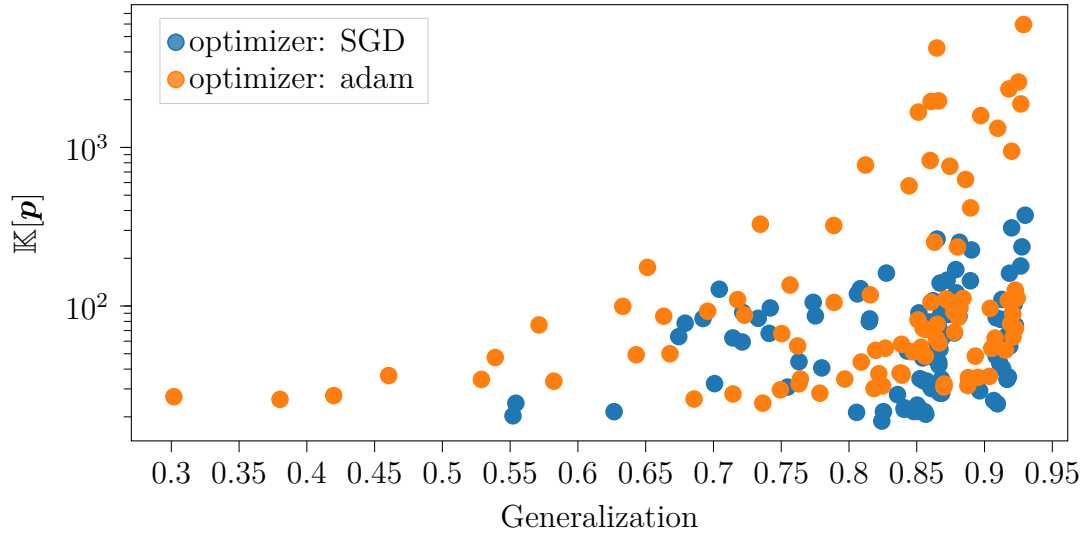


Figure 5.9: Influence of optimizer hyperparameter on kurtosis of EW product distribution. Each color corresponds to a different value of the optimizer. The vertical axis is logarithmically scaled. The Adam optimization algorithm clearly leads to an increase in EW product distribution peakedness.

5.3 Conclusion

In this chapter, we have laid the foundations for the investigations of the next chapters, through the introduction of significant components in section 5.1, and patterns in section 5.2. More specifically, we have shown strong indications that networks with good generalization performance tend to learn weights whose distribution is peaked. We have also shown that networks learn configurations which allow them to have a very peaked distribution of the so-called excitation-weight products.

In chapter 6, we revisit this notion of significant component, and show that it is fundamental to the emerging behavior of neurons as binary classifiers. Then, in chapter 7, we look at what these significant components truly represent at the neuron level, namely the union of a set of pattern indices.

Chapter 6

Neurons as Binary Classifiers

In this section, we look at the implicit partition created by neurons as they are activated for certain internal representations of samples, and not for others. We henceforth call this partition the *activation partition*. By studying this binary partition, and how its quality relates to generalization, we hope to learn more about the generalization properties of neural networks. Our motivation for studying this phenomenon is explained in section 3.1.2, which covers the ideas presented in [6].

We also look at the influence of the number of significant components on this partition quality.

6.1 Terminology and Notation

In this section, we describe some terminology and notation that we use throughout this chapter as well as chapter 7.

6.1.1 Terminology

Let us start by defining some terminology that we use throughout the rest of this thesis. We say a neuron *activates* a sample $\mathbf{x}^{(i)}$ when the pre-activation value computed by that neuron when $\mathbf{x}^{(i)}$ is given as input is strictly positive (or equivalently, when the activation value is strictly positive). Conversely, the neuron is not activated (or *inhibited*) when the pre-activation is negative (or when the activation is zero). Note that strictly speaking, the computed pre-activation only looks at the internal representation (i.e. representation at the neuron level) of the sample given as input, not necessarily at the original sample as it was given as input.

Next, we formalize the notion of *significant component*, introduced in section 5.1.2. Indeed, our previous definition was purposely vague. We now define a

significant weight component as a weight component whose magnitude is larger than a certain percentile of the vector of weights $\mathbf{k}$. When we say section 6.3 looks at the influence of the number of significant components, we mean it looks at the influence of the percentile threshold.

6.1.2 Metrics to Assess the Binary Partition

In order to evaluate the quality of a binary partition, we make use of the literature on clustering and information theory. As we mentioned earlier, our goal is to study the quality of the binary partition one gets at the neuron-level when labeling the internal representations of samples according to whether they activate the neuron in question, or not. More specifically, we wish to know how the quality of this partition, which we denote $\mathcal{A}$, relates to the generalization performance of the model in question.

We look at three tools of doing so:

- The *silhouette* [99] of the implicit partition. The silhouette is a value between -1 and 1 measuring how well a data point is matched to the cluster it is assigned to.

More formally, let data point i be assigned to cluster $\mathcal{C}_j$. We define two auxiliary measures:

$$a(i) = \frac{1}{|\mathcal{C}_j| - 1} \sum_{k \in \mathcal{C}_j, k \neq i} d(i, k), \quad (6.1)$$

$$b(i) = \min_{k \neq j} \frac{1}{|\mathcal{C}_k|} \sum_{l \in \mathcal{C}_k} d(i, l). \quad (6.2)$$

The first, $a(i)$, defines the mean distance from i to the other points in its cluster, whereas $b(i)$ is the minimum mean distance from i to the points in another cluster¹. From this, we define the silhouette of i as

$$s(i) = \begin{cases} \frac{b(i) - a(i)}{\max\{a(i), b(i)\}} & \text{if } |\mathcal{C}_j| > 1, \\ 0, & \text{otherwise.} \end{cases} \quad (6.3)$$

Finally, we define the *mean silhouette* [100] as

$$\bar{s} = \frac{1}{n} \sum_{i=1}^n s(i). \quad (6.4)$$

We sometimes refer to the latter as simply “silhouette,” and rarely consider the silhouette for a single data point.

¹In theory, d can be any distance metric, but unless specified otherwise, we consider it to be the Euclidean distance. This choice is motivated by the fact that the k -means algorithms we use bases its labeling on the Euclidean distance metric.

- The *variation of information* (VI) [101] between the activation partition and a reference clustering. The VI is an information theoretic metric for comparing two clusterings $\mathcal{C}$ and $\mathcal{C}'$ of the same dataset.

Consider the probability of a point i being in cluster $\mathcal{C}_k$:

$$P(k) = \frac{|\mathcal{C}_k|}{n}. \quad (6.5)$$

The discrete random variable (written upright to distinguish it from a simple iteration variable) k is thus uniquely associated to the clustering $\mathcal{C}$, which we assume contains K clusters. The *entropy* associated with clustering $\mathcal{C}$ is given by

$$H(\mathcal{C}) = - \sum_{k=1}^K P(k) \log P(k). \quad (6.6)$$

From this, we can define the *mutual information* between two clusterings $\mathcal{C}$ and $\mathcal{C}'$, i.e. the information one has about the other. Let $P(k, k')$ represent that a point belongs to $\mathcal{C}_k$ in $\mathcal{C}$ and $\mathcal{C}_{k'}$ in $\mathcal{C}'$:

$$P(k, k') = \frac{|\mathcal{C}_k \cap \mathcal{C}_{k'}|}{n}. \quad (6.7)$$

The mutual information $I(\mathcal{C}, \mathcal{C}')$ is then given by the mutual information between the associated random variables k and k' :

$$I(\mathcal{C}, \mathcal{C}') = \sum_{k=1}^K \sum_{k'=1}^{K'} P(k, k') \log \frac{P(k, k')}{P(k)P'(k')}. \quad (6.8)$$

Finally, we then define the variation of information as

$$VI(\mathcal{C}, \mathcal{C}') = H(\mathcal{C}) + H(\mathcal{C}') - 2I(\mathcal{C}, \mathcal{C}') \quad (6.9)$$

$$= H(\mathcal{C} | \mathcal{C}') + H(\mathcal{C}' | \mathcal{C}) \quad (6.10)$$

$$= 2H(\mathcal{C}, \mathcal{C}') - H(\mathcal{C}) - H(\mathcal{C}'), \quad (6.11)$$

where $H(\mathcal{C} | \mathcal{C}') = H(\mathcal{C}) - I(\mathcal{C}, \mathcal{C}')$ represents the conditional entropy, and $H(\mathcal{C}, \mathcal{C}')$ is the joint entropy of the two clusterings, or equivalently, the entropy of $P(k, k')$.

- The ratio of cluster sizes. We compare both clusterings on the ratio $|\mathcal{C}_{\min}|/|\mathcal{C}_{\max}|$, i.e. the ratio between the size of the smaller cluster and the size of the larger cluster.

2-means as Reference Binary Partition Note that the variation of information is a tool for comparing clusterings; we thus also make use of the k -means algorithm [102, 103, 104, 105, 106] to find the reference clustering for a given neuron, with $k = 2$. We refer to this reference clustering as the *2-means clustering*, $\mathcal{R}$. We are thus comparing the quality of the implicit activation-based clustering to the best possible binary partition of the data.

6.2 Experimental Study of Binary Partition Characteristics and Generalization

As explained in section 5.2, the pre-activation (and hence also the activation) of a given neuron for a certain sample is computed as a sum of products of weights and excitations. Each weight component thus has an associated excitation. As we discussed previously in sections 5.1 and 5.2, only the components with non negligible weights (which we call *significant components*) are important. In order to define the clustering task formally, we make use of this notion: rather than clustering the data in the representation space containing all components, we only look at the excitations associated to significant weight components. To illustrate this, consider the vectorized weights $(0, 0.9, 0.03, 0.8, 0.93)$, with associated vectorized excitations $(0.4, 0.7, 0.2, 0.03, 0.9)$, with the significance threshold being 60 %. The data point to which we associate a label would then be $(0.7, 0.9)$. This masking process is also shown graphically on figure 4.2.

There are various reasons, theoretical and empirical, to make this choice:

- computation time to find the reference clusterings is reduced;
- considering components with negligible weights makes little sense, as they do not matter in the pre-activation computation, and taking them into account would needlessly add noise to the process;
- our experiments from chapter 5 show that models which generalize well have a tendency to have a very limited set of non negligible weights;
- pruning even a fairly high percentage of weights does not affect accuracy, indicating that only a few weight components are truly involved in the final decision-making process [107, 108, 109].

In section 6.3, we study the influence of the number of significant components in more detail, but for now we consider the top-1 % magnitude weight components. Results for other percentages are given in appendix B. For some of these figures, a handful of models had to be omitted because of artefacts of the randomization process.

6.2.1 As Measured Using Silhouette

6.2.1.1 Relation with Generalization

Figure 6.1 shows a side-by-side view of the mean silhouette score for the activation partition and the reference clustering obtained with 2-means. We limit ourselves to only considering top-1 % magnitude weight components, and show results for other percentages in appendix B.1.1. We observe the following:

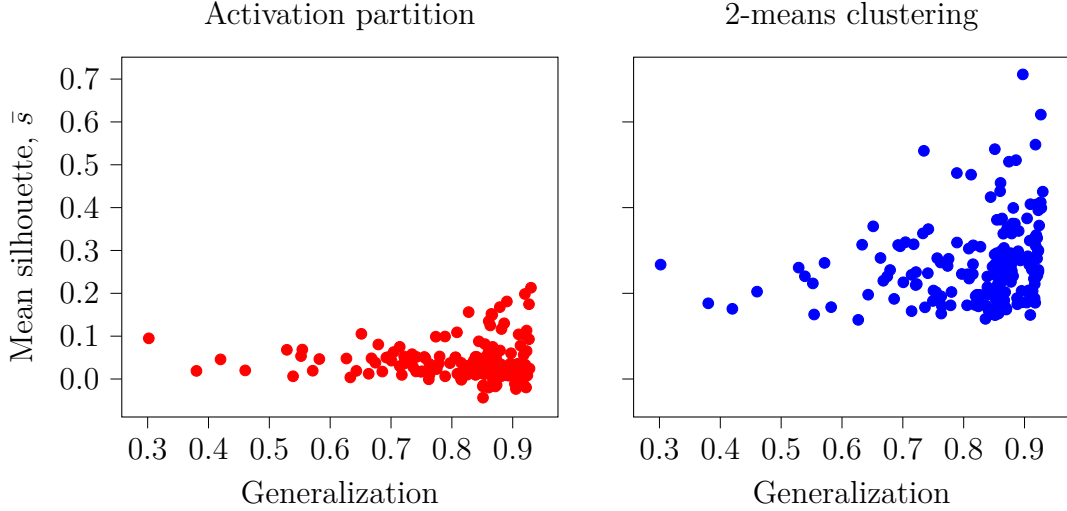


Figure 6.1: Comparison of mean silhouette of activation partition and 2-means clustering with internal representation using top-1 % magnitude weight components. As expected, the mean silhouette of the activation partition is lower than the silhouette of the 2-means clustering, but well-generalizing models tend to have better implicit partitions, and are more easily clustered with 2-means than their worse-generalizing counterparts.

- The mean silhouette of the activation partition is lower than the mean silhouette of the reference clustering; this is not surprising, when one considers that the 2-means algorithm minimizes an objective closely related to what the silhouette measures.
- Models whose internal representation is well-clustered (according to the silhouette metric) both by the activation partition and 2-means are more likely to have good generalization performance.

6.2.1.2 Influence of Hyperparameters

Figures 6.2 through 6.5 show how these plots look when we isolate families of hyperparameters, as we did in section 5.1.3. We again limit ourselves to only considering top-1 % magnitude weight components, and only show the hyperparameters that are worth further discussing; for the sake of completeness, we show results for other threshold percentages and hyperparameters in appendix B.1.1.

Noise Regularization Figure 6.2 shows the various values of the noise regularization hyperparameter, for the plots of figure 6.1. We observe that noise

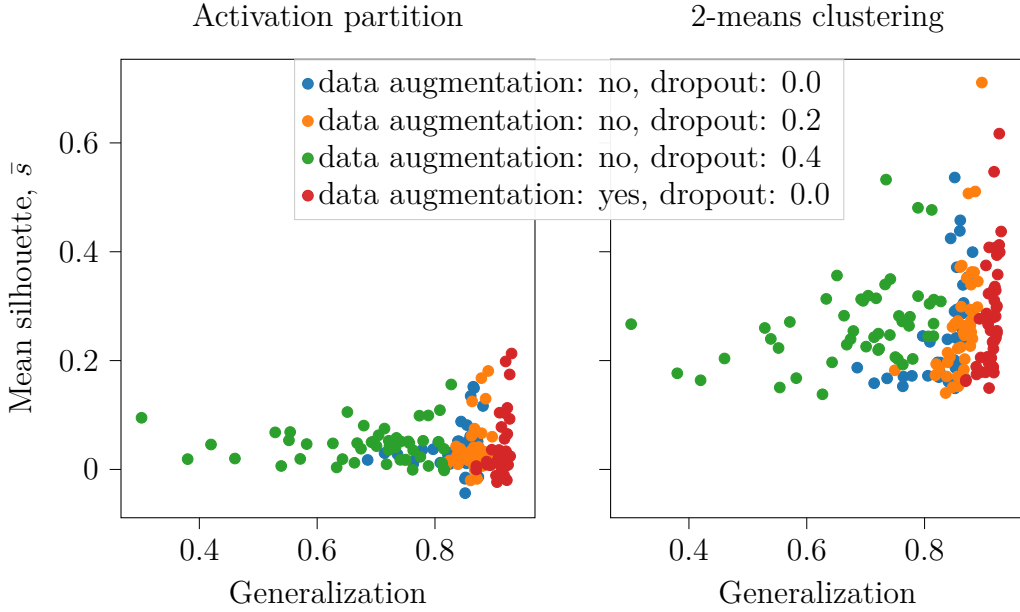


Figure 6.2: Influence of noise regularization on mean silhouette with internal representation using top-1 % magnitude weight components. As we already observed in figure 5.3, the noise regularization accounts for most of the difference in generalization performance; within one noise regularization “family,” better-generalizing models tend to display better implicit clustering, and can be clustered more effectively using 2-means (according to the mean silhouette).

regularization is largely responsible for the large differences in generalization performance between models, and that for a given noise regularization choice, models that generalize better also have better activation partition clustering and 2-means clusterability (as measured with mean silhouette) than models that do not generalize as well.

Architecture Figure 6.3 shows the various values of the architecture hyperparameter, for the plots of figure 6.1. We observe that the deeper models in green

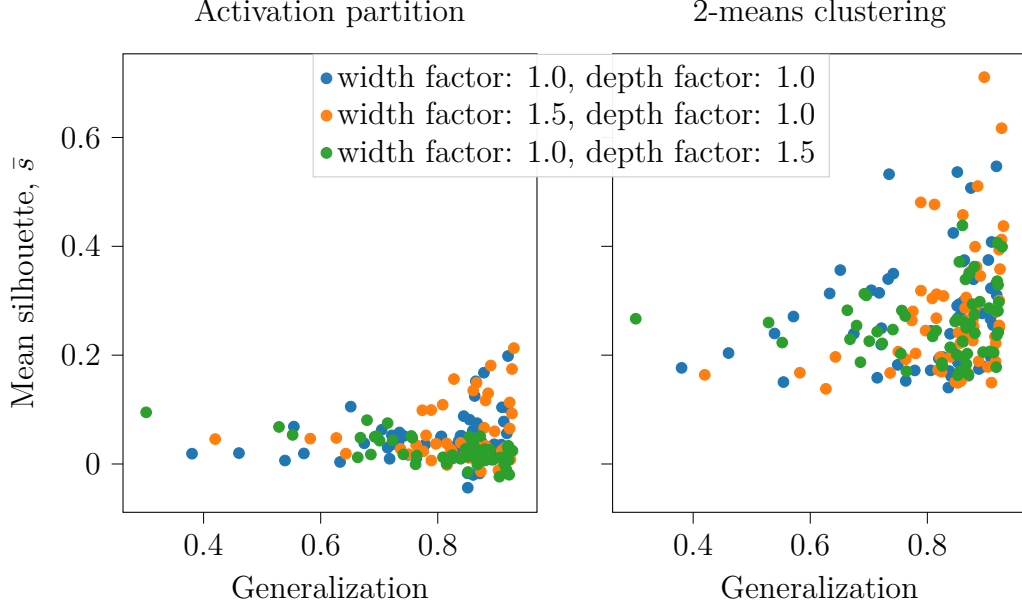


Figure 6.3: Influence of architecture on mean silhouette with internal representation using top-1 % magnitude weight components. We see that models with greater depth (in green) have worse implicit clustering and 2-means clusterability (as measured by silhouette) than their shallower counterparts.

have a lower mean silhouette than the more shallow models in blue and orange, both for the implicit partition and 2-means clustering.

Effective Learning Rate Figure 6.4 shows the various values of the effective learning rate hyperparameter, for the plots of figure 6.1. We observe two things:

- models with weight decay have better implicit clustering than models without; and
- increasing the learning rate of a model also increases its 2-means clusterability, as does adding weight decay.

Optimizer Figure 6.5 shows the influence of various optimizers, for the plots of figure 6.1. We observe that models trained with stochastic gradient descent with momentum have better implicit clustering than models trained with Adam, but perform worse from the point of view of 2-means clusterability.

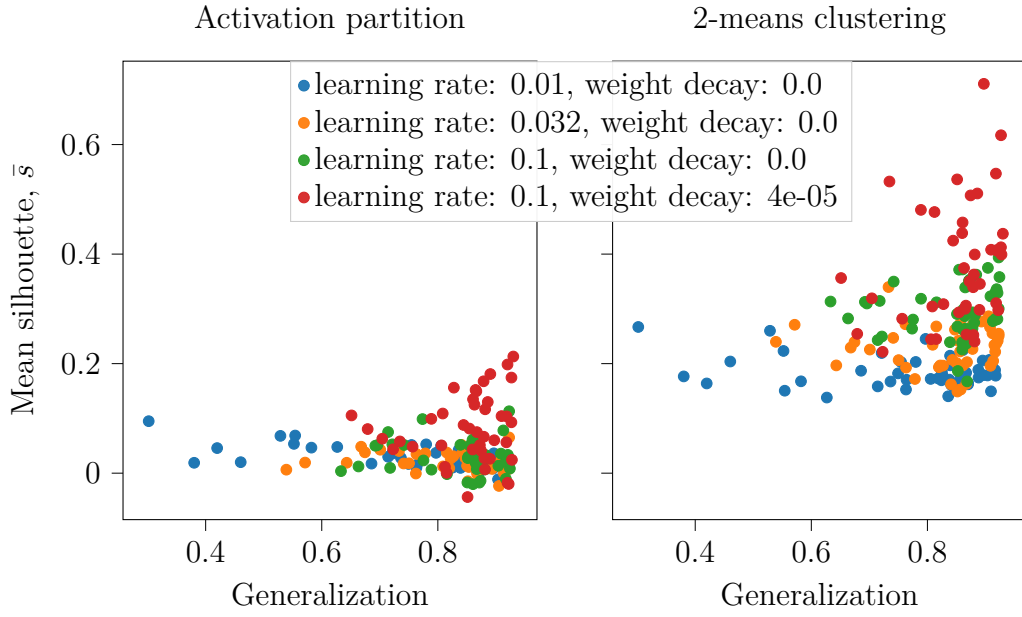


Figure 6.4: Influence of effective learning rate on mean silhouette with internal representation using top-1 % magnitude weight components. We observe two things: first, models with weight decay have better implicit clustering (as measured using silhouette) than their non regularized counterparts; second, increasing the learning rate of a model increases its 2-means clusterability, and adding weight decay on top does as well.

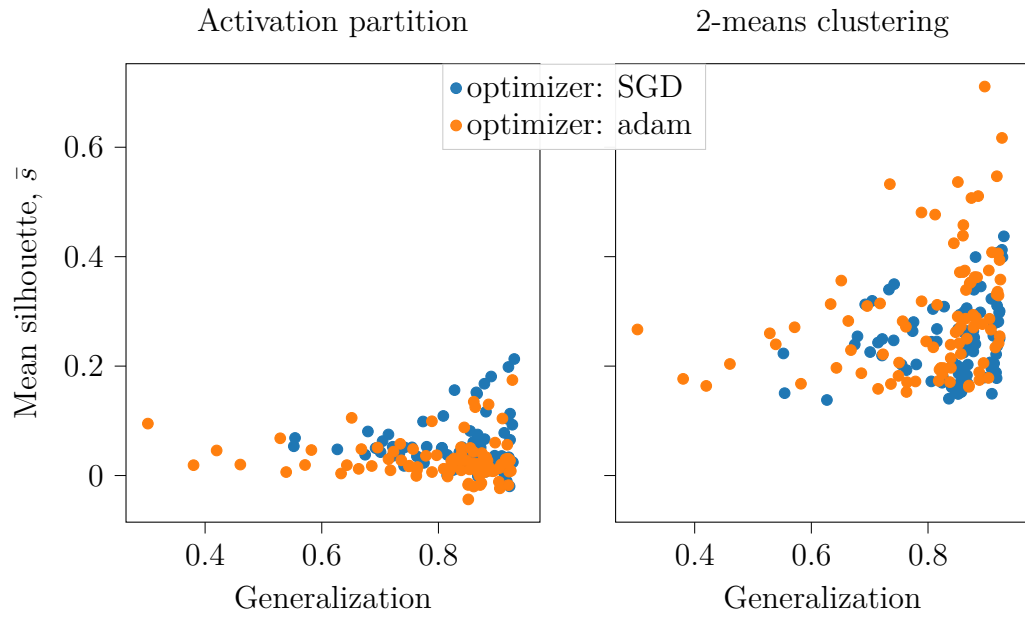


Figure 6.5: Influence of optimizer on mean silhouette with internal representation using top-1 % magnitude weight components. We observe that while models trained with SGD (with momentum) have better implicit clustering than models trained with Adam, the opposite is true for their 2-means clusterability.

6.2.2 As Measured Using Information Theory

6.2.2.1 Relation with Generalization

Figure 6.6 shows the variation of information, as defined in (6.9), between the activation partition and reference 2-means clustering. We limit ourselves to the case where only the top-1 % magnitude weight components are kept; results for other percentages are in appendix B.1.2. We observe that the variation of information is

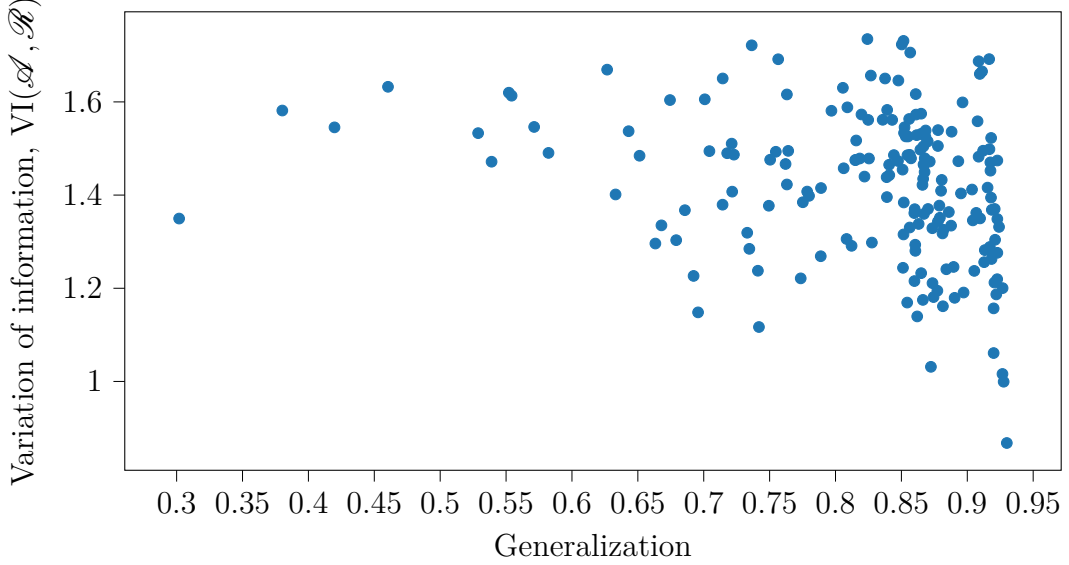


Figure 6.6: Variation of information of activation partition and 2-means clustering with internal representation using top-1 % magnitude weight components. We observe that for models that generalize well, the variation of information between the activation partition and the reference 2-means clustering decreases, i.e. they become more similar.

noticeably smaller for models with good generalization performance. This means that the activation partition that is implicitly created gets closer to the optimal 2-means partition of the space.

6.2.2.2 Influence of Hyperparameters

Effective Learning Rate Figure 6.7 shows the influence of the effective learning rate hyperparameter on the variation of information between $\mathcal{A}$ and $\mathcal{R}$. We see that models with a higher learning rate tend to arrive at configurations for which the activation partition $\mathcal{A}$ is closer to the reference 2-means clustering $\mathcal{R}$, as measured by the variation of information. Similarly, adding weight decay also decreases that metric, indicating more similar clusterings.

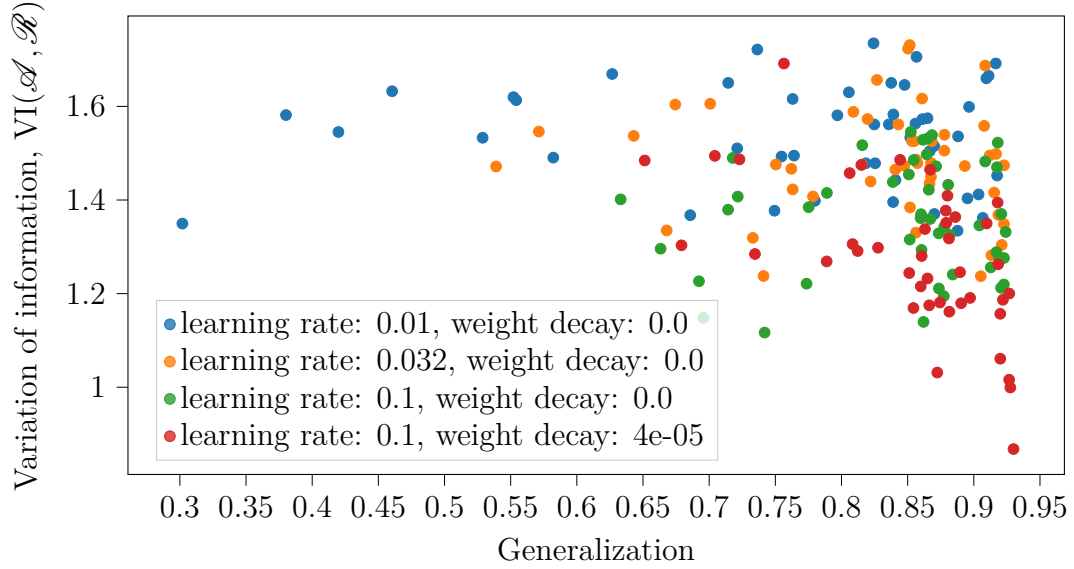


Figure 6.7: Influence of effective learning rate on variation of information with internal representation using top-1 % magnitude weight components. We observe that increasing the learning rate leads to models with an activation partition that is closer to the reference 2-means clustering, as does adding weight decay.

Optimizer Figure 6.8 shows the influence of the effective learning rate hyperparameter on the variation of information between $\mathcal{A}$ and $\mathcal{R}$. We see that models trained using SGD with momentum have lower variation of information between $\mathcal{A}$ and $\mathcal{R}$ than their counterparts trained using Adam.

Batch Size Figure 6.9 shows the influence of the batch size hyperparameter on the variation of information between $\mathcal{A}$ and $\mathcal{R}$. We see that models with a smaller batch size have a lower variation of information between $\mathcal{A}$ and $\mathcal{R}$, indicating a higher level of overlap between the two clusterings.

6.2.3 As Measured Using Cluster Size

6.2.3.1 Relation with Generalization

Figure 6.10 shows a comparison the ratio of cluster sizes for the activation partition and reference 2-means clustering. We limit ourselves to the case where only the top-1 % magnitude weight components are kept; results for other percentages are in appendix B.1.3. We observe two things:

- the cluster sizes of the activation partition do not present a clear link to generalization performance; but

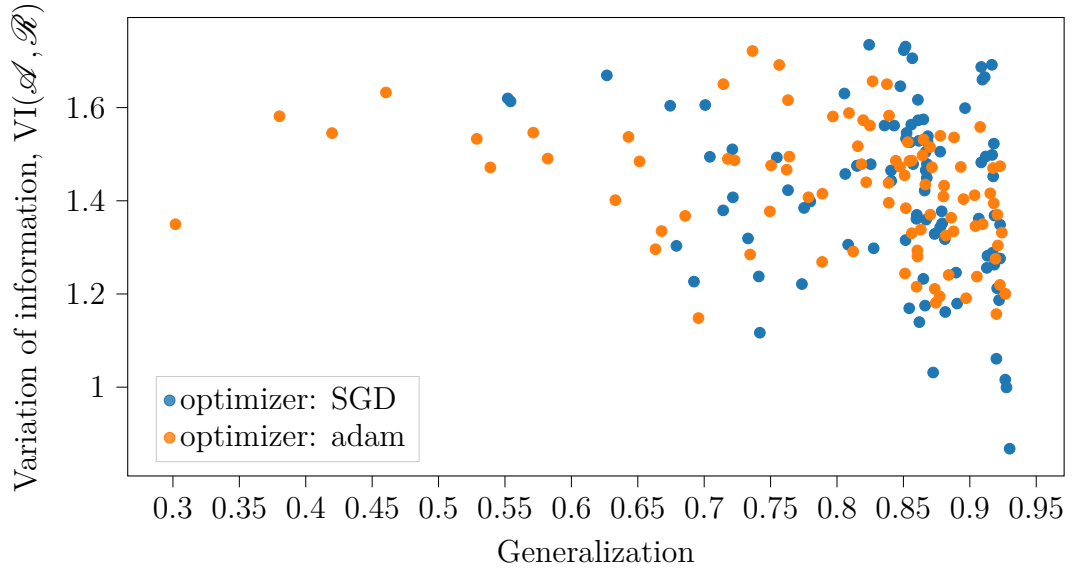


Figure 6.8: Influence of optimizer on variation of information with internal representation using top-1 % magnitude weight components. We observe that models trained using SGD with momentum arrive at lower values of the variation of information between the activation partition and the 2-means reference clustering.

