

Abstract

The process of learning can be thought of as correctly extrapolating from a set of examples. It is the process of trying to guess some ‘true’ function Q from some examples, \mathcal{D} , of the action of Q on a subset of its domain. Since only partial information about Q is given in \mathcal{D} , a highly expressive learning agent can fit many functions to \mathcal{D} . To decide which function is a good approximation for Q , such an agent needs an inductive bias.

Deep Neural Network (DNN) based learning algorithms are highly expressive and still generalise well in many different situations, meaning that they must have a ‘good’ inductive bias. However, the nature of this bias is not well understood.

In the first part of the dissertation, we will explain how arguments from Algorithmic Information Theory show that certain classes of input-output maps will, if biased, be biased towards simple outputs. We will show in the second part that the map from the parameters of DNNs to the functions they express on some input \mathcal{X} (the parameter-function map) satisfies the conditions for these theorems to apply. As a result, we discuss empirical evidence and analytic results which suggest that DNNs trained by randomly sampling parameters indeed have a strong inductive bias towards simple functions. However, DNNs are not trained in this way, due to computational complexity – they are typically trained by a Stochastic optimiser like Stochastic Gradient Descent (SGD).

One of the main arguments in the literature is that the inductive bias of DNNs arises from some (as yet unknown) property of SGD. In the third part, we will empirically investigate this claim. We will do this in the following way. For some DNN N , training data \mathcal{D} and stochastic optimiser O , we calculate the probability $P_O(f|\mathcal{D})$ that a DNN N trained with O converges on a function f consistent with training set \mathcal{D} . We will then use Gaussian processes to approximate the probability $P_{Bayes}(f|\mathcal{D})$ that N expresses f when training by random sampling of its parameters. We find that $P_{Bayes}(f|\mathcal{D}) \approx P_{SGD}(f|\mathcal{D})$ for a wide variety of architectures and datasets, implying that the main source of the inductive bias of DNNs is a strong bias in the parameter-function map, and is not due to a special property of SGD.

We further show how this (and other) experiments shed new light on how variations in architecture and hyperparameter settings (e.g. batch size, learning rate, and optimiser choice) affect DNN performance.

Acknowledgements and Notes

The majority of the ideas and results focusing on DNNs in this dissertation are a result of the work of Guillermo Valle Pérez and Ard Louis; and much of the work on the AIT bounds are due to Kamaludin Dingle, Ard Louis and Guillermo Valle Pérez.

With regards to the text, Part I and Part II is a reformulation of work predominately due to Guillermo, Kamal and Ard (except work focusing on the perceptron, which was a result of work by Joar Skalse, Vladimir Mikulik, David Martínez Rubio, Guillermo, Ard and myself). The aim was to explain the ideas in those papers in a slightly different way, plus provide some extra background. However, there is no substitute for the original texts [1, 2, 3, 4], where substantially more detail (and probably better prose) can be found.

The work in Part III comes from is original work. A huge contribution was made by Joar Skalse (interpreting the results and finding better ways of understanding them as well as help with the code), Guillermo Valle Pérez (interpreting the results and providing much of the Gaussian Processes code and a limitless supply of associated knowledge) and Ard Louis (interpreting the results, suggesting new and invariably interesting directions, and compiling a list of related work found in Appendix B).

Some of the background detail (in Parts I and II) is likely to be considered elementary by experts, although may prove helpful for those less familiar with algorithmic information theory and associated topics. Detail is provided on Turing Machines in Section 2, Kolmogorov complexity in Section 3, Solomonoff Induction in Section 4, and DNNs in Section 7. A knowledgeable reader may prefer to skip these sections.

1 Introduction

1.1 Development - Conventional AI - A brief history of AI

The idea of an ‘artificial intelligence’ has dated back at least as far as the ancient Greeks (for example, the robot Talos from *Argonautica*); and many philosophers, including Aristotle, Descartes, and Leibniz, have written on the topic. But how could you actually make a machine that can learn from its experiences, the hallmark of AI? To answer such a question, we first need to know what type of ‘machine’ can ‘learn.’ One of the most important theories for AI, the *Church-Turing Thesis* [5], implies that a machine shuffling symbols according to formal rules can imitate any process of mathematical deduction. If we assume that ‘learning’ can be reduced to mathematical deduction, then perhaps such a ‘machine’ could ‘learn.’

One example of such a machine is the ‘Universal Turing Machine’ [5]. Versions of such devices – computers – were built in the 1940s and 1950s. At a similar time, research in neurology suggested that the brain was an electrical network of neurons [6] – it was hoped analogously to computers. With these breakthroughs came much optimism and discussions that culminated in the foundation of the field of AI [7]. One of the early pioneers, Marvin Minsky, wrote in the 1960s that: “within a generation ... the problem of creating ‘artificial intelligence’ will substantially be solved” [8].¹ However, many of the early ‘learning agents’ which enjoyed success on toy problems failed to scale well (see for example [9, 10]).

To understand why most early algorithms failed, one needs to consider what an effective learning agent would look like. We can judge a learning agent based on the following attributes²: its *scalability*, its *expressivity* and its ability to *generalise*. Firstly, a learning agent should not just perform well on simplified/small-scale versions of a real-world problem: its performance should *scale* to the actual real-world problems (or between problems of a similar nature). Secondly, we want our learning agent to be *expressive* – it needs to be able to represent a sufficiently complex

¹Arguably, this problem has still not been solved, although significant progress has been made since then.

²Note that this is not an exhaustive list; but we will find it useful to think about learning agents in this way in the rest of the dissertation.

function to accurately model the data. Finally, and most relevant to this dissertation, we need it to *generalise* well – i.e. successfully extrapolate from examples.

Many of the early learning algorithms satisfied only two of these criteria: for example, Inductive Logic Programming (ILP) is very expressive and generalises well, but is exponential in training time [11]. Lookup tables can be expressive and fast, but clearly do not generalise (arguably they are not learning agents, as generalisation is synonymous with learning). Finally, algorithms with low VC dimension (for example, the perceptron) generalise well and scale well, but are not expressive (a perceptron cannot represent an XOR function) [12, 13, 14]. One especially common problem with conventional learning agents that performed tasks such as image recognition was an inability to generalise from the raw input [15]. Preprocessing the data into a form the learning agent could generalise from “required careful engineering and considerable domain expertise” [15], resulting in issues of *scalability*.

1.2 In this dissertation

In this dissertation, we will concentrate on Deep Neural Networks (DNNs) [15, 16]. DNNs are parameterised learning agents that are (in supervised learning) typically trained with a stochastic optimiser that seeks to minimise a loss function on a training dataset \mathcal{D} (a loss function measures the difference between the function expressed by the DNN and the target function). DNN based learning agents have made major advances in solving problems with which conventional learning agents struggle, without substantial engineering by hand [15].

They satisfy all three of the above conditions to a greater extent than other learning agents. There are theories which prove that DNNs are highly *expressive* (Universal Function Approximation Theories, [17]). It is known that DNNs *scale* (e.g. in complexity of problems [15]) and *generalise* well in many different situations, *without careful preprocessing of raw data*. The following list of problems on which DNN based learning algorithms have performed well is far from complete (see [15] for a more extensive list), but illustrates many of their capabilities: image recognition [18, 19, 20], speech recognition [21, 22], analysing particle accelerator data [23], and sentiment analysis [24]. DNN based algorithms have beaten the world champion in Go [25], and demolished the best conventional

chess AIs [26].

DNNs are typically used when highly expressive: they are typically able to fit many different functions to training data \mathcal{D} . Standard learning theory approaches [27], based for example on model capacity, typically predict that such highly expressive [17, 28] DNNs should fail to generalise. Therefore, there must be an implicit inductive bias in DNNs towards functions that generalise well. This conundrum was famously illustrated in [29]. It was shown that AlexNet (a DNN which has 61 million parameters [19]) achieves good generalisation on the CIFAR10 dataset³, and yet is sufficiently expressive to fit functions to randomly labelled images *that generalise poorly*.

The source and nature of the inductive bias is not well understood, despite much work on the topic, a significant proportion of which suggests that optimisers like Stochastic Gradient Descent (SGD) are responsible [30, 31, 32, 33, 34, 35, 36]. For a detailed summary of this work, see Appendix B. In this dissertation, we will present an alternative explanation: that the inductive bias in DNNs is predominately due to their parameter-function map (the map from the parameters of a DNN to the function it expresses).⁴ We will do so in the following three parts.

In Part I, we will define many of the terms necessary to fully understand the theory behind learning – we need to know what ‘learning’ is to start with! We outline why if a ‘learning’ agent is to successfully ‘learn’ an explanation for some data, then the correct ‘inductive bias’ is necessary for highly expressive learning agents. We briefly argue that a bias which chooses the ‘simplest’ possible explanation of the data with high probability is likely to work well for real world problems. We then lay the required foundations to prove the following: that upon feeding random inputs into ‘simple’ input-output maps, the probability that a function f is expressed by the map satisfies $p(f) \leq 2^{-K(f)+\mathcal{O}(1)}$, where $K(f)$ is the Kolmogorov complexity of f and $\mathcal{O}(1)$ is a constant independent from f . The more general version of the bound is from [37], although this version was first used in [3], and more recently applied to DNNs in [1].

In Part II, we will turn our attention to applications of this result on DNNs. We consider empirical

³a dataset consisting of 60000 images with 10 classes of images; classes include horses and airplanes

⁴first hypothesised in [1]

evidence and analytic results from [1, 4], which suggest that the parameter-function map of DNNs is biased towards simple functions. Concretely, the results suggest that upon randomly sampling parameters of DNNs (e.g. from i.i.d. Gaussians), the network is exponentially more likely to express simple functions than complex functions. We argue (with help from other results in Part I) that *if* DNNs were trained by randomly sampling parameters, this property would be sufficient for good generalisation. However, in practice DNNs are (typically) trained with a stochastic optimiser (because training by randomly sampling parameters is impossible in practice for reasons of computational complexity). [1, 3, 4, 38].

In Part III we provide new extensive empirical evidence that optimiser-trained DNNs and DNNs trained by randomly sampling parameters find functions with similar probabilities. We will do this in the following way. For some DNN N , training data \mathcal{D} and stochastic optimiser O , we calculate the probability $P_O(f|\mathcal{D})$ that a DNN N trained with O converges on a function f consistent with training set \mathcal{D} . We will then use Gaussian processes to approximate the probability $P_{Bayes}(f|\mathcal{D})$ that N expresses f when trained on \mathcal{D} by random sampling of its parameters. We find that $P_{Bayes}(f|\mathcal{D}) \approx P_{SGD}(f|\mathcal{D})$ for a wide variety of architectures and datasets, implying that the main source of the inductive bias of DNNs is a strong bias in the parameter-function map, and is not due to a special property of SGD. We also show that $P_{Bayes}(f|\mathcal{D})$ is many orders of magnitude larger for functions which generalise well than those which generalise badly. The empirical evidence will also shed light on a number of related questions, such as how the training method (commonly a variant of Stochastic Gradient Descent) can affect the bias at initialisation. For example, our approach allows a fine-grained study of effects due to varying the *batch size* or *overtraining*, first pointed out in [30, 39].

This represents a (hopefully significant) step towards understanding generalisation in DNNs.

Part I

Learning Agents & Algorithmic Information Theory

Suppose a computer program Q is known to output n integers, but all other properties are unknown. Assume that it is suddenly aborted, after displaying m integers, where $m < n$. You want to write a computer program $P \in \mathcal{H}$ (where \mathcal{H} is the set of computer programs know how to write) that behaves exactly like Q : it should output *all n integers correctly*, exactly as Q would have done had it not been aborted. There are an *infinite number of candidates*, if P is permitted to be arbitrarily complex. A silly example of a valid P would be a program that memorises the m integers and then outputs 1, $n - m$ times. But how do we select some P as our candidate for Q ? There is no a-priori probability distribution over Q (i.e. no way of deriving from first principles how probable any particular Q is) [40], *and thus no a-priori probability distribution over P* . To progress, we need to make an assumption about the distribution over Q – which would be an ‘inductive bias’ (or prior). Before we continue discussing these ideas, it is important to rigorously define some of the terms used loosely in the above example, such as ‘learning’ and ‘inductive bias.’

Guillermo Valle: tho finite modulo their behavioru on the n output integers

Guillermo Valle: and thus no a-posteriori distribution over P?

Definition 1.1 ((Supervised) Learning and Learning Agents). *In computational learning theory, we formalise ‘learning’ in the following way. Consider a set \mathcal{X} of possible inputs, a set \mathcal{Y} of possible observations, and a set of (potentially non-deterministic) functions from \mathcal{X} to \mathcal{Y} (denoted $\mathcal{Y}^{\mathcal{X}}$). [For example, \mathcal{X} could be the space of images of cats and dogs, \mathcal{Y} could be the labels {cat,dog} and $\mathcal{Y}^{\mathcal{X}}$ is the set of possible labellings]. Assume there is some ‘true’ function $Q \in \mathcal{Y}^{\mathcal{X}}$, that represents some phenomenon. [In our example, this would define the ‘true’ labelling of images]. A training set \mathcal{D} is a finite set $\{\langle x_1, Q(x_1) \rangle \dots \langle x_n, Q(x_n) \rangle\}$ of examples of instances $x_1 \dots x_n \in \mathcal{X}$ and the action of Q on these instances. [In our example, this would be a set of example pictures and their labels]. A supervised learning agent is an algorithm which receives \mathcal{D} and returns a function P in $\mathcal{H} \subset \mathcal{Y}^{\mathcal{X}}$ (the hypothesis class; functions the learning agent knows about) being successful if $P = Q$ (to some accuracy). In the case that such a $P \notin \mathcal{H}$, the learning agent is insufficiently expressive. In the case of multiple functions $P \in \mathcal{H}$, an inductive bias is required. A similar definition is given in [41].*

Guillermo Valle: or more specifically, supervised learning,

Guillermo Valle: outputs, "inputs" are also observed xD XD :P

Guillermo Valle: this is is the set of deterministic functions uwu but welll

Guillermo Valle: btw \mathcal{D} is usually used for the true distribution (including the distribution over \mathcal{X} and the true labels), while \mathcal{S} is typically used for training set.

Just saying in case u want the convention

Guillermo Valle: is able to output

Guillermo Valle: if u are gonna say to some accuracy just say P is close to Q , not $P=Q$ >

Guillermo Valle: $u \text{ mean } Q \setminus \text{nin } H$

Guillermo Valle: may be . It is not, if D leaves only one $P \setminus \text{in } H$

Guillermo Valle: I would add a paragraph somewhere before this, explaining that a learning agent most often tries to find Q , by first looking at functions in H which are compatible with the training data (which is the only empirical evidence about Q which the agent has). After that, it chooses according to the inductive bias, which we define informally in the following.

Definition 1.2 (Inductive Bias and Priors). *An inductive bias is a mechanism that determines which function a learning agent does output if more than one function in \mathcal{H} is considered, by the learning agent, to be compatible with the training data \mathcal{D} . Informally, it is a set of facts of assumptions that dictate which possible extrapolation from \mathcal{D} is most likely to be true. A prior is a probability distribution over functions (prior to seeing \mathcal{D}). This is used when discussing Bayesian learning algorithms. See Section 4 for details. A prior can be used as an inductive bias.*

It should be clear that Definitions 1.1 and 1.2 are consistent with our intuitions of learning and inductive biases – compare these definitions to the example at the beginning of the section.⁵ We can conceptualise a learning agent as a function approximator equipped with an inductive bias.

Remark 1.3 (Which prior?). *There have been many attempts to justify that a ‘simple’ hypothesis is more likely to be correct [42], the essence of Occam’s razor (with varying degrees of success). We will gradually formalise this idea over the next few sections. A brief informal justification goes as follows: many of the problems we wish to solve by ‘learning’ involve finding a simple explanation underlying complex observations. It is asserted in [43] that many AI researchers agree with this principle. We will give this argument more formal treatment in Section 4.*

Remark 1.4 (The no free lunch theorem for classification). *Consider a learning agent. Then, if a uniform distribution over possible problems (Q) is assumed, and if \mathcal{H} is sufficiently expressive, the learning agent will on average achieve a performance no better than random. [44]. A less rigorous related fact: a prior will only give good performance on its corresponding problem. For example, an Occam’s razor type prior would not give good results if Q produced random outputs.*

Guillermo Valle: on a matching problem? whatever, just wording

Guillermo Valle: (and thus non-simple)

Guillermo Valle: UTMs are not really "simplest machines", also i thought we will be using them to model the distribution of problems (i.e. the "world") rather than the learning agent (which need not be a UTM, e.g. a DNN)

Now we have our initial definitions and assumptions in place, we can begin to move on towards understanding how inductive biases favouring ‘simplicity’ might arise. To do this, we will begin by discussing the simplest examples of machines from which we can build our ‘learning agents’ (Universal Turing Machines). We will then use Kolmogorov complexity and Solomonoff Induction (ideal behaviour in the task of sequence prediction) to formalise Occam’s razor with Kolmogorov complexity. Finally, we will show the main result in this part: that certain classes of input-output maps are biased towards functions with low Kolmogorov complexity.

⁵Note that we will only discuss supervised learning in this dissertation (learning from ‘training data’ as in Definition 1.1. It is very common when performing classification or regression tasks [41]. We will refer to it as learning.

Guillermo Valle: with labels

2 Turing Machines & Universal Turing Machines

A Turing Machine (TM) is a hypothetical device **invented** by Alan Turing to capture the notion of mechanical computation [5] – broadly speaking, the process of mapping inputs to outputs with rules (which for our purposes are taken to be deterministic). It allows us to reason rigorously about the limits and possibilities of computation, and was originally used to prove the existence of uncomputable functions. A TM is commonly thought of as a program.

Guillermo Valle: or discovered XD xD :P

For a formal definition of a Turing Machine, see Section 3.1 in [45]. We also give extensive treatment in Figure 1, but in brief, a Turing Machine accepts inputs (which can be considered to be strings of symbols) and performs actions based on rules, until instructed to halt and return a string. **As mentioned in the introduction, one of the most important conjectures concerning computation is Church's Thesis:** for every partial function (a function $f : \mathcal{X} \rightarrow \mathcal{Y}$ that is not necessarily defined on all elements of \mathcal{X}) that is computable by an effective method (i.e. is computable), there is a corresponding Turing Machine. The correspondence is with partial functions because some Turing Machines will never stop running (not 'halt') and return a string, for some inputs. Determining whether this will occur is a famous problem (we will provide the solution in Theorem 2.3).

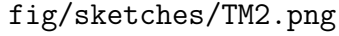
Guillermo Valle: why are you introducing this ? Is it used anywhere?
Also it's a big vague currently as "effective method" is not defined (tbh Church's Thesis is a big vague itself too most of the time)

In summary, we have described an abstract definition of computation: a device shuffling symbols, which is in 1-1 correspondence with the collection of (computable) partial functions. However, this means that this sort of machine will be insufficient if we want to construct a learning algorithm, because each TM can only represent one partial function. If we are to construct a learning agent, we need a notion of a **Generalised** TM, which is able to receive an encoding of any TM M (a 'program') and an input to M , and simulate M . This is called a Universal Turing Machine (UTM).

Guillermo Valle: dont capitalize, seems like u are definingn a new term

Definition 2.1 (Universal Turing Machines). *A UTM U is a Turing Machine that can imitate any other Turing Machine M : given an input string that encodes the description of M , and some input to M , U will enter the **same final state** (or not halt) as M would have done on the same input. This can be done by associating each TM with a natural number (See [38] for proof). Note that, as every UTM is also a TM, any UTM U can simulate any other UTM V .*

Guillermo Valle: including the output



fig/sketches/TM2.png

Figure 1: There are many provably equivalent definitions of a Turing Machine (TM). We will use the one from [45]. A TM consists of a tape which can be indexed by \mathbb{N} (i.e. the tape has one ‘end’ and stretches off to infinity in the other direction), and a ‘head’ which is positioned over one of the squares on the tape. A finite number of symbols are allowed on the tape (e.g. $\{0, 1, \text{blank}\}$). At the start of computation, the tape is filled with non-blank symbols (for this example. $\{0, 1\}$) up to a finite index m , beyond which all symbols are ‘blank.’ A Turing Machine consists of this tape, and a ‘head’ which has an associated ‘state.’ States can be thought of as a finite deck of cards, each card having a finite label $l \in \mathcal{N}$ (the state), and rules based on the symbol underneath the head. In this dissertation the rules are considered to be deterministic. These rules allow the head to (1) print a symbol, (2) move left or right, and (3) select a new card from the deck. There is one other possibility, that the machine halts and returns either 1 or 0. Thus, a Turing Machine is a map from programs (which can be represented by strings encoding the above information) to output strings (in the case above, these strings are either 1 or 0). While there are different ways of thinking about ‘programs’ or ‘inputs’, we will consider the ‘input’ of a Turing Machine to be the initial state q_0 and tape in state \mathcal{T}_0 . For a ‘Universal Turing Machine’ (UTM), see Definition 2.1, which is a Turing Machine capable of simulating other Turing Machines, the initial state is the concatenation of the encoding of M (on the tape), the encoding of the input to M , and the initial state of the UTM q_0 . The part of the tape which encodes M is called the ‘program’, and in a slight abuse of notation, the ‘input’ is considered to be the input of M . The length of the program is the number of squares on the tape encoding M .

Definition 2.2. A program P is a string which can be accepted by a UTM, U , that encodes a TM M . The length of P is the length of the string. See Figure 1 for this definition in terms of the ‘tape.’

Theorem 2.3 (The Halting Problem). No program exists that can determine whether any general Turing Machine Q will halt (stop computing) or run forever. [5].

Proof sketch. Suppose, for the sake of contradiction, there exists a program P which solved the ‘halting problem’. It receives a pair, a program Q and an input S , and returns either ‘Halt’ or ‘Not halt’, depending on whether program Q would halt eventually if given S . Now, consider a program N , which runs forever if it receives ‘Halt’ and prints 1 otherwise. Consider a program M which takes any input ‘ x ’ and returns the pair ‘(x , x)’. Now we form a pathological Q , by composing M P N (so the output of M is fed into P etc.). Now suppose Q receives Q itself as input. We see that if Q halts on Q then Q will not halt, and vice versa. So we have a logical contradiction. But we can certainly compose machines, and M and N can be built. Thus P cannot be built. \square

Guillermo Valle: don't call it same as previous dummy?

There are infinitely many such UTMs that are inequivalent in some sense (e.g. different ways of encoding M), a consequence of which being that measurable properties of UTMs differ – for example, the number of operations a UTM U performs when running a particular computation may be different from those made by a different UTM V [45]. We can divide UTMs into categories based on their properties, which can be useful for proving certain facts. We will only need to know about prefix UTMs for our later proofs.

Definition 2.4 (Prefix UTM). A prefix UTM U is a UTM that satisfies the following extra condition: if P is a valid program, then it is not possible to create a valid program by removing symbols from the end of the *tape* (see Figure 1 for details).

Guillermo Valle: input tape

The fact that **UTMs exist**, and are relatively computationally inexpensive (i.e. computers are reasonably powerful and do not require **impractical space or energy**) is a major result. In the rest of this part, we will implicitly assume that the computations we are performing are carried out by UTMs: for example, they can simulate any method for function approximation which we know how to mechanically perform. Finally, it is worth noting that real computers are not implemented in the

Guillermo Valle: well teeechnically no computer has infinite memory UwU

Guillermo Valle: tell that to Jonathan

same way as a Turing Machine (Figure 1), although they can obviously simulate Turing Machines. Now we have some formalisation for talking about machines, we can start thinking about learning. In the following two sections we will discuss a measure of complexity of strings (Kolmogorov complexity), and then a hypothetical learning agent that would assign high probability to simple strings (Solmonoff Induction). This will formalise the argument we discussed briefly in Remark 1.3.

3 Kolmogorov Complexity

There were many attempts to define complexity before a theory of formal computation existed (i.e. Turing Machines). Complexity was (as now) identified with the notion that a sequence is ‘simple’ if there is a short rule (i.e. a rule much shorter than the sequence) that describes it; and is ‘complex’ if there is no such rule. A highly complex sequence can, therefore, be thought of as a ‘random’ sequence. Early attempts used probability theory to describe the ‘randomness’ of sequences – however, probability theory cannot express **this**. For example, a sequence of heads from flipping a coin is as likely as *any other sequence s of the same length, no matter the apparent ‘randomness’ of s* . See Section 1.8 of [38] for more detail. With the framework of computation however, the notion of ‘complexity’ be given a satisfactory definition: Kolmogorov complexity [46].

Guillermo Valle: express what
Guillermo Valle: and why
Guillermo Valle: explain this more.
Guillermo Valle: what were the early attempts?

Definition 3.1 (Kolmogorov Complexity). *For some Universal Turing Machine U and string w computable on U , the Kolmogorov complexity of w , $K_U(w)$, is the length of the shortest computer program on U that outputs w . We defined the length of a program in Definition 2.2.*

Remark 3.2. *Because a UTM U can simulate any other UTM V , $|K_U(w) - K_V w| < \mathcal{O}(1)$, where $\mathcal{O}(1)$ terms are constants independent of the string w but dependent on U and V . A trivial bound on the $\mathcal{O}(1)$ term is, therefore, the longer of either the length of the program U needs to run to simulate V , or the length of the program V needs to run to simulate U . If the strings we are dealing with are much larger than the $\mathcal{O}(1)$ terms, we will usually drop the subscript U .*

Example 3.3. *Consider the following sequences: $s_1 = 1111111111\dots$, $s_2 = 011011100101\dots$, $s_3 = 110011101011\dots$. We would naturally say that s_1 is the simplest, on closer inspection we would see that s_2 is the binary numbers printed without delimiters, and observe that s_3 is completely random (there is no rule shorter than s_3 that describes it). The Kolmogorov complexities of the strings would*

reflect this: the program required for s_1 should be shorter than that required for s_2 ; and the only way to produce s_3 is to memorise and return the digits. This grows with the sequence length, but the other two **descriptions do not**, which becomes significant for long sequences.

Guillermo Valle: they do but slower. they grow like $\log(\text{length})$

The idea that a sequence is simple if it can be produced by a simple algorithm is very natural. There is one further fact about Kolmogorov complexity we need to know: it is uncomputable.

Theorem 3.4. *Kolmogorov complexity is uncomputable. [46]*

Proof. Assume for contradiction that a program K can calculate the Kolmogorov complexity of some string w . Now let us define a pathological program \mathfrak{K} , which performs the following operations: for all $i \in \mathbb{N}$, smallest to largest, take all strings v with length i , and calculate their Kolmogorov Complexity using K . Return the first string where $K(v) > n$, for some $n \in \mathbb{N}$. Now, we can make n arbitrarily larger than the length of the program K . Therefore, eventually **\mathfrak{K} is shorter than $K(w)$** for some w , giving a contradiction. \square

Guillermo Valle: is this supposed to be natural numbers?, coz u just used a different notation

Guillermo Valle: than the program {gothic K} or whatever that is ?

