Understanding Deep Learning: Deep Linear Neural Networks and Fisher Information

Luzzatto, Leone

Understanding Deep Learning

Deep Linear Neural Networks and Fisher Information

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
PHYSICS

Author :	Leone Luzzatto
Student ID :	3271048
Supervisor :	Dr. Subodh Patil
Second corrector :	Prof. Koenraad Schalm

Leiden, The Netherlands, June 16, 2023

Understanding Deep Learning

Deep Linear Neural Networks and Fisher Information

Leone Luzzatto

Instituut-Lorentz, Leiden University
P.O. Box 9500, 2300 RA Leiden, The Netherlands

June 16, 2023

Abstract

In recent years, deep neural networks have attracted the attention of both the academic community and the general public. An effort to theoretically understand the intricacies of these systems is ongoing and physics-inspired approaches may have a part to play. In this thesis, we will discuss recent results in the theoretical study of deep linear neural networks. This class of neural networks has very limited real-world applications, but it could provide a good training ground for developing theoretical techniques that could prove useful beyond the simple linear case. We will also argue that Fisher information, and in particular "sloppy model" logic, can be a useful tool for future research on deep neural networks, in particular for network architecture optimization.

Contents

Chapter 1

Introduction

Over the past few years, deep neural networks have been a subject of increased interest on the part of both the academic community and the public at large. Research on neural networks exists at the intersection of a diverse collection of academic fields: people working on this topic have backgrounds in computer science, mathematics, and neuroscience. And, of course, physics. Physicists have contributed to the field from an early stage, as evidenced by the many textbooks written on the subject of physics and machine learning [2, 3], and physicists may still have a role to play in this evolving field going forward.

Research on this subject takes a variety of different forms, but from the point of view of a theoretical physicist, one of, if not the most fascinating is a pursuit of a theoretical understanding of how these machines operate. Neural networks are large machines built out of relatively simple units. Such a system naturally lends itself to be studied through many of the techniques that physicists have developed over the years, in particular statistical mechanics.

The present thesis is concerned with the theoretical study of deep neural networks. In particular, we will discuss a series of recent papers dealing with deep linear neural networks [4, 8, 14]. These are simple enough systems that analytical calculations are possible, but still complex enough that their behavior is not trivial. Unfortunately, deep linear neural networks (DLNN) have little to no real-world applications. Nevertheless, studying simplified models can often provide useful insights that can be carried over to the study of more realistic systems.

With this perspective in mind, we will discuss the work of Li and Sompolinsky [8], who introduced a physics-inspired procedure to study DLNNs: the "back-propagating kernel renormalization." As the name

Version of June 16, 2023– Created June 16, 2023 - 07:37

suggests, this procedure presents intriguing parallels with the renormalization group that may be worth exploring in more detail. Then, we will discuss the work of Zavatone-Veth and Pehlevan [14] and that of Hanin and Zlokapa [4], who study DLNNs using the tools of Bayesian inference. We will show how these three papers, using the two different theoretical frameworks of statistical mechanics and Bayesian statistics, reach similar conclusions in their descriptions of DLNNs.

Finally, we will argue that Fisher information [1] can be an effective tool for studying machine learning systems. In particular, the logic of "sloppy models" [9, 12] has the potential to be useful in designing, as well as understanding neural networks. We will show numerically that simple neural network models can be characterized as sloppy models, and we will discuss how this could be relevant for future research.

Chapter 2

A Physicist's Introduction to Neural Networks

Machine learning[1] problems often follow a similar script. Broadly, we have a dataset $\mathcal{D}(\mathbf{X}, \mathbf{Y})$, a model $f_{\boldsymbol{\theta}}$ of parameters $\boldsymbol{\theta} = (\theta_1, \ldots, \theta_n)$, and a cost function $E(\mathbf{Y}, f_{\boldsymbol{\theta}}(\mathbf{X}))$; i.e. a set of pairs of independent and dependent variables $\{(\mathbf{x}^\mu, \mathbf{y}^\mu)\}_{\mu=1,\ldots,P}$, a function that, given input $\mathbf{x}$, outputs a prediction for the corresponding $\mathbf{y}$, and a procedure to measure how well this function describes the observed data. The final missing ingredient is an optimization procedure: we need a way to tweak the parameters $\boldsymbol{\theta}$ to improve the model's performance.

Most physicists will be familiar with this basic structure, as it is used to fit experimental data with a theoretical prediction. For example, we could take measurements of the voltage applied across a metal wire and the current running through it (the dataset), postulate a linear relationship between the two (the model), and estimate the resistance of the wire (the parameter) using the method of least squares, which consists in minimizing the squared difference between each measured point and the corresponding theoretical prediction (the cost function).

One important difference between a simple regression problem, like fitting Ohm's law, and a typical problem in machine learning, is that often in machine learning an exact, or even approximate, mathematical description of the process generating the data is not available. Overcoming this problem requires more sophisticated techniques as well as a variety of different strategies to deal with different tasks.

[1] Machine learning is a vast field, that can be broadly divided into supervised, unsupervised, and reinforcement learning. The present thesis will only deal with supervised learning.

In this chapter, we will provide an overview of some concepts and techniques common in machine learning, with a particular focus on neural networks, which will be the subject of most of Chapters 3 and 4. The source of the information found in the rest of this chapter is a 2019 review written by Mehta et al. [10].

2.1 Concepts in Machine Learning

2.1.1 Fitting and predicting

One important problem in machine learning is overfitting. Naively, we could think that a model with a larger number of free parameters would be more flexible to describe subtler features of a dataset. In practice, however, models with a large number of free parameters perform worse than comparatively simpler models. The more complex a model, the higher the chance that some of the features it is describing are due to statistical noise, rather than to a fundamental aspect of the underlying phenomenon[2]. The goal of many machine learning models, like most scientific theories, is not just to accurately describe the available data, but rather to accurately predict the output given a previously unseen input.

Over decades of research, many techniques have been developed to improve the performance of machine learning models, and many of them can be understood as acting to avoid overfitting, to some degree. The goal is to strike a balance: the model needs to be complex enough to capture the key features of the dataset while losing as little accuracy as possible when it is applied to unseen data. Most importantly, it is common practice when approaching any machine learning problem to randomly divide the dataset $\mathcal{D}$ into two separate datasets: a training dataset $\mathcal{D}_{\text{train}}$, used to train the model, and a test dataset $\mathcal{D}_{\text{test}}$, used to test the trained model's performance.

2.1.2 Gradient descent

A key ingredient in any supervised learning task is the training procedure. This is a procedure that updates the model's parameters in an effort to improve performance. During this procedure, model performance is typically measured on the training dataset, the only one we have access to

[2]Physicists have long been wary of models with a large number of free parameters. Rather famously, John von Neumann is said to have told Enrico Fermi: "With four parameters I can fit an elephant, and with five I can make him wiggle his trunk."

during training. The most commonly used training procedures are variations on the theme of gradient descent, a simple and commonly used minimization algorithm. We can employ it to minimize the cost function $E(\mathbf{Y}, f_{\boldsymbol{\theta}}(\mathbf{X})) \equiv E(\boldsymbol{\theta})$, thereby fitting the model to the data. It can sometimes be useful to think of the cost function as the energy of a system. Training a model is then analogous to navigating through an energy landscape, trying to find its minimum.

Plain gradient descent

The simplest gradient descent algorithm consists of iteratively updating the parameters in the direction where $E(\boldsymbol{\theta})$ is steepest. At each time step, update the parameters according to

$$\Delta\boldsymbol{\theta}_t = -\eta_t \nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}) \,, \tag{2.1a}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \Delta\boldsymbol{\theta}_t \,, \tag{2.1b}$$

where we have introduced the learning rate η_t.

Gradient descent is simple, but it has several limitations. The energy landscapes associated with machine learning models can get rather complicated, and gradient descent can easily get stuck in local minima, or take a very long time to escape a plateau.

Adding stochasticity

In most cases, the cost function (a measure of how well a model is performing on a dataset) can be written as a sum over the individual data points[3],

$$E(\boldsymbol{\theta}) = \sum_{\mathbf{x}\in\mathbf{X}} E(\mathbf{x}, \boldsymbol{\theta}) \,.$$

This offers the chance of speeding up calculations by using random subsets of the training data to approximate the gradient during training. These are known as mini-batches.

In practice, if our training dataset contains N data points, we can randomly subdivide it into N/M mini-batches of $M \ll N$ points. If we denote the mini-batches as $\mathcal{B}_i$, with i running over different batches, the gradient can be approximated as

$$\nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}) = \sum_{\mathbf{x}\in\mathbf{X}} \nabla_{\boldsymbol{\theta}} E(\mathbf{x}, \boldsymbol{\theta}) \approx \frac{N}{M} \sum_{\mathbf{x}\in\mathcal{B}_i} \nabla_{\boldsymbol{\theta}} E(\mathbf{x}, \boldsymbol{\theta}) \,. \tag{2.2}$$

[3]Usually, "cost function" refers to the performance on a dataset, while "loss function" is used for the performance on individual data points.

This approximate gradient can then be used instead of the exact result in Eq. (2.1), cycling over the mini-batches as we update the parameters.

In addition to speeding up calculations, this procedure has an additional benefit: computing the gradient using a small random sample of data points instead of the full dataset introduces a degree of stochasticity to the training process, which makes it harder for the model to get stuck in a local minimum.

Adding momentum

One other common variation on gradient descent (these variations are often used in combination with one another) is the addition of a "memory term":

$$\Delta\boldsymbol{\theta}_t = -\gamma\Delta\boldsymbol{\theta}_{t-1} - \eta_t\nabla_{\boldsymbol{\theta}}E(\boldsymbol{\theta}) \,, \tag{2.3a}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \Delta\boldsymbol{\theta}_t \,. \tag{2.3b}$$

where we introduced a new hyperparameter[4], $0 \leq \gamma \leq 1$, which determines the characteristic time scale over which the algorithm keeps the memory of previous updates. In a way, this new term acts as an inertia term, resisting sudden changes to the velocity through parameter space. In fact, Eq. (2.3) has the same form as the discretized equation of motion of a particle moving through a viscous medium. In this analogy, plain gradient descent is akin to the overdamped limit.

The advantages of including such an inertia term should be clear: if in one direction the cost function has a small, but slowly varying gradient, the inertia term will cause the algorithm to build up speed in that direction. At the same time, directions in which the gradient is quickly fluctuating will have a diminished effect on training.

Moments of the gradient

Ideally, we would like to compute higher derivatives of the cost function. This would allow us to adapt the step size to avoid getting stuck on saddle points or skipping over deep, but narrow minima. Computing the Hessian matrix of the cost function at each step, however, is extremely computationally expensive for models with many parameters. For this reason, there are several different algorithms that keep track of some information regarding the landscape to adaptively change the step size based on curvature.

[4]Hyperparameters are parameters that are not optimized through training.

Version of June 16, 2023– Created June 16, 2023 - 07:37

One such algorithm is RMSprop:

$$\mathbf{s}_t = \beta\, \mathbf{s}_{t-1} + (1-\beta)[\nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta})]^2 \,, \tag{2.4a}$$

$$\Delta\boldsymbol{\theta}_t = -\frac{\eta_t}{\sqrt{\mathbf{s}_t + \epsilon}} \nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}) \,, \tag{2.4b}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \Delta\boldsymbol{\theta}_t \,. \tag{2.4c}$$

Here, $\mathbf{s}_t$ is essentially keeping a running average of the squared gradient over the last few time steps (β is typically taken to be around 0.9). This is referred to as the second moment of the gradient since it is a local estimate of $\mathbb{E}\big[[\nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta})]^2\big]$. RMSprop then uses the information from this estimate to adapt the step size to the curvature of the energy landscape: the smaller the mean squared gradient, the larger the step size.

ADAM

A widely used, similar algorithm is ADAM, which combines different features of the previous methods:

$$\mathbf{m}_t = \beta_1\, \mathbf{m}_{t-1} + (1-\beta_1)\nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta}) \,, \tag{2.5a}$$

$$\mathbf{s}_t = \beta_2\, \mathbf{s}_{t-1} + (1-\beta_2)[\nabla_{\boldsymbol{\theta}} E(\boldsymbol{\theta})]^2 \,, \tag{2.5b}$$

$$\widehat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-(\beta_1)^t} \,, \tag{2.5c}$$

$$\widehat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1-(\beta_2)^t} \,, \tag{2.5d}$$

$$\Delta\boldsymbol{\theta}_t = -\frac{\eta_t}{\sqrt{\widehat{\mathbf{s}}_t + \epsilon}}\, \widehat{\mathbf{m}}_t \,, \tag{2.5e}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \Delta\boldsymbol{\theta}_t \,. \tag{2.5f}$$

In addition to the second moment, ADAM keeps track of the first moment, the "momentum of the particle." It also implements a correction to account for the fact that the moments are estimated through a running average. Typical values of the hyperparameters are $\beta_1 = 0.9$ and $\beta_2 = 0.99$.

These various methods each have their advantages and drawbacks. The choice of one over another (and many other variations) often comes down to the observed performance on the specific task, though ADAM usually performs better than the other methods mentioned here.

Version of June 16, 2023– Created June 16, 2023 - 07:37

2.1.3 Regularizers

It is common in machine learning to modify the cost function to include a regularizer[5]. This is usually an extra term that punishes models whose parameters are too large. Common choices are the L_2 norm,

$$E(\boldsymbol{\theta}) \to E(\boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\|_2^2 \,, \tag{2.6}$$

where $\|\boldsymbol{\theta}\|_2^2 = \sum_i \theta_i^2$ and $\lambda \geq 0$ is a hyperparameter, and the L_1 norm

$$E(\boldsymbol{\theta}) \to E(\boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\|_1 \,, \tag{2.7}$$

where $\|\boldsymbol{\theta}\|_1 = \sum_i \theta_i$.

Regularizers can be understood as constraining the model to a smaller region of parameter space, artificially reducing model complexity, and helping to prevent overfitting. They are also intimately related to Bayesian inference, which is the subject of the next section.

2.1.4 Bayesian inference

Bayesian inference provides a powerful alternative lens through which to view machine learning problems, especially for theoretical study, as we will see in Chapter 3.

Formulating a problem in Bayesian terms is initially similar to what we have seen so far: we have a dataset $\mathcal{D}(\mathbf{X}, \mathbf{Y})$ and a model that depends on a set of parameters $\boldsymbol{\theta}$. The first difference we encounter is that now we assume that there is some inherent uncertainty to the dataset. Then, we use the language of probability to describe this uncertainty. The central object of our discussion is the "likelihood function" $p(\mathcal{D}|\boldsymbol{\theta})$, which is the probability that we would observe dataset $\mathcal{D}$ given a model of parameters $\boldsymbol{\theta}$.

In this framework, fitting the model to the data means finding the values $\widehat{\boldsymbol{\theta}}$ of the parameters that maximize the probability of producing the observed dataset:

$$\widehat{\boldsymbol{\theta}}_{MLE} = \arg\max_{\boldsymbol{\theta}} p(\mathcal{D}|\boldsymbol{\theta}) \,. \tag{2.8}$$

Since the logarithm is a monotonic function, the same set of parameters $\widehat{\boldsymbol{\theta}}$ that maximizes the likelihood also maximizes its logarithm:

$$\widehat{\boldsymbol{\theta}}_{MLE} = \arg\max_{\boldsymbol{\theta}} \log p(\mathcal{D}|\boldsymbol{\theta}) \,,$$

[5]The term "regularization" is used to describe any one of a variety of procedures that act to prevent overfitting. Modifying the cost function is one commonly used option.

and working with this "log-likelihood" is often useful for calculations. This procedure is known as "maximum likelihood estimation" (MLE), and it is roughly analogous to our previous discussion of machine learning, where the goal was to minimize a cost function.

The next ingredient is the "prior distribution" $p(\boldsymbol{\theta})$. This contains any information we have about the parameters, independently of any measurements in the dataset. Remember that we are using the language of probability to describe knowledge and uncertainty. The prior can be "uninformative," if we do not have prior information about the parameters, or it can be "informative," if it expresses some kind of prior knowledge about the parameters. Common choices of informative priors are the Gaussian prior

$$p(\boldsymbol{\theta}|\lambda) = \prod_i \sqrt{\frac{\lambda}{2\pi}} \exp\{-\lambda\theta_i^2\} \,, \tag{2.9a}$$

and the Laplace prior

$$p(\boldsymbol{\theta}|\lambda) = \prod_i \frac{\lambda}{2} \exp\{-\lambda|\theta_i|\} \,, \tag{2.9b}$$

where λ is a hyperparameter.

Our goal is to extract information about the model's parameters from the observed data. We do so through the "posterior distribution", defined using Bayes' rule as

$$p(\boldsymbol{\theta}|\mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{\theta})\, p(\boldsymbol{\theta})}{\int \mathrm{d}\boldsymbol{\theta}' p(\mathcal{D}|\boldsymbol{\theta}')\, p(\boldsymbol{\theta}')} \,. \tag{2.10}$$

While the prior distribution encodes our knowledge about the parameters prior to taking into account the observed data, the posterior distribution includes the dataset into our calculation and uses it to update our knowledge of the parameters.