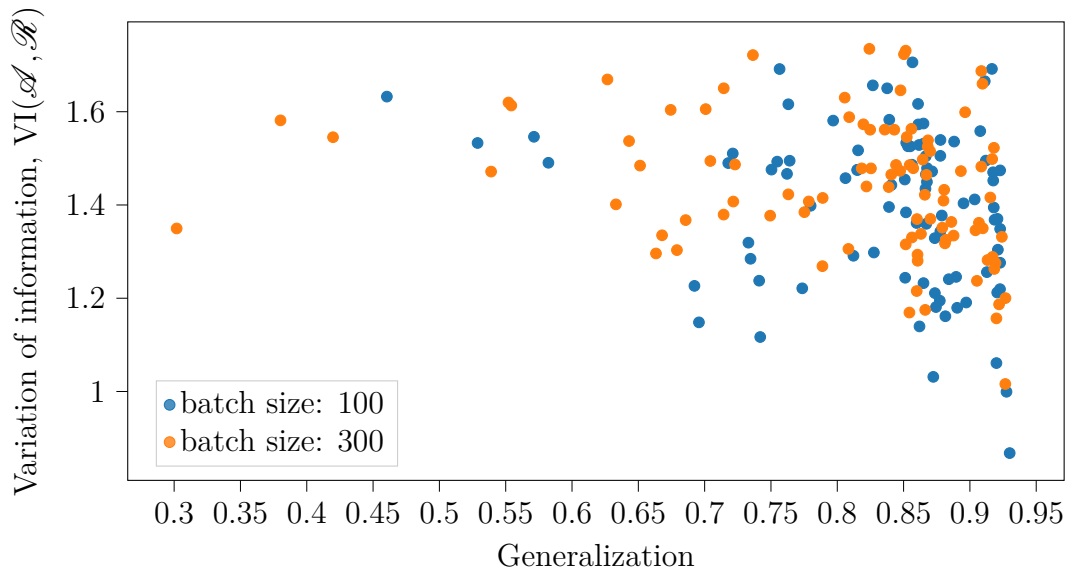


Figure 6.9: Influence of batch size on variation of information with internal representation using top-1 % magnitude weight components. We observe that a smaller batch size tends to give models with more similar activation partition and 2-means reference clustering.

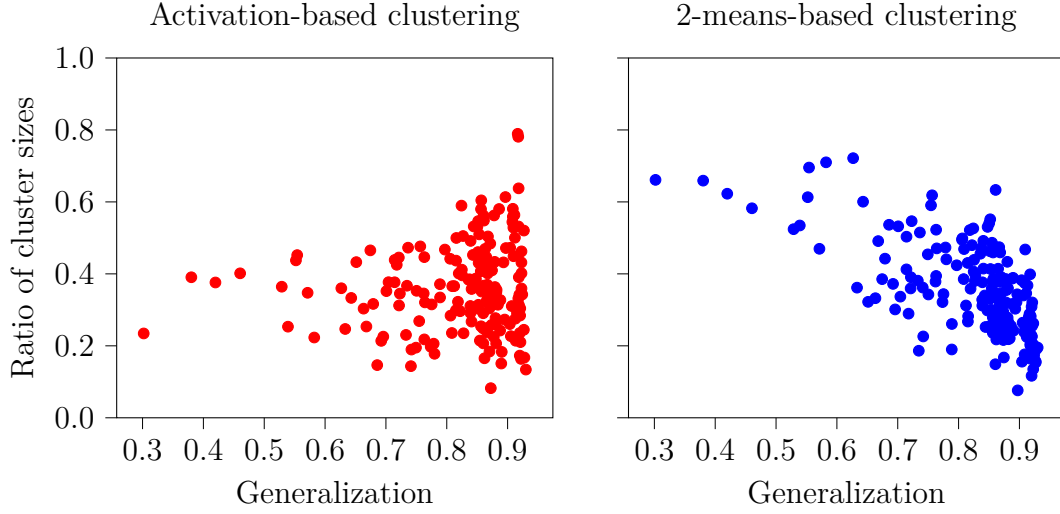


Figure 6.10: Comparison of cluster size of activation partition and 2-means clustering with internal representation using top-1 % magnitude weight components. We observe two things: the ratio of cluster sizes for the activation partition has no definite relation to generalization performance; on the other hand, the ratio for the reference 2-means clustering is smaller for models that generalize better.

- the ratio of 2-means reference cluster sizes decreases as models generalize better.

The influence of hyperparameters was not conclusive for this last metric, and has thus been omitted.

6.3 Influence of Number of Significant Components

In previous sections, we have always shown results when the top-1 % magnitude weight components are kept. This particular value is not important; we simply use it because it provides a good trade-off between computation time and clarity of results. For the interested reader, appendix B contains results presented in previous sections, with other pruning percentages. The influence of the number of significant components is also discussed, be it for a different purpose, in section 7.2.

We observe on figure 6.11 that the percentage threshold above which weights are kept does not affect results qualitatively, as most lines on the plot do not cross. However, higher threshold percentages make for clearer plots, because there is less noise from parasite connections. We also see on the figure that models with good

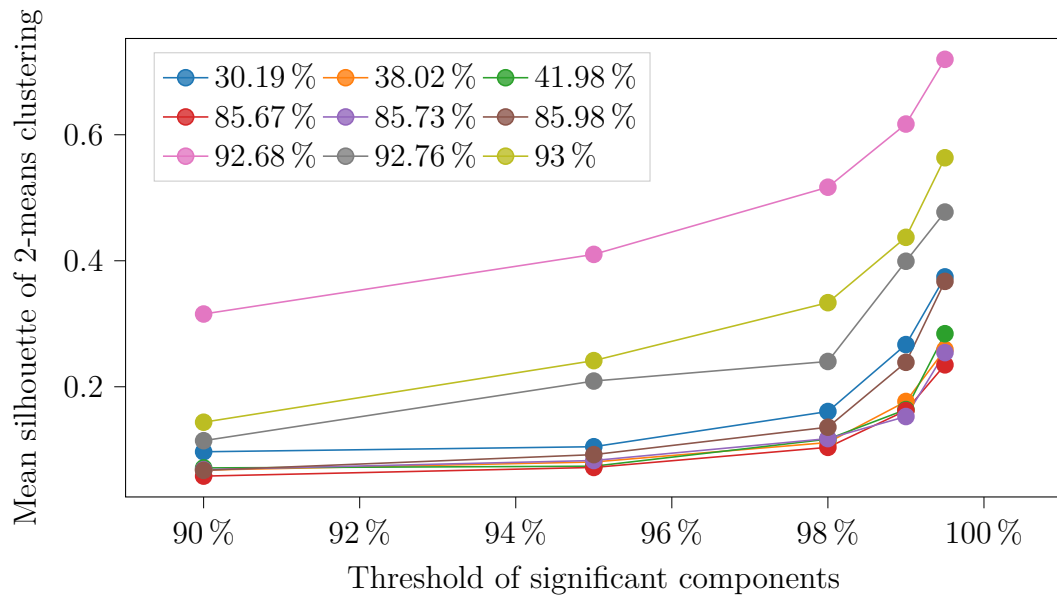


Figure 6.11: Influence of number of significant components on 2-means clusterability for a variety of models with different generalization performances. We observe that the choice of threshold does not matter qualitatively, but can reduce noise on the results. Additionally, we verify that in accordance with what we observed on figure 6.1, models with higher generalization performance also have better 2-means clusterability, for the most part.

generalization performance tend to have higher mean silhouette $\bar{s}$ than models with worse generalization, which is in line with our observation from figure 6.1.

6.4 Conclusion

In this chapter, we have introduced the notion of significant component, and have presented some metrics to evaluate the quality of a clustering, in a broad sense. We did this in order to study one of the theme questions of this thesis: do models with good generalization performance exhibit better implicit binary partitioning at the neuron-level, and/or are their internal representations more easily 2-means clusterable?

We looked at the relation between this binary partition quality and generalization performance, concluding that while there is definitely a relation between the two, it is not abundantly clear, and that the choice of significant components strongly influences the clarity of the resulting figures. In order to better grasp this relation, we studied the effect the various hyperparameters of our models have on it, and finally, we explored the effect of the threshold in the definition of significant components.

Thanks to this extensive investigation, we are now able to conclude with a reasonable amount of certainty that models with good generalization tend to have better implicit partitioning of their thresholded neuron-level representations and better 2-means clusterability than their worse-generalizing counterparts, with the effect being the result of a combination of hyperparameters.

The notions we introduced in this chapter and the one before it will also be central to the discussions in the next. Indeed, having examined how the binary partition of internal representations relates to generalization, one could now wonder whether a similar clustering-based approach could detect patterns in neuron inputs, as we defined them in section 5.2.1, and see how these patterns influence or correlate with generalization performance. The idea of patterns in deep neural networks has been discussed before in the literature, as we briefly showed in section 3.2, and it is precisely the one we study in the next chapter.

Chapter 7

Neurons as Pattern Discriminators

After having studied binary partitions in chapter 6, we now broaden our perspective, looking at neuron-level *patterns* instead. As we discussed in section 3.2, the idea of patterns in visual classification tasks is not new, and has been studied extensively in the literature. In this chapter, we start in section 7.1 by refreshing our definition of the term “pattern,” introduced in section 5.2.1, and lay out the tools we use to detect them. Then, in section 7.2 we present some simulations on synthetic data created for this thesis, in order to verify that our tools are appropriate for the task at hand; we also indicate which underlying assumptions we make. Finally, we present the results of our search for patterns on the real data and models in section 7.3. They are some of the most compelling pieces of evidence known to date of the existence of patterns at the neuron level.

7.1 Terminology and Notation

Throughout this chapter, we make use of much of the same terminology as was introduced in section 6.1, notably the notion of significant component, as well as the definition of the mean silhouette $\bar{s}$ and the mutual information in (6.8). Other than the fact we use a normalized, chance-adjusted version of the latter [110], these notions are unchanged. For clustering, we again use the k -means algorithm.

We also repeat the notion of a pattern at the neuron-level, which we introduced in section 5.2.1. Intuitively, a pattern in our case is a set of incoming significant weight components along which the neuron has excitations different from the typical excitations. More formally, let $\mathbf{k}$ represent the vectorized weight components linking the neuron to its input. A pattern is then a set of indices $\mathcal{P}$ such that $|k_i| \gg 0$ for $i \in \mathcal{P}$, and $k_i \approx 0$ otherwise. The intuition we want to empirically study in

this chapter is that neurons identify a finite set of patterns in their input, and that the better they are at recognizing such patterns, the better the generalization performance of the model. Note that these patterns do not need to be bijective with respect to the class labels of the dataset; indeed, multiple patterns can correspond to the same label [111]. To visualize an example of this many-to-one mapping, consider that objects can look different when viewed from different angles. As such, we expect that neurons in models that generalize well should be able to recognize the visual patterns associated to a given class label.

Our hypothesis, which we verify experimentally in section 7.3, is that such patterns correspond to clusters that can be detected by using k -means, for increasing values of k . Indeed, we hypothesize that a neuron will learn to recognize a set of patterns $\mathcal{P} = \{\mathcal{P}_1, \dots, \mathcal{P}_k\}$. This means that its weights k_i will be large (in absolute value) when $i \in \bigcup_j \mathcal{P}_j$, and negligible otherwise. In order to correctly identify the pattern $\mathcal{P}_l$ a sample presents, the neuron then needs to also know how to zero out the input coming in through components i for $i \in \bigcup_j \mathcal{P}_j \setminus \mathcal{P}_l$. If this is not done well, the classification process will be polluted by noise introduced through the parasitic connections.

7.2 Investigation on Synthetic Data

As mentioned previously, we now present a study showing that patterns of the form we described in section 7.1 can be detected using the clustering tools and evaluation metrics that we introduced.

7.2.1 Synthetic Data Generation Process and Clustering

We simplify the problem at hand by considering that there are five variable parameters to the problem:

- n_{dim} refers to the number of dimensions of the data; in the terminology we introduced previously, this is the number of significant components.
- l_{pat} denotes the length of each pattern. We make the simplifying assumption that all patterns in a given simulation have the same length.
- n_{pat} is the number of patterns in a given simulation.
- n_{data} refers to the number of samples in the dataset.
- δ is the amount by which components included in a pattern are separated from the initialization value. We again make a simplifying assumption, namely that this value is constant both within a pattern as well as for all the other

patterns in a given simulation. By default, we consider it to be a positive value.

We present the values these parameters take in table 7.1.

Parameter	Values	Default
n_{dim}	30, 50, 75, 100, 200, 300, 500, 1000	100
l_{pat}	1, 2, 3, 5, 10, 15, 20, 25, 30, 50, 75, 90	10
n_{pat}	1, 2, 3, 5, 10, 25, 30, 50, 100, 200	10
n_{data}	100, 200, 500, 1000, 3000, 5000, 10000	1000
δ	0.1, 0.2, 0.5, 0.75, 1, 1.25, 1.5, 2, 3, 5, 10	2

Table 7.1: Parameters for synthetic data. The meanings of these parameters are explained in the main body of the text.

The process by which the synthetic data is then generated is presented in algorithm 7.1.

Algorithm 7.1 Generation of synthetic pattern data.

- 1: **procedure** GEN($n_{\text{dim}}, l_{\text{pat}}, n_{\text{pat}}, n_{\text{data}}, \delta$) $\triangleright$ number of dimensions, pattern length, number of patterns, number of samples, and pattern separation
 - 2: initialize n_{data} normally distributed samples $\mathbf{x}^{(i)}$, $i = 1, \dots, n_{\text{data}}$, such that $x_j^{(i)} \sim \mathcal{N}(0, 1)$ for $j = 1, \dots, n_{\text{dim}}$
 - 3: assign a pattern label $p^{(i)} \in \mathbb{Z} \cap [1, n_{\text{pat}}]$ to each sample $\mathbf{x}^{(i)}$, creating a reference clustering $\mathcal{P}$
 - 4: assign a set of indices $\mathcal{P}^{(i)} \subseteq \mathbb{Z} \cap [1, n_{\text{dim}}]$, $|\mathcal{P}^{(i)}| = l_{\text{pat}}$, to each pattern label $p^{(i)}$, such that if $p^{(i)} = p^{(j)}$, then $\mathcal{P}^{(i)} = \mathcal{P}^{(j)}$
 - 5: **for** $i \leftarrow 1, \dots, n_{\text{data}}$ **do**
 - 6: **for** $j \in \mathcal{P}^{(i)}$ **do**
 - 7: $x_j^{(i)} \leftarrow x_j^{(i)} + \delta$ $\triangleright$ add separation to pattern components
 - 8: **end for**
 - 9: **end for**
 - 10: L^2 -normalize each sample $\mathbf{x}^{(i)}$
 - 11: **end procedure**
-

Once the data has been generated for a given parameter choice, it is clustered using k -means, for $k = 2, \dots, 2n_{\text{pat}} + 10$. For each resulting clustering $\mathcal{C}_k$, we then compute its mean silhouette, as well as the normalized chance-adjusted mutual information between itself and the (known) reference clustering $\mathcal{P}$, $I(\mathcal{C}_k, \mathcal{P})$. By repeating this process for the various possible parameter choices, we then get an

idea of how well our tools detect the presence of patterns of the form we defined in section 7.1.

7.2.2 Synthetic Data Parameter Influence

We now show the effect of varying each hyperparameter, keeping the others at their default values as described in table 7.1. In every figure, each color corresponds to a different value of the parameter being studied.

7.2.2.1 Influence of n_{dim}

Figure 7.1 shows the effect of the number of dimensions on the pattern detectability through k -means clustering. This parameter is thus the synthetic analog of the number of significant components. We observe that lower-dimensional data is more easily clustered than higher-dimensional data, an observation that is in line with the one we made in figure 6.11, when we mentioned that larger pruning percentages correspond to better clusterable data. This also intuitively makes sense when one considers that the Euclidean distance between two samples is computed as a sum of squared differences along each axis. When the number of axes increases, but the length of patterns stays the same, the influence of the pattern components on the distance between samples is decreased.

7.2.2.2 Influence of l_{pat}

Figure 7.2 shows the effect of the length of the patterns on pattern detectability through k -means clustering. We see that very short patterns are poorly detected by k -means clustering. On the other hand, $l_{\text{pat}} = n_{\text{dim}}/2$ appears to be the optimal pattern length for cluster detection, possibly because for this value, pattern overlap should be minimal, and the average distance between samples of different patterns is maximal. We also see that $l_{\text{pat}} = x$ behaves the same as $l_{\text{pat}} = n_{\text{dim}} - x$, which is logical considering that both cases are functionally identical.

7.2.2.3 Influence of n_{pat}

Figure 7.3 shows the effect of the number of patterns on pattern detectability through k -means clustering. We see that unclustered data ($n_{\text{pat}} = 1$) results in very poor k -means clusterings, as expected. Increasing the number of patterns for a given length l_{pat} inevitably leads to more overlap, and thus explains why the mean silhouette for $k = n_{\text{pat}}$ decreases as n_{pat} increases. However, we find that for extremely large values of n_{pat} , k -means no longer allows us to deduce the correct number of patterns.

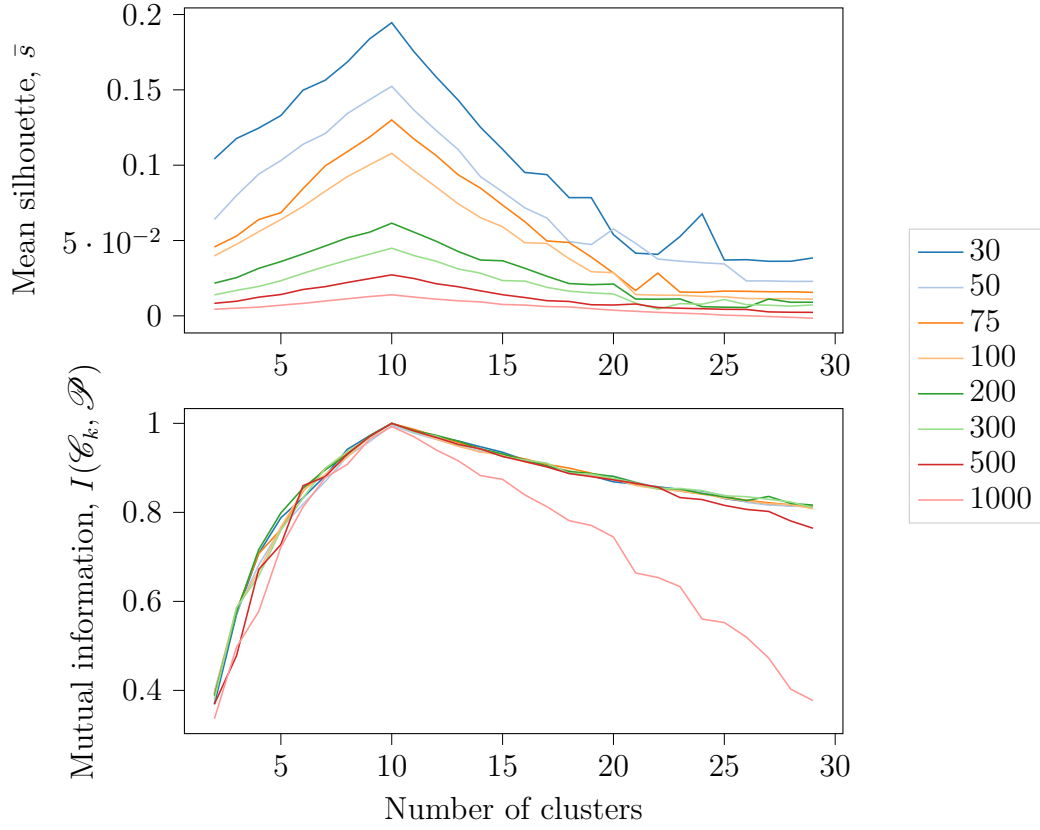


Figure 7.1: Influence of number of dimensions on pattern detectability through k -means clustering. Each color corresponds to a different value of n_{dim} . We observe that k -means clustering performs better the lower the number of dimensions, n_{dim} . This ties into what we observed previously in figure 6.11 concerning the number of significant weight components; when the threshold percentage is increased, data becomes more easily clusterable (as measured by mean silhouette). This is because as the number of dimensions increases, the influence of the components included in the sample’s pattern on the distance between samples decreases (it is “diluted” in a way).

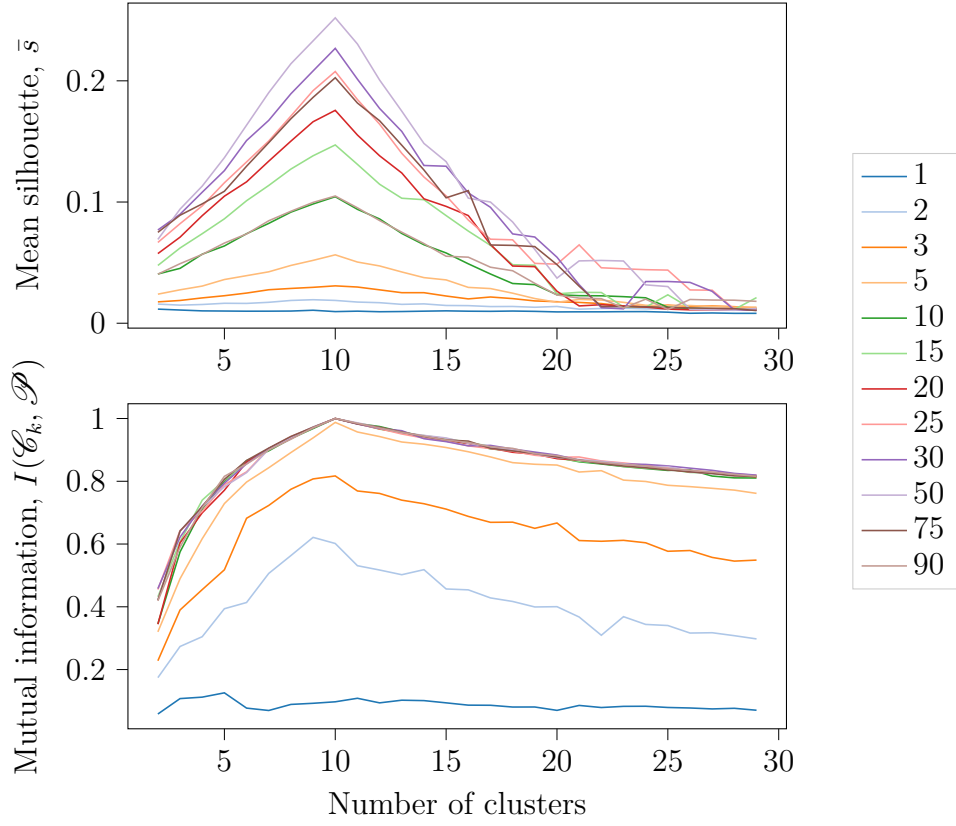


Figure 7.2: Influence of pattern length on pattern detectability through k -means clustering. Each color corresponds to a different value of l_{pat} . We observe that for very short patterns, k -means can fail to distinguish patterns correctly. We also observe that the optimal pattern length for detection is $n_{\text{dim}}/2$ (presumably due to it being the value for which patterns overlap the least), and that $l_{\text{pat}} = x$ behaves the same as $l_{\text{pat}} = n_{\text{dim}} - x$.

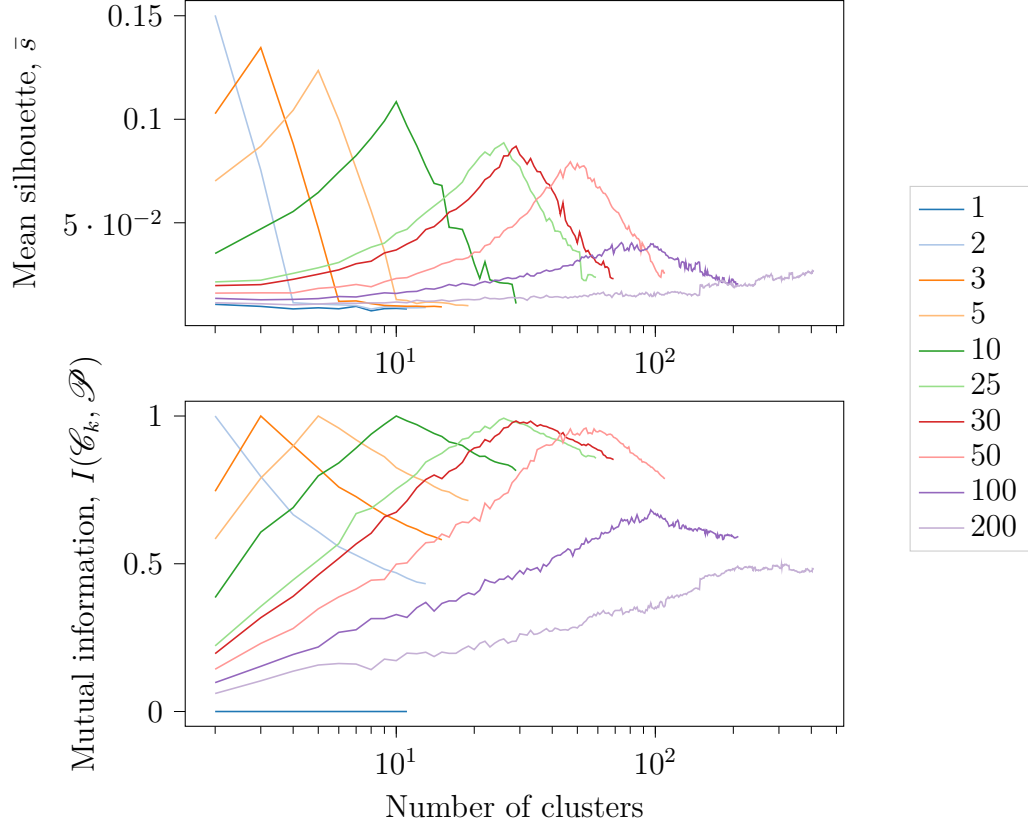


Figure 7.3: Influence of number of patterns on pattern detectability through k -means clustering. Each color corresponds to a different value of n_{pat} . We observe that for data without patterns $n_{\text{pat}} = 1$, no clusters are found, as expected. We also notice that as the number of patterns increases, k -means becomes worse at identifying clusters; this is probably again due to an increase in overlap. We also note that when the number of patterns becomes extremely large, k -means no longer reaches the highest mean silhouette at $k = n_{\text{pat}}$.

7.2.2.4 Influence of n_{data}

Figure 7.4 shows the effect of the number of samples on pattern detectability through k -means clustering. We see that increasing the number of samples positively

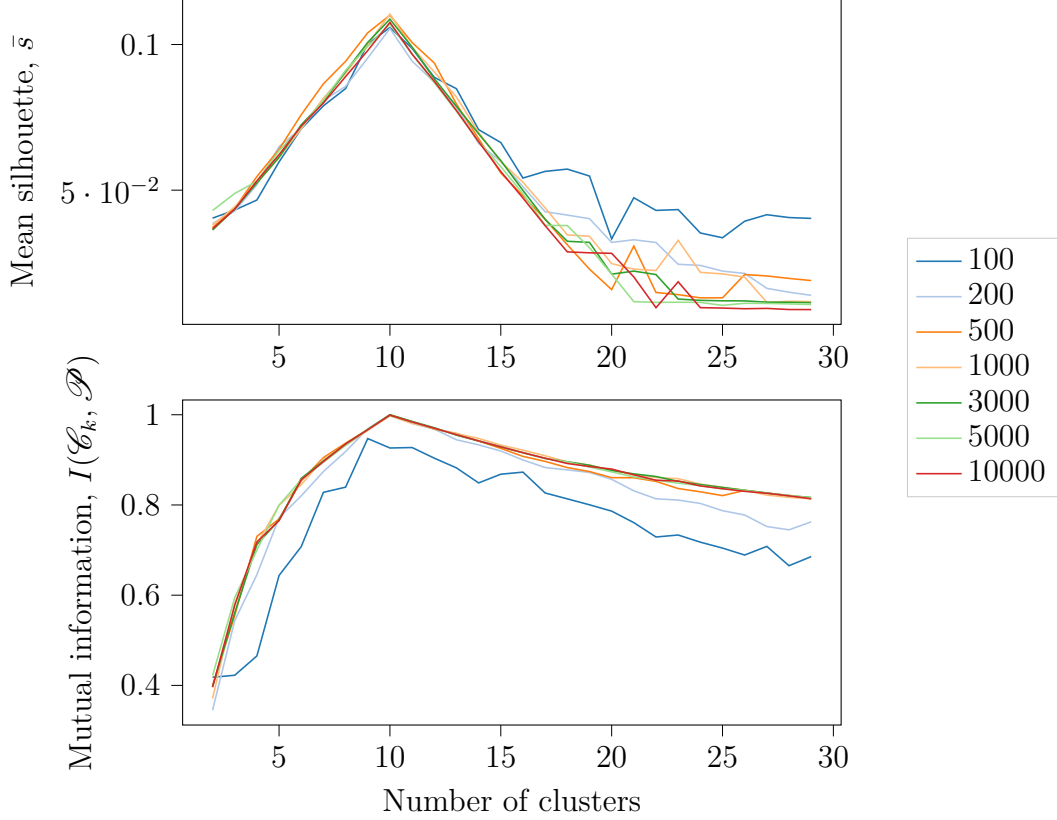


Figure 7.4: Influence of number of samples on pattern detectability through k -means clustering. Each color corresponds to a different value of n_{data} . As expected, having more data allows k -means to be more robust to noise, and to cluster the data better. This is especially visible when $k > n_{\text{pat}}$, as simulations with larger n_{data} show a marked decrease in parasitic clusterability compared to simulations with smaller datasets.

influences the ability of k -means to identify the correct clusters, as well as reducing parasitic clusterability for $k > n_{\text{pat}}$. This observation is not surprising.

7.2.2.5 Influence of δ

Figure 7.5 shows the effect of the pattern separation on pattern detectability through k -means clustering. Unsurprisingly, we observe that the larger the separation δ , the easier it is for k -means to cluster the data correctly. When one considers that

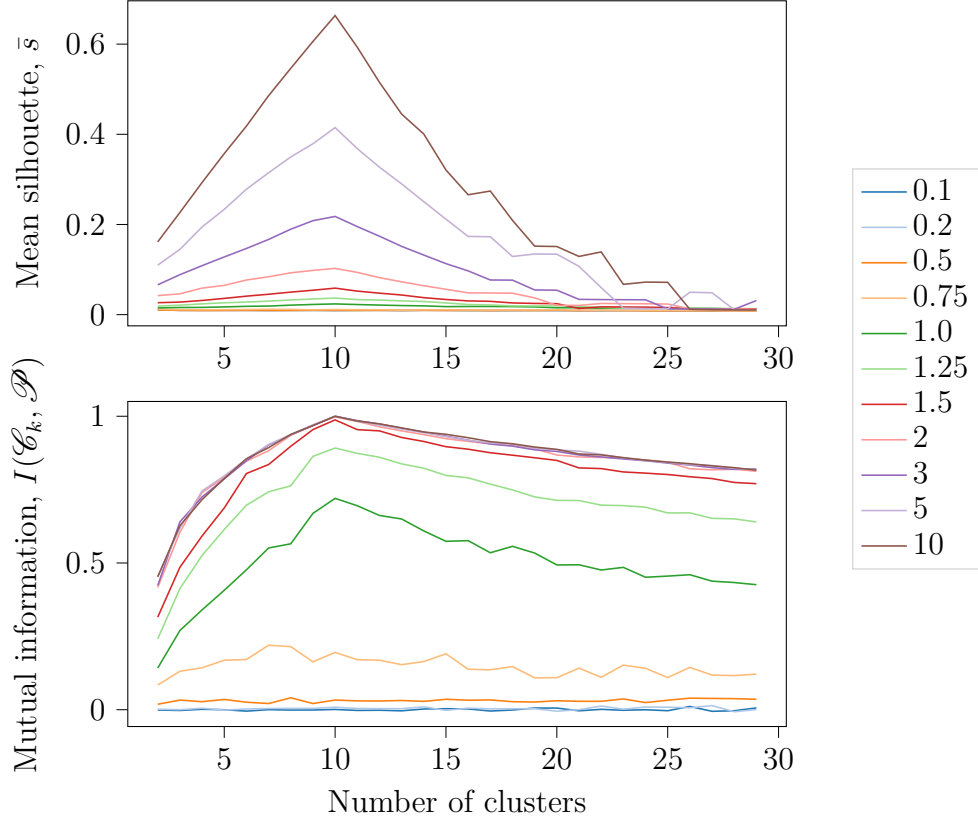


Figure 7.5: Influence of pattern separation on pattern detectability through k -means clustering. Each color corresponds to a different value of δ . Visibly, higher separation improves clusterability; this can be explained by again examining the effect of increasing δ on the average distance between samples presenting different patterns. It appears that the separation should at least be around one standard deviation of the initialization (1 in our case) for k -means to be able to cluster data correctly.

the Euclidean distance between samples is the sum of squared differences along each axis, this makes intuitive sense. We also notice that in order for k -means to clearly identify the true number of patterns in the data, the separation should be approximately greater than or equal to the standard deviation of the initialization noise, which was 1 in our case.

7.3 Empirical Evidence for the Presence of Neuron-Level Patterns

Having demonstrated experimentally that clustering tools can detect patterns of the form we described, we now wish to study whether such patterns are present in the internal representations of data at the neuron level, for a randomly chosen neuron.

Figure 7.6 shows the results of this experiment for three classes of generalization performance: bad, average, and good, each containing three models, for three different significant component thresholds. We observe that

- Models with poor generalization performance barely (if at all) identify any patterns in the internal representation, independent of threshold percentage.
- Models with good generalization are very consistently able to identify patterns, even across different threshold percentages. The number of identified patterns is usually slightly above 10, which is the number of supervision labels in the dataset, a result which we predicted in section 7.1 when talking about e.g. the effect of different viewing angles on classification.
- Models with average generalization are sometimes able to discriminate between patterns, with models having high pattern detection ability for some threshold percentages, and low pattern detection ability for others.

Interestingly, the results for models with good generalization are quite similar to what we observed when detecting patterns with δ slightly smaller than the standard deviation of the initialization.

We also note that choosing the right significant components is crucial, as evidenced by the fact the results obtained with a lower number of significant components are clearer than their more noisy counterparts.