Guillermo Valle: and outputs w

Remark 3.5. *Note that at a glance this appears incorrect: to calculate $K(w)$ for some w one could imagine just starting at the shortest program and testing what output it gives, until K produces w . However, some of the programs with length less than $K(w)$ **may not halt**.*

Guillermo Valle: and it's in general impossible to know which

Remark 3.6 (How many strings are simple?). *Consider binary strings of length n : there are 2^n different strings. Consider a fixed compression algorithm. What is the maximum number of strings compressible by p bits or more? There are a total of **$2^{n-p+1} - 2$** strings with length less than or equal to $n - p$. Therefore, as a fraction of the total number of strings length n , we can compress a maximum of $2^{n-p+1} - 2^{n+1}$ strings by p bits or more. This implies that almost all strings are highly complex for large n . So strings created by randomly sampling bits are almost always highly complex; but strings made by running a randomly generated program are (with some conditions) likely to be simple. See the next section for more information.*

Guillermo Valle: u ignoring the empty string i guess?

In summary, we have defined a measure of complexity, 'Kolmogorov complexity', in terms of Turing Machines. By this definition, a string is 'simple' if it can be described by a short computer program. As a computable function can be represented as a (binary) **string**, we can talk about the Kolmogorov complexity of functions too. We will now use these definitions to make a learning agent.

Guillermo Valle: (its program)

4 Solmonoff Induction

Solmonoff induction is a mathematical formulation of Occam's razor using Kolmogorov Complexity. It will provably converge to the correct belief about any hypothesis (with data drawn from a computable measure) [47, 48]. It first defines the universal a-priori probability Q to a string w – *a prior over strings*. Informally, $Q_U(f)$ is the probability that some prefix UTM U will output y , upon running some randomly generated program p on \mathcal{U} .

$$Q_U(w) := \sum_{p \in \mathcal{P} : \mathcal{U}(p)=w} 2^{-l(p)} \quad (= \mathcal{O}(2^{-K_U(w)})), \quad (1)$$

where \mathcal{P} is the set of self-delimiting programs valid on U , and $l(p)$ is the length of a program p . The subscript on Q denotes that this is defined for a particular prefix UTM U (the sum can diverge for non-prefix UTMs). The second equality shows that this probability is dominated by $K_U(w)$ [49]. Then, *Solmonoff Induction* proceeds in the following way: given our *prior* over strings (Equation (1)), some observed string w_0 and some candidate continuation w , the conditional probability of observing w (the posterior distribution) is given by Bayes' rule ($++$ means concatenate strings):

$$Q_U(w|w_0) = Q_U(w ++ w_0) / Q_U(w_0) \quad (2)$$

Unfortunately constructing a *general* learning agent in this way is impossible due to the **Halting problem**. Were this not an issue, the agent would calculate $Q(w|w_0)$ for all possible w – the conditional distribution over future sequences, which would allow us to select a 'best guess' for w .

Remark 4.1 (Extension of definition of learning agent). *Note that for this learning agent to be properly described by Definition 1.1, we would have to add the extra step of extending strings to functions. A learning agent is a triple (\mathcal{L}, R, A) : where L is a set of strings, R is an invertible map from strings to functions in $\mathcal{Y}^{\mathcal{X}}$, and A is an algorithm which receives \mathcal{D} and outputs a string in \mathcal{L} . For example. we can extend Solmonoff induction to functions on discrete and finite data $g(x^{(i)}) = y^{(i)}$ in the following way: encode some example pair of inputs and outputs $\mathcal{D} = (x^{(i)}, y^{(i)})$ in sequence, as a string: this will be f_0 . Encode for test inputs and outputs to generate w , and*

Guillermo Valle: the denominator is abusing notation a bit, its actually the sum of probabilities over all continuations of w_0 , not the probability of w_0 as an output (i.e. TM halting with w_0 in output tape)

Guillermo Valle: more concretely, the uncomputability of $K(w)$

Guillermo Valle: computable map

Guillermo Valle: what does this mean?

also Solomonoff induction is trying to predict both the sequence of Xs and Ys, while normal supervised learning makes an extra assumption that the Xs are i.i.d. distributed from some probability distribution.

proceed with induction ($Q(f|f_0)$ depends on the encoding procedure).

It is an Occam-like (or Kolmogorov simple) prior, or bias, because it assigns the high probabilities to strings produced by short programs. Solomonoff has taken to be a gold standard in inductive inference [47, 50], and as a result we make the common hypothesis that an ideal learning agent would do something similar. We hope that, given \mathcal{D} (training data, see Definition 1.1 for notation), it would choose with high probability candidate functions $P \in \mathcal{H}$ which have low Kolmogorov complexity. In order to discuss this, we will need the Levin Bound [37].

5 Levin's Bounds

We now come to prove the main result: The Coding Theorem (Section 4.3.4 from [38]). The full (and slightly more general) proof can be found there, but we will provide the proof in SI2 of [3], because it lends intuition and can be done with the results we currently have.

Theorem 5.1 (The Coding Theorem). *Consider some prefix-free binary UTM, U (a binary UTM has only two non-blank symbols in γ and Σ , the input and **tape symbols** respectively – see Figure 1). Then, the probability of U expressing some output x , assuming programs p are drawn by selecting input bits until p is a valid program satisfies the following bounds*

$$2^{-K_U(x)} \leq Q_U(x) \leq 2^{-K_U(x) + \mathcal{O}(1)}, \quad (3)$$

where the $\mathcal{O}(1)$ terms are dependent on U , finite, and independent from x . Statement of theorem copied from Section 4.3.4 [38]. We will prove a slightly different version of the upper bound, where we consider a binary (but not prefix) UTM U which has a fixed program size, because each program is made up of (1) a map f which accepts (2) inputs of length n . We assume that the map f is computable so that every such combination of f and input of length n is a valid program. Then,

$$P(x|n, f) \leq 2^{-K_U(x|n, f) + \mathcal{O}(1)}, \quad (4)$$

where $P(x|n, f)$ is the probability of output x , when uniformly sampling inputs, given the map f , and an input length n . This result is valid for maps f of fixed input size n . Note that we write

Guillermo Valle: the input and output tape?
Guillermo Valle: btw cant see fig 1 coz it dont compile

$K_U(x|n, f)$ because this complexity is defined relative to the UTM U on which f is run. The $\mathcal{O}(1)$ terms predominately come from the complexity of the map f . Statement of theorem from SI note 2 of [3].

Guillermo Valle: so you need to say you have assumed map f has complexity of order 1

Proof. Proof of lower bound in Equation (3): This inequality is proved simply by noting that the probability of x is at least $2^{-K_U(x)}$ (because, by definition of $K_U(x)$, there is a program which produces x and is $K_U(x)$ in length, see Equation (1)). It is a lower bound because there may be longer programs that also produce x .

Proof of upper bound: We will consider the upper bound in Equation (4). This is because this form of the upper bound is the one required for Part II. Consider the following algorithm \mathcal{A} :

1. Enumerate all inputs (i.e. programs of length n)
2. Map these inputs to their outputs according to f
3. From this list, return the pairs of outputs and their probabilities (assuming all programs are equally likely)

Since n and f are given, the relative complexity of this algorithm is $\mathcal{O}(1)$. From this, we can generate a Huffman Code. We have from Appendix A that the following bound for Huffman Codes \mathcal{H} holds:

$$l(\mathcal{H}(x|n)) \leq \lceil \log_2 \left(\frac{1}{p(x|n)} \right) \rceil, \quad (5)$$

and creating this code is an $\mathcal{O}(1)$ procedure (this can be shown by explicitly writing down the algorithm) [51]. Therefore, we have an upper bound on Kolmogorov complexity:

$$K_U(x|n) \leq \lceil \log_2 \left(\frac{1}{p(x|n, f)} \right) \rceil + \underbrace{\mathcal{O}(1)}_{\text{Enumerating all } f} + \underbrace{\mathcal{O}(1)}_{\text{Creating Huffman Code}} \quad (6)$$

And, rearranging this gives

$$p(x|n, f) \leq 2^{-K_U(x|n, f) + \mathcal{O}(1)} \quad (7)$$

Under the assumption that the relative complexity of the map f is $\mathcal{O}(1)$ (i.e. the map is equally expensive to compute regardless of f), then an extra $\mathcal{O}(1)$ term associated with the complexity of the map is added to Equation (6), and the bound Equation (7) becomes the bound we will predominately use:

$$p(x|n, f) \leq 2^{-K_U(x|n) + \mathcal{O}(1)}, \quad (8)$$

□

In essence, Equation (8), which we will now refer to as the *Levin Bound*, tells us that if we have a map f , then, if the map is simple (i.e. the $\mathcal{O}(1)$ terms are small), there is an upper bound on probability as an exponentially decreasing function of Kolmogorov complexity. In other words, if the map is **biased**, then it will be biased towards simple functions. For some intuition, consider a uniquely decodable code that encodes letters in the alphabet as binary numbers and has a decoding process (which turns binary code into letters) which is unambiguous (i.e. a sequence of code could not have been made from two different sequences of letters). See Appendix A for a more formal treatment of codes. Consider feeding in random code into a decoder. After decoding this random code, we expect to find letters with short codewords more frequently than letters with long code words, simply because the probability of randomly picking a code word is 2^{-l} , where l is the length of the codeword. Feeding random code will, therefore, be more likely to produce ‘simple’ strings of letters than expected – although note that this is a very coarse measure of simplicity.⁶

Guillermo Valle: sufficiently biased

Obviously, the result we have just proved for computable maps is only an upper bound. **In fact, a uniform distribution over functions could in principle satisfy the bound.** However, arguments from [2] provide a lower bound on $p(x)$, also as a function of $K(x)$. We will briefly go through the argument below.

Guillermo Valle: you can say, the identity map would satisfy the map, and produce a uniform distribution over outputs.

Here we are talking about outputs, not functions, so this is a bit confusing

Theorem 5.2 (Lower bound). *Consider a system identical to the system laid out for Equation (4). Then the probability $p(x|f, n)$ of a map f with fixed input size n producing an output x is lower*

⁶There are further subtleties with which this example needs to contend (for example, whether the input is a valid code etc.). However, as this is meant for intuition only, they will not be addressed here.

bounded by the following expression:

$$p(x|n, f) \geq 2^{-K(x|f, n) - [n - K(p|f, n)] + \mathcal{O}(1)},$$

where $K(p|f, n)$ is the Kolmogorov complexity of the description of the parameters (in the map f) that produce x .

Proof. We will assume that f accepts binary inputs of length n , meaning that there will be 2^n possible inputs. Let $f^{-1}(x)$ be the preimage of x – i.e. the set of inputs to the map f which produce x . Then we can write $p(x|n, f) = \frac{|f^{-1}(x)|}{2^n}$. An arbitrary input j can be described using the following $\mathcal{O}(1)$ procedure: As in the proof of the upper bound, enumerate all 2^n inputs and map them to outputs using f . The index of a specific input p within $f^{-1}(x)$ can be described using at most $\log_2(|f^{-1}(x)|)$ bits. Thus,

$$K(p|f, n) \leq K(x|f, n) + \log_2(|f^{-1}(x)|) + \mathcal{O}(1),$$

where $K(p|f, n)$ is the Kolmogorov complexity of the input p . We can substitute $|f^{-1}(x)|$ for an expression in terms of p using our expression from earlier, $p(x|n, f) = \frac{|f^{-1}(x)|}{2^n}$. Thus,

$$p(x|n, f) \geq 2^{-K(x|f, n) - [n - K(p|f, n)] + \mathcal{O}(1)},$$

which indicates that, given that almost all of the inputs must be complex (see Remark 3.6), indicates that a lot of the probability mass must be in the maximum complexity region. \square

Guillermo Valle: this should go in a remark. also u mean the maximum complexity region for inputs? thus indicating that most inputs maps to outputs close to the bound right?

Remark 5.3. *The upper bound has been applied to several systems in [3], including a map from RNA to proteins via a ribosome. It was shown empirically that, upon feeding random RNA strings of fixed length into an RNA SS map, ‘simple’ proteins (e.g. proteins with symmetry) were much more likely to be produced than ‘complex’ proteins. The lower bound has been applied to several systems, such as the perceptron [14] in [2], and was shown to give qualitatively good predictions.*

So why is this useful? We stated in Section 4 that an ideal learning agent would approximate

Solmonoff induction – out of the functions consistent with training data \mathcal{D} , it would pick the simpler with high probability. The Levin bound (and the lower bound) implies that certain classes of input-output maps may produce low-complexity outputs with high probability upon random inputs. We will show empirical evidence in Part II which suggests that this is the case for DNNs, and explain how a learning agent that assigns high probability to simple functions can be constructed.

Guillermo Valle: noo, these bounds still don't imply the map is biased...

6 Summary

In summary, we have defined a learning agent as a function approximator with an inductive bias. This is because a learning agent must be able to approximate a function on some training data, and correctly choose the most likely extrapolation on unseen data. We argued that an Occam's razor type bias is good for learning, because tasks learning agents encounter are likely to involve finding a simple explanation or description for an apparently complex system.

We then gave brief treatment to the idea that a mechanical device shuffling symbols (a Turing Machine) can simulate any process of mathematical deduction, and stated that learning can be reduced to such a process. We used this construction to define the 'Kolmogorov complexity', where the complexity of a string is defined as the length of the shortest program which produces it. We defined the 'universal a-priori probability' of a string s to be the probability that a randomly generated program running on a prefix UTM will output s . We then argued that this would be a good prior for sequence prediction, and how this generates an 'Occam like' posterior distribution, where Kolmogorov simple strings are favoured.

Guillermo Valle: tbh the whole TM section seems more background to introduce KC and solomonoff than doing what you claim here.

What you claim here is just that "learning" can be done by a program.

We then stated that a learning agent that chooses the simplest function consistent with the training data, in a process approximating Solmonoff induction would be close to an ideal learning agent. In the next section, we will show how the Levin Bound can be used to argue that DNNs assign higher probability to simple functions, before and after training.

Part II

Neural Networks and the Levin Bound

In this part, we apply the Levin bound (see Equation (7)) to the parameter-function map of Deep Neural Networks. We show empirical evidence that DNNs, upon randomly sampling parameters, are more likely to express ‘simple’ functions – a ‘simplicity bias’.⁷ Given that the predictions made by good learning agents are thought to assign high probability to simple functions, we will argue that this may be a secret to the DNNs success. We will first give an in-depth description of Deep Neural Networks in the following few sections. We will then apply the Levin bound, and then discuss one major caveat at the end of the section – that the training process may disrupt this bias (which is observed at initialisation). This will be the topic of Part III.

7 A brief introduction to Deep Neural Networks

A (Deep) Neural Network [15] is perhaps best thought of as a function approximator – keeping in mind that part of the task of a learning agent is to approximate functions. It is a parameterised system $N_\theta : \mathcal{X} \rightarrow \mathcal{Y}$, where \mathcal{X} is the input vector space, \mathcal{Y} is the output vector space, and θ are the **network parameters**. It consists of **layers**. Each layer is made up of two parts: (1) an affine transform applied to the input from the previous layer, followed by (2) a point-wise non-linear function. Each layer’s output becomes the input for the next layer. Formally, this is defined (recursively) below.

$$y_i^n = \sigma^{(n)}(b_i^n + \sum_{j=1}^{i=W_n} w_{ij}^n y_j^{n-1}), \quad (9)$$

$$y_i^1 = \sigma^{(1)}(b_i^1 + \sum_{j=1}^{j=W_1} w_{ij}^1 x_j), \quad (10)$$

where the matrix W_{ij}^n is the **weight** matrix in the n ’th layer; the vector b_i^n is the **bias** vector in the n ’th layer, and $\sigma^{(n)}(z_i) = \sigma^{(n)}(\mathbf{z})_i$ is a point-wise nonlinear function for the n ’th layer. x_j is

⁷Note that this is not a provable result of the Levin Bound, because the bound states only that if a map is biased, it will be biased towards simple functions.

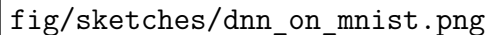
the input. W_i are the **widths** of the layers. The sizes of weight matrices and bias vectors are such that the matrix multiplications are well-defined. The values of the **(network) parameters** θ (the set of **weights**, w_{ij} and **biases** b_i), are what determines the function the DNN expresses. Later, we will examine a procedure called Stochastic Gradient Descent that will search **parameter-space** for parameters that make N_θ close to some target function, where **parameter-space** is defined to be the set of possible values of the network parameters. We give a pictorial representation of how a DNN acts as a function approximator in Figure 2a.

There are many conventions for how the number of layers is counted: we will use the ‘hidden layer’ convention, where the input layer and output layer are not counted (so if the final output of a network is \mathbf{y}^k , the network, the layer has $k - 1$ hidden layers). It is clear at a glance that the structure of DNNs allows for much customisation – the layer widths, number of layers and non-linear functions (a.k.a. nonlinearities) are all customisable. The **architecture** of a DNN is the particular choice of all these parameters. Note that there are also many more architecture choices not captured by Equation (9) here (for example, one can set the values of certain weights to be synchronised or fixed at 0 to build invariances into the network, see Section 7.3 for further details). Choosing the correct architecture for a DNN is very task-specific, and currently there are only heuristics and precedent to guide one’s choice. We give an example of a DNN architecture designed to classify handwritten digits in Figure 2, with ReLU nonlinearities:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise.} \end{cases} \quad (11)$$

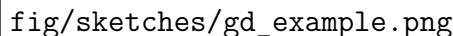
Other nonlinearities used in practice include tanh functions and sigmoid functions, although the ReLU nonlinearities are probably the most popular.

The basic idea behind supervised learning with DNNs is to find some **architecture** which has certain desirable properties, and then find some **parameters** θ such that N_θ is accurate on some training data \mathcal{D} . Then, we hope that N_θ generalises well. We will aim to explain how this is done in the next three sections, where we will first discuss the *Expressivity* of DNNs, their *scalability* (including training) and the extent to which they *generalisation*.



fig/sketches/dnn_on_mnist.png

(a) A pictorial representation of a 1-hidden layer fully-connected DNN.



fig/sketches/gd_example.png

(b) A cartoon designed to provide intuition for how SGD works

Figure 2: A pictorial representation of how DNNs (a) act as a function approximator and (b) how it trains. In (a), we give an example of a 1 hidden layer DNN which acts on images in the MNIST database, which consists of images of handwritten digits, from 0 – 9. There are 5000 examples for each digit, and each image is size 28×28 . We will design a DNN which can map each image to a label. First, we ‘reshape’ each image from a 28×28 grid into a vector of length 784, for example, by mapping the pixel at position (i, j) to the $28i + j$ ’th element of the vector. We construct a DNN with 1 hidden layer and ReLU nonlinearities: (1) an input layer of width 784, (2) a hidden layer width 128, and (3) an output layer of width 10. The classification of the image is taken to be the index of the maximum element in the output vector. In (b) a cartoon contour plot of the loss landscape is shown, and the process of gradient descent is shown. We could use this method to train the DNN in (a). For each input $x^{(i)}$, let $y^{(i)}$ be a one-hot vector of length 10 where the 1 is at the index of the label (for example, an image of an 8 would have $y = [0, 0, \dots, 1, 0]$). Train the network with SGD (use a batch size, for example of 128) using the process described in Section 7.2. Training to 100% accuracy on the 50000 MNIST images can take as little as a few minutes on a decent laptop.

7.1 Expressivity

Before we consider training, or generalisation, we first need to ask ourselves what kinds of functions DNNs can represent. This is best done by example; we will first describe two results which show how expressive DNNs are, and DNNs with more layers can be better than those with fewer layers:

- It was first shown in [17] that single hidden layer DNNs with sigmoid nonlinearities, for some sufficiently large w_1 and choice of weights, approximate any function $f : \mathbb{R} \rightarrow [0, 1]$, such that $f(x \rightarrow \infty) = 1$ and $f(x \rightarrow -\infty) = 0$, with arbitrary accuracy
- It was shown in [52] that DNNs (with ReLU non-linearities) approximating sawtooth functions require exponentially fewer weights if the network is made deeper (more hidden layers), rather than wider (each layer is made bigger).

Whilst these results are useful illustrations of important properties of DNNs, they make no claims about the specific details relevant for learning – for example, precisely how many parameters are required for tasks such as image classification, where the function required may not be as constrained?

- It was shown in [29] that DNNs (e.g. AlexNet [19], which has 61 million parameters in 8 layers) can learn the function from images in the CIFAR10 dataset (a dataset consisting of 60000 images with 10 classes of image; classes include horses and airplanes) to random label assignments on these images (as well as the correct labels).
- A DNN with 380 million parameters learned a function from the 14 million images in the Imagenet database [18] to their labels.

It should be very clear from these examples that DNNs are capable of representing very complex functions, even if they require enormous numbers of parameters. However, all we have seen so far is that some (presumably) very small part of parameter-space in some DNNs produces these correct functions. We will, in the next section, discuss how stochastic optimisers can find this part of parameter-space, and then discuss how well DNNs generalise,

7.2 Scalability and Training

How to find this small part of parameter space? Let us assume that we have a DNN $N_\theta : \mathcal{X} \rightarrow \mathcal{Y}$, which seeks to model a target function $f : \mathcal{X} \rightarrow \mathcal{Y}$ (by finding the right parameters θ). Typically we will be interested in the action of f on a subset of \mathcal{X} . We will assume for sake of simplicity that we have a discrete set of n inputs and their associated outputs, written as $\{(x^{(i)}, y^{(i)})\}_{i=0}^{i=n}$, where $y^{(i)} = f(x^{(i)})$, $x^{(i)} \in \mathcal{X}$ and $y^{(i)} \in \mathcal{Y}$. Consider an example where f is the map from images of handwritten digits, size 28×28 , to their labels, $0 - 9$. \mathcal{X} would be the space of images; \mathcal{Y} the space of labels and $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=0}^{i=n}$ would be n images with their appropriate labels. We want a general way of *quantifying* how close N_θ is to f on \mathcal{D} . To do this, we define a *loss function*:

$$l_\theta(\mathcal{D}) = \sum_{i=0}^n (N_\theta(\mathbf{x}^{(i)}) - \mathbf{y}^{(i)})^2 \quad (12)$$

where we denote the dependence on the parameters θ . This gives a measure of the accuracy of the network on some part of the input space. The example above is the *Mean-Square Error* (MSE) loss function, and it is clear that it is smaller when the network is most accurate. $l_\theta(\mathcal{D})$ is also sometimes called loss landscape, and points of low loss are called ‘minima.’ There are many other choices, the most popular being cross-entropy loss [15]:

$$l_\theta(\mathcal{D}) = - \sum_{i=0}^{i=n} p(x^{(i)}) \log(q(x^{(i)})), \quad (13)$$

where $p(x^{(i)})$ is the true probability distribution over $y^{(i)}$, and $q(x^{(i)})$ is the probability distribution the network outputs (this requires the final layer to be passed through a suitable nonlinearity).

Remark 7.1. *It is worth noting that while the loss function does track the performance of a network, if the function the DNN seeks to model is not continuous, then the loss function does not quite capture everything about the its performance. For example, if our network acted as a binary classifier by thresholding the outputs at 0, the $y^{(i)}$ ’s could be labelled -1 or 1 , and the loss function would be 0 if the network correctly mapped each $x^{(i)}$ to its respective label. However, for binary classification, usually the output is thresholded at 0 – so if the network outputs a negative number it is classified as -1 . otherwise 1 . This means if the network outputs a very small value it may register high loss*

but correctly classify the image.

So, how can we use this loss function to train a DNN? We now can consider $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^m$ to be the ‘training examples’. We need to find parameters which minimise l_θ on \mathcal{D} . To do this, we can use an algorithm called Stochastic Gradient Descent. Conceptually, the algorithm works by ‘randomly initialising’ network parameters, usually from i.i.d. Gaussian distributions, and taking small steps in the direction of steepest descent (‘downhill’) in the loss landscape until a ‘minima’ is reached – until the DNN accurately models the target function f .

Algorithm 1: Stochastic Gradient Descent

input: DNN N with parameters θ randomly initialised (from, for example i.i.d. Gaussian distributions), training data $\mathcal{D} = \{x^i, y^i\}$. a loss function, $l_\theta(x)$, a threshold value q , a ‘batch size,’ b , and a learning rate γ .

while $N_\theta(x^i) \neq y^i$ **do**

Randomly select sets (batches) of images from \mathcal{D} , of size b , so that there are $\lceil |\mathcal{D}|/b \rceil$.

for each batch of images $\mathcal{B} \subset \mathcal{D}$ **do**

Calculate $\nabla_\theta l(\mathcal{B})$

Update $\theta \rightarrow \theta - \gamma \nabla_\theta l(x^{(i)})$.

end for

end while

We provide a physical intuition for this process in Figure 2b. SGD with batch size $|\mathcal{D}|$ is called Gradient Descent (GD). It is inferior to SGD for a number of reasons – firstly, it can get trapped in local minima in the loss landscape – which SGD can avoid. Secondly, it is very slow (typically though, SGD with very small batches is also very slow; there is a dataset and architecture specific optimum batch size). There are many improvements to SGD – the Adam optimiser [53] is commonly used, and is a package of such improvements. For example, it adds a ‘momentum’ term (where a fraction of the gradient computed at the previous step is added to the current step). There are many other variants (e.g. Adagrad, RMSprop, Adedelta). These methods do work (see examples of the success of optimiser-trained DNNs in Section 1). However, computing the gradient of a function with respect to large matrices thousands of times is very computationally expensive. GPU based machines can be 10 times faster than CPU machines (for large DNNs), but even with this speed-up,

optimisation (the process of applying SGD until small training loss is achieved) can take many hours on supercomputers for sufficiently complex problems (e.g. AlphaZero took many hours to train).

Remark 7.2. *During the training process, an **epoch** is the duration of one iteration of the while loop in Section 7.2 (i.e. the time taken to train on all the images once). DNNs can take thousands of epochs to finish training. Also note that the condition in Section 7.2 is only one method for stopping training – there are more sophisticated methods, based on, for example, rate of change of loss function as a function of training iterations,*

Remark 7.3. *Experiments have shown that DNNs are expressive enough to learn randomly labelled datasets (so each $x^{(i)} \in \mathcal{D}$ is associated with a randomly selected label in \mathcal{Y}) [29] – so training as in Figure 2 could be done with random labels, and the optimiser would eventually find parameters which can map images to the randomised labels, whatever they are.*

DNNs scale better than other similar learning agents (e.g. Gaussian processes [54] in both time and space complexity), but training large networks still takes a long time. For example, some of the DNNs we trained for experiments in Part III took 20 hours to train to 100% accuracy on \mathcal{D} .

