FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Petr Houška

Deep-learning architectures for analysing population neural data

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Ján Antolík, Ph.D.
Study programme: Computer Science
Study branch: Artificial Intelligence

Prague 2021

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

First and foremost, I would like to thank my supervisor Mgr. Ján Antolík, Ph.D. for inviting me to a field that was entirely foreign to me at the beginning, providing thorough and ever-friendly guidance along the (not so short) journey. On a similar note, I want to express my deep gratitude to Prof. Daniel A. Butts for his invaluable and incredibly welcoming consultations regarding both the NDN3 library and the field of computational neuroscience in general.

Although I did not end up doing my thesis under his supervision, RNDr. Milan Straka, Ph.D. deserves recognition for patiently lending me his valuable time while I was trying to figure out my eventual thesis topic.

Last but not least, I want to acknowledge at least some of the people who were, albeit indirectly, instrumental to me finishing this thesis. Be it through friendly nudging, emotional support, or even just being explicitly busy with their work and thus shaming me out of procrastination. In atmospherically random order, thank you: Václav & Marcela Houškovi, Petra Millarová, Tomáš Thon, Eliška Kopecká, Ondřej Trhoň, and Adéla Syrová.

Title: Deep-learning architectures for analysing population neural data

Author: Petr Houška

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Ján Antolík, Ph.D., Department of Software and Computer Science Education

Abstract: Accurate models of visual system are key for understanding how our brains process visual information. In recent years, deep neural networks (DNN) have been rapidly gaining traction in this domain. However, only few studies attempted to incorporate known anatomical properties of visual system into standard DNN architectures adapted from the general machine learning field, to improve their interpretability and performance on visual data.

In this thesis, we optimize a recent biologically inspired deep learning architecture designed for analysis of population data recorded from mammalian primary visual cortex when presented with natural images as stimuli. We reimplement this prior modeling in existing neuroscience focused deep learning framework NDN3 and assess it in terms of stability and sensitivity to hyperparameters and architecture fine-tuning. We proceed to extend the model with components of various DNN models, analysing novel combinations and techniques from classical computer vision deep learning, comparing their effectiveness against the bio-inspired components.

We were able to identify modifications that greatly increase the stability of the model while securing moderate improvement in overall performance. Furthermore, we document the importance of small hyperparameters adjustments versus architectural advantages that could facilitate further experiments with the examined architectures. All-new model components were contributed to the open-source NDN3 package.

Overall, this work grounds previous bio-inspired DNN architectures in the modern NDN3 environment, identifying optimal hyper-parametrization of the model, and thus paving path towards future development of these bio-inspired architectures.

Keywords: deep-learning, computational neuroscience, v1 modeling, bio-inspired architectures, visual computation

Contents

Introduction **4**
History & context . . . 4
Motivation . . . 5
Goals . . . 5
Thesis structure . . . 6

1 Background **8**
1.1 Visual sensory neuroscience . . . 8
1.1.1 Early visual system . . . 8
1.2 Computational neuroscience . . . 10
1.2.1 General overview . . . 10
1.2.2 Classical models . . . 11
1.3 ML and Deep neural networks . . . 12
1.3.1 Feed forward neural network . . . 12
1.3.2 DNN training . . . 14
1.3.3 Transfer learning . . . 14
1.4 DNNs and computational neuroscience . . . 15
1.4.1 Poisson versus Gaussian loss function . . . 15
1.4.2 Correlation as performance metric . . . 15
1.4.3 (Soft) Regularizations . . . 16

2 Key and related works **17**
2.1 Antolík et al. and the DoG layer . . . 17
2.1.1 DoG layer . . . 17
2.1.2 V1 neurons . . . 18
2.1.3 Training regime . . . 18
2.2 Klindt et al. and the separable layer . . . 19
2.2.1 Architecture . . . 19
2.3 Related works . . . 20
2.3.1 Neural convolutional models . . . 20
2.3.2 Transfer learning . . . 20

3 Implementation **22**
3.1 Overview of NDN3 library . . . 22
3.1.1 NDN3 toolkit . . . 24
3.2 Implemented NDN3 extensions . . . 26
3.2.1 DoG layer & convolutional variant . . . 26
3.2.2 Pearson's correlation tracking . . . 27
3.3 Experiments and analysis pipeline . . . 27
3.3.1 Experiment execution . . . 27
3.3.2 Experiment analysis . . . 28

4 Methodology **30**
4.1 The system identification task . . . 30
4.1.1 Dataset . . . 30
4.1.2 Prior works results . . . 31
4.2 Experiments . . . 32
4.3 Analysis . . . 34
4.4 Training regime . . . 34

5 Experiments and results **36**
5.1 Assessment and improvements of HSM . . . 36
5.1.0 Differences introduced by reimplementation . . . 36
5.1.1 Reimplementation . . . 37
1.1.1 Initial reimplementation: Baseline 1 . . . 37
5.1.2 Training hyperparameters tuning . . . 38
1.2.1 Input scaling: Baseline 2 . . . 38
1.2.2 Gaussian versus Poisson . . . 39
1.2.3 Bias initialization . . . 39
1.2.4 Learning rate . . . 40
1.2.5 Learning rate and bias initialization: Baseline 3 . 41
1.2.6 Batch size . . . 42
5.1.3 Regularizations and non-linearity . . . 43
1.3.1 Dropout . . . 43
1.3.2 L1 and L2 regularizations . . . 44
1.3.3 Separate L2 regularizations: Baseline 4 . . . 44
1.3.4 DoG layer non-linearity . . . 45
5.1.4 Input scaling . . . 45
1.4.1 Implicit and explicit input scaling . . . 45
5.1.5 Discussion . . . 47
5.2 Variations on the filter and hidden layers . . . 48
5.2.1 Fully connected models . . . 49
2.1.1 LNLN model . . . 49
2.1.2 LN model . . . 50
5.2.2 Convolutional and separable models . . . 51
2.2.1 Convolutions instead of DoG . . . 51
2.2.2 Convolutions and separable . . . 52
2.2.3 Convolutional DoG . . . 53
2.2.4 Convolutional DoG without non-linearity . . . 55
2.2.5 Reimplementing what/where model . . . 56
5.2.3 Discussion . . . 57
5.3 All regions experiments . . . 59
5.3.1 Combined dataset . . . 59
3.1.1 Region pooled dataset . . . 59
5.3.2 Testing on other regions . . . 62
3.2.1 Various architectures across regions . . . 62
5.3.3 Discussion . . . 64

Conclusion **66**
Main findings 66
Lessons learned 67
Future work 67

Bibliography **69**

List of Figures **73**

List of Tables **74**

List of Abbreviations **75**

Glossary **76**

A Attachments **77**
A.1 Digital attachments 77
A.1.1 msc-neuro 77
A.1.2 NDN3 77

Introduction

History & context

According to the annals of history, the term *computational neuroscience* was coined by Eric L. Schwartz in 1985 when he organized a conference[1] to provide a summary of the current status of a field at that time known under a variety of names such as neural modeling, brain theory, and neural networks. The central idea of computational neuroscience, to study the brain through mathematical models, is much older than that, however. First modern traces can be found as early as in 1907 when Louis Lapicque introduced[2] the integrate and fire model. While crude, the model managed to describe observed interactions of a nerve fiber long before the exact mechanisms responsible for generation of neuron action potentials were discovered. Doing so, it greatly contributed to our understanding of the brain and laid a strong foundation for further research.

One of the main approaches in computational neuroscience goes under the name *system identification*: first, through experimentation one obtains a large dataset of input and output pairs of a given neural system – e.g. pairs of images and responses of neurons in primary visual cortex. These are subsequently fitted through machine learning methods with a model, in the hope of identifying the computations that the neural system does to translate the inputs to its outputs.

Such methods effectively enable us to predict the response of a system to an arbitrary plausible input. Doing so is one of the best ways to test our understanding of such a system. While having a model that predicts the response accurately does not necessarily have to mean our understanding of the biological fundamentals is correct, it is a clear signal it is at least plausible. On the other hand, the opposite, when a model based on our knowledge is not accurate, is proof we should revisit our assumptions. Such an approach is particularly effective in early stages of sensory systems whose primary function is to produce a response encoding to a clearly defined input – sensory stimulus.

Unsurprisingly, going as far back as to the 1960s, visual neuroscience has been close to traditional image processing and has used its toolkit for computational modeling. The influence was not one-sided. Several ideas, such as convolution layers, inspired by biological observations[3] have found their way back to classical computer vision. In recent years, the combination of advancements in deep learning and higher volume of better data has caused an increased focus on deep learning inspired neural models and, at the same time, a slow decline of classical system identification approaches.

Deep learning inspired models are only a rough approximation of biological reality, not only at the level of single neuron biophysics but also in terms of overall network architecture. For example, deep neural network (DNN) architectures typically do not model interactions between neurons within a single layer or do not account for direct nonlinear interactions between neurons, such as those provided by neuromodulation or attentional mechanisms in the biological brain.

[1] [Schwartz, 1990]
[2] [Abott, 1999]
[3] [Lindsay, 2020]

Regardless, classic DNNs are still immensely useful. Through the ability to fit their many parameters to real data, and access to ever increasing toolkit of high performance methods borrowed from classical machine learning, they allow for effective means of approximating the stimulus-response function in many neural subsystems. And thanks to their abstraction level, rapid experimentation of higher level concepts is also easier. Much like the integrate and fire model that also abstracted over certain details, they can still inform our understanding of the nature of vision processing.

Motivation

Despite the success of DNNs in neuroscience, having outclassed classical system identification methods in most domains, there remains a substantial gap between the predictions these models give and the actual measured neural responses. This is true even in the most peripheral stages of visual processing such as primary visual cortex. Furthermore, the poor interpretability of the DNN models has been identified as a major challenge limiting their usefulness for understanding neural computation.

To address these issues, a recent model by Antolík et al. [2016], which will be the focus of the present thesis, explored the idea of incorporating more biologically plausible components into the DNN framework. It showed that such an approach leads to a model with fewer parameters, better interpretability, and outperformed, at the time of writing, other state-of-the-art approaches.

However, the Antolík et al. 2016 study also showed shortcomings. The model was very sensitive to random initialization, most hyper-parameters of the model were not thoroughly studied, and more direct one-to-one comparison of the biologically inspired versus classical DNN components was missing. Furthermore, since its publishing, several studies using classical DNN techniques managed to demonstrate slight improvement over the Antolík et al. data. Finally, the model has been implemented in an ad-hoc way in now discontinued ML framework Theano, which poses a problem for further exploration of the bio-inspired DNNs architecture idea.

Goals

The goal of this thesis is to address the outlined outstanding issues with the Antolík et al. study, in order to provide a stable, fine-tuned, and well characterized implementation in a modern neuroscience oriented DNN framework to enable future experimentation with this bio-inspired architecture. Following contributions are the goals of this thesis:

- **Assess the Antolík et al. model (and its variants) in terms of stability and sensitivity to hyperparameter finetuning:** Especially on low quantities of noisy data, which is common in the field of sensory neuroscience, models tend to be relatively unstable with respect to both hyperparameters fine-tuning but also random initialization. We want to quantify the effects and hopefully explore ways to mitigate them to ensure

any conclusions drawn are not due to a chance but rather architectural properties.

- **Evaluate novel architectures inspired by classical computer vision:** We want to compare the benefits of biologically restricted techniques with more computationally generic approaches from classical deep computer vision and investigate whether hard constraints could be replaced with well chosen regularization schemes. Furthermore, test the impact of several deep learning techniques that were shown to work on classical computer vision problems.

- **Improve upon Antolík et al. model:** Decrease the gap between the original model and more recent less biologically constrained architectures that demonstrated improved performance.

- **Contribute new functionality to the NDN3 library toolbox:** We contribute all tools developed to conduct experiments in this thesis upstream to the NDN3 framework or, where not applicable, publish them separately under open source license. The goal is to enable others to iterate on our findings and evaluate the results in a shared environment.

- **(Re)implement and assess the Antolík et al. model and several of its variants within the NDN3 framework:** Similar to the previous point, implementing and analysing various models within a shared framework aims to enable rapid prototyping, and generally facilitate further research.

- **Identify opportunities for further research:** Since our set of goals is relatively broad, we will not be able to dive deep into every encountered question. As a result, we want to identify opportunities for more focused further research.

This way, the present thesis represents a stepping stone that will accelerate the future long-term research program on bio-inspired DNN architectures for neuroscientific data underway in the Antolík et al. group.

Thesis structure

This thesis is divided into 4 parts. First, in chapter 1 we provide the theoretical background. We start with a high level overview of the initial part of the visual processing system. Then, we introduce both computational neuroscience generally and the tools it uses for system identification tasks, as well as provide a brief summary of the deep learning techniques that will be relevant for our experiments. In chapter 2, we introduce the Antolík et al. model and other especially relevant work in more detail. Second, in chapters chapter 3 we describe the implementation of the additional methods necessary to realize our architectures and the experiments pipeline. Then, in chapter 4, we introduce our methodology for both the model exploration and results analysis, and finish with a training regime overview.

Chapter 5 with experiments and their results follows. We start by reimplementing the Antolík et al. model. In the first section, we analyse it in terms of stability and sensitivity to training hyperparameters, attempting to get the best and most stable version possible. Then we test regularization on the fully connected layers, impact of additional non-linearity, and input scaling. In the second section, we move towards larger architectural changes, testing elements from other state of the art models, traditional deep computer vision, but also drawing inspiration from simpler tools of computational neuroscience. We conduct a comparison between various computationally similar architectures differing only by the explicitness of imposed regularization. In the third section we explored the effects of transfer learning, by training one instance of a model on all of the dataset's three separate subsets pooled together. Further, we train the best variants of architectures from previous sections and compare their results to assess the universality of their improvements.

Finally, the last chapter offers a summary of our findings, provides lessons learned, and suggests ample opportunities for further research.

1. Background

In this chapter, we start by introducing the initial part of the visual processing system. Then, we describe both computational neuroscience generally and the tools it uses for system identification tasks. We finish with a brief overview of deep learning, followed by a few particularities that will be especially relevant for our experiments.

1.1 Visual sensory neuroscience

In this section, we provide an introduction to visual neuroscience necessary for understanding the biological plausibility of our models. For a more comprehensive and in-depth introduction to neuroscience of vision please refer to *Neuroscience: Exploring the Brain* [Bear et al., 2007] or the excellent summary provided as part of Carandini et al. [2005]'s review.

1.1.1 Early visual system

In this thesis we restrict our focus to the first 3 early stages of visual processing, corresponding to three neural structures: the retina, lateral geniculate body (LGN), and primary visual cortex (V1).

The journey of a visual stimuli starts on the retina at the back of an eye, where photosensitive cells – rods and cones measure the intensity of the light reflected by objects in our field of vision and projected at the back of our eye by cornea. While the particularities of rods and cones are interesting, especially when it comes to color processing, for the purpose of this introduction we will treat them as simple detectors that output electrical signal proportional to the sensed luminance, resulting in what could be abstracted as a matrix of luminescence levels – a raster image.

The first processing step happens directly in the eye through retinal ganglion cells. Again, these cells are more complicated than presented here, but on a high level they detect local features such as contrast, directional movement of small stimuli, etc. The two main types are ON and OFF cells, both having a circular receptive field. Receptive field is the portion of sensory space that elicits neural responses when stimulated. In case of vision that means the section of the visual field a particular neuron responds to. ON cells fire when a high luminance hits the the center of its receptive fields and the disk at the edge is, proportionally, less excited. OFF cells function in the exact opposite way. For illustration, refer to figure 1.1.

From the eye most of the signal travels to LGN[1], which serves as an active bridge to the visual cortex at the back side of the brain. Much like later V1, LGN is composed of 6 layers. The processing that happens here could be summarized as very similar to the retinal ganglion cells, only with bigger receptive fields. It is important to note that LGN does not get input only or even mainly from retina. It also receives substantial feedback from the primary visual cortex, and

[1] Describing the visual pathway as a strictly feed-forward network is a great simplification.

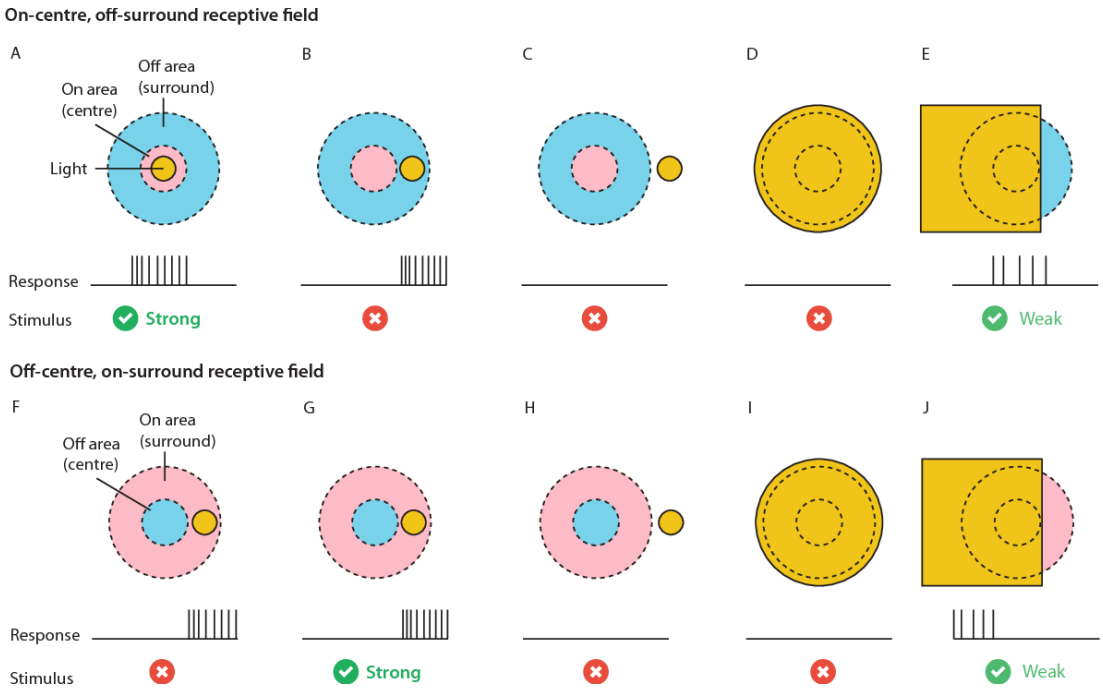

Figure 1.1: Receptive fields of retinal ganglion and LGN cells. Adapted from [Maguire, 2015].

has synapses from parts of the thalamus and the brain stem. For example, the brain stem connection is believed to be responsible for perceived visual impacts of alertness changes, such as a flash of light when one is startled in a dark room. However, the exact role of these extra-retina sources in LGN function remains largely not understood. This means that the activity of both LGN, and all downstream stages of processing, such as V1, is modulated not only by the visual stimuli but also other neural activity inducing factors, for example emotions, locomotion, or other sensory inputs. Any models predicting the activity of LGN, V1, or higher stages of visual processing that are based solely on visual stimuli cannot, therefore, explain the entirety of recorded data. Nevertheless, under restricted experimental conditions, a wide range of studies have shown that the response of LGN cells can be very well approximated by a linear filter composed of difference-of-Gaussians.

At last, the signal reaches the primary visual cortex. It is divided into 6 layers. Within V1, we recognize two main functional cell types – the so-called simple and complex cells. Simple cells, as their name suggests, can be represented as mostly linear combinations of ON and OFF cells. This means that within their receptive fields, they have well defined excitatory and inhibitory regions. In essence, they elicit response when stimulus, in this case light, hits the excitatory regions substantially more than the inhibitory regions. Their receptive field tends to be axis aligned, making them sensitive to oriented edges and gratings. They are also commonly described as Gabor-like filters (see Fig. 1.2).

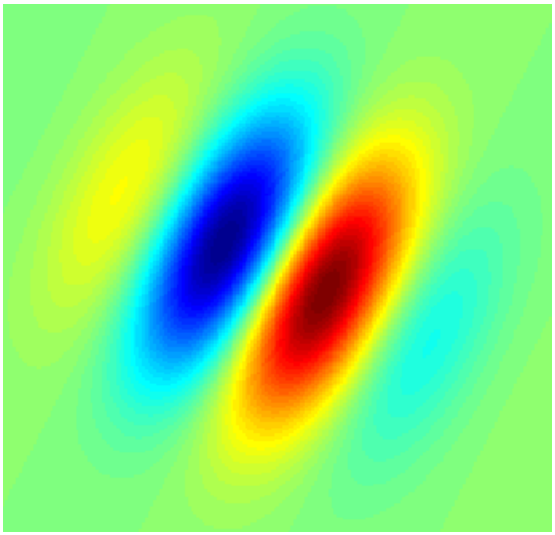

Figure 1.2: Gabor filter-type receptive field typical for a simple cell. Blue regions indicate inhibition, red excitation. Figure was adopted from [Pharos, 2006].

The nature of complex cells is not yet entirely settled, but it is understood that the relationship between the visual input and their response cannot be explained with linear function. Complex cells are known to be orientation selective, just as their simple counterparts, but their hallmark is that they are not sensitive to the exact position of the oriented stimulus within their receptive field. It is thus hypothesized that their output is a non-linear combination of a number of co-oriented simple cells. If the hypothesis was true, it would essentially create a hierarchy, where the LGN cells roughly correspond to concentric ON/OFF filters of the stimuli, simple cells to a linear combination of LGN neurons, and complex cells to nonlinear combinations of simple cells. In addition to aforementioned stimuli, V1 neurons are also selective to other visual features, such as color, phase, spatial and temporal frequencies etc.

1.2 Computational neuroscience

In the following subsections, we provide an introduction to the traditional system identification methods used in visual neuroscience. For further details, refer to *Data-driven approaches to understanding visual neuron activity* [Butts, 2019] or Carandini et al. [2005]'s review.

1.2.1 General overview

In computational neuroscience, we can generally distinguish two lines of inquiry. The first one, attempts to model biological reality even on the lowest level, focusing on particularities of every single neuron. Due to the inherent computational requirements of evaluating the biological neuron model, including the necessity to properly simulate temporarily to allow for mechanisms like refractory period, they usually do not scale well to larger systems. Such models are usually too complex, with too many free parameters to be fitted directly to recorded neural data through gradient based techniques.

The second approach abstracts away details of singular neurons and focuses on higher level computation and topological properties. As such, it can leverage tools from classical machine learning and most recently even deep learning. It is prevalent in visual sensory neuroscience where it is commonly needed to model 'black-box' systems, such as the whole path from retina to primary visual cortex across dozens of neurons. In such models, the neural function is represented by a relatively generic computation whose parameters are fit to data to correspond to each neuron's specific processing. Formally, for stimuli s, response r, parameters (also called weights) o, and model m, we have (Eq. 1.1).

$$r = m(s|o) \tag{1.1}$$

In the rest of this chapter, and thesis in general, we will consider only statistical models for solving system identification tasks within the domain of visual sensory neuroscience. Our inputs will be images presented to a subject, in case of our data a mouse, in a form of 2D pixel matrices. The outputs will be real values, one per recorded neuron, representing an estimate of the measured neural activity elicited by presentation of the given image.

1.2.2 Classical models

The simplest method possible for predicting a single neuron's response to an image stimuli is a *linear model*, essentially a linear regression over the pixels of the input. Mathematically, for parameters o (also called *weights* or in this case *filter*) and an image of height h and width w pixels that gives us the equation 1.2. In such a linear model, the weights can be fitted in a number of ways, either through gradient descent methods, or analytically to compute the optimal solutions from the whole dataset. The second approach is commonly called *Spike Triggered Average (STA)*. Both ways of fitting can incorporate regularization for the weights, for example to punish a high first derivative in both spatial dimensions, ensuring smoothness [Willmore and Smyth, 2003]. This is particularly necessary in the neuroscientific settings where datasets tend to be of limited size, and overfitting of models is thus a major challenge.

$$r = \sum_{x,y=0}^{w,h} s_{x,y} o_{x,y} \tag{1.2}$$

A natural extension of a linear model is a *linear-nonlinear (LN) model*. As the name suggests, it is a linear model whose output is modulated by a single nonlinear function f (Eq. 1.3). This type of model is commonly used in a *regularized variant (rLN)*. Due to its direct interpretability – we can easily visualise the filter, ease of analysis – the parameters (filters) are easy to compare and work with using image processing toolkit, and computational simplicity – it is just a linear operation, LN models were the de facto standard model of computational neuroscience until relatively recently.

$$r = f(s \cdot o) \tag{1.3}$$

In an effort to further improve the prediction power of the LN model, several of its extensions appeared. First as a *generalized LN model (GLM)*, where one

filter is replaced with a set of k filters (Eq. 1.4).

$$r = f(s \cdot o_0, s \cdot o_1, ..., s \cdot o_k) \tag{1.4}$$

Since the k-ary non-linearity function can combine the individual results in a non-linear way, this increases the computational expressiveness of the model. There are two limitations of this approach. It makes the k-ary function an additional hyperparameter and, in turn, does not allow the way individual filters are combined to be fitted to data. The *LNLN model* solves both problems, replacing the k-ary non-linear function with a linear combination of several LN models (Eq. 1.5):

$$r = f(\sum_{i=0}^{k} w_i f_i(s \cdot o_i)) \tag{1.5}$$

In addition to solving the two problems of GLM and being more interpretable, as it is simply a linear combination of LN models whose filters we can visualise with a non-linearity at the end, the fact that the combination can be individually parametrized opened the door to a particular multi-step learning strategy. When modeling multiple neurons that are believed to be similar in function, we can share the first level filters (o_i) between them, and only individually fit the w parameters. This allows the filters o_i to be based on more data – from all neurons instead of just one, and in turn, decreases the chance for overfitting.