Recalling our definitions and intuitions from sections 5.2.1 and 7.1, we can try to provide an explanation for the behavior of the various models in figure 7.6. We defined a pattern as a set of significant components, and explained how a single neuron could learn to discriminate between multiple patterns. There are thus two “steps” to learning patterns: first, the neuron needs to learn which of its input

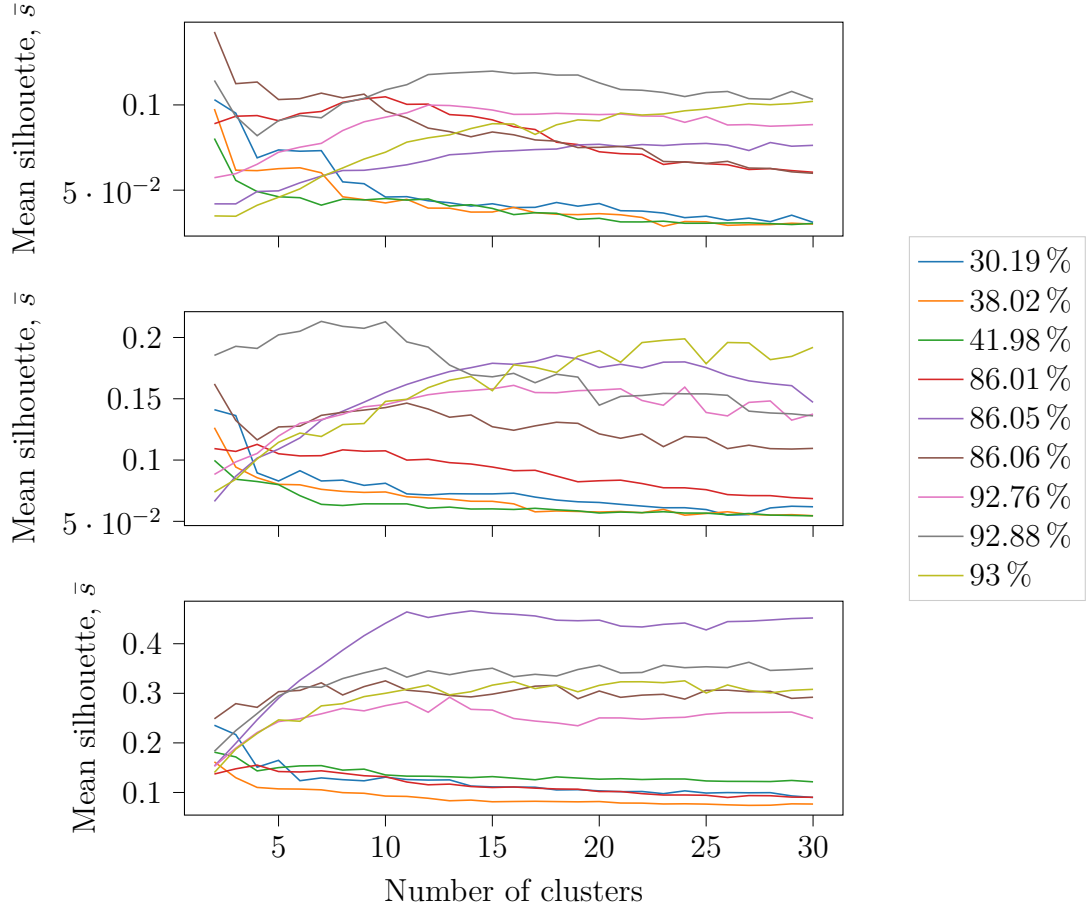


Figure 7.6: Relation between k -means pattern clusterability (by a randomly selected neuron) and generalization for a variety of models. The color of a curve defines its generalization capabilities, as indicated by the test accuracy percentage table provided on the right of the figure. Each figure corresponds to a different threshold percentage to define the significant components associated to a neuron (top: top-5 %, center: top-2 %, bottom: top-1 %). We see that models with poor generalization perform very poorly, and do not detect patterns in the data across all threshold percentages. On the other hand, models with good generalization are able to identify patterns; interestingly, the mean silhouette of the clustering on their internal representations seems to consistently hit its maximum value slightly beyond $k = 10$, which is the number of supervision labels in the dataset. They are also consistent across different threshold percentages. Average models are all over the place, clearly identifying patterns for some threshold percentages but not for others. Finally, we note the importance of correctly selecting significant components, as evidenced by the fact the results are clearer on the plots with smaller numbers of significant components.

components are important (the answer being those with indices in the union of all patterns the neuron detects); second, it needs to learn to cluster inputs according to their values along these components, a process we studied in section 7.2.

The behavior of the good and bad models is quite expected: the former perform well at both tasks, while the latter do a poor job of identifying important connections, and thus cannot learn to cluster properly either. The case of average models is more interesting. As we showed in chapter 5, some of them are able to learn peaked weight distributions. Similarly, figure 7.6 shows that some average models are able to learn the clustering step well. Our hypothesis is that average models are those that are able to do one of the two steps well, but not the other. This would explain why the clustering ability of the neurons in average networks is so dependent on the threshold percentage in some cases; through imposing a cutoff point, we are “helping” the model with the first step, leaving them to do mainly the second step, which they are good at. In other average models, the significant components are chosen correctly, but the clustering step is not done well, and this is shown through their poor performance in figure 7.6.

7.4 Conclusion

In this chapter, we have looked at the ability of networks to detect patterns in the internal representation of samples. We showed on a synthetic dataset what the effect of various parameters is on their detectability through clustering tools like k -means and the mean silhouette.

We then applied these techniques to a case with real-world data, and observed that this pattern detection is related to generalization, observing that models with good generalization performance consistently find patterns, and that the number of these patterns is slightly higher than the number of supervision labels of the dataset. We explain this phenomenon by noticing that in image classification, it is reasonable for a dataset to contain widely different images of the same object, an example being the front and side views of it. On the other hand, we see that models with bad generalization performance perform poorly at pattern detection, leading us to think there is a causal link between pattern detection and generalization performance.

The correlation between model generalization and the clustering of data in the subspace of significant components associated to a neuron makes the characterization of those clusters especially attractive. Some of the questions of interest are: (i) Are the clusters captured by distinct neurons in a layer related or not? (ii) Is it important that each neuron captures only a small number of clusters?

In addition to demonstrating the emergence of patterns/clusters in the subspace of significant components associated to a neuron, our approach also provides a

means to define those patterns explicitly, as the centroid of the k -means clusters. Hence, it becomes possible to identify which patterns are matched by a test sample, and to investigate whether a good matching between neuron patterns and the associated internal representation of a sample is an indicator of a reliable decision. This idea is explained in more detail in section 8.2.

This investigation has shown promising results, but as mentioned, the work presented here would benefit from additional research. Chapter 8 contains some of our ideas we would like to explore in future work on this topic.

Chapter 8

Future Work

Many of the ideas developed in this thesis are still in an early stage, and present interesting avenues for future work. This chapter contains some ideas for future work based on the investigations in this thesis.

8.1 Leveraging Neuron Behavior to Predict Generalization Performance

One of the key goals of this thesis is to work towards a way of predicting generalization performance in models through the study of the properties of neuron-level representations. In this sense, the observations of chapters 6 and 7 are very promising when combined with the notion of significant components of chapter 5.

Indeed, we showed evidence that models with good generalization performance appear to consistently (i) arrive at a peaked distribution of weights isolating a set of significant components, and (ii) learn to cluster samples through manipulation of the incoming excitations. These observations could be used to predict whether a given model will generalize well (an important problem in the literature, as evidenced by [4]), or even to guide research towards training methods that arrive at models that perform well at both these tasks.

8.2 Pattern-Based Confidence Measures for Classification and Out-of-Distribution Detection

In chapter 7, we studied the hypothesis that neurons in a network learn to recognize patterns in their input. More precisely, they learn to isolate a subset of their components that matters for classification, and learn to cluster the inputs received through these components.

Through studying the internal representations of neurons, we have come to believe that samples will be clustered according to the presence of certain patterns in the neuron inputs; one possible way of leveraging this observation would be to establish and verify confidence measures showing how close a new sample is to existing clusters of samples. Indeed, our approach also provides a means to define those patterns explicitly, as the centroid of the k -means clusters that are found. It then becomes possible to identify which patterns are matched by a test sample, and it would be interesting to investigate whether a good matching between neuron patterns and the associated internal representation of a sample is an indicator of a reliable decision. This would be timely work, with recent research having been oriented towards training networks that help users understand their predictions through the use of confidence measures [112]. A similar approach could be considered to detect adversarial attacks [113, 114], for the same reasons.

In a similar vein, the problem of out-of-distribution (OOD) detection is a growing one in computer vision and deep learning [115, 116, 117, 118, 119]. It is concerned with what happens to models when they are tasked with classifying images whose label is not part of the training set on which the model was trained. In real world applications, we want our models to be able to detect when they are in such a situation, which is precisely the problem of OOD detection.

Combining these ideas, it would be very interesting to study whether OOD samples can be detected using pattern-based measures, such as the distance between the internal representation of a sample and the average distance of other samples assigned the same label. In essence, what this would be doing is studying whether the sample presents one of the patterns that were learned during training.

8.3 Class-Specific Study of Clustering and Pattern Detection

In chapters 6 and 7, we showed that models with good generalization tend to learn cluster their inputs better. However, the precise effect of class labels was not studied. One avenue for future work would be to further investigate what can be learned from isolating samples on a per-class basis.

In particular, we would like to know whether our hypothesis that the pattern-to-class label mapping is many-to-one holds in general [111]; one way to do this would be to observe how many patterns are observed when only samples of a given class are considered.

8.4 Learning Dynamics of Good Models

Throughout this thesis, we have only looked at results obtained on models that had been trained to approximate convergence. None of our experiments explicitly look at the differences in learning dynamics between bad models and good models. This is especially important in view of known results in the literature such as those in [120] concerning the importance of the early phase of network training in generalization performance.

It would be particularly interesting to study whether the emergent behaviors described in chapters 6 and 7 emerge at the beginning of training, or whether they gradually develop throughout the entire training process.

8.5 Conclusion

In this chapter, we have given some perspectives for future work related to the main topics of this thesis. Several interesting directions present themselves and will be explored in future research.

In chapter 9, we will recapitulate the main observations made during this thesis, and give an overview of the current state of our intuitions.

Chapter 9

Conclusion

In this thesis, our overarching goal as defined in section 1.1 was to study how internal representations of deep neural networks are related to the generalization performance of these networks.

We started this study in chapter 2 by giving an overview of the fundamentals of neural networks, as well as introducing some concepts which are particularly important for the reader’s comprehension of the work done in this thesis.

We then gave an overview of related work in the literature in chapter 3, and showed how the problem we study is relevant to other work in the deep learning space, as well as giving some insight into our motivation for performing the precise experiments we did.

In chapter 4, we introduced the reader to our methodology, and explained the precise experimental setup that was used throughout this thesis in hopes of allowing reproducibility.

Chapter 5 detailed our analysis of the statistical distribution of layer weights. In particular, we found that its peakedness (a measure we defined more formally in section 5.1.1) is a good indicator of generalization performance, and showed which hyperparameters of the network contribute to this. We also made the fundamental observation that only a small subset of weight components is responsible for the majority of the “information” entering a neuron. This observation allowed us to define the concept of significant components, a notion that we used in later chapters to facilitate the use of clustering metrics based on distance.

In chapter 6, we showed that neurons implicitly partition their inputs into two categories through the use of the ReLU activation function, leading to a fundamentally different treatment of samples with negative and positive pre-activations. Combining this with our notion of significant components, we used clustering tools and measures such as k -means and the silhouette in order to study how the quality of this implicit partition is related to generalization performance. Again, we enriched our results by integrating an analysis of how network hyperparameters

interact with the various measures that were used.

The results of chapter 7 built off of these clustering-based observations and looked at how the input to a neuron is clusterable into more than two categories. We started by experimentally validating the tools we used in the rest of the chapter to evaluate clustering quality, noting the effect of various dataset properties on the clarity of the results, then proceeded with our pattern detection experiments. These experiments helped us validate our hypotheses concerning the presence of patterns at the neuron level, and we found compelling evidence that the presence of these patterns combined with the learning of a peaked weight distribution is a strong indicator of generalization.

In chapter 8, we gave some perspectives on this work, and showed how future research (both theoretical or more practical in nature) could build on the results obtained in this thesis.

Our work has shown strong evidence that studying internal representations can help to better understand how deep neural networks generalize. We hope it will stimulate further investigation on this topic, and ultimately help lead the theoretical deep learning community towards a higher level of understanding about how the surprisingly good performance of deep neural networks comes to be.

Appendix A

Omitted Figures for Chapter 5

This appendix gives the figures that had to be omitted from chapter 5 for reasons of brevity.

A.1 Weights

A.1.1 Influence of Hyperparameters

A.1.1.1 Kurtosis

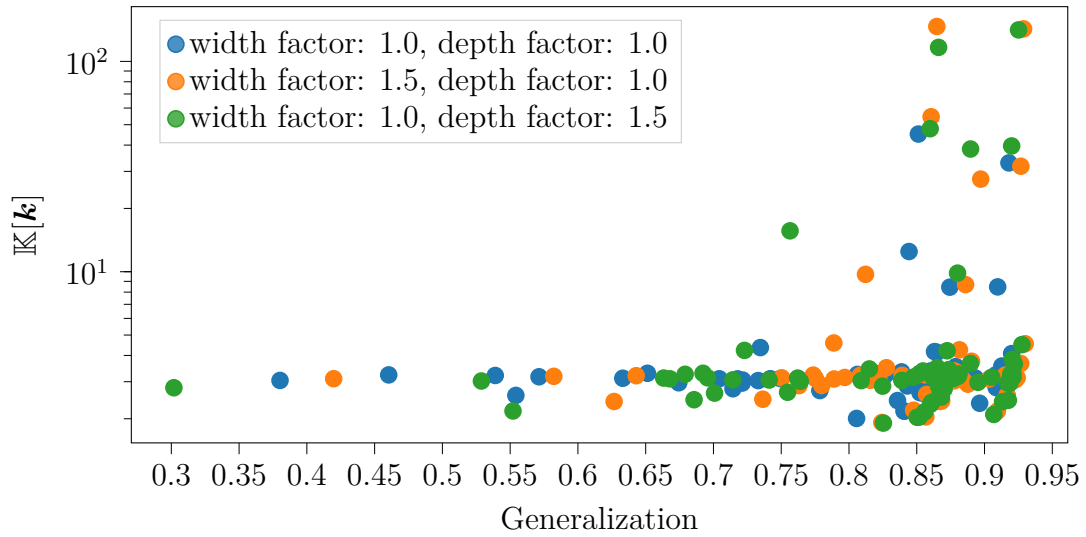


Figure A.1: Influence of architecture hyperparameter on kurtosis of weight distribution. Each color corresponds to a different value of the architecture hyperparameter tuple. The vertical axis is logarithmically scaled.

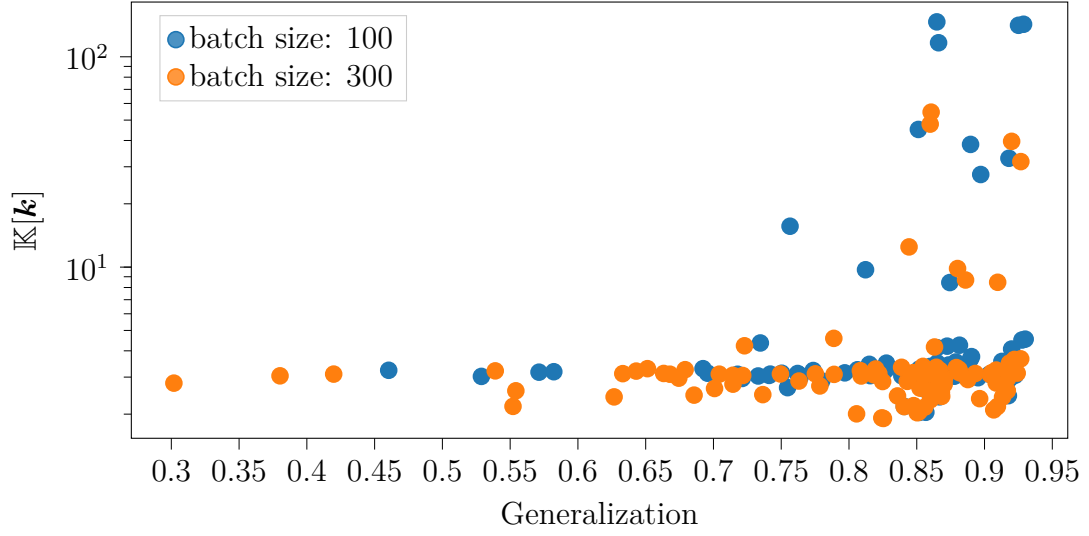


Figure A.2: Influence of batch size hyperparameter on kurtosis of weight distribution. Each color corresponds to a different value of the batch size hyperparameter. The vertical axis is logarithmically scaled.

A.1.1.2 L^∞ -Normalized Variance

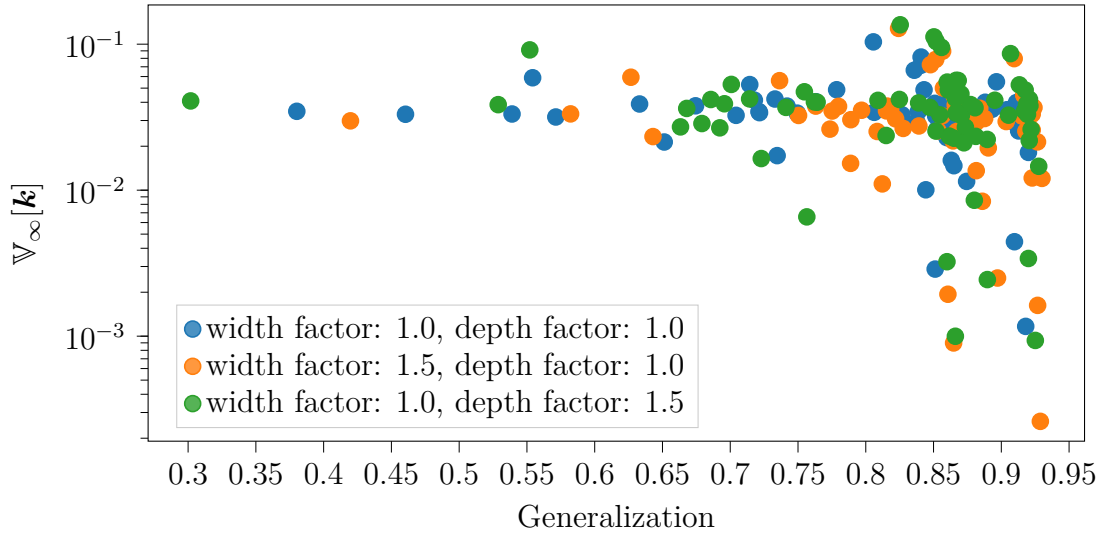


Figure A.3: Influence of architecture hyperparameter on L^∞ -normalized variance of weight distribution. Each color corresponds to a different value of the architecture hyperparameter tuple. The vertical axis is logarithmically scaled.

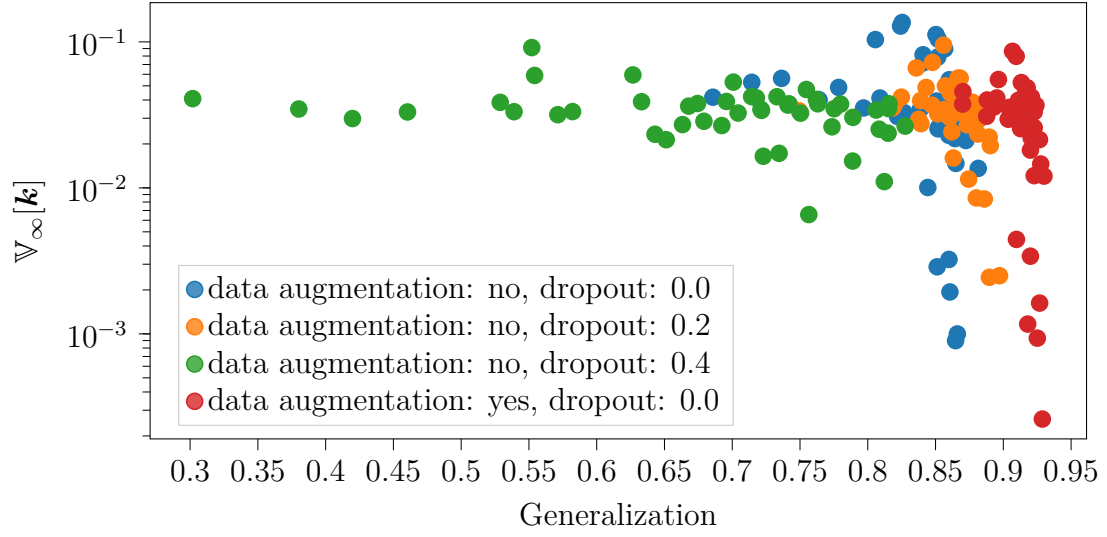


Figure A.4: Influence of noise regularization hyperparameter on L^∞ -normalized variance of weight distribution. Each color corresponds to a different value of the noise regularization hyperparameter tuple. The vertical axis is logarithmically scaled.

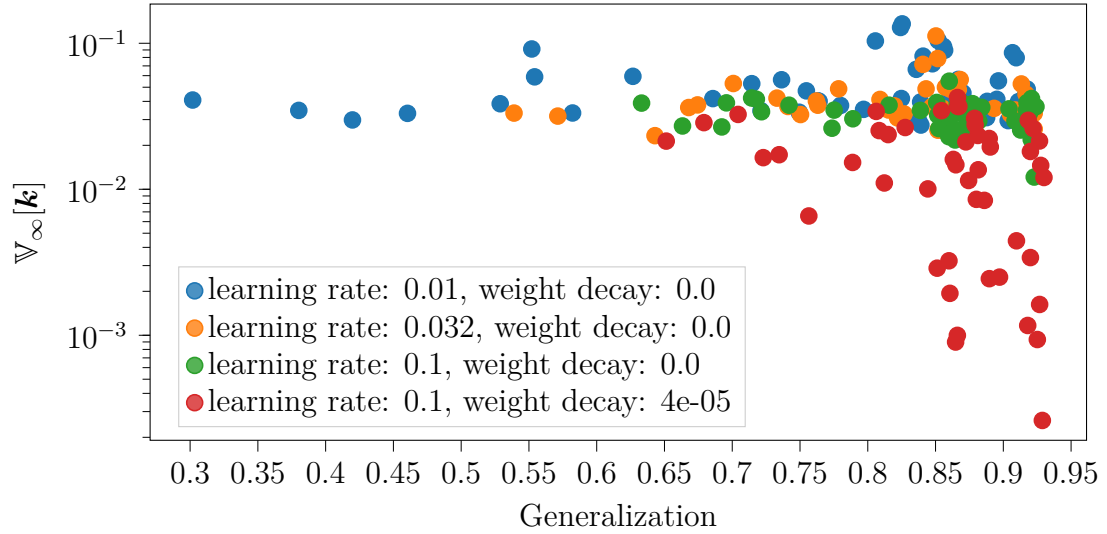


Figure A.5: Influence of effective learning rate hyperparameter on L^∞ -normalized variance of weight distribution. Each color corresponds to a different value of the effective learning rate hyperparameter tuple. The vertical axis is logarithmically scaled.

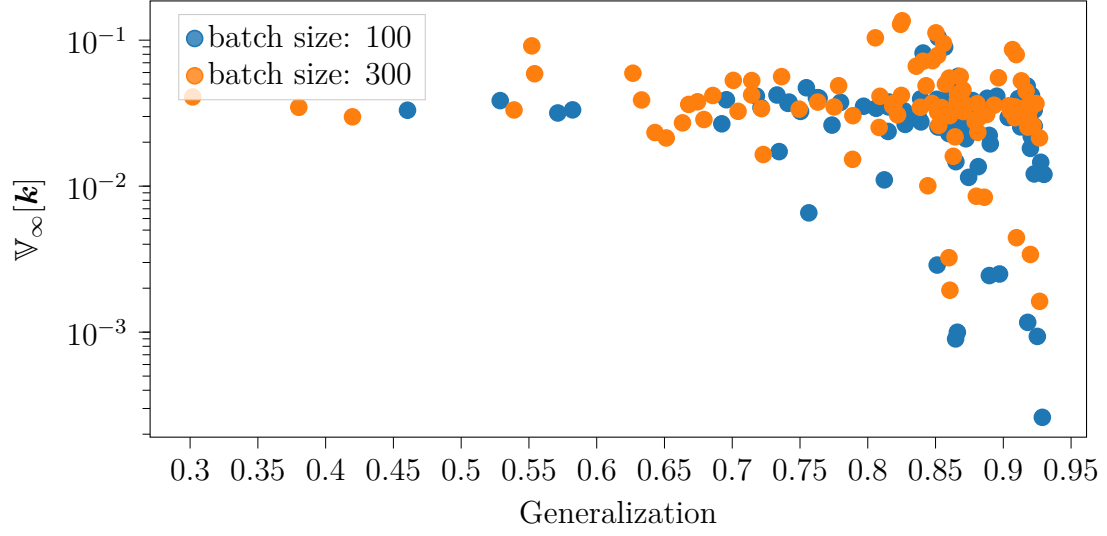


Figure A.6: Influence of batch size hyperparameter on L^∞ -normalized variance of weight distribution. Each color corresponds to a different value of the batch size hyperparameter. The vertical axis is logarithmically scaled.

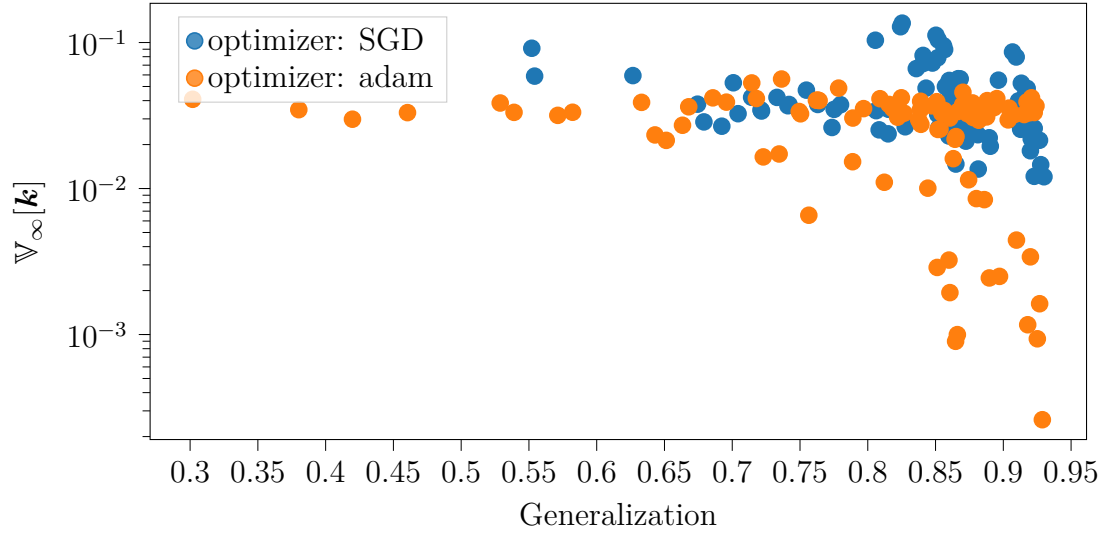


Figure A.7: Influence of optimizer hyperparameter on L^∞ -normalized variance of weight distribution. Each color corresponds to a different value of the optimizer. The vertical axis is logarithmically scaled.

A.2 Excitation-Weight Products

A.2.1 Influence of Hyperparameters

A.2.1.1 Kurtosis

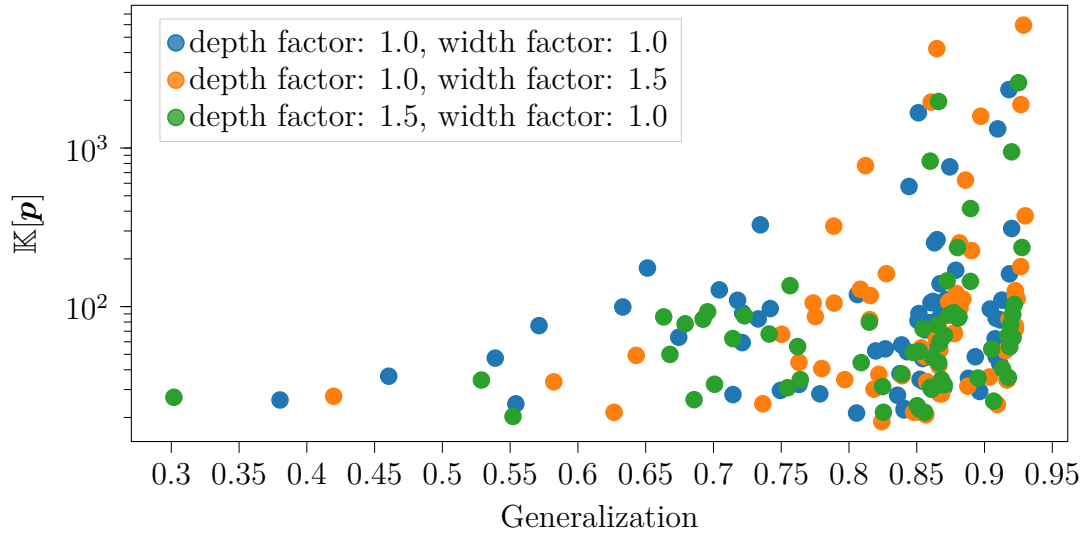


Figure A.8: Influence of architecture hyperparameter on kurtosis of EW product distribution. Each color corresponds to a different value of the architecture hyperparameter tuple. The vertical axis is logarithmically scaled.

A.2.1.2 L^∞ -Normalized Variance

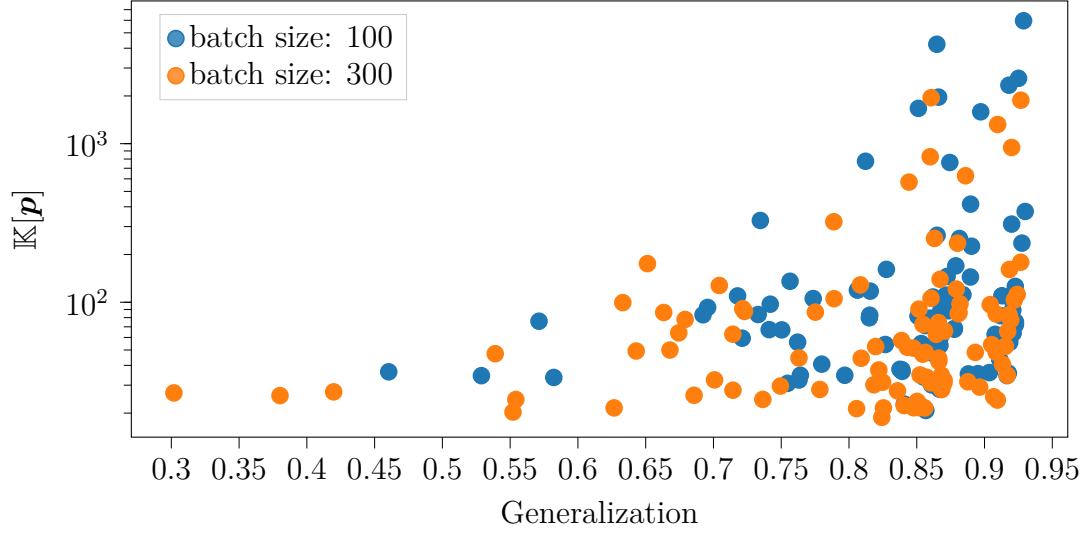


Figure A.9: Influence of batch size hyperparameter on kurtosis of EW product distribution. Each color corresponds to a different value of the batch size hyperparameter. The vertical axis is logarithmically scaled.

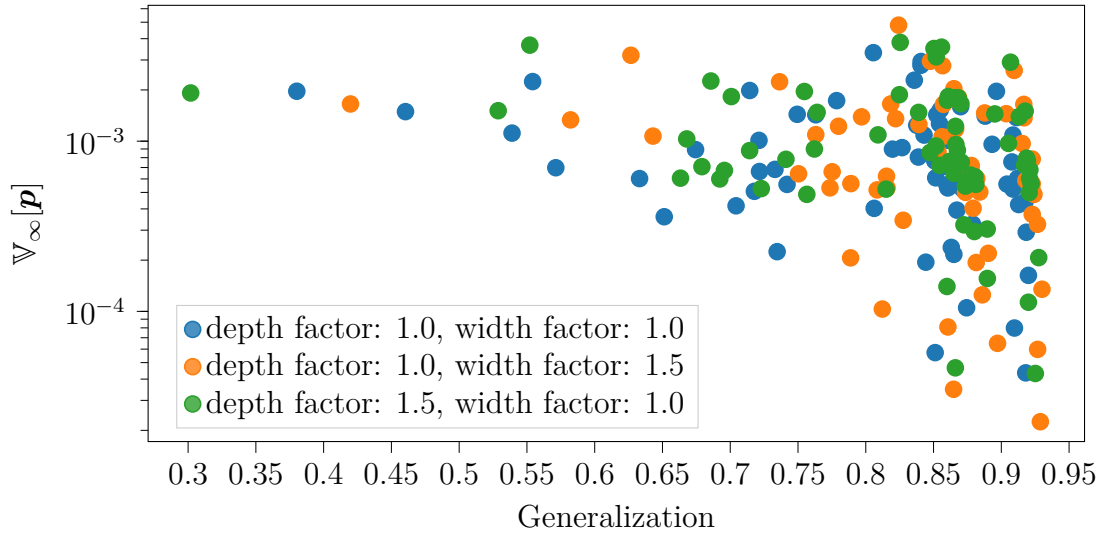


Figure A.10: Influence of architecture hyperparameter on L^∞ -normalized variance of EW product distribution. Each color corresponds to a different value of the architecture hyperparameter tuple. The vertical axis is logarithmically scaled.