7.3 Generalisation

So, the third and final question to ask about DNNs: given that our network is expressive enough to model the correct function on some training data \mathcal{D} , and that we can find some parameters θ which produce this function, does this function generalise? In other words, does the optimisation process pick a function which is also accurate on unseen data? We will refer to such data as to be in the ‘test set,’ which we denote $\mathcal{E} = \{x^{(i)}, y^{(i)}\}$. We will also define the ‘generalisation error’ $\langle \epsilon_G \rangle$ as the expected fraction of errors (for discrete data⁸) for any test set \mathcal{E} .

The short answer: yes, and remarkably well, on tasks such as image classification and sentiment analysis (see Section 1 for further examples). However, this is not the full story. For complex tasks (for example, the CIFAR10 dataset), to obtain good performance, careful selection of the architecture is necessary. Equation (9) defines a so called ‘fully-connected feed-forward network’

⁸For continuous data, the loss function itself can be a measure of generalisation error

(FCN) because no entries in weight matrices are fixed at 0 (fully-connected) and information passes in one direction down the network (feed-forward). However, many architectures used today are neither feed-forward nor fully-connected.

For example, Convolutional Neural Networks (CNNs) [55] have a built in translational invariance, which makes them suitable for image classification (because a CNN will not be sensitive to the location of objects in the image). They are feed-forward but not fully-connected. ResNets [56] have skip connections (connections between layers not next to each other), and Recurrent Neural Networks (RNNs) [57] have connections in loops. One example of RNN is an LSTM (Long short-term memory) [58], which is particularly suitable for language models. Carefully selecting the type of network required (note that the architectures above can be combined; e.g. a Convolutional Layer can come before a fully connected layer and so on) is a typically a hard problem and other than educated guesses and rules of thumb like ‘CNNs are good for image classification’, there is no formula for determining the best network for any problem. Furthermore, evidence suggests that the choice of optimiser can also have an effect on generalisation. If we think about this in terms of inductive biases, this indicates that (1) the inductive biases of DNNs can be tuned by changing the architecture, and (2) allowing for architecture changes, the inductive bias is suitable for a wide range of tasks (ranging from image classification to chess playing). We will devote the next section to understanding why this might be.

Remark 7.4 (Tasks where DNNs do not generalise). *DNNs cannot learn identity functions. No current DNN-based algorithm (to the knowledge of the author) can reliably count the number of instances of an object in an image, even if they can recognise that the object is in the image. Standard architectures (e.g. FCNs and basic CNNs) do not generalise well on complex datasets such as CIFAR100 [59] (a dataset of images in 100 different classes), and require more complex architectures for good generalisation. Reinforcement learning systems (for example. AlphaZero) typically take a lot of training time to achieve good generalisation (even comparatively simple games like Breakout can take hours and hours of training [60]).*



(a) A cartoon depicting the bias variance tradeoff.



(b) An example of a polynomial fitter (polyfit) and different DNNs fitting functions to 10 datapoints

Figure 3: (a) demonstrates the tradeoff between expressivity and generalisation, for a parameterised learning agent without a good inductive bias. Making the agent less expressive (for example, by reducing the number of parameters) will prevent overfitting, as the agent cannot fit the noise, but not so much that it cannot represent the ‘true’ function, otherwise you ‘underfit’. This is called the ‘bias variance tradeoff.’ (b) demonstrates neatly this tradeoff in action, and is an example of how DNNs appear to avoid it. It shows 10 datapoints, and the behaviour of various fitting algorithms, including a 5th order polynomial fitter (which underfits), a 20th order polynomial fitter (which overfits), and a Fully Connected DNN, each with over 1000 parameters, which both fit the data very accurately and do not overfit in the same way that the 20th order polynomial fitter does.

7.4 Summary and further details

Remark 7.5 (Classical Learning Theory). *It is a well-known result from classical learning theory that a learning agent that is highly expressive will overfit if the agent is trained to 100% accuracy on the training data \mathcal{D} , without an inductive bias (e.g. a form of regularisation like penalising large coefficients in a polynomial fitter). For more information about these problems, see Figure 3. We know that DNNs are highly expressive (if sufficiently large). A famous result [29] shows that DNNs are capable of memorising training sets consisting of images in the CIFAR10 image classification dataset, with completely randomised labels (see Section 7.1 for other examples). Given that they do not overfit, even when trained to 100% accuracy [39] and highly expressive, they must have a good inductive bias.*

So we have summarised the key features of DNNs. We have seen that they are highly expressive yet generalise well, and that this means we need a good inductive bias (see Definition 1.1). Recall from Part I, that: ‘an ideal learning agent would approximate Solomonoff induction – out of the functions consistent with training data \mathcal{D} , it would pick the simpler with high probability.’ With this in mind, we will now apply the Levin Bound to the parameter-function map of DNNs.

8 Applications of the Levin Bound to DNNs

We will copy the bound (with slightly different notation for ease of use in this section) below:

$$p(f|n, M) \leq 2^{-K(f|n) + \mathcal{O}(1)}. \quad (14)$$

Recall that this was proved for a computable map M with a fixed input size n , and that this version required the map to be simple with respect to f (see Equation (8)). In this notation, the output of M is f . One example of such a map is the parameter function map [1, 4].

Definition 8.1 (Parameter-function map). *Consider a parameterised supervised model, and let the input space be \mathcal{X} and the output space be \mathcal{Y} . The space of functions the model can express is $\mathcal{F} \subset \mathcal{Y}^{|\mathcal{X}|}$. If the model has some number of real valued parameters, taking values within a set*

$\Theta \subseteq \mathcal{R}^p$, the parameter function map M is defined by

$$\begin{aligned} M : \Theta &\rightarrow \mathcal{F} \\ \theta &\mapsto f_\theta \end{aligned}$$

where f_θ is the function corresponding to parameters θ .

In essence this map is simply the map from the parameters of a DNN to the function that DNN expresses on data \mathcal{X} . The function space \mathcal{F} of a network N could (in general) be considered to be the entire space of functions that N can express on the input vector space \mathcal{X} , but it could also be taken to be the set of partial functions N can express on some subset of \mathcal{X} . For example, we can restrict this subset to be MNIST images. In this case, the parameter-function map would be the map from network parameters to the corresponding classification of MNIST images. See Figure 4a).

We will now apply the Levin bound to the parameter-function map. It is a computable map of fixed input size n : the number of parameters in the network (note that this is different from the input of the network). The outputs (inputs) we will enumerate will be the function the network expresses (on some input data). Thus, in our notation, M is the parameter function map with n parameters, and f would be the output function expressed on the dataset. Then, according to the Levin bound, if the parameter-function map of the DNN is biased, then the DNN will be much more likely to express simple functions, if parameters are randomly sampled.

Remark 8.2 (Randomly sampling parameters). *We will typically measure a bias in the parameter function map by randomly sampling parameters from i.i.d. truncated Gaussians. It is demonstrated in [1, 4] that the sensitivity to this initial distribution is small for ReLU activated DNNs, although in [4] a stronger dependence on the distribution for tanh activated DNNs is observed. It is also shown that the bias can be reduced by increasing the number of layers in tanh activated DNNs – in the limit of infinite depth, this bias can be destroyed [61]. For the remainder of this dissertation, when we make statements such as ‘the parameter-function map is biased’, we will mean that upon randomly sampling parameters of a DNN from i.i.d. Gaussian distributions, certain functions are expressed more often than others.*

Remark 8.3 (Discrete map). *It is worth mentioning that the Levin bound (even in the most general*

case, Equation (3)) is valid only for discrete maps [1, 37]. DNNs are continuous (in the sense that parameters θ can take continuous values in \mathbb{R}). However, in practice network parameters are sampled from truncated Gaussian distributions, with finite precision (as are all numbers in computers). One further (although unsatisfying) justification is that it appears to work well.

As previously mentioned in Section 8, there are currently no general rigorous methods for determining that the upper bound is tight for outputs over a wide range of complexities (necessary to imply simplicity bias). The bound sketched in Theorem 5.2 guarantees that when sampling inputs, the probability mass of outputs is concentrated near the bound. However, this alone doesn't imply simplicity bias, as that condition is satisfied by maps producing an uniform output distribution (where the probability mass simply concentrates in the maximum complexity region). On the other hand, empirical evidence on simple systems such as neural networks, does suggest that the bound is relatively tight for many systems [2]. This empirical evidence suggests that the parameter-function map of DNNs is biased towards simple functions. Because DNNs are initialised by randomly sampling parameters before training (typically from an i.i.d. Gaussian), this would be a **bias towards simple functions at initialisation of network parameters**.

Further evidence that the parameter-function maps of DNNs are biased comes from studying arguably the simplest neural network, the perceptron, which have no hidden layers. They are essentially a linear decision boundary, and, as we mentioned earlier, generalise well and are scalable, but are not sufficiently expressive to represent complex functions [14]. A perceptron N performing binary classification maps an input x to an output y as follows

$$y = \mathbb{1}(\mathbf{w} \cdot \mathbf{x} + b), \quad (15)$$

where $\mathbb{1}(X)$ is an indicator function (1 if $X > 0$, otherwise 0), \mathbf{x} is the input vector, \mathbf{w} is the 'weight' (vector), and b is the bias term. In [4], one of the main results concerned an n dimensional perceptron (a perceptron with input size n) with $b = 0$ and weights sampled from a distribution symmetric about coordinate planes (see Theorem 8.4). Consider the set of functions $\{0, 1\}^n \rightarrow \{0, 1\}$, and a perceptron with input size n with inputs from $\{0, 1\}^n$. Denote the number of inputs from $\{0, 1\}^n$ which are mapped to 1 by the perceptron for a particular choice of w as T .

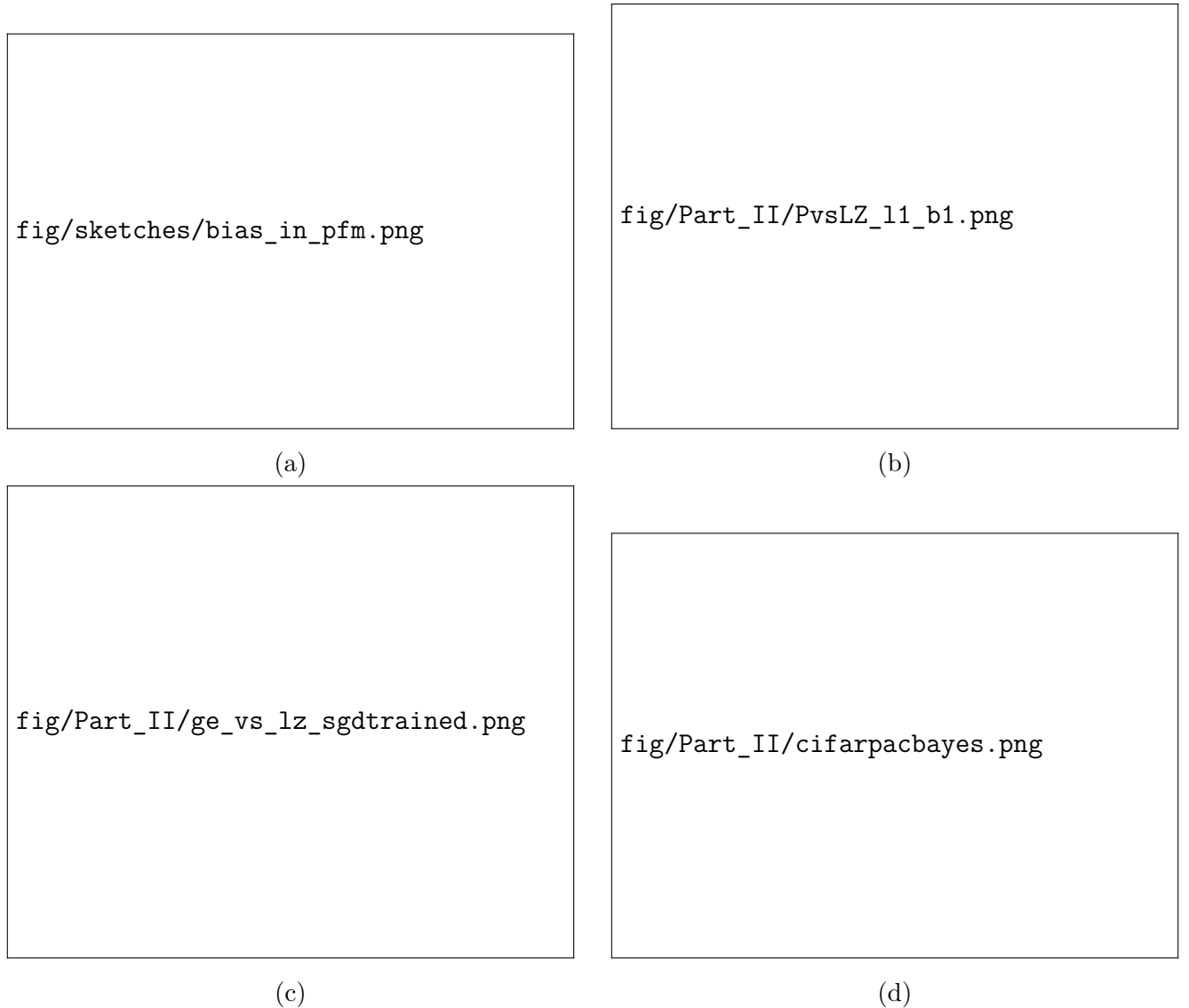


Figure 4: (a) A cartoon meant to show Definition 8.1, and why this means that, upon randomly sampling parameters, DNNs are more likely to express simple functions. (b) From [1]. It shows the probability $p(f)$ vs $K_{LZ}(f)$, an approximation to the Kolmogorov Complexity of f (the LZ complexity of the output function encoded as a string), for a 2-hidden layer (widths 40), ReLU activated DNN modelling functions $\{0, 1\}^7 \rightarrow \{0, 1\}$ (acting as a binary classifier). Consider an unbiased learning agent, where every function appeared with uniform probability. Then each function would have $p = 2^{-128} \approx 10^{-29}$ – contrast with (b), where some functions have $p > 0.08$. (c) From [1], for the same system as in (b). It shows the generalisation error achieved after training the DNN with SGD with a training set of half the inputs in $\{0, 1\}^7$ as a function of K_{LZ} of the target function. (d) From [1]. An example of the PAC-Bayes bound (see Section 9 for details) applied to various datasets and architectures. The PAC-Bayes bound upper bounds the generalisation error for a system (with some probability) assuming that the DNN was trained by randomly sampling parameters (see Section 9 for details). The Gaussian Processes approximation (see Appendix C) is used to approximate this training method. Clearly fairly tight bounds are possible without taking into account the effects of the optimiser.

Theorem 8.4 (Perceptrons are biased towards low entropy functions). *Let P be a probability measure on \mathbb{R}^n , which is symmetric under reflections along the coordinate axes, so that $P(x) = P(Rx)$, where R is a reflection matrix (a diagonal matrix with elements in $\{-1, 1\}$). Let the weights of a perceptron without bias, w , be distributed according to P . Then $P(T = t)$ is the uniform measure. Statement and proof sketch from [4].*

Proof sketch. We consider the sampling of the normal vector w as a two-step process: we first sample the absolute values of the elements, giving us a vector w_{pos} with positive elements⁹, and then we sample the signs of the elements. Our assumption on the probability distribution implies that each of the 2^n sign assignments is equally probable, each happening with a probability 2^{-n} . The key of the proof is to show that for any w_{pos} , each of the sign assignments gives a distinct value of T (and because there are 2^n possible sign assignments, for any value of T , there is exactly one sign assignment resulting in a normal vector with that value of T). This implies that, provided all sign assignments of any w_{pos} are equally likely, the distribution on T is uniform. \square

This proof can be used to create an analytic and computable bound on the Boolean Circuit complexity (see Appendix B of [4] for details), of a form that decreases exponentially in complexity (qualitatively similar to the Levin Bound).

In summary, we have discussed analytic results and empirical evidence in [1, 4] which suggests that small DNNs are exponentially more likely to express simple functions, upon *randomly sampling network parameters* θ . Inspired by the Levin Bound Equation (8), in [1], empirical evidence was presented which showed that the parameter-function map of simple DNNs was strongly biased towards simple functions. An analytic proof from [4] was also provided, which proved that perceptrons (0-hidden layer DNNs) are biased towards *low entropy* functions, which can be used to make a qualitatively similar bound to the Levin Bound. In Part I we suggested that an ideal learning agent would approximate Solomonoff induction – it would assign high probability to simple functions. We suggested that simple functions would generalise well because ‘learning’ is commonly finding a simple pattern describing apparently complex observations. The next question to ask is

⁹almost surely, assuming 0 has zero probability measure

how can we construct a learning agent biased towards simple functions using DNNs?

9 PAC Bayes theories

We will construct this learning agent to be *Bayesian*.¹⁰ In the following section, we will first describe a Bayesian learning agent, and then explain how a DNN can be trained in a Bayesian fashion. We refer the reader back to Definition 1.1 for a definition of learning.

We begin with a *prior* over functions, $P(f)$. If our ‘observation’ – the training set \mathcal{D} – corresponds to the exact values of the ‘true’ function Q which we wish to infer (i.e. no noise), at some points, then we need to use a 0-1 likelihood $P(D|f)$, which just indicates whether the data is consistent with the function. Note that the symbol D denotes the event that the DNN is 100% accurate on the training data (i.e. consistent with) \mathcal{D} . Formally, if $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^m$ corresponds to the set of training pairs:

$$P(D|f) = \begin{cases} 1 & \text{if } \forall i, f(x^{(i)}) = y^{(i)} \\ 0 & \text{otherwise} . \end{cases}$$

Note that this quantity is technically $P(D|f, \{x^{(i)}\})$, but we denote it as $P(D|f)$ to simplify notation. We will use a similar convention throughout, whereby the input points are (implicitly) conditioned over. Bayesian inference then corresponds to assigning probabilities to candidate functions according to Bayes rule

$$P(f|D) = \frac{P(D|f)P(f)}{P(D)}, \quad (16)$$

to obtain the *Bayesian posterior*. $P(D)$ is also called the *marginal likelihood*, and it is the *total probability of all functions compatible with the training set*. This step is the ‘learning’ part of the algorithm: the function f with the highest $P(f|D)$ is typically taken to be the ‘best guess’ for the true function Q .

For an example of a Bayesian learning agent, see Section 4. We can train DNNs in a Bayesian fashion by following Algorithm 2: $P(f)$ and $P(D)$ can be calculated by randomly sampling parameters,

¹⁰Bayesian learning agents are considered to be ideal in a sense illustrated by the *Dutch Book arguments*. They can be used to show that a non-Bayesian learning agent can be duped into making sub-optimal decisions (in the argument, there is always a wager that such a learning agent will accept that will cause it to lose money).

allowing us to generate $P(f|D)$. The Levin bound predicts that *simple f will have higher $P(f)$, and thus higher $P(f|D)$ – which is what we want!*. This is an example of a ‘Bayesian supervised learning agent’. However, ideally we would be able to link this to generalisation error – the probability of incorrectly predicting an unseen datapoint.

For such an algorithm with training set \mathcal{D} with m examples, the PAC-Bayes theorem [1, 62], roughly states that the generalisation error $\langle \epsilon_G \rangle$ is bounded, with probability $1 - \delta$

$$\langle \epsilon_G \rangle \lesssim \frac{-\log P(D) - \log(\delta)}{m}.$$

In [1], the authors applied the bound to some deep learning models. There, $P(D)$ and $P(f)$ were calculated by approximating the process of randomly sampling parameters with Gaussian processes (as opposed to involving the Levin Bound). We will provide an in depth treatment of Gaussian Processes in Appendix C; and in Part III. It was found that using this technique gave relatively tight predictions for optimiser-trained DNNs – see Figure 4d. In other words, such a learning agent would give good generalisation.

Remark 9.1 (Link between good generalisation and simplicity). *While simple systems have been shown to be strongly biased towards simple functions, it is unclear that this extends to larger DNNs and more complex datasets (even though the Levin Bound tentatively suggests that it may do so). However, in [1], estimates of $P(f)$ for functions f on the CIFAR10 dataset were calculated with the Gaussian Processes approximation, and functions that had better generalisation had higher $P(f)$. Only later was this linked to complexity via a complexity measure called CSR, and more CSR-complex functions did indeed have lower $P(f)$ (more on this in Part III). So, experiments regarding the PAC-Bayes bound do not rely on the Levin bound in their calculations.*

In summary, the argument from the previous two sections is as follows: simple functions have high probabilities in the prior distribution ($P(f)$) generated by randomly sampling parameters, which leads to simple functions having high probability in the posterior $P(f|D)$, and thus low generalisation error (quantified with PAC-Bayes bound), for DNNs trained with Algorithm 2 (randomly sampling parameters).

Algorithm 2: Training a DNN by randomly sampling parameters

input: DNN N , prior over parameters $P_{\text{init}}(\theta)$, training data \mathcal{D} , test data \mathcal{E} and large $n \in \mathbb{N}$.
 $F \leftarrow \emptyset$
for $i \in \mathbb{N}, i < n$ **do**
 Sample network parameters $P_{\text{init}}(\theta)$
 If the DNN with these parameters expresses a function consistent with the training data \mathcal{D} ,
 save the function generated by these parameters on \mathcal{E} to F .
end for
Calculate the distribution $P(f|D)$

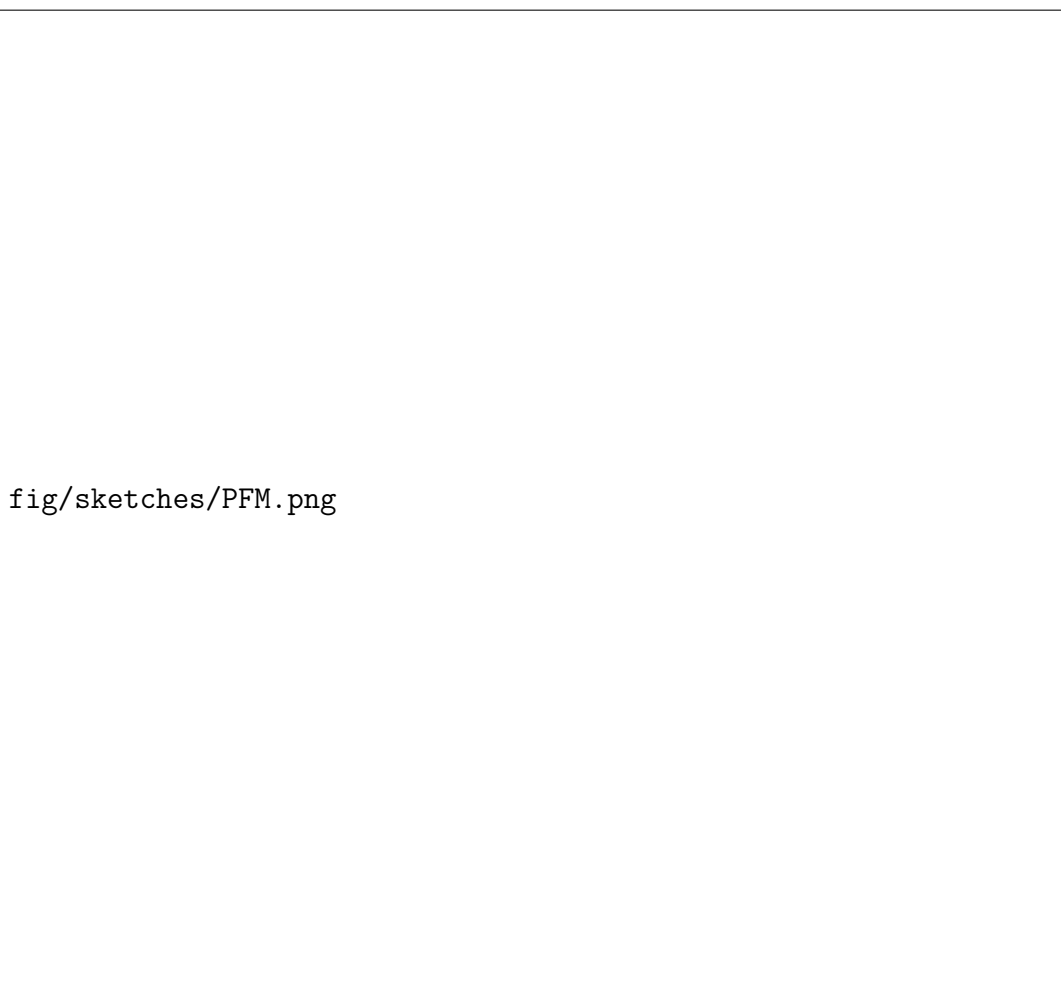


Figure 5: Pictorial summary of argument given in Section 9. Simple functions have more associated parameters, so upon randomly sampling parameters, and removing functions inconsistent with training data \mathcal{D} , the DNN is more likely to express a simple function. This is equivalent to a ‘bias at initialisation,’ however we do not know whether the optimiser will find functions with similar probabilities due to its interaction with the loss landscape.

10 Conclusion for Part II

In this part, we first described in detail how a Deep Neural Network works. We discussed the three key parts of the learning algorithm – its expressivity (DNNs are highly expressive), its scalability (DNNs scale well) and its generalisation (DNNs generalise well for a wide variety of tasks, with the appropriate architecture). We then briefly discussed how, given the high expressivity of a DNN, they must have a good inductive prior in order to generalise. In other words, as they can express many incorrect functions, they need a mechanism that chooses correct functions with high probability.

We then applied the Levin Bound (see Equation (8)) to the *parameter-function map of DNNs*, and presented evidence that it appears to be tight on small DNNs – equivalently, simple functions are more likely to be found upon randomly sampling network parameters. We then applied PAC-Bayes bounds to DNNs trained by randomly sampling parameters (see Algorithm 2). It was shown empirically that the bias in the parameter-function map is sufficient for good generalisation on datasets like CIFAR10 (see Figure 4d). However, is this sufficient to guarantee good generalisation?

Not quite. The Levin bound and PAC-Bayes bound are applicable to DNNs trained with Algorithm 2, but DNNs are in practice not trained with Algorithm 2 because due to computational complexity – (although other learning agents like Gaussian Processes approximate this training algorithm). They are instead trained with variants of Algorithm 1 (SGD). However, we do not know whether Algorithm 1 and Algorithm 2 find functions with the same probabilities. If they do, then the bounds are applicable (although not provably so); if not then the arguments are not applicable. In the next part we provide empirical evidence suggesting that the probabilities are very similar.

Part III

Is SGD a Bayesian Sampler?