Furthermore, the filters can be parametric, e.g. gabor functions [Kay et al., 2008], gaussians, or pre-computed bank of filters – for example borrowed from classical image processing toolkit. This can serve two goals. First, significantly decrease the number of free parameters of the model. While a normal filter for a 31x31 image has 961 parameters, a simple gaussian filter has only 4, strength, x and y coordinates of the center, and width. Second, it can ground the model in biological reality. For example, the computational properties of retinal ganglion cells are known to be well approximated in space by a *difference-of-Gaussian function*[2] and so one can inject such priors in the model. We will call this approach *hard regularization*. Similar effect can be achieved on arbitrary filters with regularization during parameter fitting which penalizes forms of the filter that diverge from our desired shape, as explained above for the LN model. We will call that *soft regularization*.

1.3 ML and Deep neural networks

In this section we will provide a brief overview of deep neural networks (DNN) fundamentals to introduce the most relevant principles and unite terminology. For a comprehensive description of general machine learning and particularly DNN methods please refer to the *Deep Learning book* [Goodfellow et al., 2016].

1.3.1 Feed forward neural network

Deep neural network (DNN) is a statistical machine learning model vaguely inspired by the brain. The term neural comes from the fact that it is traditionally

[2] A subtraction of two concentric 2D Gaussians.

composed of computational neurons[3] (also called perceptrons[4]). Computational neurons are mathematically defined as a weighted sum of their inputs followed by, usually a non-linear, activation function. Deep and networks, because they are usually composed of several interconnected layers of neurons. A single layer of a DNN is defined by its inputs x and outputs y tensors. The output of the k-th neuron of the simplest, a fully connected, layer with n inputs is then (Eq. 1.6):

$$y_k = f(\sum_{i}^{n} x_i w_{k,i}) \tag{1.6}$$

When multiple layers $f_1, f_2, \ldots, f_n$ are stacked on each other – to form a deep neural network, the outputs of the preceding layer are used as the inputs of the following one. The output of a complete DNN is then $y = f_n(\ldots f_2(f_1(x))\ldots)$. To tie this back to computational neuroscience, a LN model would be a single layer neural network. Its input size would be equal to the size of stimulus and output size to 1, as we are predicting a single neuron. Similarly, a single neuron LNLN model would be two layers DNN, where the size of the first layer would correspond to the number of LNLN filters, and the second would have size 1. If we wanted a LNLN model where the filters are shared across multiple fitted neurons, it would again be two layers DNN. The second layer would have size corresponding to the number of recorded neurons, however. Each output neuron (output of the second layer) would then be an independent combination of shared filters (the outputs of the first layer).

Name	**Function**
ReLU	$f(x) = max(0, x)$
Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$
SoftPlus	$f(x) = ln(1 + e^x)$
Linear/Identity	$f(x) = x$
TanH	$f(x) = tanh(x)$

Table 1.1: Some of the commonly used activation functions[5].

Much like the LNLN filters, DNN layers do not have to be generic. Similar to what we termed *hard regularization*, they can exploit known properties of the computation they are trying to model. Most notably, convolutional layers make use of local properties of inputs with continuous dimensions, such as the spatial dimensions of image inputs[6]. Their outputs are the result of a location invariant filter applied at all spatial locations of the input. For more information, refer to introduction to *Convolutional networks* in [Štěpán Hojdar, 2019].

[3]Not to be confused with biological neurons [Bear et al., 2007], [London and Häusser, 2005].

[4][Rosenblatt, 1958]

[5]For neural modeling, strictly positive functions such as ReLU or SoftPlus are especially interesting due to the non-negative nature of firing rate.

[6]Deep neural networks with convolutional layers are commonly called convolutional neural networks (CNN).

1.3.2 DNN training

Due to the number of free parameters in even relatively shallow DNNs, these models cannot be fitted analytically. Instead, iterative gradient descent based methods, such as stochastic gradient descent (SGD [Kiefer and Wolfowitz, 1952]) or more recently ADAM [Kingma and Ba, 2014], are commonly used. In gross simplification, the fitting process, also called model training, works as follows. A small random subset (*batch*) of input and corresponding desired output (*gold data*) is taken from the dataset. The input portion of a batch is used as the model's input, computation happens, and we get a prediction from the model. Then, the model's prediction and corresponding gold data are put into a *loss function*. It serves as the metric of how far the model prediction was from the desired output. The loss function can also penalize (*soft regularization*) certain properties of the model's parameters (e.g. increase the loss for every non-zero weight). As the last step, first derivatives of the loss function with respect to all the model's free parameters are taken (*gradient*) and subtracted (*model update*) from the parameters to, in theory, decrease the loss value next time. This step is repeated for all batches in our dataset[7]. The whole process up until now constitutes one *epoch*. Epochs are then repeated until convergence.

In addition to layers that provide direct computation, DNNs can also feature layers whose primary objective is to help during training. One such example is the dropout layer as introduced by Srivastava et al. [2014]. It randomly zeros portion of its output neurons during training, thus forcing the rest of the network to not to rely on individual inputs, and prevents overfitting[8]. Another is the batch normalisation layer introduced by Ioffe and Szegedy [2015], that forces its outputs to be of 0 mean and 1 variance within a single batch, and – if needed – to learn any linear transformation of its outputs explicitly. This aims to stabilise the learning process.

1.3.3 Transfer learning

Training a DNN, e.g. for a vision task, from scratch requires large amounts of data. Fortunately, the observation that the initial layers of any DNN trained on image data are functionally similar, starting at low-level feature detection and eventually building up to higher level concepts, allows us to leverage models trained on generic tasks with huge pre-existing datasets.

We can simply take an initial portion of an already trained network, use its layers including trained weights as the foundation of a new model, and only append it with a few new problem specific layers that are then trained from scratch. This can substantially cut down on the amount of data required, as we only need to constrain the parameters of newly added layers, and also serves as a regularization technique, making sure the initial low level filters are not overfitted on our limited data[9].

[7] This means a larger batch size proportionally leads to less model updates per epoch.

[8] Alternative interpretation: the dropout layer leads to training an ensemble of models that share the majority of parameters.

[9] [Tan et al., 2018]

1.4 DNNs and computational neuroscience

There are several differences between system identification tasks of computational neuroscience and classical problems of computer vision. Mainly, the amount and quality of data. Where for example image classification datasets usually feature tens of thousands of precisely annotated examples[10], system identification tasks on early visual processing have often only few thousands of substantially noisy recordings. This has vast implications for not only viable architectures, as more layers mean more free parameters (e.g. 65 millions in case of YOLOv3[11]) that have to be constrained by data to prevent overfitting, but – as we will see, also has the potential to invalidate our intuition around many aspects such as regularization and model stability we might have from working with larger models and datasets.

In the following section we will introduce particular tools and techniques that are commonly used by system identification methods in neuroscience and which will be featured in our exploration, but are not as well known in the classical machine learning or computer vision domain.

1.4.1 Poisson versus Gaussian loss function

Since our system classification problem is essentially a regression, the output consists of a real valued response for each neuron. We need an appropriate loss function to measure the distance of the model's prediction to the recorded data. It is common to assume that the data contains noise with *Gaussian distribution* and then construct the loss function through most likelihood estimation (MLE) method, leading to mean squared error (MSE). Gaussian distribution is commonly used due to it being the maximum entropy probability distribution for a fixed mean and standard deviation. Intuitively speaking, it is the distribution that assumes the least given mean, which is zero for the noise, and static variance.

In our case, with outputs that are proportional to the spike rates of neurons, we do not have to assume the most generic distribution, however. There is evidence [Goris et al., 2014] that the variance of neural response roughly corresponds to the neuron's firing rate. Furthemore, spiking outputs are strictly non-negative, invalidating the assumption of Gaussian distribution. That means *Poisson distribution*, which assumes variance proportional to its mean and is non-negative, is a more optimal choice for modeling them. Through MLE, for measured firing rate x and predicted firing rate y that gives us the loss function on equation 1.7 .

$$loss(x, y) = y - x \log(y) \tag{1.7}$$

1.4.2 Correlation as performance metric

Even though the loss function based on Poisson noise assumption is the optimal metric to optimise during training, it is not very good for eventual analysis and comparison of different models, potentially across multiple datasets. Not only is it dependent on the assumed noise distribution, which is a technical detail of the fitting process, rather than a feature of the actual resulting fitted model, but it

[10] Cifar-10: [Krizhevsky et al., 2015], Fashion-MNIST: [Xiao et al., 2017]

[11] [Redmon and Farhadi, 2018]

is also influenced by the scale of output data, and generally does not have an intuitive interpretation.

For those reasons, either the *Pearson's correlation coefficient* (Eq. 1.8) or the proportion of explained variance between each neuron's model predicted output and its measured response is more commonly used instead. In this thesis, we will use Pearson's correlation coefficient averaged across all recorded neurons for simplicity.

$$r(x, y) = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2(y_i - \overline{y})^2}} \tag{1.8}$$

1.4.3 (Soft) Regularizations

In addition to the wide set of traditional parameters regularizations common to all machine learning methods, such as L1 and L2, that simply punish either the size of or the number of non-zero parameters, computational neuroscience commonly uses regularizations inspired by biological plausibility. One such regularization, that will be prominent in our experiments, is *Laplacian regularization*, an L2 penalty on a Laplacian convolution filter. By punishing high first order derivatives in spatial dimensions, it ensures smoothness of regularized filters.

Name	**Function**
L1	$\sum_i ||w_i||$
L2	$\sum_i w_i^2$
Laplacian	$\sum_{x,y,i,o}(W_{:,:,i,o} * L)^2_{x,y}, \quad L = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$

Table 1.2: Relevant weights *soft regularizations*.

2. Key and related works

In this chapter, we start by introducing the *HSM model* by Antolík et al. [2016] model, which will be the main focus of our experiments. Further, we describe a model by Klindt et al. [2017], whose *separable layer* contribution we will also explore. To provide some background, we finish with a quick look at a few additional related works.

2.1 Antolík et al. and the DoG layer

Published in 2016, the paper *Model Constrained by Visual Hierarchy Improves Prediction of Neural Responses to Natural Scenes* by Antolík et al. introduced a three layer DNN model to fit primary visual cortex responses to image stimuli. As its main contribution, it explored incorporating biologically plausible components with traditional DNN methods. Doing so, it managed to outperform[1], at the time of writing, other state of the art methods while providing greater interpretability and requiring less free parameters.

The model is grounded in known hierarchical properties of the early visual system in three ways. First, it assumes that LGN units can be well modeled using difference-of-Gaussians filters. Second, that local population of V1 neurons share inputs from a limited number of such LGN units. Third, that simple cells can be constructed as a combination of several LGN neurons and complex cells similarly from simple cells. Based on these assumptions, the model consists of 3 layers (Fig. 2.1): the first represents both the retinal and LGN computation, and the second and third the two levels of V1 neurons.

2.1.1 DoG layer

Retinal and LGN computations are modeled together using a set of parallel difference-of-Gaussian filters followed by an identity activation function. In the context of the HSM architecture, we will call this the filter layer. The usage of difference-of-Gaussian filters significantly decreases the number of free parameters introduced by this layer while being grounded in biological reality. Instead of `input_width*input_height` parameters per filter of a fully connected layer, the *Difference of Gaussians layer (DoG)* is parametrized by only 6 per filter: weight and width of the center and surrounding Gaussians and x, y coordinates of their shared center. The published model contains 9 filters DoG layer. For input Sx, y, weights α_1 and α_2, widths σ_1 and σ_2, and center μ_x, μ_y the output of a single DoG filter is:

$$\sum_{x,y}^{w,h} S_{x,y} \left(\frac{\alpha_1}{2\sigma_1\pi} e^{\frac{(x-\mu_x)^2+(y-\mu_y)^2}{2\sigma_1}} - \frac{\alpha_2}{2(\sigma_1+\sigma_2)\pi} e^{\frac{(X-\mu_x)^2+(Y-\mu_y)^2}{2(\sigma_1+\sigma_2)}} \right) \tag{2.1}$$

[1]For results refer to *Prior works results* (section 4.1.2).

2.1.2 V1 neurons

The two levels of V1 neurons are modeled as two consecutive fully connected layers. We will call these the hidden and output layers. The output layer has the same size as the number of measured neurons. The hidden layer is proportional to the size of the output layer. In the case of the reported model, its size is 20 % of the number of measured neurons. Both fully connected layers are followed by a SoftPlus[2] nonlinearity.

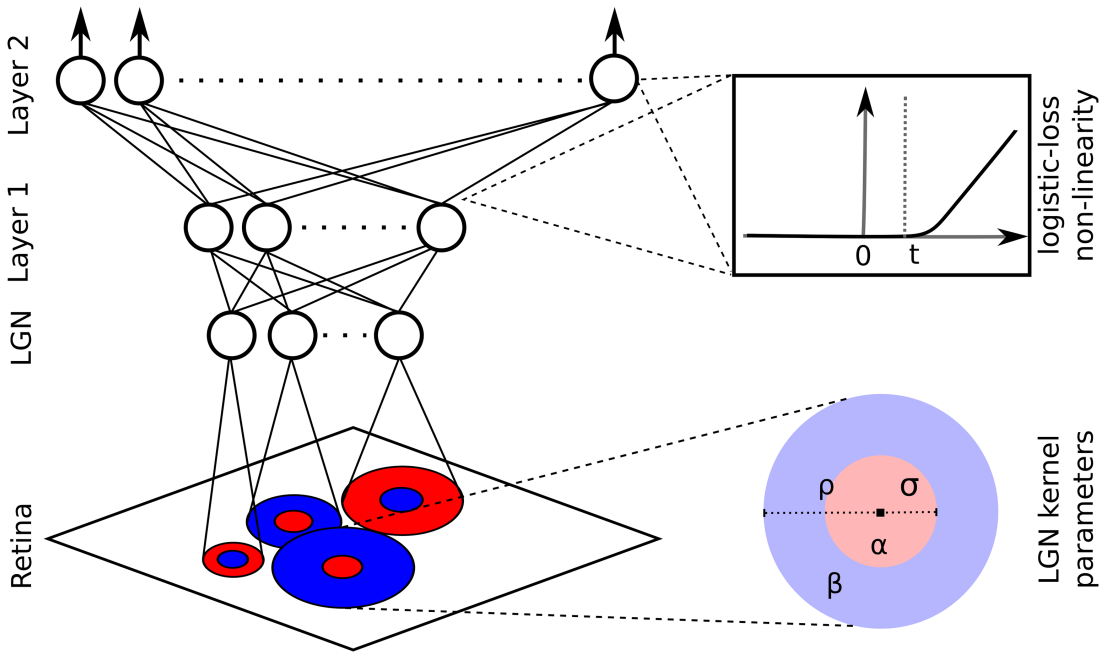

Figure 2.1: Three layer architecture of *HSM model* and corresponding neural sections. Figure was adopted from [Antolík et al., 2016].

2.1.3 Training regime

The published model was trained using a Newton Conjugate-Gradient algorithm[3] with 100 epochs, each consisting of 10 000 evaluations maximum. The batch size for this optimiser is the entire dataset. In addition to the *hard regularization* provided by the parameterized filters of the DoG layer, all parameters – of the weights of DoG layer and both fully connected layers – were kept within predefined bounds by the optimiser throughout the fitting. These bounds were also used for random initialization of the parameters, during which values were drawn from uniform distributions with corresponding bounds. No other *soft* or *hard regularization* is present in the model.

The reported values for the hidden layer size ratio and the number of DoG filters hyperparameters were found empirically using two one-dimensional searches through the parameter space.

[2]Refer to table 1.1.

[3]https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.fmin_tnc.html.

2.2 Klindt et al. and the separable layer

In 2017 Klindt et al. published a paper *Neural system identification for large populations separating "what" and "where"* that explored deep convolutional neural networks in the context of V1 neural data. It specifically focused on the estimation of individual neurons' receptive field locations through a novel approach to readout layer[4] factorization, in an effort to allow effective fitting of thousands of neurons on relatively little data simultaneously. In addition to artificial data, it was also evaluated on the same dataset as Antolík et al. [2016] where it achieved moderate improvements[5] over the HSM model by Antolík et al.

2.2.1 Architecture

The model (further also referred to as the *what/where model*) consists of two parts, feature space and receptive fields (Fig. 2.2). The feature space is a cascade of – in the variant for Antolík et al. dataset – 3 convolutional layers, each followed by a batch normalisation[6] and a SoftPlus activation function. The convolution layers have 48 feature maps per each layer, with the first one having 13 pixel and the other two 3 pixel kernels. In addition, the convolution layers feature two types of *soft regularizations*. A Laplacian regularizations to ensure smoothness of the filters and an L2 based group sparsity regularization[7] to encourage filters to pool from only a small set of feature maps in the previous layer.

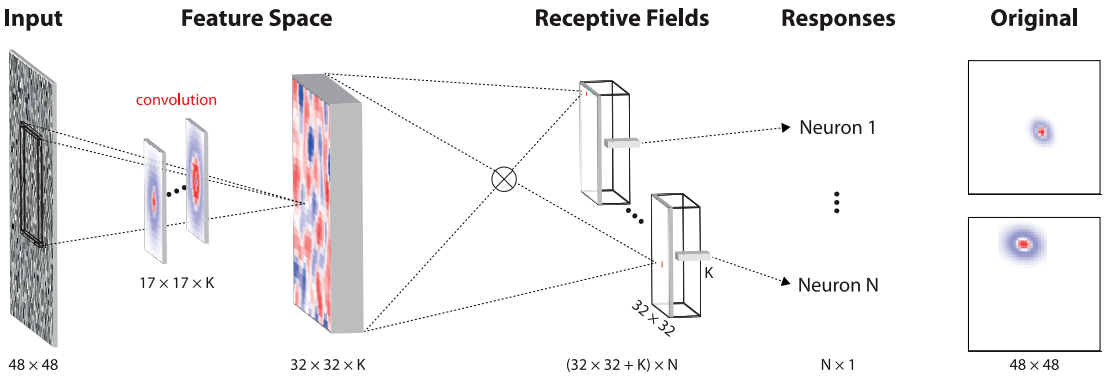

Figure 2.2: Architecture of Klindt et al. model, consisting of convolution layers and a factorized readout layer. Figure was adopted from [Klindt et al., 2017].

The receptive field consists of a single instance of a specific readout layer (further referred to as the *separable layer*). It is a fully connected layer factored into two masks per each output neuron. First mask, with dimensionality `feat_space_output_spatial_dims`, selects the input locations the output neuron will use, in essence its receptive field – the *where*. The other, with dimensionality of `feat_space_output_channels`, determines what channels of the feature space at each of the locations will make the neuron's output – its *what*. This factorization achieves two things. It decreases the number of free parameters, from

[4] Readout layer is the first fully connected layer after a set of convolutional layers.
[5] For exact results refer to *Prior works results* (section 4.1.2).
[6] [Ioffe and Szegedy, 2015]
[7] For a definition, please refer to the original paper [Klindt et al., 2017].

`spatial_dims*channels` to `spatial_dims+channels` but also enables more direct interpretability of the parameters. Comparing the *what* masks, for example, allows us to identify similar cell types. Both masks feature L1 regularization to encourage sparsity.

The model was trained using ADAM optimiser and early stopping on 20/80 training set split. Reported hyperparameters, such as the number and size of convolution filters, were found and cross-validated using grid search for each region.

2.3 Related works

To provide some background, this section lists a few related works that are relevant to V1 system identification but are not directly used or in any way referenced by our experiments. Specifically, we will introduce a few convolutional architectures, as they have seen an increase in usage for V1 modeling, similarly as they have taken over classical computer vision[8].

2.3.1 Neural convolutional models

Ecker et al. [2018] inspired by aforementioned Klindt et al. [2017], further explored convolutional architectures. In addition to the Klindt et al.'s observation that many neurons of early visual processing are similar but only have their receptive fields at different locations, Ecker et al.'s model leveraged the fact that even more V1 neurons are functionally the same, if we assume not only arbitrary receptive field positions but also arbitrary orientations. Based on the same two parts, 3 convolutional layers architecture introduced by Klindt et al., their model used *Group equivariant convolutions* [Cohen and Welling, 2016] instead of traditional convolution layers.

Tackling a similar problem from a different angle, Walke et al. [2018] introduced a convolutional architecture, where the readout locations – representing the receptive fields of output neurons – are modulated by a *shifter side network* based on eye movement data. In addition, the neural outputs are further adjusted by a *second side network* using behaviour data (running state, pupil dilation), trying to incorporate extra-retina sources of LGN and further downstream stages of visual processing. For the core network, the model used a three layer convolutional cascade followed by a neuron specific linear readout stage.

2.3.2 Transfer learning

Motivated by the observation that lower parts of the visual pathway can be seen analogous to early layers of classical object recognition neural networks, Cadena et al. [2019] explored transfer learning within the domain of V1 system identification. Their proposed architecture used 16 feature maps[9] from the initial section of an on-image-classification-dataset already trained VGG-16 network [Simonyan

[8]Even though they are slowly being superseded by attention/transformer-based networks: [Ramachandran et al., 2019], [Bello et al., 2019]), [Dosovitskiy et al., 2020].

[9]The output of one filter of a convolutional layer given a specific input.

and Zisserman, 2014] and then fitted per output neuron readout layer on neural data. This approach managed to achieve parity with a 3 convolutional layer CNN architecture trained entirely on neural data, while requiring a smaller training dataset.

3. Implementation

In this chapter, we are going to provide a high-level overview of the NDN3 library, introduce our extensions of it, and briefly describe the system developed for running and analysing experiments. It is intended as neither comprehensive documentation nor a detailed implementation guide. We include it to clarify the software infrastructure context of the project and to facilitate its future continuation.

All code implemented as part of this thesis, including all experiment scripts, can be found within or is explicitly referenced by the `msc-neuro` (attachment A.1.1).

3.1 Overview of NDN3 library

NDN3 is a modeling framework for describing nonlinear computations in experimentally-measured neural data. It is focused on feed-forward stimulus processing but can facilitate latent variable models as well. It comes with a comprehensive set of readily available layer types, regularizations, and tools that are either biologically inspired or of particular relevance to neural data analysis. Through its TensorFlow foundations, it provides high performance and allows easy extension with new tools using state of the art machine learning primitives.

It is an ongoing project developed by the Neurotheory Lab[1] of prof. Dan Butts[2] at the University of Maryland. It started as a python2 library FFNetwork in late 2017 and was soon transformed into a comprehensive NDN framework in early 2018. In 2019 work on its port to python3 in the form of NDN3[3] began. Recently, following prior collaboration, it has been adopted by Jan Antolík's group at the Faculty of Mathematics and Physics of Charles University.

Under the hood, NDN3 is implemented using TensorFlow 1.x (TF)[4], NumPy (NP)[5], and a few other libraries from the SciPy stack. Its intended usage philosophy is similar to TensorFlow 2.x sequential approach[6]. The centerpiece is an object-oriented model instance whose architecture is defined using a list of layer types and parameters as they sequentially form the model.

At the highest level, there is an instance of an NDN model class. The NDN instance encapsulates the complete computational model, its architecture, hyperparameters, trained weights, etc. For manipulation, it provides methods for fitting to data, accuracy evaluation, or generating new predictions based on provided stimuli. It can also serialize and save itself to a file or load itself from one.

On creation, NDN model's constructor receives a list of dictionaries, each describing a network that will be part of the model (Fig. 3.1). For our experiments, we will limit ourselves to just one network per model but NDN3 allows multi-

[1] http://neurotheory.umd.edu/
[2] http://biology.umd.edu/daniel-butts.html
[3] https://github.com/NeuroTheoryUMD/NDN3
[4] https://www.tensorflow.org/
[5] https://numpy.org/
[6] https://www.tensorflow.org/guide/keras/sequential_model

ple in various configurations, such as side networks and sampler networks. Each network is a feed-forward stack of layers. It is fully described by a dictionary of number-of-layers-long lists where every key represents a parameter and its value – in the form of a list – dictates what the value of that parameter should be for each layer of that specific network.

```
[{
 'network_type': 'normal',
 'input_dims': [1, 31, 31],
 'layer_sizes': [20, 102],
 'layer_types': ['normal', 'normal'],
 'activation_funcs': ['softplus', 'softplus'],
 'reg_initializers': [{'d2x': 0.1}, {'l2': 0.1}],
 ...
}]
```

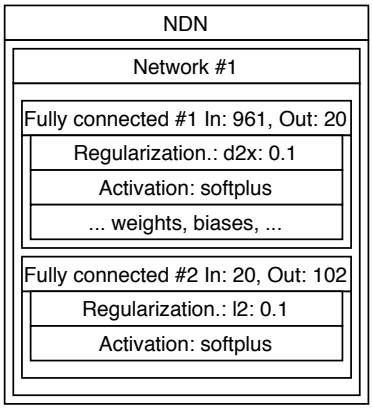

Figure 3.1: Model description and the resulting NDN instance for an LNLN model with 31x31 pixels input, 102 output neurons, 20 filters, SoftPlus non-linearities, and regularizations.

The NDN model class recursively contains instances representing its parts. The NDN object includes an array of network objects, each network object contains a stack of layer objects, and each layer object has, among others, regularization objects. The layer object also contains fields representing all of its hyperparameters, specifying its activation function, and – among a few more – also maintains NP arrays backing all its free parameters such as weights and biases. All of this structure is created as part of the network's definition which happens from within the NDN's constructor. This is also where parameters initialization happens through the backing NP arrays.

Whenever computation is invoked – e.g. when `train` or `generate_prediction` methods are called, a new TF session is initialized, a computational graph is recursively created from scratch using the description saved on the internal network and layer objects, and finally, weights are loaded to the session from the aforementioned layers' parameters backing NP arrays. When the computation finishes, all weights are gathered from the TF session to plain NP arrays and saved within layers' objects of the model again. For visualisation, refer to figure 3.2.

All shapes are inferred from input dimensions, specified layer sizes, and – e.g. in the case of convolutional layer – its stride and filter size, with the last layer size serving as the output dimensions. The interpretation of input dimensions is (`time`[7], `width`, `height`). Internally, the framework assumes `(batch_shape, height, width, channels` or `time)` dimensionality and uses flatten representation to float data between layers. When necessary, for example to apply 2D convolution, a layer restores the full dimensionality, both spatial and channels/time dimensions, and, after applying its function, flattens the data again.

[7]This is to support temporal models which we will not cover.