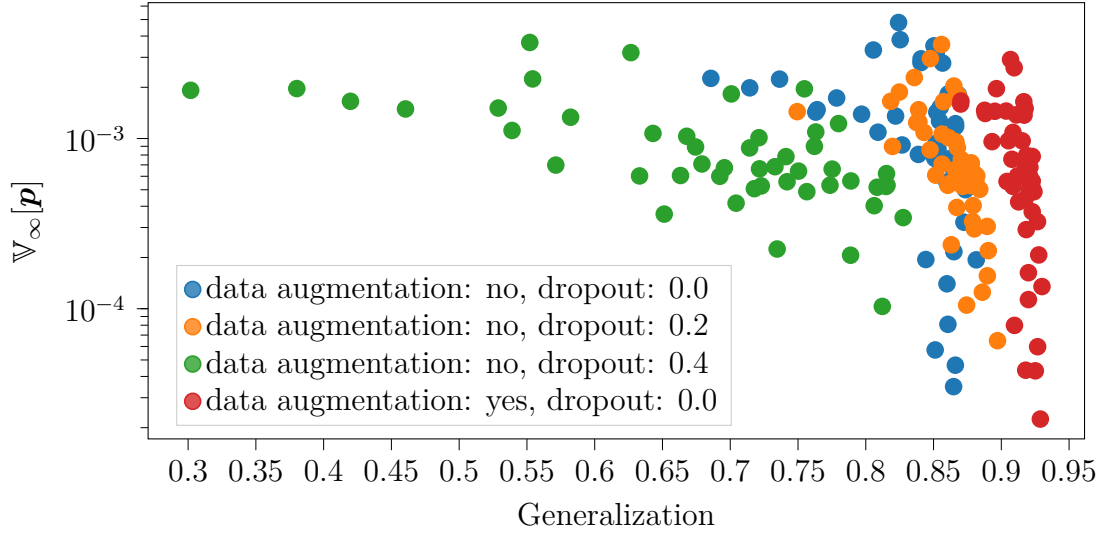


Figure A.11: Influence of noise regularization hyperparameter on L^∞ -normalized variance of EW product distribution. Each color corresponds to a different value of the noise regularization hyperparameter tuple. The vertical axis is logarithmically scaled.

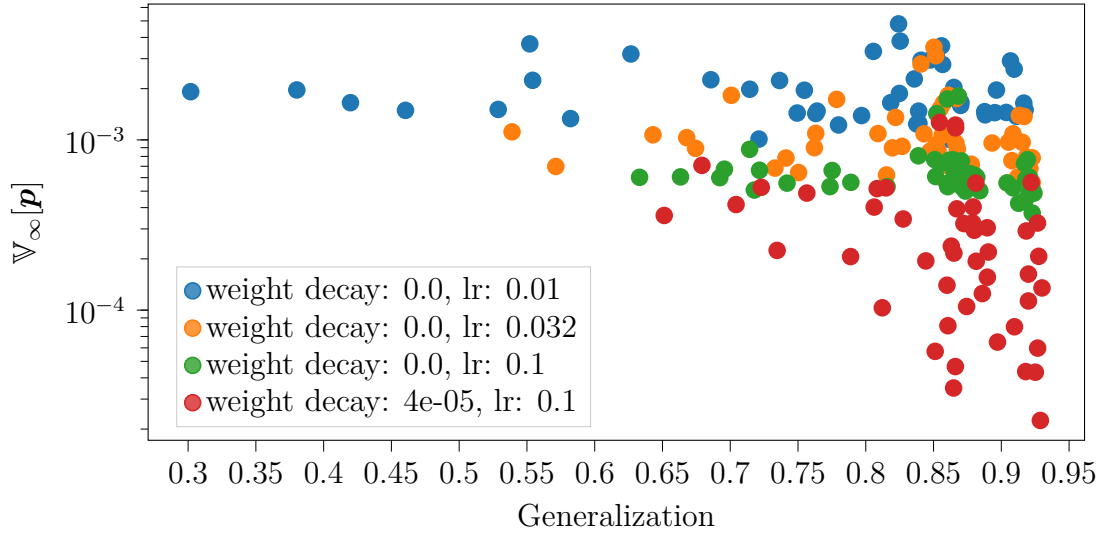


Figure A.12: Influence of effective learning rate hyperparameter on L^∞ -normalized variance of EW product distribution. Each color corresponds to a different value of the effective learning rate hyperparameter tuple. The vertical axis is logarithmically scaled.

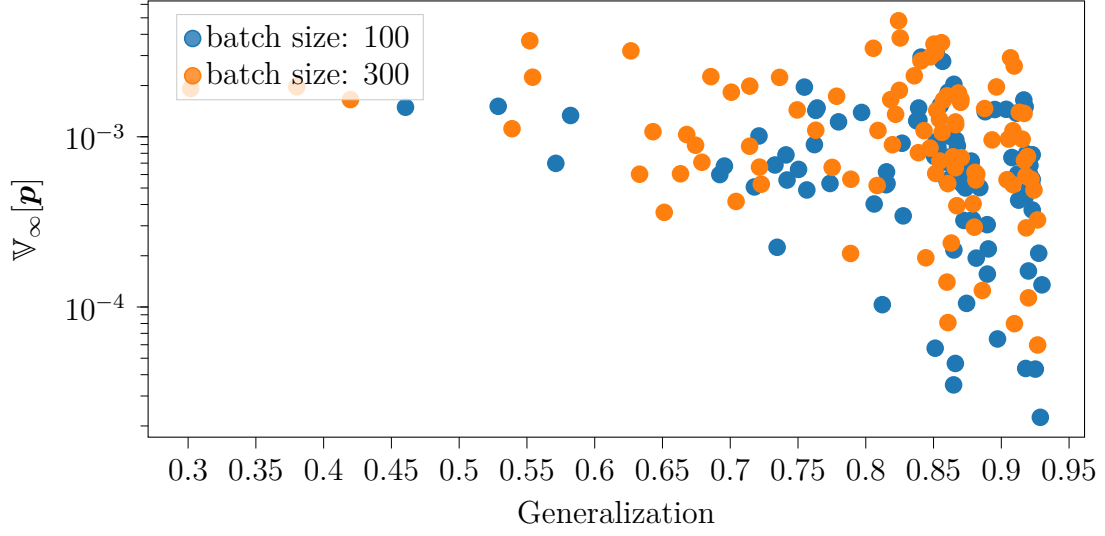


Figure A.13: Influence of batch size hyperparameter on L^∞ -normalized variance of EW product distribution. Each color corresponds to a different value of the batch size hyperparameter. The vertical axis is logarithmically scaled.

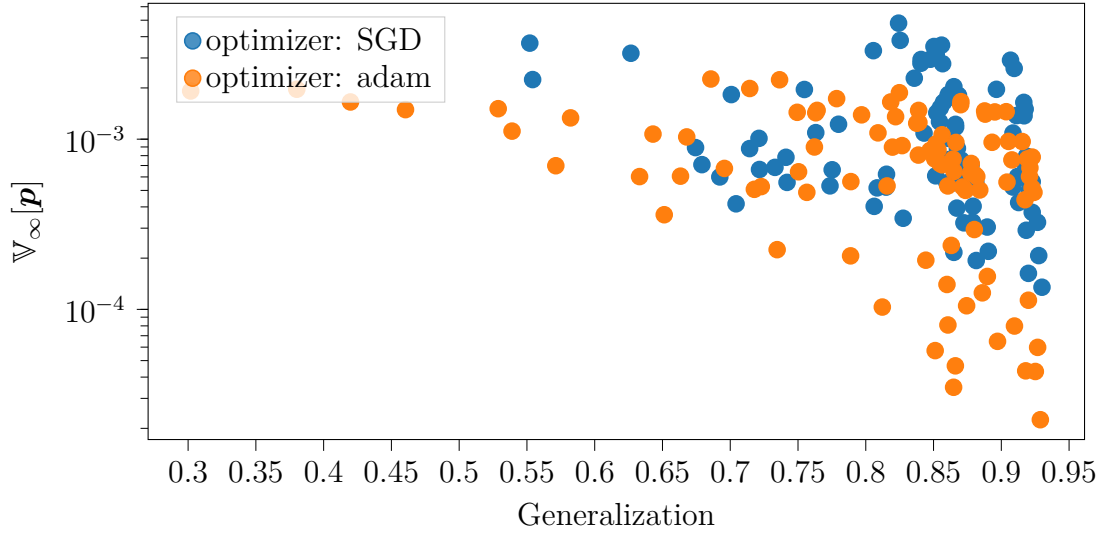


Figure A.14: Influence of optimizer hyperparameter on L^∞ -normalized variance of EW product distribution. Each color corresponds to a different value of the optimizer. The vertical axis is logarithmically scaled.

Appendix B

Omitted Figures for Chapter 6

This appendix gives the figures that had to be omitted from chapter 6 for reasons of brevity.

B.1 Experimental Study of Binary Partition Characteristics and Generalization

B.1.1 As Measured Using Silhouette

B.1.1.1 Relation with Generalization

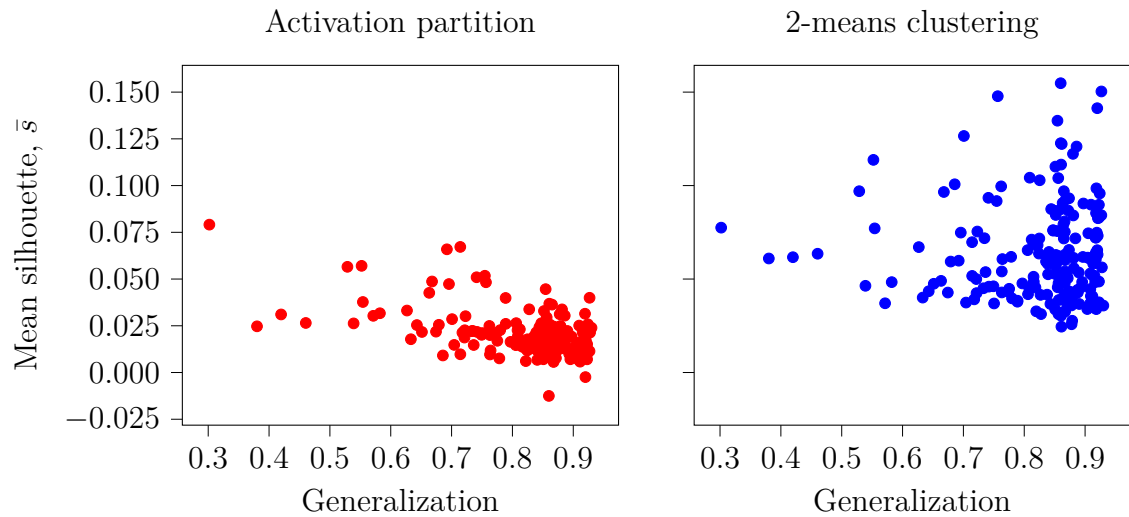


Figure B.1: Comparison of mean silhouette of activation partition and 2-means clustering with internal representation using all weight components.

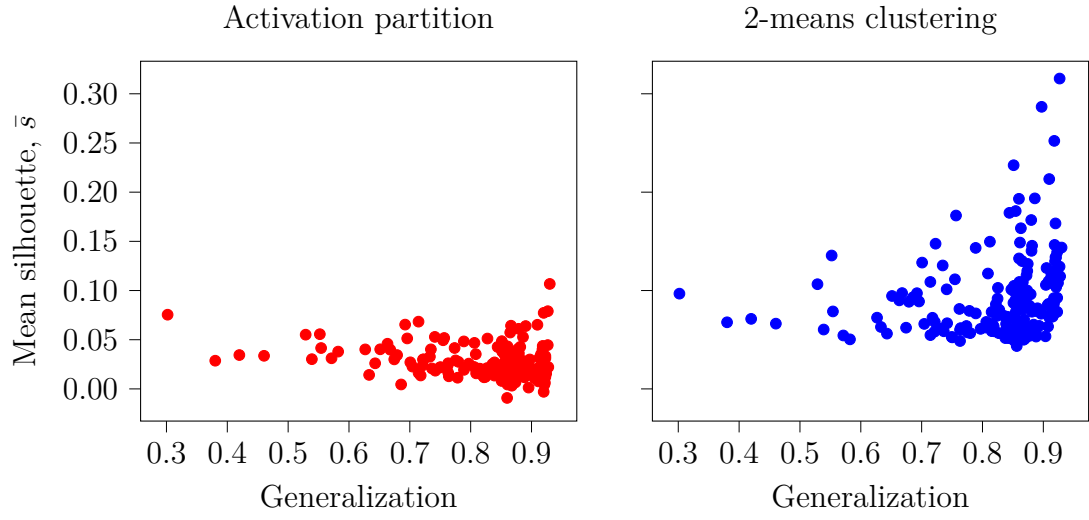


Figure B.2: Comparison of mean silhouette of activation partition and 2-means clustering with internal representation using top-10 % magnitude weight components.

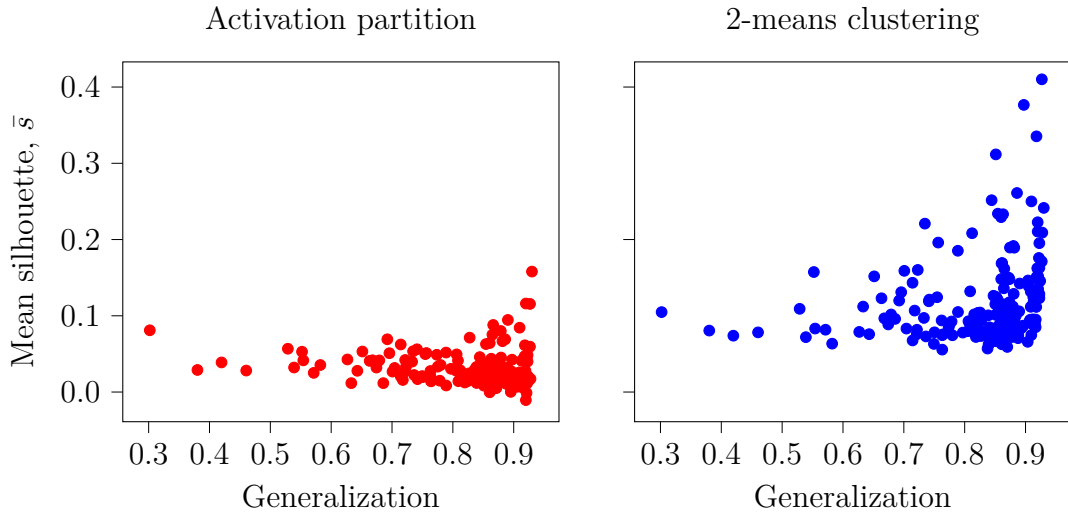


Figure B.3: Comparison of mean silhouette of activation partition and 2-means clustering with internal representation using top-5 % magnitude weight components.

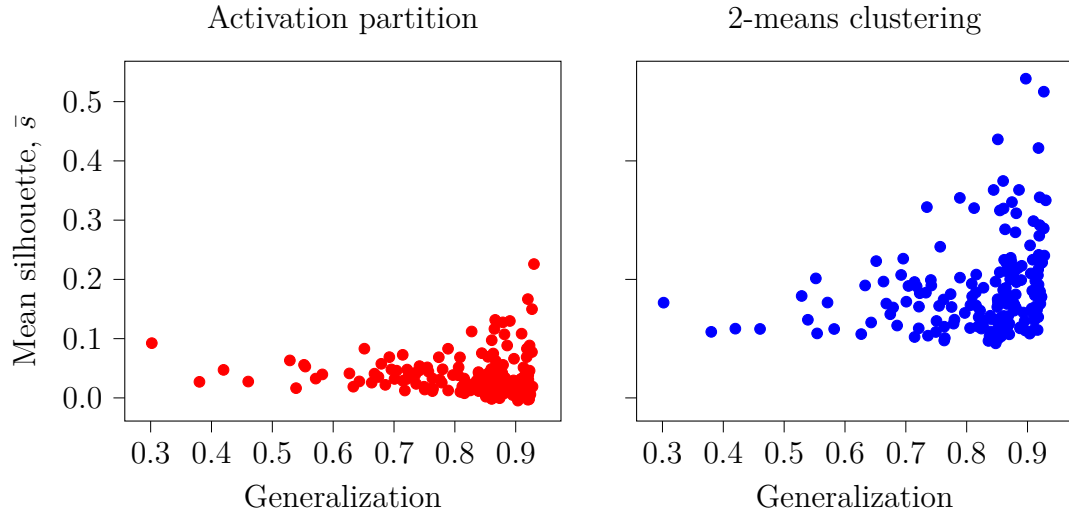


Figure B.4: Comparison of mean silhouette of activation partition and 2-means clustering with internal representation using top-2% magnitude weight components.

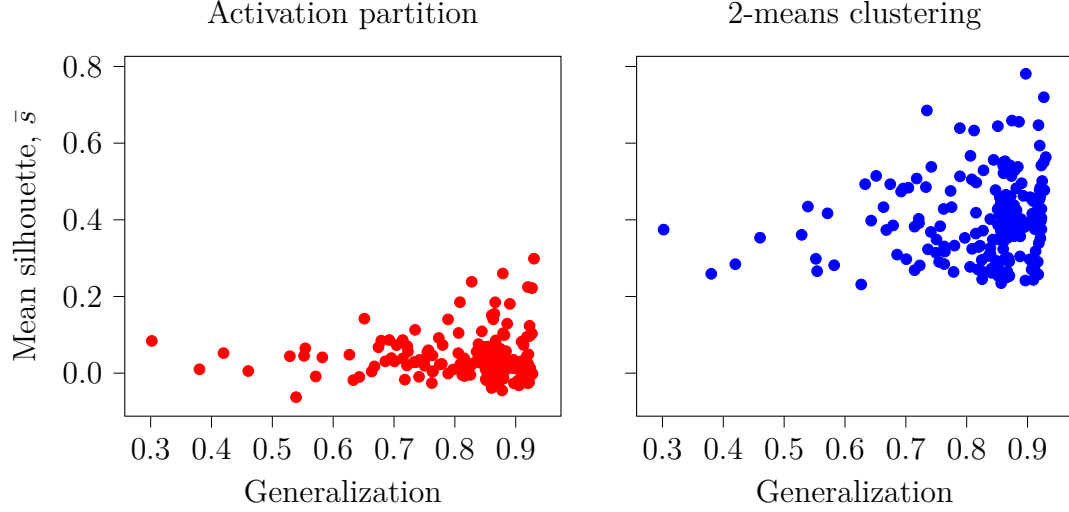


Figure B.5: Comparison of mean silhouette of activation partition and 2-means clustering with internal representation using top-0.5% magnitude weight components.

B.1.1.2 Influence of Hyperparameters

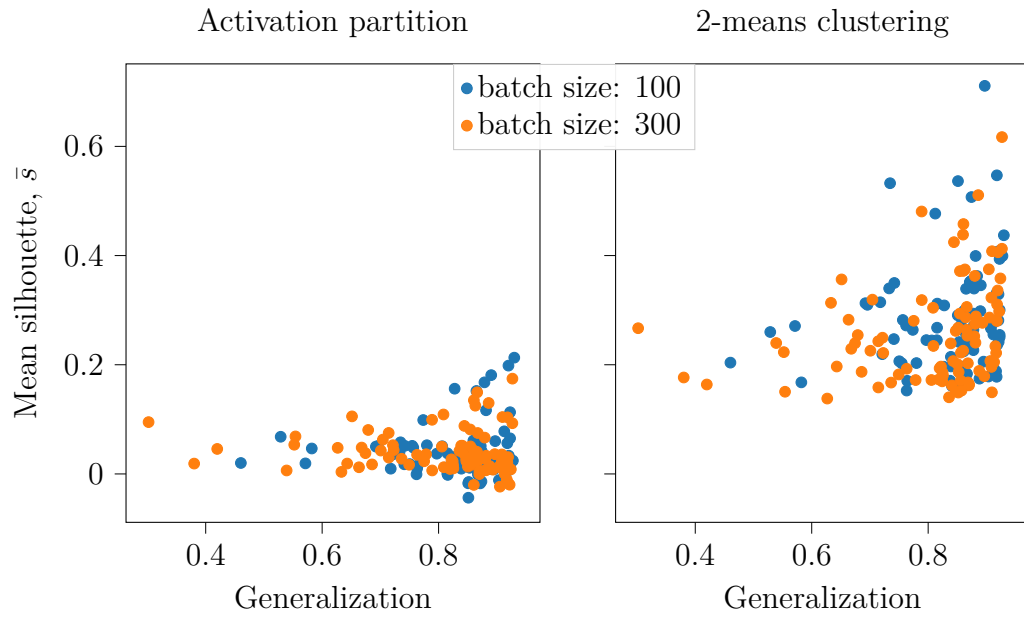


Figure B.6: Influence of batch size on mean silhouette with internal representation using top-1 % magnitude weight components.

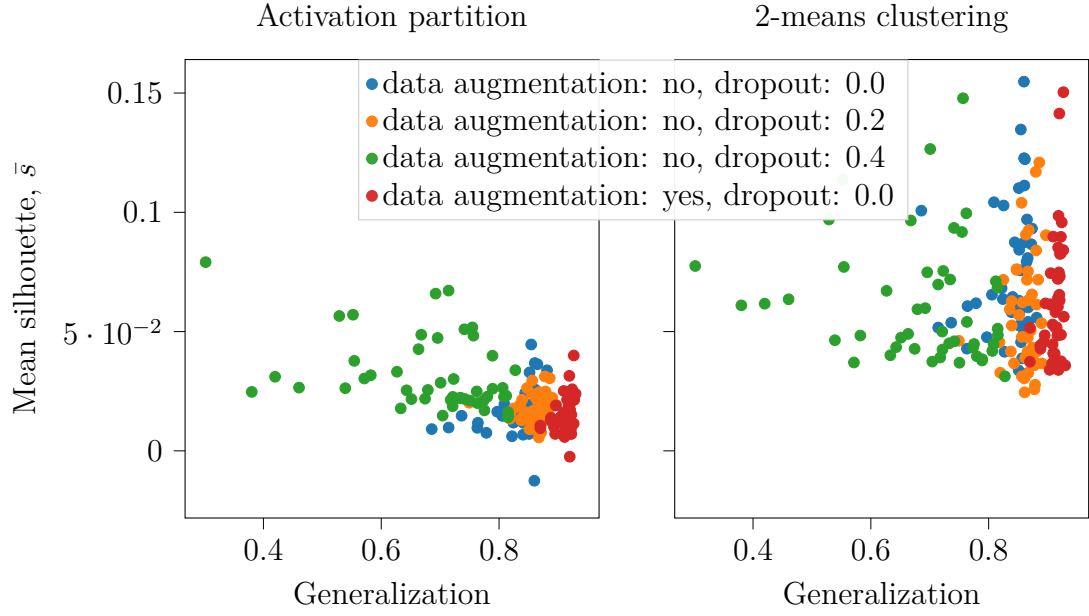


Figure B.7: Influence of noise regularization on mean silhouette with internal representation using all weight components.

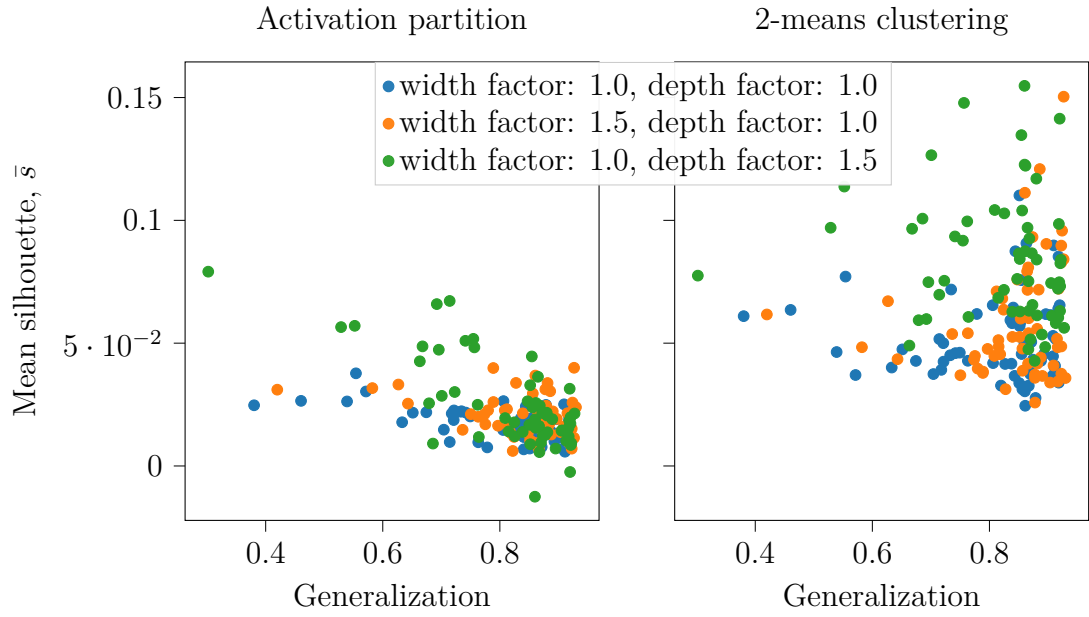


Figure B.8: Influence of architecture on mean silhouette with internal representation using all weight components.

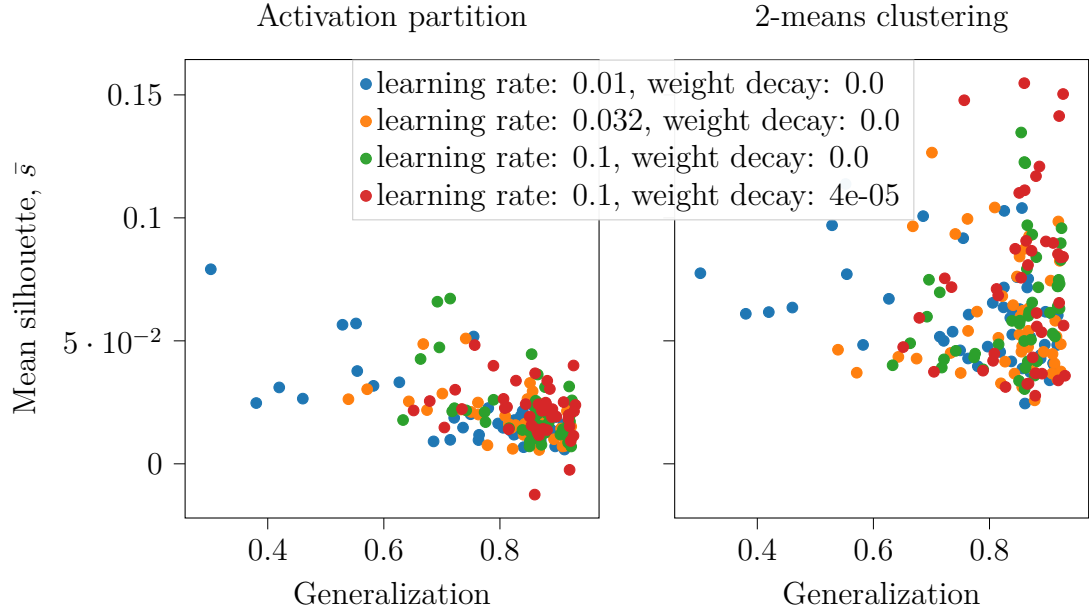


Figure B.9: Influence of effective learning rate on mean silhouette with internal representation using all weight components.

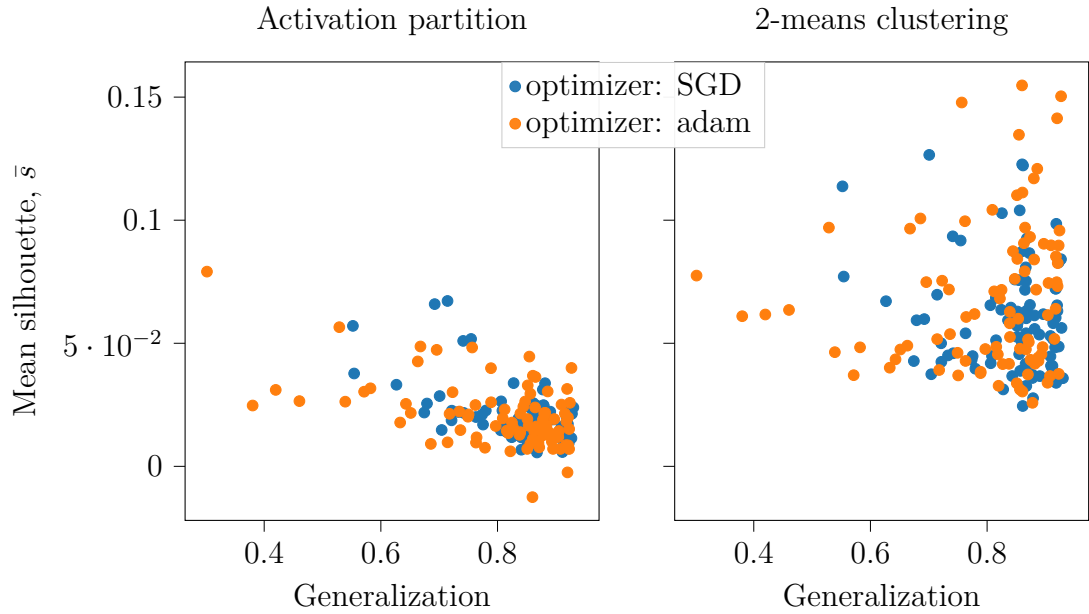


Figure B.10: Influence of optimizer on mean silhouette with internal representation using all weight components.

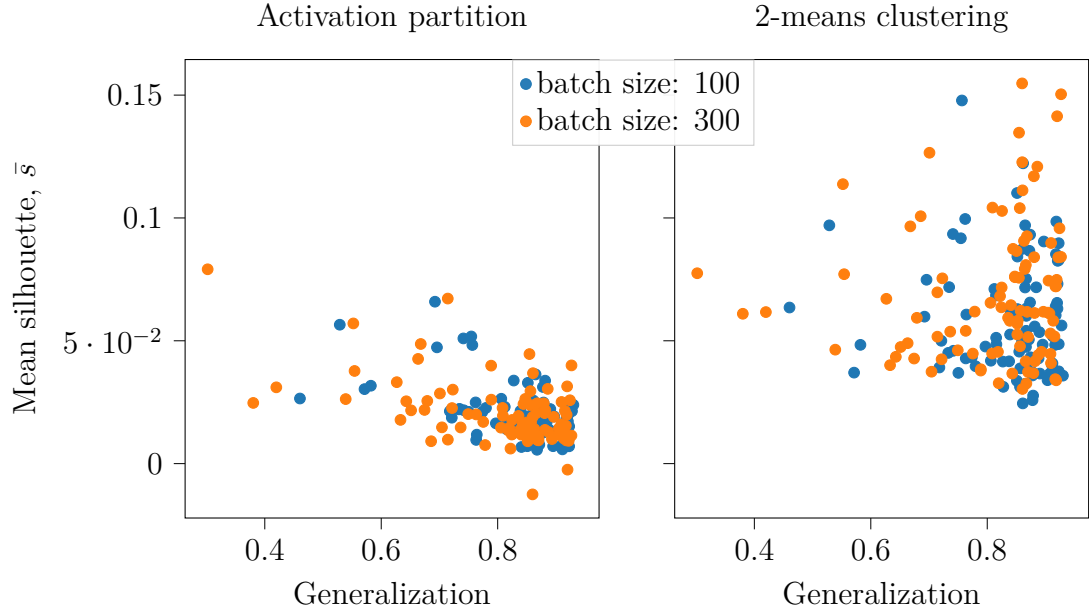


Figure B.11: Influence of batch size on mean silhouette with internal representation using all weight components.

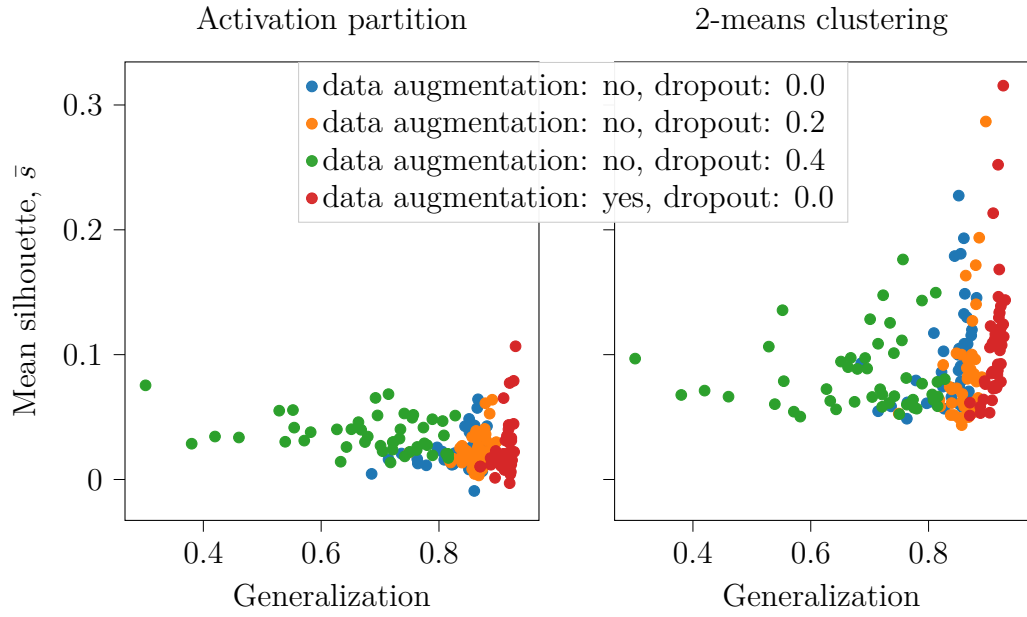


Figure B.12: Influence of noise regularization on mean silhouette with internal representation using top-10% magnitude weight components.