In Part I we proved that the following bound holds, for certain classes of input-output maps:

$$P(f) \leq 2^{-K(f)+\mathcal{O}(1)}. \quad (17)$$

The authors of [1] applied a computable approximation of this bound to the parameter-function map of simple DNNs. Note that the Levin Bound does not guarantee that simple functions have high probabilities (e.g. if the $\mathcal{O}(1)$ terms are very large), however the empirical evidence on these simple DNNs suggests that the parameter-function map is indeed *strongly* biased towards simple functions (see Figure 4b), which are by arguments in Part I expected to generalise better than complex functions. Inspired by this, it was further argued with the help of PAC-Bayes bounds that an observed ‘bias in the parameter function map’ towards functions with good generalisation, for more complex datasets like CIFAR10, can be sufficient to guarantee low generalisation error, $\langle \epsilon_G \rangle$ (see Figure 4d). This is all applicable in the case where DNNs are trained with Algorithm 2: randomly sampling parameters until the DNN is consistent on some training set \mathcal{D} . This bound has been successfully applied to real world datasets like CIFAR10. This implied that bias in the parameter-function map (or, informally, the architecture) is the main reason for good generalisation in DNNs.

So we now turn to (arguably) the main problem that this dissertation addresses. As mentioned in Part II, there is a big caveat: DNNs are not trained by randomly sampling parameters until consistent with a dataset \mathcal{D} , as described in Algorithm 2, but are instead trained by an optimiser until consistent with a dataset \mathcal{D} , as described in Algorithm 1. If however, Algorithm 1 samples parameter space similarly to Algorithm 2, then the arguments in Part II would explain why DNNs generalise well.¹¹

¹¹If not, then this is arguably even more interesting, because it would mean that the each method achieves good generalisation for completely different reasons.

It is first worth noting that there is much circumstantial evidence for this hypothesis. Recent work on kernel methods has shown that Gaussian Process based learning agents (see Appendix C for details) exhibit extremely similar generalisation to optimiser-trained wide-layer DNNs [63]. This is interesting because the behaviour of a trained Gaussian Process are equivalent to those obtained by training by randomly sampling from a DNN (in the limit of infinite width)¹². However, generalisation error is an extremely coarse-grained measure of how a learning agent works, and to make the two systems (DNN and Gaussian Process) equivalent also takes a lot of hyperparameter tuning. A finer grained perspective is required in order to understand properly how different optimiser-trained DNNs are to DNNs trained by randomly sampling parameters. To do this, we will first distinguish between two related (but arguably distinct) questions:

First-order generalisation: Why do DNNs generalise at all when highly expressive?

Second-order generalisation: How and why do hyperparameter changes to the optimisation algorithm affect generalisation?

To provide even a partial answer to both of these questions, we will need to study the interaction of the optimiser with the parameter function map. We will do this in a more fine-grained fashion than was done in [63] – where just generalisation error was used – by looking at the *function-space* of both systems. The datasets we will use are all binary classification datasets, so we will be studying the functions the DNN finds from the input data to $\{0, 1\}$. This means that studying the function space is equivalent to *coarse-graining the probability masses in the output space by quadrant*.

11 Explanation of our hypothesis

In this section we will provide the definitions necessary to ask the question: does the optimiser find functions similar to those that would be found were the DNN trained by randomly sampling parameters (as described in Section 9). In other words, is **first-order generalisation** primarily due to the bias in the parameter function map? We first explicitly write down $P(f)$ – the probability that the network expresses f on a subset \mathcal{D} of an input space \mathcal{X} upon random sampling of parameters

¹²We will use Gaussian Processes in our experiments to approximate randomly sampling with DNNs

(i.e. the same $p(f)$ as found in Equation (17)) below:

$$V(f) := P(f) = \int \mathbb{1}[\mathcal{M}(\theta) = f] P(\theta) d\theta, \quad (18)$$

where $\mathbb{1}[X = x]$ is 1 if $X = x$, otherwise 0. We note that this is equivalent to defining a ‘volume’ $V(f)$ of f – the volume in parameter-space which produces f , with Gaussian measure. We will discuss the advantages of thinking about simpler functions having higher ‘volumes’ in parameter space at the end of this section.

We now consider our main question: how much does the bias in the parameter-function map affect the inductive bias of DNNs? DNNs are typically trained on some training examples, $\mathcal{D} \subset \mathcal{X}$, to high accuracy. We will denote the Bayesian posterior distribution, conditioning on the event D – that f has $1 - \epsilon$ training accuracy on some training set \mathcal{D} – to be the following:

$$V_{Bayes}(f|D) := P_{Bayes}(f|D) = \frac{P(f \cap D)}{P(D)}. \quad (19)$$

This corresponds to the Bayesian posterior using $P(f)$ as the prior, and a 0-1 likelihood. This is the distribution Algorithm 2 would produce. The expression to which we want to compare $P_{Bayes}(f|D)$ is the the posterior distribution of the optimiser, also trained on data \mathcal{D} to $1 - \epsilon$ accuracy:

$$P_{opt}(f|D) = \int \mathbb{1}[\mathcal{M}(\theta_t) = f] P(\theta_t | \theta_i, \text{opt}, D) \times \tilde{P}(\theta_i) d\theta_i d\theta_t \quad (20)$$

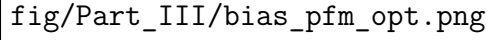
where $P(\theta_t | \theta_i, \text{opt}, D)$ denotes the probability that after training from initial weights θ_i with an optimiser ‘opt’ with training data set D , the model has weights θ_t (t for trained). We will assume that the optimiser always converges for a normalised probability. $\tilde{P}(\theta_i)$ is the initialisation distribution (unlike $P(\theta)$ in Equation (18), see comment below). $P_{opt}(f)$ is a measure of the volume of f ’s ‘basin of attraction’ in parameter space (with Gaussian measure).

We have argued in Part II that, assuming ‘simple’ functions (functions with low Kolmogorov Complexity) are associated with exponentially higher ‘volumes’ than more complex functions (i.e. have large $P_{Bayes}(f|D)$), the optimiser will be overwhelmingly likely to find these ‘simple’ functions (so

these functions would also have large $P_{\text{opt}}(f|D)$), which would explain the good generalisation. Were the bias in the parameter-function map sufficiently strong, properties of the loss landscape would impact on the optimiser to a much smaller extent than the bias, and the posterior distribution due to the optimiser (still conditioning on D) would be similar to that in Section 9 – so **first-order generalisation** would be due to the fact that functions with good generalisation have high $P_{\text{Bayes}}(f|D)$, and that $P_{\text{Bayes}}(f|D) \approx P_{\text{opt}}(f|D)$, and **second-order generalisation** would be due to small deviations from this approximate equality caused by the optimiser.

Let us provide an intuition for why one might expect this to be true. First of all, given that SGD (and similar optimisers) move a bounded number of steps, each of which are of a bounded size, we are interested in a quantification of the number of parameters which produce some function f within a *bounded* region. Because the Gaussian distribution falls off fast, it seems plausible that defining a volume with a Gaussian measure would be a good estimate for such a quantification. Evidence in Section 13 also suggests that the distance moved during training is not large. It also seems plausible that functions with large ‘volumes’ should have a correspondingly higher chance of being found by SGD – although this is a big assumption, and the focus of this section is testing it. Moreover, if a function f has a large ‘volume’ then this means that it can be compressed (relative to the full description of all the parameter assignments). This might lead one to expect that functions with large ‘volume’ should be simpler (this is similar to a suggestion made [32]). If these intuitions are accurate then this could explain why DNNs have an inductive bias in favour of simple functions.

For us to sensibly compare $P_{\text{Bayes}}(f|D)$ and $P_{\text{opt}}(f|D)$, $P(\theta)$ in Equation (18) should be a distribution with support concentrated on the region of parameter space which SGD would typically end up in *after training*, and be otherwise uninformative (flat) within this region (hence the intuition of $P(f)$ as a ‘volume’). Note that by this criterion $P(\theta)$ could be quite different from the initialisation distribution $\tilde{P}(\theta_i)$. We find that while $P(\theta) \approx \tilde{P}(\theta)$ often gives a good approximation, this is not always the optimal choice. For example, when initialising the biases to 0 for training, we found that setting the variance of the bias term in $P(\theta)$ to a small constant (usually around $0.1 \times$ the variance of the weights) appears to give the the greatest correlation between $P_{\text{Bayes}}(f|D)$ and $P_{\text{opt}}(f|D)$.



fig/Part_III/bias_pfm_opt.png

Figure 6: A visualisation of our explanation for why $P_{Bayes}(f|D) \approx P_{opt}(f|D)$: the ‘basins of attraction of the optimiser of f ’, the region of parameters from which the optimiser will find f (weighted by Gaussian measure to reflect probability of initialising in that region of parameter space) correlate with the ‘volume’ of the functions, $P_{Bayes}(f|D)$, which we approximate with a Gaussian measure.

12 Methodology

In this section we will explain three experiments we use. We will provide empirical evidence which suggests that the majority of the inductive bias of DNNs comes from their parameter-function map. To do this we will need some way of disentangling the effects of the parameter-function map from the effects of the optimiser. Specifically, we want to know:

1. Is probability that an optimiser finds parameters that produce some function f determined primarily by the parameter ‘volume’ of f ? Put symbolically, this is equivalent to asking whether **first-order generalisation** is due to $P_{Bayes}(f|D) \approx P_{opt}(f|D)$
2. Do functions with low $\langle \epsilon_G \rangle$ (generalisation error) have larger $P_{Bayes}(f|D)$?

We will present a method for answering these questions, and provide extensive empirical evidence that the answer to (1) is close, but in general the measures correlate log-linearly, i.e. $\log(P_{Bayes}(f|D)) \propto \log(P_{opt}(f|D))$. For typical architectures though, $P_{Bayes}(f|D) \approx P_{SGD}(f|D)$ can be a good approximation. The answer to (2) is in the affirmative for systems that are known to exhibit good generalisation (e.g. MNIST, FashionMNIST and the IMDB movie review dataset). Changing hyperparameters in these experiments will allow us to answer questions pertaining to **second-order generalisation**.

12.1 Experiments 1, 2 and 3.

Our first experiment (see Experiment 1) aims to test whether $P_{Bayes}(f|D) \approx P_{opt}(f|D)$. D was defined in Section 12 to be the condition that the accuracy on the training set \mathcal{D} is $1-\epsilon$ (with $\epsilon = 0$ for ease of calculation). However, because we cannot calculate $P_{Bayes}(f|D)$ for DNNs analytically, nor obtain it by sampling, we will have to approximate it using Gaussian Processes (GP) approximation.

The Gaussian Processes approximation has been proven to be accurate for DNNs in the limit of infinite width [64, 65], and has been empirically shown to be mostly accurate for DNNs of lower widths (but still wide relative to the input; see [1]). We refer the reader to Appendix C for a more

detailed explanation of Gaussian Processes, and Appendix C.4 for details on how the approximation is calculated with the different loss functions. Briefly, for DNNs with mean-square error (MSE) loss functions, no further approximation is required; and $P_{Bayes}(f|D)$ can be calculated by sampling from the exact GP posterior distribution. For DNNs with cross-entropy (C-E) loss, the posterior does not have an analytic form, and a further approximation, the EP approximation is required. Once we approximate the posterior, we can either sample from it or use a further approximation to estimate the numerator and denominator in Equation (19). We find that both these methods give near-identical results in the appropriate range (see Appendix C.4). We refer to the two methods with the EP approximation as the GP/EP approximation. One major issue is that the EP approximation introduces uncontrolled errors – we will discuss these later in the section.

Experiment 1

input: DNN \mathcal{N} , training data \mathcal{D} , test data \mathcal{E} .
 $F \leftarrow \langle \rangle$ {the ‘functions’ found during training}
do n **times:**
 re-initialise the weights of \mathcal{N} from an i.i.d. Gaussian distribution
 train \mathcal{N} on \mathcal{D} until it reaches 100 % training accuracy
 record the classification of \mathcal{N} on \mathcal{E} and save it to F
 $A \leftarrow \emptyset$ {the frequency and ‘volume’ of each ‘function’}
for each distinct $f \in F$ **do**
 calculate the probability $P_{\text{opt}}(f|D) = \rho_f/n$ of f in F
 use GP or GP/EP approximation to estimate $P_{Bayes}(f|D)$ of f
 save $\langle P_{\text{opt}}(f|D), P_{Bayes}(f|D) \rangle$ to A
end for
return A

A similar method can be used to test if the parameter-function map of DNNs is biased towards functions with low generalisation error, $\langle \epsilon_G \rangle$. Specifically, given a training dataset \mathcal{D} and test dataset \mathcal{E} we can generate a random sample of different partial functions with varying levels of error on \mathcal{E} (by taking the true test set classification and corrupting some percentage of labels). We can then use the GP/EP approximation to estimate $P_{Bayes}(f|D)$. If it were found that functions with larger $P_{Bayes}(f|D)$ tend to have low generalisation error (and vice versa) then this would corroborate

the hypothesis that the parameter-function map of DNNs is biased towards ‘good’ functions. This method is described in Experiment 2.

Experiment 2

input: DNN \mathcal{N} , training data \mathcal{D} , test data \mathcal{E} .
for $\epsilon \in \{0.0, 0.5, \dots 1.0\}$ **do**
 $V_\epsilon \leftarrow \langle \rangle$
 generate classification c with error ϵ on \mathcal{E} (by randomly choosing $|\mathcal{E}| \times \epsilon$ distinct labels in the correct function (restricted to the test set) to switch to incorrect).
 use GP/EP to estimate the $P_{Bayes}(f|D)$ of c
 save the relative volume $P_{Bayes}(f|D)$ of f to V_ϵ
end for
return $V_{0.0} \dots V_{1.0}$

Note that some of the classifications c generated during the run of the algorithm will have very small $P_{Bayes}(f|D)$. This means that it would be impractical to estimate them by sampling the posterior. This experiment will therefore only be performed with the GP/EP approximation because it allows direct calculation of estimated $P_{Bayes}(f|D)$.

Experiment 3

input: DNN \mathcal{N} , training data \mathcal{D} , test data \mathcal{E} .
 $F \leftarrow \emptyset$ {functions sampled from the GP or GP/EP posterior}
do n **times:**
 sample a function f from the GP or GP/EP posterior when conditioning on \mathcal{D}
 calculate the error ϵ of f on \mathcal{E}
 save $\langle f, \epsilon \rangle$ to F
 $\mathfrak{R} \leftarrow \emptyset$ {probabilities and errors}
for each distinct $f \in F$ **do**
 let ρ_f be the frequency of f in F
 calculate $P_{Bayes}(f|D) = \rho_f/n$
 save $\langle P_{Bayes}(f|D), \epsilon \rangle$ to \mathfrak{R}
end for
return \mathfrak{R}

While the experiment given by Experiment 2 will be informative for how the space is biased, it does

not guarantee that all high-probability functions will be found (and is also not entirely reliable due to the EP approximation). Since these functions affect generalisation the most, we perform a third type of experiment to ensure that there are no high-probability functions that Experiment 2 misses. We do this by sampling the posterior directly, and this can be done with either the GP or GP/EP approximations. This experiment is described in Experiment 3.

12.2 Datasets & Architectures

We will define a ‘standard implementation’ of Experiment 1 to have the ‘standard parameters’ in Definition 12.1, so we can refer back to them easily and not have to list all the parameters each time we implement Experiment 1 in the following section. Where relevant, these parameters will also be considered the ‘standard parameters’ for Experiments 2 and 3.

Definition 12.1 (Standard implementation of Experiment 1). *We perform Experiment 1 and take 10^5 samples, with the following parameters: Dataset – binarised MNIST. Training set size – 10000 images, Test set size – 100 images, Number of hidden layers – 2, Width of hidden layers – 1024, Optimiser – Adam, Batch size – 128, Loss function – Cross-entropy (C-E). Parameters relevant to Experiment 2 and 3 will be considered the standard parameters for those experiments.*

We also define the binarisation of common datasets below.

Definition 12.2 (Binarised MNIST). *The MNIST database¹³ with even numbers classified as 0 and odd numbers as 1.*

Definition 12.3 (Binarised Fashion MNIST). *The FashionMNIST database¹⁴ with T-shirts, coats, pullovers, shirts and bags classified as 0 and trousers, dresses, sandals, trainers and ankle boots classified as 1.*

We use the IMDb movie review dataset, in a preprocessed form found here¹⁵. We also use four custom architectures, which we detail below.

¹³<http://yann.lecun.com/exdb/mnist/>

¹⁴<https://github.com/zalandoresearch/fashion-mnist>

¹⁵<https://www.kaggle.com/nilanml/imdb-review-deep-model-94-89-accuracy>

CNN: Layer 1: Convolutional Layer with 32 features size 3×3 . Layer 2: Flatten. Layer 3: FCN with 1024 neurons, Layer 4: FCN with 1 neuron.

CNN+Pool: Layer 1: Convolutional Layer with 32 features size 3×3 . Layer 2: Max Pool 2×2 . Layer 3: Flatten. Layer 4: FCN with 1024 neurons, Layer 5: FCN with 1 neuron.

CNN+Pool+BatchNorm: Layer 1: Convolutional Layer with 32 features size 3×3 . Layer 2: Max Pool 2×2 . Layer 3: Batch Norm. Layer 4: Flatten. Layer 5: FCN with 1024 neurons. Layer 5: Batch Norm. Layer 6: FCN with 1 neuron.

LSTM architecture: Layer 1: Embedding layer. Layer 2: LSTM, 256 outputs. Layer 3: Fully-Connected, 512 outputs. Layer 4: Fully-Connected, 1 output.

Remark 12.4 (Finite size effects). *For all instances of Experiments 1 and 3, approximations to discrete probability distributions $P_{\text{Bayes}}(f|D)$ and $P_{\text{opt}}(f|D)$ will be obtained by sampling, typically between 10^5 and 10^7 times. It is not possible to determine the probability of functions with a frequency of 1 with any kind of accuracy, and just to be safe we cut off functions with frequencies < 10 from all our graphs. Those functions are included in calculations involving generalisation errors, but are not included in calculations involving the number of functions found in Distribution X that were not found in Distribution Y in so many samples. Finally, note that all error bars in implementations of Experiment 2 are 2 standard deviations of the relevant sample.*

13 1st Order Generalisation

In this section we will present experiments on a wide range of datasets and architectures which we hope will convince the reader that the answers to the problems laid out in Section 12 are consistent with our main hypothesis.

Our first demonstration of Experiments 1, 2 and 3 can be found in Figure 7, where we present examples of the experiments on Binarised MNIST (Definition 12.2) on fully connected DNNs. We perform Experiment 1 twice, once with MSE loss (to avoid the uncontrolled errors in the GP/EP approximation) and once with C-E loss (as this is a more commonly used loss function, and it has to be used in Experiment 2). We performed Experiment 2 with C-E loss, and finally Experiment 3 with MSE loss. It is clear from the implementations of Experiment 1 that $P_{\text{Bayes}}(f|D) \approx P_{\text{SGD}}(f|D)$ (at

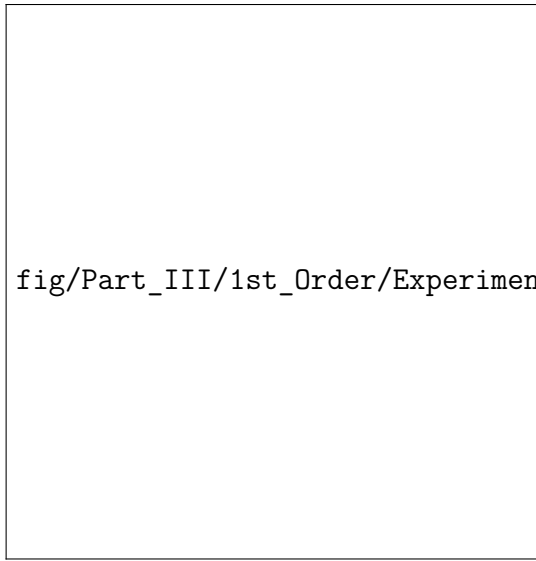
least for this system); Experiment 2 shows that the network is sufficiently expressive to express all functions on the test data yet the Bayesian posterior $P_{Bayes}(f|D)$ is highly biased towards functions which generalise well; and Experiment 3 shows that, with high confidence, that there are no functions with large $P_{Bayes}(f|D)$ but which generalise poorly. For completion, see Figure 23 for Experiment 3 with C-E loss, and Experiment 1 with SGD and C-E loss. While this original experiment is encouraging, this is a simple system (MNIST is considered a very simple task), and the architecture is fairly basic. We have also not tested other optimisers or datasets yet.

We find that, for the experiment in Figure 7d (with parameters from Definition 12.1 but the Adam optimiser), the distance moved by the optimiser in weight space is not large. Each weight moved on average a distance of 0.0071 with a standard deviation of 0.0019. Contrast to the standard deviation of the weights at initialisation, 0.036. Overtraining (training until 64 epochs had passed with 100% accuracy) affected these values by $< 10\%$ (this will be relevant for Figure 9b). We provide this as evidence for our intuition about volumes in Section 11 (where we required the parameters to not move by a great distance in parameter-space).

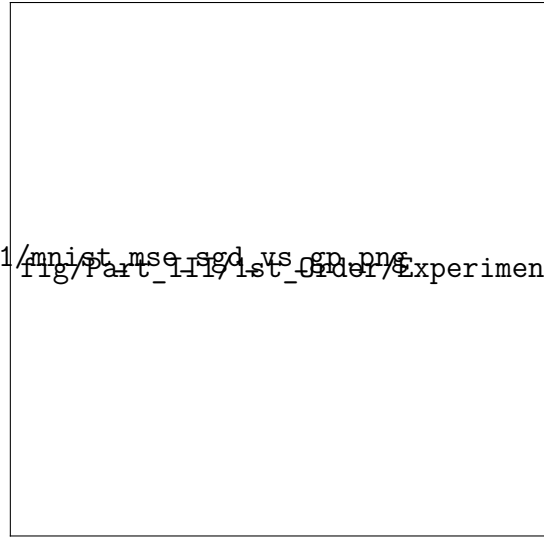
Finally note that $\sum P_{Bayes}(f|D) = 9.63\%$ in Figure 7b. This either shows that (a) the sample mean at each $\langle \epsilon_G \rangle$ is underestimated by sampling in this fashion or (b) that the uncontrolled errors in the EP approximation affect this sum. This is expected to predominately be the effect of (a) (due to the large error bars, which are 2 standard deviations in $\log(p(f))$, and because the distribution after the EP approximation is still a valid probability distribution). However it does appear that the EP approximation underestimates $\log(p(f))$ for high $p(f)$ and overestimates for low $p(f)$; see Appendix C.4 for details. This would imply that the gradient of the slope is underestimated; and also suggests that the line of best fit in Figure 7d should also be much closer to $y = x$. This is backed up by evidence in Appendix C.6, where we compare the GP/EP approximation to the GP approximation on this system, and show that the EP approximation does indeed appear to cause errors in $\log(p(f))$ as described above. Despite these issues, we argue that the EP approximation displays qualitatively correct behaviour (particularly as we can compare to experiments with MSE loss).

These plots raise further questions. First, the test set is very small – only 100 images. What would happen if we made it bigger? Well, in the limit of a very test set in the thousands, we would be highly unlikely to find a particular function more than once in 10^6 samples (given, for example, a generalisation of 1.88% as in Figure 7a), simply due to a combinatorial explosion. We would know however, given that each example in the test set should be i.i.d. (valid for small test sets), the behaviour of Experiments Algorithm 1, 2 and 3 should have a scale invariance – so doubling the test set size should look like mapping each probability for the smaller test set to half its value. Also note that were DNNs unbiased, $P_{Bayes}(f|D)$ would equal $2^{-100} \approx 10^{-32}$ for each function. Contrast this to the actual values of $P_{Bayes}(f|D)$ and $P_{SGD}(f|D)$, which can be higher than 0.5.

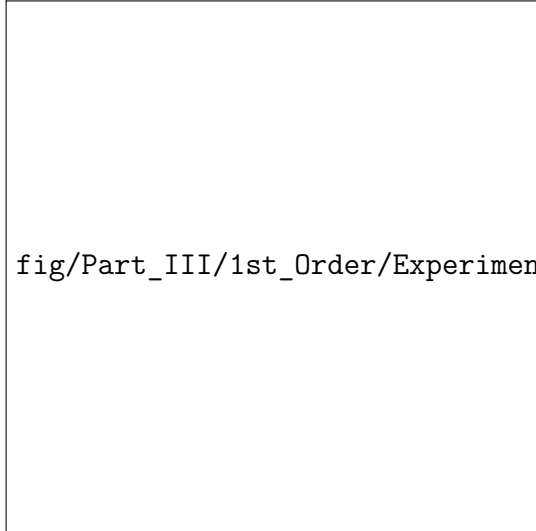
Finally, Experiment 2 has a very striking straight line (for the unweighted curve). we provide in detail analysis of this in Appendix D, but in summary, it appears that the probability of the DNN classifying any image i in the test set incorrectly is well modelled as a Bernoulli variable with probability p_i , where $\langle \epsilon_G \rangle = \langle p_i \rangle$.



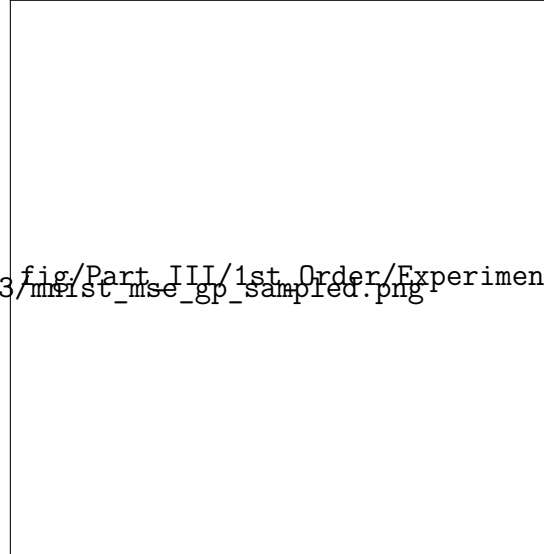
(a) Experiment 1. SGD. $\langle \epsilon_G \rangle = 1.88\%$, 10^6 samples. $\sum P_{Bayes}(f|D) = 99.96\%$



(b) Experiment 2. 20 samples per ϵ_G . $\sum P_{Bayes}(f|D) = 9.63\%$.



(c) Experiment 3. $\langle \epsilon_G \rangle = 1.61\%$. 10^7 samples.



(d) Experiment 1, Adam optimiser. $\langle \epsilon_G \rangle = 2.20\%$. $\sum P_{Bayes}(f|D) = 0.27$

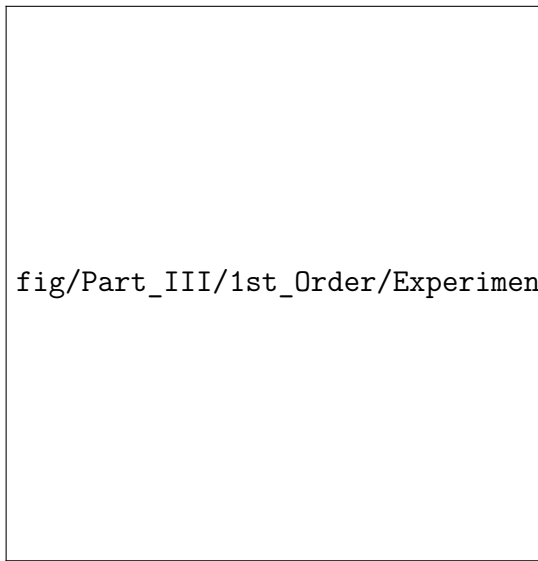
Figure 7: Our main results. All architectures used are fully connected with 2 hidden layers (where appropriate hidden layers are of width 1024). The images from Binarised MNIST (Definition 12.2) were reshaped to a vector of length 784 at input. (a) shows Experiment 1, $P_{SGD}(f|D)$ vs $P_{Bayes}(f|D)$ with MSE loss. $P_{Bayes}(f|D)$ was obtained by sampling from the GP posterior distribution. (b) shows Experiment 2, $P_{Bayes}(f|D)$ vs $\langle \epsilon_G \rangle$ with C-E loss. The light blue line indicates the weighted contribution at each generalisation error, assuming that the sample mean is representative. See Appendix C.4 for explanation of $\sum P_{Bayes}(f|D)$. (c) shows Experiment 3. There were 357 functions found by SGD in (a) that are not found by sampling; 288 only found by sampling and 913 found by both (the latter taking up 97.70% of the probability by SGD, and 99.96% by GP). (d) Shows Experiment 1 with the Adam optimiser and C-E loss. See Appendix C for explanation of the line of best fit not being $y = x$ and low value for $P_{Bayes}(f|D)$.



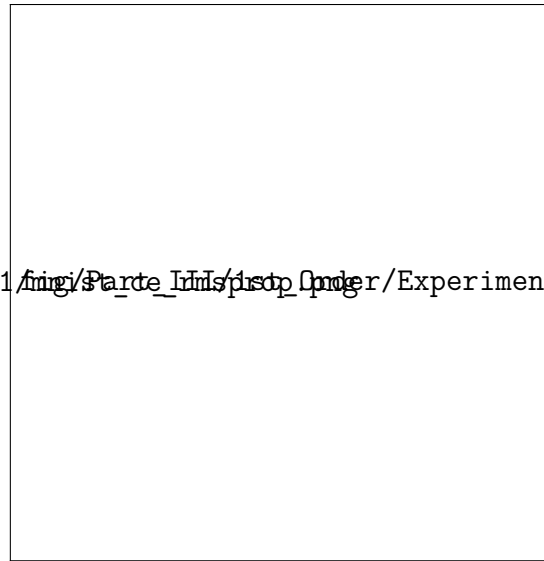
Figure 8: See Section 12.2 for architecture and dataset details. See Appendix C.4 for explanations of why the lines of best fit deviate strongly from $y = x$. (a) Experiment 1 on a CNN with a pooling layer on Fashion MNIST. (b) shows Experiment 1 with an LSTM performing sentiment analysis (dataset: IMDB movie reviews). Low $\sum P_{Bayes}(f|D)$ thought to be an artefact of EP approximation. (c) Shows Experiment 1 with NTK instead of SGD. Clearly there is strong correlation, although NTK appears to miss out on a large number of functions found by SGD (although find most by probability). (d) shows a complexity measure, CSR, for functions found in the posterior distribution of Figure 7b. Clearly there is a strong correlation between the complexity and error (and $P_{Bayes}(f|D)$). The absolute values of CSR are hyperparameter-dependent.

In Figure 8, we provide examples of Experiment 1 with different architectures and datasets: (a) shows Binarised Fashion MNIST trained on a CNN with pooling [15] and (b) shows the result of using an LSTM to learn whether a movie review from the IMDB movie review dataset is positive or negative. (c) shows the result of training with the Neural Tangent Kernel method (NTK) rather than SGD. NTK approximates the behaviour of SGD with infinitesimal learning rate in the limit of infinite width (see [63] for details about NTK). Interestingly, $P_{\text{NTK}}(f|D) \approx P_{\text{Bayes}}(f|D)$ for a subset of the functions found by NTK and by the GP; but there are many functions which are found with high probability by the GP but not found by NTK. We do not currently have a good explanation for this; except that suggesting that the infinitesimal learning rate in NTK may be unable to overcome small barriers in the loss landscape which obviously do not affect the GP, and would not affect SGD. Finally (d) shows an experiment to determine the complexity of functions in the posterior as a function of $P_{\text{Bayes}}(f|D)$ and the generalisation error $\langle \epsilon_G \rangle$. Clearly functions with worse generalisation have, on average, a higher CSR complexity. The complexity measure is called Critical Sample Ratio (CSR) measure [66] also used in [1]. Briefly, it works by calculating something close to the following: what fraction of images in the training and test sets lie a distance in weight space δ from an image with the opposite classification. A high ratio is supposed to imply a ‘complex’ decision boundary, and thys a complex function.

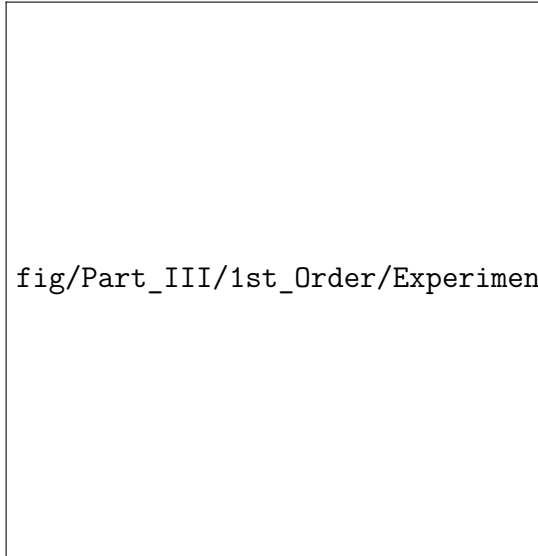
Figure 9 is predominately meant to summarise tests that our broadly-speaking positive results above are not flukes due to lucky hyperparameter choices (although the variety of optimisers, datasets and architectures should already be sufficient for all but the most skeptical). Figure 9 shows a selection of architectures trained on Binarised MNIST. Clearly tuning the hyperparameters does affect the distribution, whether by ‘boosting’ the probability of the most likely functions in $P_{\text{Bayes}}(f|D)$ and affecting the line of best fit, or by affecting the scatter. Optimisers such as RMSprop display qualitatively similar behaviour to Adam in Figure 7d; overtraining (training until 64 epochs were observed at 100% training accuracy) with the Adagrad optimiser does not disrupt the strong correlation (see Section 14.1 for more details about overtraining); neither does using 5 hidden layers; nor does normalising the images in Binarised MNIST (dividing pixel values by 255 so they fall in the $[0,1]$ range).



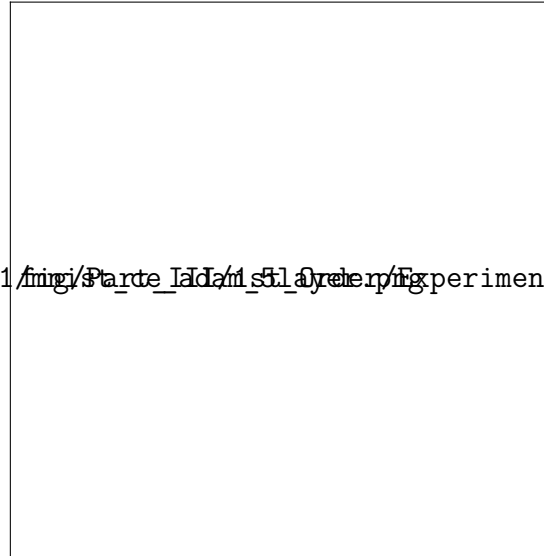
(a) $\langle \epsilon_G \rangle = 1.02\%$, $\sum P_{Bayes}(f|D) = 8.7\%$.
Sample size 6×10^5



(b) Overtraining. $\langle \epsilon_G \rangle = 2.19\%$. Sample
size: 2.1×10^5 . $\sum P_{Bayes}(f|D) = 25.0\%$



(c) $\langle \epsilon_G \rangle = 1.17\%$. Sample size 10^5 .
 $\sum P_{Bayes}(f|D) = 5.5\%$



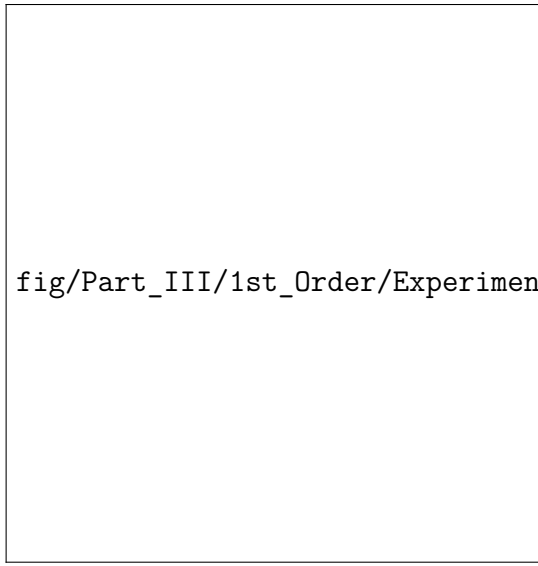
(d) $\langle \epsilon_G \rangle = 1.01\%$. Sample size 10^5
 $\sum P_{Bayes}(f|D) = 5.5\%$

Figure 9: This figure is designed to provide a sample of runs of Experiment 1 with C-E loss and different hyperparameters as evidence that our positive results in earlier figures were not due to careful hyperparameter tuning. These graphs suffer from errors introduced by the EP approximation, but in general show good correlation between $P_{opt}(f|D)$ and $P_{Bayes}(f|D)$. All parameters from Definition 12.1, except: (a) the optimiser is RMSprop, (b) the optimiser is Adagrad, and we *overtrain* (halt training after 64 epochs passed with 100% training accuracy), (c) has 5 hidden layers and for (d), Binarised MNIST was normalised (pixel values divided by 255 such that they fall in the range $[0,1]$).

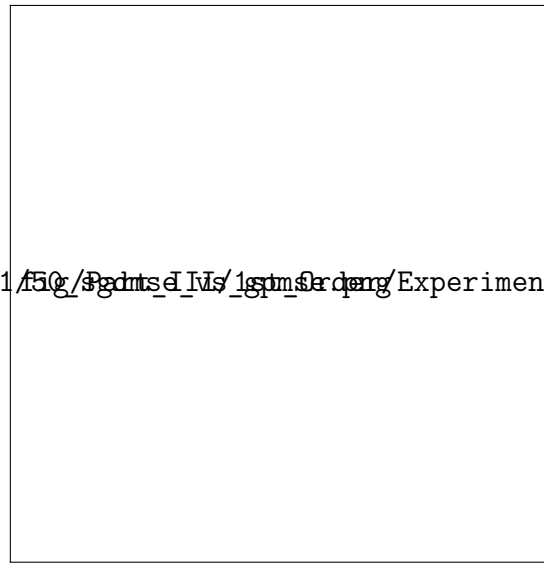
Figure 10 demonstrates results which predominately concern whether $P_{Bayes}(f|D) \approx P_{SGD}(f|D)$ when there is less bias in $P_{Bayes}(f|D)$ – to that end we randomised 20% of the labels in Binarised MNIST and performed Experiment 1 and Experiment 3. We compared this to Experiment 1 with unrandomised Binarised MNIST (the ‘clean’ data). The randomisation is the only difference between the experiments, which have all parameters from Definition 12.1, except the optimiser and batch size: SGD and 32 respectively. We also had to use a smaller test set (size 50) due to a smaller range $P_{SGD}(f|D)$ for the randomised data). It is clear from Experiment 3 in Figures 10c and 10d that the parameter-function map is less biased for the 20% corruption (than with no corruption), although still biased towards good functions, with a generalisation error of 5.80%. In fact, this is noticeably better than the performance of the optimiser, which had a generalisation error of 13.4%. However, the randomised data does not produce the nice correlation we observed with stronger bias in the parameter-function map. Is this a serious problem for our argument? Well, no.

First, 24.33% (by probability mass) of the functions found by the optimiser with probability $> 10^{-4}$ with corrupted data also appeared in the posterior distribution with probability $> 10^{-5}$ – indicating that the behaviour of the optimiser is still strongly, but not exclusively influenced by the parameter-function map. However, it may be possible to reduce the impact of the loss landscape (i.e. move the optimiser closer to Bayesian sampling) with better hyperparameter tuning (for example, smaller batch size, or higher learning rate). It is worth noting that the functions found by SGD but not in the Bayesian posterior generalised poorly, hence the large discrepancy in generalisation error. The optimiser was able to find functions with a combined probability mass of 99.1% in the GP posterior – indicating that it manages to find the majority of the functions with high $P_{Bayes}(f|D)$.

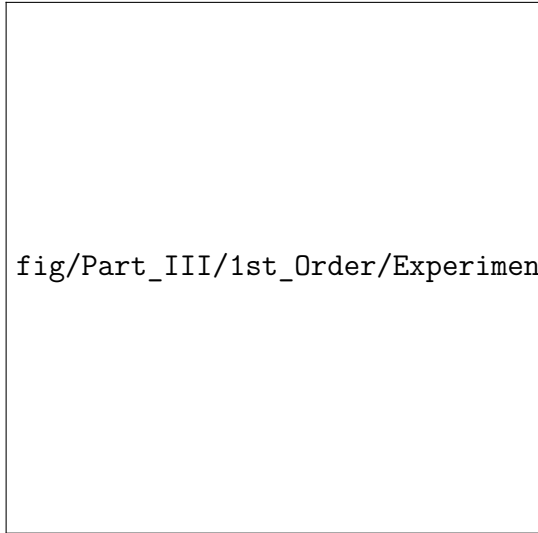
Second, our explanation for why $P_{Bayes}(f|D) \propto P_{SGD}(f|D)$ was essentially that the ‘volume’ of the ‘basin of attraction’ of f and the ‘volume’ of f will correlate for a very biased parameter-function map. In the limit of a completely unbiased map (e.g. in the case of completely random data), we would expect details in the loss landscape and workings of the optimiser to have the biggest impact in $P_{opt}(f|D)$. Presumably between extreme bias and no bias the two effects will be of a similar order of magnitude – and with 20% randomisation, we are definitely between the two extremes.



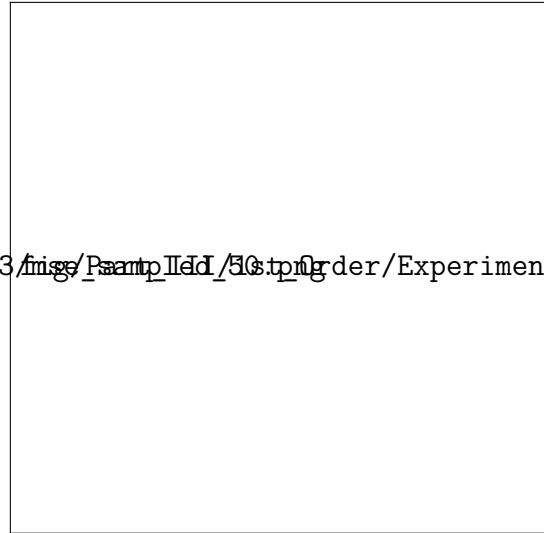
(a) Binarised MNIST. $\langle \epsilon_G \rangle = 2.64\%$.
 $\sum P_{Bayes}(f|D) = 99.996\%$



(b) Binarised MNIST, 20% label corruption. $\langle \epsilon_G \rangle = 13.4\%$.
 $\sum P_{Bayes}(f|D) = 99.1\%$



(c) Experiment 3 with Binarised MNIST.



(d) Experiment 3. Binarised MNIST with 20% label corruption. $\langle \epsilon_G \rangle = 5.80\%$

Figure 10: All of the above experiments used parameters from Definition 12.1, except with MSE loss and SGD. Figures (a) and (c) show Experiment 1 and 3 for Binarised MNIST; and (b) and (d) show the same for Binarised MNIST with 20% training label corruption (i.e. 20% of the labels are incorrect). (c) and (d) show that $P_{Bayes}(f|D)$ is much more biased towards functions with good generalisation with clean data (as expected), and the lower confidence in the true function in the Bayesian posterior is reflected by a significant drop in the probability of the mode function. (b) shows a much weaker correlation between $P_{Bayes}(f|D)$ and $P_{opt}(f|D)$. (b) and (d) also show that there are many functions the optimiser finds but do not have high probability in the GP – these functions account for 75.7% of the probability weight by SGD have negligible $P_{Bayes}(f|D)$.

13.1 Brief summary

Thus, we have provided preliminary evidence that (a) the parameter-function map is strongly biased towards functions that generalise well, and (b) that the probability that SGD-like optimisers (in this case Adam) find a function f is primarily dictated by the volume of f in parameter-space. Our results thus corroborate the argument made in [1] and [4], as they imply that the main source of generalisation is due to the parameter-function map, and not the optimiser.

However, there are a number of unanswered questions. First, how exactly the GP/EP approximation affects the distribution is still an open question. We provide some insights in Appendix C.4, and our analysis suggests that the EP approximation decreases the gradient of the line of best fit on a log-log plot (for the experiment with Binarised MNIST on a FCN with 2 layers of width 1024) by 166% (conveniently about the right amount to put the gradient around $y = x$). Second, exactly how and why $P_{Bayes}(f|D)$ diverges from $P_{opt}(f|D)$ more when there is less bias in the parameter function map. Finally, can Experiment 1 shed how hyperparameter tuning affect generalisation? This is the question we will discuss in the following section.

14 2nd Order Generalisation

In this section, we will go through results which suggest that using Experiment 1 can provide insight into the effect of hyperparameter tuning. The majority of the experiments used C-E loss, and thus require the GP/EP approximation. Due to the nature of the GP/EP approximation, the majority of the conclusions can be qualitative only.

14.1 Optimisers and Overtraining

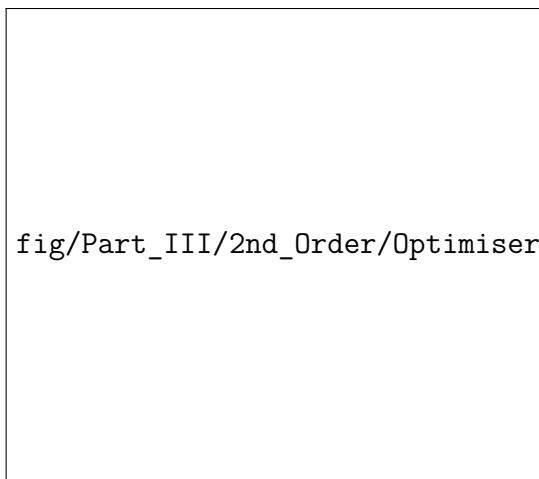
In order to investigate how different optimisation algorithms sample parameter-space, we implemented Experiment 1 with C-E loss and four different optimisers – Adam, Adagrad, Adadelta and RMSprop. Other than the optimisers, all the parameters are from Definition 12.1. We also investigated how overtraining affected the functions each optimiser found – which was done by halting training after 64 epochs passed with 100% accuracy. The results of these eight experiments are

shown Figure 11. We also plotted the correlations between these optimisers (without overtraining) in Figure 13. This plot does not suffer from the errors with the EP approximations (for obvious reasons; we don’t need $P_{Bayes}(f|D)!$), and it shows that Adam and Adagrad sample function space in a very similar way; and RMSprop and Adadelta are similar, but there is a noticeable difference between the pairs: RMSprop/Adadelta boosts the probability for functions with high $P_{opt}(f|D)$ compared to Adagrad/Adadelta. Understanding such effects could guide engineers in choosing the best optimiser. We also performed these experiments with MSE loss and the Adam optimiser (see Figure 12) to avoid the EP approximation.

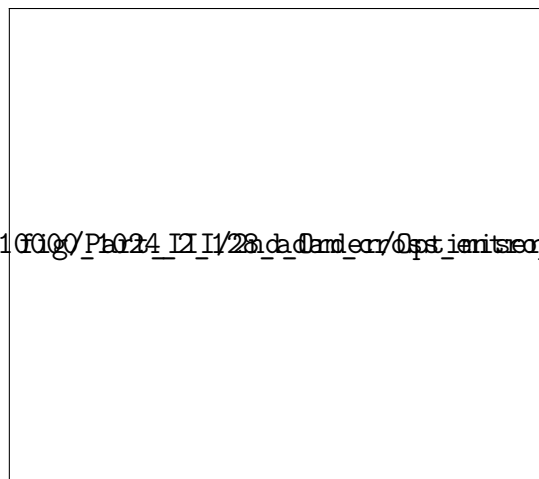
The experiments with MSE loss and the Adam optimiser (Figure 12), with unnormalised data, show that overtraining appears to have a significant effect on how the optimiser samples parameter-space – overtraining appears to tilt the line of best fit to boost the probability of functions with high $P_{Bayes}(f|D)$, and also increases the scatter. With normalised data, this effect was negligible.

Experiments in Figure 11 (with 4 different optimisers, unnormalised data and C-E loss) show qualitatively similar results – for each optimiser, $P_{Bayes}(f|D)$ correlates well with $P_{SGD}(f|D)$. Deviation from $y = x$ appears to be predominately a result of the EP approximation (see Appendix C.4) for details. Overtraining also appears to boost the probability of functions with high $P_{Bayes}(f|D)$, although in this case the scatter was reduced.

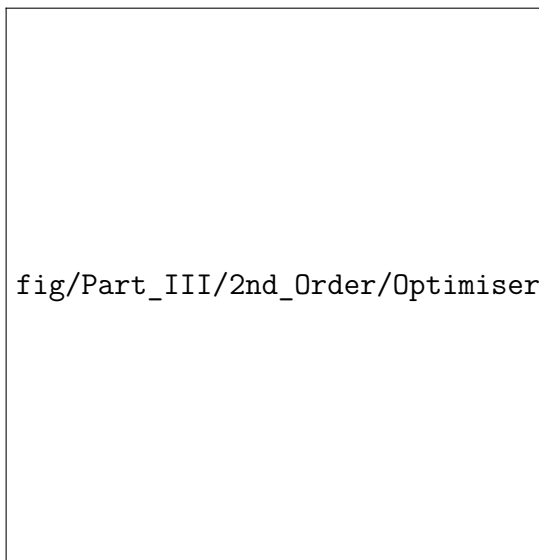
In summary, preliminary results suggest that different optimisers sample the space slightly differently, although the dominating factor deciding $P_{opt}(f|D)$ is, as expected, $P_{Bayes}(f|D)$. Overtraining and normalisation of data also appear to affect the correlation between $P_{opt}(f|D)$ and $P_{Bayes}(f|D)$. See Figure 25 in Appendix C.6 for some of these experiments corrected for the EP approximation – which were not put here as the correction is not rigorously justified.



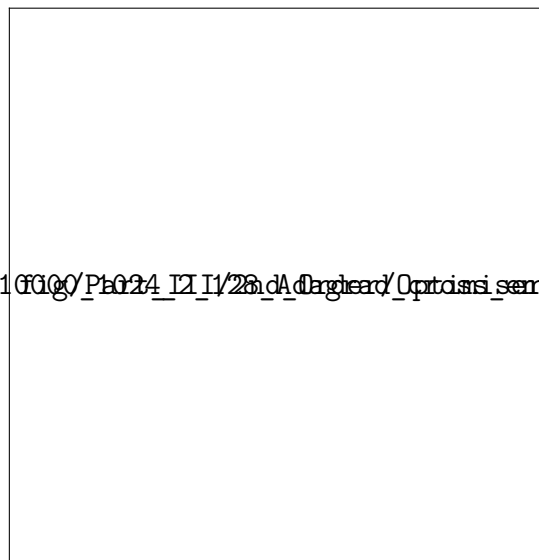
(a) No overtraining. $\langle \epsilon_G \rangle = 2.20\%$. Sample size: 10^6 .



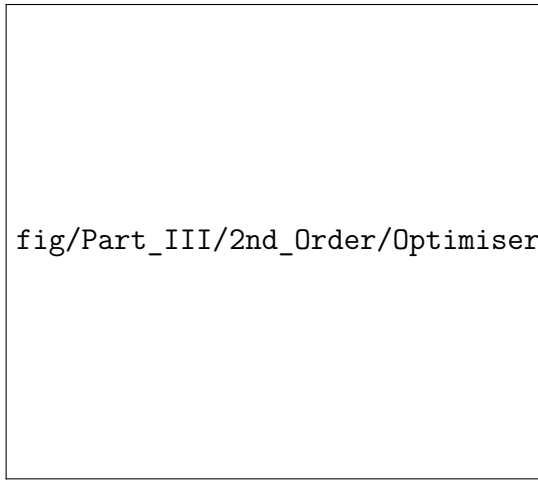
(b) Overtraining. $\langle \epsilon_G \rangle = 1.73\%$. Sample size: 10^6 .



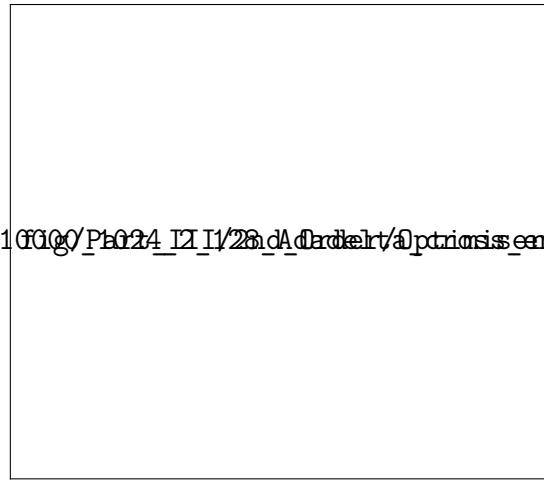
(c) No overtraining. $\langle \epsilon_G \rangle = 2.63\%$. Sample size: 10^6 .



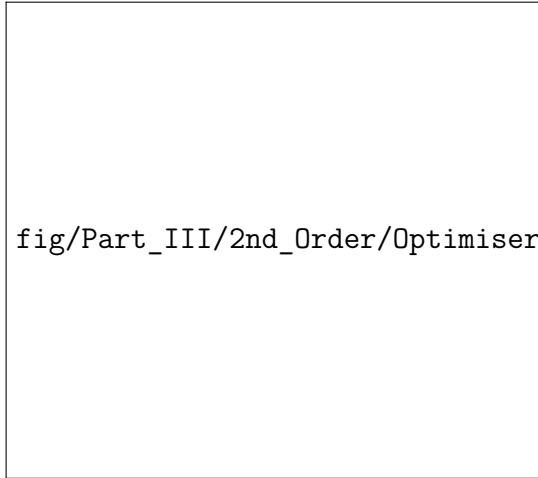
(d) Overtraining. $\langle \epsilon_G \rangle = 2.19\%$. Sample size: 2.1×10^5 .



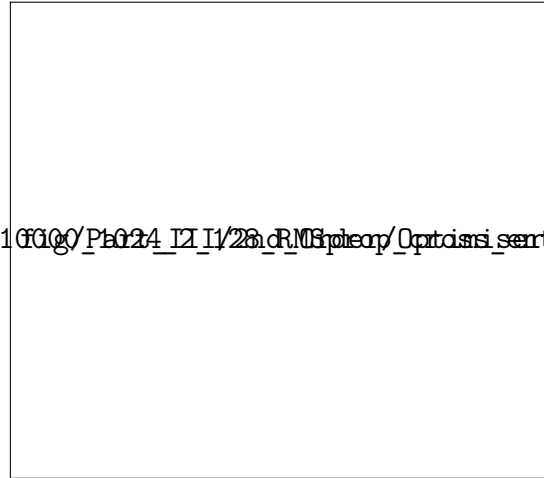
(e) No overtraining. $\langle \epsilon_G \rangle = 1.23\%$. Sample size: 10^6 .



(f) Overtraining. $\langle \epsilon_G \rangle = 1.17\%$. Sample size: 10^5 .

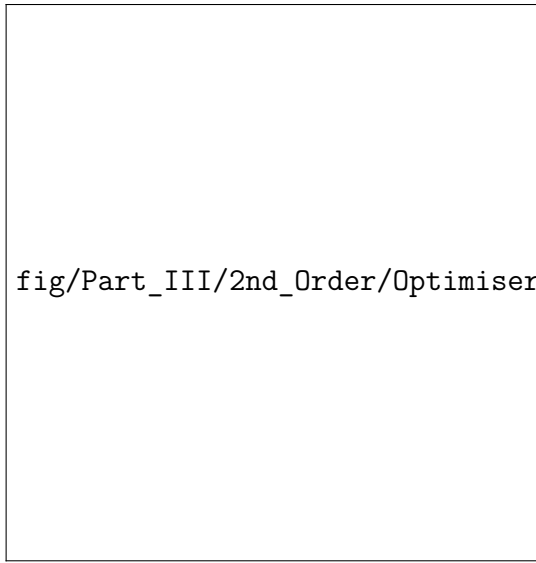


(g) No overtraining. $\langle \epsilon_G \rangle = 1.02\%$. Sample size: 6.1×10^5 .

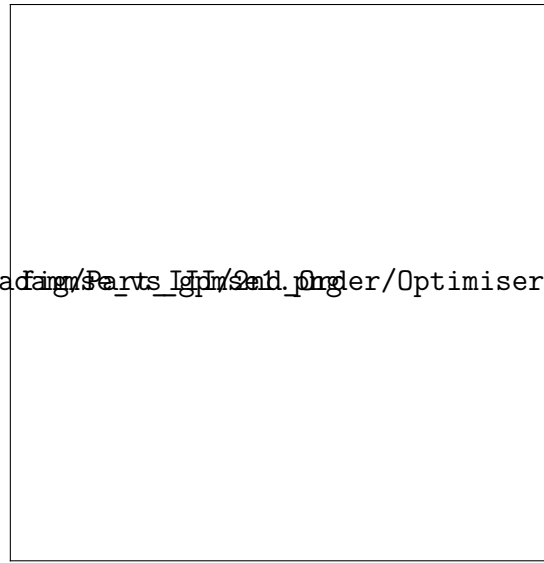


(h) Overtraining. $\langle \epsilon_G \rangle = 1.01\%$. Sample size: 2.5×10^5 .

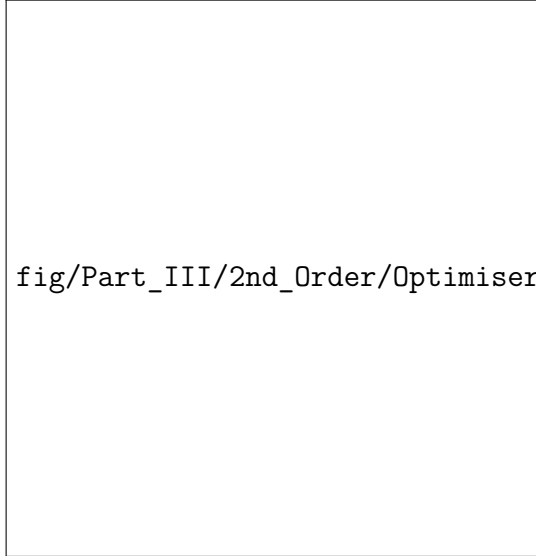
Figure 11: We performed Experiment 1 on binarised MNIST, with all parameters from Definition 12.1, except for the optimiser, which is: (a) Adam (c) Adagrad (e) Adadelta and (g) RMSprop. We also implemented Experiment 1 with the different optimisers and overtraining, where we halted training after 64 epochs passed with 100% accuracy, in figures (b) Adam, (d) Adagrad, (f) Adadelta and (h) RMSprop. The probability each optimiser finds a function correlate well with the relative volumes of the functions. The correlations are log-linear, although with different exponents for each optimiser. The log-linearity is thought to be due to the EP approximation (see Appendix C.6 for a suggested method for correcting this). We calculated the error from the full sample size, and then removed frequencies below 10 in these graphs to avoid finite-size effects. The functions to the right of the blue dotted lines make up 90% of the total probability.



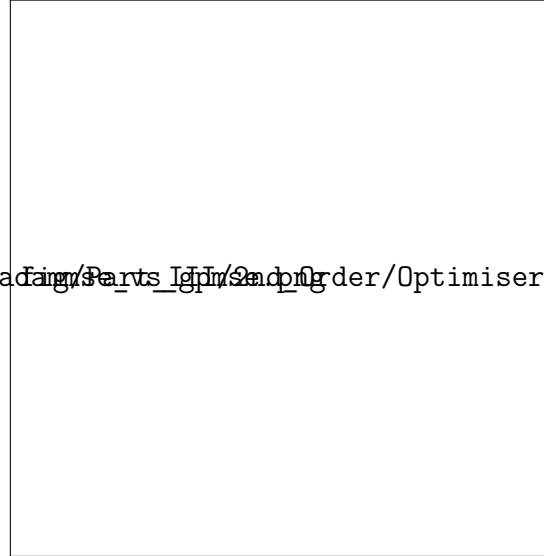
(a) No overtraining. $\langle \epsilon_G \rangle = 1.72\%$ Sample size: 10^5 .



(b) Overtraining. $\langle \epsilon_G \rangle = 1.384\%$. Sample size: 10^5 .

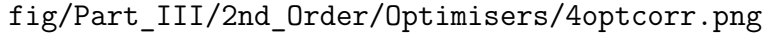


(c) No overtraining. Normalised Data. $\langle \epsilon_G \rangle = 0.936\%$ Sample size: 10^5 .



(d) Overtraining. $\langle \epsilon_G \rangle = 0.923\%$. Normalised data. Sample size: 10^5 .

Figure 12: We performed experiments analogous to Figure 11 but with MSE loss to avoid the EP approximation. The experiments were performed with the parameters from Definition 12.1, but with MSE loss and the Adam optimiser. (a) and (b) have unnormalised image data (pixels in range $[0,255]$); (c) and (d) have normalised image data (pixels in range $[0,1]$). In (a) and (c) we halted training once 100% training accuracy was reached; in (b) and (d) we halted training after 64 epochs had passed with 100% training accuracy (mirroring Figure 11). Interestingly, overtraining made a substantial impact on the correlation when using unnormalised data (as in Figure 11, where similar effects were observed) . However, with normalised data, the effect of overtraining was negligible.



fig/Part_III/2nd_Order/Optimisers/4optcorr.png

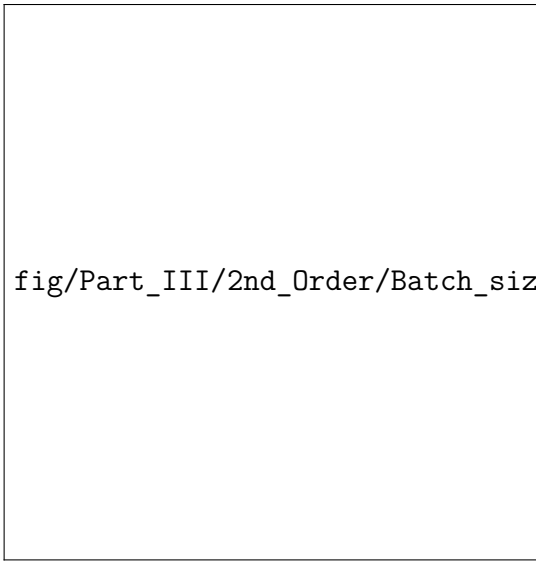
Figure 13: Correlation between optimisers shown in Figure 11. All functions which had a probability of $< 10^{-4}$ had their probabilities set to 10^{-4} for these graphs. Note that Adagrad and Adam correlate very well; and the line of best fit for Adadelata and RMSprop is very close to $y = x$. The other correlations clearly show a line of best fit that is not $y = x$. consistent with results from Figure 11. These plots do not rely on the EP approximation, and we believe that this sort of experiment may prove useful for understanding differences in the behaviour of the optimisers. For example, RMSprop and Adadelata appear to assign higher probability to functions with already high $P_{\text{opt}}(f|D)$ – a desirable property for a correctly biased parameter-function map.

14.2 Batch size

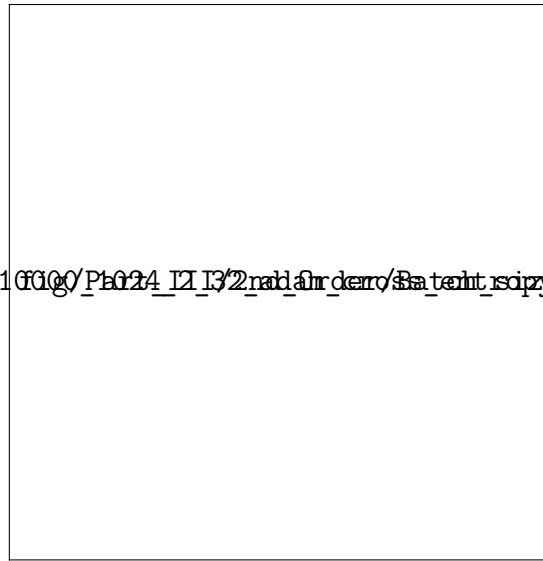
We consider the effect of changing the batch size. As observed in [30], with all other parameters held constant, different batch sizes cause differences in the behaviour of the optimisers. Adam with a batch size of 32 will find the function with the highest GP/EP volume more frequently than Adam with higher batch sizes (with probability 0.66, compared to 0.16 with batch size 128, 0.07 for 512 and 0.07 for 1024). The line of best fit is clearly shallower for smaller batch sizes, suggesting that small batch Adam amplifies the bias in the parameter function map more than large batch files. If the bias in the parameter function map is good for the task in hand, this would be beneficial for generalisation. This method may provide a lens for analysing results in [30].

It was suggested in [39] that in order to reduce the difference in generalisation induced by changing batch size b , the learning rate η should scale as $\eta \propto \sqrt{b}$ (because smaller batch sizes perform more updates per epoch, and are thus able to search more parts of the parameter space per epoch). They observed that scaling the learning rate in this way led to an improvement in generalisation for large batch SGD relative to small batch SGD, we used a learning rate 4x bigger than the default rate for Adam (default rate is 0.001), with a batch size of 512. We observe that it has $P = 0.16$ for the highest function, which is most similar to that of the 128 batch size (and not 32, as the square-root scaling would suggest; this is an area which could be studied further using this method). Overall, the high probability region of the graph is most similar to Figure 14b, where a batch size of 128 was used. However, low probability regions seem more similar to Figure 14c, where a batch size of 512 was used.

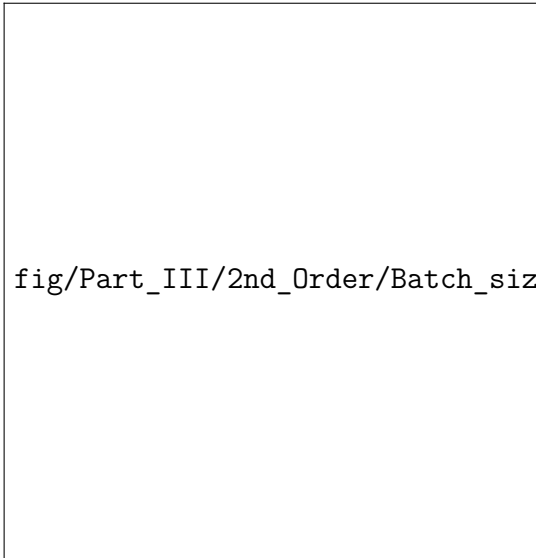
Finally, we compare the effect of batch size with an MSE loss function and normalised data in Figure 15. We observe that it appears to have the opposite effect – large batches appear to amplify the bias. It is therefore hard to come up with a direct comparison, which we leave to future work. Note that for these two graphs we calculated the line of best fit from functions with probabilities $p > 10^{-3}$ as this gave a better qualitative fit (for the most frequently found functions).



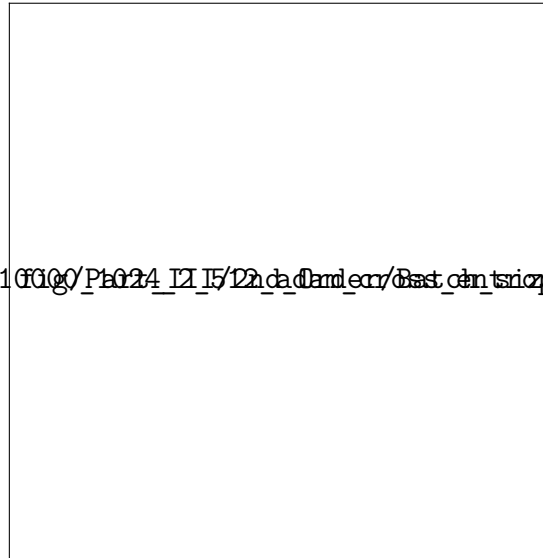
(a) Experiment 1, Binarised MNIST, $\langle \epsilon_G \rangle = 1.13\%$.



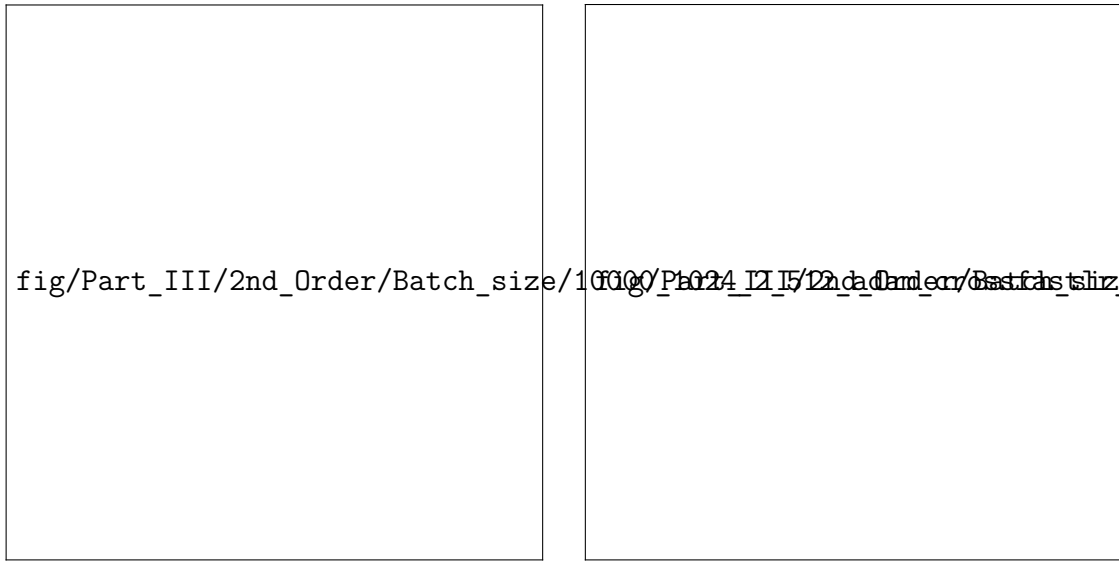
(b) Experiment 1, Binarised MNIST, $\langle \epsilon_G \rangle = 2.20\%$.



(c) Experiment 1, Binarised MNIST, $\langle \epsilon_G \rangle = 2.67\%$.



(d) Experiment 1, Binarised MNIST, $\langle \epsilon_G \rangle = 2.68\%$.



(e) $\langle \epsilon_G \rangle = 2.14\%$. Learning rate 0.004 (4x default).

(f) $\langle \epsilon_G \rangle = 1.73\%$, batch size 128 with overtraining (64 epochs).

Figure 14: We performed Experiment 1 on binarised MNIST, with all parameters from Definition 12.1, except for the batch size, which is (a) 32, (b) 128, (c) 512, (d) 1024, (e) 512 and with learning rate 4x default and (f) 128 and with overtraining. Clearly changing the batch size does affect how the optimiser samples the space; with smaller batches amplifying the bias to a greater extent than larger ones. Increasing the learning rate has a similar effect to decreasing the batch size, at least for the most common functions (compare to results in [39]). Overtraining also appears to have a similar effect to decreasing the batch size.



(a) Experiment 1, Binarised MNIST, $\langle \epsilon_G \rangle = 1.88\%$. Normalised Data.

(b) Experiment 1, Binarised MNIST, $\langle \epsilon_G \rangle = 1.22\%$. Normalised Data.

Figure 15: We performed Experiment 1 on binarised MNIST, with all parameters from Definition 12.1, except for the loss function (MSE) and batch size, which is (a) 32, and (b) 128. For this loss function and normalised data, we observe that increasing the batch size leads to a shallower line of best fit (in contrast to Figure 14, where cross-entropy loss was used). However, in Figure 14, unnormalised data was used, and we observed in Figure 12a that this also plays a role in optimiser behaviour.

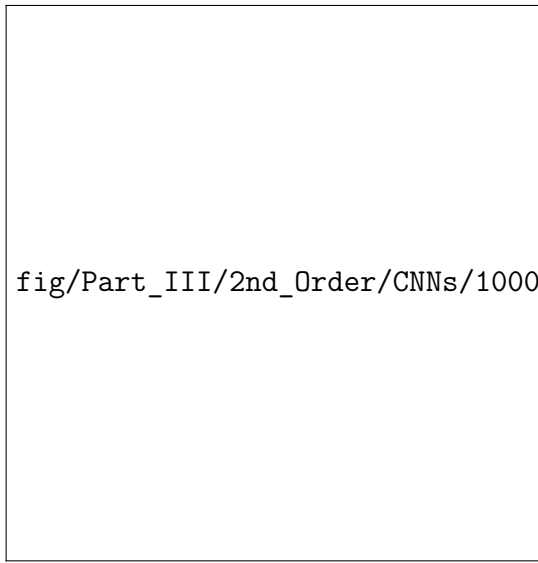
14.3 CNNs, LSTMs, NTK and the Ionosphere Dataset

In this appendix, we use binarised Fashion MNIST (see Definition 12.3). We also use Convolutional Neural Networks (CNNs), and observe that the correlation between $P_{\text{opt}}(f|D)$ and $P_{\text{Bayes}}(f|D)$ is still strong for CNNs. This suggests that the correlation is not limited to Fully Connected Networks. We tested three different ConvNet architectures which we explained in Section 12.2. As expected, functions with good generalisation appear to have higher $P_{\text{Bayes}}(f|D)$ with the pooling layer compared to no pooling (not applicable to the architecture with batch norm, as we used the same kernel for the CNN with Pooling as the CNN with Pooling and Batch Norm). Batch norm did affect which functions were found in the posterior, but this was by a very small amount which didn't affect generalisation to a significant amount.

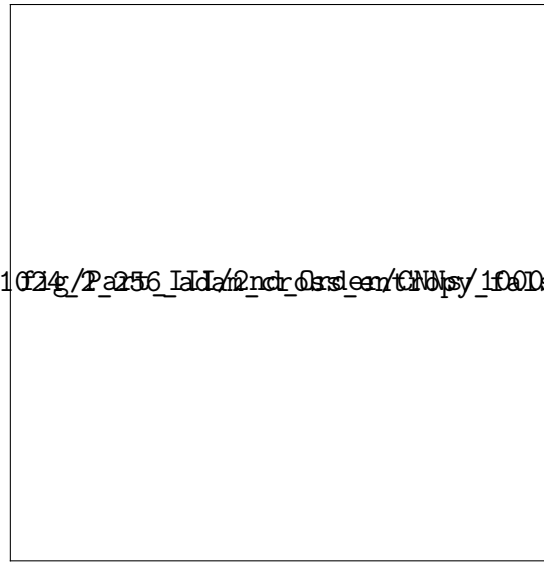
We also perform sentiment analysis on a DNN with an LSTM layer [58]. We take the IMDB movie review dataset from keras, and preprocess it (by removing the most common words and normalising it). This was to obtain good generalisation, given that we need to find the same function on the test set multiple times. However, it was found that the most common function was still insufficiently common with a test set of size 100, so a test set of size 50 was used. Unfortunately, we had many problems getting the kernel to work something something.

The final part of this section concerns NTK. Strictly speaking this belongs in Section 13, but there was not enough room. Clearly NTK (which is an algorithm modelling the behaviour of SGD with infinitesimal learning rate in the limit of infinite width, see [63]) behaves similarly to SGD and GP for the majority of functions (by probability weight). However, a substantial minority ($\sim 30\%$) of functions which have relatively high $P_{\text{Bayes}}(f|D)$ or $P_{\text{opt}}(f|D)$ are not found by NTK (meaning they must have very low $P_{\text{NTK}}(f|D)$). Understanding this effect is left to future work.

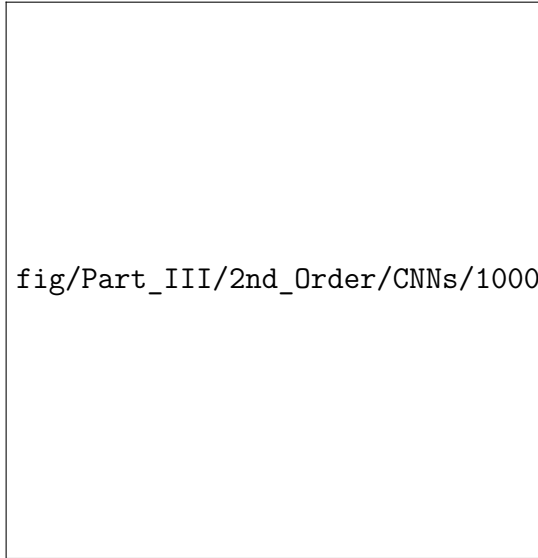
In Figure 20, we trained a fully connected DNN on a small non-image dataset with 34 features. This was to test another non-image dataset (with MSE loss to avoid the EP approximation).



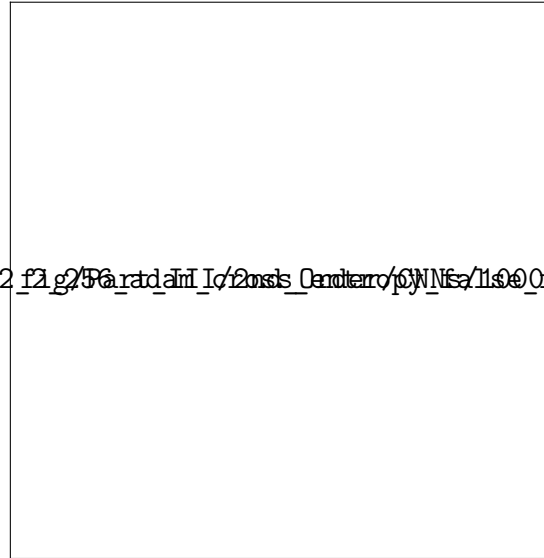
(a) Fully Connected Network (FCN).
 $\langle \epsilon_G \rangle = 2.11\%$



(b) Convolutional Neural Network (CNN).
 $\langle \epsilon_G \rangle = 2.25\%$



(c) CNN with pooling. $\langle \epsilon_G \rangle = 1.96\%$.



(d) CNN with pooling and batch norm.
 $\langle \epsilon_G \rangle = 1.93\%$

Figure 16: We performed Experiment 1 with the standard parameters (see Definition 12.1), except the batch size (256), the dataset (Fashion MNIST), and the architectures (various, see figure titles). While the FCN (a) outperforms the CNN (b), it is worth noting that the architecture of the CNN was not tuned precisely. As expected, the CNN with pooling, (c), outperforms the CNN (b), and adding batch norm, (d), also improved generalisation error marginally. The same GP kernels were used for (c) and (d) (despite it being possible to include the effect of batch norm in the kernels, but it was felt that not doing this was as interesting). There is good correlation between $P_{Bayes}(f|D)$ and $P_{SGD}(f|D)$ in all four subfigures; suggesting that this correlation is not limited to FCNs but also applied to CNNs.

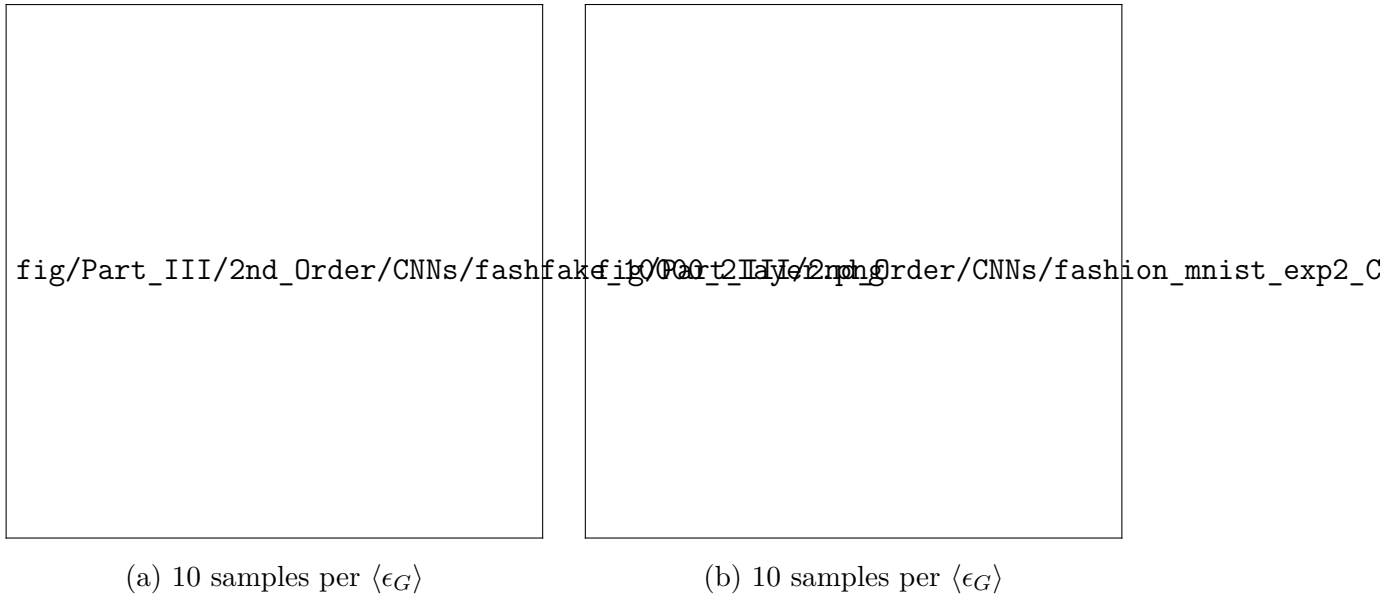


Figure 17: Experiment 2 with Fashion MNIST. (a) with FCN (same architecture as Figure 16a), and (b) with the CNN+Pool architecture (same as Figure 16c).

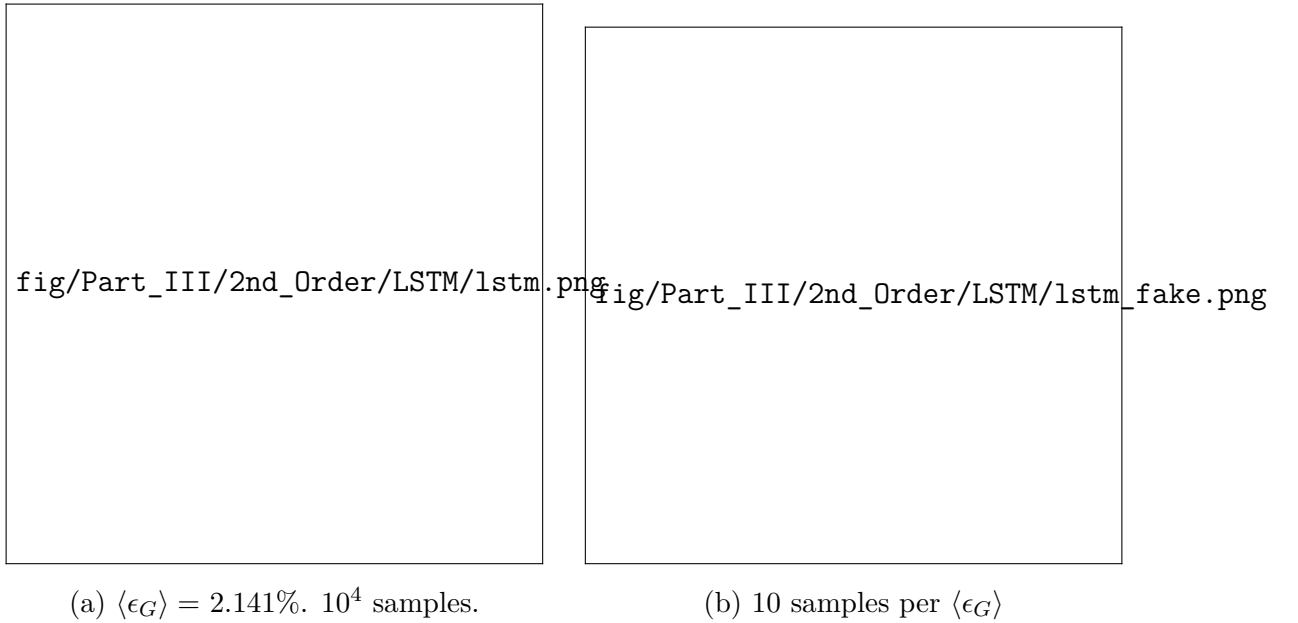


Figure 18: Here, we present examples of Experiments 1 and 2 on sentiment analysis using an LSTM and C-E loss. Because of the difficulty of the problem, we used a training set size of 45000 and a test set of size 50. In (a) there is clearly some weak correlation, and in (b) we see that the functions found by Adam are still the most common in the GP posterior. However, it is thought that the EP approximation affects the absolute values of the probabilities.


fig/Part_III/2nd_Order/NTK/ntk_vs_gpmse.png

fig/Part_III/2nd_Order/NTK/ntk_vs_sgdmse.png

(a) Dataset: Binarised MNIST. 10^7 samples from both NTK and GP. Generalisation Error (NTK): 1.69%, Generalisation error (GP): 1.61%. Weighted by probability, 77.5% of functions found by sampling from the GP found by NTK; all functions found by NTK found by sampling from the GP posterior distribution.


(b) Dataset: Binarised MNIST. 10^7 samples from NTK; 10^6 samples from SGD. Generalisation Error (NTK): 1.69%, Generalisation error (SGD): 1.88%. Weighted by probability, 65.8% of functions found by SGD found by NTK; all functions found by NTK found by SGD.

Figure 19: In (a) we show the correlation between $P_{NTK}(f|\mathcal{D})$ and the $P_{SGD}(f|D)$, using the same training and test datasets as used throughout the section (binarised MNIST). In (b), we show the correlation between the $P_{NTK}(f|\mathcal{D})$ and $P_{Bayes}(f|D)$. The functions to the right of the blue dotted lines make up 90% of the total probability.



fig/Part_III/2nd_Order/ionexp1.png

(a) Experiment 1 with the ionosphere dataset and MSE loss. Generalisation error: 4.59%



fig/Part_III/2nd_Order/ionexp2.png

(b) Experiment 3 with the ionosphere dataset and MSE loss. Generalisation error: 5.41%

Figure 20: In this figure, we perform Experiments 1 to 3 on the ionosphere dataset, a non-image dataset with 34 features and 301 training examples, with a test set of size 50. We use the MSE loss function in (a) and (b). The functions found by both Adam and GP sampling accounted for 43.1% of the probability mass in the Bayesian posterior and 99.8% of probability mass of functions found by the optimiser. Ionosphere Dataset: <https://archive.ics.uci.edu/ml/datasets/Ionosphere>

15 Unanswered Questions/Future work

A recurring issue with these experiments has been the EP approximation. We have circumstantial evidence that the EP approximation’s primary effect appears to be that it ‘flattens’ the line of best fit in Experiment 1 and 2. This is backed up by evidence in [1], and Appendix C.4. Unfortunately, this means it is very hard to make quantitative conclusions about any results when using C-E loss, especially when studying ‘Second-order generalisation’ (Section 14). The major issue it causes for evidence for ‘first-order generalisation’ is its effect on the LSTM, where it appears to underestimate probabilities by 5 orders of magnitude for the most frequent functions. Hopefully future work can either manage to train an LSTM with MSE loss or find a way of quantifying the effect of the GP/EP approximation.

The second important question concerns how the optimiser behaves when there is less bias in the parameter function map. As mentioned in Section 11, one of the assumptions is that the ‘basin of attraction’ of a function correlates well with $P_{\text{opt}}(f|D)$, which seems reasonable if there is strong bias in the parameter function map. However, we argued at the end of Section 13 (when discussing Figure 10) that in the limit of no bias in the parameter function map, $P_{\text{opt}}(f|D)$ would not correlate well with $P_{\text{Bayes}}(f|D)$, as the interaction of the optimiser with the loss landscape would be the dominating effect. Clearly in the other limit (where $P_{\text{Bayes}}(f|D) = \delta_{f,f'}$ for some f'), $P_{\text{opt}}(f|D) = P_{\text{Bayes}}(f|D)$. Therefore, between these two extremes both of these effects will dominate. It appears that with 20% random labels on Binarised MNIST, the parameter function map still plays a large role (by probability mass, 24% of functions found by the optimiser have high $P_{\text{Bayes}}(f|D)$), and 99% of functions (by mass) in the Bayesian posterior were found by SGD. However, the behaviour of the optimiser in Figure 20 was the opposite, implying that this may not be true in general, and such effects may be strongly architecture, optimiser and dataset dependent.

Further study may shed light on the exact point at which these effects become noticeable. We provide a list below of interesting avenues that could be taken. To increase the strength of results in Section 13, an experiment with MSE loss and a LSTM should be performed (currently we have an experiment with C-E loss). For the same reason, we should perform an experiment with Binarised

Fashion MNIST and a CNN with MSE loss (to avoid the EP approximation). Currently, we have only performed Experiments 1, 2 and 3 on binarised datasets – extending this to multi-class or non classification data would be nice.

While these experiments would lend strength to the general argument, understanding ‘second-order generalisation’ would likely be of more practical use. The most obvious thing to test the way changing the batch size affects the relationship between $P_{Bayes}(f|D)$ and $P_{opt}(f|D)$, for more optimisers, architectures and datasets, for MSE loss and C-E loss. Then the way overtraining affects the relationship between $P_{Bayes}(f|D)$ and $P_{opt}(f|D)$ (for MSE and CE loss) should be tested, for more optimisers, architectures, datasets, and bias in the parameter-function map for MSE loss and C-E loss. These tests should determine the extent to which effects observed in Section 14 are general.

More optimistically, based on this understanding, is there a more sophisticated method for performing architecture search? Other interesting experiments to run include the following. Given that DNNs are close to a Bayesian sampler, retraining them multiple times can give an estimate for the confidence of a DNN as to the classification of the image. This would be good news because SGD is much more scalable than Monte Carlo based algorithms often used in Bayesian DNNs, such as Gaussian processes. However, the need for multiple samples can make the training process much slower.

Does this correlate with the deviation of pre-thresholded outputs from their labels? If this confidence correlates with the confidence represented by the softmax output

Chris: list more

16 Conclusion

In this dissertation, we sought to understand why DNNs generalise well for a wide range of tasks. In Part I, we laid most of the ground work required for results in Parts II and III. We highlighted how in order to learn (guess the true explanation from example data \mathcal{D}), an inductive bias is necessary for highly expressive learning agents (to choose between the many possible descriptions). We then suggested that a good inductive bias would be to assign high probability to ‘simple functions’, and that a general learning agent would do this (approximating Solomonoff Induction).

In Part II, we discussed empirical evidence and analytic results from [1, 4]. These results implied that DNNs trained by randomly sampling parameters would exhibit a strong bias towards simple functions. We saw how this had been tested empirically on small DNNs, and how the correspondence between Gaussian Processes and infinite-width DNNs trained by randomly sampling parameters had been used to test whether the bias in the parameter-function map was sufficient to guarantee good generalisation. Preliminary results suggested that it was, but the bounds were still not perfectly tight, and are not necessarily applicable to DNNs trained by an optimiser like SGD (how DNNs are trained in practice).

The focus of this dissertation was on providing empirical evidence that optimiser-trained DNNs and DNNs trained by randomly sampling parameters/Gaussian Processes behave in a very similar way, using a more fine grained measure than generalisation error (as had been used previously [1, 63]). We measured the probability that an optimiser finds a function f (trained to consistency with some dataset, D) $P_{\text{opt}}(f|D)$; and compared this to the probability that a DNN trained by randomly sampling parameters finds the same f conditioning on D , $P_{\text{Bayes}}(f|D)$. We showed that, for a wide range of optimisers, architectures and datasets, $P_{\text{Bayes}}(f|D) \approx P_{\text{opt}}(f|D)$, and that hyperparameter changes can induce second order changes to the relationship.

So, in conclusion, we suggest that the DNNs generalise because the parameter-function map is strongly biased towards simple functions. If this is true, then it may be possible to create more better DNN-type learning agents by studying the parameter-function map.

A Codes

We begin by formally defining a code below.

Definition A.1 ((Uniquely Decodable) Code). *A code, C , is a function*

$$C : \mathcal{S} \rightarrow \mathcal{A}^*, \quad (21)$$

where \mathcal{S} is a source alphabet; and \mathcal{A} is the code alphabet, both of which are discrete (and for our purposes finite) sets. Note that the function is into \mathcal{A}^ (a sequence of symbols in \mathcal{A} of arbitrary length) – because each letter in \mathcal{S} can have an arbitrarily long code. C is uniquely decodable if it has a uniquely defined inverse, C^{-1} . We define a ‘codeword’ of some ‘word’ $w \in \mathcal{S}$ to be $C(w)$ (note that some definitions take this to be for $w \in \mathcal{S}^n$).*

Codes can be categorised into prefix and non-prefix codes. Prefix codes satisfy the following property: no codeword c_1 can be made by taking any other codeword c_2 and appending symbols in \mathcal{A} (such that c_2 is a prefix of c_1). A non-prefix code is any code that is not a prefix code [67]. There is also an advantage in making codes which ‘compress’ information as much as possible. Consider a ‘memory-less’ source Z which at each timestep outputs $s_i \in \mathcal{S}$ with constant probability p_i . A code is considered optimal (with respect to Z) if no code with a lower mean code-word exists. There is an algorithm which constructs the optimal binary code ($\mathcal{A} = \{0, 1\}$) with respect to any source Z and source alphabet \mathcal{S} . This optimal code is called the ‘Huffman Code.’ The Huffman code is required for the proof of the upper bound (Equation (4)) in Theorem 5.1. Consider $s_i \in \mathcal{S}$ and associated probabilities p_i . Then, a Huffman code is constructed in the following way:

Huffman Code

input: Labelled pairs $\{x_i, p_i\}$

while True, do:

 create a node in a binary search tree where the two branches attach to the smallest aggregate p ’s, q and q' . Assign a 0 to the smaller branch and 1 to the larger branch

 assign a probability value $q + q'$ to the node.

 update the labelled pairs to no longer include the pairs with q and q' ; include the new node with probability $q + q'$

if only one node remains, **return** the search tree

The Huffman lengths of the codewords for s_i , $L(s_i)$, satisfy $l(s_i) \leq \lceil -\log_2(p_i) \rceil$. One important theorem regarding codes is the Kraft-McMillan inequality.

Theorem A.2 (Kraft-McMillan Inequality). *Consider a code $C : \mathcal{S} \rightarrow \mathcal{A}^*$, where $a = |\mathcal{A}|$ and $S = |\mathcal{S}|$. Consider a set of lengths $l(s) = |C(s)|$. Then, if and only if*

$$\sum_{s \in \mathcal{S}} a^{-l(s)} \leq 1, \quad (22)$$

C is uniquely decodable. See section 5.2, [68] for proof.

It allows us to prove that the universal a-priori probability $Q_U(x)$ (see Equation (1)) does not diverge for prefix UTMs. This can be achieved by showing that the set of prefix UTMs is in 1 – 1 correspondence with the set of uniquely decodable codes, and applying the Kraft-McMillan inequality [38].

B Alternative hypotheses

First, we will discuss some of the alternative hypotheses, which suggest that the source of generalisation in DNNs is one of the stochastic optimisers. A *minimum* in the parameter space is, informally, a region with low training loss. One can distinguish between “good” minima and “bad” minima based on whether or not the functions associated with that region generalise well. From this perspective, a natural candidate for the source of inductive bias is the combination of the optimiser and the loss function, since they determine which minimum is found.

A well-known result from [30] shows that DNNs trained with small batch stochastic gradient descent (SGD) generalise better than identical models trained with large batch SGD (by $\sim 5\%$). Small batch SGD also finds ‘flatter’ minima (i.e. minima with smaller eigenvalues in the Hessian) than large batch SGD, and it has been suggested several times that ‘flatter’ minima lead to better generalisation than sharper minima [30, 31, 32, 33, 34, 35] (however, it was shown in [69] that sharp minima can generalise well). Martin et al. [36] also showed that using SGD with different batch sizes leads to qualitatively different functions being found. However, results from [39, 70, 71] suggest that better generalisation with small batch SGD may be caused by the fact that the number of optimization

steps per epoch decreases when the batch size increases, and showed that a similar effect can be created by increasing the learning rate, or by overtraining (i.e. by continuing to train after 100% accuracy has been reached). It was also pointed out in [39] that overtraining does not negatively impact generalisation. This also contradicts previous theoretical studies that suggested that SGD may control the capacity of the models by limiting the number of parameter updates [72].

Significant effort has been spent on extracting properties of a trained neural network that could be used to explain generalisation. Empirical works have identified properties such as flatness [30, 32], low frequency [73], and sensitivity to changes in the inputs [74, 75]. Theoretical works have also looked at sensitivity to perturbations (whether in inputs or weights), and tried to use it to explain the generalisation performance either using a PAC-Bayesian analysis [76, 77, 78], or a compression approach [79, 80]. A very comprehensive review of this line of work empirically finds that the PAC-Bayesian sensitivity approaches seem the most promising [33].

Several theoretical works have focused on proving generalisation bounds of models trained with SGD under different assumptions. Soudry et al. [81] showed that SGD finds the max-margin solution in unregularised logistic regression, whilst it was shown in [72] that overparameterised DNNs trained with SGD avoid over-fitting on linearly separable data. Recently, [82] proved agnostic generalization bounds of SGD-trained neural networks. While an impressive theoretical achievement, no empirical test of the tightness of the bounds is performed. More recent work [83] suggests that gradient descent performs a hidden regularisation in normalised weights, but other analyses suggests that such implicit regularisation may be very hard to prove in a more general setting for SGD [84]. So again, there is no consensus emerging from this direction of inquiry.

C Gaussian Processes

Gaussian Processes (GPs) are a Bayesian machine learning framework, and are equivalent to DNNs trained by Algorithm 2 with an i.i.d. Gaussian prior over their parameters, in the limit of infinite width [61, 64]. It is easier to extract confidences about a prediction from GPs than from DNNs, because of their Bayesian nature, and they are also non-parametric – so one never has to worry

about whether they will be expressive enough [85].

C.1 Definition and Example

So what is a Gaussian process? We will first explain what they do in the discrete case, and forget about the training process. Let $\{x_i\}_{i=1}^m \in \mathcal{X}$ be a set of inputs, where \mathcal{X} is the input space. We want to associate a probability measure with each possible output $\{y(x_i)\}_{i=1}^m \in \mathcal{Y}$, where \mathcal{Y} is the output space. We will first want to define a measure of correlation between the points in $\{x_i\}_{i=1}^m$ – large correlation between x_i and x_j implies that outputs where $y(x_i)$ is similar to $y(x_j)$ are likely. This is called a kernel, *and as should be clear from the example above, is not dependent on observed $y(x_i)$* . The kernel is (part of) our *prior*. As a result, the probability weight assigned to outputs depends on the kernel.

$$K(x, x') = \mathbb{E}[f(x), f(x')] \quad (23)$$

In other words, K is the $m \times m$ correlation matrix. for the inputs. We then assign a probability measure to possible outputs: $y \in \mathcal{Y}$.

$$p(Y = y_i) = \exp \left((y_i)^T K(x_i, x_j) y_j \right) \quad (24)$$

where the summations over i, j are implied. It is possible to add a mean function $\mu(x)$ – this is done by replacing y_i with $(y_i - \mu_i)$. $\mu(x)$ and $K(x, x')$ together are our prior. For ease of calculation however, we will not include $\mu(x)$ to start with. And, apart from the training, that is all there is.

C.2 How to train your Gaussian Process

Now we have most of the required definitions, and an example (see Figure 21) we can explain how it works in general. Consider a training set with m examples, $\mathcal{D} = \{x_i, y_i\}_{i=1}^m$, where $x_i \in \mathcal{X}$ are the inputs and $y_i \in \mathcal{Y}$ the outputs. Now consider a test set $\{\tilde{x}_i\}_{i=1}^n \in \mathcal{X}$ with n inputs. We wish to assign probabilities to the possible outputs, $\{\tilde{y}_i\}_{i=1}^n$. To do this, we will first rewrite the definition of the kernel (Equation (23)) for the training and test data:

$$K = \begin{bmatrix} K(x_i, x_j) & K(x_i, \tilde{x}_j) \\ K(\tilde{x}_i, x_j) & K(\tilde{x}_i, \tilde{x}_j) \end{bmatrix},$$

where the kernel K must be defined for all elements in \mathcal{X} . We will also introduce a mean function $\mu = (\mu(x), \mu(\tilde{x}))^T$. See Equation (24) for the definition. We can then use our expression for the probability measure, Equation (24), to calculate the distribution on $\tilde{\mathcal{Y}}$:

$$p(\tilde{\mathcal{Y}} = \{\tilde{y}_i\}_{i=1}^n | \mathcal{Y} = \{y_i\}_{i=1}^m) = \frac{1}{P(\mathcal{Y} = \{y_i\}_{i=1}^m)} \mathcal{N}^n(\mu', \mathcal{K}') \quad (25)$$

where $\mathcal{N}^n(0, \mathcal{K}')$ is an n-dimensional normal distribution over the variables in $\tilde{\mathcal{Y}}$, with mean $\mu_i = \mu(\tilde{x}_i) + K(\tilde{x}_i, x_k)K^{-1}(x_k, x_l)(x_l - \mu(x_l))$ covariance $K'_{ij} = K(\tilde{x}_i, \tilde{x}_j) + K(\tilde{x}_i, x_k)K^{-1}(x_k, x_l)K(x_l, \tilde{x}_j)$. This gives an analytic distribution, from which we must sample to obtain our estimates for $\tilde{y}_{i=1}^n$

C.3 Gaussian Processes as Infinite width DNNs

It was first shown in [64] that fully-connected feedforward DNNs (and valid for generic point-wise nonlinear functions found between the layers), with an i.i.d. prior over its parameters are equivalent to a Gaussian Process (GP) in the limit of infinite width. The essence of the proof is in the central limit theorem – that the output of each layer (before the activation is applied) is a Gaussian in the limit of infinite width (with a covariance matrix dependent on the layer architecture). Recent work [65] showed that this is also true for many further architectures under certain conditions (for example, CNNs, LSTMs and ResNets satisfy these conditions). We refer the reader to Section 2.3 of [64] for the correct kernel matrix for fully-connected neural networks.

In our experiments we use the correspondence to estimate $P_{Bayes}(f|D)$. For any given DNN N , we compute the associated kernel K (i.e. the kernel which makes the GP equivalent to N with an i.i.d. Gaussian prior over its parameters). In the case of fully-connected DNNs this can be calculated analytically, otherwise we calculate it by sampling the outputs pre-final layer to obtain an estimate for $\mathbb{E}(x, x')$, and thus $K(x, x')$. Extracting the thresholded outputs is loss function dependent, and we discuss how this is done with MSE loss and C-E loss in the following section.

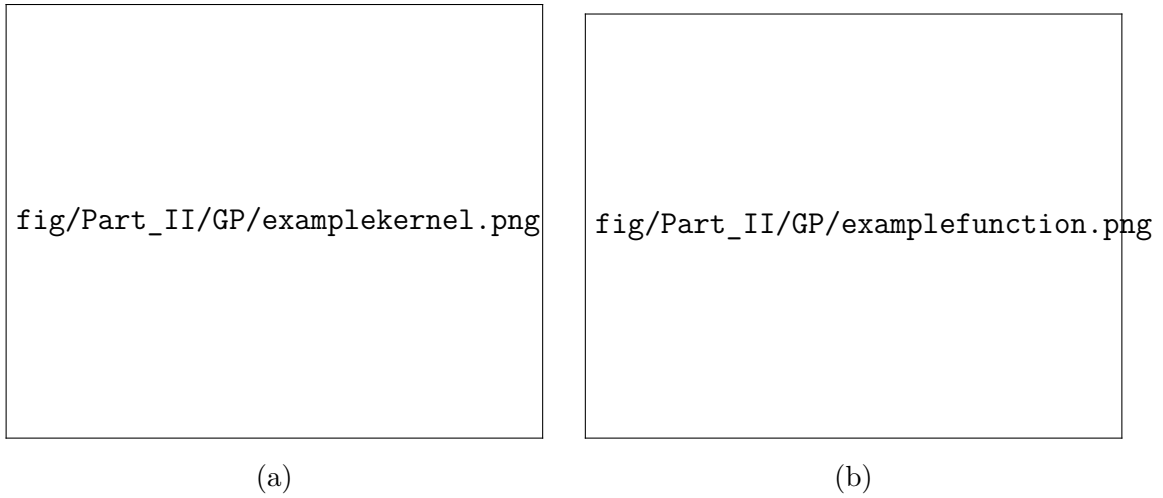


Figure 21: This figure shows an example of a Gaussian process. Let $\mathcal{X} = \{1, 2\}$ and let $K(x, x') = \exp(-(x - x')^2 / (2\sigma^2))$ with $\sigma = 1$. Clearly this kernel assigns higher prior probability to outputs where x and x' take similar values of y . (a) shows this kernel applied to the expression for $p(y)$, Equation (24). The red line shows the $y(1) = 1$ line, and we can imagine fixing that value, and then conditioning on it to produce the measure $p(y(2)|y(1))$. The result of this is shown in (b), where $y(1)$ is fixed, and $y(2)$ is allowed to vary. We imagined sampling from the red curve in (a) to produce $y(2)$, which would be a possible (although not a sensible guess). In practice, many samples would be taken to build up a distribution, and the average of that would usually be taken to be the best guess.

C.4 Notes on the approximation for DNNs with MSE loss and DNNs with C-E loss

With MSE loss, we consider the space of functions to be the space of real-valued functions on \mathcal{X} .

We then use a Gaussian process with a Gaussian likelihood defined as

$$P(D|f) = \prod_{i=1}^m \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(f(x_i) - y_i)^2\right),$$

where σ^2 is the variance.

This likelihood allows to analytically compute the exact posterior. In the experiments in the paper we therefore sampled from this exact posterior, to get values of $f(x) \in \mathbb{R}$ at the test points, which were then thresholded at 0.5 to find the predicted class label. We have chosen a small value of the variance $\sigma^2 = 0.002$, to simulate SGD achieving a small value of the MSE loss. Note that under the standard assumption that training and test instances come from the same distribution, this algorithm may be considered to be not fully Bayesian in the sense that the training and test labels are treated differently (Gaussian likelihood in training points versus Bernoulli likelihood at test points).

This formulation uses a Gaussian process prior over a space of real-valued “latent” functions¹⁶ \tilde{f} on \mathcal{X} . We then use the previously defined 0-1 likelihood where f is a binary-valued function (taking values in $\{0, 1\}$), and f is related to \tilde{f} via a Heaviside linking function, defined as

$$f(x) = \begin{cases} 1 & \text{if } \tilde{f}(x) > 0 \\ 0 & \text{otherwise} . \end{cases}$$

This likelihood makes the posterior of the GP analytically intractable. We therefore use a standard approximation technique known as expectation propagation (EP) [86]. This approximates the posterior over the latent function as a Gaussian, which we can sample from and then use the Heaviside

¹⁶In the GP limit of DNNs, this latent function corresponds to the real-valued pre-activations of the last layer of the neural network, before a final non-linearity (like softmax) is applied.

function to predict the binary labels at the test points. We use this technique to approximate posterior probabilities of the GP with 0-1 likelihood, by sampling. There is a second method which allows for probabilities such as $P(D)$ to be extracted directly, detailed in [87]. We refer to this as the GP/EP estimate in our graphs (to distinguish it from GP/EP sampling). See Figure 24 for the differences between these two methods.

C.5 Errors in the GP/EP and GP approximations

In this section we compare the behaviour of the GP approximations and SGD with different loss functions. Evidence in [1] suggests that the GP/EP approximation underestimates probabilities by a power law. As treated briefly in the above appendix, there are subtle differences in the way the GP approximation with MSE loss and the GP/EP approximation with C-E loss calculates their respective estimates for $P_{Bayes}(f|D)$. However, the thresholded function itself will classify images irrespective of the loss function used, so it is expected that the estimates should correlate¹⁷. It is clear from Figure 22a that they do correlate as a power law, as expected. Further work would be required to quantify the extent to which this behaviour is due to differences in the loss function compared to differences in the behaviour of the approximations.

One final note about the EP approximation. In our experiments, we use two sorts of likelihood functions in the GP/EP approximation: a Heaviside 0-1 function (for all but the experiments with the LSTM), and a Bernoulli likelihood (for experiments with the LSTM, due to unexplained convergence issues). To test the differences between the results (which we hope to be small), we tested 100 randomly selected functions with $\langle \epsilon_G \rangle$ ranging from 0% to 100% on Binarised MNIST. Of these, the average difference between the results as a percentage of the magnitude of the log probabilities was 0.013%, and the maximum was 0.58%.

¹⁷If this is not true, this suggests that arguments made in Section 11 are incomplete, as they do not suggest that the loss function itself should have a large impact on measures such as $P_{Bayes}(f|D)$ or $P_{opt}(f|D)$

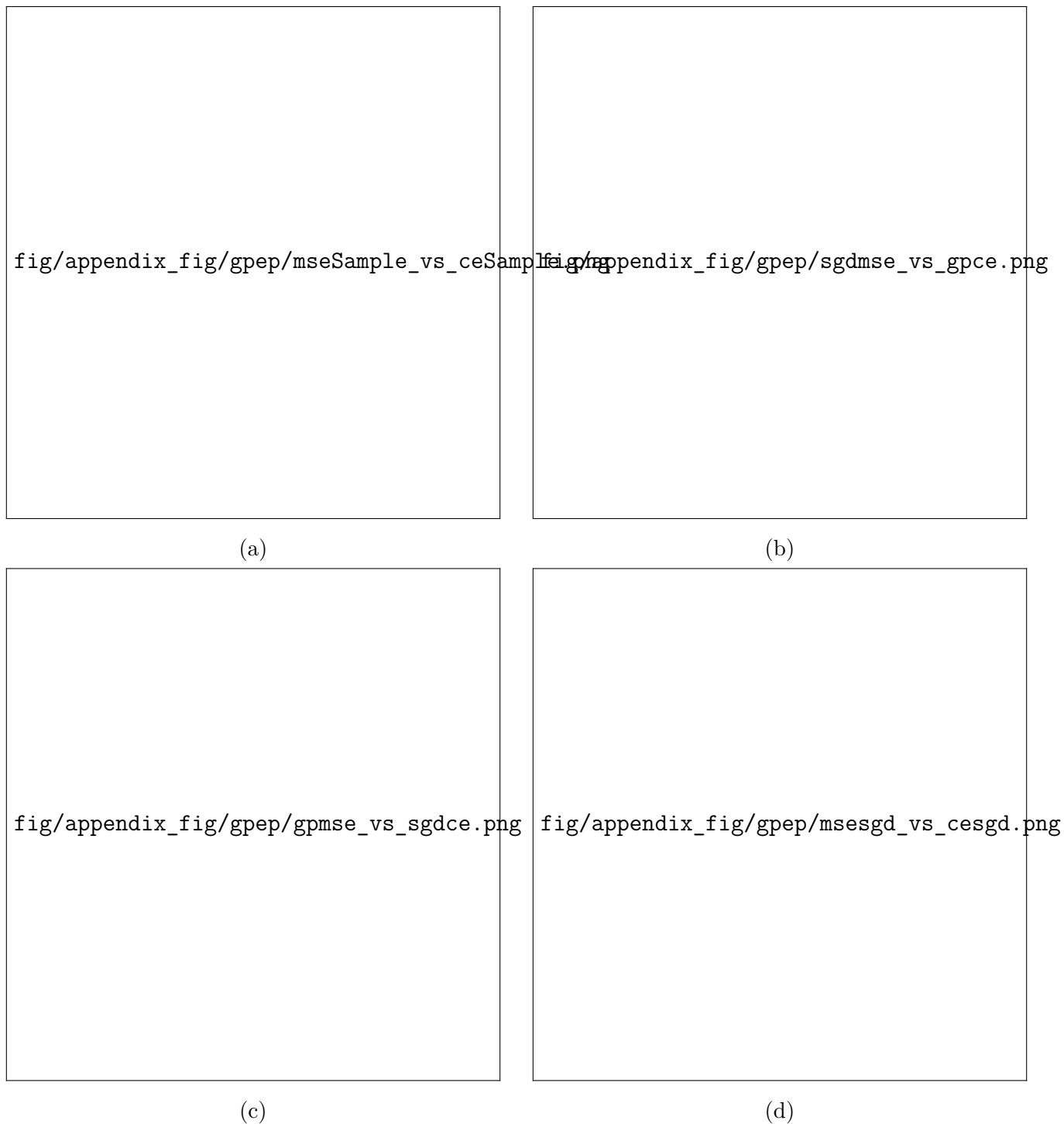


Figure 22: In (a) we correlate the functions SGD with cross-entropy finds with those SGD with MSE finds. In (b) we compare the analogous situation with the Gaussian Processes approximation – the GP/EP approximation with the GP MSE approximation. The functions to the right of the blue dotted lines make up 90% of the total probability.

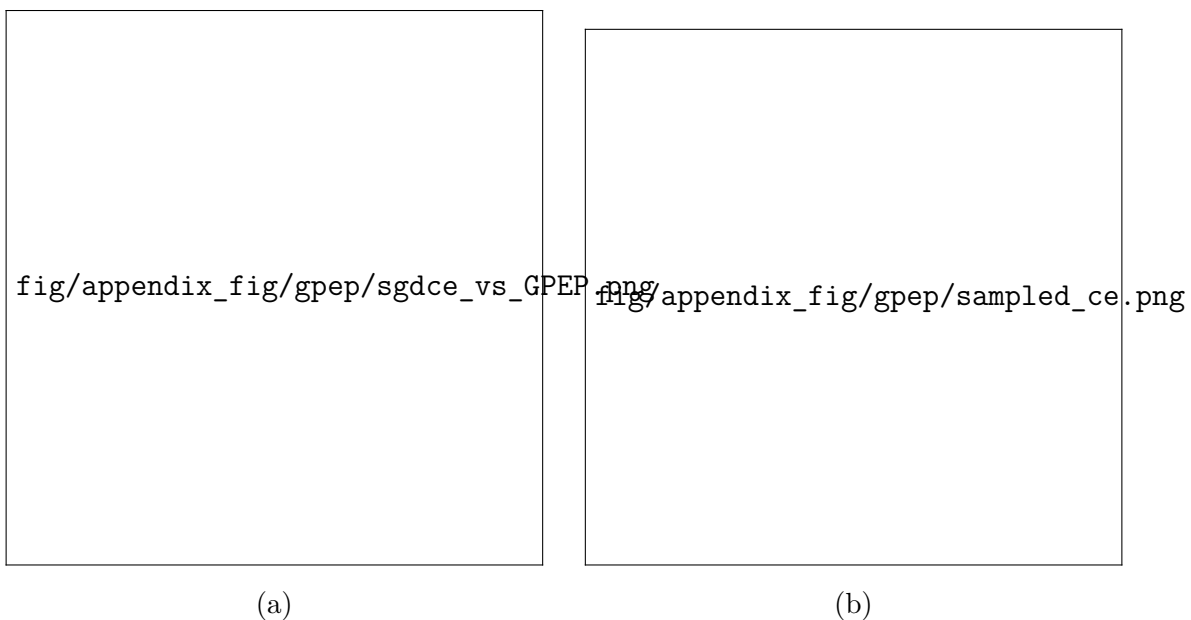


Figure 23: (a) and (b) show Experiments 1 and 3 with C-E loss. They show qualitatively similar results to the same experiment with MSE loss (Figure 7), but due to the GP/EP approximation, the y axes appear to have been ‘squashed’.



Figure 24: We compare the GP/EP $\log(p)$ approximation with GP/EP sampling. We use parameters from Definition 12.1 and Binarised MNIST. We consider functions found by Adam in 10^6 samples, and use both the GP/EP $\log(p)$ approximation and GP/EP sampling to calculate the probability. We use both these methods in our experiments, but as is clear from the above figure, there is not much difference between them.



Figure 25: Examples of the renormalised EP approximation applied to Experiment 1, with parameters from Definition 12.1, except: (a) no difference (b) overtraining by 64 epochs at 100% accuracy, (c) RMSprop, (d) batch size 32, (e) batch size 512, (f) batch size 512, 4x learning rate.

C.6 The ‘Renormalised EP’ approximation

We computed the gradient of the line of best fit from Figure 22a (the GP approximation vs the GP/EP approximation for functions in $P_{Bayes}(f|D)$ with Binarised MNIST), and, under the assumption that the uncontrolled errors produced by the EP approximation cause the line of best fit to deviate from $y = x$, we calculated a correction to the EP approximation (only valid for Binarised MNIST). Clearly the lines of best fit are much closer to $y = x$ than they are when unnormalised (compare the renormalised graphs in Figure 25 to Figure 11 and Figure 14).

However, it is worth noting that there is a hidden assumption: that the line of best fit in Figure 22a should be $y = x$. It seems plausible that the loss functions should not cause anything other than scatter around the line, but this is not yet well understood. For completion, we provide Experiments 1 and 3 with C-E loss, in Figure 23.

D Notes on Experiment 2

Experiment 2 calculates $P_{Bayes}(f|D)$ for functions in the posterior in the full range of generalisation error. In several implementations of it, on a variety of architectures (Fully connected, Convolutional and LSTM) and datasets (MNIST, FashionMNIST and the IMDB movie review dataset) it exhibits near-linear behaviour on a log plot, with fairly significant scatter. Here, we comment on this behaviour.

We tested how independent the classification of each image in the test set was using the following method: for each pair of images I, J in the test set, calculate the probability that $P(f(I) = 1)$ and $P(f(I) = 0)$ with GP sampling where $f(I)$ is the classification of the image I . If $P(f(I) = 1) > P(f(I) = 0)$ then calculate $c = |P(f(I) = 1|f(J) = 1) - P(f(I) = 1|f(J) = 0)|$ (and the converse if the equality is not satisfied). We do this because the generalisation error is so low, typically one of $P(f(I) = 1)$ or $P(f(I) = 0)$ is > 0.9 . If c is close to 0, then the variables are close to independent. We sampled from the GP posterior 10^7 times (for a visualisation of this data, see Figure 7c), and calculated this all pairs of images in the test set (unless one image never registered an error). We took an average of this measure, and obtained 0.00477 (compare to a maximum

value of 1 and minimum of 0) – indicating that the images are close to independently distributed. This is expected, as the assumption is that images in the test set are drawn independently from distribution over possible images (for a larger test set, this assumption breaks down).

Thus, we associated each image $x_{i=0}^n$ in the test set with probability p_i of being classified incorrectly conditioning on \mathcal{D} (note that $(1/n) \sum_i p_i = \langle \epsilon_G \rangle$). Using our result that the classification of each image is almost independent, the product distribution over these Bernoulli variables should approximate the probability distribution over functions on the test set. This distribution is the *Poisson binomial distribution* [88]. Using the same data as above, we calculated the probability of classifying each image incorrectly to approximate p_i on the test set. It is clear from Figure 26a that they vary over many orders of magnitude.

There exists a Chernoff type bound for the the binomial poisson distribution:

$$p(\epsilon) \leq P[\mathcal{E} > \epsilon] \leq \exp(-\epsilon(\log(\epsilon/\langle \epsilon_G \rangle) - 1) - \langle \epsilon_G \rangle) \quad (26)$$

where $p(\epsilon)$ is the pmf, $P[\mathcal{E} > \epsilon]$ is the cmf, and $\langle \epsilon_G \rangle = (1/n) \sum_i p_i$ is the mean. On a log scale, this means

$$\log_{10}(p(\epsilon)) \leq [-\epsilon(\log(\epsilon/\langle \epsilon_G \rangle) - 1) - \langle \epsilon_G \rangle] \log_{10}(e) \quad (27)$$

Which indicates an exponential like drop-off for $\epsilon \geq \langle \epsilon_G \rangle$ (similar to what is observed in implementations of experiment 2). In reality though, this is an upper bound and the actual distribution is strongly dependent on the p_i values. This bound appears to be non-vacuous only for errors much larger than $\langle \epsilon_G \rangle$ – which we often deal with in our experiments.



(a)



(b)

Figure 26: (a) shows the GP estimate for the values of p_i for Binarised MNIST, a training set of size 10000 and test set of size 100 and the MSE loss function. Other parameters (where relevant) are to be found in Definition 12.1. Clearly these vary over many orders of magnitude. The frequencies were cut off at 10^{-7} (so functions in that bin have $p_i \leq 10^{-7}$) to avoid finite size effects. (b) uses the same dataset and architecture (except the loss function where C-E and MSE are used), and compares $P(\text{Image } i \text{ incorrect})$ with $P(\text{Image } i \text{ incorrect} \cap \text{all other images correct})$. Any probabilities too low to be determined by sampling were set to 10^{-7} . The strong correlation indicates that the probability of any image being classified incorrectly is likely to be independent of the classification of other images.

References

- [1] Guillermo Valle-Pérez, Chico Q Camargo, and Ard A Louis. Deep learning generalizes because the parameter-function map is biased towards simple functions. *arXiv preprint arXiv:1805.08522*, 2018.
- [2] Dingle Kamaludin, Guillermo Valle Pérez, and Ard A Louis. Generic predictions of output probability based on complexities of inputs and outputs. *Scientific Reports (Nature Publisher Group)*, 10(1), 2020.
- [3] Kamaludin Dingle, Chico Q Camargo, and Ard A Louis. Input-output maps are strongly biased towards simple outputs. *Nature communications*, 9(1):761, 2018.
- [4] Chris Mingard, Joar Skalse, Guillermo Valle-Pérez, David Martínez-Rubio, Vladimir Mikulik, and Ard A Louis. Neural networks are a priori biased towards boolean functions with low entropy. *arXiv preprint arXiv:1909.11522*, 2019.
- [5] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.
- [6] Alan L Hodgkin and Andrew F Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4):500–544, 1952.
- [7] Pamela McCorduck and Cli Cfe. *Machines who think: A personal inquiry into the history and prospects of artificial intelligence*. CRC Press, 2004.
- [8] Herbert Alexander Simon. *The shape of automation for men and management*, volume 13. Harper & Row New York, 1965.
- [9] Jonathan Schaeffer. *One jump ahead: challenging human supremacy in checkers*. Springer Science & Business Media, 2013.
- [10] John McCarthy. Artificial intelligence: a paper symposium: Professor sir james lighthill, frs. artificial intelligence: A general survey. in: Science research council, 1973, 1974.

- [11] Ehud Y Shapiro. *Inductive inference of theories from facts*. Yale University, Department of Computer Science, 1981.
- [12] Leslie G Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.
- [13] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.
- [14] Marvin Minsky and Seymour A Papert. *Perceptrons: An introduction to computational geometry*. MIT press, 2017.
- [15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.
- [16] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.
- [17] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.
- [18] Hugo Touvron, Andrea Vedaldi, Matthijs Douze, and Hervé Jégou. Fixing the train-test resolution discrepancy: Fixefficientnet. *arXiv preprint arXiv:2003.08237*, 2020.
- [19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- [20] Jonathan J Tompson, Arjun Jain, Yann LeCun, and Christoph Bregler. Joint training of a convolutional network and a graphical model for human pose estimation. In *Advances in neural information processing systems*, pages 1799–1807, 2014.
- [21] Tomáš Mikolov, Anoop Deoras, Daniel Povey, Lukáš Burget, and Jan Černocký. Strategies for training large scale neural network language models. In *2011 IEEE Workshop on Automatic Speech Recognition & Understanding*, pages 196–201. IEEE, 2011.

- [22] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.
- [23] T Ciodaro, D Deva, JM De Seixas, and D Damazio. Online particle detection with neural networks based on topological calorimetry information. In *Journal of physics: conference series*, volume 368, page 012030. IOP Publishing, 2012.
- [24] Lei Zhang, Shuai Wang, and Bing Liu. Deep learning for sentiment analysis: A survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(4):e1253, 2018.
- [25] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.
- [26] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dhharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.
- [27] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.
- [28] Boris Hanin. Universal function approximation by deep neural nets with bounded width and relu activations. *arXiv preprint arXiv:1708.02691*, 2017.
- [29] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.
- [30] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *CoRR*, abs/1609.04836, 2016.

- [31] Stanislaw Jastrzebski, Zachary Kenton, Devansh Arpit, Nicolas Ballas, Asja Fischer, Yoshua Bengio, and Amos J Storkey. Finding flatter minima with sgd. In *ICLR (Workshop)*, 2018.
- [32] Sepp Hochreiter and Jurgen Schmidhuber. Flat minima. *Neural Computation*, 9(1):1–42, 1997.
- [33] Yiding Jiang, Behnam Neyshabur, Hossein Mobahi, Dilip Krishnan, and Samy Bengio. Fantastic generalization measures and where to find them. *arXiv preprint arXiv:1912.02178*, 2019.
- [34] Lei Wu, Zhanxing Zhu, et al. Towards understanding generalization of deep learning: Perspective of loss landscapes. *arXiv preprint arXiv:1706.10239*, 2017.
- [35] Yao Zhang, Andrew M Saxe, Madhu S Advani, and Alpha A Lee. Energy–entropy competition and the effectiveness of stochastic gradient descent in machine learning. *Molecular Physics*, 116(21-22):3214–3223, 2018.
- [36] Charles H Martin and Michael W Mahoney. Implicit self-regularization in deep neural networks: Evidence from random matrix theory and implications for learning. *arXiv preprint arXiv:1810.01075*, 2018.
- [37] Leonid Anatolevich Levin. Laws of information conservation (nongrowth) and aspects of the foundation of probability theory. *Problemy Peredachi Informatsii*, 10(3):30–35, 1974.
- [38] M. Li and P.M.B. Vitanyi. *An introduction to Kolmogorov complexity and its applications*. Springer-Verlag New York Inc, 2008.
- [39] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Advances in Neural Information Processing Systems*, pages 1731–1741, 2017.
- [40] Leah Henderson. The problem of induction. 2018.
- [41] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.
- [42] Ray J Solomonoff. A formal theory of inductive inference. part i. *Information and control*, 7(1):1–22, 1964.

- [43] Jürgen Schmidhuber. Discovering problem solutions with low kolmogorov complexity and high generalization capability. In *MACHINE LEARNING: PROCEEDINGS OF THE TWELFTH INTERNATIONAL CONFERENCE*. Citeseer, 1994.
- [44] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.
- [45] Michael Sipser. Introduction to the theory of computation. *ACM Sigact News*, 27(1):27–29, 1996.
- [46] Andrei N Kolmogorov. Three approaches to the quantitative definition of information'. *Problems of information transmission*, 1(1):1–7, 1965.
- [47] David Blackwell and Lester Dubins. Merging of opinions with increasing information. *The Annals of Mathematical Statistics*, 33(3):882–886, 1962.
- [48] Samuel Rathmanner and Marcus Hutter. A philosophical treatise of universal induction. *Entropy*, 13(6):1076–1136, 2011.
- [49] Gregory J Chaitin. A theory of program size formally identical to information theory. *Journal of the ACM (JACM)*, 22(3):329–340, 1975.
- [50] Jan Leike and Marcus Hutter. On the computability of solomonoff induction and knowledge-seeking. In *International Conference on Algorithmic Learning Theory*, pages 364–378. Springer, 2015.
- [51] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.
- [52] Matus Telgarsky. Representation benefits of deep feedforward networks. *arXiv preprint arXiv:1509.08101*, 2015.
- [53] Diederik P Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. In *Proc. 3rd Int. Conf. Learn. Representations*, 2014.

- [54] Carl Edward Rasmussen. Gaussian processes in machine learning. In *Summer School on Machine Learning*, pages 63–71. Springer, 2003.
- [55] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. Object recognition with gradient-based learning. In *Shape, contour and grouping in computer vision*, pages 319–345. Springer, 1999.
- [56] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [57] Larry Medsker and Lakhmi C Jain. *Recurrent neural networks: design and applications*. CRC press, 1999.
- [58] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [59] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.
- [60] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.
- [61] Greg Yang and Hadi Salman. A fine-grained spectral perspective on neural networks. *arXiv preprint arXiv:1907.10599*, 2019.
- [62] David A McAllester. Pac-bayesian model averaging. In *Proceedings of the twelfth annual conference on Computational learning theory*, pages 164–170, 1999.
- [63] Roman Novak, Lechao Xiao, Jiri Hron, Jaehoon Lee, Alexander A Alemi, Jascha Sohl-Dickstein, and Samuel S Schoenholz. Neural tangents: Fast and easy infinite neural networks in python. *arXiv preprint arXiv:1912.02803*, 2019.

- [64] Jaehoon Lee, Yasaman Bahri, Roman Novak, Samuel S Schoenholz, Jeffrey Pennington, and Jascha Sohl-Dickstein. Deep neural networks as gaussian processes. *arXiv preprint arXiv:1711.00165*, 2017.
- [65] Greg Yang. Tensor programs i: Wide feedforward or recurrent neural networks of any architecture are gaussian processes. In *NeurIPS 2019*, December 2019.
- [66] David Krueger, Nicolas Ballas, Stanislaw Jastrzebski, Devansh Arpit, Maxinder S Kanwal, Tegan Maharaj, Emmanuel Bengio, Asja Fischer, and Aaron Courville. Deep nets don’t learn via memorization. 2017.
- [67] Thomas M Cover and Joy A Thomas. *Elements of information theory*. John Wiley & Sons, 2012.
- [68] David JC MacKay and David JC Mac Kay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.
- [69] Laurent Dinh, Razvan Pascanu, Samy Bengio, and Yoshua Bengio. Sharp minima can generalize for deep nets. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1019–1028. JMLR. org, 2017.
- [70] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.
- [71] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don’t decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.
- [72] Alon Brutzkus, Amir Globerson, Eran Malach, and Shai Shalev-Shwartz. Sgd learns over-parameterized networks that provably generalize on linearly separable data. *arXiv preprint arXiv:1710.10174*, 2017.
- [73] Nasim Rahaman, Aristide Baratin, Devansh Arpit, Felix Draxler, Min Lin, Fred A Hamprecht, Yoshua Bengio, and Aaron Courville. On the spectral bias of neural networks. *arXiv preprint arXiv:1806.08734*, 2018.

- [74] David Krueger, Nicolas Ballas, Stanislaw Jastrzebski, Devansh Arpit, Maxinder S. Kanwal, Tegan Maharaj, Emmanuel Bengio, Asja Fischer, Aaron Courville, Simon Lacoste-Julien, and Yoshua Bengio. A closer look at memorization in deep networks. *Proceedings of the 34th International Conference on Machine Learning (ICML'17)*, 2017.
- [75] Roman Novak, Yasaman Bahri, Daniel A. Abolafia, Jeffrey Pennington, and Jascha Sohl-Dickstein. Sensitivity and generalization in neural networks: an empirical study. In *International Conference on Learning Representations*, 2018.
- [76] Peter L Bartlett, Dylan J Foster, and Matus J Telgarsky. Spectrally-normalized margin bounds for neural networks. In *Advances in Neural Information Processing Systems*, pages 6240–6249, 2017.
- [77] Gintare Karolina Dziugaite and Daniel M. Roy. Computing nonvacuous generalization bounds for deep (stochastic) neural networks with many more parameters than training data. In *Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence, UAI 2017, Sydney, Australia, August 11-15, 2017*, 2017.
- [78] Behnam Neyshabur, Srinadh Bhojanapalli, and Nathan Srebro. A PAC-bayesian approach to spectrally-normalized margin bounds for neural networks. In *International Conference on Learning Representations*, 2018.
- [79] Sanjeev Arora, Rong Ge, Behnam Neyshabur, and Yi Zhang. Stronger generalization bounds for deep nets via a compression approach. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 254–263. PMLR, 10–15 Jul 2018.
- [80] Wenda Zhou, Victor Veitch, Morgane Austern, Ryan P. Adams, and Peter Orbanz. Non-vacuous generalization bounds at the imagenet scale: a PAC-bayesian compression approach. In *International Conference on Learning Representations*, 2019.
- [81] Daniel Soudry, Elad Hoffer, Mor Shpigel Nacson, Suriya Gunasekar, and Nathan Srebro. The

implicit bias of gradient descent on separable data. *The Journal of Machine Learning Research*, 19(1):2822–2878, 2018.

- [82] Zeyuan Allen-Zhu, Yuanzhi Li, and Yingyu Liang. Learning and generalization in overparameterized neural networks, going beyond two layers. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 6158–6169. Curran Associates, Inc., 2019.
- [83] Tomaso Poggio, Qianli Liao, and Andrzej Banburski. Complexity control by gradient descent in deep networks. *Nature Communications*, 11(1):1–5, 2020.
- [84] Assaf Dauber, Meir Feder, Tomer Koren, and Roi Livni. Can implicit bias explain generalization? stochastic convex optimization as a case study. *arXiv preprint arXiv:2003.06152*, 2020.
- [85] Christopher KI Williams and Carl Edward Rasmussen. *Gaussian processes for machine learning*, volume 2. MIT press Cambridge, MA, 2006.
- [86] Thomas P Minka. Expectation propagation for approximate bayesian inference. *arXiv preprint arXiv:1301.2294*, 2013.
- [87] Chris Mingard, Joar Skalse, Guillermo Valle-Pérez, and Ard A Louis. Is sgd a bayesian sampler? well, almost. 2020.
- [88] Yuan H Wang. On the number of successes in independent trials. *Statistica Sinica*, pages 295–312, 1993.