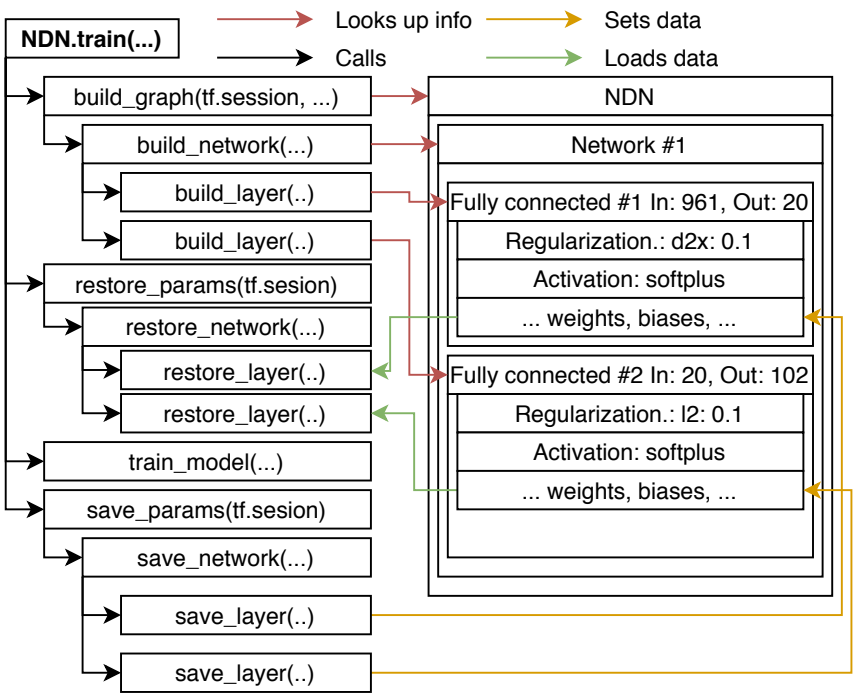

Figure 3.2: Diagram of NDN model initialization when `train(...)` method is called.

3.1.1 NDN3 toolkit

The featureset of NDN3 is vast and continuously evolving and so it is not our intention to provide a comprehensive summary. Below is just a subset of tools that are either relevant to our work (implemented additions in *italic*) or that we wanted to particularly draw attention to for potential future continuation of our exploration.

- **Loss functions (noise assumptions):** Gaussian, Poisson, Bernoulli.
- **Layers:** Fully connected (normal), Separable, Convolution, Convolutional separable[8], Readout[9], Convolutional readout, Additive[10], *DoG*, *Convolutional DoG*, *Linear scale*[11], etc.
- **Parameters regularizations[12]:** Dropout, center[13], norm2[14], max[15], L1, L2, Spatial Laplacian (d2x), etc.

[8]Each filter is factored into a convolutional *what* and *where* masks.

[9]Outputs the value of spatial location defined by an explicit parameter. Not to be confused with general *readout layer* (any first layer after convolutional layers).

[10]Combines inputs from multiple input streams.

[11]Linearly scales its input using a scalar multiplier and single additive bias: $input * x + b$.

[12]Regularizations are the only externally documented part of NDN: https://github.com/NeuroTheoryUMD/NDN/blob/master/docs/glossary.rst

[13]Penalizes higher weights far from the center.

[14]Encourages norm of weight matrix to be 1.

[15]Penalizes each pair of non-zero weights (i.e. heavily discourages more than 1 non-zero weight), both in spatial and channel dimensions.

- **Parameters normalizations**[16]**:** maxnorm, L2.
- **Activation functions:** ReLU, sigmoid, tanh, identity, SoftPlus, leaky ReLU, etc.
- **Weights/bias initializers:** zeroes, normal[17], trunc normal[18].

Apart from these traditional tools, NDN3 also provides following capabilities:

- **Metrics capture:** Loss value and *potentially*[19] *correlation metrics* can be logged as TF summaries periodically throughout the training loop as part of the summary step.
- **Layer activation logging:** In addition to model-wide metrics such as the loss value, NDN3 allows for individual layers' activations to be logged as histograms.
- **Validation while training:** If specified, a validation set can be evaluated using abovementioned metrics as part of the summary step of the training loop every few epochs. Do note that the validation set is evaluated in its entirety at once and is not batched. Respective metrics are saved as a separate TF summary.
- **Support for early stopping:** Using the same mechanism as the abovementioned validation metrics, NDN3 supports early stopping.
- **Explicit inhibitory/excitatory neurons enforcement:** It is possible to specify the ratio of excitatory/inhibitory neurons on supported layer types (normal, convolution, etc.). This works through enforcing strictly non-negative weights and then multiplying the layer's output with a +/-1 mask.
- **Data filters:** In addition to input and output data, it is possible to provide data filters. A boolean mask of the same dimensionality as the output data that specifies which individual output neurons should be ignored (zeroed for prediction, ignored for loss value computation, metrics, etc.) for what particular data points. This allows for, among other things, a common model trained on two separate recordings (e.g. 10 and 15 data examples) of two neuron regions (e.g. containing 5 and 7 neurons). The output dimensionality of such a model would be 12. Where valid output data are not available, i.e. for neurons 5 to 12 of the first 10 data points, zeroes can be used. In such case, the data filter mask has ones on the indexes 0 to 5 for the first 10 data points, and zeros for indexes 5 to 12. The situation is mirrored for data points 10 to 25. This way, when NDN model computes an output of a second region's neuron (e.g. neuron 11) for a first region's data point (e.g. 5th data instance) while training, it can see the 0 in `data_filters[5][11]`

[16]This is parameters normalization i.e weights are normalised each time before used. Activation normalization, such as batch normalisation [Ioffe and Szegedy, 2015], is notably missing.
[17]Normal distribution with 0 mean, 0.1 variance.
[18]Absolute value of normal.
[19]Not merged upstream, please refer to the `msc-neuro`'s `./README.md`.

mask and know to not include its not-trained output in the loss function computation and subsequent optimisation.

- **Many more:** Several data ingestion pipelines, support for temporal models, etc.

3.2 Implemented NDN3 extensions

To facilitate our research, we implemented several extensions to the NDN3 framework[20]. Our main contribution was in the form of the Difference of Gaussians layer[21] as introduced by Antolík et al. [2016]. We also implemented a convolutional variant of this layer. To help with training progression interpretability, support for Pearson's correlation tracking as part of TF summary was also added, as well as several general fixes and other smaller-scale contributions.

3.2.1 DoG layer & convolutional variant

The Difference of Gaussians (DoG) layer is implemented similarly to a fully connected layer, with two main differences. Instead of input dimensions worth of weights per each output neuron, it only has 6[22]. And instead of using these weights directly, only reshaped, to multiply the input with, they are used to create a full difference-of-Gaussians filter first. Namely, they are separated into the shared center x, y coordinates, and weights and diameters parameters for each of the Gaussians. Then, the full gaussian filter is computed[23] with dimensionality of `(input_width, input_height, 1, number_of_channels)`, element-wise multiplied with the layer's input, and spatially sum-reduced, creating a `number_of_channels`-wide output tensor. The output is then normally processed through activation function, bias, dropout, etc.

The convolutional DoG variant works similarly, the only two differences being the size of the difference-of-Gaussians filter and its application. The filter is of size `conv_filter_widths` and, rather than being pointwise multiplied with the input, it is applied at all of its spatial locations as per the semantics of conv2D operation[24]. The same as a classical convolutional layer, it supports convolution stride and dilations. The `SAME` option is used for padding.

Unlike the weights of fully connected and convolutional layers or their direct derivatives, the weights of either classical or convolutional variant of DoG cannot be initialized using the mechanisms already provided by NDN3[25]. Namely, the Gaussians center(s) needs to fit within the spatial boundaries of the input/convolutional filter, and the width should at least be proportional to the

[20] Most additions were merged upstream but some remain only in a fork (attachment A.1.2). For more information, refer to `msc-neuro`'s `./README.md`.

[21] For description, refer to *DoG layer* (section 2.1.1).

[22] It actually reserves 8 to support not-concentric difference-of-Gaussians filters. Such behavior is off by default but can be enabled through `impl_concentric` parameter.

[23] This is likely not the most efficient implementation but given the size of our data and other bottlenecks within the system it never showed to be an issue.

[24] https://www.tensorflow.org/api_docs/python/tf/nn/conv2d

[25] Refer to *NDN3 toolkit* (section 3.1.1).

spatial dimensionality. Given that, we opted for replicating the exact initialization scheme of Antolík et al., drawing the DoG parameters from a uniform distribution with bounds proportional to the input/convolution filter size. Namely, the weights are strictly positive, widths are bounded by 1 pixel and 82 % of the maximum of input/filter width and height, and the center cannot be closer to an edge than two pixels.

3.2.2 Pearson's correlation tracking

The Pearson's correlation tracking is implemented as part of the loss function evaluation. It is computed for each output neuron separately and then averaged. Several options are available: interpreting neurons' NaN correlations as 0, ignoring NaN correlations, and – inspired by Ecker et al. [2018] – also ignoring correlations for neurons with small variation in the recorded data. All of our experiments use the `treat NaNs as zeroes` option. The correlation tracking respects data filters[26] if set, properly ignoring outputs of masked neurons.

3.3 Experiments and analysis pipeline

To enable scalable execution and evaluation of a large number of architectures implemented using NDN3, including the capability for efficient hyperparameter search, we also created[27] an experiments and evaluation pipeline. It consists of three parts, experiment execution pipeline, results and analysis tools, and general NDN3 utils.

3.3.1 Experiment execution

The experiment execution pipeline works as follows. For each experiment, two python scripts exist, a *runner* and the *experiment* itself. The *runner* iterates over all hyperparameter combinations the experiment is supposed to test and calls the experiment script with each of the combinations as command-line arguments. It does not call it directly, however, but through a backend-specific *executors*. The one to be used is user-selected when the runner script is invoked. Currently, five backend *executors* are implemented, one for a custom Docker-based[28] environment, two for qsub[29] based systems – CPU and GPU version, and two for plain virtenv python environments on windows or linux. For each of them, separate *experiment script* invocations are either submitted as individual jobs or are natively run in parallel. All of them redirect standard output and standard error to by-convention-standardized log files.

The *experiment script* can be arbitrary, but – by convention – contains instantiation and training of a specific model architecture given a particular set of (hyper)parameters (further referred to as the *experiment instance*) passed as command-line arguments, usually by the corresponding *runner*. As further explained in *Experiments* (section 4.2), each model instance is actually instantiated

[26] Refer to *NDN3 toolkit* (section 3.1.1).
[27] The code is hosted together with all experiments in `msc-neuro` (attachment A.1.1).
[28] https://www.docker.com/
[29] https://en.wikipedia.org/wiki/Qsub

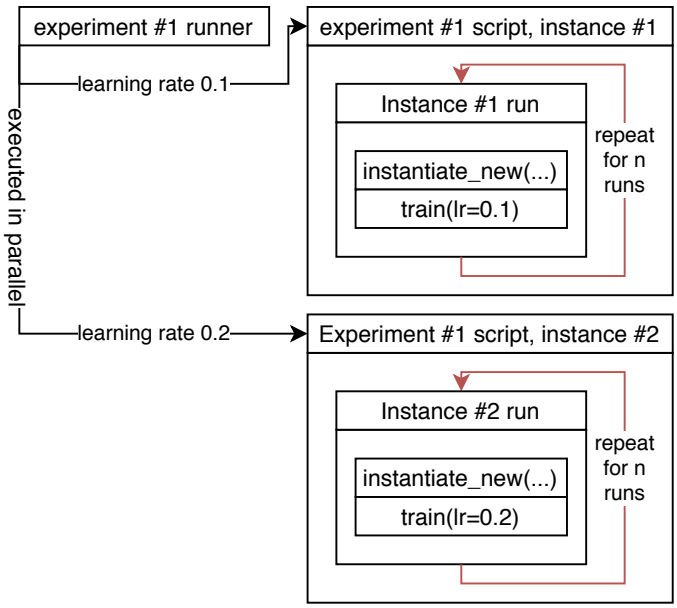

Figure 3.3: Experiments pipeline: Execution order when an `experiment #1` with two instances (learning rate 0.1 and 0.2) is run through a runner.

and trained multiple times (further referred to as the *experiment instance run*) to control for the influence of random initialization. Currently, for all implemented experiments individual *instance runs* are executed sequentially by the respective *experiment scripts*. For illustration, refer to figure 3.3.

3.3.2 Experiment analysis

Having multiple *runs* of, apart from random initialization, identical models presents a challenge for analysis. Instead of comparing performance of individual *runs*, sets of multiple *runs* need to be considered. To facilitate that, an analysis toolkit was created.

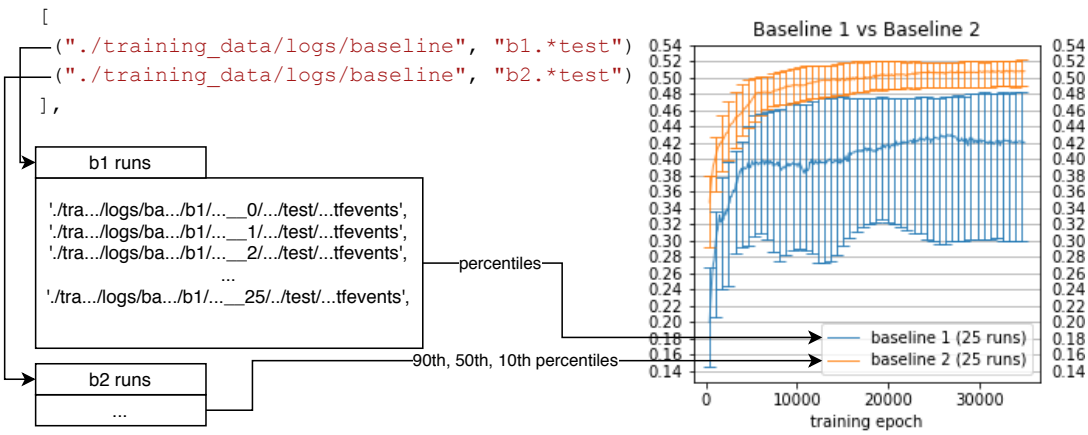

Figure 3.4: Experiment analysis toolkit: Retrieval of summaries for individual runs of two experiment instances (`b1`, `b2`).

A user can specify a list of *experiment instances* to be compared against each other as an array of root folders with TF summaries and filter regexes. Based on that, all relevant TF summaries of each *experiment instance* are gathered. For each *instance*, metrics are first read from the summaries files of all *runs*, then the values grouped by epoch number (*step*), and finally a few general statistics, such as deciles of performance, are computed across each *instance*'s individual *runs*. As the last step, the results are presented either as a graph (see figure 3.4) – showing the metric percentiles of each *instance* in time, or as a summary table – representing the final trained performance.

For more information about the specifics, such as the expected filesystem layout of the experiments pipeline or the exact API of analysis toolkit, please consult either the `msc-neuro`'s `./README.md` or the documentation that is part of respective methods' source code.

4. Methodology

In this chapter, we are first going to introduce our problem at hand – including an overview of the dataset we are going to be working with, then describe the methodologies of our architecture exploration process and results analysis, and finish with a section about our model training regime.

4.1 The system identification task

We are going to work with data recorded by Antolík et al. [2016] in mouse primary visual cortex, trying to capture the computation of retinal ganglion cells, LGN, and both simple and complex cells of V1.

- **Stimuli:** Static images of natural scenes, such as landscapes, animals, or humans, from David Attenborough's BBC documentary Life of Mammals. The images were presented to sedated mice in 256 luminance steps grayscale and downscaled to 384x208 pixels.
- **Recorded response:** The estimated average number of spikes of a neuron in a specific time window after an image stimulus presentation. Recorded by calcium imaging method and further processed by spike extraction algorithm[1]. In our case, a whole population of tens of neurons within a small region of V1 was simultaneously recorded for each stimulus.

The fact that we are working with natural stimuli is important. While the function of V1 neurons, especially simple cells, is relatively known for artificial stimuli such as white noise or small contrast lines of various orientations, the computation of complex cells on natural stimuli, containing all sorts of high and low frequency details and cross correlations, is still an open question. This is a direct consequence of the nature of simple and complex cells as intruded in *Early visual system* (section 1.1.1).

Due to the functional organization of mammalian V1, neurons recorded in a local area of visual cortex all have receptive fields within a restricted region of their visual space. Thus, there is no reason to use the entire stimulus image for response modeling. Instead, receptive fields of individual recorded neurons were estimated using a rLN model on the full stimulus and combined to form a region of interest. To account for inaccuracies, this region spanned roughly two times the area of the union of individual neurons' estimated receptive fields. Lastly, only these regions of interest were cut out of the stimuli pictures, downscaled to 31x31 pixels, and used to form our dataset's inputs.

4.1.1 Dataset

The dataset consists of three regions, each recorded separately, targeting different neurons, and with an individual set of stimuli. The regions contain 103, 55, and 102 recorded neurons respectively. Each region consists of a training set,

[1]The response is actually not a spike count but a variable directly proportional to it.

with stimuli each presented only once, as well as an explicit validation set (also referred to as the test set), where each stimulus was presented 10, 8, and 12 times respectively and the responses were averaged. The training sets contain 1800, 1260, and 1800 images for the three regions while each validation set is 50 images regardless of region. Regions 1 and 2 were recorded in one animal and region 3 with another.

	Region 1	**Region 2**	**Region 3**
Recorded neurons	103	55	102
Training stimuli	1800	1260	1800
Training repetitions	1	1	1
Validation stimuli	50	50	50
Validation repetitions	10	8	12
Animal	A	A	B

Table 4.1: Summary of regions within the dataset.

The fact that the validation sets are averaged across multiple stimulus presentations while the training set is recorded only once has consequences in terms of the measured responses' signal to noise ratio. To illustrate this: the average correlation between predicted and measured responses across region 1 neurons, as reported by Antolík et al. [2016], was between[2] 0.28 and 0.35 on the training set and between 0.41 and 0.52 on the validation set for the same models. Similarly for the other two regions, the performance on their validation sets was substantially higher than for the training sets.

The stimuli, representing 31x31 pixel 8-bit grayscale images, are stored as float64 numpy arrays of appropriate dimensionality. Inexplicably, the range of pixel values is 0-0.000255 instead of the expected 0-255. For each stimulus, the measured response is a recorded-neurons-wide floating point vector representing variables each proportional to a measured neuron's spike rate. They are stored as a float64 numpy array of respective dimensionality. The responses have a range of roughly 0-10.

4.1.2 Prior works results

Both Antolík et al. [2016] and Klindt et al. [2017] published results of several models on the Antolík et al. dataset as part of their papers. Table 4.2 shows a subset of models and their reported validation set correlations that will either be referenced later or we deem relevant for the context they provide.

Since much of our analysis is focused on stability of model performance with respect to random initialization, table 4.3 provides validation set correlations of 50th, 10th, and 90th percentiles[3] across 100 separate fittings using different initial conditions of the *HSM model*, as reported by Antolík et al. Regrettably, similar data are not available for Klindt et al. model or any other of the aforementioned models.

[2]Depending on random initialization.
[3]For the rationale refer to *Analysis* (section 4.3).

Model	1	2	3	Paper
HSM	0.51	0.43	0.46	Antolík et al.
rLN	0.31	0.24	0.30	Antolík et al.
What/Where	0.55	0.45	0.49	Antolík et al.
What/Where: fully-connected readout	0.47	0.43	0.46	Klindt et al.
Linear-nonlinear Poisson (LNP)	0.37	0.30	0.38	Klindt et al.

Table 4.2: Validation set correlation of prior works models on all three dataset's regions.

	50th	90th	10th
Region 1	0.479	0.501	0.442
Region 2	0.412	0.435	0.382
Region 3	0.437	0.449	0.419

Table 4.3: Percentiles of validation set correlations of the *HSM model* by Antolík et al. across 100 different random initializations.

4.2 Experiments

Due to the size of both hyperparameters and especially architecture space, a complete exploration that would cover all possible combinations of a model for our dataset is not possible. It is not feasible even if we assume starting off a preexisting architecture, in our case the *HSM model* by Antolík et al. [2016], and would only want to explore its close neighborhood – architecture and hyperparameter wise.

Instead of trying all combinations, we opted for the following approach. We start with a working model called and call it *baseline*[4]. With it as a base, we conduct a set of smaller exhaustive grid search *experiments*, each focused on a small set of hyperparameters or architecture aspects. After analysing these *experiments*, we combine the modifications that worked best and assess the performance of this new *baseline* model. This step is important to verify the additivity of separately tested improvements. If the new *baseline* model candidate performs better than the current one, it becomes the new *baseline*. After that, the cycle repeats with a new set of *experiments*. For future reference, the term *experiment* refers to a single focused grid-search exploration and *experiment instance* to one particular model with a specific set of hyperparameters that is, among others, assessed as part of an experiment.

Contrary to the classical deep learning assumption [Wu et al., 2017] that modern optimization techniques on real-world datasets should converge to reasonably similar levels of performance and generalization across multiple random initializations, Antolík et al. described a substantial variance[5] for different initializations and subsequent training of the same model. To control for this phenomenon, all our *experiment instances* are run multiple times, each time with a different random seed (further referred to as *experiment instance run*). To facilitate re-

[4]The first *baseline* will be a reimplementation of the *HSM model* by Antolík et al. [2016]
[5]Refer to *Prior works results* (section 4.1.2).

peatability[6], consecutive seeds starting at 0 are used.

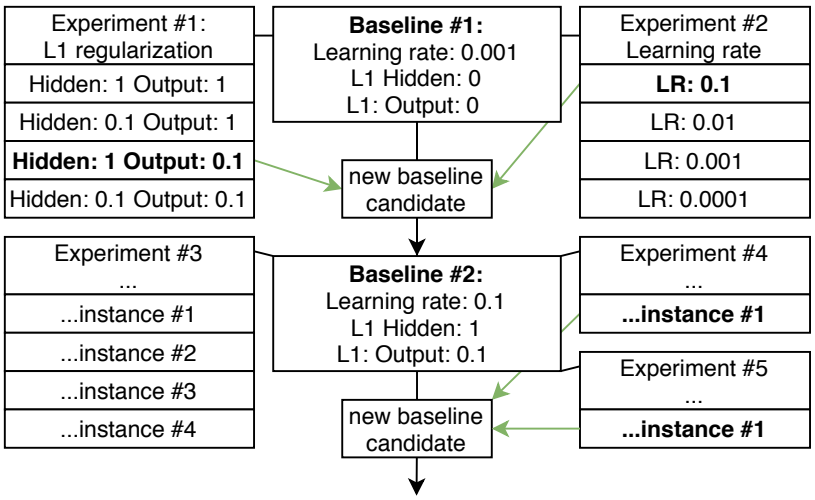

Figure 4.1: Experiments methodology: Bold instances represent better performance than respective *baseline* models. Black lines show what *baseline* model was used for an experiment. Green arrows show improvements of best instances selected to form a new *baseline* candidate.

1. `Assess the performance of a baseline model.`
2. `Run a set of experiments, each testing a modification of the baseline model.`
   (a) `Grid search experiment instances to cover all parameters of the change.`
      i. `Execute multiple runs of each experiment instance.`
3. `Take experiment instances that worked best and performed better than baseline, combine them to a new model.`
4. `Verify the new model is better than the old baseline. If it is, make it a new baseline.`
5. `Repeat, go to 2.`

The experiment exploration algorithm.

The same seeds are used across both different *experiments* and different *experiment instances*. Therefore, all other things equal, the *first run* of *instance 1* of *experiment 1* has the same seed as the *first run* of *instance 2* of *experiment 1* or the *first run* of *instance 1* of *experiment 2*. Due to that, runs of the same architecture differing only by hyperparameters that do not influence the number or order of randomly initialized parameters should be initialized with the same values.

Instead of this approach, we could have chosen to use either a fully automated or at least partially computer assisted method to search the model space, for example through evolutionary algorithms [Miikkulainen et al., 2017], Bayesian optimization [Arnold, 2019], or deep learning based autoML system [Zoph and

[6]It is important to note that TF and thus NDN3 is not entirely deterministic. For more information, refer to: https://github.com/NVIDIA/framework-determinism.

Le, 2016]. That would, however, go against our goal to test specific modifications to Antolík et al.'s model and carefully mix it with and compare to elements from traditional DNN toolkit. A comparison between our work and a more automated approach would be an interesting topic for further research but is outside of the scope of this thesis.

4.3 Analysis

As explained in the *Experiments* (section 4.2), when assessing two model instances we are not comparing two *runs*, one per each *instance*, but two *distributions of runs*. That complicates analysis as we cannot simply say that *instance A* is better than *instance B* because it achieves a higher performance metric, Pearson's correlation in our case[7], on the validation set at the end of training. Instead, we have to decide what is the representative sample out of the two distributions of runs to compare. Further, we also have to consider the variance across multiple *runs*, as it represents the stability of the model with respect to random initialization – an aspect we set out to explore. A model that has slightly better e.g. 90th percentile run but has two times as large variance between runs – leading to significantly worse 10th percentile, than the baseline model is neither clearly better nor worse, but is noteworthy.

Thus, with a few exceptions, we report three metrics per *experiment instance*. The median correlation on validation set, 90th and 10th percentiles. We chose to report these to present a clear picture of how big of a spread individual models have depending on random initialization. Together with the higher number of runs per each *experiment instance*, it also makes reasonably sure we do not present random outliers while showcasing the full potential. Unlike variance, percentiles allow for accurate representation of non-symmetrical variances[8] and directly show what exact level of performance the model instance is capable of.

It is important to note that each run is not represented by just a single scalar – its performance at the end of training, but a time series that shows the progress during training. For simplification, we will mostly consider only either the end of training or peak value, however. When relevant, we will draw explicit attention to the speed of convergence or other phenomena apparent only when the whole training in time is considered.

4.4 Training regime

We opted for a simple static and fully predetermined amount of training epochs. The reason why we did not choose early stopping instead is threefold. First, it is the simpler solution without additional hyperparameters and implementational complexity. Second, we only have limited data that is already explicitly divided into train and validation sets. Splitting either to form a test set to evaluate early stopping on might compromise either training or validation due to insufficient

[7]For reasoning refer to *Correlation as performance metric* (section 1.4.2).

[8]E.g. when half of the *runs* have good performance and the other does not train at all. In that case, the 50th and 90th percentiles would be close to each other – showcasing the potential, and the 10th would be near zero correlation.

data. Third, due to the size of the validation set, we evaluate it fully throughout the training process, capturing all relevant performance metrics[9]. Thanks to that, we always have the full information about the model's peak performance on the validation set, even if we overfit after achieving it. This leaves computational efficiency as the only benefit of early stopping, which in our opinion did not outweigh the negatives[10].

Based on early tests, we chose to train *experiments* for 5 000 epochs and *baseline* models for 35 000 epochs. This proved to be more than enough for experiment instances to show potential – if there was any, and for *baseline* models to achieve peak performance and – in case they were susceptible to it – overfit, with plenty headroom especially for later models.

In regards to the number of *runs* per model *instance*, we have chosen 10 runs for experiment instances and 25 runs for baseline model instances to balance computational resources and statistical significance of the results. The effect of potentially higher numbers of runs on one of the least stable *baseline* models can be seen on figure 4.2.

Unless explicitly mentioned, all of our experiments are evaluated only on region 1. Region 1 was chosen due to being the largest, highest performing, and least stable of the three regions – as reported by Antolík et al. In *Testing on other regions* (section 5.3.2) we evaluate the main architectures explored and fine-tuned on region 1 on regions 2 and 3 as well.

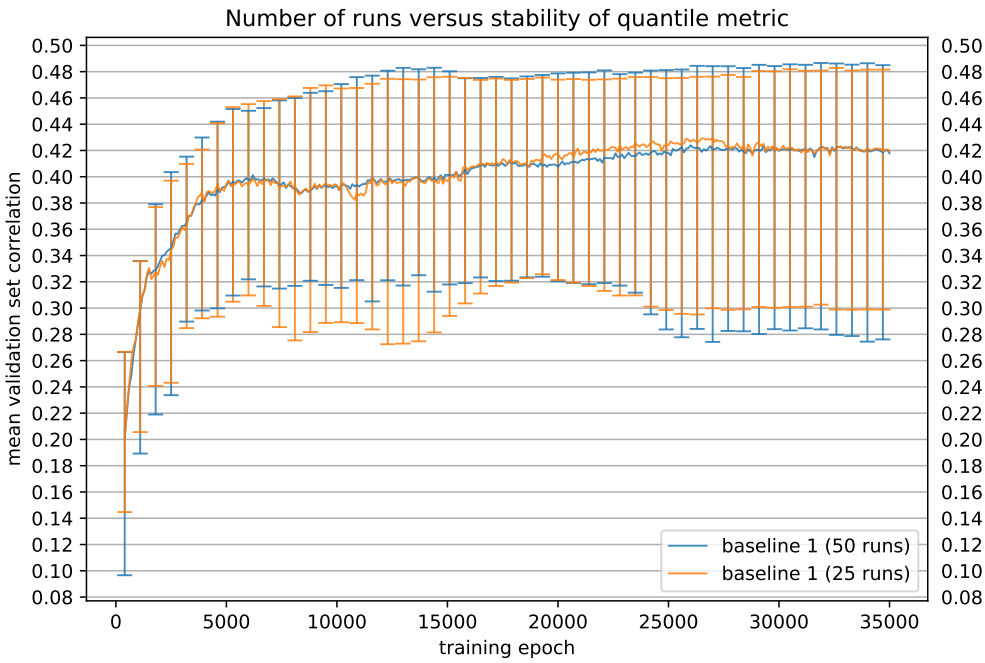

Figure 4.2: Illustrates the difference between running the least stable *baseline* model 25 and 50 times. The line represents median run, top and bottom error bars 90th and 10th percentile runs.

[9]Refer to *NDN3 toolkit* (section 3.1.1).

[10]Klindt et al. [2017] report using early stopping with a training set split of 80/20, refer to *Klindt et al. and the separable layer* (section 2.2)

5. Experiments and results

In this chapter, we will go through all notable[1] experiments and their results. It is divided into three sections. First, we focus on reimplementing, assessing, and improving the *HSM model* by Antolík et al. [2016] Second, we compare its DoG layer contribution with various less constrained variants from classical deep learning and also assess the separable layer introduced by Klindt et al. [2017]. Lastly, we explore pooling the three regions of our dataset together for a single shared model as well as test the best architectures on each of the regions separately.

5.1 Assessment and improvements of HSM

In this section, we start by listing the differences between the original *HSM model* by Antolík et al. and our reimplementation. Then, we move towards experiments focused on the effects of training regime hyperparameters on both best achieved performance and also model stability. We finish with an initial look at *soft regularizations* and additional non-linearity.

5.1.0 Differences introduced by reimplementation

We started by reimplementing the *HSM model* by Antolík et al. Due to the differences between the tools available in the general-purpose deep learning framework Theano, that was originally used by the model's authors, and the neuroscience-focused toolkit NDN3, used by us, we could not create an exact replica. Nor was exact recreation desirable due to the availability of more recent components such as modern optimizers[2].

Most notably, the originally used fitting method, the Newton Conjugate-Gradient algorithm[3], is not available in NDN3. Instead, we opted for ADAM [Kingma and Ba, 2014], a stochastic gradient descent based method introduced in 2014, and generally considered as the current go-to[4] optimizer for DNN models. This change forced us to deviate in the training regime as well. While the original model worked on full training set batches, ADAM is commonly used with appropriately sized mini-batches[5]. After a few initial tests, we empirically decided on 16 as our initial batch size – balancing expected model performance and computational efficiency. The same way, we chose to start with a learning rate of 0.0001. The last impact of a different optimization method was that unlike with Newton Conjugate-Gradient, all of our weight bounds, including those for Difference of Gaussians (DoG) layer, were enforced only on random initialization

[1] We only present a subset of conducted experiments that are implemented in the `msc-neuro` (attachment A.1.1). Further, the experiments are reordered and their naming scheme does not directly correspond between the repository and the text of this thesis. For matching, refer to `msc-neuro/experiments/readme.txt`. Similarly, we only provide a subset of figures.

[2] [Ruder, 2016]

[3] Refer to *Training regime* (section 2.1.3).

[4] Even though its dominance is being questioned [Choi et al., 2019] and more complex variants are appearing [Liu et al., 2019].

[5] [Hoffer et al., 2017], [Smith et al., 2017].

while the original optimizer restricted their values to stay within those bounds throughout the training as well.

The second difference between our reimplementation and the original model was in initialization of weights and biases. While we copied the way DoG layer weights are initialized – both in terms of bounds and distributions, for both fully connected layers – hidden and output – as well as the bias of DoG layer, we opted for the initialization schema already provided by NDN3. Instead of the original uniform distribution with relatively wide bounds, normal distribution scaled to 0.1 variance and 0 mean was used for weights, and plain zeroes were used for biases[6]. The last difference was in input. In an initial exploration phase, we discovered that the original range of stimuli data (0-0.000255) led to very poor training performance. Thus, we opted for rescaling it to what we assume was the intended range: 0-255.

5.1.1 Reimplementation

1.1.1 Initial reimplementation: Baseline 1

With aforementioned differences, the initial reimplementation, further referred to as **baseline 1**[7] model, achieved validation set correlation of 0.42, 0.48, and 0.3 for its median, 90th percentile, and 10th percentile runs by the end of the 35 000 epochs of training respectively. As shown on figure 5.1, this is worse than the fully trained results reported by Antolík et al. (0.48, 0.5, 0.44) in both top percentile performance but also the variance with respect to random initializations, demonstrating the sensitivity of DNN models to even minor changes.

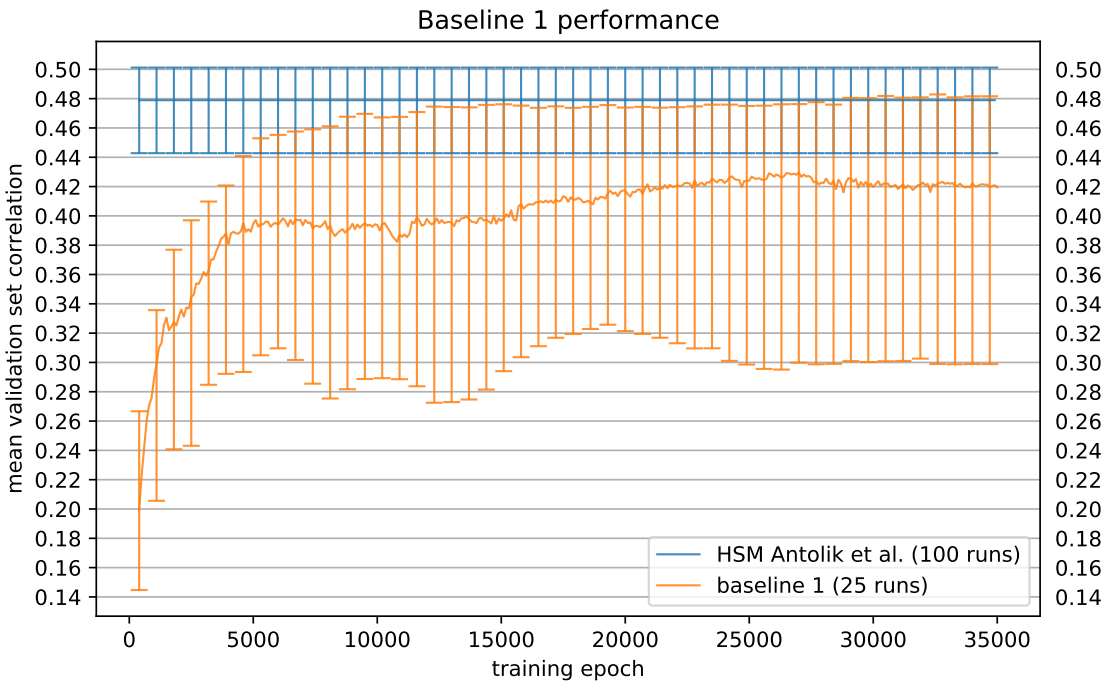

Figure 5.1: Validation set correlation of first *baseline* model versus the original Antolík et al. [2016] implementation[9]on region 1.

[6][Glorot and Bengio, 2010]

[7]For explanation what *baseline* means, refer to *Experiments* (section 4.2).

5.1.2 Training hyperparameters tuning

1.2.1 Input scaling: Baseline 2

Informed by the recommended[10] practise of data normalisation, **experiment 1.2.1** focused on input scaling[11]. Two different variations were tested. First, where only the input is linearly normalized to have 0 mean and standard deviation of 1. And second, where input is scaled the same way but the output is also scaled to have standard deviation of 1. We could not shift the mean of the output to 0 because that would cause some outputs to be negative and go against our Poisson noise assumption[12].

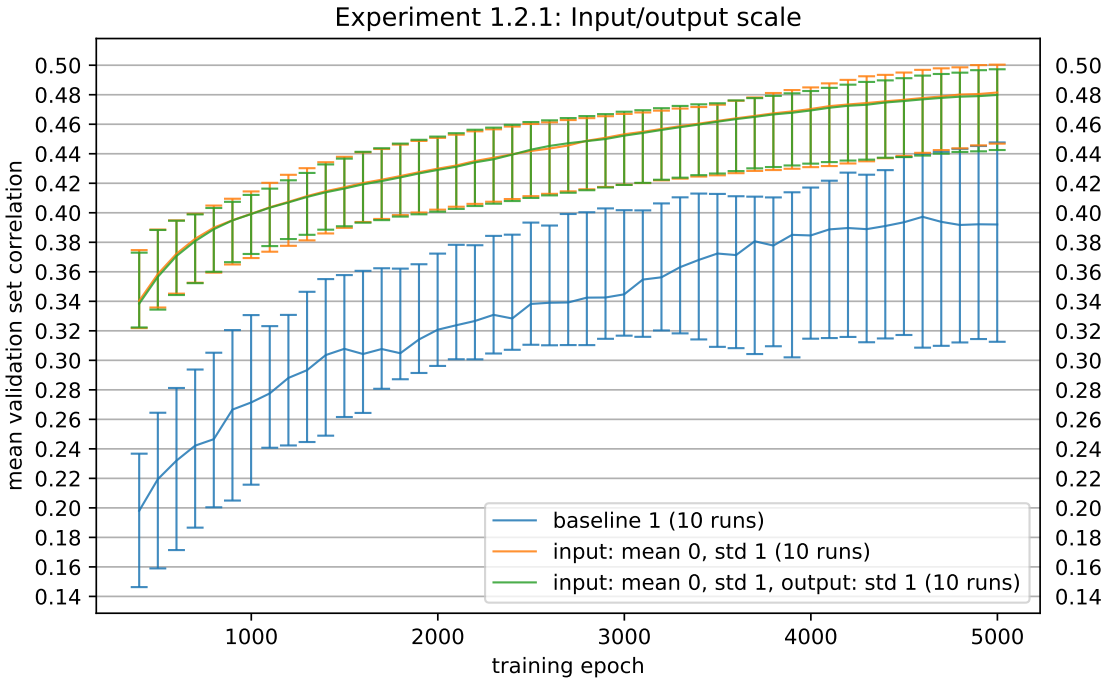

Figure 5.2: Results of *baseline 1* versus normalised inputs[13].

Figure 5.2 showcases the dramatic improvement both of these changes had in both 90th percentile run performance but especially stability. Both achieved validation set correlation of approximately 0.48, 0.5, and 0.45 for its median, 90th percentile, and 10th percentile runs respectively after only 5000 epochs[14] of training, thus improving over the original *HSM model* by Antolík et al. Out of the two, we have decided to select the simpler one, scaling only input, as our new base model, **baseline 2**. Fully trained comparison on 35 000 epochs between

[9]We only have fully trained data for the Antolík et al. *HSM model*. We thus show the final mean performance and its variance for all time points.

[10][LeCun et al., 1998]

[11]Input scaling has big influence despite the fact that both the first DoG layer and the two subsequent fully connected layers of the model can learn arbitrary linear transformation through the combination of multiplicative weights and additive biases.

[12]Refer to *Poisson versus Gaussian loss function* (section 1.4.1).

[13]Unlike the previous figure 5.1, this is restricted to 5 000 epochs and 10 runs per experiment instance. Refer to *Training regime* (section 4.4).

[14]For experiments we are only training for 5000 epochs (refer to *Training regime* (section 4.4)) and so the performance of the *baseline 1* model is not entirely converged yet.

baseline 1 and the new *baseline 2*, achieving 0.51, 0.52, and 0.49 respectively, is presented by figure 5.3.

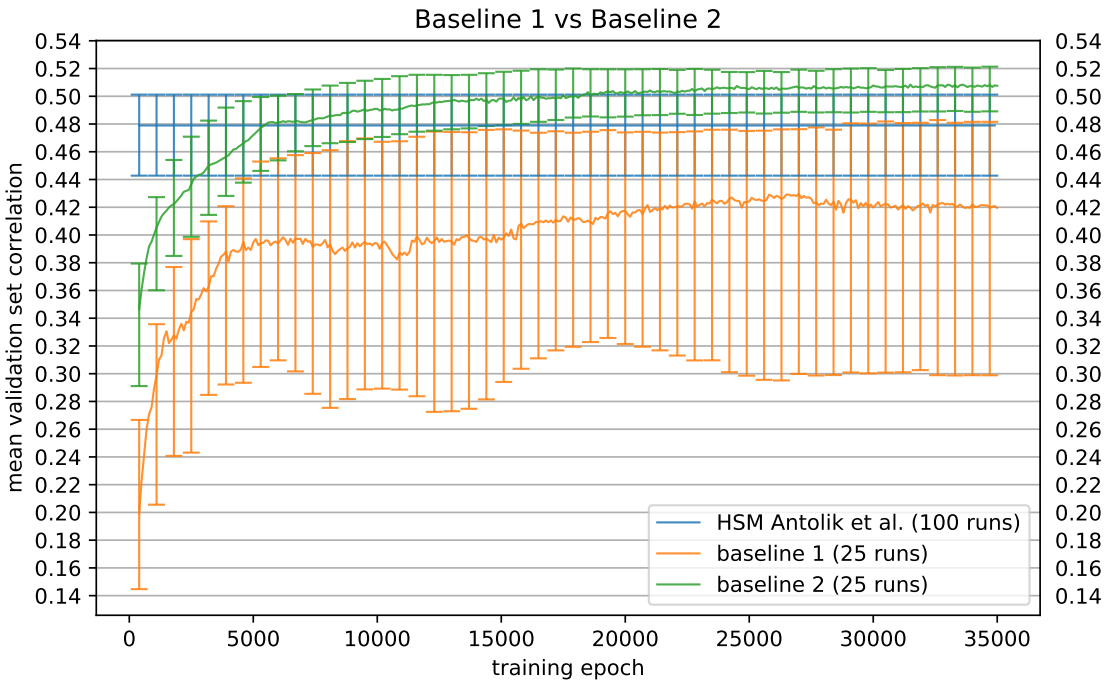

Figure 5.3: *Baseline 1*, *baseline 2*, and Antolík et al. implementation[9]. Showcasing the effect of small non-architectural changes.

1.2.2 Gaussian versus Poisson

The **experiment 1.2.2**, based on the *baseline 2* model, tested the impact of our assumptions from the Poisson versus Gaussian loss function section in terms of model performance. In line with our expectations and prior research, the version with Gaussian noise assumption loss function had slightly but consistently worse results of (0.46, 0.47, 0.42) after 5 000 epochs of training (Fig. 5.4).

1.2.3 Bias initialization

In **experiment 1.2.3**, we revisited the bias initialization scheme. Instead of the zero initialization chosen for the initial reimplementation[15] we tested truncated normal initialization[16] an absolute value of normally distributed samples with 0 mean and 0.1 variance. This was inspired by the presence of strictly positive initialization of biases in the original model. This change proved beneficial (Fig. 5.4), yielding a model with generally faster convergence, better median and 10th percentile runs, and only a slightly worse 90th percentile run of (0.48, 0.49, 0.46) after 5 000 epochs of training.

[15]Refer to *Differences introduced by reimplementation* (section 5.1.0).
[16]Refer to *NDN3 toolkit* (section 3.1.1).

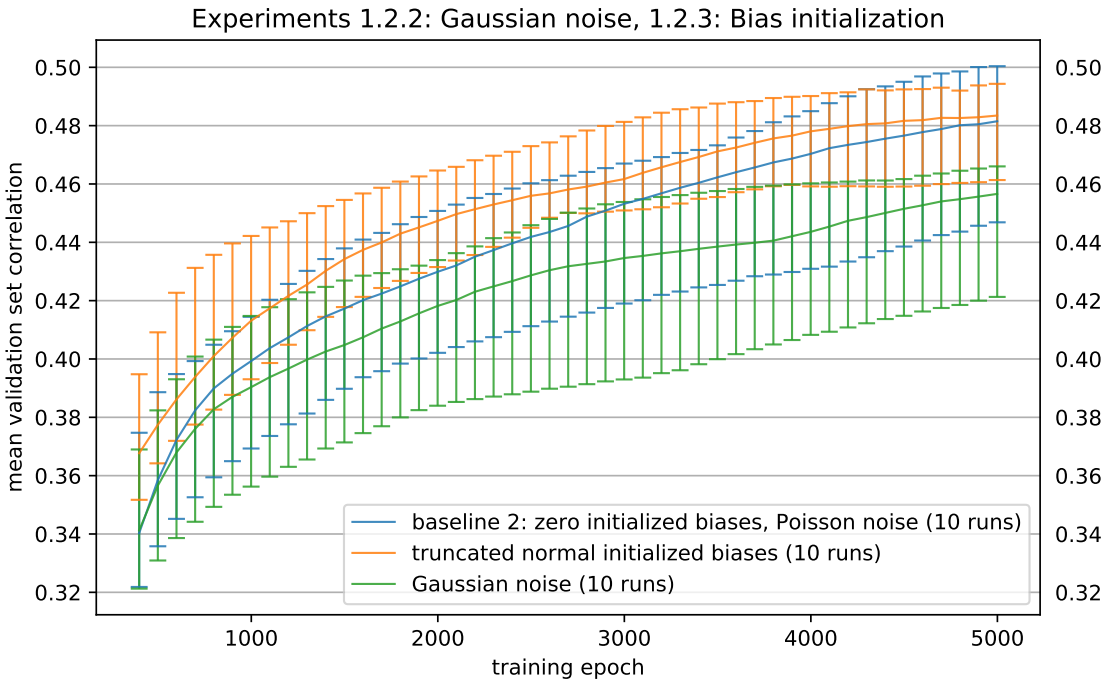

Figure 5.4: Impact of gaussian noise and truncated normal initialization of biases (experiments *1.2.2* and *1.2.3*).

1.2.4 Learning rate

The **experiment 1.2.4** built upon the foundations of our initial experimentation phase[17] and systematically tested various learning rates for our ADAM optimiser[18]. The best performance was achieved by an order of magnitude higher learning rate – 0.001. It reached correlation of (0.5, 0.51, 0.49) by the end of 5 000 epochs of training (Fig. 5.5), substantially increasing overall performance, stability, and also convergence speed over both the previous *baseline* models and the original *HSM model* by Antolík et al. Further research could be done into dynamic learning rate schedules such as exponential decay rate or step decay. Similarly, both beta and epsilon hyperparameters of ADAM could be explored[19] [Choi et al., 2019].

[17]Refer to *Differences introduced by reimplementation* (section 5.1.0).

[18]While one might assume ADAM should be relatively stable with respect to various learning rates due to its adaptivity based on first and second moments of the gradient, there is little evidence to support this.

[19]They were kept on their NDN3 defaults for our experiments.

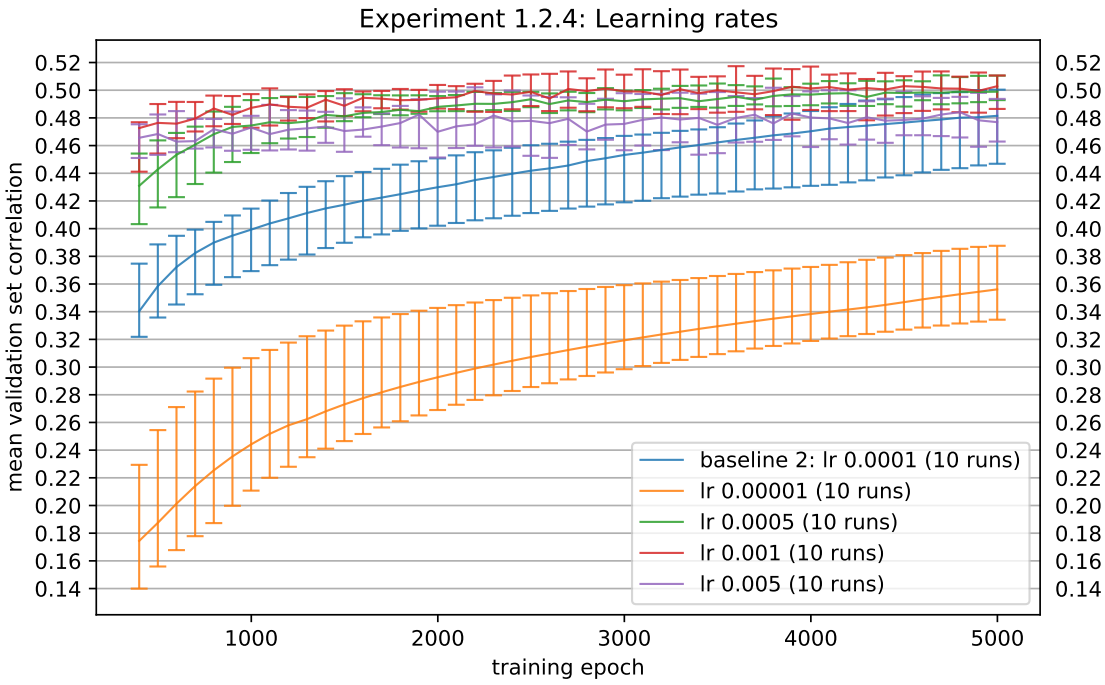

Figure 5.5: Impact of different learning rates.

1.2.5 Learning rate and bias initialization: Baseline 3

Combining improvements uncovered by *experiments 1.2.3* and *1.2.4*, we created a new base model, **baseline 3**. Upon training it fully for all 35 000 epochs, we found that the new *baseline 3* converged significantly faster and initially with a substantially smaller spread (figure 5.6).

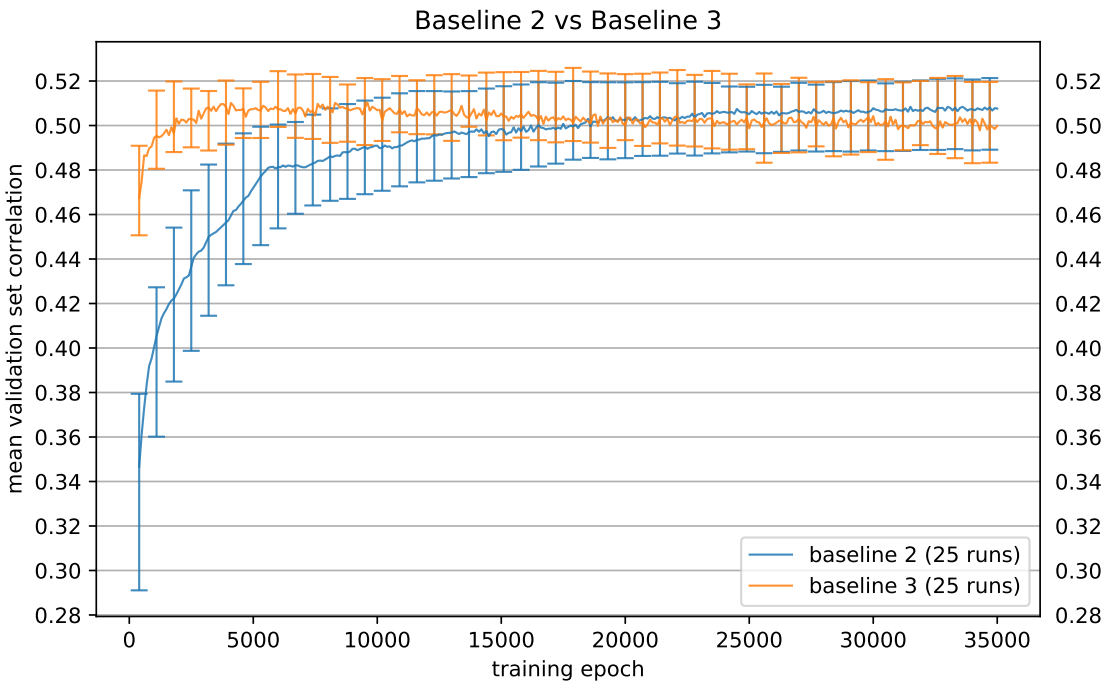

Figure 5.6: Faster convergence, initially smaller spread, and onset of overfitting.

By epoch 6 000 it reached a correlation of (0.507, 0.524, 0.499) versus (0.481,

0.5, 0.454) of *baseline 2* model. Beyond that, however, it started to overfit, eventually falling to (0.5, 0.521, 0.492) versus (0.508, 0.521, 0.49) of *baseline 2* at the end of 35 000 epochs of training.

Since the peak results were comparable and *baseline 3* showed substantially faster convergence, especially in the 5 000 epochs range – which is important for experiments evaluation, we chose to use it for all further experiments, replacing *baseline 2*.

1.2.6 Batch size

Similarly to *1.2.4*, **experiment 1.2.6** systematically reevaluated another training hyperparameter – the batch size. To ensure for fair comparison, throughout the experiment we conserved the number of model updates as opposed to the number of epochs[20]. I.e. with larger batch size and thus less updates per epoch, we scaled the number of epochs up proportionally. For smaller batch sizes, we did not decrease the number of epochs to ensure each data point is evaluated at least a certain number of times.

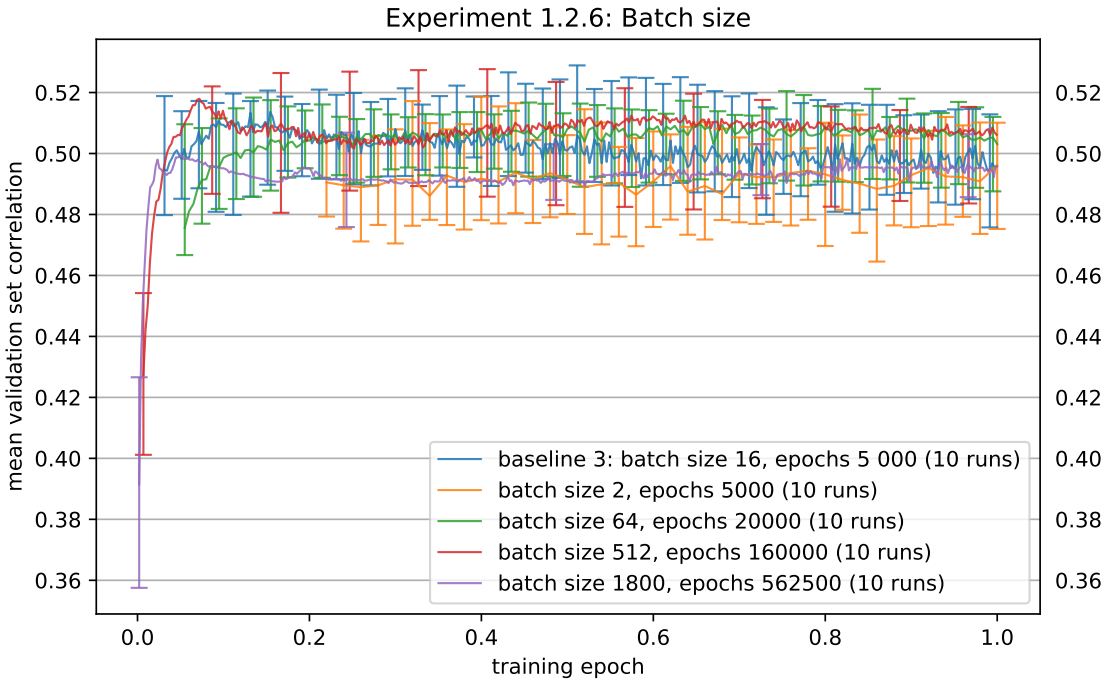

Figure 5.7: Various batch sizes. Training epochs are normalized across instances. 0 being the 0th epoch, 1 corresponding to the last training epoch for that particular instance.

Figure 5.7 illustrates that batch size had limited impact, with median run performance of all variants between 0.49-0.51 and comparable stability. Both ends of the spectrum, really small batch sizes (2, 4) and the biggest possible (spanning the entire dataset) performed slightly worse than the *baseline 3*. Otherwise, even relatively large batch sizes (512) reached good peak performance[21] and actually

[20]For more information about the relationship between the number of updates and batch size, please refer to *DNN training* (section 1.3.2).

[21]Even though not unheard of [Smith et al., 2017], it is still a relatively surprising result [Hoffer et al., 2017].

proved to be slightly more resilient to overfitting. Due to their computational requirements, however, we chose not to change our training regime for further experiments. The variance in performance between intermediately sized batch (16-128 examples) was not significant enough to draw clear conclusions from.

5.1.3 Regularizations and non-linearity

1.3.1 Dropout

Experiment 1.3.1 explored dropout[22] regularization on the hidden fully connected layer. Since dropout can decrease the effective capacity of the model, we tested three sizes of the hidden layer, the original – 20 % of the number of output neurons, 30 % and 40 %.

As Figure 5.8 illustrates, the best result was achieved by the *baseline 3* model without any dropout and with the original hidden layer size. The bigger the dropout, the worse performance. Similarly, for weak or moderate dropout, a larger hidden layer led to lower results. Only for strong dropout (50 %), a bigger hidden layer outperformed smaller one.

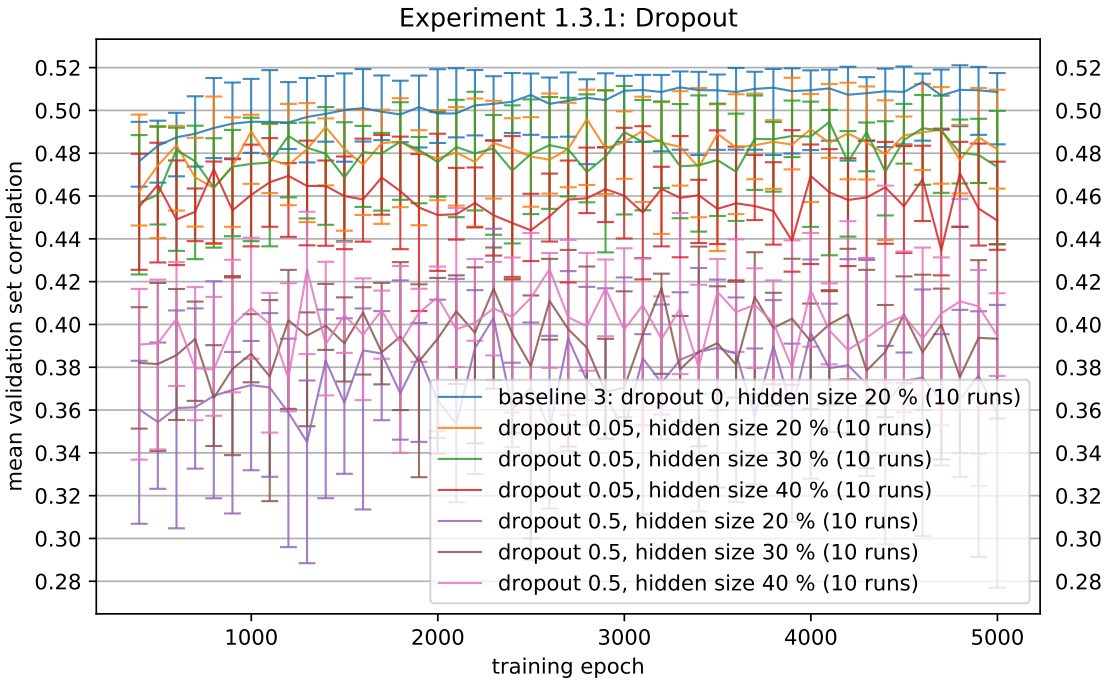

Figure 5.8: Performance loss incurred by dropout regularization (dropout probability) on the hidden layer.

Our working hypothesis for these results is that even light dropout on the hidden layer substantially changes the structure of the network every batch due to the network's relatively small size. Doing so, it causes significant instability of the gradient and thus disturbs training. This is supported by the observation that the difference in performance cannot be explained just by the smaller capacity of a layer with dropout as this should be more than mitigated by the tested larger variants. The weakest dropout setting drops just one out of the cca. 20 neurons

[22]Refer to *DNN training* (section 1.3.2).

of the hidden layer while the larger variants add cca. 10 and cca. 20 additional neurons respectively. A more thorough exploration would be needed to properly determine the cause, however.

1.3.2 L1 and L2 regularizations

Experiment 1.3.2 tested L1 and L2 regularizations on both fully connected layers together, the hidden as well as the output, in combination with bigger variants of the hidden layer. The results (Fig. 5.9) showed too strong regularization, especially L1 type, effectively prevents training. Moderate L2 mitigated the performance loss induced by a larger hidden layer and seemed to improve convergence speed even for the baseline sized hidden layer variant. L1 led to consistently worse results than L2 for this architecture.

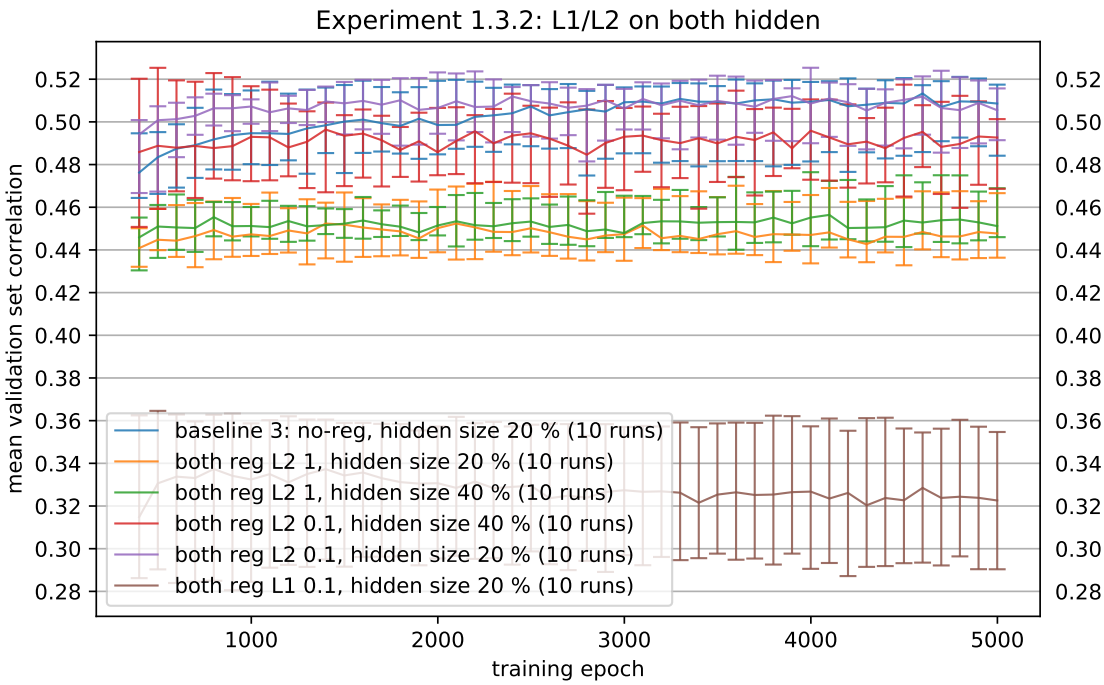

Figure 5.9: Influence of L1 / L2 regularization on both fully connected layers.

1.3.3 Separate L2 regularizations: Baseline 4

Informed by *1.3.2*, **experiment 1.3.3**, assessed different strengths of L2 regularization on the hidden and the last layer separately. To fully see the impact on both convergence speed but also overfitting – to which *baseline 3* model was susceptible to[23], we trained this experiment for full 35 000 epochs.

Figure 5.10 illustrates that moderate L2 regularization on the output layer leads to better results, especially with longer training during which it mitigates overfitting observed on previous models. The best variant reached an end of training, which was also its peak, performance of (0.513, 0.533, 0.506), improving upon *baseline 3* with (0.496, 0.511, 0.489) end of training and (0.508, 0.519,

[23]Refer to *experiment 1.2.5*

0.494) peak[24] validation set correlation. We used this instance as the new *baseline 4* model (performance comparison on figure 5.12). Furthermore, any L2 regularization on the hidden layer led to worse performance, regardless of the hidden layer size. At best the benefit of last layer regularization canceled out the drawback of hidden layer regularization, leading to *baseline 3* level of performance.

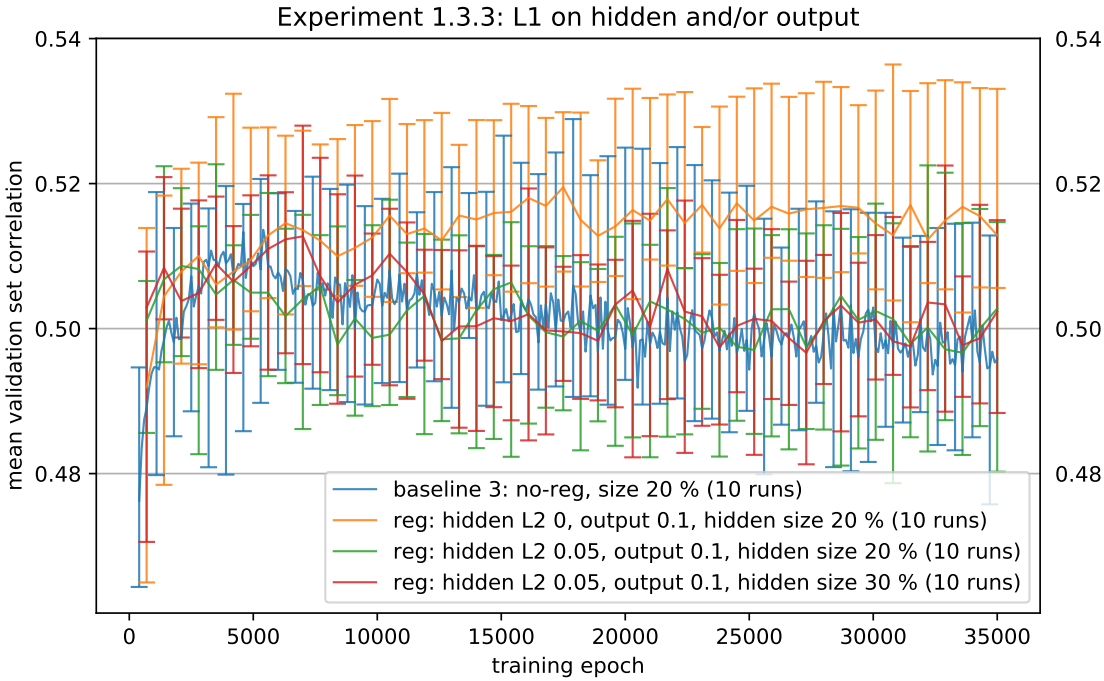

Figure 5.10: Impact of L2 regularization on the output layer with full training.

1.3.4 DoG layer non-linearity

Experiment 1.3.4 explored adding a nonlinearity after the DoG layer. While the linear activation function of the DoG layer in the HSM architecture by Antolík et al. [2016] is biologically motivated, more recent studies of V1 modeling frequently feature a cascade of convolutional filters, each followed by a non-linearity. For example, Klindt et al. [2017] – the current state of the art on our dataset, contains 3 convolutional layers and thus 3 non-linearities. Similar architecture can also be seen in Ecker et al. [2018] or Walke et al. [2018][25]. The findings of Antolík et al. proved to be correct, however, with the version containing a SoftPlus nonlinearity after the DoG layer achieving a worse result of (0.494, 0.512, 0.467) versus (0.51, 0.527, 0.502) of *baseline 4* after 5 000 epochs of training.

5.1.4 Input scaling

1.4.1 Implicit and explicit input scaling

Intrigued by the results of *1.2.1*, **experiment 1.4.1** systematically tested the impact of input data scaling and the capability of the model to train its own

[24]These numbers do not directly match those reported in *1.2.4* because they are computed using only the first 10 runs to enable direct comparison with other experiment instances. Meanwhile, *1.2.4* reports numbers from 25 runs as it is a *baselines* comparison.

[25]More information in *Related works* (section 2.3).

input transformation. We tested 3 ways of preprocessing the data in combination with an automatic, yet explicit, trainable scaling added to the model. The three modes of preprocessing were: the original scale (0-0.000255), our initial scale (0-255), and the – since *baseline 2* used – normalisation to 0 mean and 1 standard deviation. Each variant was assessed with and without an additional *linear scale layer*[26] prepended as the first layer of the model. The experiment was conducted with a *baseline 3* model[27].

The idea behind the *linear scale layer* was that if there was an explicit layer with just two parameters that is only capable of linear transformation of the whole input at once, it should be able to learn any scale relatively easily. Thus achieving close to or exceeding the *baseline* performance, regardless of the input data preprocessing. And if not that, it at least should not lead to worse results as it should be able to stay at its initial parameters, which were initialized to 1 for the multiplicative weight and 0 for the additive bias.

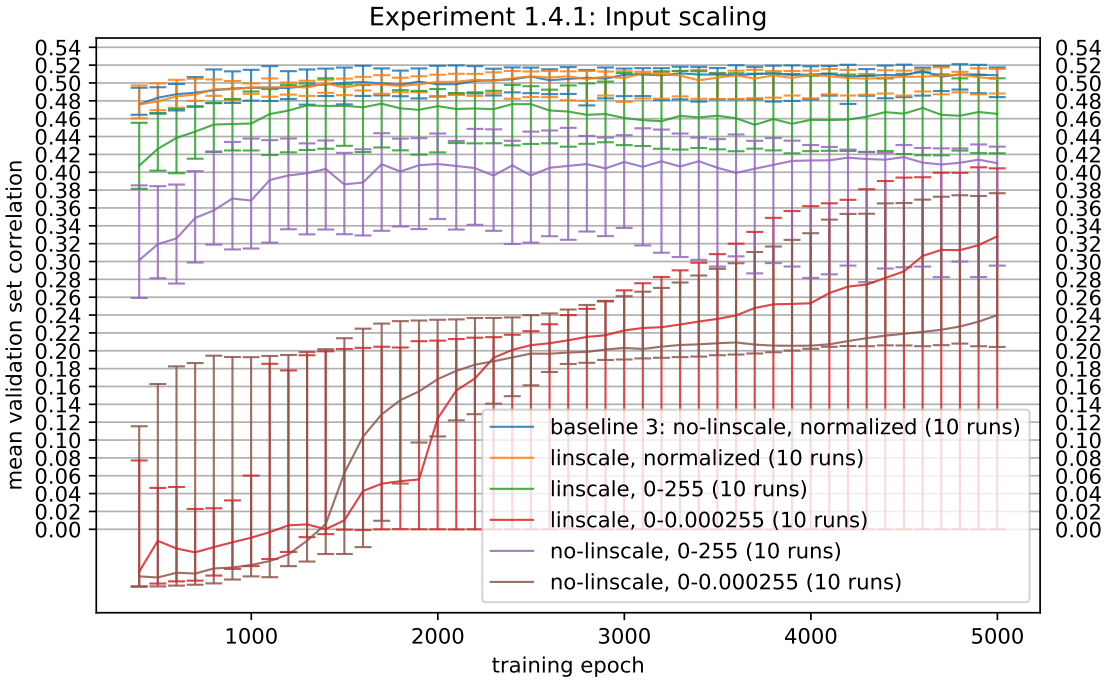

Figure 5.11: Impact of input scaling and explicit *linear scale layer*.

As illustrated by figure 5.11, adding an explicit *linear scale layer* improved the results of both *not-normalized versions* of input preprocessing. The *integer variant* (range 0-255) achieved levels of performance higher (0.465, 0.505, 0.421) than its *non-linear-layer-scale-layer* variant (0.41, 0.429, 0.295) but lower than the normalised *baseline* (0.509, 0.517, 0.484). Thus not proving our hypothesis that the *linear scale layer* might effectively learn any transformation – even the normalisation. Similarly, for the *not pre-processed variant* (0-0.000255), the explicit *linear scale layer* enabled large gains in both 50th and 90th percentile runs

[26]Refer to *NDN3 toolkit* (section 3.1.1).

[27]This is a consequence of not presenting the results in the order the experiments were conducted. We, however, believe it does not diminish the significance of the results.

but did not improve on the *baseline*[28]. Notably, its 10th percentile was worse, hinting at the possibility that for extreme distributions of the input data additional free parameters with large effect can lead to unrecoverable training when paired with certain initializations.

For already *normalized input*, both variations with and without the prepended *linear scale layer* fared indistinguishably. That does not mean normalizing is the best possible scale for our data. As shown by this very experiment, our *linear scale layer* achieves sub-par results, and so the absence of a better result with it versus without it does not prove anything. But it at least suggests normalizing might be close to a decent local optimum for our model, an observation in line with literature [Goodfellow et al., 2016], [Jin et al., 2015].

5.1.5 Discussion

That concludes the initial set of experiments focused on training hyperparameters, fully connected layers regularization, and generally smaller changes to the original HSM architecture by Antolík et al.. It showed a few things. First, non-architectural hyperparameters such as learning rate or input data scale can have substantial impact and should always be thoroughly tested. Similarly, a single additional non-linearity can have negative influence despite it being part of related state of the art architectures.

We also observed that both the median performance and the stability of a model with respect to parameters initializations can differ widely, even for the same or very similar architecture. Through non-architectural changes only[29], we managed to decrease the difference between 90th and 10th percentile runs from 0.18, with *baseline 1* model, to 0.03 of *baseline 4*, while also increasing the median run performance by 0.05 (Fig. 5.12, table 5.1). As a byproduct, this also revealed that analysing a single run with a particular random initialization might paint an incorrect picture. A problem best shown by the 10th percentile run of an *experiment 1.4.1* instance – that did not train at all while its 50th percentile run fared decently, but by far not limited to only our model or dataset [Madhyastha and Jain, 2019].

Percentile:	**50th**	**90th**	**10th**	**50th**	**90th**	**10th**
Epochs:	**5k**	**5k**	**5k**	**35k**	**35k**	**35k**
Baseline 1	0.395	0.449	0.299	0.42	0.48	0.3
HSM Antolík et al.	-	-	-	0.48	0.50	0.44
Baseline 2	0.472	0.499	0.444	0.508	0.521	0.49
Baseline 3	0.507	0.517	0.498	0.5	0.521	0.492
Baseline 4	0.511	0.527	0.495	0.514	0.531	0.498

Table 5.1: Evaluation set correlation on region 1 across *baseline* models using 25 runs. Performance at epoch 5 000 and 35 000 is shown[31].

[28] We are aware that the original scale version with an explicit input scale layer does not show its full potential within the 5 000 epochs of training. There is, however, no reason to believe its peak performance would be substantially higher.

[29] Adding an L2 regularization on the hidden layer influences only parameters fitting, and so we do not consider it an architectural change in this context.

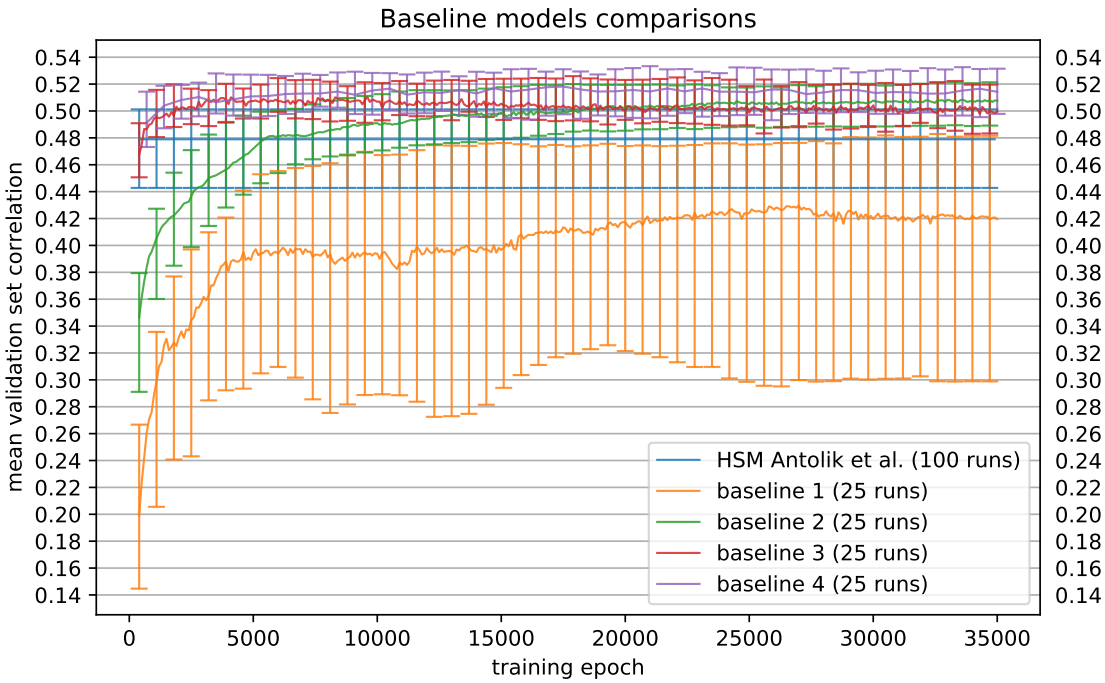

Figure 5.12: Evaluation set correlation on region 1 across *baseline* models[32].

Lastly, we significantly improved convergence speed. While the original model by Antolík et al. did 1 800 000 000 datapoint evaluations[33], our model managed with 1/200th of that: 9 000 000, assuming training for 5 000 epochs – the setting we use to evaluate experiments. If we limited ourselves further, to only 1 000 epochs – still getting a decent performance of (0.514, 0.531, 0.498) on *baseline 4* model, it would only need 1/1000th of data-point evaluations. While less impressive, our model is better even in terms of the number of model updates – with requiring roughly half of the original (1 000 000 vs 565 000[34]) implementation.

5.2 Variations on the filter and hidden layers

In this section, we examine larger architectural changes of the HSM architecture, focusing on its first two layers – *filter* and *hidden*. We compare biologically inspired techniques that feature *hard regularizations* to more generic methods from classical deep learning with *soft regularizations*. We start with fully connected layers, drawing parallels to classical LNLN and LN models, and then move to convolution layer, convolutional variants of the DoG layer, and also the separable layer introduced by Klindt et al. [2017]. Unless explicitly specified, all experiments in this section were conducted with a *baseline 4* model.

[31]For the Antolík et al. model we only have the end of training data.

[32]We only have fully trained data for the Antolík et al. *HSM model*. We thus show the final mean performance and its variance for all time points.

[33]100 epochs, each with 10 000 evaluations, using the whole dataset (1800 in case of region 1) batch.

[34]Assuming 5 000 epochs and batch size 16.

5.2.1 Fully connected models

2.1.1 LNLN model

Experiment 2.1.1 explored replacing the first two layers, the DoG filter and the fully connected hidden, with a single fully connected layer. This created an architecture equivalent to an LNLN model[35]. Two fully connected layers, the first serving as a set of filters and the second predicting neural response through per output neuron combination of said filters' outputs. The same as in an LNLN model, both fully connected layers were followed by a SoftPlus nonlinearity. Inspired by the *hard regularization* properties of the DoG layer[36], the filter layer featured a Laplacian regularization to ensure spatial smoothness of the filters and the output layer an L2 regularization, as it showed to be beneficial in *experiment 1.3.3*.

This experiment was not just to test a completely different architecture. Since the DoG layer in the original HSM architecture is not followed by a non-linearity (Ex. *1.3.4*), each output of its second layer is just a linear combination of the results of its first layer's filters followed by the second layer's non-linearity. The specific linear combination for each of the second layer's outputs is dictated by the second layer's weights. That, however, means we can replace the first two layers of an HSM architecture with a single fully connected layer, whose each filter is an appropriate linear combination (based on the second layer's weights) of difference-of-Gaussians filters (first layer's filters).

While this means, we can construct an LNLN model that can represent any computation the HSM architecture – and thus our *baseline* models – can, a question remains whether the biologically inspired *hard regularizations* brought by the DoG layer and the factoring into two separate layers – filter and hidden – are not necessary for efficient model training. We tested various strengths of both regularizations in combination with two differently sized sets of filters, 10 % of the number of output neurons and – to maintain equivalency with the first two layers of the *HSM model* – original 20 %.

[35]For more information refer to *Classical models* (section 1.2.2).

[36]The resulting difference-of-Gaussians filters are by construction spatially smooth.

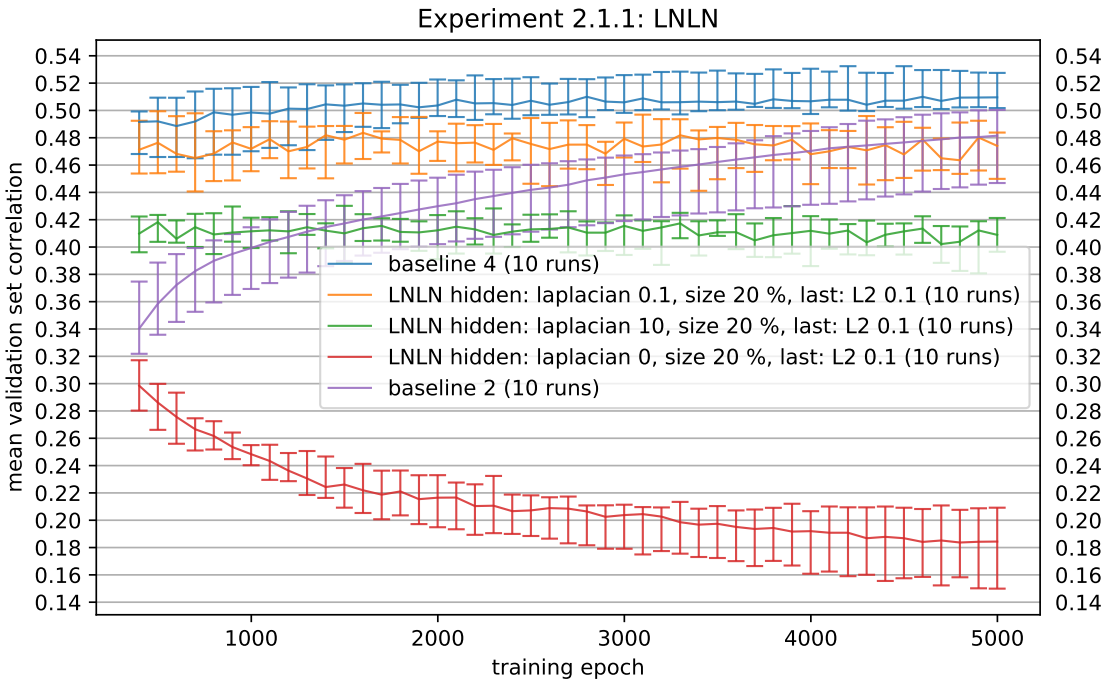

Figure 5.13: Selected LNLN instances versus *baseline 4* and *2*.

The best instance achieved validation set correlation of (0.474, 0.484, 0.45) after 5 000 epochs (Fig. 5.13). Despite being worse than *baseline 4* (0.51, 0.527, 0.502), it was comparable to less fine tuned[37] versions of our model, for example the *baseline 2* (0.482, 0.5, 0.447). We believe this result is good evidence for the hypothesis that the biological constraints of HSM only serve as well fitting explicit regularizations that make the model more stable with respect to hyperparameters, but are not a necessity to achieve good results. A finding already supported by the state of the art results achieved by Klindt et al. [2017] with a relatively more generic architecture.

A few more noteworthy observations. Moderately strong Laplacian regularization on the hidden layer was paramount. Without it, the model started overfitting very quickly and never recovered. The smaller hidden layer version, with the number of filters equivalent to 10 % of the number of output neurons, had very similar performance characteristics to the larger version but was worse across the board. Too strong regularization, both on the hidden and the output layer, also caused lower performance.

2.1.2 LN model

Experiment 2.1.2 tested an architecture equivalent to a simple LN model. The first and only layer featured a spatial Laplacian regularization, with its strength being the only free hyperparameter of the architecture.

[37]While we tried to find optimal hyper-parameters, we did not have enough time to be confident saying this is the best or even close to the best result possible with this architecture. Significantly more time was spent on properly tuning the HSM architecture.

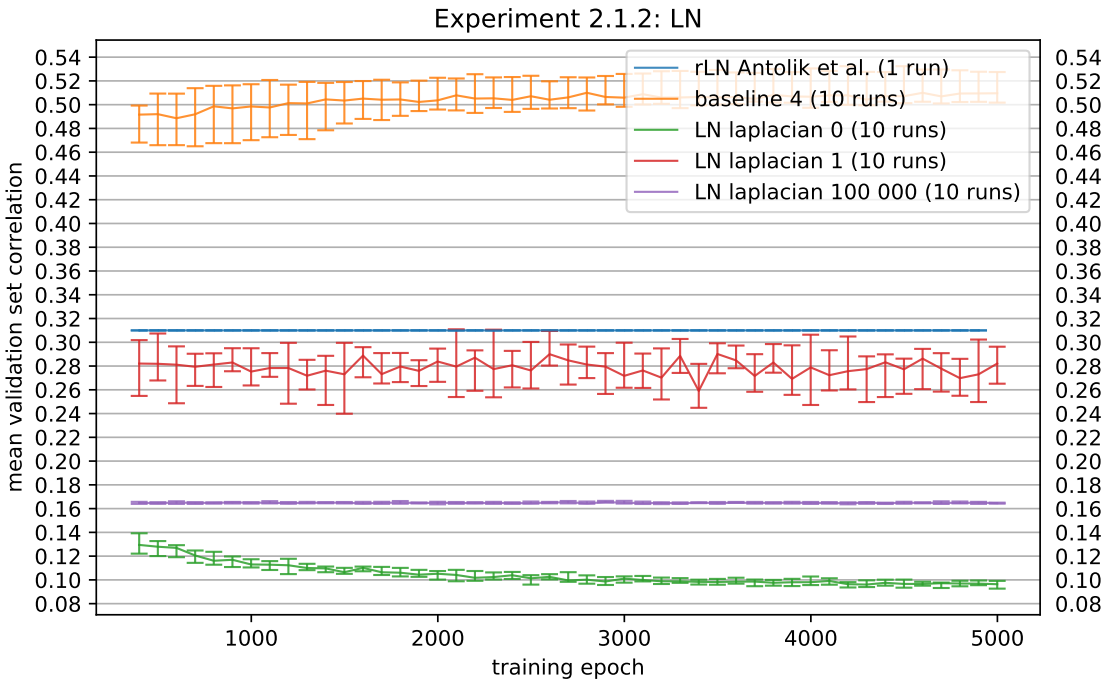

Figure 5.14: Performance of LN-like model instances. Only a fully trained performance is available for the *rLN model* reported by Antolík et al.[39].

As shown on figure 5.14, the results (0.282, 0.296, 0.265) were substantially lower than those of our previously tested and more computationally complex models and on-par with the performance of an analytically optimized version of an *rLN model* with Laplacian regularization reported by Antolík et al. [2016] (0.31). The same as in the previous experiment, some regularization proved to be a necessity while too strong led to poor, albeit very consistent, performance.

5.2.2 Convolutional and separable models

2.2.1 Convolutions instead of DoG

In **experiment 2.2.1**, we replaced the first DoG layer with a normal convolution layer featuring a Laplacian regularization. The rationale was twofold. First, we were inspired by the success of convolution based architectures as reported by Klindt et al. [2017], Ecker et al. [2018], Walke et al. [2018], etc. Given this inspiration, we reintroduced a non-linearity after the convolution layer. Second, much like in *experiment 2.1.1* with a fully connected layer, a convolution layer should be able to learn to represent anything the DoG layer can and as such might be viewed as just a more generic and less biologically constrained equivalent. We tested various sizes of the convolution (3 to 15 pixels), numbers of its filters (9 to 30), strengths of the Laplacian regularization, and additional L2 or max[40] regularization of the hidden fully connected layer. Convolution layer stride was set to half of the filter size.

[39]We only have fully trained data for the Antolík et al. *rLN model*. We thus show the final mean performance and its variance for all time points.

[40]For details, refer to *NDN3 toolkit* (section 3.1.1).

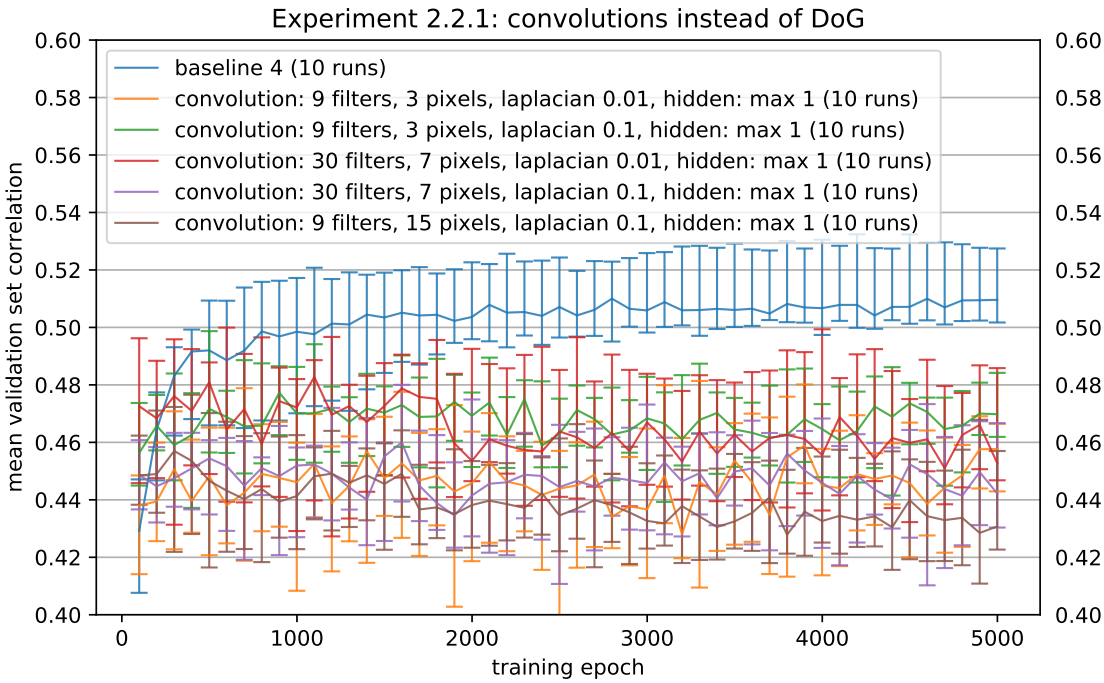

Figure 5.15: Convolution layer with regularization instead of DoG layer.

The results (Fig. 5.15) were similar to those of the LNLN model experiment (*2.1.1*), with the best model achieving validation set correlation of (0.47, 0.484, 0.462) by the end of training. Both instances with 9 filters and 30 fared very similarly. For filter size, both small, 3 pixels wide, and moderate, 7 pixels wide, convolutions reached decent performance, with large filters of 15 pixels being significantly worse (0.431, 0.457, 0.423). Strong max regularization on the hidden layer proved to be a necessity for high performance. In contrast, L2 overwhelmed the training even on the lowest tested strength.

Some observations were hard to interpret, such as the interaction of Laplacian regularization on the convolution layer and the layer's filter size. While a stronger variant worked better on a smaller convolution layer with 9 3-pixels wide filters, it worked – by roughly the same amount – worse on a larger 7-pixels wide 30 filters instance. A result that is the exact opposite of what we would expect, given the intuition that larger and more numerous filters have more opportunity to overfit and so might require stronger regularization. We hypothesize that, as regularizations in NDN3 are not normalized to layer size, the larger the layer, the higher the chance a regularization on it will overwhelm the loss function and thus negatively impact training. On the other hand, for the largest convolution filters – 15 pixels wide, stronger Laplacian regularization proved to be beneficial again. A more thorough investigation of this issue would be needed to draw conclusions. At least some Laplacian regularization on the convolution filters was a necessity, however. Without it, the best model reached (0.242, 0.281, 0.209) with clear signs of overfitting.

2.2.2 Convolutions and separable

Experiment 2.2.2 built on the architecture of *2.2.1*, only replacing the hidden fully connected layer with a *separable layer* as it was introduced by Klindt et al.

[2017][41, 42]. This test was motivated by the improvements reported by Klindt et al. where the separable readout layer increased validation set correlation on region 1 from 0.47 to 0.55 over a fully connected variant. Due to unavailability of max regularization for the separable layer, this test assessed L1 and L2 regularizations for the hidden layer.

These changes led to better results (Fig. 5.16), with the best instance reaching peak of (0.494, 0.51, 0.462) and end of training performance of (0.474, 0.499, 0.461). In contrast to the fully connected variant from 2.2.1, small convolution filters had worse performance than larger ones – with 3-pixels wide being substantially less effective. This was a surprise as we only changed the readout mechanism of the hidden layer and as such did not expect the interactions of the first layer to be different. The number of filters had, as in the previous experiment, little effect.

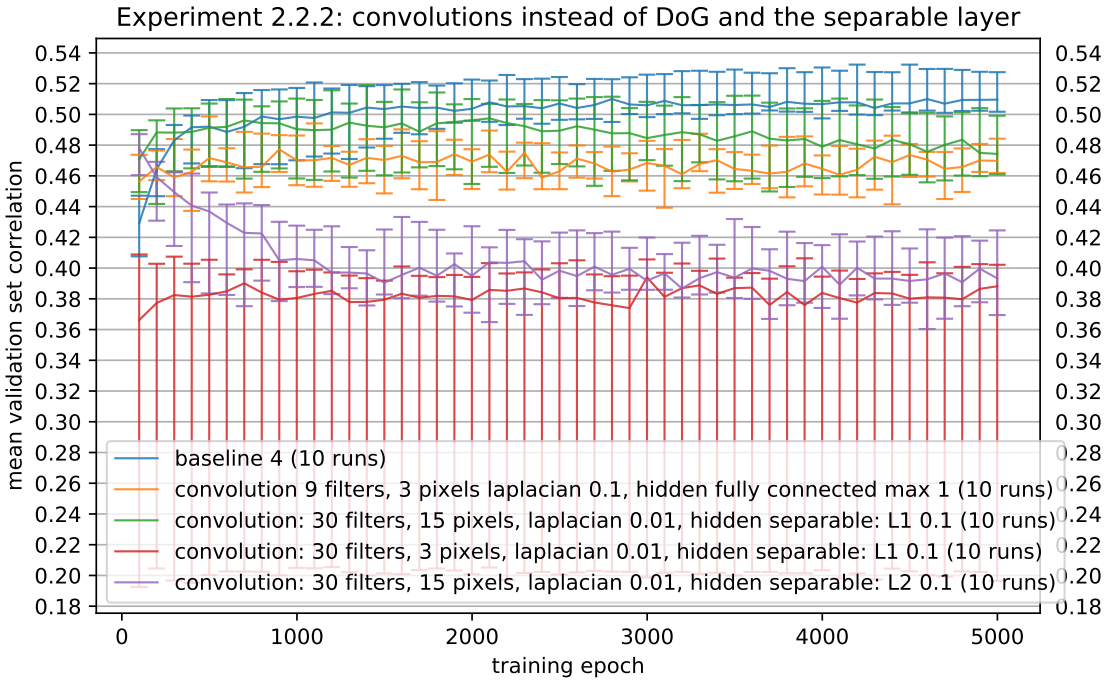

Figure 5.16: Convolution and separable hidden layer.

We also observed another instance (see experiment 2.2.1) of non-additivity of regularization benefits. Moderate L1 on hidden and Laplacian on the convolution worked best, followed by either of them stronger and the other weak, with both of them strong as the least performant variant. L2 regularization on the hidden layer led to significantly worse results across the board.

2.2.3 Convolutional DoG

With **experiment 2.2.3**, we tested two further variants of this architecture. Both featured a convolutional variant of the DoG layer as the first filter layer and

[41] For details, refer to *Klindt et al. and the separable layer* (section 2.2).

[42] There are still large differences between the *2.2.2*'s architecture and Klindt et al. [2017] model. Namely in the number of sequential convolution filters (1 vs 3), absence of batch normalisation, different regularization, and two-layer readout (hidden and output) instead of just a single separable layer.

had either a fully connected or separable layer as the hidden layer. The tested variants were the same as in *2.2.1* and *2.2.2* with the obvious lack of Laplacian regularization on the filter layer – as the convolutional variant of DoG implicitly features a *hard regularization* of its filters.

The rationale behind a convolutional variant of the DoG layer was as follows. Optimizing the locations of centers of difference-of-Gaussians within the DoG layer through gradient descent is not optimal. If the random initialization puts the center completely outside of where it should be, it is unlikely[43] the gradient for respective location parameters overweights the rest and changes the position substantially[44]. At the same time, the locational invariance of V1 neurons leveraged through convolution layers by Klindt et al. and Ecker et al. should work with explicitly parameterized difference-of-Gaussians filters just as well as with generic filters.

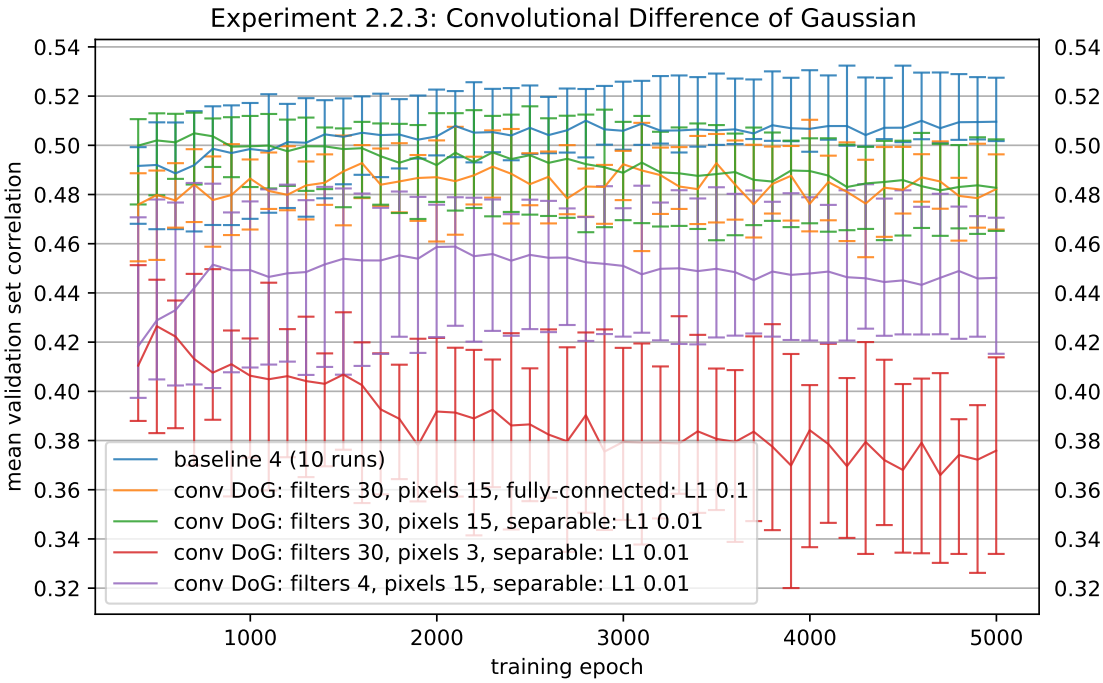

Figure 5.17: Convolutional variants of the DoG layer.

Both versions, with a fully connected as well as a separable hidden layer, achieved a similar level of performance (Fig. 5.17), (0.482, 0.496, 0.466) and (0.483, 0.502, 0.465) respectively, that was comparable to a number of recent models such as the convolution variants (*2.2.2*). The best instance of the separable version reached a peak of (0.498, 0.51, 0.484) and showed signs of overfitting. For both versions, small 3pixels wide DoG filters did not perform well[45], with

[43]Based on our very limited testing of learning properties of the DoG layer. To draw any conclusions, we suggest further research on the interaction between gradient descent based optimizers and the DoG layer.

[44]Fortunately, given the number of filters and size of our input images, this never proved to be an issue on our dataset.

[45]Similar to the separable version of convolutional variant (*2.2.2*) but unlike the fully connected version of convolution variant (*2.2.1*). A plausible explanation is that 3x3 pixels is not enough to fully realize a difference-of-Gaussians filter.

the widest 15pixels getting the best results. The number of filters had, as in the previous experiments, relatively low influence with 30 filters achieving initially higher results and then falling, likely due to overfitting.

Notably, the instances with only 4 filters fared substantially worse than their 9 filter counterparts. This was a relatively surprising result. Since 9 filters were enough for a non-convolutional DoG variant (*baseline* models), where each filter contains a single difference-of-Gaussians at a specific location, we assumed that the convolutional variant that applies each difference-of-Gaussians filter at multiple locations might work with a smaller number of filters. In essence, we assumed that the computation requires a very few types of filters applied at multiple locations instead of a very specific filter for each of relatively few locations. This result, instead, suggests nearby neurons within V1 indeed pool from a rather limited number of LGN inputs that are backed by very specific locations and receptive field topologies within the visual field. We suggest further investigation of the trained weights of this architecture and comparison to the fully connected versions as well as the original HSM version with plain DoG layer.

For the fully connected versions, both L1 and max regularizations worked similarly. The separable version performed best with a light L1 regularization, with L2 overwhelming the training.

2.2.4 Convolutional DoG without non-linearity

Experiment 2.2.4 investigated the effect of the nonlinearity after the convolutional DoG filter layer on both variants, with a separable as well as fully connected hidden layer. In contrast to convolutional models by others such as those reported by Klindt et al. [2017] or Ecker et al. [2018], where multiple nonlinearities worked well, for our architecture the variants without a nonlinearity achieved slightly higher results.

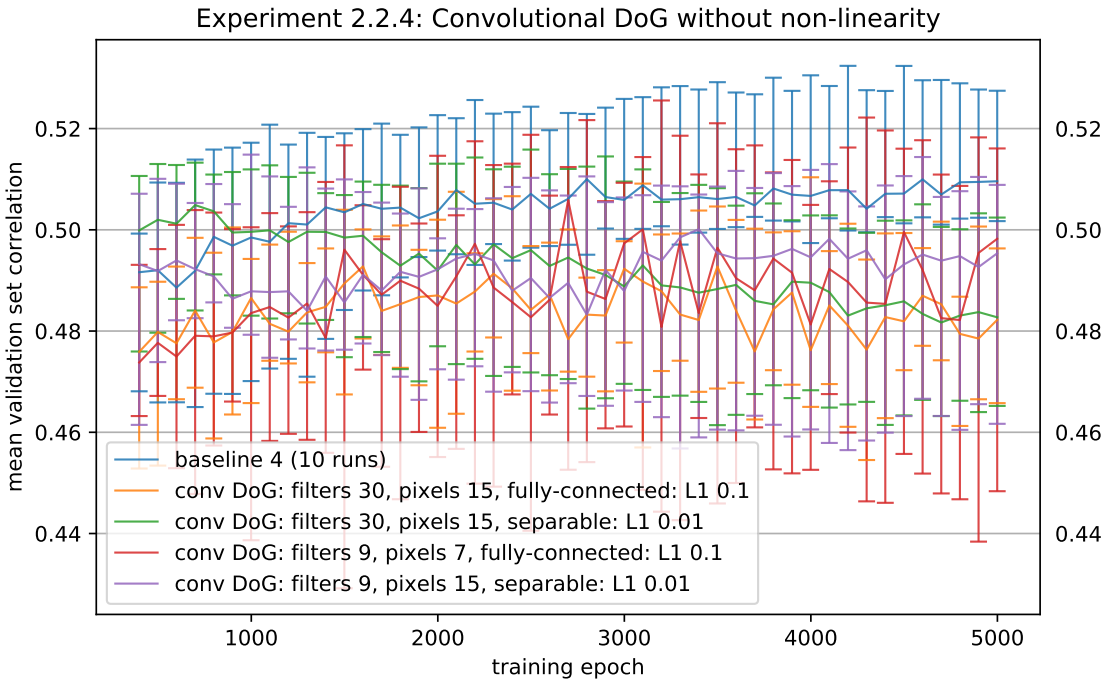

Figure 5.18: Filter layer non-linearity and the convolutional DoG variant.

While still lower than our *baseline* model, the linear version of a fully connected hidden layer variant achieved (0.498, 0.516, 0.448) versus (0.482, 0.496, 0.466) of the nonlinear version (Fig. 5.18). And the separable version reached (0.495, 0.509, 0.462) versus (0.483, 0.502, 0.465) of its nonlinear counterpart. Surprisingly, a variant with 7 pixels wide convolutions performed best for the linear activation function version of a separable variant, while 15 pixels filters worked best for all other versions. As in previous experiments, we invite further research into the exact form of the trained filters. Informed by these and *1.3.4* results, we also suggest investigating the exact effects of the number of nonlinearities in the model across various architectures.

2.2.5 Reimplementing what/where model

Finally, with **experiment 2.2.5** we tried to reimplement the what/where model architecture as introduced by Klindt et al. [2017]. Because the exact values of a number of model hyper-parameters were not reported in the Klindt et al. [2017], we decided to grid-serach them. Namely, we tested various regularization strengths, convolution strides, and also used our training regime.

Further, we replicated the architecture using only techniques already available in NDN3, which forced us to deviate from the original in a number of ways. Instead of the in-NDN3-unavailable group-sparsity regularization on the 3 convolutional layers cascade, we tested max and L2 regularizations. For the same reason, we had to leave out batch normalisation which is originally part of each convolutional layer of the model. Generally, our version was relatively different in terms of things that influence training but was close inference architecture wise.

The best instance reached peak validation set correlation of (0.48, 0.497, 0.461) and (0.479, 0.494, 0.455) at the end of training (Fig. 5.19). A result on par with our shallower convolution variant of the HSM architecture (experiment 2.2.2) but still below both our *baseline 4* model and especially the published result of 0.55 of the original Klindt et al. version. This indicates that the specific group sparsification, and possibly batch normalization, might be key for achieving the superior results reported in Klindt et al. [2017]. As this was a sideline experiment and not the focus of theis thesis, we did not dedicate further resources to properly reimplement the model[46]. However, we still believe it can serve as evidence that hyperparameters and only training influencing parts of the model can have more substantial impact than larger architectural differences.

In addition to hyperparameters that were in accordance with the paper, we also tested a few other variants. Interestingly, with this architecture L1 regularization worked better than L2 on the output layer. A complete opposite to what we observed on our HSM based models. We also observed that 2-pixel strides on the convolutional layers worked consistently better than half-the-filter-size strides. A result suggesting fine-tuned positioning of first layer filters is important.

[46]For example, we did not consult either of the existing implementations for details: https://github.com/david-klindt/NIPS2017,
https://github.com/aecker/cnn-sys-ident/tree/master/analysis/iclr2019.

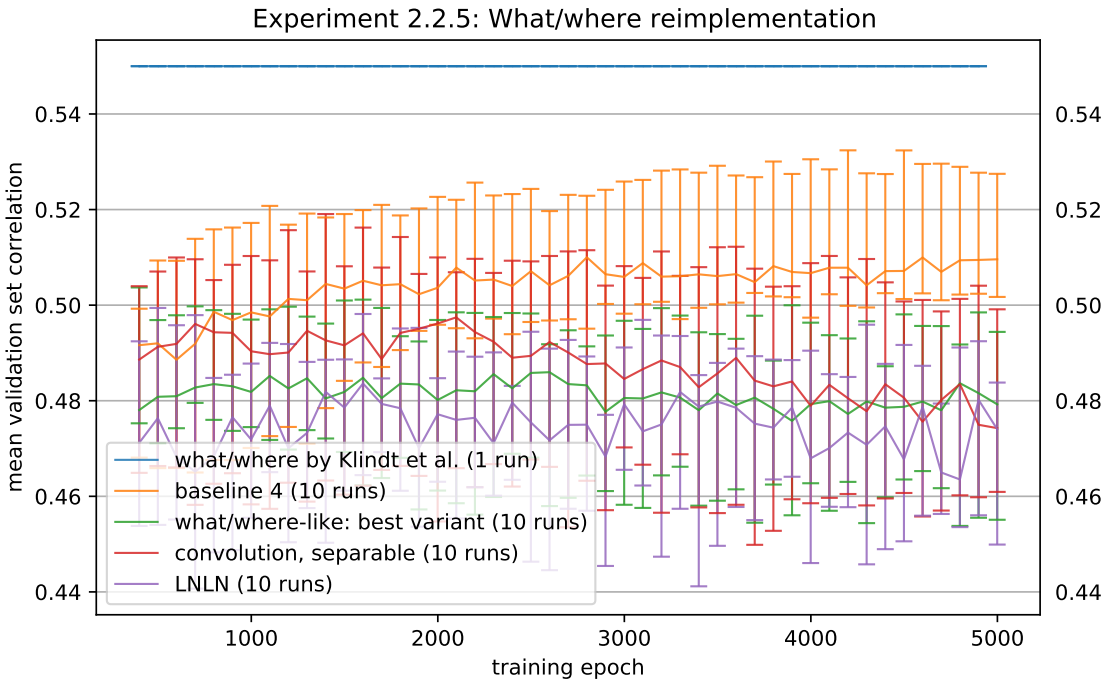

Figure 5.19: Comparing the best reimplemented what/where instance to various other model architectures[48].

5.2.3 Discussion

The theme of this section's experiments could be summarized as testing various computationally more generic architectures. The observed difference between different types of either the first or the second layer was relatively small (Fig. 5.20). At least when we assumed the best performing instances and in comparison to the variance within one architecture across different hyperparameter combinations. All three versions of the first layer fared very similarly in terms of achieved correlation on the validation set. The plain DoG variant, our *baseline*, remained the highest performing. But the difference was so small it could also be explained by not as involved finetuning of hyperparameters for the other variants as was done for the *baseline* models[49]. For this reason, we decided not to test small variations of either of the DoG layers versions, for example with only one Gaussian, a not-concentric version, etc.

This result was, in a way, expected. All tested alterations are theoretically computationally at least as expressive as the original HSM architecture with plain DoG layer. Its convolutional variant is just a positional relaxation of it, the normal convolutional layer sheds the *hard regularization* constraint on the filter weights, and the LNLN variant just goes one additional step further removing the factoring into two layers. A more surprising observation was that the convergence speed and its progression were also similar. After all, the main difference was that certain variations imposed priors in the form of *hard/soft regularization* on the layers, which we assumed would heavily impact convergence. While a more

[48] We only have fully trained data without variance for the Klindt et al. what/where model. We thus show the final mean performance for all time points.

[49] Across both the first section experiments and also the original Antolík et al. [2016]

thorough investigation would be needed, the similar but not better performance of alternatives serves as evidence that the plain difference-of-Gaussians restriction on the first layer filter might be correct but also is not necessary.

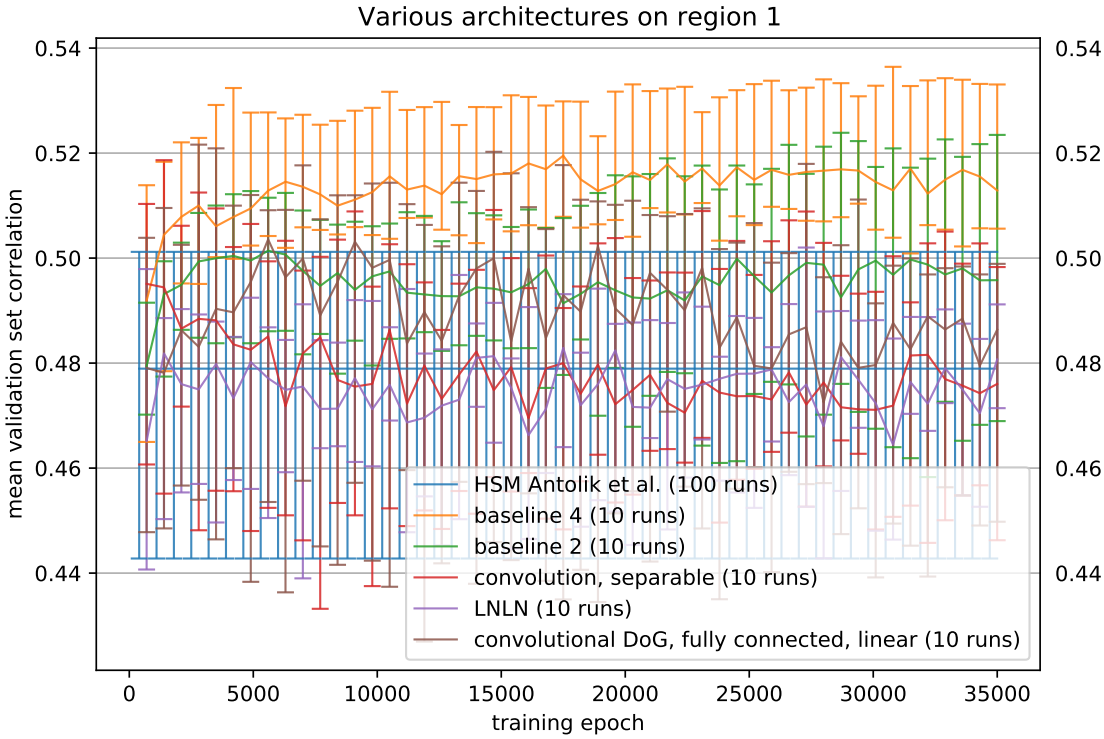

Figure 5.20: Various architectures[50]fully trained on region 1.

The small observed difference between the variation with a fully connected hidden layer and a variant with a separable layer was also relatively surprising Based on the improvements reported by Klindt et al. [2017] on their architecture, where using it increased the best-achieved performance from 0.47 to 0.55 on region 1, we were surprised by the, in comparison, smaller improvements on our normal convolution variant and almost negligible differences on the convolutional DoG version. It is, however, important to note that there are relatively big architectural differences between those two architectures, such as 1 filter versus 3 filters cascade and an additional fully connected output layer after the readout layer.

Both of the aforementioned results could be further researched. It would be especially interesting to compare the trained weights of the first layer filters and second layer masks between individual variants and figure out if they only lead to comparable levels of performance or if they are, in fact, equivalent computation-wise. Going even deeper, a comprehensive comparison between the *hard regularizations* – such as the of DoG layer – and various *soft regularizations* like a Laplacian regularization on a convolutional layer, could also lead to interesting discoveries.

Similarly, the question of why exactly does adding a non-linearity after the first layer cause diminished performance on our HSM-based architecture but seems to

[50]We only have fully trained data for the Antolík et al. *HSM model*. We thus show the final mean performance and its variance for all time points.

work for deeper models, could also be explored further in at least two ways. First, as a general ML research on the ability of a smaller network to deal with a redundant non-linearity. And second, as a more neuroscience inspired investigation into the question of whether the computation being modeled might actually be as simple as requiring only two non-linearities, with the goal of informing our understanding of visual computation processes.

In agreement with our observations regarding high-level architecture versus hyperparameters tuning and training time details, we also did not manage to achieve published results with our relatively close reimplementation of the what/where model by Klindt et al. This was mostly due to it being just an afterthought exercise without enough resources allocated to it, but it can, in our opinion, still serve as evidence of the importance of careful tuning.

We suggest further inquiry in the comparison of multilayer filters, such as those of Klindt et al. architecture, and single-layer filters of our HSM-based approach. Especially in the face of our aforementioned findings that additional nonlinearity leads to diminished performance in our HSM-based models while the what/where model features a cascade of 3 convolutions, each followed by a non-linearity.

5.3 All regions experiments

In this section, we make use of all three regions of the dataset[51]. First, we pool the regions together and test one shared model trained on all data. Second, we assess how various architectures explored in previous sections work on regions 2 and 3 and investigate how well our observations made on region 1 generalize.

5.3.1 Combined dataset

3.1.1 Region pooled dataset

Experiment 3.1.1 explored pooling multiple regions together to train a single model on higher quantity of data. The rationale behind this was the same as why record and then subsequently fit multiple neurons from a single V1 region at the same time, instead of fitting them one by one separately[52]. While the similarity in both inputs and computational properties is lower between V1 neurons from different regions than within a region, it is still there and – to a lesser degree – is even present across neurons from multiple individuals, such as between regions 1, 2 and region 3. Training one model across data from multiple regions can then help ground the common computation, thus mitigating the effect of random noise that is present in the data, and improve the accuracy of the trained model. All models trained in this experiment were based on *baseline 3*.

To assess the benefits of this approach, we first needed a baseline model that does not involve any data sharing between regions. We created an ensemble of three separate models, each trained on and responsible for predicting one of the three regions. To measure its performance, we used the average of validation set correlations across the three regions weighted by the number of output neurons

[51]Refer to *Dataset* (section 4.1.1).

[52]Notably, the LN/rLN model is effectively fitted per neuron.

in each. Thus, getting an unbiased per neuron average validation set correlation across all available data.

The shared model was trained on all three regions using the data filters capability of NDN3[53]. As such it effectively shared the first two layers, DoG filter and fully connected hidden, between all neurons from all three regions and had per output neuron[54] – and thus also per region – output. Since more data was being fit, the assumption was that the model might need bigger capacity and potentially stronger regularization. To test this, we trained variants with 9 to 25 DoG filters, hidden layer sizes of 20 % and 30 % of the number of output neurons[55], and various strengths of L2 regularization on both the hidden and the output layers.

The best instance of the model trained on all three regions pooled together achieved worse performance than its separate-models-ensemble counterpart (Fig. 5.21), with end of training validation set correlation of (0.439, 0.459, 0.426) versus (0.474, 0.486, 0.46).

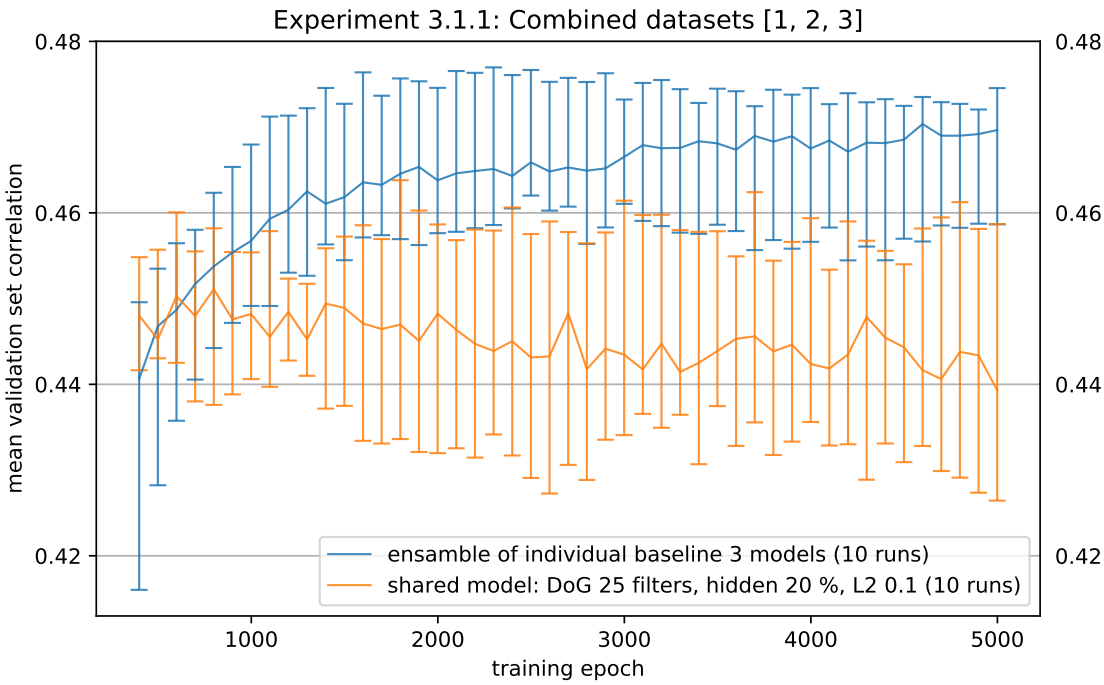

Figure 5.21: Models trained on regions 1,2,3 pooled together[56].

In addition to using all three regions, we also run the same experiment with only the first two regions pooled together, thus limiting ourselves to data recorded from the same animal[57]. In contrast the three-regions-variant, the model trained on only regions 1 and 2 pooled together reached correlation of (0.484, 0.495, 0.471), beating (0.475, 0.483, 0.475) of its 2-models-ensemble baseline (Fig. 5.22).

[53]Refer to *NDN3 toolkit* (section 3.1.1)

[54]Alternative interpretation is that we are effectively using transfer learning across the tree regions. Pre-training the first two layers of the model on all three regions together and then adding and fine tuning the last layer for each region individually.

[55]Across all three regions.

[56]Shared model represents a single model trained on data pooled together, the ensemble is just a weighted average of three separate models.

[57]Regions 1 and 2 were recorded using one animal, region 3 with another.

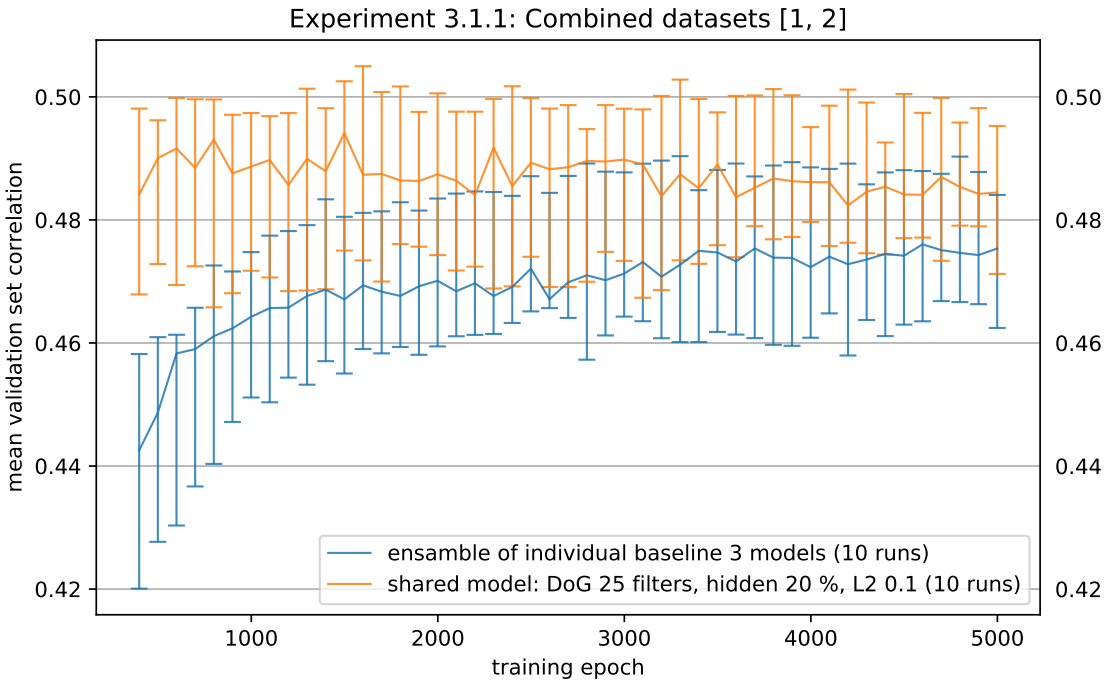

Figure 5.22: Models trained on regions 1,2 pooled together[58].

The simplest explanation of this phenomenon would be that the model, when shared across all three regions, did not have enough capacity to handle all the data. Since the hidden layer was proportional to the number of output neurons and we even tested a larger variant – with worse results, it was unlikely[59] to be the cause. Similarly, we did not observe differences between variants with 15 and 25 DoG filters that would suggest even more filters were needed. Performance of variants with various levels of L2 regularization on both fully connected layers also did not provide evidence for insufficient regularization.

With insufficient model capacity and regularization out of the way, several data focused hypotheses explaining these results remained. Notably, the third region was recorded using a different animal and so it is possible the similar computation properties assumption, as declared above, only holds within one animal. Following similar logic, it is possible that the noise distribution across the three regions substantially varies as well. A more thorough exploration would be needed to draw any conclusions, however. Specifically, we suggest further inquiry in the individual region performance of our pooled regions models and a more in depth investigation of the differences between the tree regions data.

A few additional observations. Models trained on pooled regions required more DoG layer filters than the 9 presented in the per-region models, and, with that, benefited from L2 regularization on both the output as well as the hidden fully connected layers. Stronger L2 regularization coupled with only 9 filters led to very poor performance.

[58]Shared model represents a single model trained on data pooled together, the ensemble is just a weighted average of two separate models.

[59]It is possible that we might also be experiencing critical regime of deep double descent phenomena [Nakkiran et al., 2019].

5.3.2 Testing on other regions

3.2.1 Various architectures across regions

We finish this chapter with **experiment 3.2.1** that assessed various architectures from previous sections on each of the three regions. We trained 5 models: *baseline 2* (*1.2.1*), *baseline 4* (*1.3.3*), the best *LNLN instance* (*2.1.1*), the best *convolution instance* (*2.2.2*), and the best *convolutional DoG instance* (*2.2.4*). Each was trained fully for 35 000 epochs with 10 runs. The motivation was to compare how various architectures fare and if their (relative) performance is stable across different regions

The *baseline* models with plain DoG layer outperformed all other variants across all three regions (Figs: 5.20, 5.23, 5.24, table 5.2). However, the improvements of *baseline 4* with respect to both the original reported Antolík et al. implementation and our *baseline 2* model on region 1 did not translate to regions 2 and 3. Especially on region 2, the difference between *baseline 4* (0.414, 0.434, 0.407) and numbers reported by Antolík et al. (0.412, 0.435, 0.382) was negligible for 90th and 50th percentile, with improvements only in the 10th percentile run. On region 3, the *baseline* models fared better but *baseline 2* model actually outperformed *baseline 4*.

Model	**Region 1**	**Region 2**	**Region 3**
Antolík et al.	0.479, 0.501, 0.442	0.412, 0.435, 0.382	0.437, 0.449, 0.419
Baseline 4	0.513, 0.533, 0.506	0.414, 0.434, 0.407	0.449, 0.455, 0.439
Baseline 2	0.496, 0.523, 0.469	0.415, 0.429, 0.398	0.454, 0.460, 0.449
Convolution	0.476, 0.498, 0.446	0.358, 0.391, 0.306	0.416, 0.445, 0.403
LNLN	0.481, 0.491, 0.471	0.340, 0.362, 0.327	0.404, 0.417, 0.396
Conv. DoG	0.486, 0.499, 0.450	0.318, 0.363, 0.288	0.396, 0.432, 0.384

Table 5.2: Validation set correlations (50th, 90th, 10th percentiles) of various architectures across three regions.

While on region 1 all models, apart from *baseline 4*, fared relatively similarly – with 50th percentile runs in the range of 0.476-0.496, on regions 2 and 3 there were two separate groups. With higher performance, there were the variants with plain DoG layer (*baselines 2*, *4*, and the original *HSM model* by Antolík et al. [2016]) and with a significant gap and lower performance all the other architectures (*normal convolution*, *convolutional DoG*, and *LNLN*). The relative performance in the lower group was not stable across regions either, with *convolutional DoG* model performing worst on region 2 but very comparably to others on regions 3 and 1.

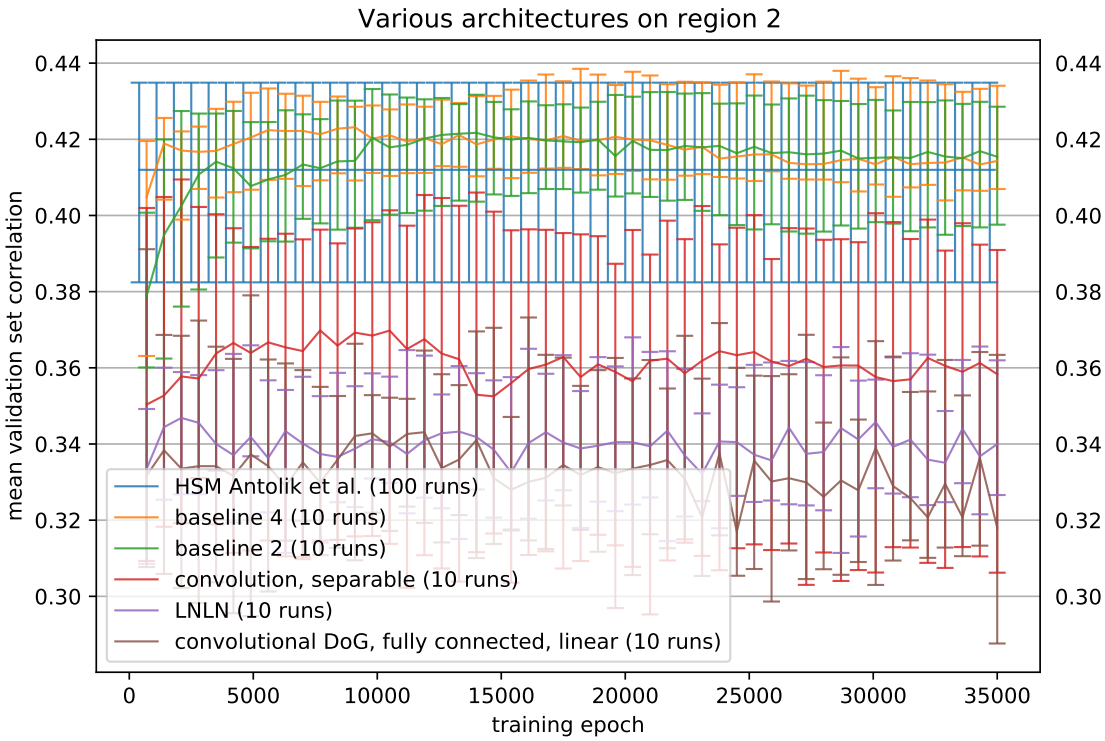

Figure 5.23: Various architectures[60]trained on region 2.

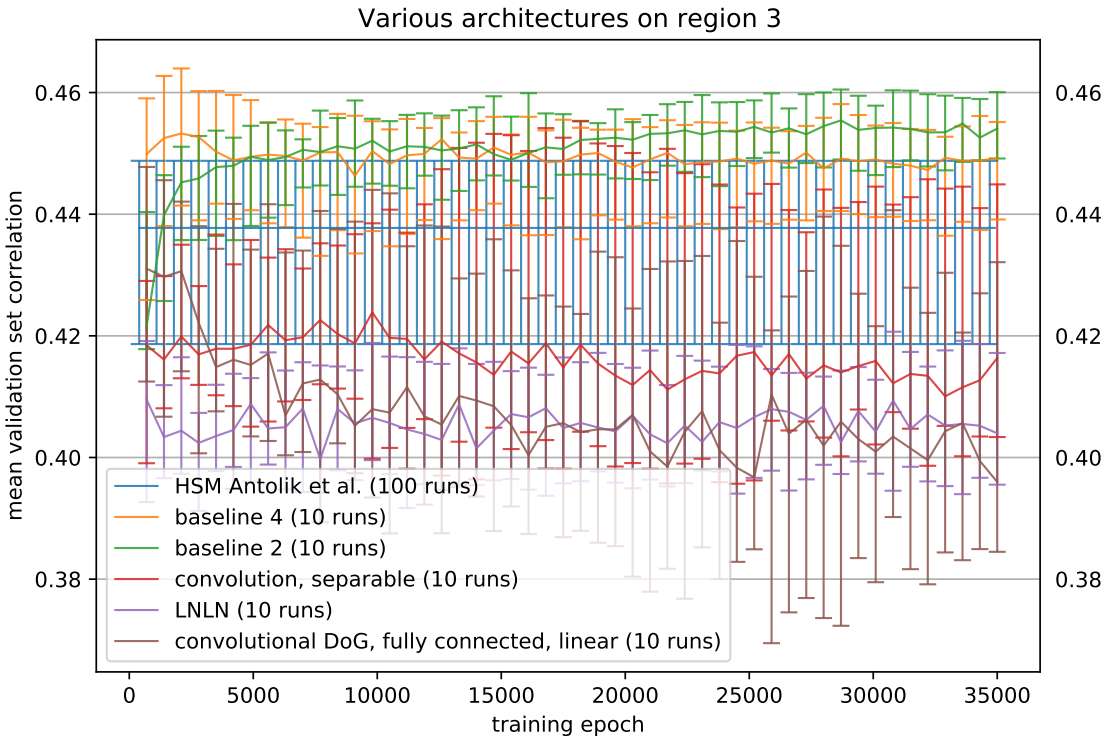

Figure 5.24: Various architectures[60] trained on region 3.

The *convolutional variants* (*normal convolution* and *convolutional DoG*) ex-

[60]We only have fully trained data for the Antolík et al. *HSM model*. We thus show the final mean performance and its variance for all time points.

hibited two noteworthy behaviours. First, they suffered overfitting on region 3. This was puzzling since, region 3 is of the same size as region 1, with region 2 being relatively smaller and thus – as we would expect – more susceptible to overfitting. Second, they were significantly less consistent in stability with respect to random initialization across the three regions. This showed especially on region 2 but was relatively clear even on region 3. Surprisingly, the *LNLN model* seemed relatively well behaved in this regard, with its 10-90th percentile run range consistent and comparable to the *plain DoG* variants across all three regions.

Given the big difference between *plain DoG* variants and both of the *convolutional architectures* together with the *LNLN model*, we suspect the main benefit of the *plain DoG* over *convolutional variants*, when it comes to robustness across multiple similar datasets without further fine-tuning, is in the very limited number of outputs from the first filter layer. That said, we were surprised the *separable hidden layer* present in the *convolution model* did not achieve a comparable effect. On similar note, we expected the *hard regularization* of the *convolutional DoG model* or the *location invariant properties* of the *normal convolution model* to also positively influence robustness in comparison to the very generic *LNLN model*.

A more thorough investigation would be needed to draw any conclusions. We especially recommend looking into the trained weights of each architecture and comparing them across the three regions. Repeating some of the grid-search experiments to fine-tune hyperparameters of the *convolutional architectures* on regions 2 and 3 might also bring relevant data. For now, however, we believe these results can still serve as evidence for the robustness of the original HSM architecture with its plain non-convolutional DoG layer.

5.3.3 Discussion

In this section, we made two important findings. First, pooling separately recorded regions together to train a single shared model did not provide expected benefits. On the contrary, for the version with all three regions pooled together, it led to worse performance than a simple ensemble of three separate models. To make things more convoluted, this method achieved slightly better results than the ensemble baseline when only the first two regions were pooled. This suggests the intriguing hypothesis that the pooling of the data across regions might work within an animal but not across animals, as regions 1 and 2 were recorded in the same animal, while region 3 in a different one.

Our conclusion is that in our hands, pooling of data across multiple recordings does not impart advantage over fitting models individually. It is, however, important to emphasize that this conclusion is strictly specific to the data available in this thesis and to our current level of understanding of both the dataset and the modeled computation.

Second, we found heterogeneity of outcomes when testing various architectures, fine-tuned on region 1 in the previous sections, on regions 2 and 3. Our improvements of the *HSM architecture* with *baseline* models proved not to be universal, with, for example, *baseline 2* achieving higher than *baseline 4* on region 3. We also found that architectures with a plain DoG layer were generally more

robust when it comes to transferring them to similar, but not equal, datasets without further hyperparameters fine-tuning. In contrast to this finding, we did not observe similar benefit to either the convolutional DoG variant or normal convolution variant with a separable readout layer over a simple LNLN architecture.

Conclusion

The incorporation of biological detail into deep neural architectures for the purpose of understanding the brain computation is a promising but still only a poorly explored area of computational neuroscience. The HSM model introduced by Antolík et al. [2016], that was the focus of this thesis, is one such example. While, at the time of writing, outperforming state-of-the-art approaches, the HSM model was implemented in a very ad-hoc fashion, relatively sensitive to initial conditions, and poorly characterized and compared with respect to similar but more traditional DNN architectures. These shortcomings posed a significant barrier for future development of this bio-inspired architecture. The main goal of this thesis was to remedy this situation. This thesis offers following contributions:

- Reimplementation of the model in modern neuroscience oriented DNN framework.
- Identification of hyper-parametrization that improves model's robustness with respect to initial conditions and input datasets.
- Comparison of HSM architecture to more traditional DNNs, showing its advantage – especially especially with respect to robustness.
- Evaluation of dataset pooling and model robustness with respect to different regions recording both within the same and across different animals.

Main findings

The first set of experiments showed us that non-architectural hyperparameters such as learning rate and the scale of input data can have a big impact. Similarly, when those are already tuned, the gains achievable by regularization are relatively minimal – at least for the HSM architecture featuring a plain DoG layer with its *hard regularization* in the form of explicitly parameterized filters. The second set went further, showing that when properly regularized and fine-tuned, various types of more generic filters (convolution, convolutional DoG, fully connected) work almost as well when used as the first layer instead of DoG layer. Similarly, it failed to show a substantial improvement for variations with a separable hidden layer instead of a fully connected one. It also revealed unexpected interactions with respect to a non-linearity after the first filter layer of the model.

The third section gave us two insights. First, even though the regions should be similar, improvements made on one do not trivially translate to others. On region 3 we observed a variant of our HSM-based architecture fine-tuned on region 1 perform worse than less tuned versions. Further, we found evidence that even though more generic variants can achieve a similar level of performance as the HSM architecture – with its plain DoG layer, when carefully optimized, they are significantly less robust when used on other regions without hyperparameter changes. Second, pooling regions together and training one shared model does not bring expected benefits, and can actually be detrimental. At least for a variation where all three regions are combined without further processing.

Lessons learned

Putting it all together, few themes emerge. First, small architectural variations of computationally very similar building blocks are not as significant when best achievable performance is concerned. The biggest difference is in the need for well-tuned regularizations. These are paramount for architectures with higher amounts of free parameters, such as with unrestricted convolution layers or the LNLN variant. With regularization and proper hyperparameters, training to a comparable level of performance similarly quickly is possible, however.

The benefit of more constrained versions, such as the plain DoG layer variant, is mainly when exhaustive grid search across – not only regularization – hyperparameters is not an option. For example, when we want to use a model fine-tuned on one region on another without further modifications. On this note, the comparable but not better performance of more generic variants of first layer filters (LNLN, convolution) serve as further evidence that the initial part of visual computation might be well approximated using difference-of-Gaussians filters.

Second, the variability of performance with respect to small hyperparameters changes both within a region but mainly across regions can be significant, especially for certain architectures, and the space of all potential hyperparameter combinations is vast. Thus, any conclusions about the underlying fundamentals of the biological computation, that we are trying to model, should be done very carefully. So as not to mistake an observation of one model with a particular set of hyperparameters being better than another one with its set of hyperparameters for a signal about the biological principles of the brain. Similarly, training a set of models per each model instance and analysing distributions of runs instead of single runs proved to be a necessity to even attempt to draw any statistically significant conclusions.

Third, basic deep learning intuition trained mainly on a few types of computer vision problems with vast not-noisy datasets can be helpful but should never be blindly relied upon. As we have seen with the learning rate experiment, explorations of the input data scaling, or additional non-linearity tests, changes that in more standard ML settings might have smaller impact can be relatively influential with this type of data. This ties back to our second lesson learned. Even a significant difference in performance across multiple instantiations of the same model can be due to, for example, the optimizer interacting with a particular data scale better on one architecture than on another and have nothing to do with the similarity between the individual architectures and the computation they are trying to model.

Future work

As already discussed at various places throughout the Experiments and results chapter, there are plenty of opportunities for further research. We identified four main areas: further investigation of bigger and computationally non-equivalent architectural changes, analysis of layers activations and the impact of additional non-linearities, comparison of data between regions and proper analysis of models trained on pooled data, and a more involved investigation of the trained weights of both the first layer filters and second layer masks for various architectures

already explored in this work.

For further architectures, several options are readily available. Proper reimplementation of the what/where model as introduced by Klindt et al. could be a great starting point that could be followed by exploration of novel deeper architectures, possibly mixing generic convolutions with convolutional and plain DoG layers. Further, we suggest incorporating other layer types in new combinations, such as rotational-equivariant convolutional filters [Ecker et al., 2018] or, in NDN3 already implemented, separable versions of normal convolutions. Additionally, variants of the HSM architecture with only a single readout layer that is not followed by a fully connected output layer or deeper architectures with less non-linearities could be explored. We also suggest continuing investigating transfer learning with classical computer vision datasets, especially for the initial filter layers.

Adding batch normalization[61] might also answer some questions – especially around the input scale impact, and potentially unblock deeper architectures. As already discussed in the Experiments and results section, proper investigation into why and how the additional non-linearity on the first filter layer impacts performance could also lead to informative conclusions.

For the third area, we suggest looking at the individual region datasets statistically, and comparing whether they are, in fact, similar in terms of distribution, within region cross-correlations, etc. On the model side, an analysis of what architectures and hyperparameters work on each region, similar to our work on region 1, could also shed some light on the differences between regions. It might also unblock the potential that, we still believe, lies in pooling the three regions and training just one shared model, essentially leveraging within dataset transfer learning. On that note, a good starting point on that might be to test why the combined model had worse performance than the ensemble. Whether it was due to worse performance on only one region or if all regions neurons were negatively impacted by the shared training.

Lastly, an exhaustive comparison of the trained first layer filters and second layer masks of the architectures explored in this thesis could help to unravel the impacts of *hard* versus *soft regularizations*. It could answer the question of whether the other first layer variants – that are essentially just more relaxed versions of the DoG layer – converge to it, or if they just achieve a similar level of performance with substantially different filters. Similar inquiry could be done for the hidden readout layer. As part of this, an investigation into how the filters change during training and what is the role of random initialization might also be worth pursuing.

[61] [Ioffe and Szegedy, 2015]

Bibliography

L. F. Abott. Lapicque's introduction of theintegrate-and-fire model neuron (1907). *Brain Research Bulletin*, 50:303–304, 1999.

Ján Antolík, Sonja B. Hofer, James A. Bednar, and Thomas D. Mrsic-Flogel. Model constrained by visual hierarchy improves prediction of neural responses to natural scenes. *PLOS Computational Biology*, 12(6):1–22, 06 2016. doi: 10.1371/journal.pcbi.1004927. URL `https://doi.org/10.1371/journal.pcbi.1004927`.

Jakub Arnold. Bayesian optimization of hyperparameters using gaussian processes. Master's thesis, Charles University, 2019. URL `https://is.cuni.cz/webapps/zzp/detail/212253/`.

Mark F. Bear, Barry W. Connors, and Michael A. Paradiso. *Neuroscience: exploring the brain*. Lippincott Williams & Wilkins, Philadelphia, PA, 3rd ed edition, 2007. ISBN 9780781776073 9780781760034. OCLC: ocm62509134.

Irwan Bello, Barret Zoph, Ashish Vaswani, Jonathon Shlens, and Quoc V. Le. Attention Augmented Convolutional Networks. *arXiv e-prints*, art. arXiv:1904.09925, April 2019.

Daniel A. Butts. Data-driven approaches to understanding visual neuron activity. *Annual Review of Vision Science*, 5(1):451–477, 2019. doi: 10.1146/annurev-vision-091718-014731. URL `https://doi.org/10.1146/annurev-vision-091718-014731`. PMID: 31386605.

Santiago A. Cadena, George H. Denfield, Edgar Y. Walker, Leon A. Gatys, Andreas S. Tolias, Matthias Bethge, and Alexander S. Ecker. Deep convolutional models improve predictions of macaque v1 responses to natural images. *PLOS Computational Biology*, 15(4):1–27, 04 2019. doi: 10.1371/journal.pcbi.1006897. URL `https://doi.org/10.1371/journal.pcbi.1006897`.

Matteo Carandini, Jonathan B. Demb, Valerio Mante, David J. Tolhurst, Yang Dan, Bruno A. Olshausen, Jack L. Gallant, and Nicole C. Rust. Do we know what the early visual system does? *Journal of Neuroscience*, 25(46):10577–10597, 2005. ISSN 0270-6474. doi: 10.1523/JNEUROSCI.3726-05.2005. URL `https://www.jneurosci.org/content/25/46/10577`.

Dami Choi, Christopher J. Shallue, Zachary Nado, Jaehoon Lee, Chris J. Maddison, and George E. Dahl. On Empirical Comparisons of Optimizers for Deep Learning. *arXiv e-prints*, art. arXiv:1910.05446, October 2019.

Taco S. Cohen and Max Welling. Group Equivariant Convolutional Networks. *arXiv e-prints*, art. arXiv:1602.07576, February 2016.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv e-prints*, art. arXiv:2010.11929, 2020.

Alexander S. Ecker, Fabian H. Sinz, Emmanouil Froudarakis, Paul G. Fahey, Santiago A. Cadena, Edgar Y. Walker, Erick Cobos, Jacob Reimer, Andreas S. Tolias, and Matthias Bethge. A rotation-equivariant convolutional neural network model of primary visual cortex. *arXiv e-prints*, art. arXiv:1809.10504, September 2018.

Xavier Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. *Journal of Machine Learning Research - Proceedings Track*, 9:249–256, 01 2010.

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

Robbe L. T. Goris, J. Anthony Movshon, and Eero P. Simoncelli. Partitioning neuronal variability. *Nature Neuroscience*, 17(6):858–865, Jun 2014. ISSN 1546-1726. doi: 10.1038/nn.3711. URL `https://doi.org/10.1038/nn.3711`.

Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. *arXiv e-prints*, art. arXiv:1705.08741, May 2017.

Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv e-prints*, art. arXiv:1502.03167, February 2015.

Jian Jin, Ming Li, and Long Jin. Data normalization to accelerate training for linear neural net to predict tropical cyclone tracks. *Mathematical Problems in Engineering*, 2015:931629, Jul 2015. ISSN 1024-123X. doi: 10.1155/2015/931629. URL `https://doi.org/10.1155/2015/931629`.

Kendrick N. Kay, Thomas Naselaris, Ryan J. Prenger, and Jack L. Gallant. Identifying natural images from human brain activity. *Nature*, 452(7185): 352–355, Mar 2008. ISSN 1476-4687. doi: 10.1038/nature06713. URL `https://doi.org/10.1038/nature06713`.

J. Kiefer and J. Wolfowitz. Stochastic estimation of the maximum of a regression function. *Ann. Math. Statist.*, 23(3):462–466, 09 1952. doi: 10.1214/aoms/1177729392. URL `https://doi.org/10.1214/aoms/1177729392`.

Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv e-prints*, art. arXiv:1412.6980, December 2014.

David A. Klindt, Alexander S. Ecker, Thomas Euler, and Matthias Bethge. Neural system identification for large populations separating what and where. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 3509–3519, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-10 (canadian institute for advanced research), 2015. URL `http://www.cs.toronto.edu/~kriz/cifar.html`.

Yann LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, page 9–50, Berlin, Heidelberg, 1998. Springer-Verlag. ISBN 3540653112.

Grace W. Lindsay. Convolutional neural networks as a model of the visual system: Past, present, and future. *Journal of Cognitive Neuroscience*, page 1–15, Feb 2020. ISSN 1530-8898. doi: 10.1162/jocn_a_01544. URL `http://dx.doi.org/10.1162/jocn_a_01544`.

Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. On the Variance of the Adaptive Learning Rate and Beyond. *arXiv e-prints*, art. arXiv:1908.03265, August 2019.

Michael London and Michael Häusser. Dendritic computation. *Annual review of neuroscience*, 28:503—532, 2005. ISSN 0147-006X. doi: 10.1146/annurev.neuro.28.061604.135703. URL `https://doi.org/10.1146/annurev.neuro.28.061604.135703`.

Pranava Madhyastha and Rishabh Jain. On Model Stability as a Function of Random Seed. *arXiv e-prints*, art. arXiv:1909.10447, September 2019.

Eamonn Maguire. *Systematising glyph design for visualization*. PhD thesis, Oxford University, 2015. URL `http://ora.ox.ac.uk/objects/uuid:b98ccce1-038f-4c0a-a259-7f53dfe06ac7`.

Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. Evolving Deep Neural Networks. *arXiv e-prints*, art. arXiv:1703.00548, March 2017.

Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. Deep Double Descent: Where Bigger Models and More Data Hurt. *arXiv e-prints*, art. arXiv:1912.02292, December 2019.

Joe Pharos. Gabor filter, 2006. URL `https://commons.wikimedia.org/wiki/File:Gabor_filter.png`.

Prajit Ramachandran, Niki Parmar, Ashish Vaswani, Irwan Bello, Anselm Levskaya, and Jonathon Shlens. Stand-Alone Self-Attention in Vision Models. *arXiv e-prints*, art. arXiv:1906.05909, June 2019.

Joseph Redmon and Ali Farhadi. YOLOv3: An Incremental Improvement. *arXiv e-prints*, art. arXiv:1804.02767, April 2018.

F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 6:386–408, 1958.

Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv e-prints*, art. arXiv:1609.04747, September 2016.

Eric Schwartz. *Computational neuroscience*. MIT Press, Cambridge, Mass, 1990. ISBN 978-0-262-19291-0.

Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv e-prints*, art. arXiv:1409.1556, September 2014.

Samuel L. Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V. Le. Don't Decay the Learning Rate, Increase the Batch Size. *arXiv e-prints*, art. arXiv:1711.00489, November 2017.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014. URL `http://jmlr.org/papers/v15/srivastava14a.html`.

Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. A Survey on Deep Transfer Learning. *arXiv e-prints*, art. arXiv:1808.01974, August 2018.

Edgar Y. Walke, Fabian H. Sinz, Emmanouil Froudarakis, Paul G. Fahey, Taliah Muhammad, Alexander S. Ecker, Erick Cobos, Jacob Reimer, Xaq Pitkow, and Andreas S. Tolias. Inception in visual cortex: in vivo-silico loops reveal most exciting images. *bioRxiv*, 2018. doi: 10.1101/506956. URL `https://www.biorxiv.org/content/early/2018/12/28/506956`.

Ben Willmore and Darragh Smyth. Methods for first-order kernel estimation: Simple-cell receptive fields from responses to natural scenes. *Network (Bristol, England)*, 14:553–77, 09 2003. doi: 10.1088/0954-898X/14/3/309.

Lei Wu, Zhanxing Zhu, and Weinan E. Towards Understanding Generalization of Deep Learning: Perspective of Loss Landscapes. *arXiv e-prints*, art. arXiv:1706.10239, June 2017.

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

Barret Zoph and Quoc V. Le. Neural Architecture Search with Reinforcement Learning. *arXiv e-prints*, art. arXiv:1611.01578, November 2016.

Štěpán Hojdar. Using neural networks to generate realistic skies. Master's thesis, Charles University, 2019. URL `https://is.cuni.cz/webapps/zzp/detail/213909/`.

List of Figures

1.1 Receptive fields of retinal ganglion and LGN cells 9
1.2 Gabor filter-type receptive field typical for a simple cell 10

2.1 HSM model architecture 18
2.2 klindt et al. model 19

3.1 NDN model description and subsequent architecture 23
3.2 NDN computation graph initialization 24
3.3 Experiments pipeline 28
3.4 Experiment analysis toolkit 28

4.1 Experiments methodology 33
4.2 Impact of 25 vs 50 runs 35

5.1 Experiment 1.1.1 37
5.2 Experiment 1.2.1 38
5.3 Experiment 1.2.1 2 39
5.4 Experiments 1.2.2 and 1.2.3 40
5.5 Experiment 1.2.4 41
5.6 Experiment 1.2.5 41
5.7 Experiment 1.2.6 42
5.8 Experiment 1.3.1 43
5.9 Experiment 1.3.2 44
5.10 Experiment 1.3.3 45
5.11 Experiment 1.4.1 46
5.12 Baseline models on region 1 48
5.13 Experiment 2.1.1 50
5.14 Experiment 2.1.2 51
5.15 Experiment 2.2.1 52
5.16 Experiment 2.2.2 53
5.17 Experiment 2.2.3 54
5.18 Experiment 2.2.4 55
5.19 Experiment 2.2.5 57
5.20 Various architectures on region 1 58
5.21 Experiment 3.1.1 60
5.22 Experiment 3.1.1 2 61
5.23 Various architectures on region 2 63
5.24 Various architectures on region 3 63

List of Tables

1.1 Commonly used activation functions 13
1.2 Relevant weights soft regularizations. 16

4.1 Regions of dataset 31
4.2 Performance of prior works. 32
4.3 Performance percentiles of HSM model 32

5.1 Evaluation of 5k/35k epochs training on region 1 47
5.2 Performance of various models across regions 62

List of Abbreviations

DNN Deep neural network.

CNN Convolutional neural network.

NDN3 Neural Deep Network (modeling framework).

LGN Lateral geniculate body.

V1 Primary visual cortex.

LN Linear-nonlinear (model).

LNP Linear-nonlinear Poisson (model).

rLN Regularized linear-nonlinear (model).

GLM Generalized linear model.

LNLN LNLN cascade (model).

RF Receptive field.

STA Spike Triggered Average.

MLE Most likelihood estimation.

ML Machine learning.

DL Deep learning.

TF TensorFlow.

NP NumPy.

DoG Difference of Gaussians.

HSM Hierarchical Structural Model [Antolík et al., 2016].

Glossary

Hard regularization Hard constraint on the computation. For example, in the form of an explicitly parameterized filter that cannot deviate from its form (e.g. difference-of-Gaussians filter instead of an arbitrary fully connected filter).

Soft regularization Gradual penalty on certain properties of a computation. Usually implemented as an additional element of the loss function, punishing certain properties of free parameters (e.g. Laplacian regularization on a fully connected filter pushing it towards spatial smoothness).

Experiment A single grid-search exploration of a small subset of hyperparameters or architecture aspects.

Experiment instance One particular model architecture with a specific set of hyperparameters.

Experiment (instance) run One initialization and fitting of a single experiment instance.

Baseline (model) An architecture + hyperparameters that is used as a basis for experiments.

HSM / Antolík et al. model Three layer neural model with DoG layer introduced by Antolík et al. [2016].

What/where model Convolutional neural model with separable layer introduced by Klindt et al. [2017].

Receptive field The portion of sensory space that elicits neuron's responses when stimulated.

Readout layer The first fully connected (or similar) layer after a set of convolutional layers.

A. Attachments

A.1 Digital attachments

A.1.1 msc-neuro

Repository containing all tools introduced in *Experiments and analysis pipeline* (section 3.3) as well as the implementation of experiments from *Experiments and results* (section 5). For description, refer to `./README.md`. The repository is also available online at: https://github.com/petrroll/msc-neuro.

A.1.2 NDN3

Fork of the NDN3 library containing all extensions described in *Implemented NDN3 extensions* (section 3.2), including the additions not (yet) merged upstream[1]. Also available online: https://github.com/petrroll/NDN3/tree/messy-Develop.

[1] https://github.com/NeuroTheoryUMD/NDN3