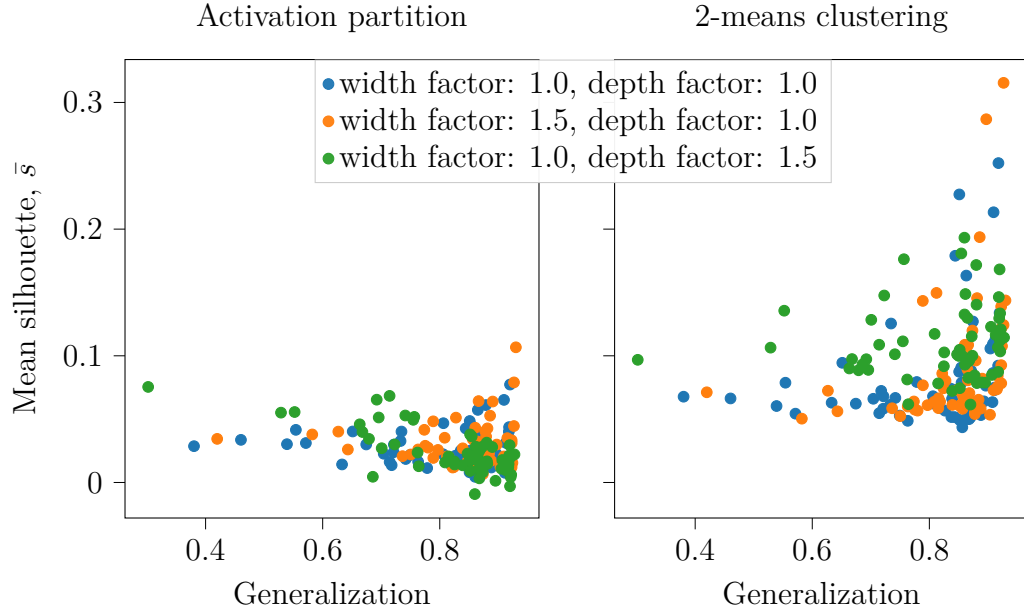


Figure B.13: Influence of architecture on mean silhouette with internal representation using top-10 % magnitude weight components.

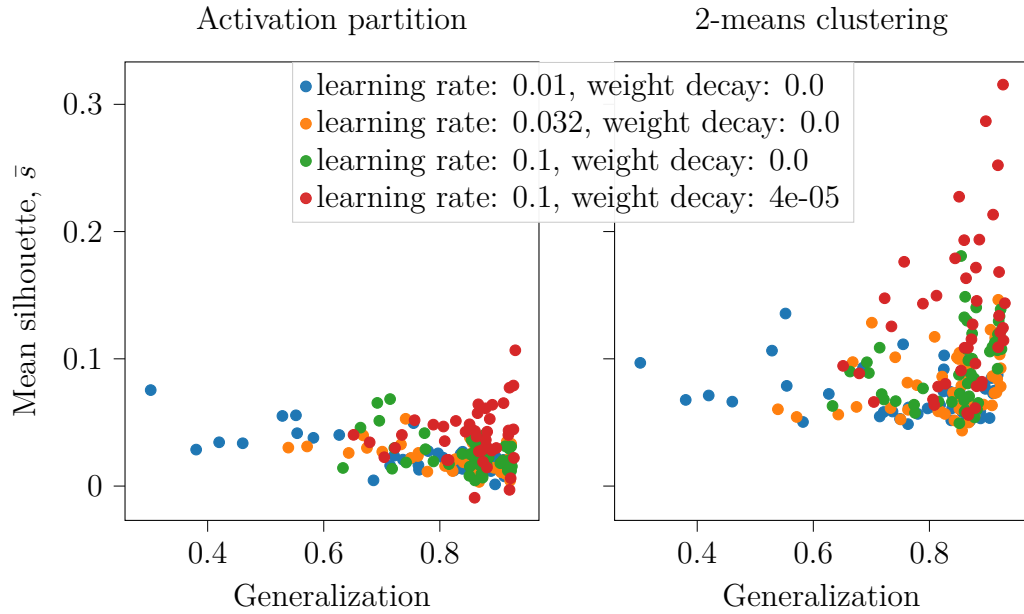


Figure B.14: Influence of effective learning rate on mean silhouette with internal representation using top-10 % magnitude weight components.

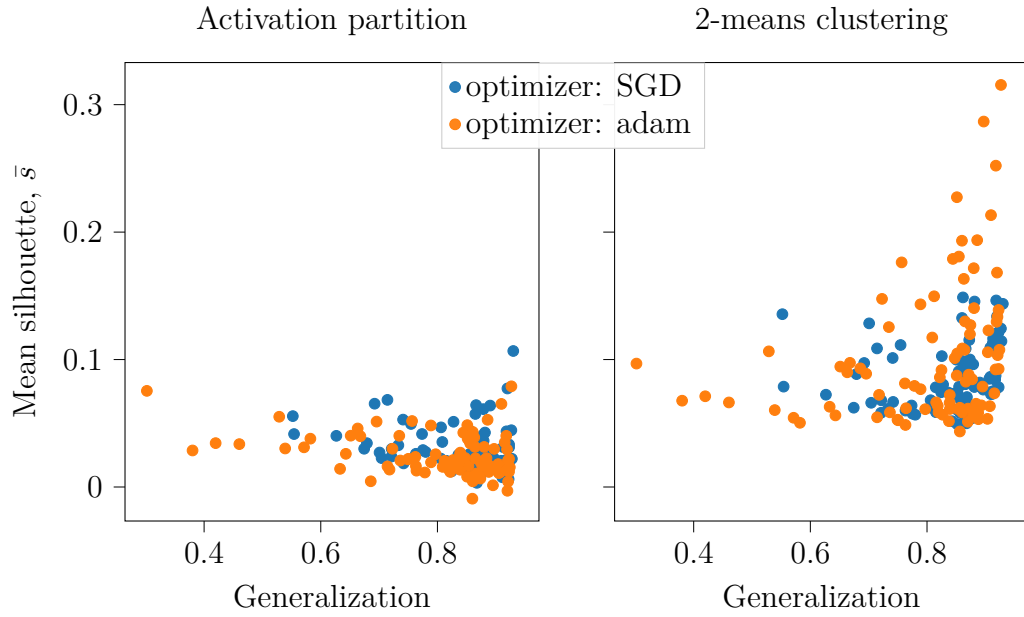


Figure B.15: Influence of optimizer on mean silhouette with internal representation using top-10 % magnitude weight components.

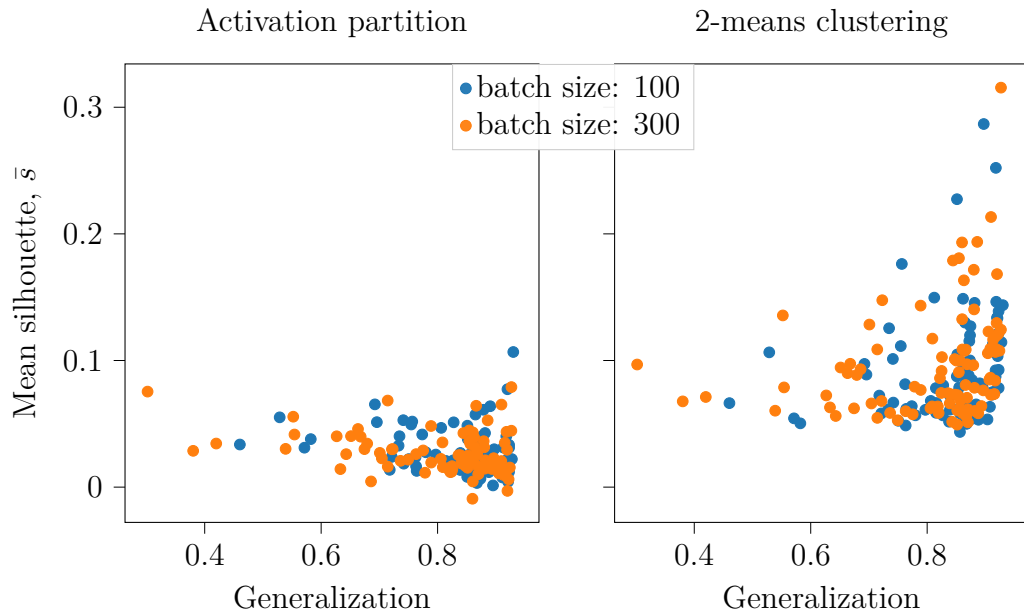


Figure B.16: Influence of batch size on mean silhouette with internal representation using top-10 % magnitude weight components.

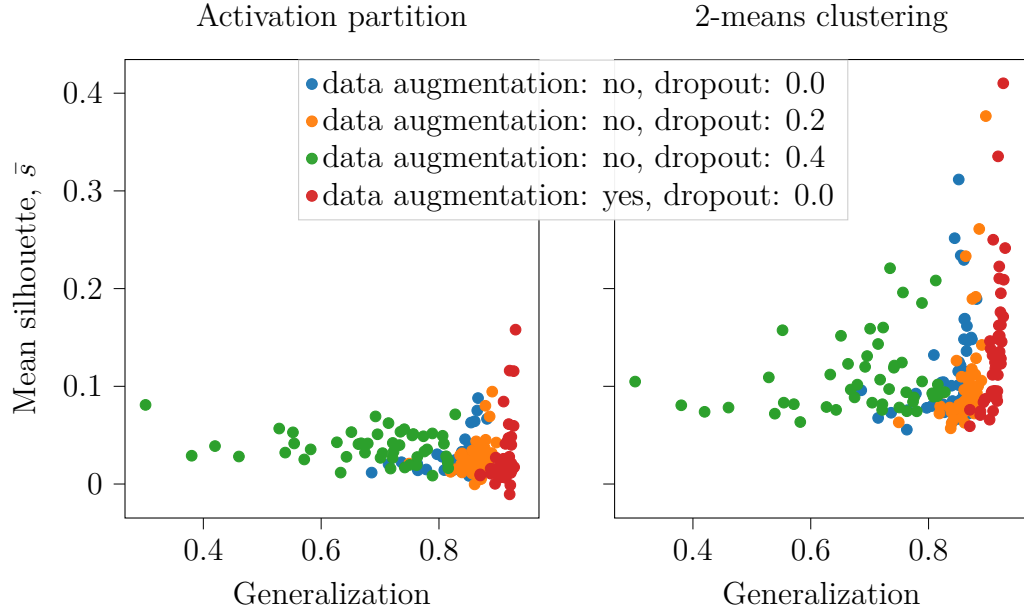


Figure B.17: Influence of noise regularization on mean silhouette with internal representation using top-5 % magnitude weight components.

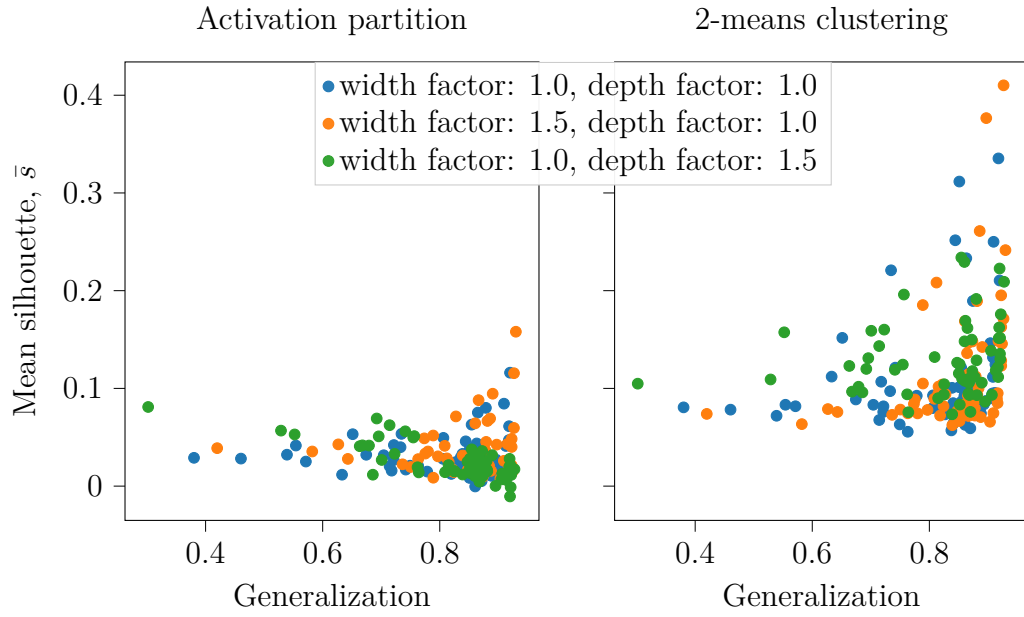


Figure B.18: Influence of architecture on mean silhouette with internal representation using top-5 % magnitude weight components.

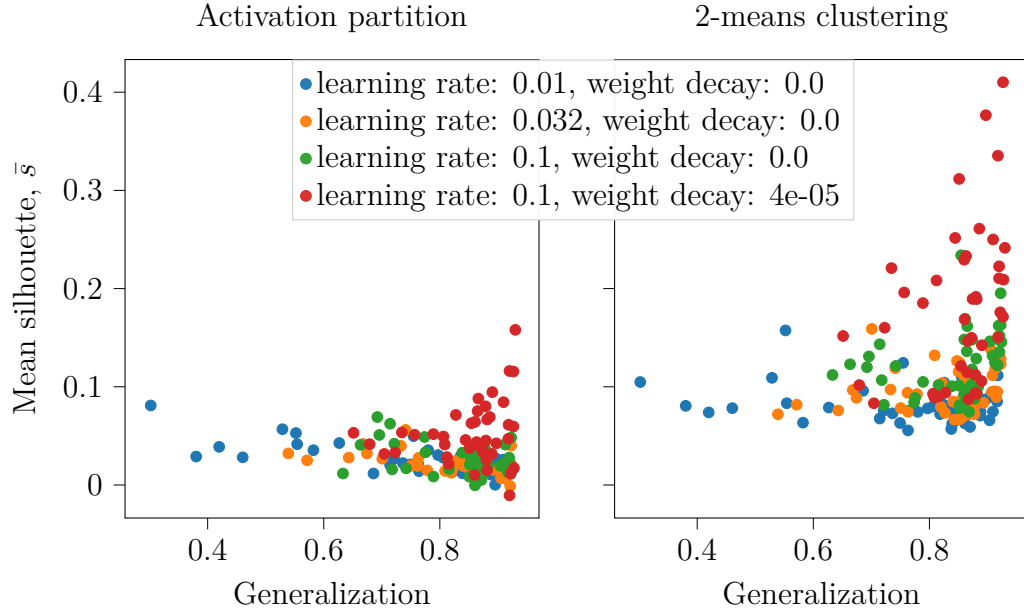


Figure B.19: Influence of effective learning rate on mean silhouette with internal representation using top-5 % magnitude weight components.

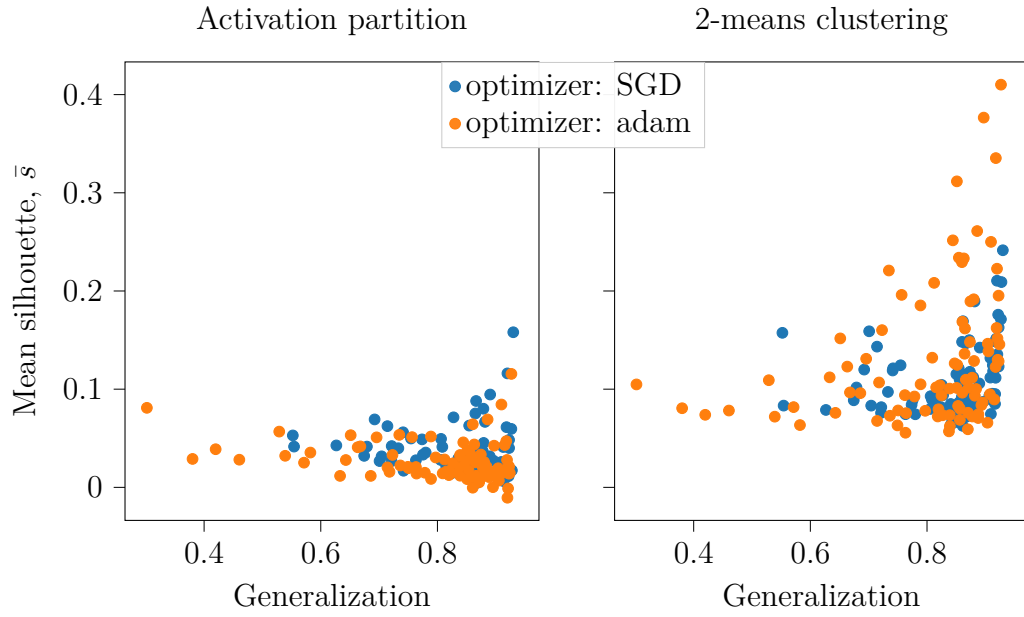


Figure B.20: Influence of optimizer on mean silhouette with internal representation using top-5 % magnitude weight components.

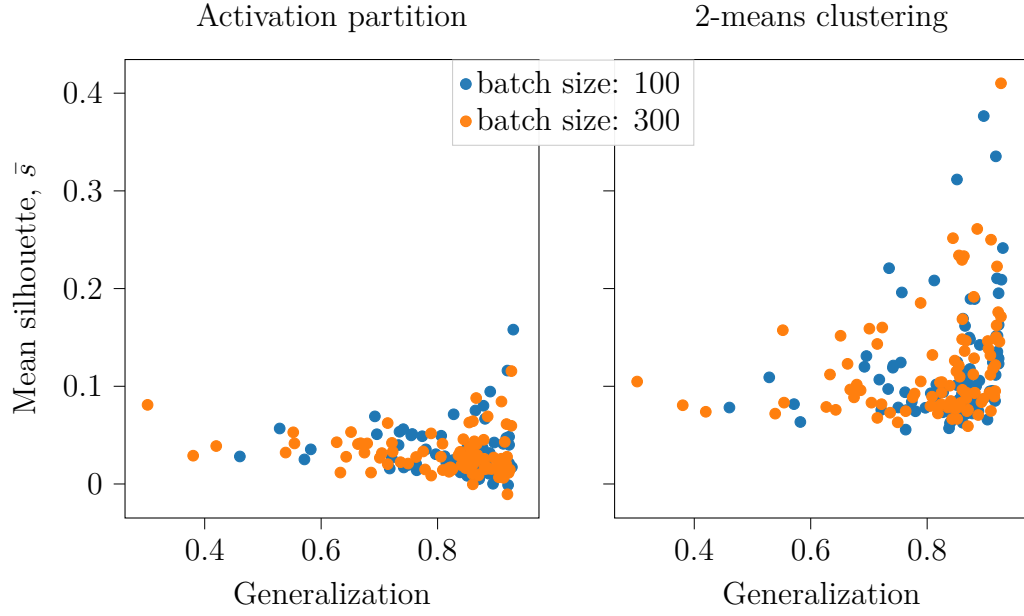


Figure B.21: Influence of batch size on mean silhouette with internal representation using top-5 % magnitude weight components.

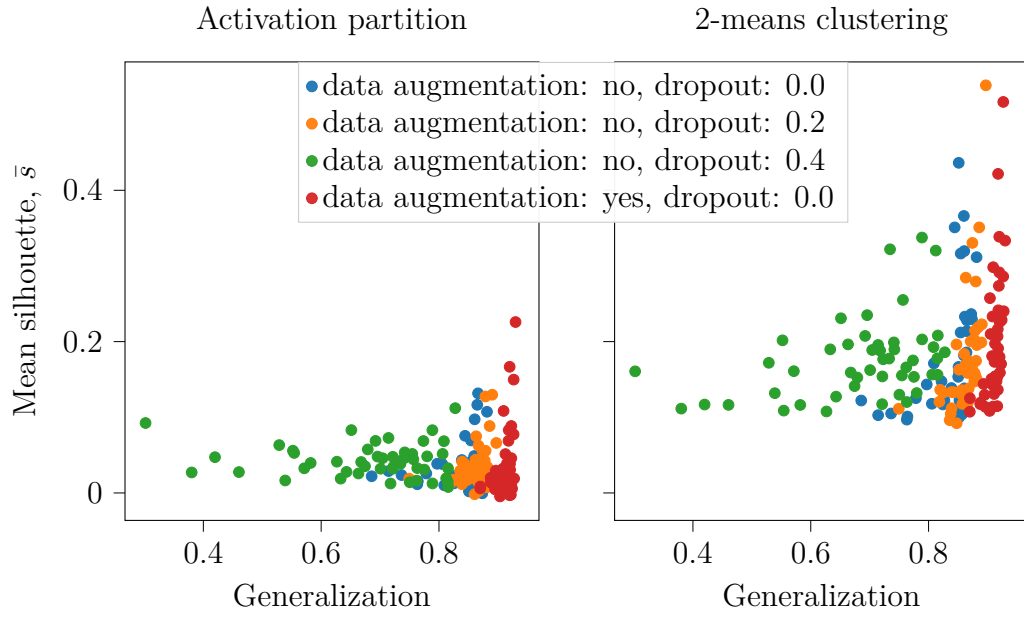


Figure B.22: Influence of noise regularization on mean silhouette with internal representation using top-2 % magnitude weight components.

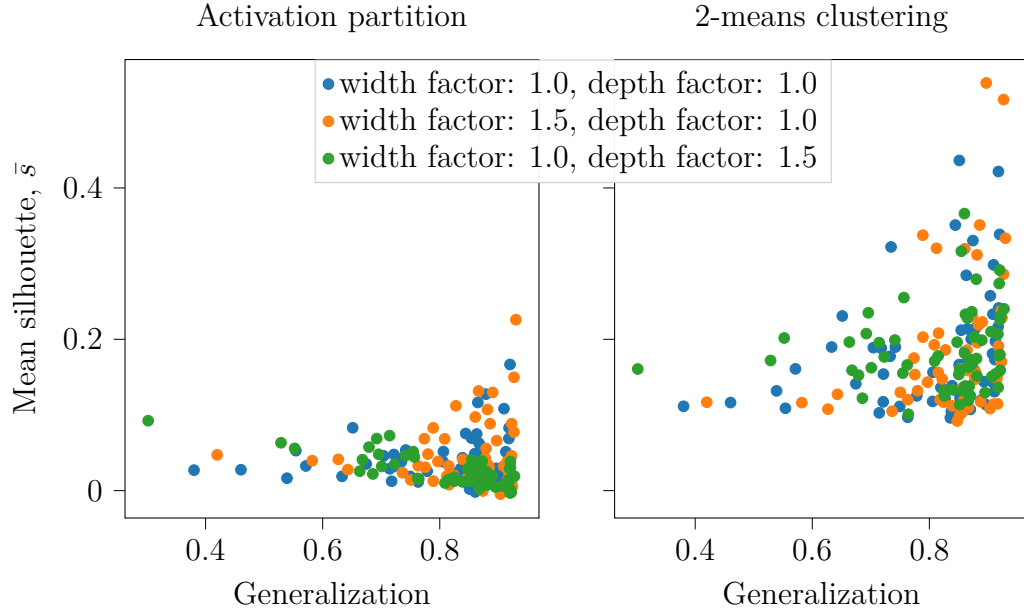


Figure B.23: Influence of architecture on mean silhouette with internal representation using top-2% magnitude weight components.

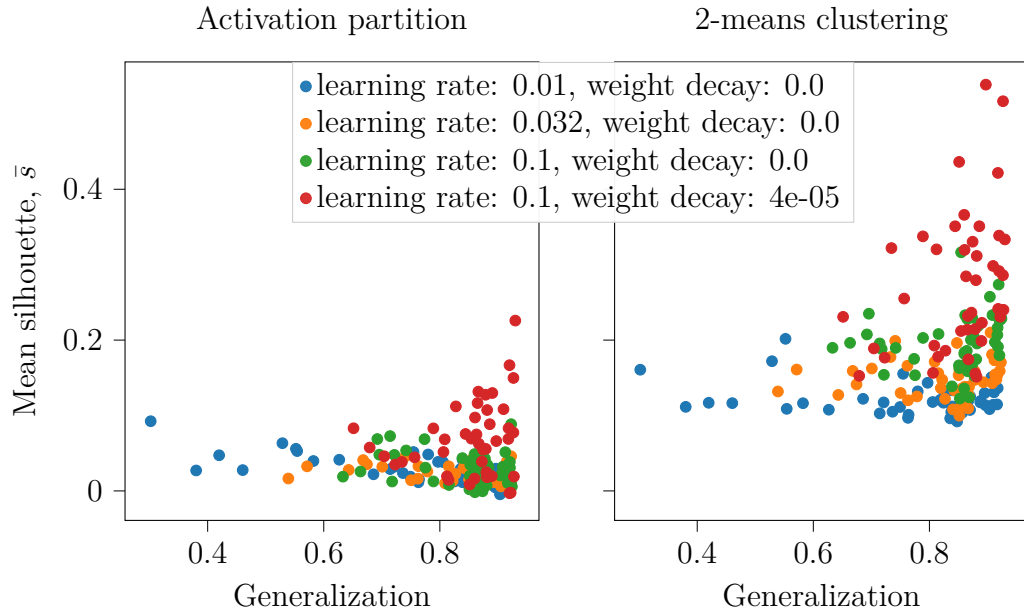


Figure B.24: Influence of effective learning rate on mean silhouette with internal representation using top-2% magnitude weight components.

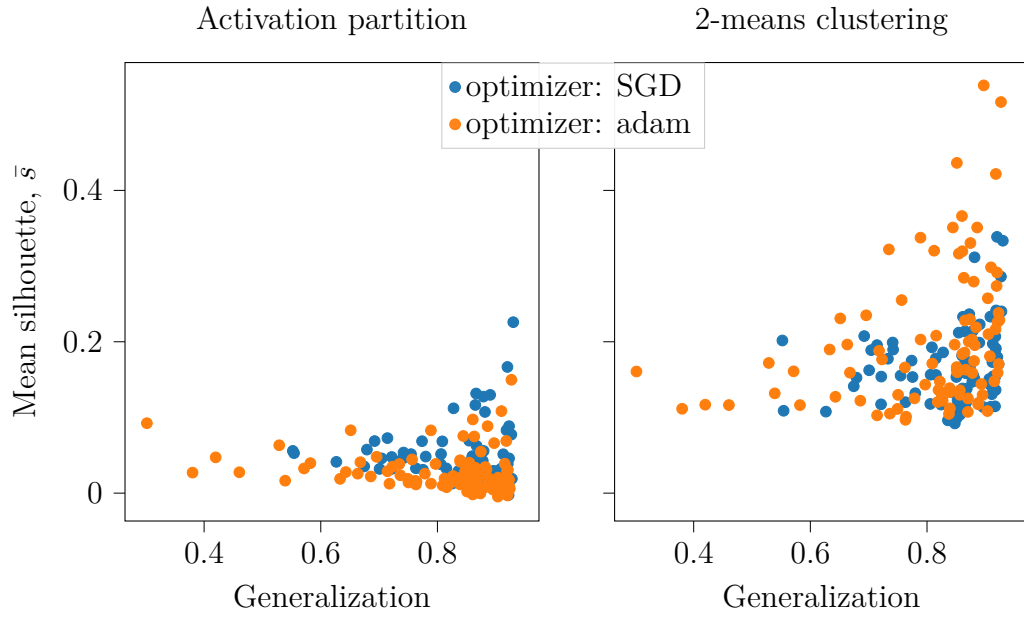


Figure B.25: Influence of optimizer on mean silhouette with internal representation using top-2 % magnitude weight components.

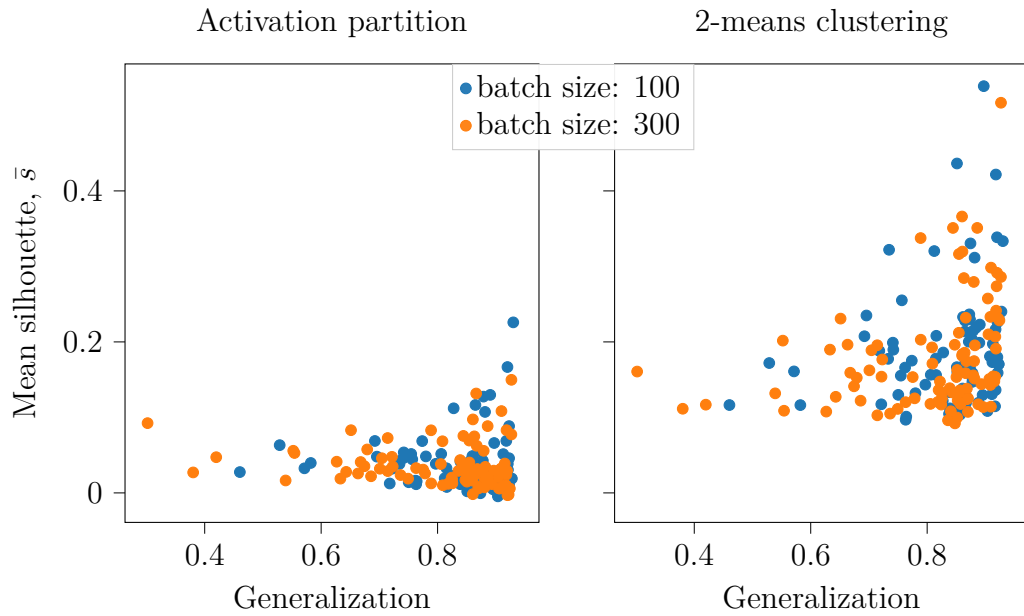


Figure B.26: Influence of batch size on mean silhouette with internal representation using top-2 % magnitude weight components.

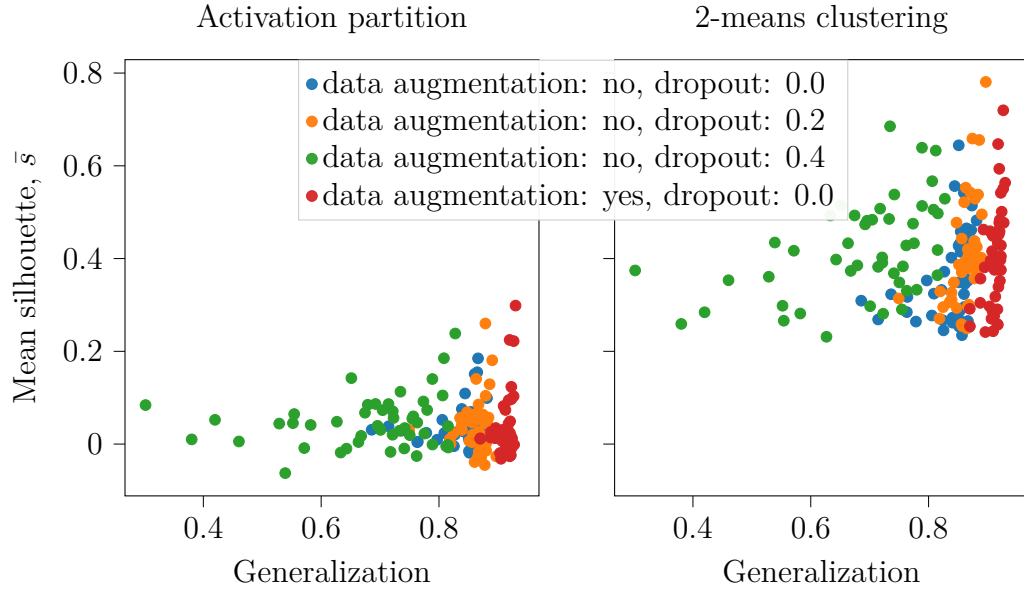


Figure B.27: Influence of noise regularization on mean silhouette with internal representation using top-0.5% magnitude weight components.