All that is left to do is find an appropriate estimate for the parameters. Common choices include the mean of the posterior distribution,

$$\langle \boldsymbol{\theta} \rangle = \int \mathrm{d}\boldsymbol{\theta}\, \boldsymbol{\theta}\, p(\boldsymbol{\theta}|\mathcal{D}) \,, \tag{2.11}$$

and the "maximum-a-posteriori" (MAP) estimate, defined in a similar way as the maximum likelihood estimator (2.8)

$$\widehat{\boldsymbol{\theta}}_{MAP} = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta}|\mathcal{D}) = \arg\max_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta}|\mathcal{D}) \,. \tag{2.12}$$

We can use equation (2.10) to write the MAP estimator as

$$\begin{aligned}\widehat{\boldsymbol{\theta}}_{MAP} &= \arg\max_{\boldsymbol{\theta}} \log p(\boldsymbol{\theta}|\mathcal{D}) \\ &= \arg\max_{\boldsymbol{\theta}} \log \left[p(\mathcal{D}|\boldsymbol{\theta})\, p(\boldsymbol{\theta})\right] \\ &= \arg\max_{\boldsymbol{\theta}} \left[\log p(\mathcal{D}|\boldsymbol{\theta}) + \log p(\boldsymbol{\theta})\right] .\end{aligned} \tag{2.13}$$

If we took the negative log-likelihood as our cost function,

$$E(\boldsymbol{\theta}) = -\log p(\mathcal{D}|\boldsymbol{\theta}) ,$$

the MAP estimator would be the set of parameters that minimize

$$\widehat{\boldsymbol{\theta}}_{MAP} = \arg\min_{\boldsymbol{\theta}} \left[E(\boldsymbol{\theta}) + R(\boldsymbol{\theta})\right] , \tag{2.14}$$

where $R(\boldsymbol{\theta}) = -\log p(\boldsymbol{\theta})$. Using a Gaussian prior (2.9a), we find that $R(\boldsymbol{\theta}) = \lambda \sum_i \theta_i^2$ is simply an L_2 regularizer, while a Laplace prior (2.9b) is equivalent to an L_1 regularizer, $R(\boldsymbol{\theta}) = \lambda \sum_i |\theta_i|$.

We can see that regularizers are not just a convenient way of improving a model's performance, but they are intimately linked to Bayesian priors, encoding our expectations of what the trained weights should look like. A Gaussian prior (and therefore an L_2 regularizer) is used when we expect many of the network's weights to be small. A Laplace prior (L_1 regularizer) encodes our belief that many of the weights will be zero after training. Both of these assumptions are often reasonable since many machine learning models have too many parameters for all of them to be equally important. We will discuss in more detail the relative importance of different parameters in Chapter 4.

2.2 Feed-Forward Neural Networks

Recent years have seen an increased interest in one specific class of models: deep neural networks (DNN). These models – and some of their variants, such as convolutional neural networks (CNN) and recursive neural networks (RNN) – have seen success after success when applied to tasks such as natural language processing, and speech and image recognition. DNNs perform especially well on large datasets and they seem to be able to extract relevant features from datasets with minimal input about what "relevant features" should look like. In essence, a DNN (most likely a

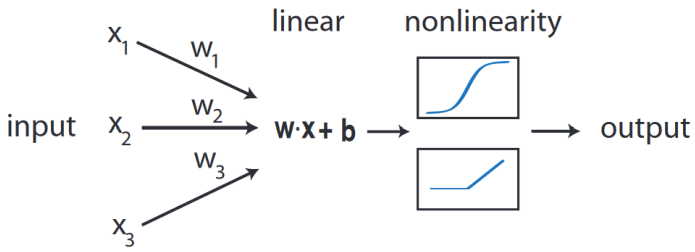

Figure 2.1: *Schematic representation of the an artificial "neuron" of the type common in machine learning. Source: Mehta et al., 2019 [10].*

CNN, in this case) could learn to recognize the presence of a cat in a picture without ever having been given much information about what a cat might look like.

This intriguing ability of DNNs to "learn on their own" has attracted the attention of many researchers, who would like to better understand these objects from a theoretical point of view.

2.2.1 Basics of FFNNs

Neural networks (NN) are a class of models often used in machine learning. As the name suggests, the structure of NNs was broadly inspired by biological brains[6], which are made up of a complicated mesh of neurons, each performing a relatively simple computational task. In a scenario that is probably familiar from statistical mechanics, highly complex behavior emerges from the interactions of a very large number of relatively simple units. Similarly, the basic building blocks of NNs, often referred to as "neurons," are simple units that receive an input in the form of a vector $\boldsymbol{x} = (x_1, \ldots, x_N)$, perform a linear transformation, multiplying the input components by a vector of weights $\boldsymbol{w} = (w_1, \ldots, w_N)$ and adding a constant b, then act on the result with a non-linear function $\varphi(\cdot)$, referred to as the "activation function." The full input-output function of a neuron is therefore

$$f(\boldsymbol{x}) = \varphi\left(\sum_{i=1}^{N} w_i x_i + b\right) . \tag{2.15}$$

It is common to define $\mathbf{x} = (1, \boldsymbol{x})$ and $\mathbf{w} = (b, \boldsymbol{w})$ and write the linear transformation as a dot product,

$$f(\mathbf{x}) = \varphi(\mathbf{w} \cdot \mathbf{x}) .$$

[6]To distinguish the two, the terms "artificial" and "biological" neural networks are often used.

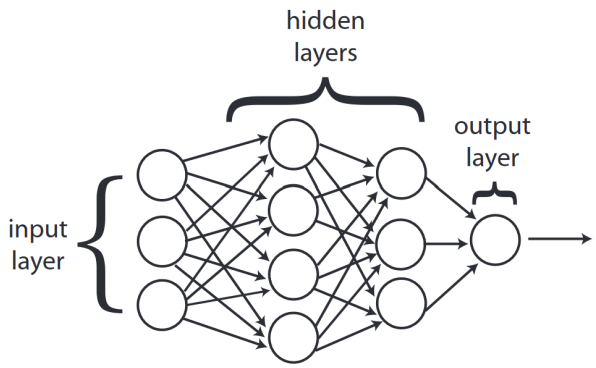

Figure 2.2: *Schematic representation of the structure of a feed-forward neural network. Source: Mehta et al., 2019 [10].*

Neurons can be connected in a variety of different ways, but a relatively simple and extremely common structure is found in Feed-Forward Neural Networks (FFNN). Here, neurons are organized in layers, each neuron using the outputs of the previous layer's neurons as inputs and sending its output onward to the next layer. The first layer is called the input layer, and it is fed the data that needs to be studied, the final layer (often a single neuron) is called the output layer, and the intermediate layers are called hidden layers. We can write the overall input-output function of an FFNN with L layers as

$$f(\mathbf{x}_0) = \varphi_L\Big(\mathbf{W}^L \cdot \varphi_{L-1}\Big(\mathbf{W}^{L-1} \cdot \varphi_{L-2}\big(\dots \mathbf{W}^2 \cdot \varphi_1(\mathbf{W}^1 \cdot \mathbf{x}_0)\dots\big)\Big)\Big)\,, \quad (2.16)$$

where $\mathbf{W}^\ell$ are matrices whose rows are the vectors $\mathbf{w}_i^\ell$, each containing all of the weights feeding into the i-th neuron of the ℓ-th hidden layer. We use $\mathbf{W} \cdot \varphi(\mathbf{x})$ to denote matrix-vector multiplication of $\mathbf{W}$ with $\varphi(\mathbf{x})$, and all activation functions $\varphi_\ell(\cdot)$ are understood as acting element-wise on their input vectors.

2.2.2 Activation functions

Common choices of activation functions include the step-function, the logistic function[7], the hyperbolic tangent, rectified linear units (ReLU), and variations on the latter (leaky ReLU, ELU).

[7] Logistic functions are sometimes referred to as "sigmoids," but this term is also used more broadly to denote any function roughly in the shape of an "S." By this definition, the hyperbolic tangent is also a sigmoid.

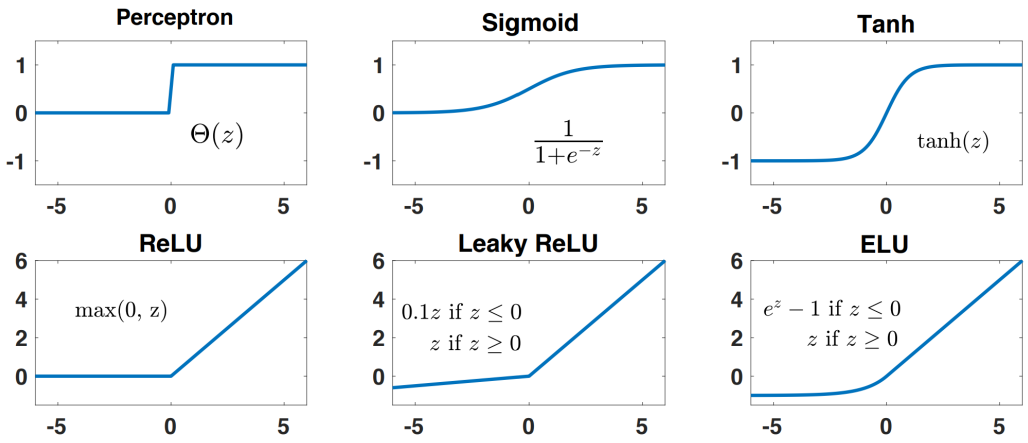

Figure 2.3: *Some common choices of activation functions in feed-forward neural networks. Source: Mehta et al., 2019 [10].*

Whether activation functions saturate for large or small inputs, i.e. they have horizontal asymptotes, will affect training. This is because, in large portions of parameter space, the neuron's output will be essentially constant, leading to flat regions in the energy landscape, which can present a challenge for optimization algorithms derived from gradient descent.

2.2.3 Training neural networks

By and large, neural networks are trained in much the same way as other supervised learning models: gradient descent. There is, however, one thing worth mentioning. Depending on the specific network architecture, the number of parameters can be enormous, and in modern applications it almost always is. Moreover, computing the derivative of the cost function with respect to weights deep in the network threatens to be computationally challenging. Employing sophisticated forms of gradient descent is of little use if we cannot efficiently compute the gradient. Fortunately, we can take advantage of the layered structure of FFNNs to more efficiently compute derivatives of the cost function.

The algorithm that allows us to do so is called "back-propagation," and it relies on the chain rule to efficiently compute derivatives with respect to parameters everywhere in the network. We are going to explain this algorithm, but let us first set up some convenient notation. Given neuron number i in layer ℓ, we denote its output, or "activation," as a_i^ℓ, while we use z_i^ℓ to refer to its "pre-activation," i.e. the linear combination of its

inputs given by

$$z_i^\ell = \sum_j W_{ij}^\ell a_j^{\ell-1} + b_j^\ell \,,$$

where W_{ij}^ℓ is the weight connecting neuron j in layer $\ell - 1$ with neuron i in layer ℓ. The input-output function of a network with L layers can be written as

$$\begin{aligned}
[f(\mathbf{x})]_i &= a_i^L \\
&= \varphi_L(z_i^L) \\
&= \varphi_L\left(\sum_j W_{ij}^L a_j^{L-1} + b_j^{L-1}\right) \\
&= \dots
\end{aligned}$$

where $[f(\mathbf{x})]_i$ is the i-th component of the output. Similarly, the cost function is explicitly a function of the output layer activations, and implicitly a function of all other variables,

$$E(\mathbf{W}) = E\Big(\{a_i^L\}\Big) = E\Big(\{a_i^L(z_i^L)\}\Big) = E\Big(\Big\{a_i^L(\{a_j^L\})\Big\}\Big) = \dots$$

With this notation in place, we are ready to describe how the back-propagation algorithm exploits the chain rule to efficiently compute the gradient of the cost function.

The back-propagation algorithm

1. **Activation at the input layer:** given an input, compute the activations a_i^1 at the input layer.

2. **Feed-forward:** due to the layered architecture of the network, a_i^1 completely determines all activations and pre-activations in the rest of the network. Compute all of these.

3. **Error at the top layer:** use the chain rule to compute the derivatives of the cost function w.r.t. the pre-activations of the output layer,

$$\frac{\partial E}{\partial z_i^L} = \frac{\partial E}{\partial a_i^L}\varphi_L'(z_i^L) \,, \tag{2.17}$$

   where $\varphi_\ell'(\cdot)$ denotes the derivative of the ℓ-th layer activation function w.r.t. its argument.

4. **Back-propagation:** the chain rule enables us to recursively relate derivatives w.r.t. $z_{\ell-1}$ to derivatives w.r.t. z_ℓ.

$$\frac{\partial E}{\partial z_i^{\ell-1}} = \sum_k \frac{\partial E}{\partial z_k^\ell} \frac{\partial z_k^\ell}{\partial z_i^{\ell-1}} = \sum_k \frac{\partial E}{\partial z_k^\ell} W_{ik}^\ell \varphi_\ell'(z_i^{\ell-1}) \,. \tag{2.18}$$

   We can use this to "move upstream" through the network and compute derivatives w.r.t. all pre-activations z_i^ℓ, $\ell = 1, \ldots, L$.

5. **Compute the gradient:** use the chain rule one last time to compute derivatives w.r.t. weights and biases everywhere in the network,

$$\frac{\partial E}{\partial W_{ij}^\ell} = \frac{\partial E}{\partial z_i^\ell} \frac{\partial z_i^\ell}{\partial W_{ij}^\ell} = \frac{\partial E}{\partial z_i^\ell} a_j^{\ell-1} \,, \tag{2.19a}$$

$$\frac{\partial E}{\partial b_i^\ell} = \frac{\partial E}{\partial z_i^\ell} \frac{\partial z_i^\ell}{\partial b_i^\ell} = \frac{\partial E}{\partial z_i^\ell} \,. \tag{2.19b}$$

Chapter 3

Recent Developments: Deep Linear Neural Networks

In this chapter, we will discuss three recent papers dealing with deep linear neural networks (DLNN) [4, 8, 14]. DLNNs are deep feed-forward neural networks that use linear activation functions. This severely limits their usefulness for real-world applications. In fact, DLNNs do not perform better than simple models without hidden layers. Nevertheless, DLNNs can be an interesting object of theoretical study: considering linear activation functions makes theoretical analysis more accessible, but still far from trivial. It is reasonable to imagine DLNNs being a good training ground, where we can develop theoretical machinery that could be applied to more realistic, non-linear neural network models in the future.

3.1 Back-Propagating Kernel Renormalization

A remarkable result in the study of DLNNs was obtained by Qianyi Li and Haim Sompolinsky (LS), who used an approach inspired by statistical mechanics to study the statistical properties of an ensemble of trained DLNNs [8]. The method they introduce, which they call Back-Propagating Kernel Renormalization (BPKR), consists of setting up an iterative procedure to calculate the partition function of the network's weights, which is then used to characterize the model's performance on previously unseen inputs. We will now discuss LS's procedure in some detail.

3.1.1 Setting the scene

Let us begin by setting up the notation. For simplicity, consider a single-output DLNN[1]. This network has L hidden layers with N hidden units each, and it takes vectors of dimension N_0 as inputs. We denote the training inputs as $\mathbf{x}^\mu = (x_1^\mu, \dots, x_{N_0}^\mu)$ and their target labels as y^μ, where the different pairs of training inputs and labels are indexed by $\mu = 1, \dots, P$. Throughout this chapter, Greek letters will be used to index training examples, while Latin letters will index neurons in a layer. We denote as W_{ij}^ℓ the weight connecting the j-th neuron in layer $\ell - 1$ with the i-th neuron in layer ℓ. We will treat $\ell = 0$ as the input layer. Since there is only one output, the weight connecting the i-th neuron in the final hidden layer with the output will be denoted as W_i^{L+1}. The weights of the network will be collectively referred to as $\mathbf{W}$.

For the remainder of this chapter, summation over repeated indices, Latin or Greek, will be left implicit unless otherwise stated. The network's output given input vector $\mathbf{x} = (x_1, \dots, x_{N_0})$ is

$$f(\mathbf{x}, \mathbf{W}) = W_{i_L}^{L+1} W_{i_L i_{L-1}}^{L} W_{i_{L-1} i_{L-2}}^{L-1} \dots W_{i_2 i_1}^{2} W_{i_1 i_0}^{1} x_{i_0} \,, \tag{3.1}$$

and we use a quadratic loss function with an L_2 regularizer, so the cost function is

$$E(\mathbf{W}) = \frac{1}{2} \sum_{\mu=1}^{P} (f(\mathbf{x}^\mu, \mathbf{W}) - y^\mu)^2 + \frac{T}{2\sigma^2} \left(\sum_{\ell=1}^{L} W_{ij}^\ell W_{ij}^\ell + W_i^{L+1} W_i^{L+1} \right) , \tag{3.2}$$

where the final term, proportional to the sum of the squared weights, is the regularizer. T and σ^2 are two hyperparameters of the network, referred to as "temperature" and "weight noise," respectively.

LS's approach to studying this system is rooted in physics. In particular, it is inspired by statistical mechanics[2]. In Chapter 2, when we discussed gradient descent, we mentioned that some variants of stochastic gradient descent can be seen as a discretized Langevin equation of motion. In this analogy, the weights $\mathbf{W}$ of the network are dynamical variables, and the cost function $E(\mathbf{W})$ defines an energy landscape. After we set initial values for the parameters, the optimization procedure will look for the global minimum of the energy. Since the procedure is in part stochastic, we can expect an ensemble of networks initialized at different points of the

[1] The theory can be generalized to larger output dimensions.

[2] The idea of using statistical mechanics to study machine learning and neural networks is not new. See, for example, [2, 3].

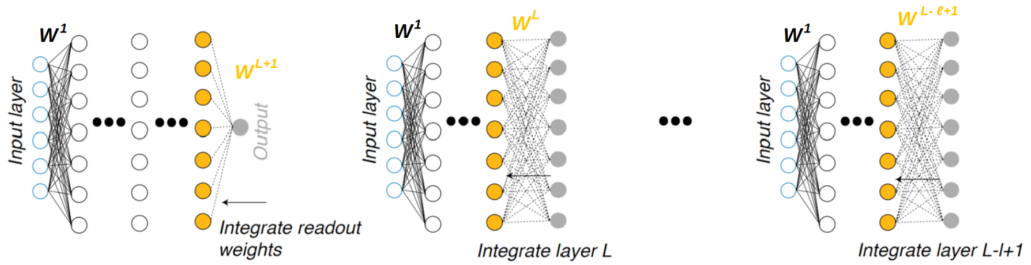

Figure 3.1: *Schematic representation of the BPKR procedure: we begin by integrating the layer of weights feeding into the network's output, then proceed to move "upstream" through the network, integrating the weights one layer at a time. Adapted from Li and Sompolinsky, 2021 [8].*

energy landscape to eventually reach an equilibrium distribution around the global minimum of the cost function.

LS use statistical mechanics to investigate the properties of an ensemble of networks all trained on the same set of examples. The equilibrium distribution is the Gibbs distribution

$$P(\mathbf{W}) = \frac{\mathrm{e}^{-\beta E(\mathbf{W})}}{Z}\,, \tag{3.3}$$

where $\beta = T^{-1}$, and the most important object to study is, of course, the partition function

$$Z = A \int \mathrm{d}\mathbf{W}\, \mathrm{e}^{-\beta E(\mathbf{W})}\,, \tag{3.4}$$

where

$$\mathrm{d}\mathbf{W} = \left(\prod_{i=1}^{N}\prod_{j=1}^{N_0} \mathrm{d}W_{ij}^{1}\right) \times \prod_{\ell=2}^{L}\left(\prod_{i=1}^{N}\prod_{j=1}^{N} \mathrm{d}W_{ij}^{\ell}\right) \times \left(\prod_{i=1}^{N} \mathrm{d}W_{i}^{L+1}\right),$$

and A can be any constant since its value has no effect on the "physical" behavior of the system.

Direct calculation of the partition function is highly non-trivial, so the goal of LS's paper is to set up an iterative procedure to perform the integrations layer by layer and compute Z in the wide network limit ($N \to \infty$). Following LS, our strategy will be to perform the integrals layer by layer, starting from the output and proceeding backward through the network.

3.1.2 Integrating over the final layer of weights

We begin by making our notation more compact by defining the Gaussian integration measures

$$\mathcal{D}W^1 = \left(\prod_{i=1}^{N}\prod_{j=1}^{N_0} \mathrm{d}W_{ij}^1\right)\left(\frac{N_0}{2\pi\sigma^2}\right)^{\frac{NN_0}{2}} \exp\left\{-\frac{N_0}{2\sigma^2}W_{ij}^1 W_{ij}^1\right\}, \tag{3.5a}$$

$$\mathcal{D}W^\ell = \left(\prod_{i=1}^{N}\prod_{j=1}^{N} \mathrm{d}W_{ij}^\ell\right)\left(\frac{N}{2\pi\sigma^2}\right)^{\frac{N^2}{2}} \exp\left\{-\frac{N}{2\sigma^2}W_{ij}^\ell W_{ij}^\ell\right\}, \tag{3.5b}$$

for $\ell = 2, \dots, L$, and

$$\mathcal{D}W^{L+1} = \left(\prod_{i=1}^{N} \mathrm{d}W_{i}^{L+1}\right)\left(\frac{N}{2\pi\sigma^2}\right)^{\frac{N}{2}} \exp\left\{-\frac{N}{2\sigma^2}W_{i}^{L+1} W_{i}^{L+1}\right\}. \tag{3.5c}$$

This allows us to write the partition function as

$$Z = \int \mathcal{D}W^1 \cdots \int \mathcal{D}W^{L+1}\, \exp\left\{-\frac{\beta}{2}\sum_{\mu=1}^{P}\left(W_{i_L}^{L+1}W_{i_L i_{L-1}}^{L}\dots W_{i_1 i_0}^{1}x_{i_0}^{\mu} - y^\mu\right)^2\right\}. \tag{3.6}$$

Our strategy to calculate the partition function is to set up an iterative procedure to perform the integrals over the weights one layer at a time, starting from the last layer and moving "upstream" through the network. In order to keep our notation simple, we start by writing $Z = \int \mathcal{D}W^1 \cdots \int \mathcal{D}W^L\, Z_L$, where

$$Z_L = \int \mathcal{D}W^{L+1}\, \exp\left\{-\frac{\beta}{2}\sum_{\mu=1}^{P}\left(W_{i_L}^{L+1}W_{i_L i_{L-1}}^{L}\dots W_{i_1 i_0}^{1}x_{i_0}^{\mu} - y^\mu\right)^2\right\}. \tag{3.7}$$

Next, we introduce the auxiliary integration variables t^μ ($\mu = 1, \dots, P$) to perform a Hubbard-Stratonovich transformation,

$$Z_L = \int \mathcal{D}W^{L+1} \int \mathcal{D}t \exp\left\{-it^\mu\left(W_{i_L}^{L+1}W_{i_L i_{L-1}}^{L}\dots W_{i_1 i_0}^{1}x_{i_0}^{\mu} - y^\mu\right)\right\}. \tag{3.8}$$

We have once again introduced a Gaussian integration measure to make the notation more compact:

$$\mathcal{D}t = \left(\prod_{\mu=1}^{P}\mathrm{d}t^\mu\right)\left(\frac{1}{2\pi\beta}\right)^{\frac{P}{2}} \exp\left\{-\frac{1}{2\beta}t^\mu t^\mu\right\}.$$

Going forward, it is useful to define the activation of neuron i in layer ℓ when input vector x^μ is fed into the network. This is

$$x_i^{\ell,\mu} = W_{ii_\ell}^{\ell} W_{i_\ell i_{\ell-1}}^{\ell-1} \dots W_{i_2 i_1}^{1} x_{i_1}^{\mu} \,. \tag{3.9}$$

Performing the integration over the readout weights is now straightforward: it is a Gaussian integral.

$$\begin{aligned}
&\int \mathcal{D}W^{L+1} \exp\Big\{ -it^\mu W_{i_L}^{L+1} x_{i_L}^{L,\mu} \Big\} = \\
&= \int \left(\prod_{i=1}^{N} \mathrm{d}W_i^{L+1} \right) \left(\frac{N}{2\pi\sigma^2} \right)^{\frac{N}{2}} \exp\Big\{ -\frac{N}{2\sigma^2} W_i^{L+1} W_i^{L+1} - it^\mu W_i^{L+1} x_i^{L,\mu} \Big\} = \\
&\qquad = \exp\Big\{ -\frac{\sigma^2}{2N} t^\mu x_i^{L,\mu} x_i^{L,\nu} t^\nu \Big\} \,.
\end{aligned} \tag{3.10}$$

We can now define the ℓ-th layer kernel matrix

$$K_\ell^{\mu\nu} = \frac{\sigma^2}{N} x_i^{\ell,\mu} x_i^{\ell,\nu} \tag{3.11}$$

and write our result as

$$\begin{aligned}
Z_L &= \int \mathcal{D}t \exp\Big\{ it^\mu y^\mu - \frac{1}{2} t^\mu K_L^{\mu\nu} t^\nu \Big\} = \\
&= \left(\frac{1}{2\pi\beta} \right)^{\frac{P}{2}} \int \left(\prod_{\mu=1}^{P} \mathrm{d}t^\mu \right) \exp\Big\{ it^\mu y^\mu - \frac{1}{2} t^\mu \left(K_L^{\mu\nu} + \frac{\delta^{\mu\nu}}{\beta} \right) t^\nu \Big\} \,.
\end{aligned} \tag{3.12}$$

The integral over t is Gaussian, and we can solve it to find $Z_L = \mathrm{e}^{-\beta H_L}$, where

$$\begin{aligned}
\beta H_L &= \frac{1}{2} y^\mu (\Gamma_L^{-1})^{\mu\nu} y^\nu + \frac{1}{2} \log\det\left(\Gamma_L\right) \,, \\
\Gamma_L^{\mu\nu} &= K_L^{\mu\nu} + \frac{\delta^{\mu\nu}}{\beta} \,.
\end{aligned} \tag{3.13}$$

3.1.3 Integrating over the next layer of weights

We turn our attention to

$$Z_{L-1} = \int \mathcal{D}W^L Z_L = \int \mathcal{D}W^L \int \mathcal{D}t \exp\Big\{ it^\mu y^\mu - \frac{1}{2} t^\mu K_L^{\mu\nu} t^\nu \Big\}. \tag{3.14}$$

We used Eq. (3.12) instead of Eq. (3.13), as this makes calculations simpler going forward. Notice from the definition of the Kernel matrix (3.11) that $K_L^{\mu\nu}$ is a function of W^L. Making this dependence explicit, we write

$$Z_{L-1} = \int \mathcal{D}t \; \mathrm{e}^{it^\mu y^\mu} \int \mathcal{D}W^L \exp\left\{ -W^L_{ji} \left[\frac{\sigma^2}{2N} t^\mu t^\nu x_j^{L-1,\mu} x_k^{L-1,\nu} \right] W^L_{ki} \right\} .$$

The integral over the L-th layer of weights is once again a Gaussian integral, and solving it gives

$$Z_{L-1} = \int \mathcal{D}t \; \exp\left\{ it^\mu y^\mu - \frac{N}{2} \log\det \Lambda_{L-1} \right\} , \tag{3.15}$$

where

$$(\Lambda_{L-1})_{ij} = \delta_{ij} + \frac{\sigma^4}{2N^2} t^\mu t^\nu x_i^{L-1,\mu} x_j^{L-1,\nu} . \tag{3.16}$$

It is worth taking a careful look at this matrix. Let us define the new matrix

$$(\lambda_{L-1})_{ij} = \frac{\sigma^4}{2N^2} t^\mu t^\nu x_i^{L-1,\mu} x_j^{L-1,\nu} , \tag{3.17}$$

so that

$$(\Lambda_{L-1})_{ij} = \delta_{ij} + (\lambda_{L-1})_{ij} .$$

This new matrix has the interesting property

$$(\lambda_{L-1}^2)_{ij} = (\mathrm{Tr}\, \lambda_{L-1})(\lambda_{L-1})_{ij},$$

which in turn implies that

$$\mathrm{Tr}\big(\lambda_{L-1}^n\big) = (\mathrm{Tr}\, \lambda_{L-1})^n . \tag{3.18}$$

Here, λ_{L-1}^n denotes repeated matrix multiplication:

$$\lambda_{L-1}^n = \underbrace{\lambda_{L-1} \cdot \lambda_{L-1} \cdot \ldots \cdot \lambda_{L-1}}_{n \text{ times}} .$$

This seemingly innocuous property is actually remarkable. The determinant of Λ_{L-1} can be written[3] as a power series in terms of traces of powers of λ_{L-1}:

$$\det \Lambda_{L-1} = 1 + \mathrm{Tr}\, \lambda_{L-1} + \frac{1}{2}\Big[(\mathrm{Tr}\, \lambda_{L-1})^2 - \mathrm{Tr}(\lambda_{L-1}^2) \Big] + \ldots$$

[3] See the Cayley-Hamilton theorem.

Equation (3.18) implies that only the first term of the series is non-zero, providing us with the very useful result

$$\det \Lambda_{L-1} = 1 + \operatorname{Tr} \lambda_{L-1} \,. \tag{3.19}$$

Substituting this result into Eq. (3.15), we have

$$\begin{aligned} Z_{L-1} &= \int \mathcal{D}t \, \exp\left\{ i t^\mu y^\mu - \frac{N}{2} \log(1 + \operatorname{Tr} \lambda_{L-1}) \right\} = \\ &= \int \mathcal{D}t \, \exp\left\{ i t^\mu y^\mu - \frac{N}{2} \log\left(1 + \frac{\sigma^2}{N} t^\mu K_{L-1}^{\mu\nu} t^\nu \right) \right\} . \end{aligned} \tag{3.20}$$

Comparing Eq. (3.20) and Eq. (3.12), there is a striking similarity. Indeed, in the wide network limit ($N \to \infty$), they would be identical. Unfortunately, taking this limit right away would be too drastic an approximation, and if we did take the limit we would miss out on much of the complex behavior in the resulting theory. We will come back to this limit in Section 3.3. One reason why taking the limit is not as straightforward as it looks is that Z_{L-1} is not the complete integral we are studying: we cannot simply take N to infinity without this affecting the integrals over all other layers of weights. Therefore, LS had to come up with an alternative way of bringing the kernel matrix out of the logarithm.

The strategy they employ is to introduce a new integral and a Dirac delta function,

$$\begin{aligned} Z_{L-1} = \int \mathrm{d}m_{L-1} \int \mathcal{D}t \, \delta\left(m_{L-1} - \frac{\sigma^2}{N} t^\mu K_{L-1}^{\mu\nu} t^\nu \right) \\ \exp\left\{ i t^\mu y^\mu - \frac{N}{2} \log(1 + m_{L-1}) \right\} , \end{aligned} \tag{3.21}$$

and then use one of the integral representations of the Dirac delta to write

$$\begin{aligned} Z_{L-1} = \int \mathrm{d}u_{L-1} \int \mathrm{d}m_{L-1} \int \mathcal{D}t \\ \exp\left\{ i t^\mu y^\mu + \frac{N}{2\sigma^2} u_{L-1} m_{L-1} - \frac{\sigma^2}{N} t^\mu \left(u_{L-1} K_{L-1}^{\mu\nu} \right) t^\nu - \frac{N}{2} \log(1 + m_{L-1}) \right\} . \end{aligned} \tag{3.22}$$

The t-integral is now in the same form as Eq. (3.12), with the only difference that the kernel is rescaled by the auxiliary variable u_{L-1}. If we wanted to keep integrating over the remaining weights, we would start

from here and repeat the same steps to go from Z_{L-1} to Z_{L-2}. Before we do that, however, it is useful to see what it would take to solve all three integrals in Eq. (3.22).

We can solve the Gaussian integral right away[4], which leaves us with

$$Z_{L-1} = \int \mathrm{d}u_{L-1} \int \mathrm{d}m_{L-1} \exp\left\{ \frac{N}{2\sigma^2} u_{L-1} m_{L-1} - \frac{N}{2} \log(1 + m_{L-1}) - \right. \\ \left. - \frac{1}{2} y^\mu (\Gamma_{L-1}^{-1})^{\mu\nu} y^\nu - \frac{1}{2} \log\det(\Gamma_{L-1}) \right\} , \quad (3.23)$$

where

$$\Gamma_{L-1}^{\mu\nu} = u_{L-1} K_{L-1}^{\mu\nu} + T\delta^{\mu\nu} .$$

Next, we can use saddle-point approximations to solve the two remaining integrals. We consider the limit $N \to \infty$ and $P \to \infty$, keeping the ratio $\alpha = P/N$ constant. First, we integrate over m_{L-1}. The saddle-point equation is

$$1 + m_{L-1} = \sigma^2 u_{L-1}^{-1} .$$

This leaves us with the integral over u_{L-1}. Its saddle point equation is

$$N(1 - \sigma^{-2} u_{L-1}) = \mathrm{Tr}\Big(\Gamma_{L-1}^{-1} u_{L-1} K_{L-1} \Big) - y^\mu (\Gamma_{L-1}^{-2})^{\mu\nu} u_{L-1} K_{L-1}^{\nu\rho} y^\rho , \quad (3.24)$$

which becomes rather simpler in the limit $T \to 0$:

$$1 - \sigma^{-2} u_{L-1} = \alpha(1 - u_{L-1}^{-1} r_{L-1}) , \quad (3.25)$$

where we have defined

$$r_\ell = P^{-1} y^\mu (K_\ell^{-1})^{\mu\nu} y^\nu . \quad (3.26)$$

LS call this "mean squared readout" because it is equal to $(\sigma^2\alpha)^{-1}$ times the squared norm of the vector of output weights that would be needed to read out the target labels y^μ directly from layer ℓ. Using these saddle-point equations in the integral (3.23) would give us a result for Z_{L-1}, which will depend on the solution of Eq. (3.24).

[4] We assume here that $P < N$, though the calculation can be generalized.

3.1.4 Iterative integration of the remaining weights

Moving on to the next layer of weights, we can start from Eq. (3.22) and follow the same steps as before to find

$$Z_{L-2} = \int \mathrm{d}u_{L-1} \int \mathrm{d}m_{L-1} \int \mathrm{d}u_{L-2} \int \mathrm{d}m_{L-2} \exp\left\{ \frac{N}{2\sigma^2} u_{L-1} m_{L-1} + \right.$$
$$\left. + \frac{N}{2\sigma^2} u_{L-2} m_{L-2} + \frac{N}{2} \log(1 + m_{L-1}) + \frac{N}{2} \log(1 + m_{L-2}) \right\}$$
$$\int \mathcal{D}t \exp\left\{ i t^\mu y^\mu - \frac{\sigma^2}{N} t^\mu \left(u_{L-2} u_{L-1} K^{\mu\nu}_{L-2} \right) t^\nu \right\} . \quad (3.27)$$

The kernel is now rescaled by two auxiliary variables: u_{L-1} and u_{L-2}. We can once again solve the Gaussian integral right away and use saddle-point approximations for the remaining four integrals. The saddle point equations for m_{L-1} and m_{L-2} read

$$1 + m_{L-1} = \sigma^2 u_{L-1}^{-1} , \quad (3.28a)$$
$$1 + m_{L-2} = \sigma^2 u_{L-2}^{-1} , \quad (3.28b)$$

while the saddle-point equations for u_{L-1} and u_{L-2} read

$$N(1 - \sigma^{-2} u_{L-1}) = \mathrm{Tr}\left(\Gamma_{L-2}^{-1} u_{L-2} u_{L-1} K_{L-2} \right) -$$
$$- y^\mu (\Gamma_{L-2}^{-2})^{\mu\nu} u_{L-2} u_{L-1} K^{\nu\rho}_{L-2} y^\rho , \quad (3.29a)$$

$$N(1 - \sigma^{-2} u_{L-2}) = \mathrm{Tr}\left(\Gamma_{L-2}^{-1} u_{L-2} u_{L-1} K_{L-2} \right) -$$
$$- y^\mu (\Gamma_{L-2}^{-2})^{\mu\nu} u_{L-2} u_{L-1} K^{\nu\rho}_{L-2} y^\rho , \quad (3.29b)$$

where

$$\Gamma^{\mu\nu}_{L-2} = u_{L-2} u_{L-1} K^{\mu\nu}_{L-2} + T\delta^{\mu\nu} .$$

These equations satisfy $u_{L-1} = u_{L-2}$, therefore Z_{L-2} will only depend on the the value of u_{L-2} that solves

$$N(1 - \sigma^{-2} u_{L-2}) = \mathrm{Tr}\left(\Gamma_{L-2}^{-1} u_{L-2}^2 K_{L-2} \right) - y^\mu (\Gamma_{L-2}^{-2})^{\mu\nu} u_{L-2}^2 K^{\nu\rho}_{L-2} y^\rho ,$$

which when $T \to 0$ becomes

$$1 - \sigma^{-2} u_{L-2} = \alpha (1 - u_{L-2}^{-2} r_{L-2}) . \quad (3.30)$$

(We use u_{L-2}^n to refer to u_{L-2} raised to the n-th power.)

The procedure can be iterated to integrate over all the weights in the neural network. After solving all of the integrals in the weights and the Gaussian integral in t^μ, we would be left with $2L$ integrals in the variables $m_0, \ldots, m_{L-1}$ and $u_0, \ldots, u_{L-1}$. Using saddle-point approximations, we would find that the final solution depends only on the value of a single parameter, u_0, which is fixed by a saddle-point equation. Notice that this procedure allows us to take the $N \to \infty$ limit at the end, thereby avoiding the issues we would have encountered if we had tried to take the limit directly in Eq. (3.20).

In the limit $T \to 0$, the resulting partition function for wide linear neural networks of arbitrary depth is $Z = \mathrm{e}^{-\beta H_0}$, where

$$\beta H_0 = \frac{1}{2} y^\mu (u_0^{-L} K_0^{-1})^{\mu\nu} y^\nu + \frac{1}{2} \log\det\Big(u_0^L K_0\Big) - \frac{LN}{2} \log u_0 + \frac{LN}{2\sigma^2} u_0 \,, \tag{3.31}$$

and u_0, referred to as the "kernel renormalization factor" by LS, is the solution to

$$1 - \sigma^{-2} u_0 = \alpha (1 - u_0^{-L} r_0) \,. \tag{3.32}$$

3.1.5 Results and thoughts on the BPKR

The ultimate goal of the BPKR is to study the statistics of an ensemble of trained networks. Given a new input vector $\mathbf{x}$, we want to study the statistical moments of $f(\mathbf{x}, \mathbf{W})) \equiv f(\mathbf{x})$, i.e. the output of a network with weights $\mathbf{W}$ given input $\mathbf{x}$. We will use $\mathbf{x}$ to denote a new input vector, not part of the training set, while $\mathbf{x}^\mu$ will be the training vectors, indexed by $\mu = 1, \ldots, P$ (or any other Greek index). Our strategy is one commonly used in statistical mechanics: we define a generalized form of the partition function (3.4),

$$Z(\lambda, \mathbf{x}) = \int \mathrm{d}\mathbf{W}\, \mathrm{e}^{-\beta E(\mathbf{W}) - i\lambda f(\mathbf{x})} \,, \tag{3.33}$$

which allows us to write the first two moments of $f(\mathbf{x})$ as

$$\langle f(\mathbf{x}) \rangle = i \frac{\partial}{\partial \lambda} \log Z(\lambda, \mathbf{x}) \bigg|_{\lambda=0} \,, \tag{3.34a}$$

$$\mathrm{Var}[f(\mathbf{x})] = \langle (\delta f(\mathbf{x}))^2 \rangle = - \frac{\partial^2}{\partial \lambda^2} \log Z(\lambda, \mathbf{x}) \bigg|_{\lambda=0} \,. \tag{3.34b}$$

We can compute this generalized partition function in much the same way as before, and LS are able to show that the expectation value of the net-

work's output is

$$\langle f(\mathbf{x})\rangle = \frac{\sigma^2}{N_0}\, x_i\, x_i^\mu (K_0^{-1})^{\mu\nu} y^\nu \,, \tag{3.35}$$

where x_i and x_i^μ are the i-th components of the new input vector $\mathbf{x}$ and the μ-th training vector, respectively. The variance of the output is

$$\mathrm{Var}[f(\mathbf{x})] = u_0^L \left[\frac{\sigma^2}{N_0} x_i x_i - \frac{\sigma^4}{N_0^2}\, x_i\, x_i^\mu (K_0^{-1})^{\mu\nu} x_j^\nu\, x_j \right] . \tag{3.36}$$

Notice that because the definition (3.11) of the kernel matrix K_0 contains a factor σ^2/N_0, the two terms in (3.36) are the same order in $1/N_0$.

Li and Sompolinsky's method uses an iterative procedure to write down a sequence of effective Hamiltonians for the weights upstream of a hidden layer ℓ. In the $T \to 0$ limit, these are given by

$$\begin{aligned}\beta H_\ell\Big(\{W^k\}_{k\le\ell}\Big) = \frac{1}{2} y^\mu \left(u_\ell^{\ell-L} K_\ell^{-1}\right)^{\mu\nu} y^\nu + \\ + \frac{1}{2}\log\det\Big(u_\ell^{L-\ell} K_\ell\Big) - \frac{(L-\ell)N}{2}\log u_\ell + \frac{(L-\ell)N}{2\sigma^2} u_\ell \,. \end{aligned} \tag{3.37}$$

It is not hard to see a parallel with the renormalization group. In a typical renormalization group study of a physical system, microscopic degrees of freedom are integrated over and the resulting effective Hamiltonian is cast in a similar mathematical form as the full Hamiltonian of the system, some physical quantities having been rescaled by the procedure. Here, it seems that the layers of the neural network that are closest to the output are playing the role of "microscopic" degrees of freedom, and the kernel matrix is the quantity being renormalized. Fully developing and exploring this analogy has the potential to lead to interesting results.

The iterative nature of the BPKR makes it a promising candidate for studying the properties of hidden layers in deep neural networks. However, attempting to generalize this method to the case non-linear activation function is not trivial. In particular, the linearity of the activation function ensures that all integrals over the weight throughout the procedure are Gaussian, as can be seen in the steps leading to Eq. (3.15). Applying a similar procedure beyond the case of linear activation functions risks leading to unsolvable integrals. Any attempt at solving this problem needs to carefully avoid ruining the special property of λ_ℓ, Eq. (3.18), or it could become impossible to continue the integration beyond the first layer. Moreover, when a source term is included in the partition function, as in (3.33), the integrals only remain Gaussian if $f(\mathbf{x})$ is linear in the weights.

A notable feature of the BPKR is that it can be generalized to the case of multiple outputs. The overall derivation and the resulting theory are essentially unchanged, the only difference being that the mean square readout r_ℓ and the kernel renormalization factor u_ℓ are both promoted from scalars to matrices.

3.2 Validity of the BPKR: Exact Calculations

Does the BPKR provide an accurate description of a DLNN's behavior? In this section we will present an alternative approach to the study of DLNNs, based on Bayesian inference rather than statistical mechanics. We will report results from two papers [4, 14] and see how this alternative approach allows us to recover the BPKR results.

3.2.1 Introducing the Meijer G-functions

We have seen that LS study the properties of an ensemble of trained networks. The object of their study is the statistical distribution of outputs found when the same input is passed through different networks, all of them trained on the same dataset. A related investigation was carried out by Jacob A. Zavatone-Veth and Cengiz Pehlevan (Z-VP). In their paper [14], Z-VP consider a Bayesian neural network[5] with linear activation functions and Gaussian priors over the weights. For the purposes of this thesis, we can think of an ensemble of deep linear neural networks in which all of the weights are independently drawn from a Gaussian distribution. Z-VP consider networks with a layer-dependent number of neurons N_ℓ and noise σ_ℓ, and output dimension of one. in contrast to LS, they are not concerned with training. Instead, given the ensemble of untrained networks and an input $\mathbf{x}$, they ask about the distribution $p(\mathbf{x}^L|\mathbf{x})$ of the neuron activations at the last hidden layer before the output.

Remarkably, they find an exact answer, expressed as a Meijer G-function:

$$p(\mathbf{x}^L|\mathbf{x}) = \left[\prod_{\ell=1}^{L-1}\Gamma\left(\frac{N_\ell}{2}\right)\right]^{-1}\left(\pi\|\mathbf{x}\|^2\right)^{-\frac{N_L}{2}}\left[\prod_{\ell=1}^{L}\frac{2\sigma_\ell^2}{N_{\ell-1}}\right]^{-\frac{N_L}{2}} G^{L,0}_{0,L}\left(\left[\prod_{\ell=1}^{L}\frac{N_{\ell-1}}{2\sigma_\ell^2}\right]\frac{\left\|\mathbf{x}^L\right\|^2}{\|\mathbf{x}\|^2}\,\middle|\, \begin{matrix} - \\ 0, \frac{N_1-N_L}{2}, \ldots, \frac{N_{L-1}-N_L}{2}\end{matrix}\right). \quad (3.38)$$

[5] Bayesian neural networks are a type of neural networks where weights do not have a specific value, but rather an associated probability distribution.

Meijer G-functions are rather obscure but extremely well studied special functions, defined as a line integral of Gamma functions in the complex plane:

$$G_{p,q}^{m,n}\left(x \,\middle|\, \begin{matrix}\mathbf{a}\\ \mathbf{b}\end{matrix}\right) = \frac{1}{2\pi i}\int_{\mathcal{C}} \mathrm{d}s\; x^s\, \frac{\prod_{j=1}^{m}\Gamma(b_j - s)\prod_{k=1}^{n}\Gamma(1-a_k+s)}{\prod_{k=n+1}^{p}\Gamma(a_k - s)\prod_{j=m+1}^{q}\Gamma(1-b_j+s)}\,,$$

where $\mathbf{a} = (a_1, \ldots, a_n, a_{n+1}, \ldots, a_p)$ and $\mathbf{b} = (b_1, \ldots, b_m, b_{m+1}, \ldots, b_q)$, $\Gamma(x)$ is the Gamma function, and $\mathcal{C}$ is the Mellin-Barnes contour (see Appendix A). Meijer G-functions are extremely general and reduce to many simpler special functions for specific choices of the parameters $n, m, p, q, \mathbf{a}$, and $\mathbf{b}$. They are very well studied: plenty of identities and formulas are known involving Meijer G-functions.

3.2.2 An exact formula for the partition function

Z-VP's result was later used by Boris Hanin and Alexander Zlokapa (HZ) to obtain a full description of the behavior of trained deep linear neural networks of any width and depth, as long as they're limited to a single neuron in the output layer. In their paper [4], rather than dealing with the intricacies of gradient descent-based training, HZ consider Bayesian inference (Section 2.1) as a form of training, i.e. a way to fit the model's parameters to a dataset. In their Bayesian framework, the partition function is defined as the expectation value of the cost function w.r.t. the prior probability distribution of the weights:

$$Z(\lambda, \mathbf{x}) = \mathbb{E}_{\text{prior}}[\exp\{-\beta E(\mathbf{W}) - i\lambda f(\mathbf{x})\}]\,. \tag{3.39}$$

Using Gaussian priors over the weights and a quadratic loss as the cost function, this formula is fully equivalent to Eq. (3.33).

HZ manage to obtain an exact, rather complicated formula for the partition function (3.39) in the zero temperature limit, written in terms of an infinite sum of Meijer G-functions:

$$Z(\lambda, \mathbf{x}) = \left(\frac{4\pi}{\|\boldsymbol{\theta}^*\|^2}\right)^{\frac{P}{2}} \mathrm{e}^{-i\lambda\theta_i^* x_i}\left[\prod_{\ell=1}^{L}\Gamma\left(\frac{N_\ell}{2}\right)\right]^{-1}\sum_{k=0}^{\infty}\frac{(-\|\lambda\mathbf{x}_\perp\|^2)^k}{k!\,4^k}$$
$$\left[\prod_{\ell=0}^{L}\frac{2\sigma^2}{N_\ell}\right]^k G_{0,L+1}^{L+1,0}\left(\left[\prod_{\ell=0}^{L}\frac{N_\ell}{2\sigma^2}\right]\|\boldsymbol{\theta}^*\|^2\,\middle|\, \begin{matrix}-\\ \frac{P}{2}, \frac{N_1}{2}+k, \ldots, \frac{N_L}{2}+k\end{matrix}\right). \tag{3.40}$$

One important new object in this formula is $\boldsymbol{\theta}^* = (\theta_1^*, \ldots, \theta_{N_0}^*)$, defined as

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \sum_{\mu=1}^{P} \left(\theta_i x_i^\mu - y^\mu\right)^2 . \tag{3.41}$$

In other words, $\boldsymbol{\theta}^*$ is the vector of weights that, given a linear network without hidden layers, would result on the best fit on the training set $\{(\mathbf{x}^\mu, y^\mu)\}_{\mu=1,\ldots,P}$. In (3.40), we find a second new object: $\mathbf{x}_\perp$. The case considered here is the one where $\alpha_0 = P/N_0 < 1$, meaning that there are more neurons in the input layers than there are training vectors. One common thread throughout HZ's analysis is the decomposition of vectors into two components, one parallel and one perpendicular to the subspace spanned by linear combinations of the training inputs, $\{\mathbf{x}^\mu\}_{\mu=1,\ldots,P}$. $\mathbf{x}_\perp$ denotes the component of the unseen input $\mathbf{x}$ perpendicular to this subspace.

The procedure employed by HZ to arrive at their formula for the partition function is rather complicated. They begin in much the same way LS did: with a Hubbard-Stratonovich transformation, followed by direct integration of the last layer of weights. After this point, however, the two procedures are entirely different. HZ write one of the terms in the integral as a Laplace transform, which they are then able to write as a Meijer G-function using Z-VP's result. What follows are a series of clever manipulations of the integral involving various properties of the Meijer G-functions. Their procedure allows HZ to integrate over the weights in the network all at once, however many layers there are, without the need for an iterative procedure.

Extracting the statistical moments of $f(\mathbf{x})$ is done in the same way as LS did in Eq. (3.34), this time using the exact partition function (3.40). The result is

$$\langle f(\mathbf{x})\rangle = \theta_i^* x_i \,, \tag{3.42a}$$

$$\mathrm{Var}[f(\mathbf{x})] = \frac{1}{2}\left[\prod_{\ell=0}^{L} \frac{2\sigma^2}{N_\ell}\right] \|\mathbf{x}_\perp\|^2 \, \frac{G[1]}{G[0]} \,, \tag{3.42b}$$

where we have used the shorthand notation

$$G[k] \equiv G_{0,L+1}^{L+1,0}\left(\left[\prod_{\ell=0}^{L} \frac{N_\ell}{2\sigma^2}\right] \|\boldsymbol{\theta}^*\|^2 \,\middle|\, \begin{matrix} - \\ \frac{P}{2}, \overbrace{\frac{N_1}{2}+k, \ldots, \frac{N_L}{2}+k}^{} \end{matrix} \right) .$$

HZ do not stop here. They go on to use the Laplace method to produce novel asymptotic expansions of the Meijer G-functions in three distinct cases as P, N_ℓ, and L are taken to infinity in different ways. The one we

care about here is the limit considered by LS in their paper: set $N_1 = \cdots = N_L = N$, fix $P/N_0 = \alpha_0 < 1$, then take the limit $N, P \to \infty$ while $P/N \to \alpha$. In this limit, HZ find

$$\log G[k] = \frac{N\alpha}{2}\left[\log\left(\frac{N\alpha}{2}\right) + \log\left(1+\frac{z_*}{\alpha}\right) - \left(1+\frac{z_*}{\alpha}\right)\right] + \\ + \frac{NL}{2}\left[\log\left(\frac{N}{2}\right) + \log(1+z_*) - (1+z_*)\right] + \mathcal{O}(\log N)\,, \quad (3.43a)$$

and

$$\log G[k] - \log G[0] = kL\left[\log\left(\frac{N}{2}\right) + \log(1+z_*)\right] + \mathcal{O}\left(\frac{\log N}{N}\right)\,, \quad (3.43b)$$

where z_* is the solution to the saddle-point equation

$$\left(1+\frac{z_*}{\alpha}\right)(1+z_*)^L = \frac{\|\boldsymbol{\theta}^*\|^2}{\sigma^{2(L+1)}\alpha_0}\,, \quad z_* > \min\{-\alpha, -1\}\,. \quad (3.43c)$$

3.2.3 Recovering the BPKR result

While comparing the partition functions directly seems to be a non-trivial task, we can check that the mean and variance of $f(\mathbf{x})$ found by LS and HZ match in the wide network limit.

Saddle-point equation

Our first clue that this might be the case is that both LS's and HZ's results rely on a parameter defined implicitly through similar-looking saddle-point equations: (3.32) and (3.43c). We reproduce the two equations here, to compare them.

$$1 - \sigma^{-2}u_0 = \alpha(1 - u_0^{-L}r_0)\,, \quad \text{(LS)}$$

$$\left(1+\frac{z_*}{\alpha}\right)(1+z_*)^L = \frac{\|\boldsymbol{\theta}^*\|^2}{\sigma^{2(L+1)}\alpha_0}\,. \quad \text{(HZ)}$$

In both equations, α, α_0, L, and σ^2 are parameters of the network, while u_0 and z_* are the parameters we want to fix through their respective equations. The remaining two parameters, r_0 and $\|\boldsymbol{\theta}^*\|^2$, are defined by Eq. (3.26) and Eq. (3.41), respectively. LS show that the mean squared readout r_0 is proportional to the squared norm of the vector of weights that would

read off the target labels directly from the input layer, but this is precisely the definition of $\|\boldsymbol{\theta}^*\|^2$ given by HZ. In fact, we can state that

$$\alpha_0\,\sigma^2\,r_0 = \|\boldsymbol{\theta}^*\|^2\,, \tag{3.44}$$

having accounted for the different definitions in the two papers. Having identified these two parameters, the two saddle-point equations are completely equivalent, with

$$u_0 = \sigma^2(1+z_*)\,. \tag{3.45}$$

Output's mean

Next, we can compare the two results for the mean of the output, $\langle f(\mathbf{x})\rangle$, given in Eq. (3.35) and Eq. (3.42a), which we reproduce here for comparison.

$$\langle f(\mathbf{x})\rangle = \frac{\sigma^2}{N_0}\,x_i\,x_i^\mu\,(K_0^{-1})^{\mu\nu}y^\nu\,, \qquad \text{(LS)}$$

$$\langle f(\mathbf{x})\rangle = \theta_i^*\,x_i\,. \qquad \text{(HZ)}$$

In both equations, we are taking a dot product between the new input $\mathbf{x}$ and another vector: $\boldsymbol{\theta}^*$ in the HZ result, and $\frac{\sigma^2}{N_0}\mathbf{x}^\mu(K_0^{-1})^{\mu\nu}y^\nu$ in the LS result. We have to compare these vectors. Once again, we need to look at the definition of $\boldsymbol{\theta}^*$, given in Eq. (3.41). Since there are fewer training vectors than neurons in the input layer ($\alpha_0 < 1$), it should be possible to find a vector $\boldsymbol{\theta}^*$ that solves

$$\sum_{\mu=1}^{P}(\theta_i^* x_i^\mu - y^\mu)^2 = 0\,.$$

As it turns out, $\frac{\sigma^2}{N_0}\mathbf{x}^\mu(K_0^{-1})^{\mu\nu}y^\nu$ is such a vector. We can easily show that this is the case by remembering the definition of the kernel matrix, Eq. (3.11):

$$K_0^{\mu\nu} = \frac{\sigma^2}{N_0}x_i^\mu x_i^\nu\,.$$

Using this definition, we have

$$\left(\frac{\sigma^2}{N_0}x_i^\rho(K_0^{-1})^{\rho\tau}y^\tau\right)x_i^\mu = (x_i^\mu x_i^\rho)(x_j^\rho x_j^\tau)^{-1}y^\tau = \delta^{\mu\tau}y^\tau = y^\mu\,. \tag{3.47}$$

The two results for $\langle f(\mathbf{x})\rangle$ are not strictly equivalent, since the vector $\boldsymbol{\theta}^*$ may not be unique. Nevertheless, Eq. (3.47) shows that the LS's and HZ's procedures lead to extremely similar results.

Output's variance

We conclude this section by comparing the two results for the network output's variance $\mathrm{Var}[f(\mathbf{x})]$, given in Eq. (3.36) and Eq. (3.42b), which we once again reproduce here for comparison.

$$\mathrm{Var}[f(\mathbf{x})] = u_0^L \left[\frac{\sigma^2}{N_0} x_i x_i - \frac{\sigma^4}{N_0^2}\, x_i\, x_i^\mu (K_0^{-1})^{\mu\nu} x_j^\nu\, x_j \right], \qquad \text{(LS)}$$

$$\mathrm{Var}[f(\mathbf{x})] = \frac{\sigma^2}{N_0} \left(\frac{2\sigma^2}{N} \right)^L \|\mathbf{x}_\perp\|^2 \, \frac{G[1]}{G[0]} \,. \qquad \text{(HZ)}$$

We can use Eq. (3.43b) and Eq. (3.45) to write, in the wide network limit $N, P \to \infty$,

$$\frac{G[1]}{G[0]} \simeq \left(\frac{N u_0}{2\sigma^2} \right)^L, \tag{3.49}$$

which we can use to write HZ's result for the variance as

$$\mathrm{Var}[f(\mathbf{x})] \simeq u_0^L \frac{\sigma^2}{N_0} \|\mathbf{x}_\perp\|^2 \,. \tag{3.50}$$

Here, we have used both HZ's asymptotic expansion of the Meijer G-functions and the equivalence between the two saddle-point equations we found earlier.

The last step is to once again recall the definition of the kernel matrix, Eq. (3.11), and notice that

$$\frac{\sigma^2}{N_0} x_i^\mu (K_0^{-1})^{\mu\nu} x_j^\nu = x_i^\mu (x_k^\mu x_k^\nu)^{-1} x_j^\nu \equiv P_{ij} \tag{3.51}$$

is a projection operator, and so

$$\frac{\sigma^2}{N_0} x_i x_i^\mu (K_0^{-1})^{\mu\nu} x_j^\nu x_j = x_i P_{ij} x_j = \|\mathbf{x}_\parallel\|^2 \tag{3.52}$$

is simply the squared norm of the projection of input $\mathbf{x}$ onto the subspace spanned by the set of training vectors $\{\mathbf{x}^\mu\}_{\mu=1,\ldots,P}$. Substituting this into LS's formula, we have

$$\mathrm{Var}[f(\mathbf{x})] = u_0^L \frac{\sigma^2}{N_0} \left[\|\mathbf{x}\|^2 - \|\mathbf{x}_\parallel\|^2 \right] = u_0^L \frac{\sigma^2}{N_0} \|\mathbf{x}_\perp\|^2 \,. \tag{3.53}$$

The two results for the variance are therefore completely equivalent to one another, in the wide network limit.

3.2.4 Comments and conclusions

It should be noted that both procedures we considered avoid dealing explicitly with training through gradient descent-based methods. Instead, both LS's statistical mechanical approach and HZ's Bayesian approach assume that an ensemble of networks will follow a theoretically motivated distribution after training. The applicability of their results to real-world neural networks hinges on this assumption being at least approximately correct.

The two methods both have their strengths and drawbacks. The BPKR has intriguing similarities with well-studied renormalization procedures, common in physics, and the implications of these similarities may be worth investigating further. Moreover, it enables us to integrate over some of the network's weights and find an effective description of the resulting truncated network, which has the potential to provide insights into the role of depth in DLNNs.

On the other hand, there is no doubt that the biggest strength of HZ's approach is its ability to provide an exact solution for networks of any width or depth. The BPKR, in contrast, relies on the wide network limit, $N \to \infty$, which may restrict its applicability to some real-world networks.

Another useful feature of LS's method is that it can be generalized to networks with multiple outputs, whereas our attempts at generalizing HZ's method to output dimensions greater than one have shown this to be a non-trivial task.

One more aspect to consider is how easily interpretable each method's results are. HZ's exact result is undoubtedly remarkable, but its form in terms of Meijer G-functions (and an infinite sum of them at that) is rather obscure and difficult to interpret. Gaining intuition about the inner workings of deep neural networks is a task worth pursuing, and this is not an especially straightforward task when Meijer G-functions are involved, though asymptotic expansions help. On a related note, very interesting results and considerations about the combined roles of depth and width in neural networks can be found in HZ's paper.

Finally, both approaches seem to rely heavily on the activation functions being linear. Unfortunately, it is difficult to see a clear way to extend either of the two procedures to more realistic, non-linear networks. One promising result in this direction can be found in the Z-VP paper [14], where the authors find an exact equivalent of equation (3.38) for the case of ReLU activation functions. Unfortunately, building on this result in a similar way as HZ did for the linear case does not seem to be quite as straightforward.

3.3 The Gaussian Limit

So far, we have used statistical mechanics to describe the behavior of an ensemble of trained DLNNs. We have shown how to compute the partition function by iteratively integrating over the layers of weights (BPKR procedure). Then, we have described how a different approach to the same problem, based on Bayesian inference, could be used to recover the same results, renewing our confidence in the validity of the BPKR calculation.

There is another question we could ask: does the BPRK lead to any non-trivial results? In Section 3.2, we have shown how the BPKR recovers the correct leading order results in the wide network limit, $N \to \infty$, but is the leading order enough to capture any non-trivial behavior? In this section, we will show how a simplified version of the BPKR can be used to recover what we will call the "Gaussian limit" of DLNNs. The fact that the full BPKR procedure manages to go beyond this simple limit indicates that it is indeed capturing at least some non-trivial features of DLNNs.

3.3.1 The Gaussian limit as a saddle-point approximation

Let us consider a linear neural network with L hidden layers of $N_1, \ldots, N_L$ neurons each, an input layer of N_0 neurons, and N_{L+1} outputs. Denote the Gaussian probability distribution for the weights in layer ℓ prior to training as

$$p(W_{ij}^{\ell}) = \sqrt{\frac{N_{\ell-1}}{2\pi\sigma_{\ell}^2}}\, \exp\left\{ -\frac{N_{\ell-1}}{2\sigma_{\ell}^2}\, (W_{ij}^{\ell})^2 \right\} . \tag{3.54}$$

The combined prior probability distribution of all of the weights in the network is

$$p(\mathbf{W}) = \prod_{\ell=1}^{L+1} \prod_{i=1}^{N_\ell} \prod_{j=1}^{N_{\ell-1}} p(W_{ij}^{\ell}) . \tag{3.55}$$

Let $\mathrm{d}\mathbf{W}$ be the plain integration measure over all of the network's weights:

$$\mathrm{d}\mathbf{W} = \prod_{\ell=1}^{L+1} \prod_{i=1}^{N_\ell} \prod_{j=1}^{N_{\ell-1}} \mathrm{d}W_{ij}^{\ell} . \tag{3.56}$$

We can define the expectation value of an arbitrary function of the weights $\Phi(\mathbf{W})$ with respect to the probability distribution $p(\mathbf{W})$ as

$$\langle \Phi(\mathbf{W}) \rangle = \int \mathrm{d}\mathbf{W}\, p(\mathbf{W})\, \Phi(\mathbf{W}) . \tag{3.57}$$

In our analogy with statistical mechanics, the trained networks are described by a Hamiltonian. In our case, the quadratic loss function plays the role of the Hamiltonian, and it is a function of the weights, the training inputs, and the target labels,

$$H(\mathbf{W}; \{\mathbf{x}^\mu, \mathbf{y}^\mu\}_{\mu=1,\dots,P}) = \frac{1}{2}\sum_{\mu=1}^{P} \left[f(\mathbf{W}, \mathbf{x}^\mu) - \mathbf{y}^\mu\right]^2 , \tag{3.58}$$

where $f(\mathbf{W}, \mathbf{x}^\mu) \equiv f(\mathbf{x}^\mu)$ is once again the output of the network when vector $\mathbf{x}^\mu$ is used as the input:

$$f(\mathbf{x}) = W^{L+1}_{i\,i_L} W^{L}_{i_L i_{L-1}} \cdots W^{1}_{i_1 j}\, x^\mu_j \, . \tag{3.59}$$

Notice that, since we are considering an output layer with N_{L+1} neurons, the target labels have been promoted to vectors: $\mathbf{y}^\mu = (y^\mu_1, \dots, y^\mu_{L+1})$.

The partition function can be written as the expectation value of the Boltzmann factor,

$$Z = \left\langle \mathrm{e}^{-\beta H} \right\rangle \, . \tag{3.60}$$

This is analogous to Eq. (3.39), used by HZ. In the limit $\beta \to \infty$ (low-temperature limit), we can use a saddle-point approximation on integral (3.57) to find

$$Z = \left\langle \mathrm{e}^{-\beta H} \right\rangle \xrightarrow{\beta\to\infty} \mathrm{e}^{-\beta \langle H \rangle} \, . \tag{3.61}$$

Computing the expectation value of the Hamiltonian we find

$$\begin{aligned} \langle H \rangle = \frac{1}{2}\sum_{\mu=1}^{P} \Big[\langle [f(\mathbf{x}^\mu)]^2 \rangle - \langle f(\mathbf{x}^\mu) \rangle\, \mathbf{y}^\mu + (\mathbf{y}^\mu)^2 \Big] = \\ = \frac{1}{2}\sum_{\mu=1}^{P} \Big[\langle [f(\mathbf{x}^\mu)]^2 \rangle + (\mathbf{y}^\mu)^2 \Big] \, , \end{aligned} \tag{3.62}$$

where we have used the fact that $f(\mathbf{x}^\mu)$ is linear in the weights. Meanwhile, $[f(\mathbf{x}^\mu)]^2$ is quadratic in the weights and its expectation value is a product of Gaussian integrals. We can compute this explicitly. For this

computation, no summation over Greek indices is implied.

$$\begin{aligned}
\langle [f(\mathbf{x}^\mu)]^2 \rangle &= \left\langle \delta_{i_{L+1}j_{L+1}} \delta^{\mu\nu} \left(W^{L+1}_{i_{L+1}i_L} \dots W^1_{i_1 i_0} x^\mu_{i_0}\right)\left(W^{L+1}_{j_{L+1}j_L} \dots W^1_{j_1 j_0} x^\nu_{j_0}\right) \right\rangle \\
&= \delta_{i_{L+1}j_{L+1}} \left\langle W^{L+1}_{i_{L+1}i_L} W^{L+1}_{j_{L+1}j_L} \right\rangle \dots \left\langle W^1_{i_1 i_0} W^1_{j_1 j_0} \right\rangle x^\mu_{i_0} x^\mu_{j_0} \\
&= \delta_{i_{L+1}j_{L+1}} \left(\frac{\sigma^2_{L+1}}{N_L} \delta_{i_{L+1}j_{L+1}} \delta_{i_L j_L}\right) \dots \left(\frac{\sigma^2_1}{N_0} \delta_{i_1 j_1} \delta_{i_0 j_0}\right) x^\mu_{i_0} x^\mu_{j_0} \\
&= \frac{\sigma^2_{L+1} \dots \sigma^2_1}{N_L \dots N_0} \underbrace{\delta_{i_{L+1}j_{L+1}} \delta_{i_{L+1}j_{L+1}}}_{=N_{L+1}} \underbrace{\delta_{i_L j_L} \delta_{i_L j_L}}_{=N_L} \dots \underbrace{\delta_{i_1 j_1} \delta_{i_1 j_1}}_{=N_1} x^\mu_{i_0} x^\mu_{i_0} \\
&= \frac{N_{L+1}}{N_0} \sigma^2_{L+1} \cdots \sigma^2_1 \left\|\mathbf{x}^\mu\right\|^2 , \qquad (3.63)
\end{aligned}$$

where δ_{ij} denotes a Kronecker delta and $\left\|\mathbf{x}^\mu\right\|^2 = \sum_{i=1}^{N_0} (x^\mu_i)^2$. In the computation, we have used the fact that the probability distributions of weights in different layers are independent of each other, and therefore the full expectation value can be written as a product of expectation values of weights in different layers.

Finally, going back to Eq. (3.62) we find

$$\langle H \rangle = \frac{1}{2} \sum_{\mu=1}^{P} \left[\frac{N_{L+1}}{N_0} \sigma^2_{L+1} \dots \sigma^2_1 \left\|\mathbf{x}^\mu\right\|^2 + (\mathbf{y}^\mu)^2 \right]. \qquad (3.64)$$

We found that in the low temperature limit, the partition function can be approximated as $Z = e^{-\beta \langle H \rangle}$, which is purely Gaussian since the expectation value of the Hamiltonian is given by (3.64). From now on, we will refer to this result as the "Gaussian limit."

3.3.2 The Gaussian limit through iterative integration

In Section 3.1, we explained the procedure used by LS to compute the network's Hamiltonian (the back-propagating kernel renormalization) by proceeding upstream through the network, iteratively integrating one layer of weights at a time and finding a sequence of effective Hamiltonians. Here, we will show how we can use a simplified version of the same procedure to recover the Gaussian limit.

First, recall that LS considered the case in which $N_1 = \dots = N_L = N$, $N_{L+1} = 1$, and $\sigma^2_1 = \dots = \sigma^2_{L+1} = \sigma^2$. In order to carry out the calculations, in Section 3.1 we defined a convenient set of Gaussian integration

measures for the weights, given by Eq. (3.5):

$$\mathcal{D}W^1 = \left(\prod_{i=1}^{N}\prod_{j=1}^{N_0} \mathrm{d}W_{ij}^1\right)\left(\frac{N_0}{2\pi\sigma^2}\right)^{\frac{NN_0}{2}} \exp\left\{-\frac{N_0}{2\sigma^2}W_{ij}^1 W_{ij}^1\right\},$$

$$\mathcal{D}W^\ell = \left(\prod_{i=1}^{N}\prod_{j=1}^{N} \mathrm{d}W_{ij}^\ell\right)\left(\frac{N}{2\pi\sigma^2}\right)^{\frac{N^2}{2}} \exp\left\{-\frac{N}{2\sigma^2}W_{ij}^\ell W_{ij}^\ell\right\},$$

for $\ell = 2, \ldots, L$, and

$$\mathcal{D}W^{L+1} = \left(\prod_{i=1}^{N} \mathrm{d}W_{i}^{L+1}\right)\left(\frac{N}{2\pi\sigma^2}\right)^{\frac{N}{2}} \exp\left\{-\frac{N}{2\sigma^2}W_{i}^{L+1} W_{i}^{L+1}\right\}.$$

With this notation, we defined the effective Hamiltonians H_ℓ as

$$Z = \int \mathcal{D}W^1 \cdots \int \mathcal{D}W^\ell \, Z_\ell = \int \mathcal{D}W^1 \cdots \int \mathcal{D}W^\ell \exp\{-\beta H_\ell\}.$$

The first step in LS's procedure was to perform a Hubbard-Stratonovich transformation, which was followed by direct integration of the last layer of weights, the result was Eq. (3.12):

$$Z_L = \int \mathcal{D}t \exp\left\{it^\mu y^\mu - \frac{1}{2}t^\mu K_L^{\mu\nu} t^\nu\right\}, \tag{I}$$

where

$$\mathcal{D}t = \left(\prod_{\mu=1}^{P} \mathrm{d}t^\mu\right)\left(\frac{1}{2\pi\beta}\right)^{\frac{P}{2}} \exp\left\{-\frac{1}{2\beta}t^\mu t^\mu\right\}$$

and

$$K_\ell^{\mu\nu} = \frac{\sigma^2}{N} x_i^{\ell,\mu} x_i^{\ell,\nu}.$$

Integrating over the next layer of weights led us to Eq. (3.20):

$$Z_{L-1} = \int \mathcal{D}t \; \exp\left\{it^\mu y^\mu - \frac{N}{2}\log\left(1 + \frac{\sigma^2}{N}t^\mu K_{L-1}^{\mu\nu} t^\nu\right)\right\}. \tag{II}$$

At this point in the procedure, we commented that the simplest thing to do would be approximating

$$\frac{N}{2}\log\left(1 + \frac{\sigma^2}{N}t^\mu K_{L-1}^{\mu\nu} t^\nu\right) \underset{N\to\infty}{\simeq} \frac{\sigma^2}{2}t^\mu K_{L-1}^{\mu\nu} t^\nu, \tag{III}$$

as this would bring Z_{L-1} into the same form as Z_L, but we opted against it, the reason being that this approximation would have been too drastic. As it turns out, this is exactly the approximation we have to use to recover the Gaussian limit. Each successive iteration of the simplified procedure starts with an integral similar to (I), then, integrating over the following layer of weights, we find an integral in the form of (II), at which point we can approximate our result using (III), and repeat for the following layer. Each iteration will add a factor of σ^2 in front of the kernel K_ℓ, and at the end we would be left with

$$\begin{aligned} Z &\approx \int \mathcal{D}t \exp\left\{ it^\mu y^\mu - \frac{\sigma^{2L}}{2} t^\mu K_0^{\mu\nu} t^\nu \right\} = \\ &= \int \left(\prod_{\mu=1}^{P} \mathrm{d}t^\mu \right) \left(\frac{1}{2\pi\beta} \right)^{\frac{P}{2}} \exp\left\{ -\frac{1}{2\beta} t^\mu \Sigma_0^{\mu\nu} t^\mu + i t^\mu y^\mu \right\} = \exp\{-\beta H\} \,, \end{aligned} \tag{3.65}$$

where

$$\beta H = \frac{1}{2} y^\mu (\Sigma_0^{-1})^{\mu\nu} y^\nu + \frac{1}{2\beta} \log\det(\Sigma_0) \tag{3.66}$$

and

$$\Sigma_0^{\mu\nu} = \delta^{\mu\nu} + \beta\, \sigma^{2L} K_0^{\mu\nu} = \delta^{\mu\nu} + \frac{\beta\, \sigma^{2(L+1)}}{N_0} x_i^\mu x_i^\nu \,.$$

If we now assume $\beta / N_0 \ll 1$, we can use

$$\Sigma_0^{\mu\nu} \approx \delta^{\mu\nu}$$

and

$$\log\det(\Sigma_0) = \mathrm{Tr}\log(\Sigma_0) = \sum_{\mu=1}^{P} \left[\log(\Sigma_0) \right]^{\mu\mu} \approx \sum_{\mu=1}^{P} \frac{\beta\, \sigma^{2(L+1)}}{N_0} \|\mathbf{x}^\mu\|^2$$

to recover our previous Gaussian limit result, Eq. (3.64):

$$H_L \approx \frac{1}{2} \sum_{\mu=1}^{P} \left[\frac{\sigma^{2(L+1)}}{N_0} \|\mathbf{x}^\mu\|^2 + (y^\mu)^2 \right] , \tag{3.67}$$

where the norm is taken over the Latin index, i.e. $\|\mathbf{x}^\mu\|^2 = \sum_{i=1}^{N_0} (x_i^\mu)^2$.

Notice that we previously arrived at Eq. (3.64) without having to require $N_\ell \to \infty$; the limit $\beta \to \infty$ was sufficient. When explicitly integrating over the weights following LS's procedure, however, we had to require the number of neurons per layer to be large. Not only that, but we had to require $\beta / N_0 \ll 1$, meaning that the input dimension N_0 has to go to infinity faster than β. The relationship between these various limits seems to be rather complex and it may be worth studying in more detail.

Chapter 4

Fisher Information in Neural Networks

Fisher information can be a powerful tool in the study of statistical models. Many models used in science rely on a large number of free parameters, but Fisher information-based approaches have shown that some parameter combinations have a vastly larger effect on the model's output than others [9, 12]. By their nature, neural network models usually have an extremely large number of free parameters that have to be tuned during training. Fisher information can provide a way to identify the most important parameter combinations in a trained network, which could hopefully lead to a prescription for reducing the number of free parameters. This could lead to a variety of benefits, from avoiding overfitting to helping understand the inner workings of deep neural network models.

4.1 Fisher Information and Sloppy Models

Information geometry is the study of statistical models using techniques from differential geometry [1]. In addition to contributing an interesting point of view on many concepts in statistics, such as distributions, samples, and inference, it can provide useful insights into the inner workings of statistical models. In particular, information geometry has been used to investigate the relative importance of different parameters in scientific models. [9, 11, 12]. The central observation of these papers, that a relatively small number of parameters often accounts for most of a model's behavior, is a rather powerful one: it explains why simple models can sometimes provide accurate descriptions of highly complex phenomena.

4.1.1 Statistical manifolds

Consider a Gaussian distribution,

$$p(x|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}}\ \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right].$$

Repeated measurements of the random variable x described by this distribution will have mean μ and will be spread out according to the variance σ^2. Now, suppose that we have good reasons to believe that our measurements of some process will return a Gaussian distribution, but we cannot predict what the mean and variance will be, so we decide to infer these parameters from measurements. One question arises: how confidently can we tell a distribution with parameters (μ_1,σ_1^2) from one with parameters (μ_2,σ_2^2)? To answer, let us consider a more general setting. Let $\mathbf{x} = (x_1, x_2, \ldots, x_m)$ be a vector of random variables, and let $p(\mathbf{x}|\boldsymbol{\theta})$ be a family of probability distributions that depend on the parameters $\boldsymbol{\theta} = (\theta_1, \theta_2, \ldots, \theta_n)$. Can we define a notion of how different a distribution $p(\mathbf{x}|\boldsymbol{\theta}_1)$ of parameters $\boldsymbol{\theta}_1$ is from a second distribution $p(\mathbf{x}|\boldsymbol{\theta}_2)$ of parameters $\boldsymbol{\theta}_2$? A notion of similarity between two probability distributions is known as a statistical distance, and one commonly used distance is the Kullback-Leibler (KL) divergence, defined as

$$D_{KL}[p(\mathbf{x}|\boldsymbol{\theta}_2)\ ||\ p(\mathbf{x}|\boldsymbol{\theta}_1)] = \int \mathrm{d}^m\mathbf{x}\ p(\mathbf{x}|\boldsymbol{\theta}_2)\ \log\left(\frac{p(\mathbf{x}|\boldsymbol{\theta}_2)}{p(\mathbf{x}|\boldsymbol{\theta}_1)}\right), \tag{4.1}$$

This notion of distance lacks many characteristics that we might find useful: it is not symmetric in its arguments and it does not satisfy the triangle inequality. In order to fix these problems, we notice that the KL divergence is minimized by $\boldsymbol{\theta}_1 = \boldsymbol{\theta}_2 \equiv \boldsymbol{\theta}$, and we expand around this minimum, finding

$$D_{KL}[p(\mathbf{x}|\boldsymbol{\theta}+d\boldsymbol{\theta})\ ||\ p(\mathbf{x}|\boldsymbol{\theta})] \simeq \frac{1}{2}\, g_{\alpha\beta}(\boldsymbol{\theta})\, \mathrm{d}\theta_\alpha\, \mathrm{d}\theta_\beta + \mathcal{O}(\mathrm{d}\boldsymbol{\theta}^3),$$

where we have defined

$$g_{\alpha\beta}(\boldsymbol{\theta}) = \left.\frac{\partial^2 D_{KL}}{\partial\theta_\alpha\partial\theta_\beta}\right|_{d\boldsymbol{\theta}=0} = \int \mathrm{d}^m\mathbf{x}\ p(\mathbf{x}|\boldsymbol{\theta})\ \frac{\partial \log p(\mathbf{x}|\boldsymbol{\theta})}{\partial\theta_\alpha}\ \frac{\partial \log p(\mathbf{x}|\boldsymbol{\theta})}{\partial\theta_\beta}. \tag{4.2a}$$

This is known as the Fisher information metric (FIM) and can be written more concisely in terms of $\ell = \log p(\mathbf{x}|\boldsymbol{\theta})$ as

$$g_{ab}(\boldsymbol{\theta}) = \left\langle \partial_\alpha\ell\ \partial_\beta\ell \right\rangle = -\left\langle \partial_\alpha\partial_\beta\ell \right\rangle, \tag{4.2b}$$

where $\partial_\alpha \equiv \partial/\partial\theta_\alpha$ and $\langle\cdot\rangle$ denotes the expectation value with respect to $p(\mathbf{x}|\boldsymbol{\theta})$. The two formulas in terms of first or second derivatives are equivalent because $\langle\partial_\alpha \ell\rangle = 0$.

As the name suggests, the FIM is a metric. To understand how this can be, we need to look at our probability distributions in a different way. Given a family of probability distributions $p(\mathbf{x}|\boldsymbol{\theta})$, we can define a Riemannian manifold on which the parameters $(\theta_1, \theta_2, \ldots, \theta_n)$ provide a set of coordinates. This is known as a statistical manifold, and each point corresponds to a probability distribution. The FIM can be used as a Riemannian metric on this statistical manifold, providing a natural notion of distance between probability distributions. The field of information geometry [1] is concerned with studying the nature of these manifolds, using the tools of differential geometry to gain insight into the nature of statistical models and techniques.

One result in particular, the Cramér-Rao bound, provides some useful intuition about the FIM itself. The Cramér-Rao bound is a statement about estimators. An estimator is a procedure to extract an estimate for one of the parameters of the probability distribution from measurements. Going back to our original example of a Gaussian distribution, if we wanted to estimate the mean μ given a set $\{x_i\}_{i=1,\ldots,N}$ of measurements, we could use the empirical mean $\widehat{\mu} = \frac{1}{N}\sum_{i=1}^{N} x_i$ as an estimator of the true mean μ. Moreover, the empirical mean is an example of what is called an "unbiased" estimator since

$$\langle\widehat{\mu}\rangle = \frac{1}{N}\sum_{i=1}^{N}\int \mathrm{d}x_i\; x_i\; p(x_i|\mu,\sigma^2) = \mu\;.$$

Applying the same logic to the more general case of a distribution $p(\mathbf{x}|\boldsymbol{\theta})$, we can define the bias B of an estimator $\widehat{\theta}_a$ of the parameter θ_a as $B(\widehat{\theta}_a) = \langle\widehat{\theta}_a\rangle - \theta_a$. The Cramér-Rao bound states that, given two unbiased estimators $\widehat{\theta}_\alpha$ and $\widehat{\theta}_\beta$, and a set $\{\mathbf{x}_i\}_{i=1,\ldots,N}$ of independent measurements,

$$\left\langle \left(\widehat{\theta}_\alpha - \theta_\alpha\right)\left(\widehat{\theta}_\beta - \theta_\beta\right)\right\rangle \geq \frac{1}{N}\, g^{\alpha\beta}(\boldsymbol{\theta})\;, \tag{4.3}$$

where $g^{\alpha\beta}(\boldsymbol{\theta}) = [g_{\alpha\beta}(\boldsymbol{\theta})]^{-1}$ is the inverse FIM. If we wanted to determine the values of parameters θ_α and θ_β from measurements up to a given accuracy, a larger value of $g^{\alpha\beta}(\boldsymbol{\theta})$ would mean that we need a larger sample. Given a finite sample, some combinations of parameters will be better estimated than others. Conversely, this means that some combinations of parameters have a larger, more noticeable effect on the outcome of measurements than others, and this information is contained in the FIM.

4.1.2 Sloppy models

In recent years, a series of papers have brought to light how the FIM can provide insight into the models we use to describe vastly different phenomena, in diverse areas of science [9, 12]. The central notion in these papers is that of "sloppiness" in statistical models. The authors notice that the FIM in many models of natural phenomena is characterized by an eigenvalue spectrum spanning many orders of magnitude. Some directions in parameters space, corresponding to large eigenvalues, are extremely important in describing the data and are labeled "stiff", while others are labeled "sloppy" since even large changes to the parameters in these directions have little effect on the model's predictions. Often, the eigenvalues seem to roughly follow a geometric sequence (Fig. 4.1), so a very limited number of parameter combinations account for much of the observations. The model manifold is characterized by a "hyper-ribbon" structure: just as a ribbon is much longer than it is wide and much wider than it is thick, each subsequent direction in the model manifold is much shorter than the previous one (length is defined using the FIM).

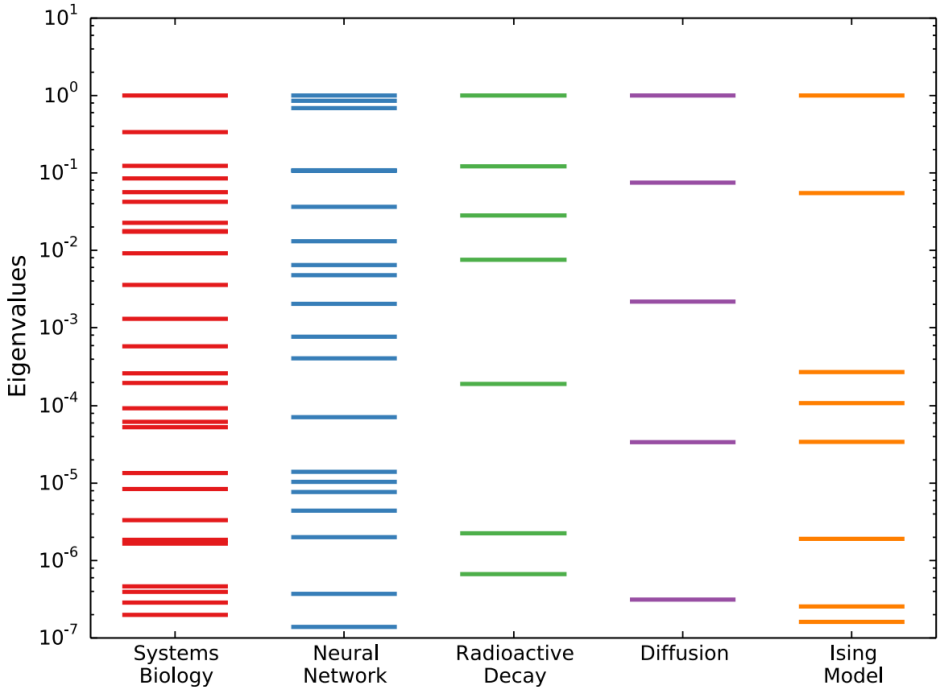

Figure 4.1: *Largest (normalized) eigenvalues of the FIM for models in various fields. Source: Transtrum et al., 2015 [12].*

4.2 Analytical Results

Are neural networks sloppy? If so, can this be useful for understanding their properties and behavior? Neural networks usually have a very large number of parameters, and it is reasonable to assume that not all parameters are equally important for the network's performance. A preliminary investigation of simple neural networks can be found in a paper by Transtrum et al. [13], and it seems to suggest that they can be characterized as sloppy models. A more extensive exploration of Fisher information in neural networks was carried out by Karakida et al. [5, 6]. In this section, we will investigate the FIM associated with simple neural network models, with a particular focus on its eigenvalue spectrum.

4.2.1 A classification problem

We consider a classification problem, where an input x sampled from a set $\mathcal{X}$ has to be correctly classified as belonging to a class y chosen from set $\mathcal{Y}$. For example, $\mathcal{X}$ could be a set of images representing handwritten digits and $\mathcal{Y} = \{0, 1, 2, \dots, 9\}$. To perform this classification, we consider a model $f_{\boldsymbol{\theta}}(x)$, where $\boldsymbol{\theta} = (\theta_1, \theta_2, \dots, \theta_n)$ are the parameters. Our model takes $x \in \mathcal{X}$ as its input and determines the probability $p(y|x, \boldsymbol{\theta})$ that x should be classified as belonging to class $y \in \mathcal{Y}$. We use a set of pairs of inputs and target labels $\{(x_i, y_i)\}_{i=1,\dots,P}$ to estimate the set of parameters $\bar{\boldsymbol{\theta}}$ that maximizes the likelihood that our model would correctly classify each x_i as belonging to class y_i. In order to do this we need a procedure to quantify the model's performance with a given set of parameters and a mechanism for training, i.e. updating the values of the parameters to improve performance.

We choose to treat both the input and output as vectors. As input we will use N_0-dimensional vectors $\mathbf{x} \in \mathbb{R}^{N_0}$. The model's output will be a N_L-dimensional vector $\mathbf{y} \in \mathbb{R}^{N_L}$, where N_L is the number of classes. Different classes are represented by so-called "one-hot" vectors, so $\mathbf{y}^{(1)} \equiv (1, 0, 0 \dots)$ is the first class, $\mathbf{y}^{(2)} \equiv (0, 1, 0, \dots)$ the second, etc., and our model is a function $f_{\boldsymbol{\theta}} : \mathbb{R}^{N_0} \to \mathbb{R}^{N_L}$.

Given an input $\mathbf{x}$ and its true label $\mathbf{y}$, we can define a loss function

$$E(\boldsymbol{\theta}) = \|\mathbf{y} - f_{\boldsymbol{\theta}}(\mathbf{x})\|^2 , \tag{4.4}$$

which is an indicator of the model's performance as a classifier for input $\mathbf{x}$. Such a quadratic loss is common, but not the only choice. It is usually considered appropriate when the process that generates the data leads to

Gaussian uncertainty on the output. In our example, if we denoted as $F : \mathbb{R}^{N_0} \to \mathbb{R}^{N_L}$ the true process generating the data, this would mean that

$$p(\mathbf{y}|\mathbf{x}) \sim \exp\left[-\frac{1}{2\sigma}\|\mathbf{y} - F(\mathbf{x})\|^2\right] . \tag{4.5}$$

Such an assumption is usually considered appropriate for processes with continuous outputs, rather than for classification tasks, where the output is discreet. Nevertheless, using Eq. (4.4) as a loss function can lead to good results for classification tasks, too, and using Eq. (4.5) will allow us to write the FIM in a remarkably simple form. Formulating an FIM analysis of neural networks in a form more appropriate for categorical data is left to future work; in the present thesis, we will concern ourselves with whether useful insights about neural networks can be obtained using Equations (4.4) and (4.5) to compute the FIM.

4.2.2 Calculations

Using (4.5) as our probability distribution, we can calculate the FIM using its definition (4.2)[1]. The resulting FIM is

$$g_{\alpha\beta}(\bar{\boldsymbol{\theta}}) = \sum_{\mu} \frac{1}{\sigma^2}\left[\partial_{\theta_\alpha} f_{\boldsymbol{\theta}}(\mathbf{x})\Big|_{\boldsymbol{\theta}=\bar{\boldsymbol{\theta}}}\right]^{\mu}\left[\partial_{\theta_\beta} f_{\boldsymbol{\theta}}(\mathbf{x})\Big|_{\boldsymbol{\theta}=\bar{\boldsymbol{\theta}}}\right]^{\mu} . \tag{4.6}$$

Summation of repeated indices will *not* be left implicit for the remainder of the present discussion on the Fisher information of artificial neural networks. Notice that (4.6) is a function of the input $\mathbf{x}$. Let us not apply this general formula to some simple neural network architectures.

No hidden layers

Let us first look at a very simple model, without hidden layers. Each component of the input vector $\mathbf{x} = x^\mu \in \mathbb{R}^{N_0}$ is connected by a weight to each component of the output vector $\mathbf{y} = y^\mu \in \mathbb{R}^{N_1}$. A generic non-linearity $\varphi(\cdot)$ is then applied to each component of the output. Denoting as θ^μ_ν the weight connecting the ν-th component of the input to the μ-th component of the output, the model is[2]

$$[f_{\boldsymbol{\theta}}(\mathbf{x})]^\mu = \varphi\Big(\sum_\nu \theta^\mu_\nu x^\nu\Big) = \varphi(a^\mu) ,$$

[1] The expectation value is taken over $\mathbf{y}$, so $\langle\cdot\rangle = \int \mathrm{d}\mathbf{y}\, p(\mathbf{y}|\mathbf{x})\,(\cdot)$.

[2] Upper and lower indices do not have any deep meaning, but this notation is good for telling at a glance the direction of the weights.

Version of June 16, 2023– Created June 16, 2023 - 07:37

where we have introduced $a^\mu = \sum_\nu \theta^\mu_\nu x^\nu$. The corresponding FIM is

$$g_{\theta^\mu_\alpha \theta^\nu_\beta}(\bar{\boldsymbol{\theta}}) = \frac{1}{\sigma^2}\, \delta^{\mu\nu}\, \varphi'(a^\mu)\, \varphi'(a^\nu)\, x^\alpha\, x^\beta = \frac{1}{\sigma^2}\, \left[\varphi'(a^\mu)\right]^2 \delta^{\mu\nu}\, x^\alpha\, x^\beta\,, \tag{4.7}$$

where $\varphi'(z) \equiv \frac{d\varphi}{dz}(z)$ and a^μ is computed using the trained parameters $\bar{\boldsymbol{\theta}}$. The presence of the Kronecker delta allows us to write this FIM in the form of a block-diagonal matrix, where each block is made of the same matrix $\mathbf{B} = [B^{\alpha\beta}] = x^\alpha x^\beta \in \mathbb{R}^{n\times n}$ scaled by a factor $\alpha^{(\mu)} = \sigma^{-2}[\varphi'(a^\mu)]^2 \in \mathbb{R}$, where μ labels the different blocks.

$$g_{\theta^\mu_\alpha \theta^\nu_\beta}(\bar{\boldsymbol{\theta}}) = \begin{pmatrix} \alpha^{(1)}\mathbf{B} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \alpha^{(2)}\mathbf{B} & \cdots & \mathbf{0} \\ \vdots & \vdots & & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \alpha^{(m)}\mathbf{B} \end{pmatrix}.$$

One hidden layer

We introduce one hidden layer with N_1 units to our model. We denote the input vector as $\mathbf{x}_0 = x^\mu_0 \in \mathbb{R}^{N_0}$, and the output vector as $\mathbf{y} = y^\mu \in \mathbb{R}^{N_2}$. $\theta^\mu_{1\nu}$ is the weight connecting the ν-th component of $\mathbf{x}_0$ with the μ-th hidden unit, and $\theta^\mu_{2\nu}$ is the weight connecting the ν-th hidden unit with the μ-th component of $\mathbf{y}$. We also define the vectors $\mathbf{x}_1 = x^\mu_1 \in \mathbb{R}^{N_1}$, denoting the values of hidden layer units, and $\mathbf{a}_1 = a^\mu_1 = \sum_\nu \theta^\mu_{1\nu} x^\nu_0$ and $\mathbf{a}_2 = a^\mu_2 = \sum_\nu \theta^\mu_{2\nu} x^\nu_1$. With these definitions in place, the model can be written as

$$[f_{\boldsymbol{\theta}}(\mathbf{x}_0)]^\mu = \varphi_2(a^\mu_2) = \varphi_1\Big(\sum_\nu \theta^\mu_{2\nu}\varphi_1(a^\nu_1)\Big) = \varphi_1\Big(\sum_\nu \theta^\mu_{2\nu}\varphi_1\Big(\sum_\rho \theta^\nu_{1\rho} x^\rho_0\Big)\Big)\,,$$

where $\varphi_1(z)$ denotes the non-linearity acting of the hidden layer neurons and $\varphi_2(z)$ is a second, possibly different non-linearity.

Three distinct cases arise when calculating the FIM. First, components that refer to two weights in the second layer are a straightforward generalization of the case without hidden layers.

$$g_{\theta^\mu_{2\alpha} \theta^\nu_{2\beta}}(\bar{\boldsymbol{\theta}}) = \frac{1}{\sigma^2}\, \left[\varphi'_2\big(a^\mu_2\big)\right]^2 \delta_{\mu\nu}\, x^\alpha_1\, x^\beta_1\,. \tag{4.8a}$$

Next, FIM components that refer to two weights in the first layer are

$$g_{\theta^\mu_{1\alpha} \theta^\nu_{1\beta}}(\bar{\boldsymbol{\theta}}) = \frac{1}{\sigma^2} \sum_\rho \Big[\left[\varphi'_2\big(a^\rho_2\big)\right]^2 \varphi'_0\big(a^\mu_1\big)\, \varphi'_1(a^\nu_1)\, \bar{\theta}^\rho_{2\mu}\, \bar{\theta}^\rho_{2\nu}\Big]\, x^\alpha_0\, x^\beta_0\,, \tag{4.8b}$$

which is no longer block diagonal, instead having a considerably more complex internal structure. Finally, components that refer to weights in two different layers are

$$g_{\theta^{\mu}_{1\alpha}\theta^{\nu}_{2\beta}}(\bar{\boldsymbol{\theta}}) = \frac{1}{\sigma^2}\left[\varphi_2'(a_2^{\nu})\right]^2 \varphi_1'\left(a_1^{\mu}\right) \bar{\theta}^{\nu}_{2\mu}\, x_0^{\alpha}\, x_1^{\beta}\,, \tag{4.8c}$$

We can write the full FIM concisely by defining some matrices $\mathbf{B}_{ij} = B_{ij}^{\alpha\beta}$ and $\mathbf{A}_{ij} = A_{ij}^{\mu\nu}$, where $i, j \in \{0, 1\}$:

$$\begin{aligned}
B_{11}^{\alpha\beta} &= x_1^{\alpha} x_1^{\beta} \in \mathbb{R}^{N_1 \times N_1}\,, \quad A_{11}^{\mu\nu} = \frac{1}{\sigma^2}\left[\varphi_2'\left(a_2^{\mu}\right)\right]^2 \delta_{\mu\nu} \in \mathbb{R}^{N_2 \times N_2}\,; \\
B_{10}^{\alpha\beta} &= x_1^{\alpha} x_0^{\beta} \in \mathbb{R}^{N_1 \times N_0}\,, \quad A_{10}^{\mu\nu} = \frac{1}{\sigma^2}\left[\varphi_2'\left(a_2^{\mu}\right)\right]^2 \varphi_1'(a_1^{\nu})\, \bar{\theta}^{\mu}_{2\nu} \in \mathbb{R}^{N_2 \times N_1}\,; \\
B_{01}^{\alpha\beta} &= x_0^{\alpha} x_1^{\beta} \in \mathbb{R}^{N_0 \times N_1}\,, \quad A_{01}^{\mu\nu} = \frac{1}{\sigma^2}\left[\varphi_2'(a_2^{\nu})\right]^2 \varphi_1'\left(a_1^{\mu}\right) \bar{\theta}^{\nu}_{2\mu} \in \mathbb{R}^{N_1 \times N_2}\,; \\
B_{00}^{\alpha\beta} &= x_0^{\alpha} x_0^{\beta} \in \mathbb{R}^{N_0 \times N_0}\,, \\
A_{00}^{\mu\nu} &= \frac{1}{\sigma^2} \sum_{\rho} \left[\left[\varphi_2'\left(a_2^{\rho}\right)\right]^2 \varphi_1'\left(a_1^{\mu}\right) \varphi_1'(a_1^{\nu})\, \bar{\theta}^{\rho}_{2\mu}\, \bar{\theta}^{\rho}_{2\nu}\right] \in \mathbb{R}^{N_1 \times N_1}\,.
\end{aligned}$$

The FIM can be written as

$$g(\bar{\boldsymbol{\theta}}) = \begin{pmatrix} \mathbf{A}_{00} \otimes \mathbf{B}_{00} & \mathbf{A}_{01} \otimes \mathbf{B}_{01} \\ \mathbf{A}_{10} \otimes \mathbf{B}_{10} & \mathbf{A}_{11} \otimes \mathbf{B}_{11} \end{pmatrix} = \begin{pmatrix} \mathbf{A}_{00} & \mathbf{A}_{01} \\ \mathbf{A}_{10} & \mathbf{A}_{11} \end{pmatrix} * \begin{pmatrix} \mathbf{B}_{00} & \mathbf{B}_{01} \\ \mathbf{B}_{10} & \mathbf{B}_{11} \end{pmatrix}\,, \tag{4.9}$$

where $\otimes$ denotes the Kronecker product, and this specific operation – Kronecker products applied independently to different blocks of two matrices – is known as a Khatri-Rao product, denoted here as $*$.

4.3 Numerical Implementation

In this section, we will report some preliminary results about the FIM eigenvalue spectra of models trained on the MNIST. The MNIST[3] is a dataset of 70,000 handwritten digits, each in the form of a 28×28 pixel image, where pixels are assigned a numerical value from 0 to 255 to indicate color (white to black).

As we have already mentioned, the analytical results in Section 4.2 were obtained by making use of assumptions that are most appropriate for models with continuous outputs, whereas the data in the MNIST is

[3] http://yann.lecun.com/exdb/mnist/

Figure 4.2: *Sample of images of handwritten digits in the MNIST. Source: LeCun et al., 1998 [7].*

characterized by 10 discreet labels (one for each digit). Nevertheless, we are going to use these analytical results to study neural networks trained on the MNIST, hoping that we will still gain some useful insights.

We consider a simple neural network, with an input layer of size 784 (28^2), a single hidden layer of size 25 with ReLU activation functions, and an output layer of size 10 with SoftMax activation functions[4]. The network was trained on the 60,000 images in the MNIST training dataset using a quadratic loss function, ADAM optimizer, and mini-batches of size 128 (see Section 2.1). The trained model had an accuracy of 0.96 on the 10,000 images in the MNIST test dataset. After choosing one input, shown in Fig. 4.3, we have used Eq. (4.8) to compute the FIM. The largest eigenvalues of the resulting FIM are shown in Fig. 4.3. We can notice that the eigenvalues span many orders of magnitude, which is ins line with the conclusions of Transtrum et al. [13] that neural networks can be characterized as sloppy models. Furthermore, we notice that out of the 19,885 eigenvalues, only 8, shown in Fig. 4.3, are large enough that we were able to easily compute them without running into numerical errors.

Since there is some stochasticity in the training procedure, due to the use of mini-batches and the random initial parameters, we trained several networks to better test our results. In total, we trained 30 networks: 10 with one hidden layer of 25 neurons, 10 with one hidden layer of 16 neurons, and 10 without any hidden layers. We computed the FIM eigen-

[4]Given the output vector $\mathbf{y}$, $\text{SoftMax}(y_i) = e^{y_i} / \sum_k e^{y_k}$

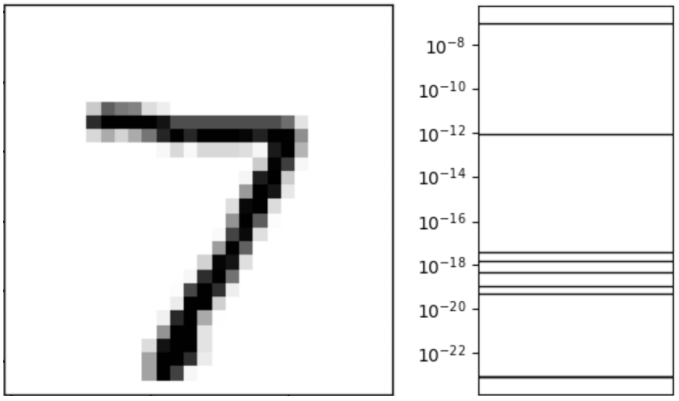

Figure 4.3: *The largest eigenvalues of the FIM of a trained neural network (right) and the input image used in the computations (left), labeled as "seven."*

values for each network using the same input image as before. The results are shown in Fig. 4.4. We see that the addition of hidden neurons seems to have the only effect of spreading out the distribution of eigenvalues.

One factor that may affect the eigenvalue spectrum is whether the network is classifying the input image correctly. All 30 networks correctly classified the handwritten "seven" we have used as input so far, so we have looked in the MNIST for an image that was harder for the networks to classify. We settled on the odd-looking number "four" shown in Fig. 4.5. Only one out of the ten trained networks with no hidden layers cor-

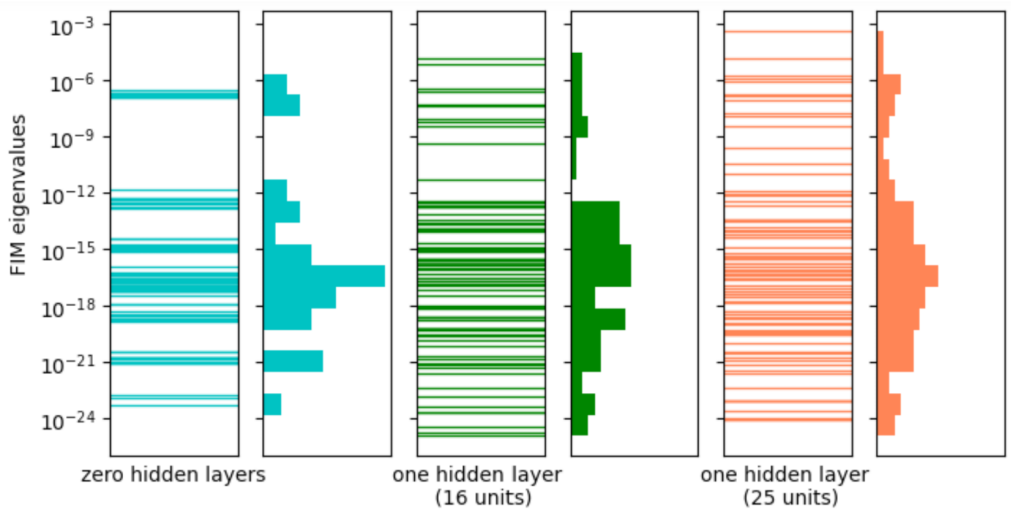

Figure 4.4: *The largest eigenvalues of the FIM of 30 trained models.*

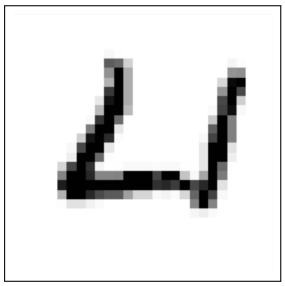

Figure 4.5: *Image of a hand-written digit, labeled as "four." This image was especially difficult to classify for the neural networks we trained.*

rectly classified the image. This number increased slightly to two out of ten for the networks with a 16-neuron hidden layer, while networks with 25 neurons in the hidden layer performed much better: eight out of ten correctly classified the image as "four".

Figure 4.6 shows the FIM eigenvalues spectra of all 30 networks when this hard-to-classify input is used. We can see that, in addition to spreading out the eigenvalue distribution, the FIM eigenvalues are larger for networks without hidden layers, which for the most part classify the input incorrectly, and smaller for networks with one 25-neuron hidden layer, most of which are successful in classifying the image.

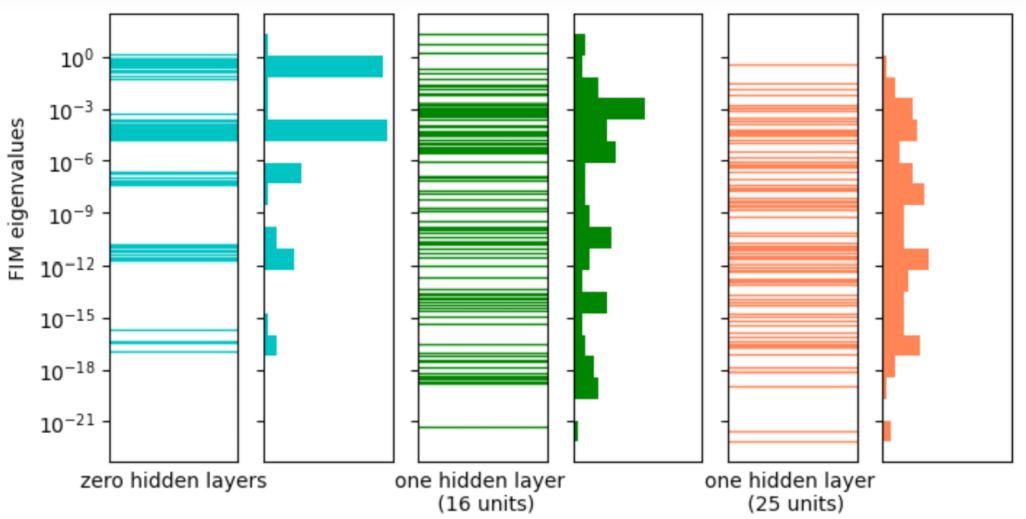

Figure 4.6: *The largest eigenvalues of the FIM of 30 trained models when the hard-to-classify number "four" is used as the input. Only one out of ten networks without hidden layers classified the input correctly, while eight out of ten networks with 25 hidden neurons did.*

4.4 Directions for Future Work

In the previous section, we only reported some preliminary results, but they clearly show the presence of a hierarchy of parameter combinations in neural network models. For any one input, a handful of parameter combinations account for most of the model's behavior. In light of this observation, several questions arise. First of all, while neural networks are sloppy models when a single input is considered, can they still be said to be sloppy in their performance on an ensemble of inputs? In other words, could some parameter combinations that were unimportant for classifying the handwritten "seven" and "four" we used as our inputs actually be crucial for classifying a different digit? It would be interesting to study whether neural networks' sloppy model behavior depends on the input we use, and what the eigenvalue spectra look like when more than one input is considered.

Another question concerns optimization: if relatively few parameter combinations account for most of the model's behavior, is it possible to isolate the relevant parameter combinations and construct a simpler statistical model, with far fewer free parameters than the full network, but performing about as well? Reducing the number of free parameters could help avoid overfitting, and potentially improve the network's performance on unseen inputs.

Finally, could studying the FIM's eigenvectors provide a way to better understand the inner workings of a trained network? The eigenvectors that correspond to the largest eigenvalues are the combinations of parameters that have the largest effect on the network's behavior. We can imagine that, since they have the most noticeable effect, they are capturing the most important features of the training dataset: is this true? Can we find a way to identify these features? Perhaps, the neural network is extracting features we can intuitively understand – e.g. one large eigenvalue could be interpreted as identifying the typical round shape of a handwritten "zero" in the input image – or they may not easily line up with our intuition of what it takes to identify a handwritten digit. Either possibility, in its way, would be an interesting finding.

We leave the exploration of these promising questions to future work.

Appendix A

The Meijer G-Functions

Meijer G-functions are very general and extremely well-studied special functions. Given parameters $m, n, p, q, \mathbf{a} = (a_1, \ldots, a_p), \mathbf{b} = (b_1, \ldots, b_q)$ and complex argument x, Meijer G-functions are defined as line integrals in the complex plane:

$$G_{p,q}^{m,n}\left(x \,\middle|\, \begin{matrix} \mathbf{a} \\ \mathbf{b} \end{matrix}\right) = \frac{1}{2\pi i} \int_{\mathcal{C}} \mathrm{d}s \; x^s \, \frac{\prod_{j=1}^{m} \Gamma(b_j - s) \prod_{k=1}^{n} \Gamma(1 - a_k + s)}{\prod_{k=n+1}^{p} \Gamma(a_k - s) \prod_{j=m+1}^{q} \Gamma(1 - b_j + s)}, \tag{A.1}$$

where $\mathcal{C}$ is the Mellin-Barnes contour, shown in Fig. A.1.

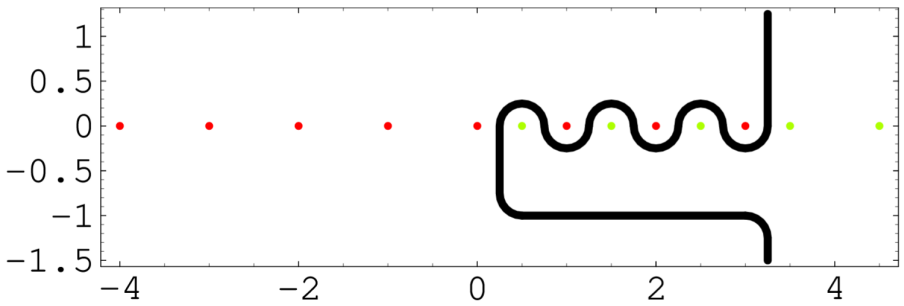

Figure A.1: *The Mellin-Barnes contour in the complex plane for* $G_{1,1}^{1,1}\left(x \,\middle|\, \begin{matrix} 4 \\ 1/2 \end{matrix}\right)$. *The green dots represent the poles of* $\Gamma(\frac{1}{2} - s)$, *while the red dots are the poles of* $\Gamma(1 - 4 + s)$. *The Mellin-Barnes contour* $\mathcal{C}$ *is represented by the solid black line and goes from* $-i\infty$ *to* $+i\infty$. *Source: Weisstein, Eric W. "Meijer G-Function." From MathWorld–A Wolfram Web Resource. https://mathworld.wolfram.com/MeijerG-Function.html.*

Meijer G-functions reduce to other known functions for certain choices of the parameters. For example:

$$G^{1,2}_{2,2}\left(x \,\middle|\, \begin{matrix} 1,1 \\ 1,0 \end{matrix}\right) = \log(1+x)\,, \tag{A.2a}$$

$$G^{1,2}_{2,2}\left(x \,\middle|\, \begin{matrix} 1,1 \\ 1,1 \end{matrix}\right) = \frac{x}{1+x}\,, \tag{A.2b}$$

$$G^{1,0}_{0,2}\left(\frac{x}{2} \,\middle|\, \begin{matrix} - \\ 0,\frac{1}{2} \end{matrix}\right) = \frac{\cos\left(\sqrt{2}x\right)}{\sqrt{\pi}}\,, \tag{A.2c}$$

$$G^{1,0}_{0,2}\left(\frac{x^2}{4} \,\middle|\, \begin{matrix} - \\ \frac{\nu}{2},-\frac{\nu}{2} \end{matrix}\right) = J_\nu(x)\,, \tag{A.2d}$$

where $J_\nu(x)$ is a Bessel function.

There are plenty of known identities involving Meijer G-functions (see section 5 of [4]). A few examples are:

$$G^{m,n}_{p,q}\left(x \,\middle|\, \begin{matrix} \mathbf{a} \\ \mathbf{b} \end{matrix}\right) = G^{m,n}_{p,q}\left(\frac{1}{x} \,\middle|\, \begin{matrix} 1-\mathbf{a} \\ 1-\mathbf{b} \end{matrix}\right), \tag{A.3a}$$

$$x^\rho\, G^{m,n}_{p,q}\left(x \,\middle|\, \begin{matrix} \mathbf{a} \\ \mathbf{b} \end{matrix}\right) = G^{m,n}_{p,q}\left(x \,\middle|\, \begin{matrix} \rho+\mathbf{a} \\ \rho+\mathbf{b} \end{matrix}\right), \tag{A.3b}$$

$$G^{m,n}_{p,q}\left(x \,\middle|\, \begin{matrix} \alpha,\mathbf{a} \\ \mathbf{b},\alpha \end{matrix}\right) = G^{m,n-1}_{p-1,q-1}\left(x \,\middle|\, \begin{matrix} \mathbf{a} \\ \mathbf{b} \end{matrix}\right), \tag{A.3c}$$

$$z^h\, \frac{\mathrm{d}^h}{\mathrm{d}z^h} G^{m,n}_{p,q}\left(x \,\middle|\, \begin{matrix} \mathbf{a} \\ \mathbf{b} \end{matrix}\right) = G^{m,n+1}_{p+1,q+1}\left(x \,\middle|\, \begin{matrix} 0,\mathbf{a} \\ \mathbf{b},h \end{matrix}\right). \tag{A.3d}$$

Version of June 16, 2023– Created June 16, 2023 - 07:37

Bibliography

[1] S. Amari and H. Nagaoka. *Methods of Information Geometry*. Translations of mathematical monographs. American Mathematical Society, 2000.

[2] A. C. C. Coolen. Statistical mechanics of neural networks. lecture notes, king's college london., 1997.

[3] A. Engel and C. van den Broeck. *Statistical Mechanics of Learning*. Cambridge University Press, 2001.

[4] B. Hanin and A. Zlokapa. Bayesian interpolation with deep linear networks, 2022.

[5] R. Karakida, S. Akaho, and S. Amari. Universal statistics of fisher information in deep neural networks: mean field approach. *Journal of Statistical Mechanics: Theory and Experiment*, 2020, 2018.

[6] R. Karakida, S. Akaho, and S. Amari. Pathological spectra of the fisher information metric and its variants in deep neural networks. *Neural Computation*, 33:2274–2307, 2019.

[7] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[8] Q. Li and H. Sompolinsky. Statistical mechanics of deep linear neural networks: The backpropagating kernel renormalization. *Physical Review X*, 11(3), Sep 2021.

[9] B. Machta, R. Chachra, M. Transtrum, and J. Sethna. Parameter space compression underlies emergent theories and predictive models. *Science*, 342(6158):604–607, Nov 2013.

[10] P. Mehta, M. Bukov, C. Wang, A. Day, C. Richardson, C. Fisher, and D. Schwab. A high-bias, low-variance introduction to machine learning for physicists. *Physics Reports*, 810:1–124, May 2019.

[11] A. Raju, B. Machta, and J. Sethna. Information loss under coarse-graining: A geometric approach. *Physical Review E*, 98(5), Nov 2018.

[12] M. Transtrum, B. Machta, K. Brown, B. Daniels, C. Myers, and J. Sethna. Perspective: Sloppiness and emergent theories in physics, biology, and beyond. *The Journal of Chemical Physics*, 143(1), 07 2015. 010901.

[13] M. Transtrum, B. Machta, and J. Sethna. Geometry of nonlinear least squares with applications to sloppy models and optimization. *Phys. Rev. E*, 83:036701, Mar 2011.

[14] J. Zavatone-Veth and C. Pehlevan. Exact marginal prior distributions of finite bayesian neural networks. In M. Ranzato, A. Beygelzimer, Y Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 3364–3375. Curran Associates, Inc., 2021.