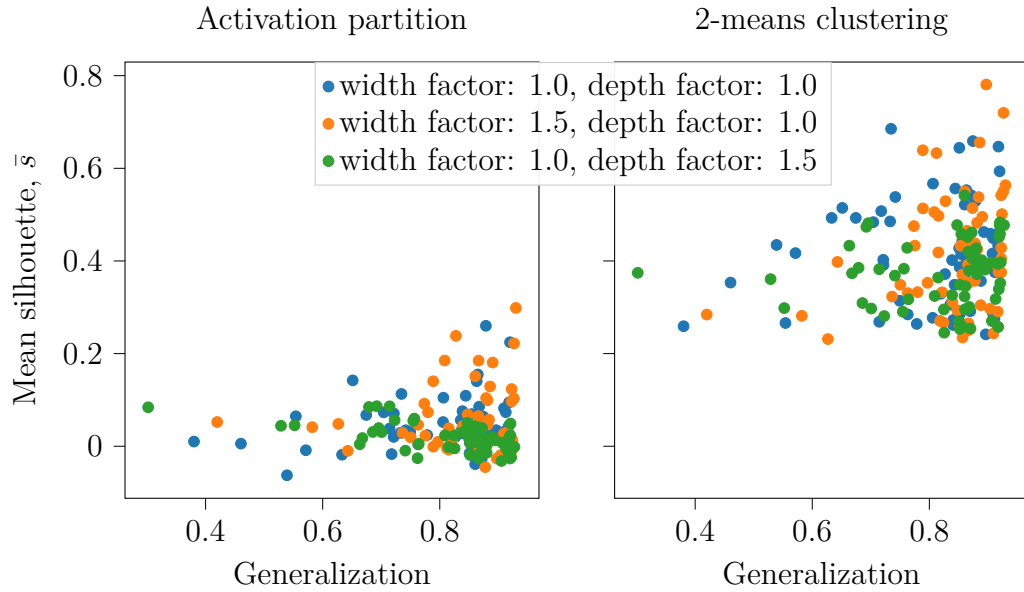


Figure B.28: Influence of architecture on mean silhouette with internal representation using top-0.5% magnitude weight components.

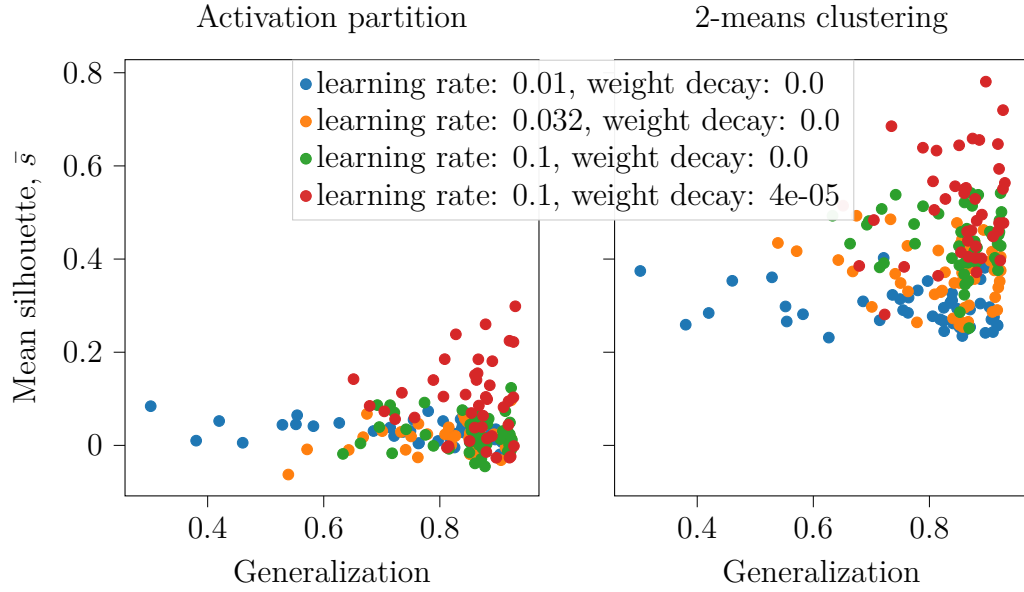


Figure B.29: Influence of effective learning rate on mean silhouette with internal representation using top-0.5% magnitude weight components.

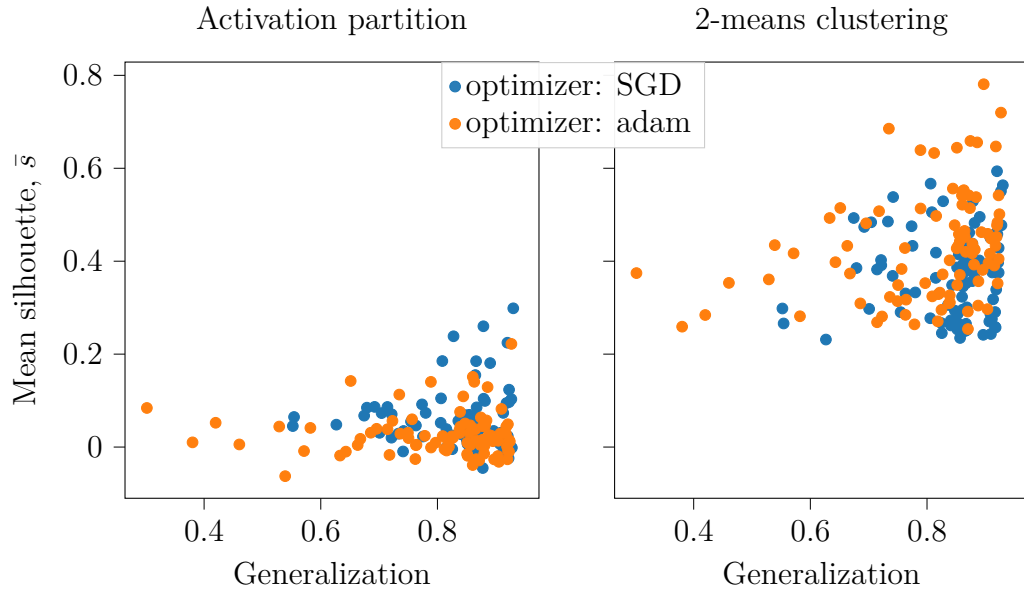


Figure B.30: Influence of optimizer on mean silhouette with internal representation using top-0.5% magnitude weight components.

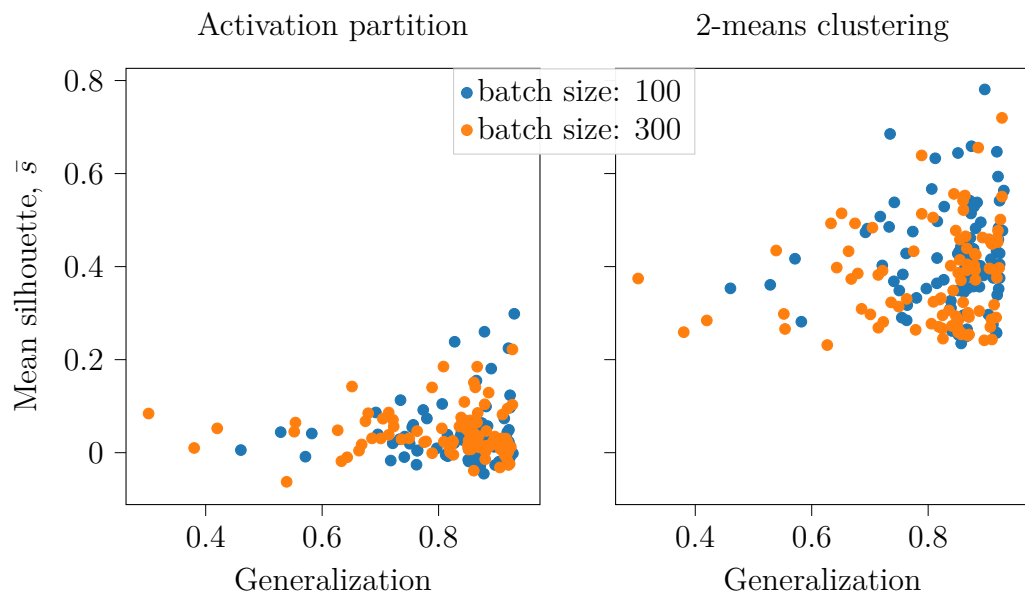


Figure B.31: Influence of batch size on mean silhouette with internal representation using top-0.5 % magnitude weight components.

B.1.2 As Measured Using Information Theory

B.1.2.1 Relation with Generalization

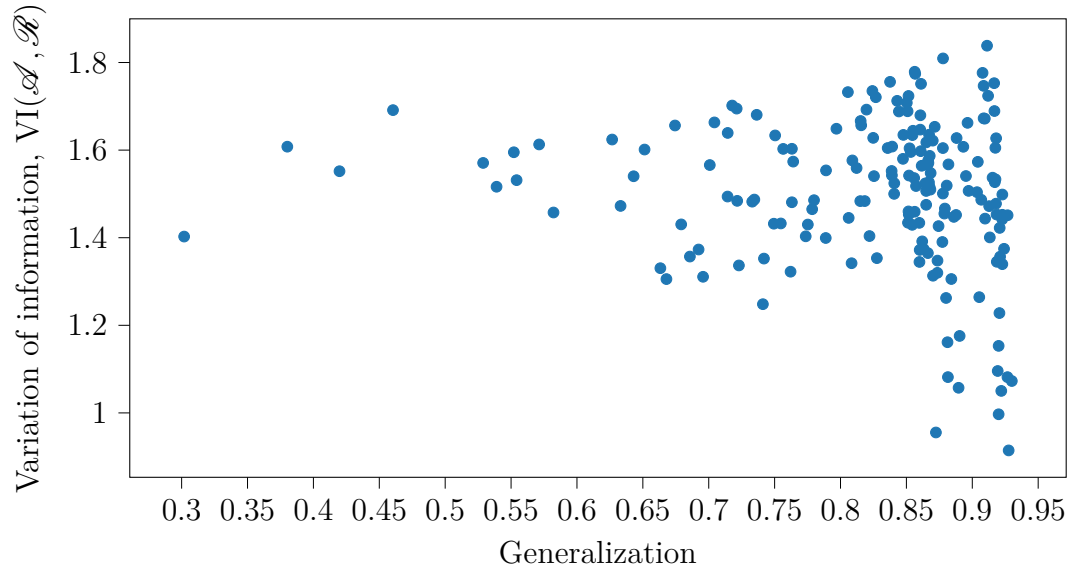


Figure B.32: Variation of information of activation partition and 2-means clustering with internal representation using all weight components.

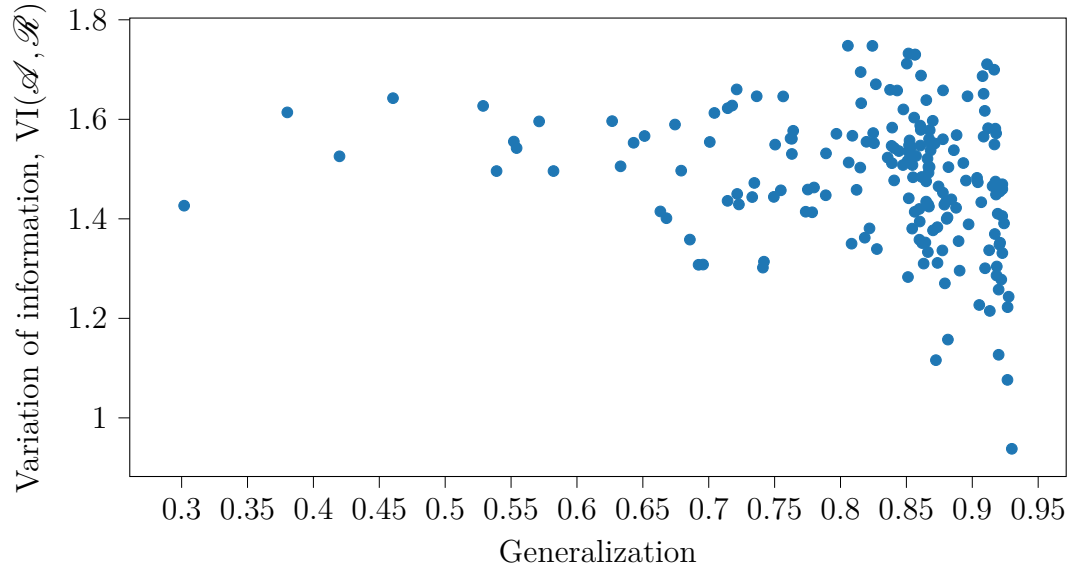


Figure B.33: Variation of information of activation partition and 2-means clustering with internal representation using top-10 % magnitude weight components.

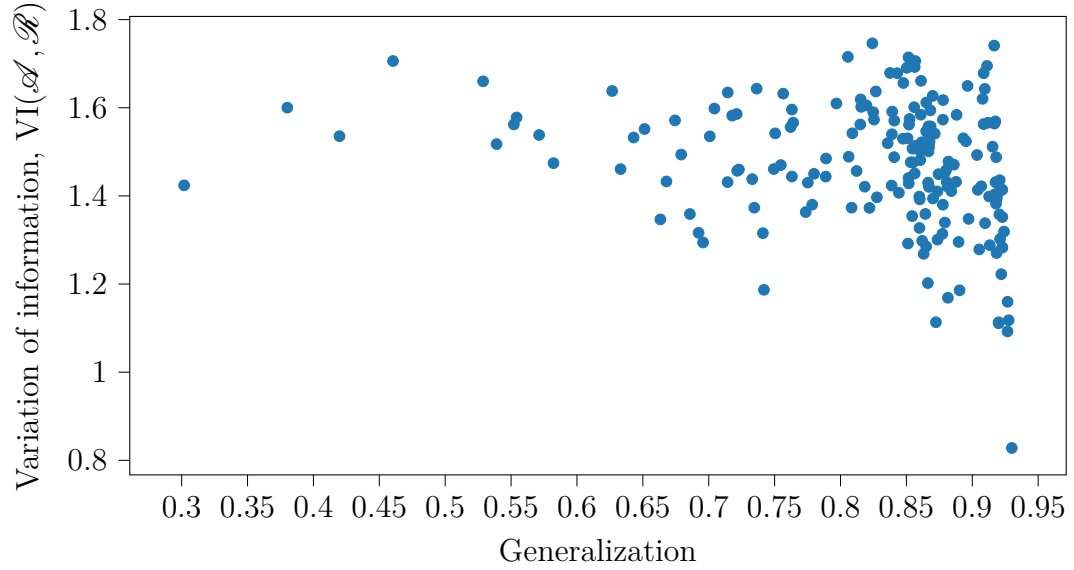


Figure B.34: Variation of information of activation partition and 2-means clustering with internal representation using top-5 % magnitude weight components.

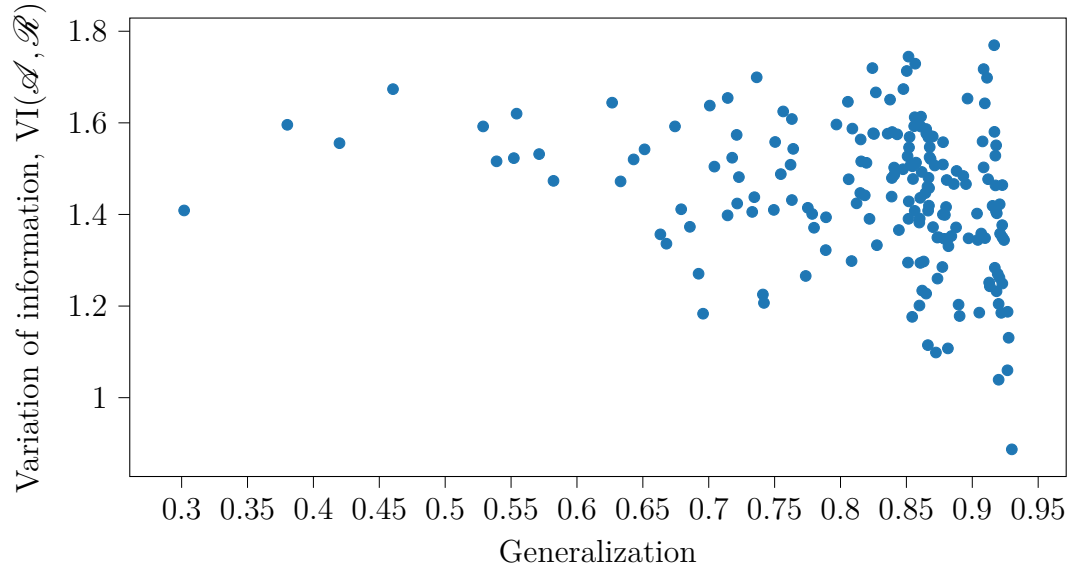


Figure B.35: Variation of information of activation partition and 2-means clustering with internal representation using top-2% magnitude weight components.

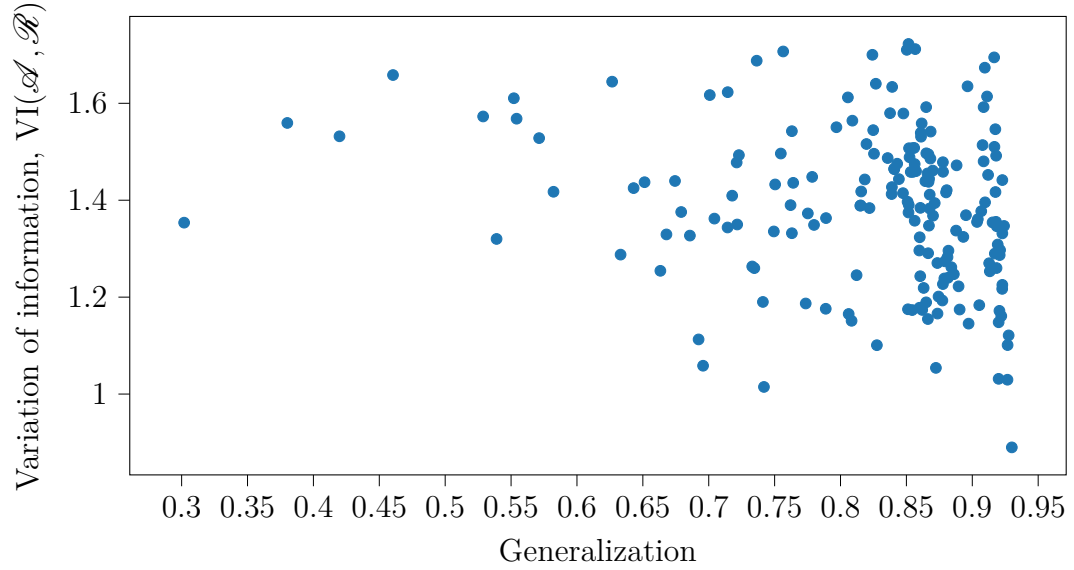


Figure B.36: Variation of information of activation partition and 2-means clustering with internal representation using top-0.5% magnitude weight components.

B.1.2.2 Influence of Hyperparameters

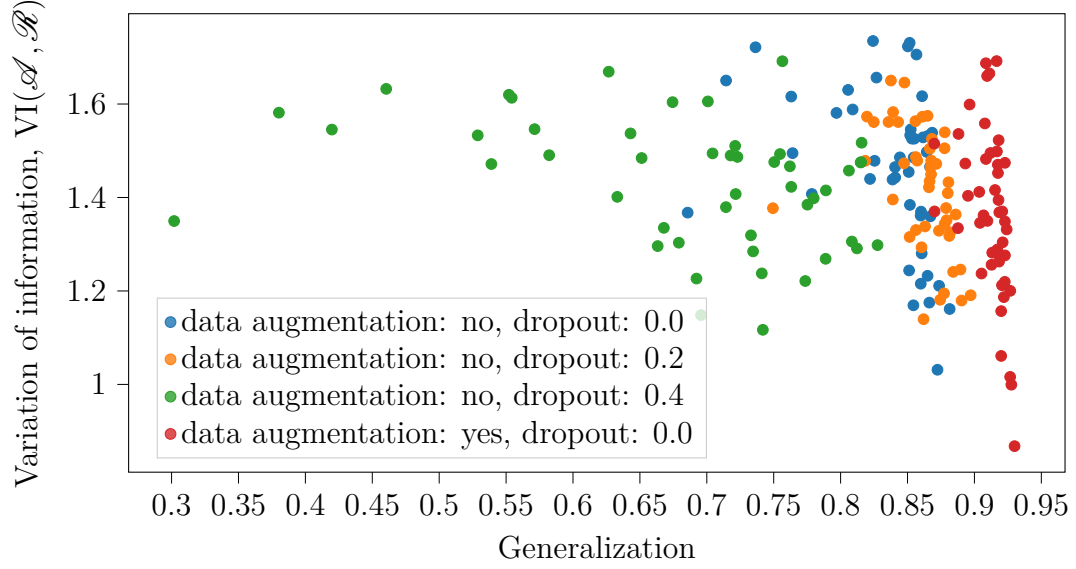


Figure B.37: Influence of noise regularization on variation of information with internal representation using top-1 % magnitude weight components.

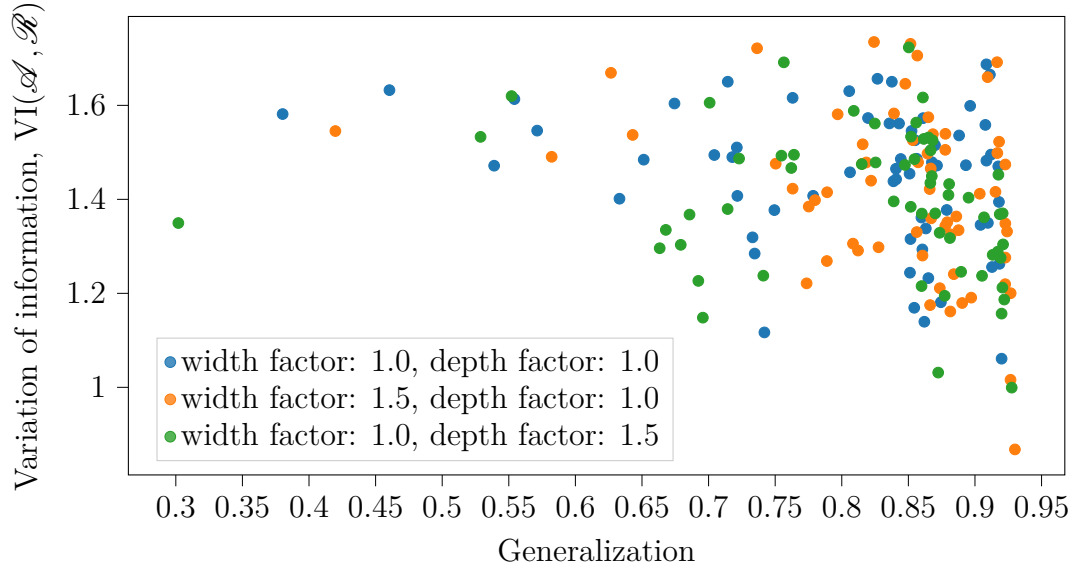


Figure B.38: Influence of architecture on variation of information with internal representation using top-1 % magnitude weight components.

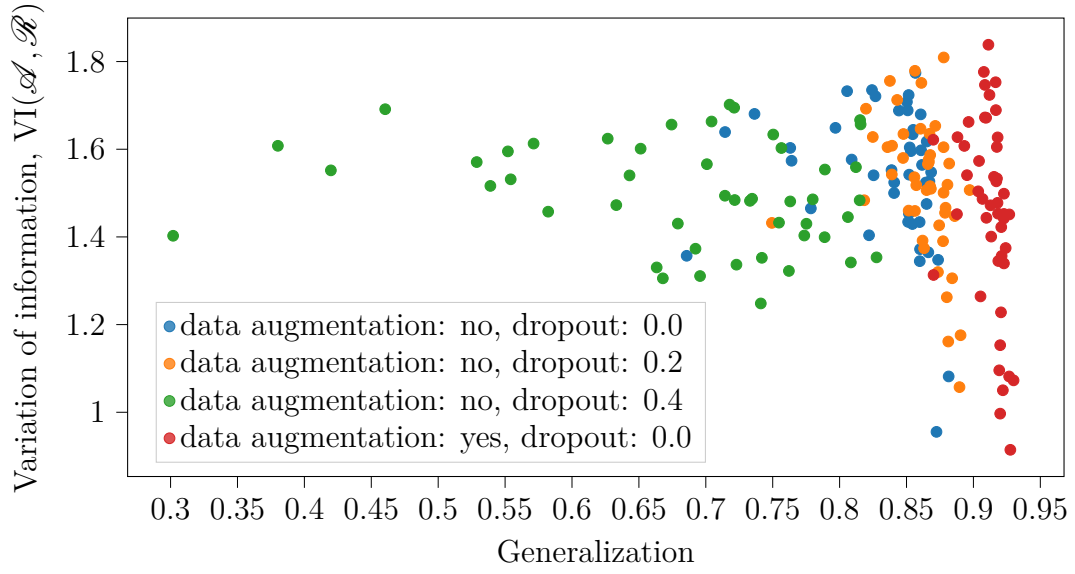


Figure B.39: Influence of noise regularization on variation of information with internal representation using all weight components.

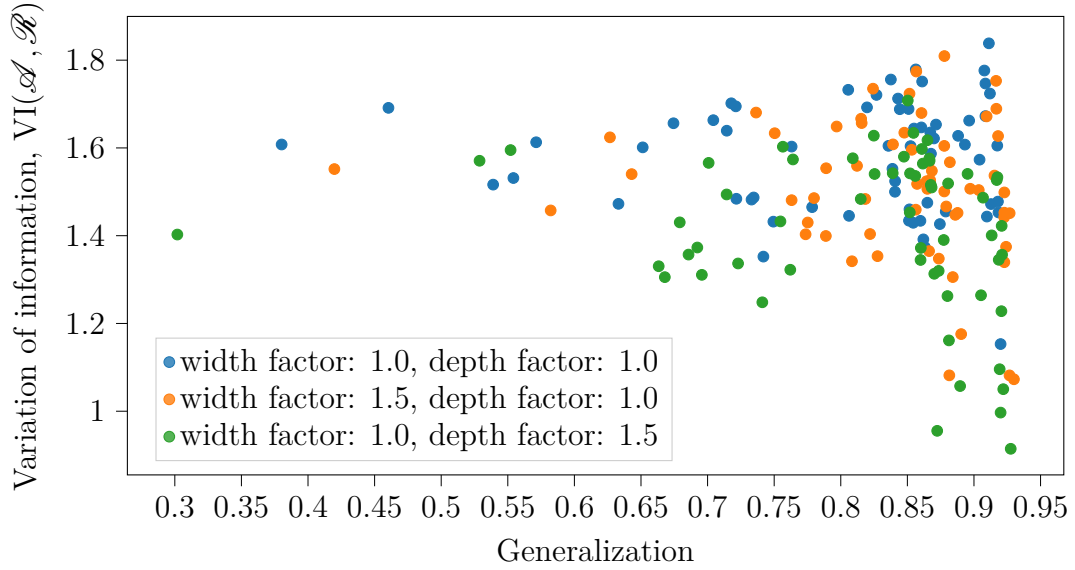


Figure B.40: Influence of architecture on variation of information with internal representation using all weight components.

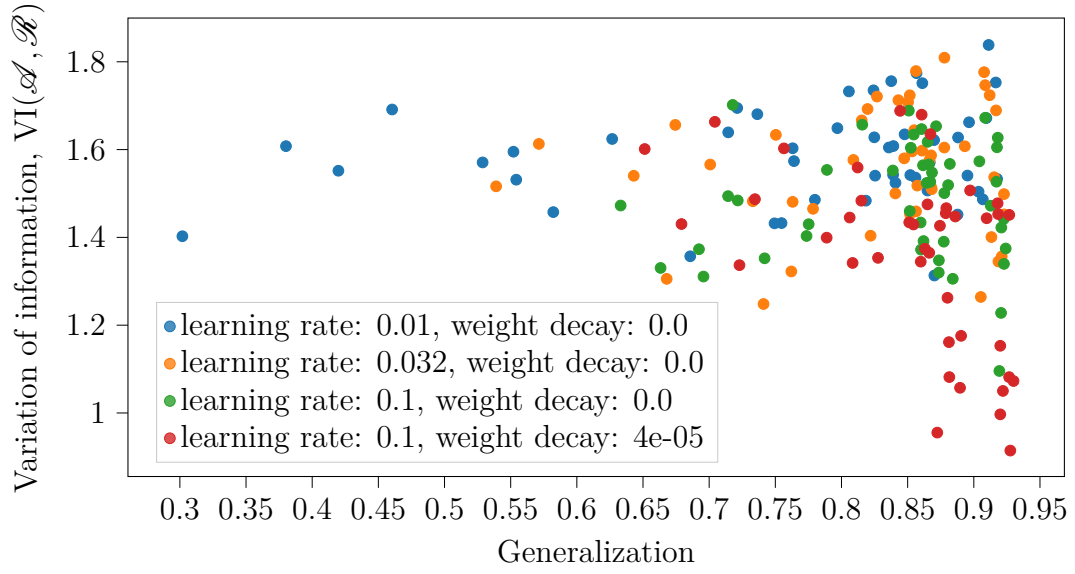


Figure B.41: Influence of effective learning rate on variation of information with internal representation using all weight components.

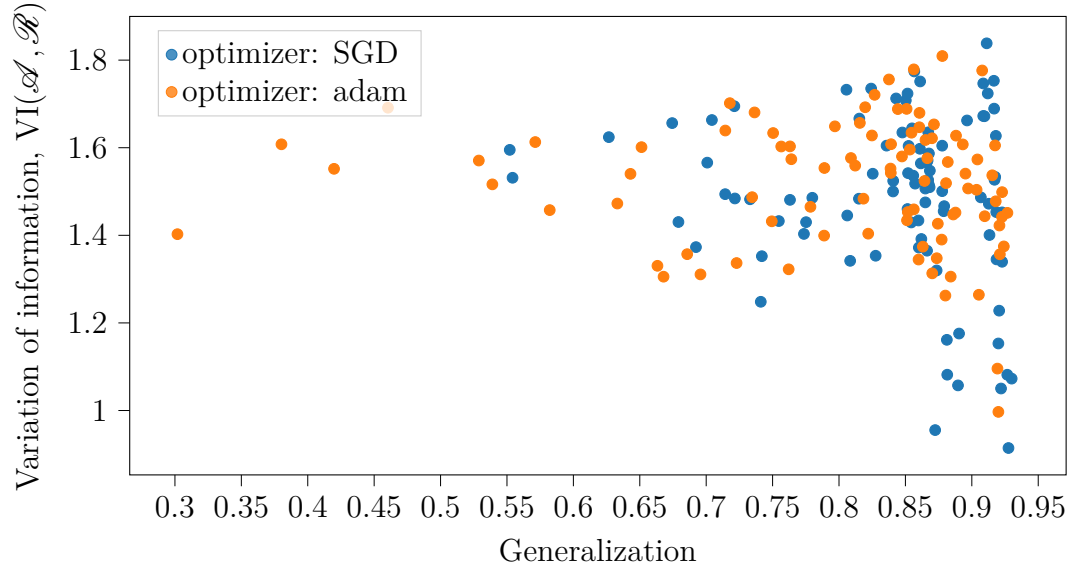


Figure B.42: Influence of optimizer on mean silhouette with internal representation using all weight components.

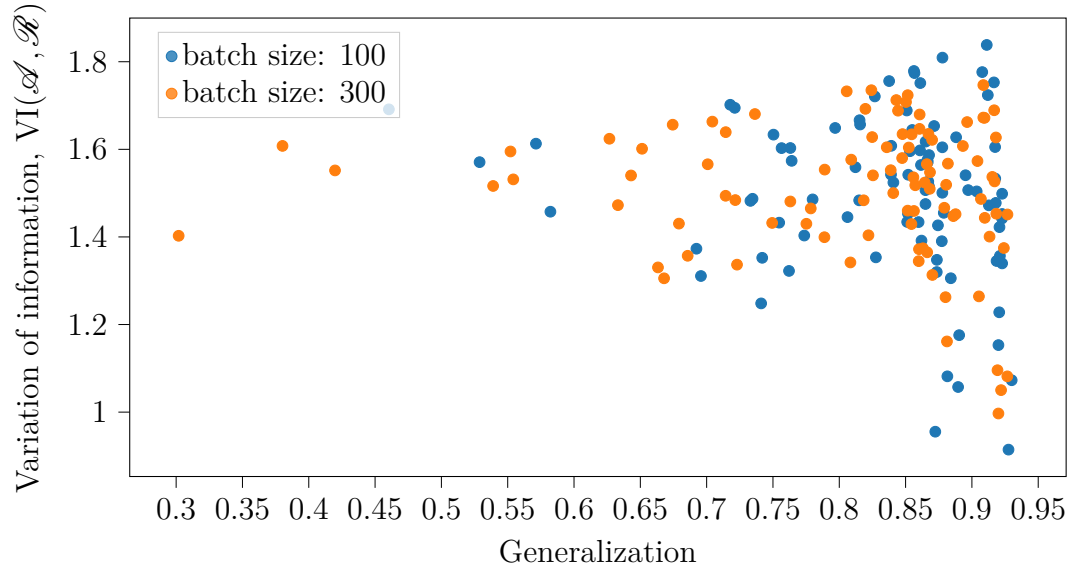


Figure B.43: Influence of batch size on variation of information with internal representation using all weight components.

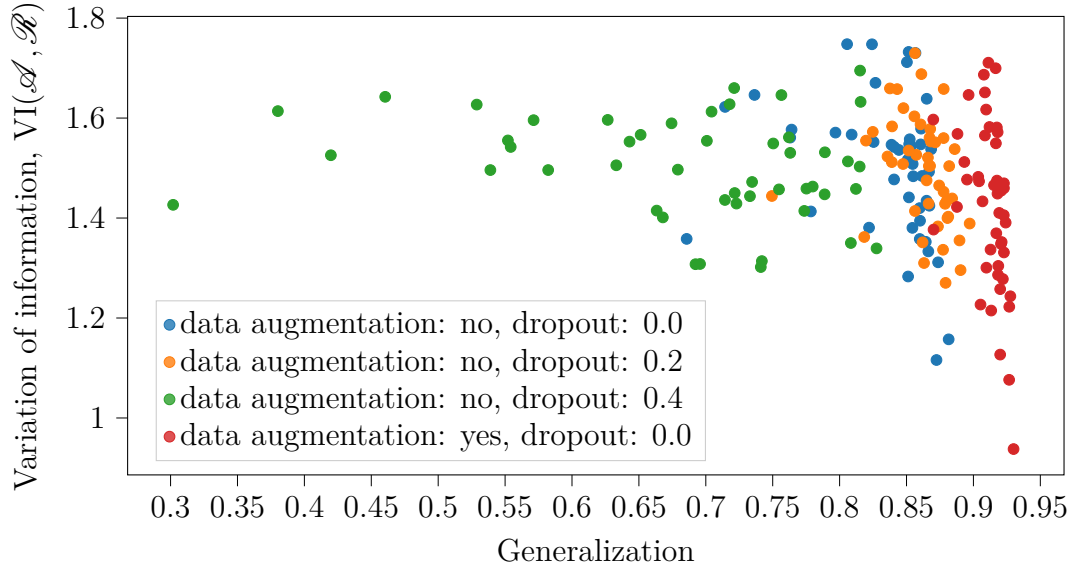


Figure B.44: Influence of noise regularization on variation of information with internal representation using top-10 % magnitude weight components.

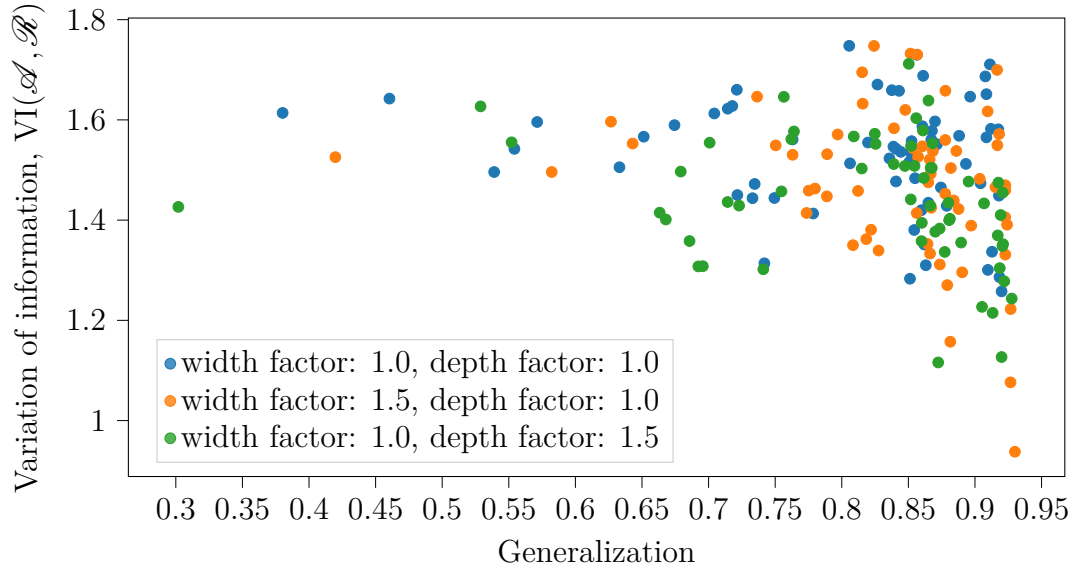


Figure B.45: Influence of architecture on variation of information with internal representation using top-10 % magnitude weight components.

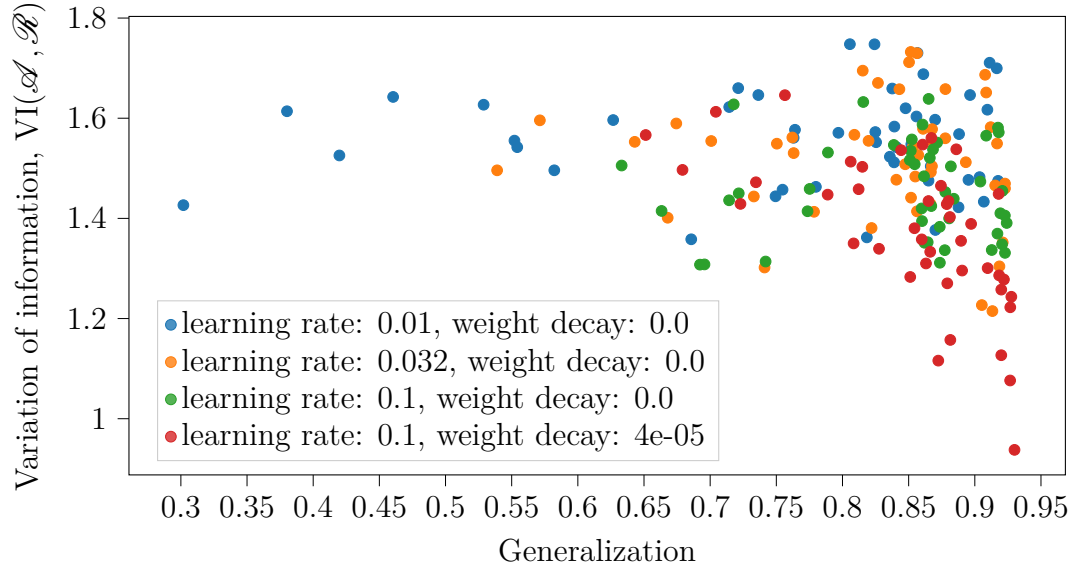


Figure B.46: Influence of effective learning rate on variation of information with internal representation using top-10 % magnitude weight components.

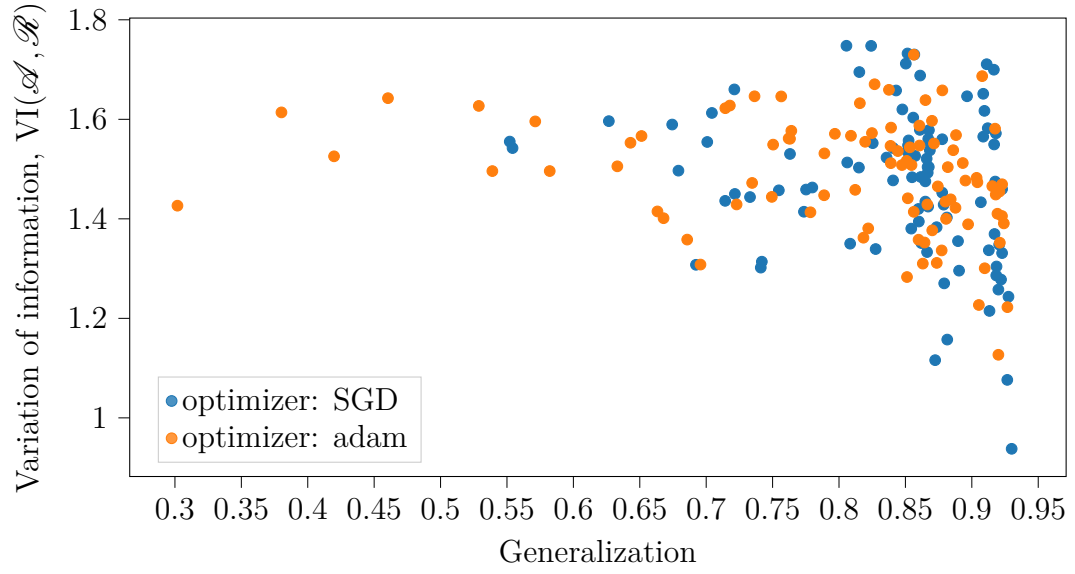


Figure B.47: Influence of optimizer on variation of information with internal representation using top-10 % magnitude weight components.

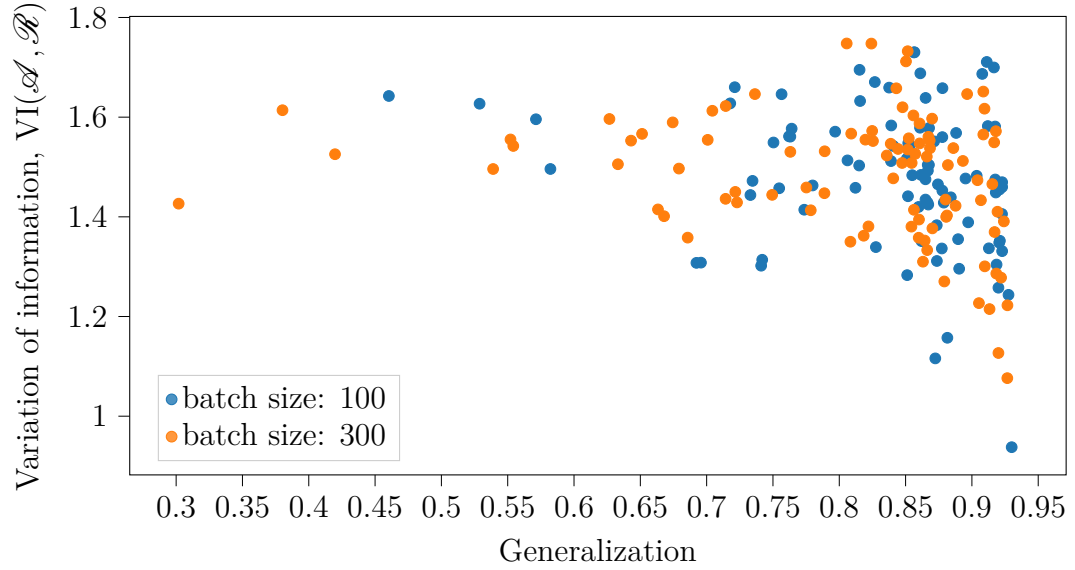


Figure B.48: Influence of batch size on variation of information with internal representation using top-10 % magnitude weight components.

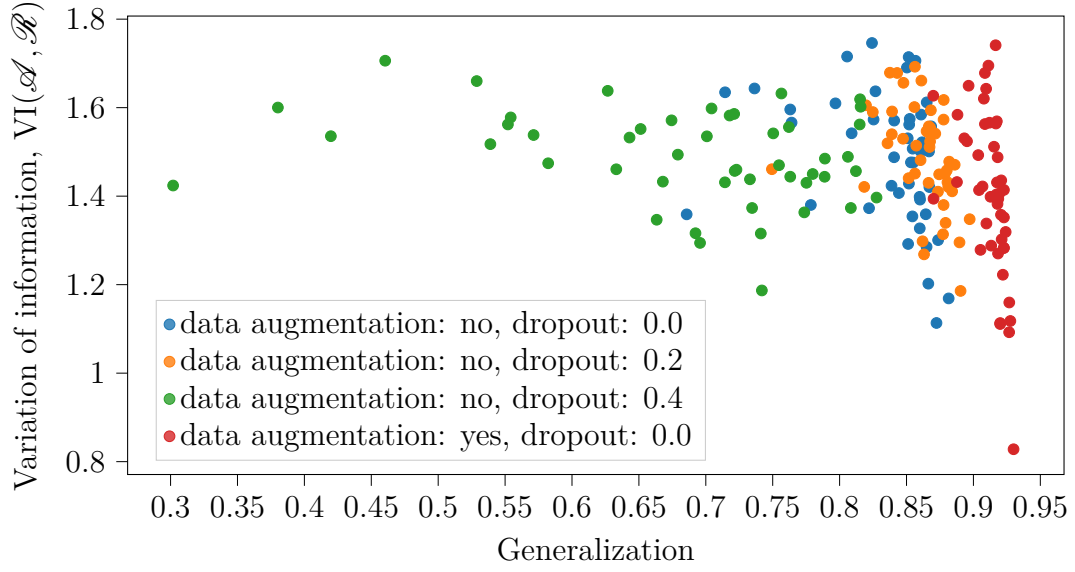


Figure B.49: Influence of noise regularization on variation of information with internal representation using top-5 % magnitude weight components.

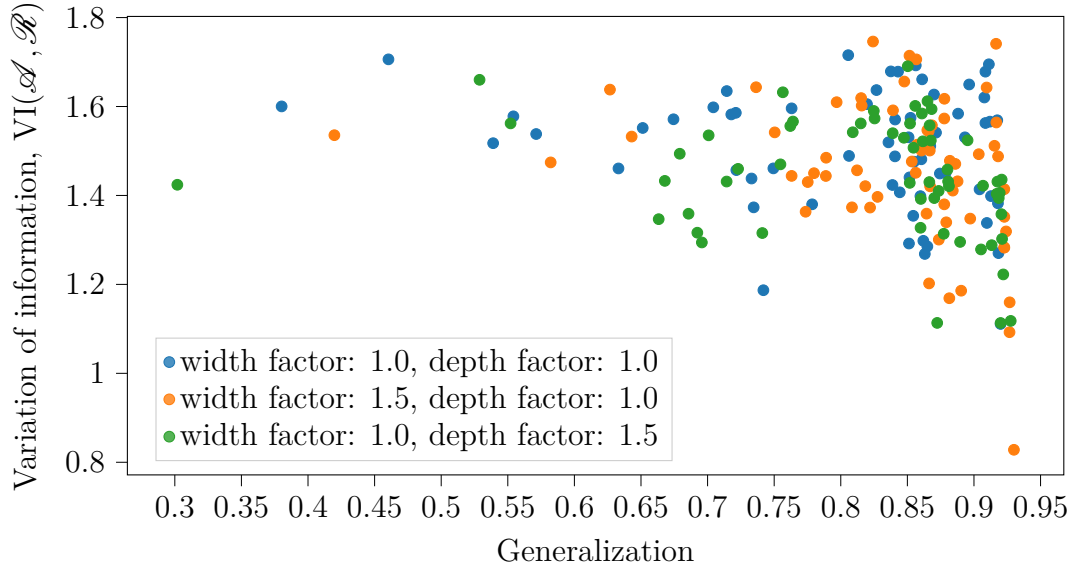


Figure B.50: Influence of architecture on variation of information with internal representation using top-5 % magnitude weight components.

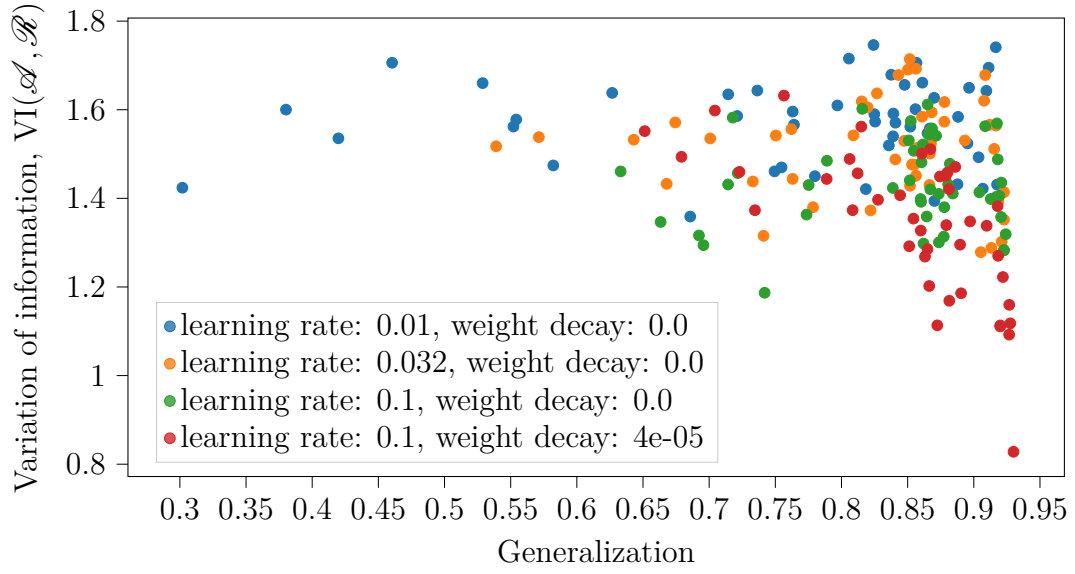


Figure B.51: Influence of effective learning rate on variation of information with internal representation using top-5 % magnitude weight components.

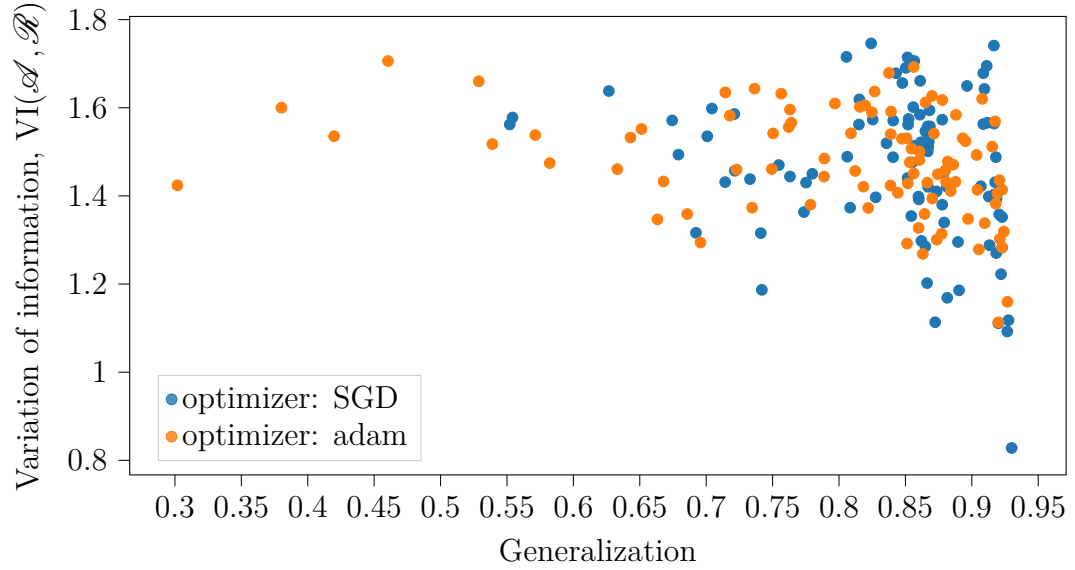


Figure B.52: Influence of optimizer on variation of information with internal representation using top-5 % magnitude weight components.

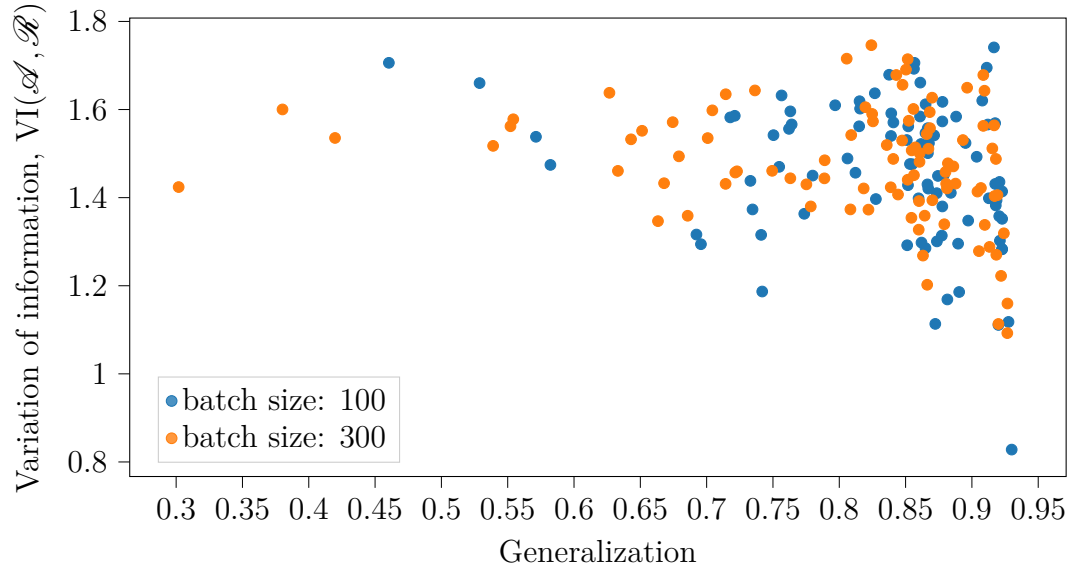


Figure B.53: Influence of batch size on variation of information with internal representation using top-5 % magnitude weight components.

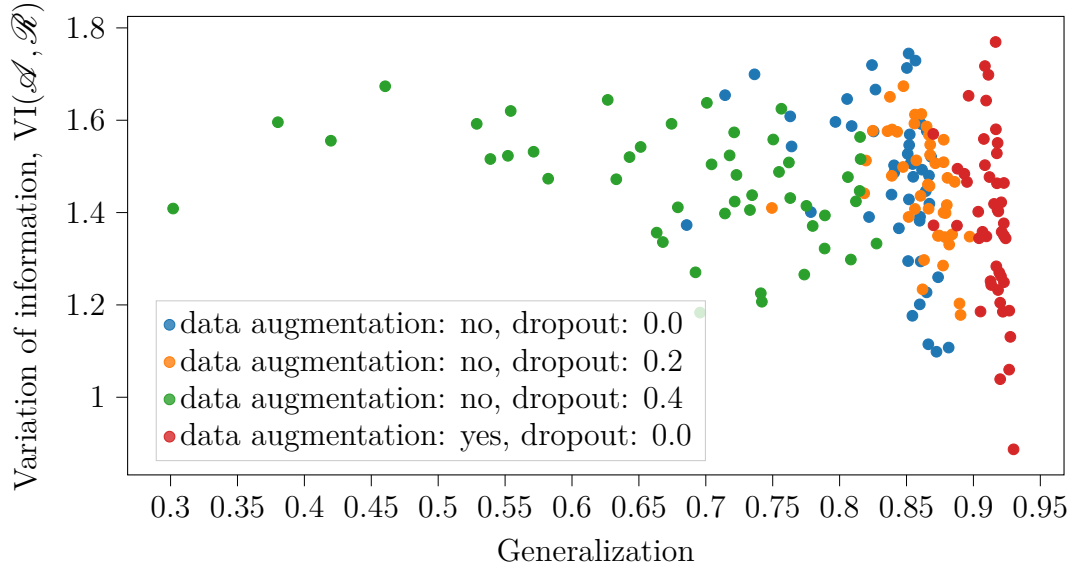


Figure B.54: Influence of noise regularization on variation of information with internal representation using top-2 % magnitude weight components.

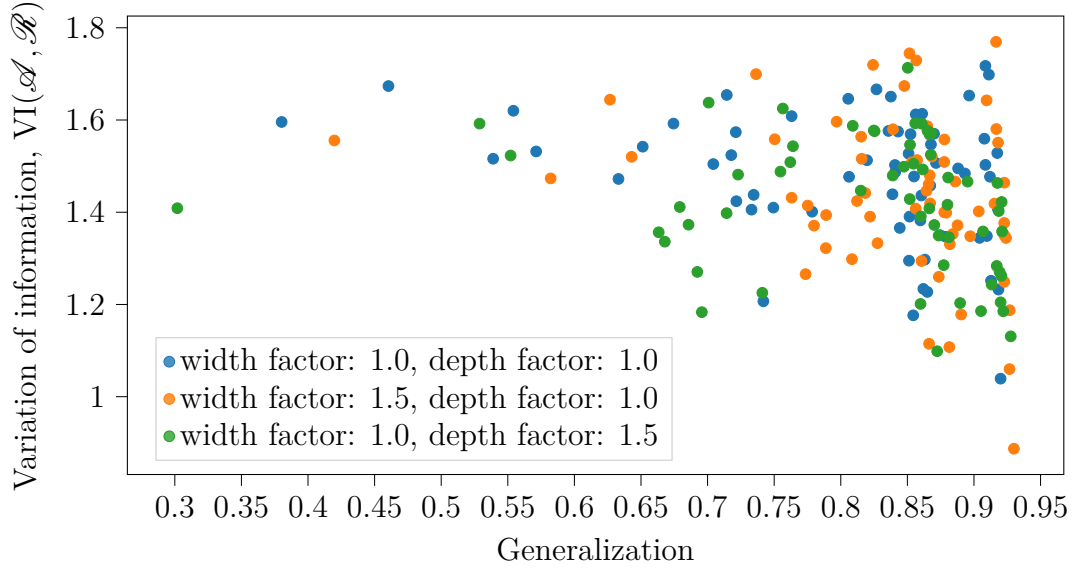


Figure B.55: Influence of architecture on variation of information with internal representation using top-2% magnitude weight components.

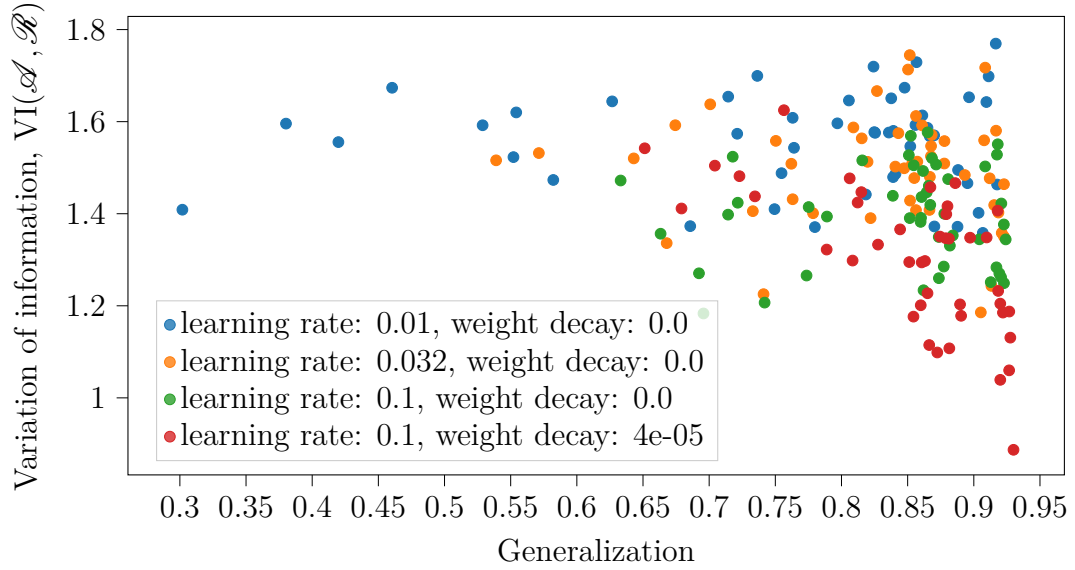


Figure B.56: Influence of effective learning rate on variation of information with internal representation using top-2% magnitude weight components.

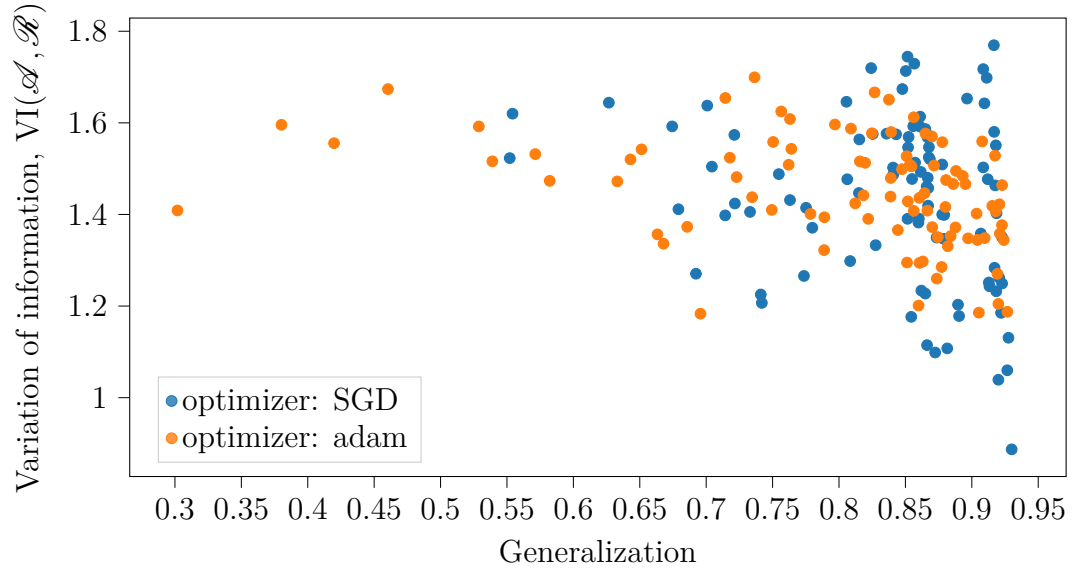


Figure B.57: Influence of optimizer on variation of information with internal representation using top-2% magnitude weight components.

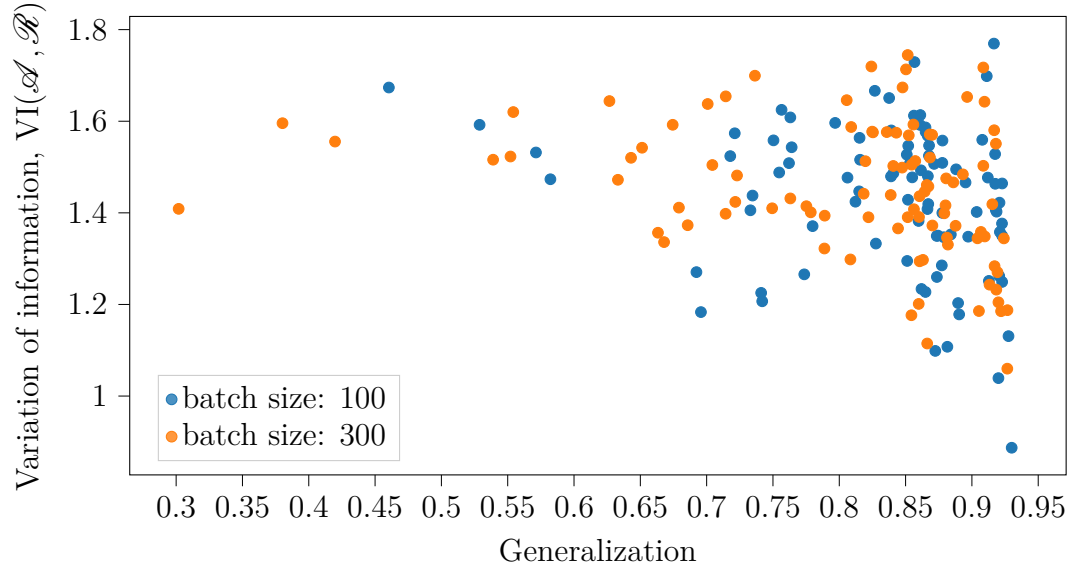


Figure B.58: Influence of batch size on variation of information with internal representation using top-2% magnitude weight components.

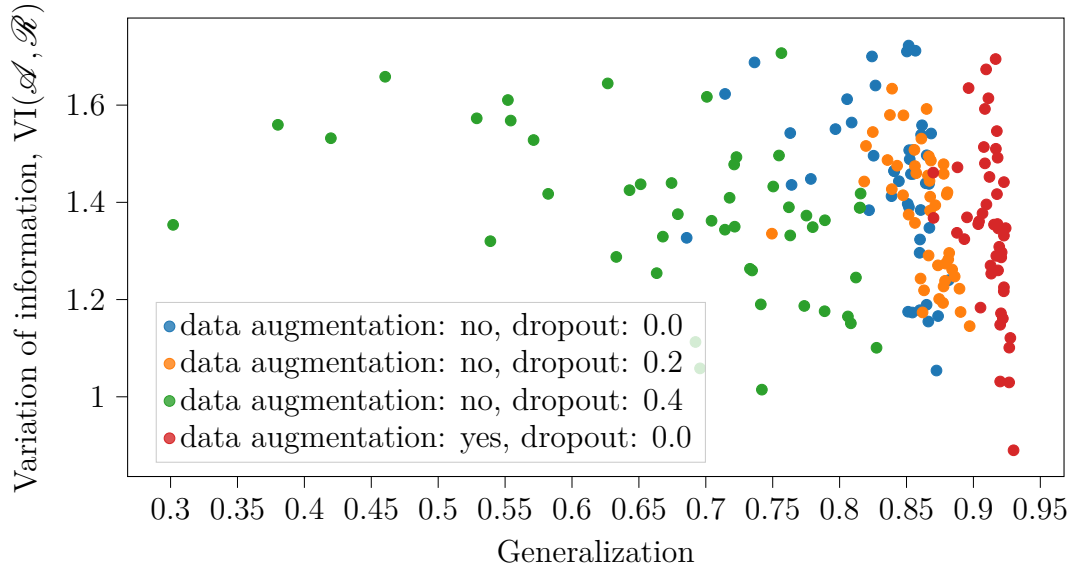


Figure B.59: Influence of noise regularization on variation of information with internal representation using top-0.5 % magnitude weight components.

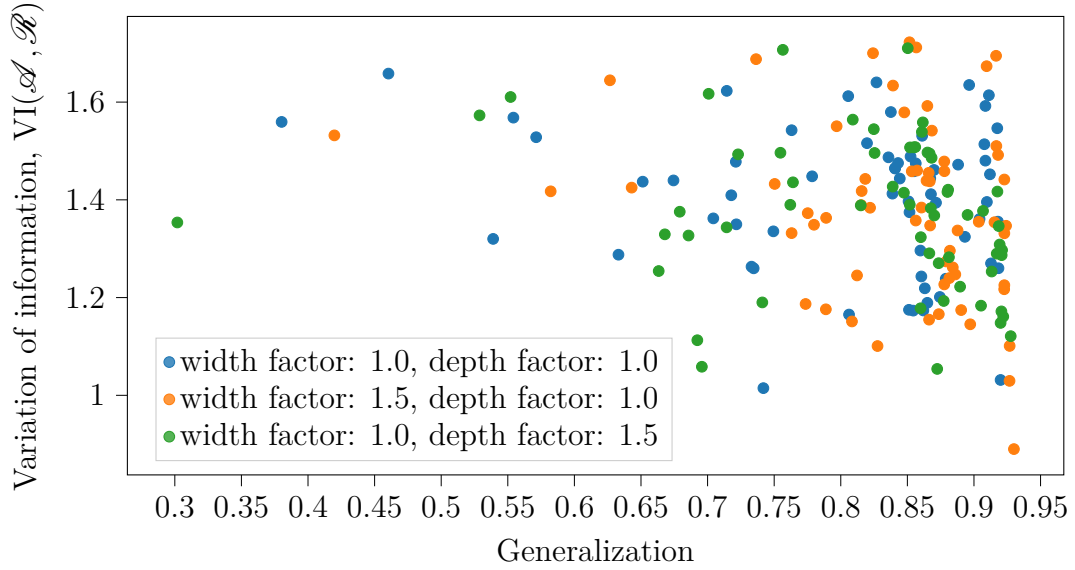


Figure B.60: Influence of architecture on variation of information with internal representation using top-0.5 % magnitude weight components.

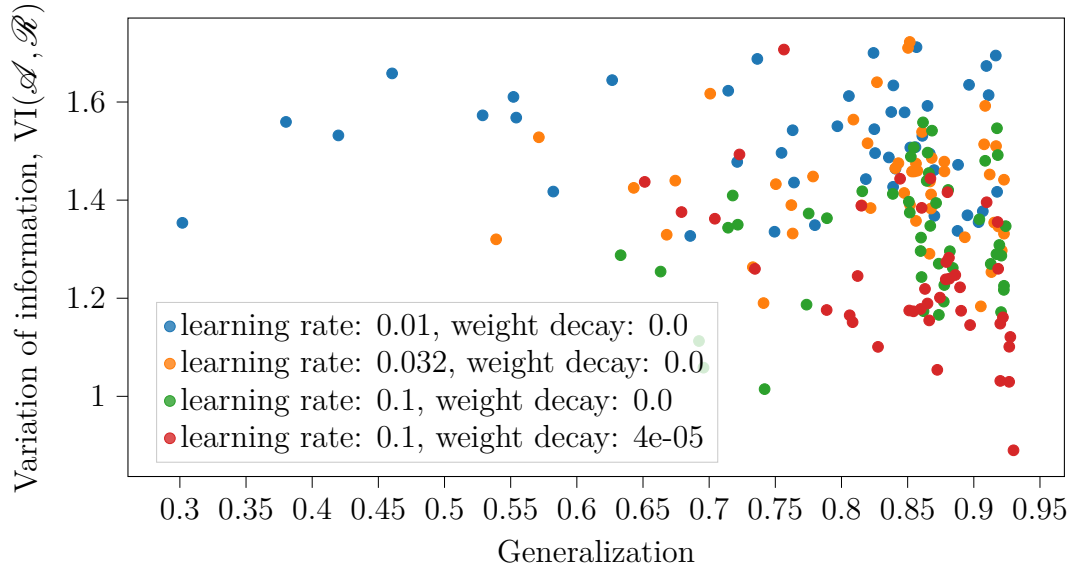


Figure B.61: Influence of effective learning rate on variation of information with internal representation using top-0.5 % magnitude weight components.

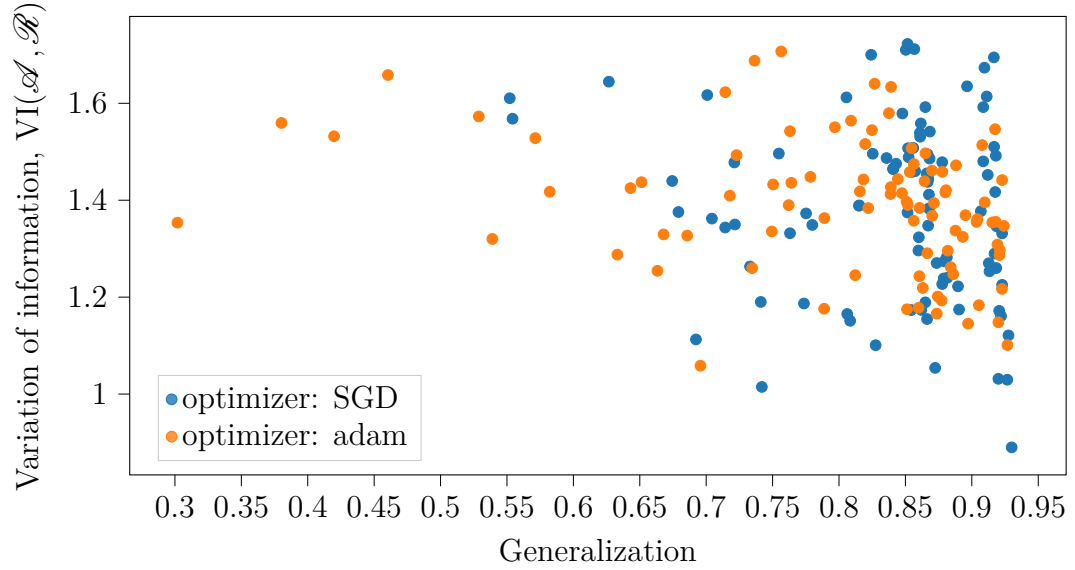


Figure B.62: Influence of optimizer on variation of information with internal representation using top-0.5 % magnitude weight components.

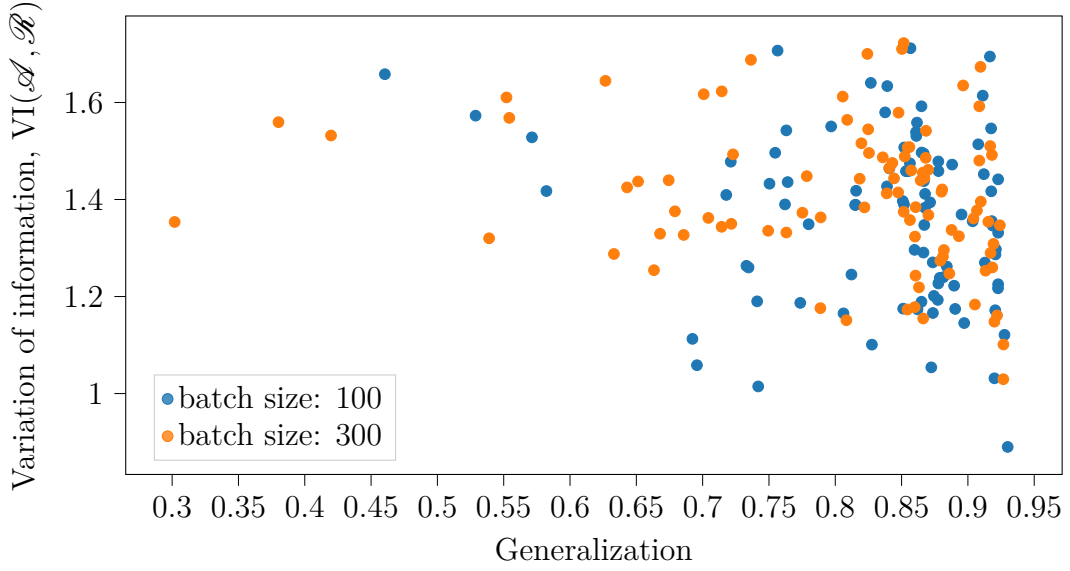


Figure B.63: Influence of batch size on variation of information with internal representation using top-0.5 % magnitude weight components.

B.1.3 As Measured Using Cluster Size

B.1.3.1 Relation with Generalization

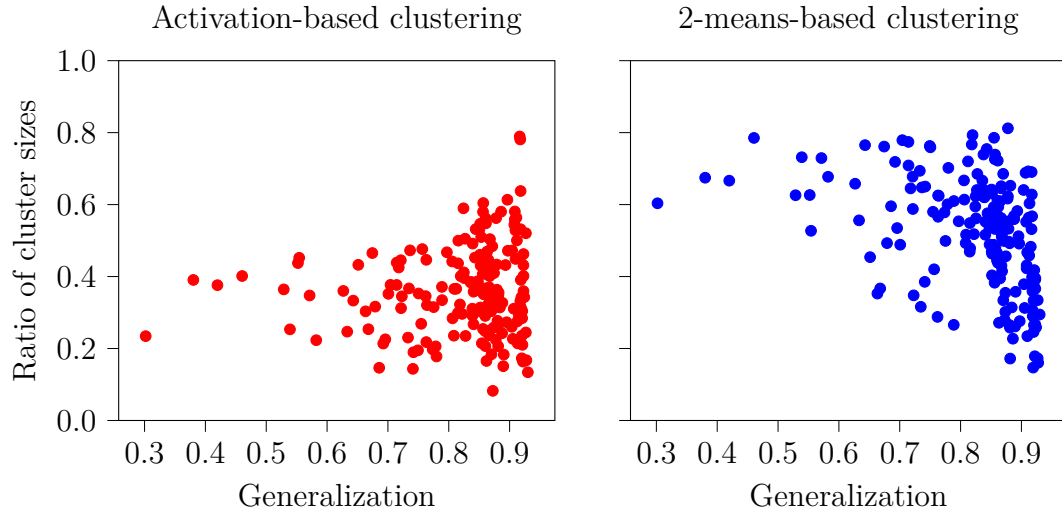


Figure B.64: Comparison of cluster size of activation partition and 2-means clustering with internal representation using all weight components.

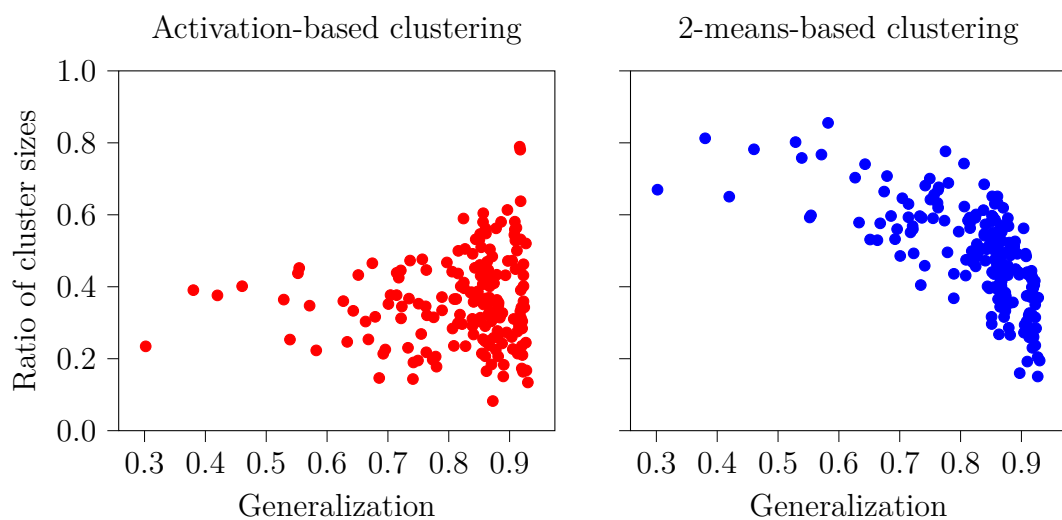


Figure B.65: Comparison of cluster size of activation partition and 2-means clustering with internal representation using top-10 % magnitude weight components.

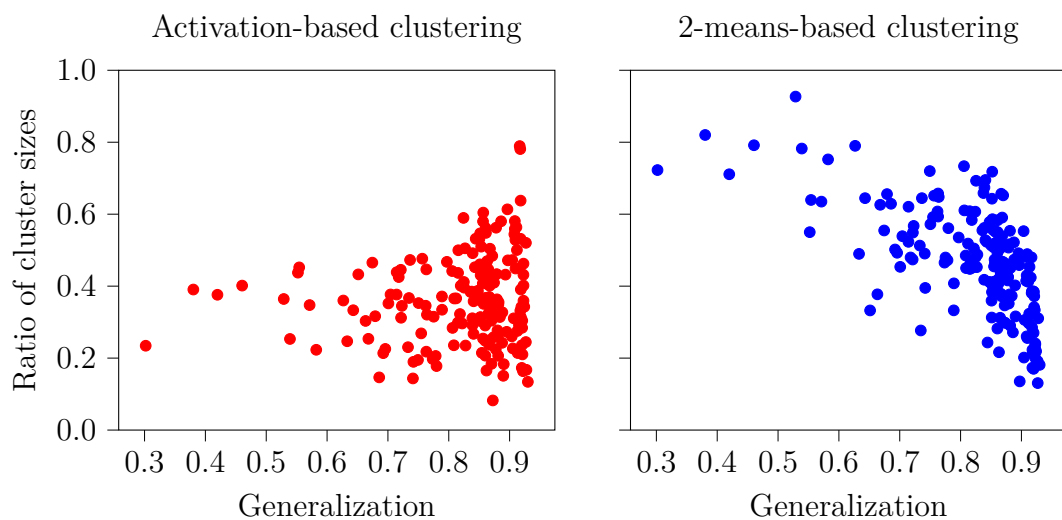


Figure B.66: Comparison of cluster size of activation partition and 2-means clustering with internal representation using top-5 % magnitude weight components.

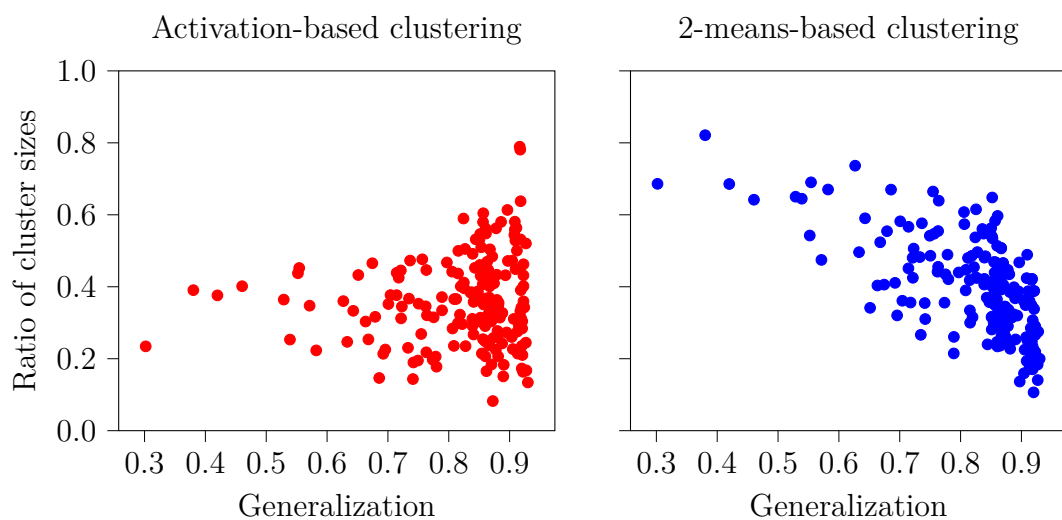


Figure B.67: Comparison of cluster size of activation partition and 2-means clustering with internal representation using top-2% magnitude weight components.

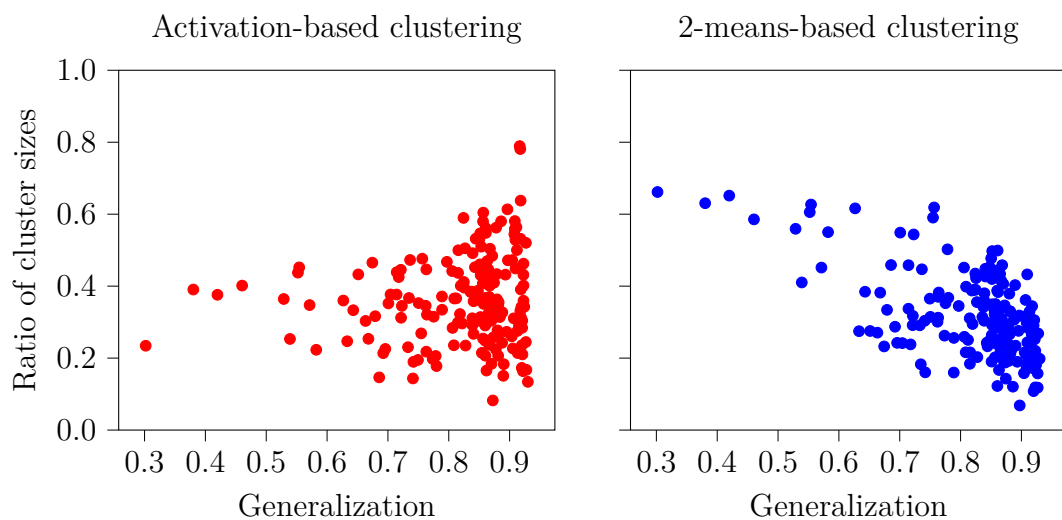


Figure B.68: Comparison of cluster size of activation partition and 2-means clustering with internal representation using top-0.5% magnitude weight components.

Bibliography

- [1] Simon Carbonnelle and Christophe De Vleeschouwer. Intraclass clustering: an implicit learning ability that regularizes dnns. *ICLR*, abs/2103.06733, 2021.
- [2] Behnam Neyshabur, Srinadh Bhojanapalli, David McAllester, and Nati Srebro. Exploring generalization in deep learning. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5947–5956, 2017.
- [3] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.
- [4] Yiding Jiang, Pierre Foret, Scott Yak, Daniel M. Roy, Hossein Mobahi, Gintare Karolina Dziugaite, Samy Bengio, Suriya Gunasekar, Isabelle Guyon, and Behnam Neyshabur. Neurips 2020 competition: Predicting generalization in deep learning. *CoRR*, abs/2012.07976, 2020.
- [5] Simon Carbonnelle and Christophe De Vleeschouwer. Layer rotation: a surprisingly powerful indicator of generalization in deep networks? *ICML*, abs/1806.01603, 2019.
- [6] Simon Carbonnelle and Christophe De Vleeschouwer. Experimental study of the neuron-level mechanisms emerging from backpropagation. In *27th European Symposium on Artificial Neural Networks, ESANN 2019, Bruges, Belgium, April 24-26, 2019*, 2019.
- [7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.

- [8] David H. Wolpert. The lack of a priori distinctions between learning algorithms. *Neural Computation*, 8(7):1341–1390, 1996.
- [9] D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.
- [10] Kevin Jarrett, Koray Kavukcuoglu, Marc’Aurelio Ranzato, and Yann LeCun. What is the best multi-stage architecture for object recognition? In *2009 IEEE 12th International Conference on Computer Vision*, pages 2146–2153, 2009.
- [11] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML’10, page 807–814, Madison, WI, USA, 2010. Omnipress.
- [12] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.
- [13] Jun Han and Claudio Moraga. The influence of the sigmoid function parameters on the speed of backpropagation learning. In José Mira and Francisco Sandoval, editors, *From Natural to Artificial Neural Computation*, pages 195–201, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.
- [14] Tom M. Mitchell. *Machine Learning*. WCB–McGraw–Hill, 1997. <http://www.cs.cmu.edu/~tom/mlbook.html>.
- [15] John S. Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In Françoise Fogelman Soulié and Jeanny Hérault, editors, *Neurocomputing*, pages 227–236, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.
- [16] Bolin Gao and Lacra Pavel. On the Properties of the Softmax Function with Application in Game Theory and Reinforcement Learning. *arXiv e-prints*, page arXiv:1704.00805, April 2017.
- [17] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.
- [18] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, Dec 1989.

- [19] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Netw.*, 4(2):251–257, March 1991.
- [20] Moshe Leshno, Vladimir Ya. Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861–867, 1993.
- [21] Allan Pinkus. Approximation theory of the mlp model in neural networks. *Acta Numerica*, 8:143–195, 1999.
- [22] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, USA, 2nd edition, 1998.
- [23] Anastasis Kratsios. The universal approximation property. *Annals of Mathematics and Artificial Intelligence*, Jan 2021.
- [24] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. On calibration of modern neural networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1321–1330. PMLR, 06–11 Aug 2017.
- [25] P. Marcotte and G. Savard. Novel approaches to the discrimination problem. *Zeitschrift für Operations Research*, 36(6):517–545, Nov 1992.
- [26] Christophe De Vleeschouwer. LELEC2885: Lecture notes in Image Processing and Computer Vision. November 2020.
- [27] Kevin Patrick Murphy. *Machine Learning: a Probabilistic Perspective*. MIT Press, 2012. <https://probml.github.io/pml-book/book0.html>.
- [28] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, October 1986.
- [29] D. Randall Wilson and Tony R. Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Netw.*, 16(10):1429–1451, December 2003.
- [30] S. Sra, S. Nowozin, and S.J. Wright. *Optimization for Machine Learning*. Neural information processing series. MIT Press, 2012.
- [31] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning:

- Generalization gap and sharp minima. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.
- [32] Guillaume Leclerc and Aleksander Madry. The two regimes of deep network training. *CoRR*, abs/2002.10376, 2020.
 - [33] Léon Bottou. *Stochastic Learning*, pages 146–168. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.
 - [34] Krzysztof C. Kiwiel. Convergence and efficiency of subgradient methods for quasiconvex minimization. *Mathematical Programming*, 90(1):1–25, Mar 2001.
 - [35] Stanislaw Jastrzebski, Zachary Kenton, Devansh Arpit, Nicolas Ballas, Asja Fischer, Yoshua Bengio, and Amos J. Storkey. Three factors influencing minima in SGD. *CoRR*, abs/1711.04623, 2017.
 - [36] Shengchao Liu, Dimitris S. Papailiopoulos, and Dimitris Achlioptas. Bad global minima exist and SGD can reach them. In Hugo Larochelle, Marc’Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.
 - [37] B.T. Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.
 - [38] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1139–1147, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.
 - [39] Yu. E. Nesterov. A method of solving a convex programming problem with convergence rate $O\left(\frac{1}{k^2}\right)$. *Dokl. Akad. Nauk SSSR*, 269:543–547, 1983.
 - [40] Yu. E. Nesterov. *Lectures on Convex Optimization*. Springer Optimization and Its Applications. Springer International Press, 2018.
 - [41] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

- [42] Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nati Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 4148–4158, 2017.
- [43] Devansh Arpit, Stanisław Jastrzundefinedbski, Nicolas Ballas, David Krueger, Emmanuel Bengio, Maxinder S. Kanwal, Tegan Maharaj, Asja Fischer, Aaron Courville, Yoshua Bengio, and Simon Lacoste-Julien. A closer look at memorization in deep networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML’17, page 233–242. JMLR.org, 2017.
- [44] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterton, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.
- [45] Andrew M. Saxe, James L. McClelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks, 2014.
- [46] David Sussillo and L. F. Abbott. Random walk initialization for training very deep feedforward networks, 2015.
- [47] Dmytro Mishkin and Jiri Matas. All you need is a good init. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.
- [48] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996.
- [49] Jocelyn Sietsma and Robert J. F. Dow. Creating artificial neural networks that generalize. *Neural Netw.*, 4(1):67–79, January 1991.
- [50] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning*, ICML ’08, page 1096–1103, New York, NY, USA, 2008. Association for Computing Machinery.

- [51] Ben Poole, Jascha Sohl-Dickstein, and Surya Ganguli. Analyzing noise in autoencoders and deep networks, 2014.
- [52] Yichuan Tang and Chris Eliasmith. Deep networks for robust visual recognition. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML’10, page 1055–1062, Madison, WI, USA, 2010. Omnipress.
- [53] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML’15, page 448–456. JMLR.org, 2015.
- [54] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.
- [55] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, Aug 1996.
- [56] David Warde-Farley, Ian J. Goodfellow, Aaron Courville, and Yoshua Bengio. An empirical analysis of dropout in piecewise linear networks, 2014.
- [57] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors, 2012.
- [58] Ian Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1319–1327, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.
- [59] Yarin Gal and Zoubin Ghahramani. Bayesian convolutional neural networks with bernoulli approximate variational inference, 2016.
- [60] Christopher Bishop. Regularization and complexity control in feed-forward networks. In *Proceedings International Conference on Artificial Neural Networks ICANN’95*, volume 1, pages 141–148. EC2 et Cie, January 1995.
- [61] J. Sjöberg and L. Ljung. Overtraining, regularization and searching for a minimum, with application to neural networks. *International Journal of Control*, 62(6):1391–1407, 1995.

- [62] Y. LeCun. Generalization and network design strategies. In R. Pfeifer, Z. Schreter, F. Fogelman, and L. Steels, editors, *Connectionism in Perspective*, Zurich, Switzerland, 1989. Elsevier. an extended version was published as a technical report of the University of Toronto.
- [63] Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2013.
- [64] Y. Zhou and R. Chellappa. Computation of optical flow using a neural network. *IEEE 1988 International Conference on Neural Networks*, pages 71–78 vol.2, 1988.
- [65] Y-Lan Boureau, J. Ponce, and Y. LeCun. A theoretical analysis of feature pooling in visual recognition. In *ICML*, 2010.
- [66] Y-Lan Boureau, Nicolas Le Roux, Francis Bach, Jean Ponce, and Yann Lecun. Ask the locals: multi-way local pooling for image recognition. In *ICCV'11 - The 13th International Conference on Computer Vision*, Barcelone, Spain, November 2011.
- [67] Yangqing Jia, Chang Huang, and T. Darrell. Beyond spatial pyramids: Receptive field learning for pooled image features. pages 3370–3377, 06 2012.
- [68] Andrew Gelman, John B. Carlin, Hal S. Stern, and Donald B. Rubin. *Bayesian Data Analysis*. Chapman and Hall/CRC, 3rd ed. edition, 2013.
- [69] Yann LeCun. Learning process in an asymmetric threshold network. In E. Bienenstock, F. Fogelman Soulié, and G. Weisbuch, editors, *Disordered Systems and Biological Organization*, pages 233–240, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.
- [70] Karo Gregor and Yann LeCun. Emergence of complex-like cells in a temporal product network with local receptive fields, 2010.
- [71] Jiquan Ngiam, Zhenghao Chen, Daniel Chia, Pang Koh, Quoc Le, and Andrew Ng. Tiled convolutional neural networks. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010.
- [72] Vladimir A. Kazeev, Boris N. Khoromskij, and Eugene E. Tyrtysnikov. Multilevel toeplitz matrices generated by tensor-structured vectors and convolution with logarithmic complexity. *SIAM Journal on Scientific Computing*, 35(3):A1511–A1536, 2013.

- [73] Ian J. Goodfellow. Technical report: Multidimensional, downsampled convolution for autoencoders. Technical report, Université de Montréal, 2010.
- [74] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning (still) requires rethinking generalization. *Commun. ACM*, 64(3):107–115, 2021.
- [75] Haowei He, Gao Huang, and Yang Yuan. Asymmetric valleys: Beyond sharp and flat local minima. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d’Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 2549–2560, 2019.
- [76] Behrooz Ghorbani, Shankar Krishnan, and Ying Xiao. An investigation into neural net optimization via hessian eigenvalue density. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 2232–2241. PMLR, 2019.
- [77] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.
- [78] Iro Laina, Ruth Fong, and Andrea Vedaldi. Quantifying learnability and describability of visual concepts emerging in representation learning. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 13112–13126. Curran Associates, Inc., 2020.
- [79] Chris Olah, Nick Cammarata, Ludwig Schubert, Gabriel Goh, Michael Petrov, and Shan Carter. Zoom in: An introduction to circuits. *Distill*, 2020. <https://distill.pub/2020/circuits/zoom-in>.
- [80] Anh Mai Nguyen, Jason Yosinski, and Jeff Clune. Multifaceted feature visualization: Uncovering the different types of features learned by each neuron in deep neural networks. *CoRR*, abs/1602.03616, 2016.
- [81] Anh Nguyen, Jason Yosinski, and Jeff Clune. Understanding neural networks via feature visualization: A survey. *CoRR*, abs/1904.08939, 2019.

- [82] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In David J. Fleet, Tomás Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I*, volume 8689 of *Lecture Notes in Computer Science*, pages 818–833. Springer, 2014.
- [83] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Workshop Track Proceedings*, 2014.
- [84] Jason Yosinski, Jeff Clune, Anh Mai Nguyen, Thomas J. Fuchs, and Hod Lipson. Understanding neural networks through deep visualization. *CoRR*, abs/1506.06579, 2015.
- [85] Jesse Mu and Jacob Andreas. Compositional explanations of neurons. In Hugo Larochelle, Marc’Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.
- [86] Yiding Jiang, Behnam Neyshabur, Hossein Mobahi, Dilip Krishnan, and Samy Bengio. Fantastic generalization measures and where to find them. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.
- [87] Alex Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto*, 05 2012.
- [88] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.
- [89] Haris Iqbal. Harisiqbal88/plotneuralnet v1.0.0, December 2018.
- [90] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.
- [91] François Chollet et al. Keras. <https://keras.io>, 2015.

- [92] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.
- [93] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.
- [94] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [95] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.
- [96] Nico Schlömer, Sandro Braun, Mads Adrian, danielhkl, Armin Wehrfritz, Samo Penic, Eric Wieser, Aikhjarto, Holger Berndt, Spyros Stathopoulos, Andrew Spott, Daniel Edler, Lukas Müller, Christoph Boeddeker, Gaëtan Cassiers, Philipp A.; satej soman, Marco Rossi, Douglas Dawson, Moritz Lipp, harenbrs, Tony Mason, senselessDev, hgwd2, pwohlhart, edmundus, mrtnschltr, ngc92, Alexey Kuzmin, and David Haberthür. tikzplotlib v0.9.8, February 2021.
- [97] Peter H. Westfall. Kurtosis as peakedness, 1905 - 2014. r.i.p. *The American statistician*, 68(3):191–195, 2014. 25678714[pmid].
- [98] S. Kokoska and D. Zwillinger. Crc standard probability and statistics tables and formulae, student edition. 1999.

- [99] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987.
- [100] Leonard Kaufman and Peter J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley Series in Probability and Statistics. Wiley, 2005.
- [101] Marina Meilă. Comparing clusterings—an information based distance. *Journal of Multivariate Analysis*, 98(5):873–895, 2007.
- [102] J. MacQueen. Some methods for classification and analysis of multivariate observations. Proc. 5th Berkeley Symp. Math. Stat. Probab., Univ. Calif. 1965/66, 1, 281-297 (1967)., 1967.
- [103] Hugo Steinhaus. Sur la division des corps matériels en parties. *Bull. Acad. Pol. Sci., Cl. III*, 4:801–804, 1957.
- [104] S. Lloyd. Least square quantization in pcm. *Bell Telephone Laboratories Paper*, 1957.
- [105] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.
- [106] E. Forgy. Cluster analysis of multivariate data: efficiency versus interpretability of classifications. *Biometrics*, 21:768–780, 1965.
- [107] Pravendra Singh, Vinay Kumar Verma, Piyush Rai, and Vinay P. Namboodiri. Play and prune: Adaptive filter pruning for deep model compression. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 3460–3466. International Joint Conferences on Artificial Intelligence Organization, 7 2019.
- [108] Woojeong Kim, Suhyun Kim, Mincheol Park, and Geunseok Jeon. Neuron merging: Compensating for pruned neurons. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 585–595. Curran Associates, Inc., 2020.
- [109] Hattie Zhou, Janice Lan, Rosanne Liu, and Jason Yosinski. Deconstructing lottery tickets: Zeros, signs, and the supermask. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d’Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 3592–3602, 2019.

- [110] Nguyen Xuan Vinh, Julien Epps, and James Bailey. Information theoretic measures for clusterings comparison: Is a correction for chance necessary? In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, page 1073–1080, New York, NY, USA, 2009. Association for Computing Machinery.
- [111] Nanxuan Zhao, Zhirong Wu, Rynson W.H. Lau, and Stephen Lin. What makes instance discrimination good for transfer learning. *ICLR*, 2021.
- [112] Kazuaki Hanawa, Sho Yokoi, Satoshi Hara, and Kentaro Inui. Evaluation of similarity-based explanations. *ICLR*, 2021.
- [113] Roger Granda, Tinne Tuytelaars, and José Oramas M. Can the state of relevant neurons in a deep neural networks serve as indicators for detecting adversarial attacks? *CoRR*, abs/2010.15974, 2020.
- [114] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deep-fool: A simple and accurate method to fool deep neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2574–2582, 2016.
- [115] Jie Ren, Peter J. Liu, Emily Fertig, Jasper Snoek, Ryan Poplin, Mark A. DePristo, Joshua V. Dillon, and Balaji Lakshminarayanan. Likelihood ratios for out-of-distribution detection. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d’Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 14680–14691, 2019.
- [116] Joan Serra, David Álvarez, Vicenç Gómez, Olga Slizovskaia, José F. Núñez, and Jordi Luque. Input complexity and out-of-distribution detection with likelihood-based generative models. *CoRR*, abs/1909.11480, 2019.
- [117] Vaishnavh Nagarajan, Anders Andreassen, and Behnam Neyshabur. Understanding the failure modes of out-of-distribution generalization. *CoRR*, abs/2010.15775, 2020.
- [118] Martín Arjovsky. Out of distribution generalization in machine learning. *CoRR*, abs/2103.02667, 2021.
- [119] Stanislav Fort, Jie Ren, and Balaji Lakshminarayanan. Exploring the limits of out-of-distribution detection. *CoRR*, abs/2106.03004, 2021.

- [120] Jonathan Frankle, David J. Schwab, and Ari S. Morcos. The early phase of neural network training. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. Open-Review.net, 2020.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl