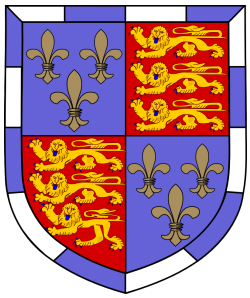
UNIVERSITY OF CAMBRIDGE

# Automated Architecture Search for Bayesian Neural Networks



**Michael Hutchinson**

Supervisor: Dr. R.E. Turner

Department of Engineering
University of Cambridge

This dissertation is submitted for the degree of
*Master of Engineering*

Christ's College

May 2019

# Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 12,000 words including appendices, footnotes, tables and equations and has fewer than 50 pages.

Michael Hutchinson
May 2019

# Acknowledgements

Not sure if I need to put stuff in here - whether this is a technical matter or not.

People I might mention:

- Siddharth
- Marcin
- Thang

# Abstract

This is where you write your abstract ...

# Table of contents

# List of figures

# List of tables

# Nomenclature

**Acronyms / Abbreviations**

| | |
|---|---|
| BNN | A Bayesian Neural Network |
| VI | Variational Inference |

# Notes

# Chapter 1

# Introduction

Standard neural networks have revolutionised the machine learning field, providing orders of magnitude better performance over previous methods in a wide variety of applications, in particular in tasks with significant quantities of data. A key part of the success of these methods has come from designing bespoke architectures which work well with deep learning optimisation methods. The deign of these networks has two distinct parts: The choice of building blocks for the networks, and the method of connecting these building blocks together.

The design of new building blocks has been the main driver in performance of these networks. Generally these have increase performance in one of two ways: by providing a significant reduction on the number of parameters of the network while maintaining the networks level of expressibility on a given task. Since these layers are usually a constrained form of a fully connected layer, this could also be viewed as providing a form of "hard regularisation" in the network by building in invariance in sensible ways. Examples of this include conventional layers, building in local connectivity and translation invariance, recurrent networks, building in time invariance or

Add a third option?

Add in some references

. Designing these new layers has yet to be automated, and given the complexity involved in doing so, it is unlikely this will happen in the near future. The other way new building blocks have improved performance in neural networks is by improving the optimisation of neural networks with gradient based methods. Examples of this include the use of the RELU unit, residual layers, skip connections and highway connections.

citations

The other part in designing neural networks is the configuration in which these building blocks are connected together, and the various hyperparemeters associated with them. The configuration of these blocks defines the "predictive power"

better way of phrasing?

of the network, and encodes a specific belief about how the outputs should be predicted from the inputs. This configuration can play a significant role in the performance of neural networks,

cite a chain of papers that are just reconfigurations? E.g. GRU-> LSTM?

shows that this is a significant factor.

The point of architecture search therefore is to explore some defined space of possible architectures and find architectures that perform best. This is a non-trivial process. Usually the search space is extremely large, and individual architectures expensive to evaluate. Traditionally this search has been done by hand utilising human intuition. This a therefore lead to a situation where significant amounts of research time are spent testing different configurations of neural networks, looking for minor improvement. Initially therefore then benefits of an automated approach to architecture search would be

- A reduction in research time spent directly on searching for incrementally better architectures. An automated procedure could take over the larger part of the menial work on this task, freeing research time for alternative pursuits.
- Reducing the influence of human bias on the search procedure. Human designers of networks have a preference for regular and seemingly ordered structures for neural networks. As can be observed in

  citations

  these are often not optimal architectures. Automated methods would explore the space of architectures without this particular bias.
- Models can be tailored to specific problems. Much of the work done on discovering new architectures is done either on looking at performance over a set of datasets and looking for good performance across them all, or looking at performance on a single dataset

  e.g. CIFAR, ImgNet

  Community over-fitting to specific datasets?

  and transferring architectures discovered on these datasets to other problems. Am automated method with little human interaction required could be used to tailor design networks to specific problems for better performance.

The case for automated architecture search therefore is clear, and there has been some substantial work in this area, which will be discussed later. None of this work however has looked at architecture search in designing Bayesian Neural Networks (BNNs) . This Bayesian interpretation of neural networks has several attractive properties in contrast to regular neural networks. To name a few

- ◇ BNNs provide uncertainty estimates in their predictions. In comparison to standard neural networks which produce point estimates, or can be confidently wrong when using softmax layers in classification, BNNs provide uncertainty in their predictions as a result of their probabilistic formulation. These uncertainty estimates provide significant advantages in making decisions in situations where there is significant risk in making incorrect prediction, such as medicine or finance, or in exploration based tasks such as reinforcement learning.
- ◇ BNNS are more robust to a series of effects that exploit the highly specific configuration of weights in a neural network, for example adviserial attacks. This robustness comes from integrating or sampling across the posterior parameter space.
- ◇ Due to the probabilistic nature of the weights, several techniques that have been developed for probabilistic models that cannot be applied to regular neural networks, such as continual learning, distributed learning or active learning, can be applied to BNNs.

The advantages of BNNs and the lack of work on architecture search on them naturally leads to the topic of this project. The first aim is to provide a characterisation of the performance of fully connected networks, trained under a Variational Inference framework (VI) . The objective of this is to provide a preliminary investigation into how the various hyper-parameters of this particular space affect network performance. Given the very large search space and the inefficiency in exploring this manually, the second aim is to investigate automatic methods of exploring the search space.

# Chapter 2

# Introductory matter

## 2.1 Variational Inference for BNNs

Should I sub-section for clarity/skipping?

The objective of this project its to provide a form for searching for optimal architectures of Bayesian Neural Networks. The methodology of training these however is diverse and can lead to significant differences in performance. Variational Inference was selected as the methodology for training for two reasons, it is stable and produces state of the art results on many problems (Bui et al., 2016) and it is the most scalable method for BNNs at present, able to scale up to at least the size of medium sized convolutions networks (Shridhar et al., 2018).

Unlike the training of standard neural networks, the training of Bayesian Neural Networks (BNNs) is not a straightforward task. In standard Neural Networks we have a series of point weights $\mathbf{w}$ for which we optimise the predictions of some output $\mathbf{y}_i$ for some input data $\mathbf{x}_i$, drawn from a dataset $\mathcal{D} = \{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^N$. In this Bayesian interpretation, we place distributions over the weights of the parameters in the network, and attempt to maximise the likelihood of the data observed. This likelihood can be found by

$$P(\mathbf{y}|\mathbf{x}) = \mathbb{E}_{P(\mathbf{w}|\mathbf{y},\mathbf{x})}\left[P(\mathbf{y}|\mathbf{w},\mathbf{x})\right] \tag{2.1}$$

This calculation is however, with the exception of very simple cases is highly intractable when our prediction for $\mathbf{y}$ comes from a neural network. As such, it is necessary to use approximations in order to be able to compute this likelihood and to be able to maximise it with respect to the weight distributions.

The form utilised in this project is a variational approximation. This simplifies the above expression by introducing a variational distribution, $q_\phi(\mathbf{w})$, over the weights of a form that is more simple to work with. This is developed by (Graves, 2011; Hinton et al.,

1993). Here the log likelihood is maximised, as it is easier

$$\log P(\mathbf{y}|\mathbf{x}) = \int \log\left[P(\mathbf{y},\mathbf{w}|\mathbf{x})\right] d\mathbf{w} \tag{2.2}$$

$$= \int \log\left[P(\mathbf{y}|\mathbf{w},\mathbf{x})P(\mathbf{w})\frac{q_\phi(\mathbf{w})}{q_\phi(\mathbf{w})}\right] d\mathbf{w} \tag{2.3}$$

$$\geq \int q_\phi(\mathbf{w}) \log\left[\frac{P(\mathbf{y}|\mathbf{w},\mathbf{x})P(\mathbf{w})}{q_\phi(\mathbf{w})}\right] d\mathbf{w} \tag{2.4}$$

$$= \int q_\phi(\mathbf{w}) \log\left[P(\mathbf{y}|\mathbf{w},\mathbf{x})\right] d\mathbf{w} - \int q_\phi(\mathbf{w}) \log\left[\frac{q_\phi(\mathbf{w})}{P(\mathbf{w})}\right] d\mathbf{w} \tag{2.5}$$

$$= \mathbb{E}_{q_\phi(\mathbf{w})}\left[\log P(\mathbf{y}|\mathbf{w},\mathbf{x})\right] - \mathcal{D}_{KL}\left(q_\phi(\mathbf{w})\|P(\mathbf{w})\right) \tag{2.6}$$

Where 2.4 uses Jensen's inequality and $\mathcal{D}_{KL}(\cdot\|\cdot)$ is the KL-divergence. These terms are commonly called the reconstruction loss and the prior fit terms. This bound is known as the *Variational Free Energy* or the *Expected Lower Bound (ELBO)*. Two alternative interpretations of this for exist. First as a *minimum descriptive length* (Hinton et al., 1993) method for training neural networks, by minimising the amount of information encoded in the weights. Second it can be shown that minimising this lower bound on the log likelihood is equivalent to minimising the KL divergence $\mathcal{D}_{KL}\left(q_\phi(\mathbf{w})\|P(\mathbf{w}|\mathbf{x},\mathbf{y})\right)$, i.e. the distance between the posterior weight distribution and the variational distribution.

These terms however ar not always tractable. Using (Blundell et al., 2015) we approximate the terms with a Monte Carlo estimate, drawing samples from $q_\phi(\mathbf{w})$

$$\mathbb{E}_{q_\phi(\mathbf{w})}\left[\log P(\mathbf{y}|\mathbf{w},\mathbf{x})\right] - \mathcal{D}_{KL}\left(q_\phi(\mathbf{w})\|P(\mathbf{w})\right)$$

$$\approx \frac{1}{N}\sum_{i=1}^{N} \log\left[P(\mathbf{y}|\mathbf{w}_i,\mathbf{x})\right] - \log\left[\frac{q_\phi(\mathbf{w}_i)}{P(\mathbf{w})}\right],\ \mathbf{w}_i \sim q_\phi(\mathbf{w}_i) \tag{2.7}$$

In our experiments 10 samples are used at train time and 100 at test time. Computing $\log P(\mathbf{y}|\mathbf{w}_i,\mathbf{x})$ relies on deciding a form of this property. Here we model this a normal distribution $\mathcal{N}(f_{\mathbf{w}_i}(\mathbf{x}),\sigma_y^2)$ with the parameter $\sigma_y^2$ joint optimised with the weights. An alternative option for this is using a second head to predict the variance. (Blundell et al., 2015) shows then that standard back propagation techniques can be used to train the parameters $\phi$ of the $q_\phi(\mathbf{w})$ distribution and $\sigma_y^2$.

This estimator unfortunately has very high variance. Two things can be one to reduce this. First by constraining our variational distribution $q_\phi(\mathbf{w})$ and prior $P(\mathbf{w})$ to be from the exponential

should this be the whole family, or just Gaussian?

family with no dependence between weights there is a tractable form for the KL divergence between 2 instances. For a pair of $d$-dimensional Normal distributions

$$\begin{aligned}&\mathcal{D}_{KL}\left(\mathcal{N}(\mu_1, \mathbf{\Sigma}_1) \| \mathcal{N}(\mu_2, \mathbf{\Sigma}_2)\right)) \\ &= \frac{1}{2}\left[\log \frac{|\mathbf{\Sigma}_2|}{|\mathbf{\Sigma}_1|} - d + \operatorname{Tr}\left(\mathbf{\Sigma}_1^{-1}\mathbf{\Sigma}_2\right) + (\mu_1 - \mu_2)^T \mathbf{\Sigma}_2^{-1}(\mu_1 - \mu_2)\right]\end{aligned} \tag{2.8}$$

Second by employing the local reparametisation trick of (Kingma et al., 2015) by writing our weights as

$$w_{i,j} = \mu_{i,j} + \sigma_{i,j}\epsilon_{i,j}, \quad \epsilon_{i,j} \sim \mathcal{N}(0, 1) \tag{2.9}$$

Combined these drive down the variance of the approximations significantly, and lead to significantly improved training.

The final improvement used in later experiments is the use of *hyperpriors* on the width of the priors in the network. The width of the prior can have a significant effect on the performance of the network, and adds an additional parameter to optimise over in the search space for architecture search. There is no way rigorous to determine what the optimal width will be a priori beyond intuition. By introducing hyperpriors on the prior width however we can optimise the prior width at train time and avoid having to add this parameter to the search space. We follow the method set out in (Wu et al., 2018). In this case heirachical prior takes the form

$$\mathbf{s} \sim P(\mathbf{s}), \quad \mathbf{w} \sim P(\mathbf{w}|\mathbf{s}) \tag{2.10}$$

Allowing too many degrees of freedom in the hyperpriors can be detrimental, so a single hyperprior is introduced per layer, covering all the weights in that layer (labelled $s_\lambda$ for the hyperprior on layer $\lambda$). The hyperpriors are inverse-Gamma distributed to be conjugate priors to the intendant gaussian distributions of the weights.

$$s_\lambda \sim \text{Inv-Gamma}(\alpha, \beta), \quad \mathbf{w}_\lambda \sim \mathcal{N}(\mathbf{0}, s_\lambda \mathbf{I}) \tag{2.11}$$

Our new ELBO is simply

$$\mathbb{E}_{q_\phi(\mathbf{w})}\left[\log P(\mathbf{y}|\mathbf{w}, \mathbf{x})\right] - \mathcal{D}_{KL}\left(q_\phi(\mathbf{w}) \| P(\mathbf{w}|\mathbf{s})P(\mathbf{s})\right) \tag{2.12}$$

As an approximation, at each update step Type-2 empirical Bayes is used to estimate the MAP solution for the prior width to provide the tightest ELBO.

$$s_\lambda^* = \arg\min_{\mathbf{s}_\lambda}\left[\mathcal{D}_{KL}\left(q_\phi(\mathbf{w}_\lambda) \| P(\mathbf{w}_\lambda|s_\lambda)\right) - \log s_\lambda\right] \tag{2.13}$$

There is a closed form solution to this optimisations

$$s_\lambda^* = \frac{\text{Tr}\left[\boldsymbol{\Sigma}_\lambda^q + \mu_\lambda^q (\mu_\lambda^q)^T\right] + 2\beta}{N_\lambda + 2\alpha + 2} \tag{2.14}$$

Where $N_\lambda$ is the number of weights under the prior $s_\lambda$. (Wu et al., 2018) shows that this method is almost as good as or better than manual tuning of the prior width for the standard UCI datasets.

One drawback of BNNs is their significant cost in training. Compare to regular neural networks they can take significantly longer. One of the main reasons for picking the variational form of BNNs with mean field weight is for the scalability in network size they present. Even these however are orders of magnitude slower than regular neural networks. Significant parts of these costs come from the sampling required during training, and the multiple evaluations of the network for each data-point required to produce good gradient estimates.

## 2.2 Bayesian Optimisation

The body of scientific literature focused on the optimisation of some function $f(\mathbf{x})$ over some set of possibility $\mathcal{A}$ is extensive. Simply

$$\max_{\mathbf{x} \in \mathcal{A} \subset \mathbb{R}^\mathbf{n}} f(x) \tag{2.15}$$

The large part of this assumes that $f(\mathbf{x})$ contains some particular property in combination with the bounds of $\mathcal{A}$, such as convexity, a know mathematical representation, or cheapness to evaluate.

While some problems in machine learning have applications for these techniques, not all do. The assumptions of cheapness is used in the gradient based update methods used for many algorithms, but due to the large number of iterations usually required to converge models, overall training machine learning models become incredibly expensive. This therefore makes the results of trained algorithms comparatively expensive and usually an objective function that one might cast on the results of these, such as finding optimal hyper-parameter setting for an algorithm for accuracy, a highly non-convex function in their inputs. Standard optimisation methods are therefore inappropriate for performing meta-tasks on the hyper-parameters of machine learning algorithms.

Bayesian optimisation is powerful method for finding the maxima or minima of black box functions with expensive objectives with no special structure to the objective. It is also gradient free method, and so can handle situation where the gradients of the objective with

respect to the parameters is intractable. Finally it is possible to handle noisy evaluations of the objective function with this method.

These properties have made Bayesian Optimisation a key method in applications such as clinical trials and finance, and also make them highly suitable to the architecture search problem.

Bayesian optimisation relies on the Bayesian inference of the most likely model ($M$) given the evidence seen ($E$). This can be done via Bayes rule

$$P(M|E) \propto P(E|M)P(M) \tag{2.16}$$

This relies on placing a prior over the possible functions we could see via $P(M)$ and computing the likelihood of the evidence seen under this model, $P(E|M)$. In terms of a function $f$ and our observed data so far $\mathcal{D}_{1:t} = \{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^{t}$

$$P(f|\mathcal{D}_{1:t}) = P(\mathcal{D}_{1:t}|f)P(f) \tag{2.17}$$

The Bayesian optimisation process is then an iterative one. Taking the distribution over functions, we maximise some objective function, usually called the *acquisition function* with respect to this distribution and sample the new point at the optima of the acquisition function. Algorithm 1 set out the simple procedure for performing Bayesian optimisation.

**Algorithm 1:** Bayesian Optimisation

**for** *t=1,2, ...* **do**
- Find $\mathbf{x}_t$ by optimising the aquisition function over the seen data,

$$\mathbf{x}_t = \arg\min_{\mathbf{x}} u(\mathbf{x}|\mathcal{D}_{1:t-1}) \tag{2.18}$$

- Sample the objective function $y_t = f(\mathbf{x}_t) + \epsilon_t$
- Update the dataset $\mathcal{D}_{1:t} = \{\mathcal{D}_{1:t-1}, (\mathbf{x}_t, y_t)\}$

Two questions are therefore raised by this algorithm. Firstly how to perform the inference of $P(f|\mathcal{D}_{1:t})$ and what form the acquisition function should take. A discussion of both follows.

### 2.2.1 Gaussian Processes as function priors

Gaussian processes are an elegant way to deal with inferring a distribution over possible functions from some observations and a prior. The prior over the function is defined by some mean function on $\mathbf{x}$, $m(\cdot)$, and some covariance function between data-points, $k(\cdot, \cdot)$

$$f(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')) \tag{2.19}$$

In essence this returns the mean and variance of a Normal distribution for a given point $\mathbf{x}$. It is usual to set the mean function to zero both for convenience and the fact that for data we know little about, this is likely the best assumption (in particular if we normalise observed data). This therefore leaves us just with the choice of covariance function.

In the predictive setting, we want to infer predictions about new data points from previously seen data. We know that all points on our function $f$ are jointly Gaussian. If we have observed the points $\mathbf{x}_{1:t}, \mathbf{f}_{1:t}$ and wish to make predictions about $f_{t+1}(\mathbf{x})_{t+1}$ then we can write

$$\begin{bmatrix} \mathbf{f}_{1:t} \\ f_{t+1} \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} \mathbf{K} & \mathbf{k} \\ \mathbf{k}^T & k(\mathbf{x}_{t+1}, \mathbf{x}_{t+1}) \end{bmatrix}\right) \tag{2.20}$$

where

$$\mathbf{K} = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & k(\mathbf{x}_1, \mathbf{x}_2) & \cdots & k(\mathbf{x}_1, \mathbf{x}_t) \\ k(\mathbf{x}_2, \mathbf{x}_1) & k(\mathbf{x}_2, \mathbf{x}_2) & \cdots & k(\mathbf{x}_2, \mathbf{x}_t) \\ \vdots & \vdots & \ddots & \vdots \\ k(\mathbf{x}_t, \mathbf{x}_1) & k(\mathbf{x}_t, \mathbf{x}_2) & \cdots & k(\mathbf{x}_t, \mathbf{x}_t) \end{bmatrix} \tag{2.21}$$

$$\mathbf{k} = \begin{bmatrix} k(\mathbf{x}_{t+1}, \mathbf{x}_1) & k(\mathbf{x}_{t+1}, \mathbf{x}_2) & \cdots & k(\mathbf{x}_{t+1}, \mathbf{x}_t) \end{bmatrix} \tag{2.22}$$

Using the Sherman-Morrison-Woodbury formula we can compute the distribution of $f_{t+1}(\mathbf{x}_{t+1})$

$$f_{t+1}(\mathbf{x}_{t+1}) \sim \mathcal{N}(\mu_t(\mathbf{x}_{t+1}), \sigma_t^2(\mathbf{x}_{t+1})) \tag{2.23}$$

where

$$\mu_t(\mathbf{x}_{t+1}) = \mathbf{k}^T \mathbf{K}^{-1} \mathbf{f}_{1:t} \tag{2.24}$$

$$\sigma_t^2(\mathbf{x}_{t+1}) = k(\mathbf{x}_{t+1}, \mathbf{x}_{t+1}) - \mathbf{k}^T \mathbf{K}^{-1} \mathbf{k} \tag{2.25}$$

Note that once $\mathbf{K}^{-1}$ has been computed, and this is an $\mathcal{O}(n^3)$ operation, then computing $\mu_t(\mathbf{x}_{t+1})$ and $\sigma_t^2(\mathbf{x}_{t+1})$ is comparatively fast as it only involves computing kernels and linear matrix multiplications. This means predicting multiple points comes at little cost compared to a single point. If we wish to model some noise $\sigma_y^2$ on the function, then the equations become

$$\mu_t(\mathbf{x}_{t+1}) = \mathbf{k}^T \left(\mathbf{K} + \sigma_y^2 \mathbf{I}\right)^{-1} \mathbf{f}_{1:t} \tag{2.26}$$

$$\sigma_t^2(\mathbf{x}_{t+1}) = k(\mathbf{x}_{t+1}, \mathbf{x}_{t+1}) - \mathbf{k}^T \left(\mathbf{K} + \sigma_y^2 \mathbf{I}\right)^{-1} \mathbf{k} \tag{2.27}$$

and this parameter $\sigma_y^2$ can be optimised like any other parameter.

A more in depth treatment of GPs can be found in (Williams and Rasmussen, 2006).

The choice of kernel then is the prior that we place on the function, and the conditioning on the data our computation of the likelihood of different models having observed some data. We can then observe the distribution of the posterior by sampling from it at points of interest, as described above.

The choice of kernel is an area with significant literature. In this project we use Exponential Quadratic (EQ) kernels and Linear kernels. EQ kernels have become the default choice for GPs as they have several attractive properties: It is infinitely smooth in it derivatives and it has only two hyper-parameters. It has the form

$$k_{SE}(x, x') = \sigma^2 \exp\left(-\frac{(x - x')^2}{l^2}\right) \tag{2.28}$$

This has two hyper-parameters:

- $l$ - The length scale controls the length scale of the function, how fast the function will change.
- $\sigma$ - The output variance determines the average distance of the function from the mean.

The linear kernel has the form

$$k_{Lin}(x, x') = \sigma_b^2 + \sigma_v^2(x - c)(x' - c) \tag{2.29}$$

This has 3 hyper-parameters:

- $\sigma_b^2$ This places a prior on how far from 0 the hight of the function will be when $x = c$ or $x' = c$.
- $\sigma_v^2$ This determines the functions average distance from the mean
- $c$ This determines the x coordinate at which all posterior function lines will pass through. At this point the function will only have covariance from $\sigma_b^2$ and noise.

We use additive combinations of these kernels to produce more powerful function priors.

The choice of hyper-parameters in the chosen kernel will clearly have a significant effect on the form of the posterior function. It is common therefore to iteratively optimise these hyper-parameters to optimise the train set log-likelihood.

More information on different kernels, and an excellent guide on kernel choice see (Duvenaud, 2014). (Williams and Rasmussen, 2006) also contains information on more complex kernels, combing kernels and advanced GP methods.

### 2.2.2 Acquisition functions



Three main acquisition functions are considered, and the most commonly used in Bayesian Optimisation. These are the Expected Improvement (EI) function, the Probability of Improvement (PI) function and the Upper Confidence Bound function (UCB).

#### Probability of Improvement

The most intuitive of these is the Probability of Improvement function (Kushner, 1964). This simply is the probability that a given point will be better than the current best point.

$$\text{PI}(\mathbf{x}) = Pr(f(\mathbf{x}) \geq f(\mathbf{x}^+) \tag{2.30}$$

Where $f(\mathbf{x}^+)$ is the value of the best point found so far. This is exactly

$$\text{PI}(\mathbf{x}) = \Phi\left(\frac{\mu(\mathbf{x}) - f(\mathbf{x}^+)}{\sigma(\mathbf{x})}\right) \tag{2.31}$$

with $\Phi(\cdot)$ as the Normal cumulative density function. This is an inherently highly exploitative function, and will always pick the most likely place to get a gain. Points with a small variance and slightly higher mean than the best point will be favoured over those with higher variance and mean, leading to strong exploration of local maxima and less likelihood of exploration far away from local maxima. This can present issues if local maxima is not globally maximal. This is best seen by considering PI in another form. If we consider the reward for a given point as

$$R(\mathbf{x}) = \begin{cases} 1 & f(\mathbf{x}) > f(\mathbf{x}^+) \\ 0 & f(\mathbf{x}) \leq f(\mathbf{x}^+) \end{cases} \tag{2.32}$$

Then taking the expectation of this with respect to the current GP posterior at point $\mathbf{x}$ gives us

$$\mathbb{E}_{f(\mathbf{x})\sim\mathcal{GP}(\mu(\mathbf{x}),\sigma^2(\mathbf{x})|\mathcal{D})}\left[R(\mathbf{x})\right] = \Phi\left(\frac{\mu(\mathbf{x}) - f(\mathbf{x}^+)}{\sigma(\mathbf{x})}\right) \tag{2.33}$$

#### Expectation of Improvement

This leads to the form of the second acquisition function, Expected Improvement (Jones et al., 1998; Mockus et al., 1978). Instead of defining the reward for rewarding any improvement, the reward is defined as the amount it will improve over the current best

point Draft - v1.0 Friday 24th May, 2019 – 19:18

$$R(\mathbf{x}) = \max\left[0, f(\mathbf{x}) - f(\mathbf{x}^+)\right] \tag{2.34}$$

Taking the expectation of this over the distribution of $f(\mathbf{x})$

$$\mathbb{E}_{f(\mathbf{x})\sim\mathcal{GP}(\mu(\mathbf{x}),\sigma^2(\mathbf{x})|\mathcal{D})}\left[R(\mathbf{x})\right] \tag{2.35}$$

$$= \int_{f(\mathbf{x}^+)}^{\infty} (f(\mathbf{x}) - f(\mathbf{x}^+))\mathcal{N}(f(\mathbf{x}); \mu(\mathbf{x}), \sigma^2(\mathbf{x}))df(\mathbf{x}) \tag{2.36}$$

$$\mathrm{EI}(\mathbf{x}) = \begin{cases} (\mu(\mathbf{x}) - f(\mathbf{x}^+))\Phi(Z) + \sigma(\mathbf{x})\phi(Z) & \sigma(\mathbf{x}) > 0 \\ 0 & \sigma(\mathbf{x}) = 0 \end{cases} \tag{2.37}$$

This acquisition function should be more exploratory in nature than the PI acquisition function, favouring places where the is the possibility to make larger improvements.

### Upper Confidence Bound

The final acquisition function used is the Upper Confidence Bound acquisition (Cox and John, 1992). This is defined as

$$\mathrm{UCB}(\mathbf{x}) = \mu(\mathbf{x}) + \kappa\sigma(\mathbf{x}),\ \kappa \geq 0 \tag{2.38}$$

The parameter $\kappa$ is left to the user to determine. In experiments in the report it is set to 1. The value of $\kappa$ will determine the algorithm's balance between exploration (high $\kappa$) and exploitation (low $\kappa$).

These three options give rise to significantly different sampling patterns and performance on different tasks. The choice between these three, or any other acquisition function is unclear and there are few heuristics to help make a decision.

## 2.3 The Gaussian Process Autoregressive Regression Model (GPAR)

The Gaussian Processes discussed so far are single output - they can only model one target variable for the given set of inputs. One option in modelling multi-output GPs is to train $k$ independent GPs for $k$ outputs desired. This however will ignore and dependency between outputs and so can loose a significant amount of information.

One option for introducing dependency between the various outputs is to use the Gaussian Process Auto Regressive model (GPAR) introduced by (Requeima et al., 2018).

This model builds on a particular decomposition of the joint likelihood of the outputs. Consider

$$P(\mathbf{y}_{1:k}(\mathbf{x})|\mathbf{x}) = P(y_1(\mathbf{x}), y_2(\mathbf{x}), ..., y_k(\mathbf{x})|\mathbf{x}) \tag{2.39}$$

$$= P(y_1(\mathbf{x})|\mathbf{x})P(y_2(\mathbf{x})|y_1(\mathbf{x}), \mathbf{x})...P(y_k(\mathbf{x})|y_{k-1}(\mathbf{x}), ..., y_1(\mathbf{x}), \mathbf{x}) \tag{2.40}$$

$$= \prod_{i=1}^{k} P(y_i(\mathbf{x})|\mathbf{y}_{1:i-1}(\mathbf{x}), \mathbf{x}) \tag{2.41}$$

This form assumes that each $y_i$ has been sequentially generated as a function of the previous $\mathbf{y}_{1:i-1}$, but as we shall see later this is a property we desire. We then view these individual $y_i$ as random functions with inputs of the input and previous outputs.

$$y_1(\mathbf{x}) = f_1(\mathbf{x}) \tag{2.42}$$

$$y_2(\mathbf{x}) = f_2(\mathbf{x}, y_1(\mathbf{x})) \tag{2.43}$$

$$\vdots \tag{2.44}$$

$$y_k(\mathbf{x}) = f_k(\mathbf{x}, y_1(x), ..., y_{k-1}(\mathbf{x}) \tag{2.45}$$

The GPAR model then models each of these individual $f_i$'s as Gaussian processes such that

$$y_i|y_{1:i-1} \sim \mathcal{GP}(0, k_i(y_{1:-1}(\mathbf{x}), \mathbf{x}, y_{1:-1}(\mathbf{x}'), \mathbf{x}')) \tag{2.46}$$

Although each of the conditionals are Gaussian the joint distribution is not. The joint distribution is not. The moments of the joint distribution is generally intractable. It is however possible to sample from the model by incrementally sampling from each of the conditionals in turn.

Further details of training and inference can be found in (Requeima et al., 2018), however there are two key takeaways that make GPAR an good model to use.

- Inference and learning in GPAR for $M$ outputs and $N$ inputs corresponds to learning $M$ independent GPs, and so scales with $\mathcal{O}(MN^3)$, rather than the $\mathcal{O}(M^3N^3)$ of general multioutput GPs. This includes hyper-parameter optimisation techniques. Additionally standard scaling techniques such as spare approximations can be applied off the shelf.
- Also long as the dataset is *closed downward*, defined as if an observation exists for $y_i$, then there also exist observations for $\mathbf{y}_{1:i-1}$, but not necessarily for any $y_{>i}$, then the model remains exact.

Three additional elements are used in this report not considered in (Requeima et al., 2018).

First that it is possible to continue to joint-optimise the hyper-parameters of preceding GPs. This is done by fitting the GPs in order, and sequentially accumulating the log-likelihoods of the preceding GPs together. Alternatively one can fit the GPs in order and fix the hyper-parameters of the previous GPs when fitting the next. Empirically if joint optimisation is used, it is empirically to independently fit the GPS first.

Second, the tying some of hyper-parameters of kernels together. The data considered in Requeima et al. (2018) has significantly different properties in each of the outputs. However in the data considered in this project the various outputs may well share very similar length scales. While this is not much use when we have a large amount to data to consider, in situations where little data is available, tying various hyper-parameters of the kernels that consider on the inputs $\mathbf{x}$ may lead to better fits. This joint training is however more expansive and it not always beneficial. The effect fo this is investigated.

The final additional is the use of a Markov structure in the outputs. For example in time ordered outputs it may be beneficial to consider the output $i$ as a function of up to only the previous $n$ outputs. This is not quite truly a Markov

this might need to change, not quite sure on it.

structure, the hyper-parameters are still not time-independent, they will be different for each $f_i$, but they will only depend on a given number of previous time steps. The potential upside to this is twofold. First, it can significantly reduce the number of hyper-parameters. The effect of this can be a better fit in lower data situations as it an prevent over fitting. Second is a speed up in training. The downsides to this may be some loss in accuracy, but this may be acceptable, or even minimal. The effect of this is also investigated.

# Chapter 3

# Related Work

The field of architecture search has received a significant burst of work in recent years. As the advances in computational power bought advances in the training of individual networks, the same power has lent itself to searching architectures of ever larger networks. The need for automated methods has increase significantly as the size and complexity of model has increased. Simpler models with a small set of hyper-parameters that can be quickly varied and easily interpreted can be tuned by hand without significant investment of time. As models have grown to the depth of

citations for ResNet style papers

the task of tuning layer configuration, depth, and various other hyper-parameters has become somewhat of a "Dark Art" using mainly heuristic and architectures that have been show to be good on previous problems as a basis.

While there has been no direct application of these methods to Bayesian Neural Networks, the close analogy of Bayesian Neural Networks and their regular counterparts implies that much of the work of architecture search in regular neural networks could be applied to searching for architectures in Bayesian Neural Networks.

Since the work by Zoph and Le (2016) there has been an explosion in the number of papers that deal with this subject (Adam and Lorraine, 2019; Baker et al., 2016; Bender et al., 2018; Brock et al., 2017; Cai et al., 2018; Cortes et al., 2016; Fusi et al., 2018; Jenatton et al., 2017; Kandasamy et al., 2018; Li and Talwalkar, 2019; Liu et al., 2018a, 2017, 2018b; Mendoza et al., 2016; Miikkulainen et al., 2019; Negrinho and Gordon, 2017; Pham et al., 2018; Real et al., 2017; Sciuto et al., 2019; Wang et al., 2019; Xie and Yuille, 2017; Zhong et al., 2017; Zoph and Le, 2016,?; Zoph et al., 2018). Some previous work does exist, although a large portion of this work is limited to small scale neuro-evolution searches, the processes of starting with small networks and progressively growing them in a directed fashion through evolutionary algorithms, and early Bayesian Optimisation applications at general machine learning hyper-parameter optimisation, opposed to specific

| **Search Space** | **Search Method** | **Evaluation Method** |
|---|---|---|
| Continuous vs Discrete | Random search | Full training |
| Unstructured vs Structured | Evolutionary Strategies | Partial Training |
| Cell blocks | Bayesian Optimisation | One-shot models / Weight-sharing |
| Meta-architectures | Gradient based optimisation | Weight inheritance / Network Morphism |
| Restricted building block | Reinforcement Learning | Hyper-networks |

Fig. 3.1 Various groupings of methodologies choices explored by recent works in architecture search, broken into categories

architecture search (Bergstra et al., 2013; Kitano, 1990; Snoek et al., 2012,?; Stanley and Miikkulainen, 2002; Swersky et al., 2014a; Zhang et al., 2016).

The process of architecture search can be broken into three main steps: The search space, the search method, and the evaluation method. While in some methodologies the choice of one of these is necessitated by the choice of another, it is usually possible to separate the choices out. Figure 3.1 lays out the various methodologies that have been utilised in recent work, and some of the choices that have to be made when designing a search method. Figure 3.2 describes the abstract process of architecture search. The choices in search space are not mutually exclusive, larger architecture searches will generally constrain the search space in more than one way to reduce its size. The choice in Search Method and Evaluation Method are generally mutually exclusive.

Briefly, a review of these different sections to architecture search and the methodologies proposed.

## 3.1 Search space

The Search Space defines in principle the possible set of architectures that the search process might discover. The simplest of these are *chain structure spaces*. These are simply a sequence of layers. The search space is characterised by the number of layers, (ii) the number of possible operations choosable at each layer, e.g. fully connected layers,

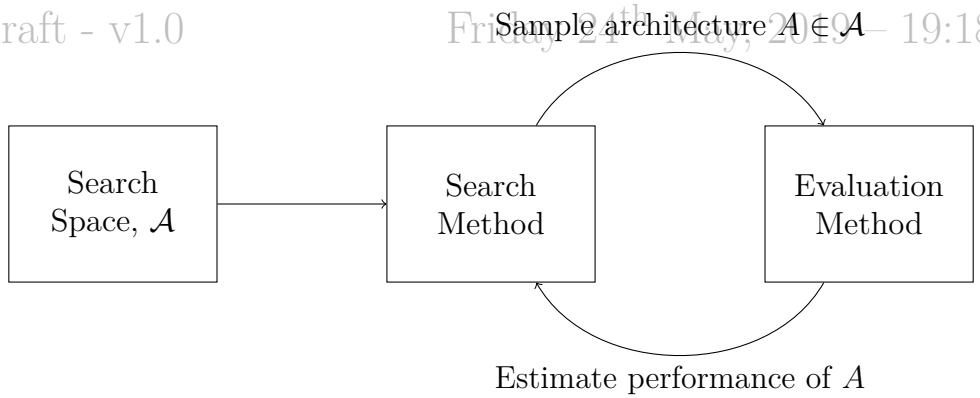
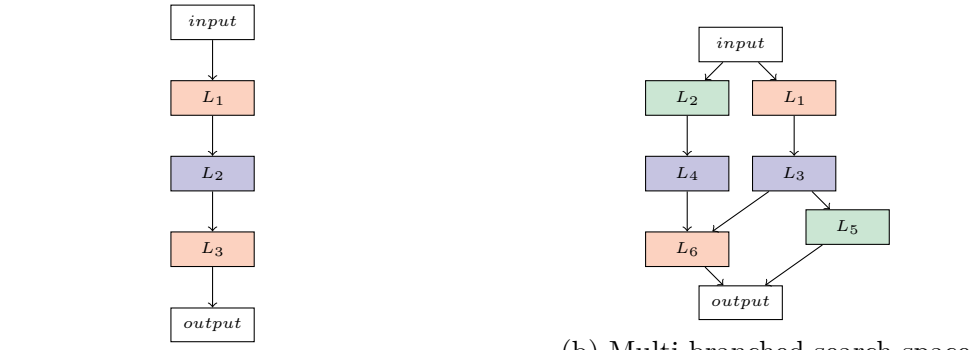



Fig. 3.2 Abstract block diagram of the architecture search procedure. The search space defines a subset of all possible architectures $\mathcal{A}$ to be searched over. The Search Method picks an architecture to sample from the search space to be evaluated by the Evaluation Method. The Evaluation Method returns the performance estimate to the Search Method.




(a) Chain structure search space

(b) Multi-branched search space

Fig. 3.3 Examples of the two cell search spaces

convolutional layers, etc., and the hyper-parameters associated with each layer e.g. number of filters, kernel size and strides for convolutions (Baker et al., 2016; Cai et al., 2018; Suganuma et al., 2018), or the number of units in fully connected layers (Mendoza et al., 2016). Since the number of hyper-parameters are conditional on the choice of operation type, the search space descriptor will in general not be fixed length. Methods either deal with this, or allow only certain choices of hyper-parameters for each layer type, fixing the descriptor length of the search space.

More recent works Brock et al. (2017); Elsken et al. (2018, 2019); Real et al. (2018); Zoph et al. (2018) have introduced the ability to include more complex design elements such as skip connections to the search space which can build *multi-branched structures*. The difference between this and simple feed forward structures is shown in figure 3.3. These search spaces are much more flexible, but are significantly larger.

Finally, following from hand designed networks, the idea of searching for *cells* or larger *building blocks* has been investigated He et al. (2016); Szegedy et al. (2016). In this form, the full network is comprised of a series of repeated, identical blocks. The architecture

of these internal blocks are search for and the blocks then assembled into a full network. This stacking is usually done in style of residual networks He et al. (2016) or DenseNets Huang et al. (2017).

These cell structure search spaces have two main advantages:

- They have significantly reduced search space as the re-usage of a consistent block drastically reduces the permutations of parameters for an equally deep network not employing the cell structure. Applying the same search methodology, Zoph et al. (2018) saw a 7 times speed up with better performance than Zoph and Le (2016)
- The cells found can be transferred to problems of varying size, simply by varying the number of blocks stacked (Zoph and Le, 2016).

In general the choice of the search space greatly affects the difficulty of the search problem. More restrictive spaces are significantly easier to search, but exclude a large number of possible architectures from them. Choosing a good search space involves optimising this trade off.

One difficultly that often arises from these search space choices is that they become discrete in parameters. This can make optimisation difficult.

## 3.2 Search Strategy

The Search Strategy details how the search space should be traversed. It considers the classic exploration-exploitation problem, the difficulty in making sure to discover the global optima while not wasting resources on exploring under-performing regions of the search space. With a few exceptions, these algorithms assume there is some underlying structured relationship between the search space and performance of models, that is from the descriptor of the network in the search space, it is possible to make a (not necessarily accurate) prediction of the performance for the architecture, or that nearby points in the search space will perform similarly.

Many strategies have been explored, including Reinforcement Learning, Evolutionary Algorithms, Bayesian Optimisation, Gradient Descent and Ransom Search.

Historically **Evolutionary Algorithms** were used by many to perform neuro-evolution of architectures, the growing of larger networks from smaller ones, guided via evolutionary strategies. These algorithms often also evolved the weights as well as the architectures Floreano et al. (2008); Jozefowicz et al. (2015); Peter Angeline et al. (1994); Stanley et al. (2018a,b), however in more modern version of these methods as gradient based optimisation of neural networks has become dominant, the evolutionary algorithms have been limited

to just evolving the architecture Elsken et al. (2019); Liu et al. (2018a); Miikkulainen et al. (2019); Real et al. (2018, 2017); Suganuma et al. (2018); Xie et al. (2018).

There are two main differentiators between neuro-evolution methods: Their method of child generation and subsequent population selection. For population selection, some use tournament selection Liu et al. (2018a); Real et al. (2018, 2017), some remove the worst performing Real et al. (2017) and some remove the oldest parents Real et al. (2018), finding this reduces the greediness of the search.

To generate children, most methods use standard mutation techniques and randomly sample new weights. One advantage of the evolutionary algorithm approach however is that child architectures are close in structure to the parents. It is therefore possible to inherit either all of the information learned by their parents Elsken et al. (2019) via network morphisms Wei et al. (2016), or some of the weights by inheriting the parent weights from parts of the network that did not change Real et al. (2017). More information on evolutionary strategies for architecture search can be found in Stanley et al. (2018a).

**Bayesian Optimisation** was successfully applied early on in architectures search. They produced the state-of-the art automatically searched vision architectures in Bergstra et al. (2013) and Domhan et al. (2015), and were the first to beat human designed networks Mendoza et al. (2016). Since the work of (Zoph and Le, 2016) while Bayesian Optimisation has remained popular for hyper-paremeter search, there has been little work applying them to architecture search, likely because BO typically uses Gaussian Processes as the surrogate function, performing well on low dimension continuous search spaces, opposed to the high dimension, discrete spaces typical of modern architecture search. A number of papers have attempted to circumvent this issue by proposing tailored kernels for MLPs Swersky et al. (2014a) and for general multi-branched architectures Kandasamy et al. (2018). Alternatively, other works have used tree based models Bergstra (2010) or random forests Hutter et al. (2018) to search large, conditional search space in architecture search Bergstra et al. (2013); Mendoza et al. (2016). There is preliminary evidence that these might outperform evolutionary algorithms Klein et al. (2016)

To cast the problem as a **Reinforcement Learning** Baker et al. (2016); Zhong et al. (2017); Zoph and Le (2016); Zoph et al. (2018) the generation of the next architecture is considered the agents action. The agents reward is then based on the evaluation of the models performance. Various approaches have been used to optimise the agent. Zoph and Le (2016) use REINFORCE Williams (1992) and Zoph et al. (2018) use proximal policy optimisation Schulman et al. (2017). Baker et al. (2016) used Q-learning.

**Monte-Carlo Tree Search** has also be used to exploit the natural tree structure of the search space Elsken et al. (2019); Negrinho and Gordon (2017); Wistuba (2017).

Finally, while most of the above have used a discrete search space, several works ave looked to introduce a *continuous relaxation* of the search space to allow **Gradient Based**

**Methods** to be employed. In general, where there is a choice of operations $\{y_1, y_2, .., y_m\}$ instead of making a hard selection the choices are weighted be a set of hyper-parameters $y = \sum_{i=1}^{m} \alpha_i y_i(x),\ \alpha > 0,\ \sum_{i=1}^{m} \alpha_i = 1$. Liu et al. (2018b) looks to optimise these directly, where as Cai et al. (2018); Xie et al. (2018) optimise a parametrised probability distribution over weightings. Final architectures are found by picking the $k$ largest options of $\alpha_i$, where $k = 1, 2$ have both been used. The downside to these methods is twofold. First they require the weighs for al possible operations to be held in memory at once, which will grow linearly with the number of options. On te largest tasks they therefore cannot be applied and instead must search on transferable tasks, e.g. CIFAR-10 to Imagenet. Second they have a tendency to become "stuck" in poor choices of $\alpha_i$ if initialised with a poor random seed Liu et al. (2018b).

## 3.3 Performance Estimation

I order to guide te search strategies, some method of evaluating the proposed networks is requires. The simplest way of performing Performance Estimation is to simply take the predicted network, train it to compilation, and report the results on an independent test set. This inevitably however is exceedingly expensive, requiring 1000s of GPU hours for a satisfactory search Real et al. (2018, 2017); Zoph and Le (2016); Zoph et al. (2018).

Naturally therefore this has lead to the development of several methods for reducing the cost of performance evaluation. These have come in 4 main styles

**Low fidelity estimates** of the actual performance after full training reduce the estimation time by reducing the cost of the training procedure. This might be by taking a subset of the data Klein et al. (2016), reducing the number of epochs trained for Zela et al. (2018); Zoph et al. (2018), reducing image resolution or reducing the umbers of filters per layer and a smaller stack of cells Real et al. (2018); Zoph et al. (2018). Problematically however these estimates can introduce biases into the estimates of performance (e.g. generally models with fewer parameters converge faster on the same dataset) and the rank order of models can drastically change if the difference between the approximation and full training is too large Zela et al. (2018).

**Learning curve extrapolation** uses predictive models to predict a models final performance from a small section of its initial training curve on the relevant metric Baker et al. (2017); Domhan et al. (2015); Klein et al. (2016); Swersky et al. (2014a). Using this information, a model can then be early stopped if it is predicted to perform poorly against the cohort Domhan et al. (2015). Baker et al. (2017); Domhan et al. (2015); Klein et al. (2016); Swersky et al. (2014a) all also consider the models hyper-parameters as predictive variables of the models performance. Liu et al. (2018a) only consider the model hyper-parameters as predictive variables. The difficulty with these methods is being able

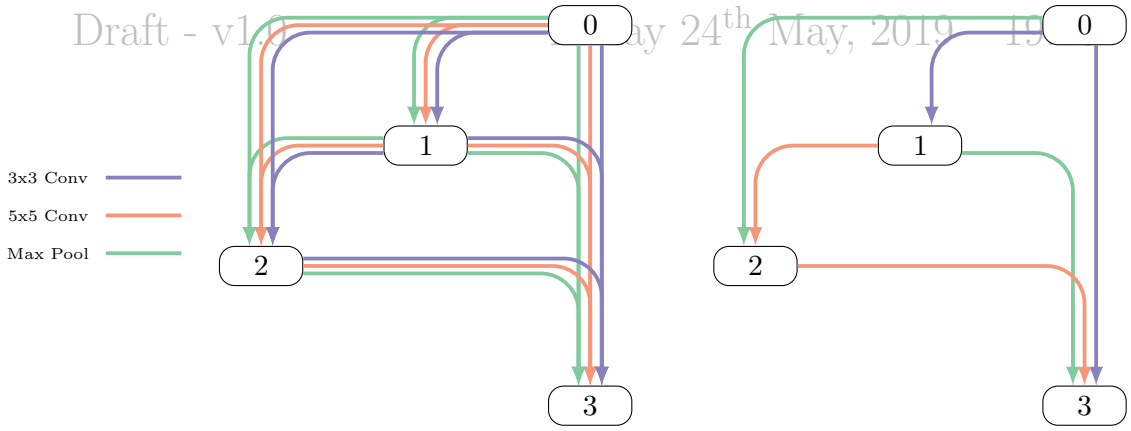



Fig. 3.4 Illustration of one-shot architecture evaluation. The one-shot model consists of a network with one input node, 0, two hidden nodes, 1,2, and one output node, 3, connected such that each node is connected to all previous nodes. Each connection has a number of choices, denoted by the three different colour lines (right). Once an architecture has been selected, only these edges are activated and the resulting architecture is simply a sub-graph of the one-shot architecture (left).

to reliably predict performance of all networks in the search space from just a small subset fully trained examples of networks, likely with a bias in examples toward networks that will perform well.

**Network Morphisms** Wei et al. (2016) is a method to initialise a larger child network with weights derived from its parent while leaving the function it represents unchanged. This allows for the increasing of a networks capacity without loosing and information, and these larger capacity networks can then be trained to convergence in orders of magnitude fewer epochs than from scratch. This allows for competitive search methods costing only a few GPU days Cai et al. (2018); Elsken et al. (2017); Jin et al. (2018). Strictly, all child networks must be larger than their parent to retain the represented function exactly. An advantage of this method is that there is no upper bound on the size of architecture discovered. If an upper bound is desired, the procedure can be approximated to keep the child networks the same size or smaller than their parents Elsken et al. (2019).

Finally **One-shot Architecture Searches** consider all architectures in the search space as sub-graphs of one super-graph, and all sub-graphs of the super-graph share the same weights inherited from the super-graph Bender et al. (2018); Brock et al. (2017); Cai et al. (2018); Liu et al. (2018b); Pham et al. (2018); Xie et al. (2018). Individual architectures can then be sampled by simply zeroing out inactive edges in the graph. Figure 3.4 demonstrates this. This also leads to methods that use only a few GPU days. The methodology for training the sub graphs can differ significantly. Pham et al. (2018) trains the weights of individual sub-graphs a single step when it is sampled by and RNN controller. Liu et al. (2018b) optimises all the weights jointly with a continuous relaxation over the choices of operation. Bender et al. (2018) train the whole super-graph at once,

applying stronger dropout to the operations over time, before fixing the graph and sampling architectures from it.

There are a number of limitations on these methods however. The search space defined by them is quite restrictive, as all architectures in the search space must be sub-graphs of the super-graph. The hindrance of this space may not be too great if the space is designed well, as it has been shown to generally contain high performing networks Bender et al. (2018). Secondly the nature of the method means that usually the whole super-graph must be held in memory, reducing the capacity of the largest possible networks searchable. Finally the can introduce significant bias into the search space. In hindsight, both Li and Talwalkar (2019); Sciuto et al. (2019) showed that the training method of Pham et al. (2018) of the super-graph resulted ranking of predicted performances of tested network had little or no correlation with the true performance. The training regime of Bender et al. (2018) appears to remove this issue, but the effects of the biases introduced by the use of this method are not yet fully understood.

The advantage of this method over network morphisms however is that it allows any network in the space to be sampled quickly, opposed to just architectures similar to the parents. This lends this method more to Reinforcement Learning, Bayesian Optimisation and gradient based approaches, and network morphisms to Evolutionary Strategies.

## 3.4 Criticism of methodology in the literature

Each of these components can have a significant impact on the efficiency of the architecture search and to effectively study the effects of each it it necessary to be able to disentangle the effect of one from another. This can usually be achieved in the form of ablation studies and comparison to effective baselines. Unfortunately this has not been particularly common in work recently. This has been in particular highlighted in (Li and Talwalkar, 2019) and this issue has prevented clear comparison between competing approaches, and the lack of ablation studies preventing the disentangling of the effects of the different components of architecture search. As a result it is hard to judge which components for NAS tested in the literature are in fact effective. One clear example of this is the fact that while Pham et al. (2018) was the first paper to clearly propose the one-shot architecture search methodology, it took over 12 months for separate authors to demonstrate that by simply using random search on the one-shot method propose it was possible to out compete the reinforcement learning approach used in the original paper Adam and Lorraine (2019); Sciuto et al. (2019). The fact that this was not picked up in the original paper is shocking

I don't know if this is appropriate.

.

The author of Li and Talwalkar (2019) identify three key points to improve the reporting and methodology of the work in the Neural Architecture Search field.

- **Inadequate baselines.** Given there are many non-specific hyper-parameter search methods, there should be comparison to these methods to ensure that NAS specific methods do indeed outperform them.
- **Complexity of methods.** There is a significant number of avenues being pursued in the NAS field. Often these methods are complex and innovate on a number of the different parts of architecture at once. However without ablation studies it is difficult to tell which of the components proposed are useful to architecture search. At a minimum, fair comparison to random search is necessary ensure that an algorithm performs better than none.
- **Lack of reproducibility.** None of the papers from 2018 onwards from ICML and NIPS/NeurIPS were fully reproducible, resource capacity aside, as code to complete these searches was not published. In a highly empirical field, it is impossible to reproduce results without full code and the random seeds used supplied. As an additional point, there is a lack of repeat experiments. Many papers report results from a single run, with comment on the variability of their proposed search method.

Please refer to Li and Talwalkar (2019) for more detail.

Reword this?

One final issue not discussed in Li and Talwalkar (2019) that I believe worth discussion is the lack of reporting on the efficiency of these algorithms. While some papers to provided a fair comparison to random search, this I believe does not tell the full story of these approaches, not do them enough justice. Efficiency is the most important metric in architecture search. Given enough samples, even random search will eventually find optimal solutions, as will any method. What should be of most concern is the progression of best results found per number of samples. The importance of this is show in Jin et al. (2018); Negrinho and Gordon (2017). Reproduced figures from Jin et al. (2018) in figure 3.5 clearly show how the proposed algorithm outperforms those compared to at every number of sampled architectures, but in the limit finds architectures of similar performance. This kind of reporting is rare, but enlightening to the performance of these algorithms

Does this sound reasonable? Does it sounds snarky?

.

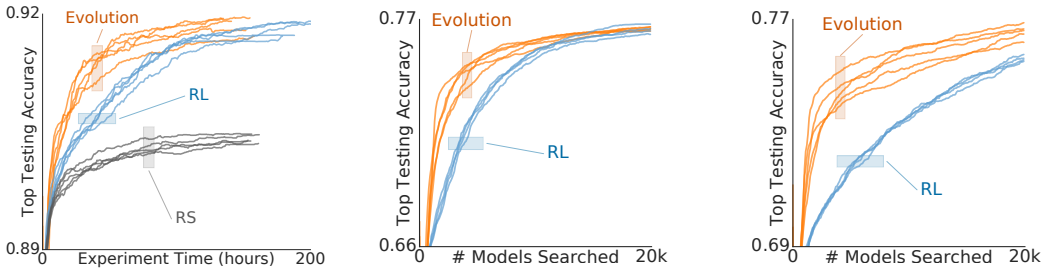



Fig. 3.5 Reproduced figures from (Jin et al., 2018) demonstrating effective reporting of NAS efficiency

# Chapter 4

# Architecture Search for BNNs using GPAR Bayesian Optimisation

## 4.1 Position of this project and experimental design

This might need moving about? Also reconciling with the end of introductory matter

Briefly, I place this project in context of the state of the rest of the field.

Due to the significantly higher computational load of BNNs, the lack of application to large scale CNNs and the limited compute resources available we elected to perform searches on MLP networks for regression problems. In light of this it is impossible to use the advances in search spaces proposed recently. Instead we

We or I? Or avoid all together??

choose a **space space** defined in 2 coordinates, the layer depth and layer width. By fixing all layer widths to be the same, we avoid the issues surrounding conditional search spaces. The search space is still a discrete space, however given that the ordinality of the coordinates will likely be strong predictors for model we treat this as a constrained continuous space, allowing the application of Gaussian Process and therefore Bayesian Optimisation methods as the **search method**. The **evaluation strategy** is a form of training curve extrapolation. Opposed to most methods proposed however we take only snapshots of the model performance at particular iteration steps, up to full training. The objectives of this are to allow the model to make predictions about the performance of all networks in the search space in light of the observed curves without having to being training them, and to allow for capturing patterns that cannot be expressed easily in the form of kernels that extrapolate well in Gaussian Processes, such as the exponential decay basis Swersky et al. (2014b).

In light of the criticisms of Li and Talwalkar (2019), we look to report effectively the results of this work in a manner which will make comparison to later work possible. There are no other existing works at this time on architecture search in BNNs, and so the results of this method cannot be compared to those. Adequate ablation studies are carried out however, comparing the proposed work to both random search and to a simplified form of GP based Bayesian Optimisation to demonstrate the effects of the proposed method

might need to edit this when I get the search results...

.

Additionally these methods will be compared on an performance per number of samples basis for a true comparison of their efficiency. These results will be run over a significant number of random seeds in order to be confident of their statistical significance. Finally the algorithm will be run over 8 different regression tasks to investigate th effect similar but different tasks have on the performance of the algorithm.

For fully published code, along with scripts to reproduce the results fully, including random seed settings and exact configurations please refer to the open-source code bases at:

**Training BNNs and investigating the effects of pruning**:
https://github.com/MJHutchinson/BayesMLP

**Performing architecture searched and GPAR model fitting experiments**:
https://github.com/MJHutchinson/GPAR_Architecture_Search

## 4.2 GPAR for architecture search

A common methodology for architecture search in regular neural networks and in other machine leaning algorithms has been to apply a standard Bayesian Optimisation scheme utilising Gaussian processes over the final metric of interest, e.g. RMSE. Much work has been done in on designing specific kernels to attempt to draw distance between different architectures. The large part of these works however have focused on looking only at the final performance of the networks.

For iterative training procedures however, it is usually possible to extract some information about the final performance of the network partway through training by looking at the validation performance of the partially trained network. Figure

add in the figure

shows the validation log-likelihood of a variety of different BNNs (trained on the power-plant data set with a variety of widths and prior widths). The progression of their validation log-likelihood clearly has some structure to it.

Given the computational expense of training neural networks of any type, it would therefore make sense to utilise the information that can be extracted from the training curves to perform early stopping on networks that appear to be performing poorly.

This is achieved here by taking snapshots of the networks performance after a series of given numbers of time steps, and modelling the performance of networks with different hyper-parameter settings at each of these checkpoints using the GPAR model. Given that the data at any given time will be closed-downward, as defined by (Requeima et al., 2018), since to get to a particular training checkpoint the model must have been trained up to the previous ones as well, and given that the time ordered nature of the checkpoints presents with a clear and sensible way of breaking the outputs down into being sequentially dependant on previous outputs, the GPAR model provides a good model for the data.

The GPAR model serves then as the central surrogate model for the performance of the models through training. Since the objective here is still to maximise final performance, we perform Bayesian Optimisation over the final output of the model. The intent however is to use earlier observed outputs of the model as a basis for predicting final performance, and using this as a method of early stopping to reduce the computational cost of performing the architecture search.

Several other parts of the algorithm must be considered:

- **Initialisation** The model cannot fit without any data to fit to. As such some number of networks are required to initialise the GPAR model. To accomplish this, the search space is randomly searched until a desired number of fully trained networks have been sampled.

- **Search space** The search space for networks must be considered. The way it is defined will impact on the kernel used, and on the performance of the search procedure as a whole. Given the simple nature of the MLP, instead of using a complex kernel to measure network distances, it was decided to define the search space in terms of 2 parameters, *Layer size* and *Layer width*. This involved making the design choice of tying the size of all the hidden layers to be the same. This reduces greatly the size of the search space and significantly reduces the complexity.

  Previous works that have searched over both layer depth and an individual layer width per layer have resulted in complex models

  citations

  or specific kernels

citations

that work only with a limited number of layers. The difficulty with this approach is that for different numbers of layers, the model needs a different number of inputs (one for each size). This issue makes applying Gaussian Processes significantly more difficult, and the direct application of GPAR unclear. As such the given method of tying the sizes was chosen.

The upsides to this choice is a significantly simplified input space, only two variables, and the size of the input fixed regardless of the number of layers. Second the input space is significantly more constrained. This artificial constraint can significant speed up the search by eliminating models of extremely similar nature (e.g. differing by the size of a single layer by one neuron).

The downside to this is that it does remove a significant number of networks that could be optimal. The assumption that the optimal architecture will have all layers the same size is a strong one, especially in light of the conventional wisdom of setting successive layers to be of a smaller size. One method to attempt to conform to this could be to define a a specific shrinkage pattern (e.g. [1, 1, 1/2, 1/4]) for successive layers. This would still leave a search space of the same size, but with networks of a perhaps more sensible description. This was not investigate in this project, but could be in future work.

The final consideration of the search space is the interaction with Bayesian Optimisation. Generally Bayesian optimisation considers inputs as continuous variables to be optimised over. We here however have two distinctly discrete inputs. We do however expect the output to vary smoothly with the changes in input. It therefore makes sense to discretise the input space into only whole numbers. We can the utilise Thompson sampling (Russo et al., 2018; Thompson, 1933) to estimate the mean and variance functions at each of these points.

Taking this a step further, it makes further sense to sub-sample the input space further. A 3 layer network of 100 hidden units per layer is unlikely to perform significantly differently from a 3 layer network of 101 hidden units per layer. As such it makes sense to sparsity the input space more strongly as the layer size and depth grows.

- **Parallel Computation** Modern computers are capable of training multiple models in parallel. In order to exploit this capability, the search procedure should be able to handle training multiple networks at once. If we wish to train $k$ models in parallel, simply taking the $k$ top values from the estimates utility function at each of the considered inputs may not lead to the most optimal search pattern as it is likely the points will be very close together in the search space all sitting in the same maxima of the utility function. A method of controlling this is investigated in this report.

By deliberately under-sampling the output of the GPAR model, we can inject some small stochastic noise into the utility function. By using a different set of samples for each new point we wish to pick, this will encourage them to spread across the maxima of the utility function. The effect of this is investigate. More advanced methods for dealing with this problem also exist, and are discussed in future work.

With these things in consideration, we can define the GPAR based architecture search algorithm, shown in algorithm 2

Clean this up

.

**Algorithm 2:** GPAR Based Bayesian Optimisation

**Input:** $\mathcal{S} \subseteq \mathbb{N}^n$: A subset of the positive integers that defines the hyper-parameter search space.
**Input:** $\text{BNN}(\cdot, \cdot)$: A function that takes $\mathbf{x} \in \mathcal{S}$ and a list of integers specifiying number of optimisations steps, which trains a Bayesian Neural Network up to the given steps and reports the validation metirc at these steps, $\mathbf{y}$
**Input:** $u(\cdot)$ the aquisition function to use
**Input:** $l$: A list of the optimisation steps to train to.
**Input:** $B$: Some budget one the optimisation
**Input:** $k$: A number of random samples to train as an initialisation set
**Input:** $m$: The number of models to train in parallel
**Output:** $\mathbf{x}^*, \mathbf{y}^*$: The optimal hyper-parameters and the trained Bayesian Neural Network

$\mathcal{D} = \{\mathbf{x}_i, \mathbf{y}_{1:l,i}\}_{i=1}^k = \{\mathbf{x}_i, \text{BNN}(\mathbf{x}_i)\}_{i=1}^k$, for $\mathbf{x}_i$ drawn randomly from $\mathcal{S}$;
Remove the $\mathbf{x}_i$ in $\mathcal{D}$ from $\mathcal{S}$;
Update the best found point so far, $\mathbf{x}^*, \mathbf{y}^*$ with the best point from the compliment of $\mathcal{S}$, $\mathcal{S}^c$;
**while** $B > 0$ **do**
    $model = fit\_GPAR(\mathcal{D})$;
    $samples_i = sample\_GPAR(model, \mathbf{x}_i)$ for $\mathbf{x}_i$ in $\mathcal{S}$;
    $\mu_i, \sigma_i^2 = mean(samples_i)$, $variance(samples_i)$;
    $u_i = u(\mu_i, \sigma_i^2)$;
    $\mathcal{D}^+ = \{\mathbf{x}_j\}_{j=1}^m$ where $\mathbf{x}_j$ is the $j^{th}$ largest $u_i$;
    Advance the training of each point in $\mathcal{D}^+$ by one optimisation step set.;
    If any in $\mathcal{D}^+$ reach the final time step, remove them from $\mathcal{S}$;
    Update the best found point so far, $\mathbf{x}^*, \mathbf{y}^*$ with the best point from the compliment of $\mathcal{S}$, $\mathcal{S}^c$;
    Update the budget $B$;

# Chapter 5

# Experiments

## 5.1 Investigation into the effects of various hyperparameters on the performance of BNNs on simple problems

There has been little systematic study into how the various hyper-parameters of a Bayesian Neural Network affects the final performance of network. This set of experiments aims to characterise the performance of small BNNs on a range of regression tasks while varying a number of hyper-parameters in an attempt to extract trends.

The form of network used is a MLP network, with a number of hidden layers and a number of hidden units in each layer. The priors placed on the weights of the network are multivariate independent Gaussians with zero mean. The width of the priors is varied in experiments. The variational distributions used are independent Gaussains. The data is whitened before training by normalising the data to have 0 mean and a standard deviation of 1

It should be noted that as these are preliminary experiments designed to judge only if there is a worthwhile dependency between the variables investigated multiple seeds were not investigated due to the computational cost. The proximity of points tested however is enough to see the small amount of variability that affects the final objective metrics of BNN training.

### 5.1.1 Hidden width

Figure 5.1 details the effects of varying the widths of the hidden layers in 2 layer networks with a fixed prior width. There is a clear dependency. Some datasets present with a clear minima in validation log likelihood with varying layer size, and some reach a minimum

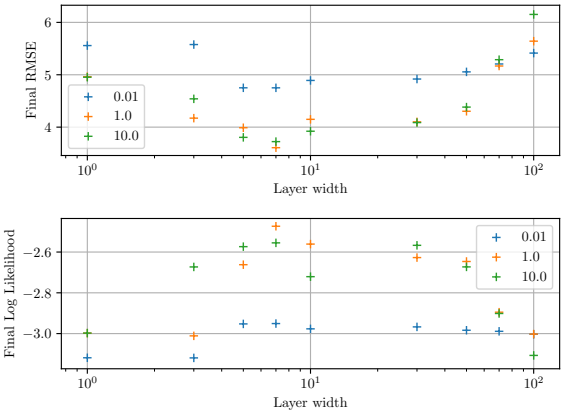
level in validation log likelihood and increasing layer width beyond this point has no effect. While it intuitively makes sense for performance to increase as the size of the network increase, the drop of in performance with even larger layer size is investigated later.

### 5.1.2 Prior width

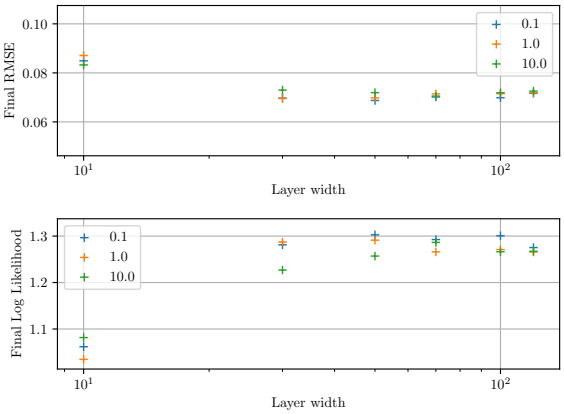
Figure 5.2 details the effects of varying the widths of the prior in 2 layer networks with a fixed architecture. There is again a clear dependency here. For some, the prior width appears to be important up to a particular width, after which it is makes no difference if the width is increased. For some, there is a minor drop off in performance with to large prior width. The clearest trend is that to small a prior width causes significant detriment to performance. This is to be expected .Too small a prior width will over penalise weights of the size required to produce the required outputs, significantly hapering the models predictive performance.

### 5.1.3 Initialisation of $\sigma_y^2$

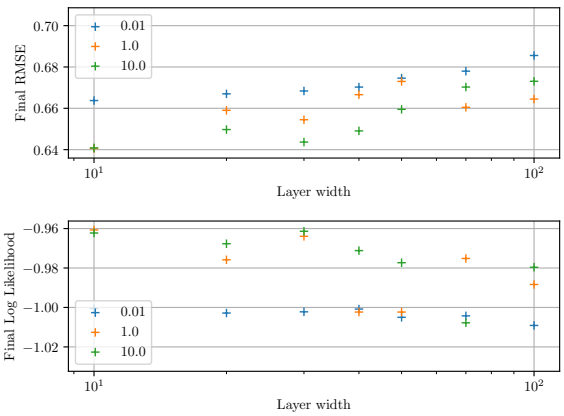
The final parameter investigated in this manner is the initial homoeoskedastic noise on the output o the network, $y \sim \mathcal{N}(f_\mu(\mathbf{x}), \sigma_y^2)$. The effect of the initialisation of this parameter appeared to have little effect on the final performance of the network. There was some effect on the convergence rate of the network. Generally larger $\sigma_y$ converged faster, but this trend as nt conclusive. Given the lack of effect on the final performance of the network, it is omitted from the architecture search.



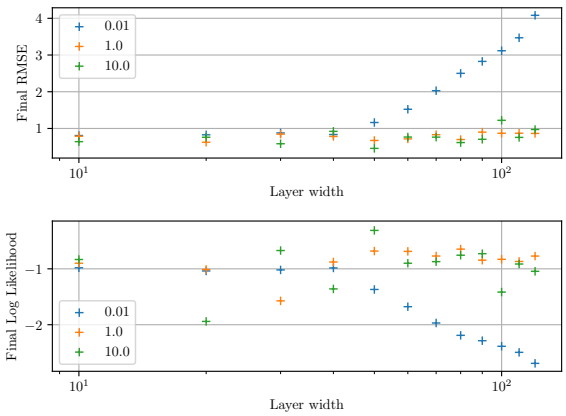

(a) Boston Housing




(b) Kinematics 8nm

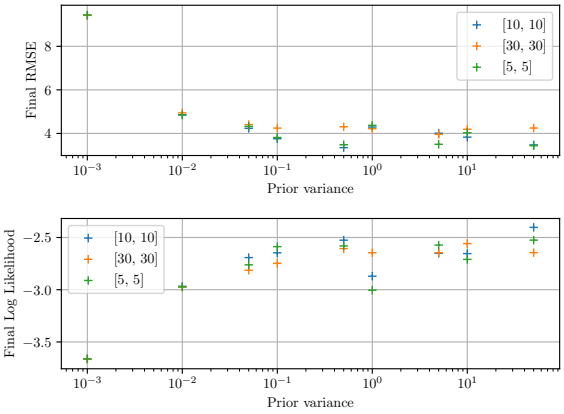



(c) Wine Quality

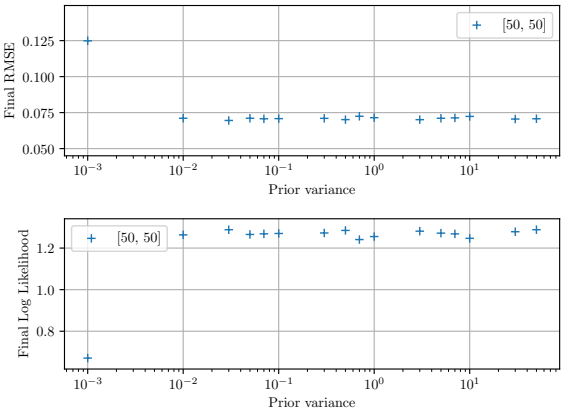



(d) Yacht

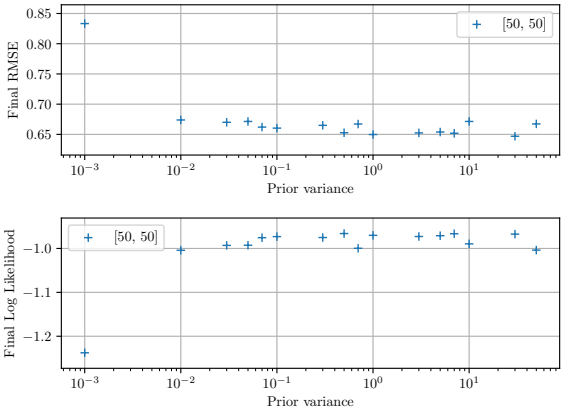
Fig. 5.1 Final validation log likelihood and RMSE error of various sizes of the hidden layers in 2 layer BNNs. Colour denotes prior width used. Plotted for a number of datasets



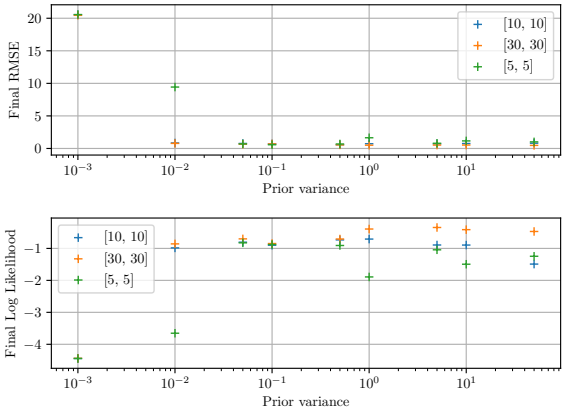

(a) Boston Housing




(b) Kinematics 8nm




(c) Wine Quality




(d) Yacht

Fig. 5.2 Final validation log likelihood and RMSE error of various prior widths in 2 layer BNNs. Colour denotes network structure. Plotted for a number of datasets

## 5.2 Data Augmentation to control over/under fitting

One explanation for the drop of in performance with larger network size comes from the scaling of the terms in the ELBO cost. The two main terms are the reconstruction loss and the prior fit terms.

$$\text{ELBO} = \underbrace{\mathbb{E}_{q_\phi(\mathbf{w})}\left[\log P(\mathbf{y}|\mathbf{w},\mathbf{x})\right]}_{\text{reconstruction loss}} - \underbrace{\mathcal{D}_{KL}\left(q_\phi(\mathbf{w})\|P(\mathbf{w})\right)}_{\text{prior fit}} \tag{5.1}$$

The reconstruction loss is a sum over the log likelihood of the individual data points in the data set therefore scaling with the quantity of data This term rewards god fit to the data. The prior fit term is a sum over the KL divergence of individual weights (when using a prior with no dependence between weights), therefore scaling with the number of weights in the network. This term rewards keeping he weights close to the prior.

If we consider a fixed network architecture, we should therefore expect the optimal weights of the network will vary with the amount of data set, as the balance between te two cost terms changes. With a large data to network size ratio, we would expect the reconstruction term to overpower the prior fit, resulting in an emphasis on fitting to the train set and potentially over-fitting. With low data to network size ratio, we should expect a prioritisation of the prior fit term and a potential for under-fitting. A balance between these terms will give optimal validation set performance.

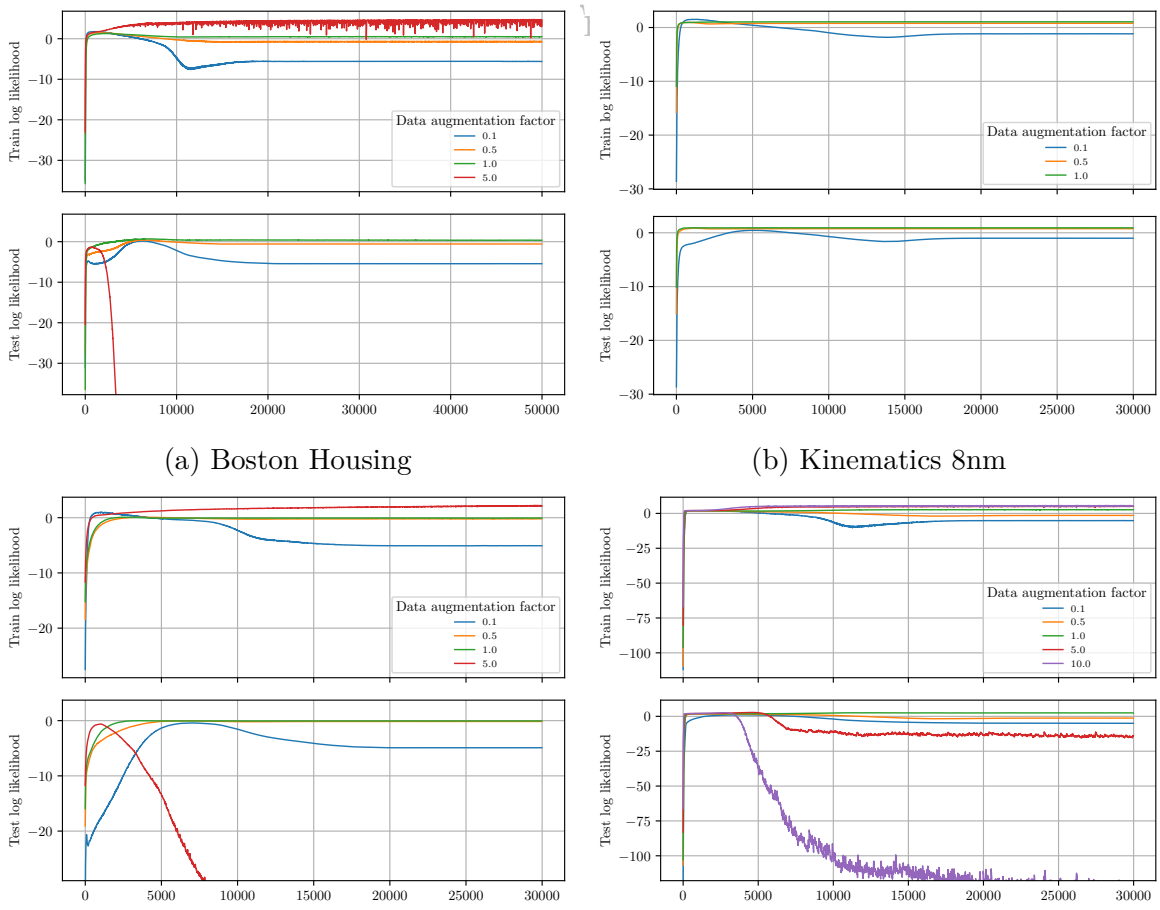
In order to test this, the architectures and prior widths that gave optimal performance in the previous experiments were taken. The training datasets were then either under-sampled by some factor by random selection, or over sampled by repeating the dataset and shuffling randomly. The networks were then trained on the new datasets and evaluated on the original validation sets.

Figure 5.3 shows the results of these experiments. We observe the clear signs of under-fitting for low amounts of data - slightly reduced train and validation log likelihood, and over fitting with large amounts of data - raised train log likelihood but severely dropped validation log likelihood.

These results explain the existence of optimal network size being at some finite size, as opposed to an infinite size.

The implications for architecture search are that the size of the network will play a significant role in the performance, as well as the underlying connective structure. Larger networks will likely systematically under-fit due to over-regularisation by the prior fit term and small networks over fit.

This does however present an interesting method to investigate for restricted size networks, e.g. on embedded devices. Given for small networks too much data in the training




(a) Boston Housing

(b) Kinematics 8nm

(c) Wine Quality

(d) Yacht

Fig. 5.3 The effects of modifying the amount of data in a dataset on the train and test log likelihood for a fixed network size. Upper plots show the train set log likelihood through optimisation, and the lower plots the validation log likelihood.

set will cause over-fitting, reducing the amount of data to an empirically appropriate level could see significant performance gains.

## 5.3 Pruning effects in mean-field BNNs

A question raised by the previous section is how the optimisation is

is it an issue I've moved to using hyper-priors here? Might affect the way things are being pruned.

how the trade off is being made between the two terms in the ELBO loss.

This is investigated by looking at the weights associated with a given hidden unit. Looking at the KL divergence between the weight and the prior, we can investigate how "active" a given weight is. A low KL would indicate that the weight is close to the prior, i.e. close to zero mean and with a width close to the prior. A higher KL would indicate

the weight is sufficiently dissimilar from the prior. By averaging the KL of all incoming weights to a unit, we can get a measure of the activity of a given neuron.

The BNNs trained utilise the VI training framework with mean field Gaussian prior, utilising Inverse-Gamma hyper-priors on the prior width, with Inverse-Gamma parameters $\alpha = 4.4798, \beta = 5.4798$ (recommended by Wu et al. (2018)). They are optimised with the ADAM optimiser, with the settings learning rate=0.001, $\beta_1$=0.9, $\beta_2$=0.999, $\epsilon$=1e-8.

Figure [*** insert kl distribution plots***] show example histograms of the distribution of the average KL per neuron. We can clearly see two distinct groups emerge. A larger group, comprising of inactive neurons, and a much smaller group showing the active neurons. Given the very small means of the weights in the larger group, they are unlikely to contribute significantly to the predictions of the network. We can therefore consider these neurons inactive, or pruned. By placing an appropriate threshold on the value of the average KL for a neuron, we can count the number of total active neurons in a network.

Figures 5.4 and 5.5 detail the results of training various networks on 8 UCI datasets. Networks are defined by a number of layers,a nd the same hidden width for every layer. Networks of various widths and depths are tested, as appropriate to the specific dataset.

Clearly visible we can see trends in the pruning effect.

the thresholds on these plots aren't 100% accurate... there's a bit of a grey area as the pruned weight group breaks away from the main group.

The number of active neurons increases with network size, utilising all neurons available, up to a point. Past a given break point there is a decreases in the number of neurons effectively utilised. There is a sharp discontinuity in many of these plots and a drop in the number of neurons utilised. This comes from the difficulty in exactly defining the boundary of what the threshold is for a neuron to be active. Before this point all neurons are active and so form a single group on the spread of neuron KLs. After this point, two groups exist, the active and inactive. In the transition region between the two, the spread of KL values widens until it separates into the two groups. Defining exactly when to consider neurons inactive is slightly arbitrary, and always leads to the sort fo discontinuity seen.

It is interesting to note that even as fewer neurons become active, in many cases the validation performance of the network continues to increase. This is likely due to the extra regularisation provided by the removal of additional neurons causing the network to generalise better than networks that are slightly smaller. In the limit of large networks, one of two effects appears to present. Either the performance of the network flat-lines, or we see a decrease in performance. The second can be explained by over-regularisation. The balance between prior fit and reconstruction tips towards prior fit, over pruning the network and reducing performance. The flat-lining in performance is less easy to explain.

I think this has something to do with "easiness" of fit - if a model can cheaply get a good fit it will. If the pruning kicks in too soon before the network is large enough to explain the data well then it cant justify the poor fit and starts pruning. Poor and anthropomorphic explanation but my guess to what's happening

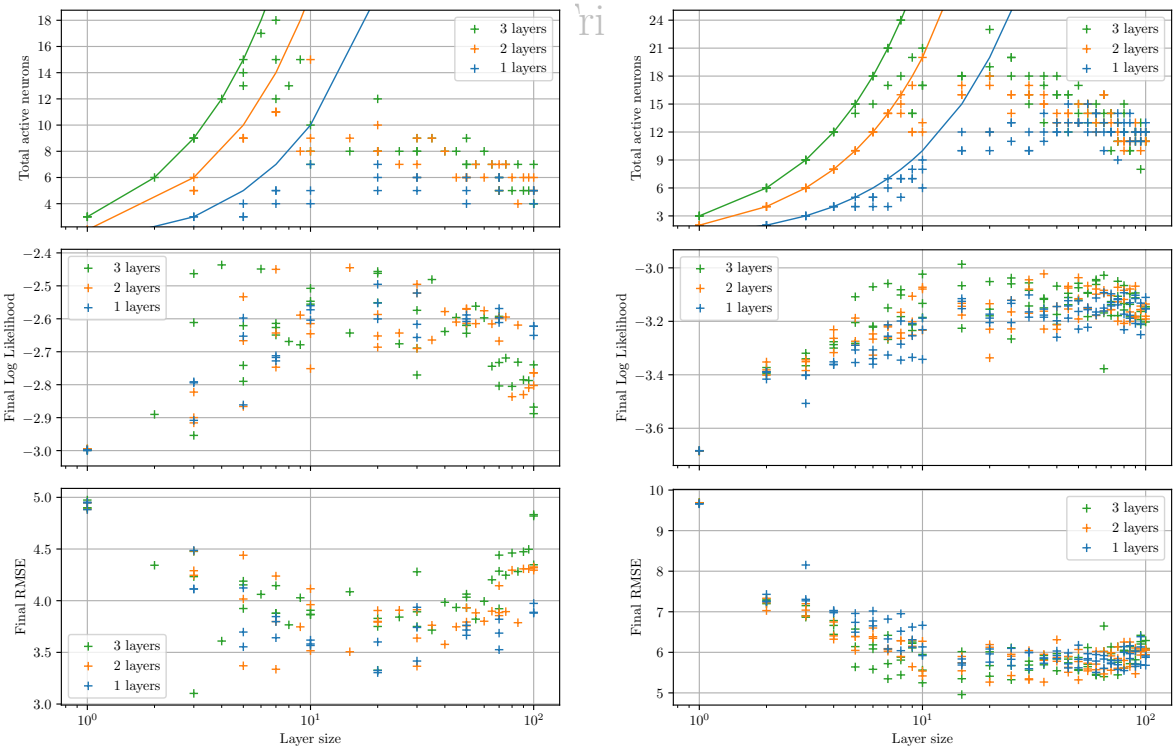
Another point of interest is that these trends tend to match with the width of the network, rather than with the total number of neurons or weights in the network. Further investigation into the pruning in individual layers of the network shows that most active neurons are in the first layer of the network, with significantly fewer in later layers. This would imply that the size of the first layer of the network is the bottle neck in performance in this case, and that subsequent layers need not be as wide, as per conventional methodology. This is a potential deficiency of the search space with all layers the same size proposed for this project. An alternative to this that attempts to resolve this issue does not change the methodology is proposed earlier, although not investigated.

## 5.4 Empirical kernel and hyper-parameter selection for GPAR models of architecture performance

Gaussian Process model fitting is a non-trivial task. To do some one must pick appropriate kernels and training hyper-parameters. In the GPAR model there are a number of other considerations, discussed earlier. This section looks to empirically investigate the choice of kernel and hyper-parameters for application to the search space of architecture search. In order to produce a good surrogate model to use in Bayesian Optimisation, it is necessary to find good training setting and kernel choices.

The datasets fitted to are the results of the models trained in the previous section. 3 checkpoints are used from the training of the models, 1000 steps, 4000 steps and the final step (typically 30,000-50,000). These points are chosen for two reasons. First from observation the first points lie in the region where it is not immediately obvious which network will end up with the better performance. Beyond this point models tend to only slowly asymptote to their optima. This makes the modelling task more simple. Second, it means that the computation saved from early stopping is significant. We only incur the significant additional cost of going from the second output to the final if we believe to to be a good idea in light of the observed training so far. This subset of models define a search space with a discrete set of network deaths and widths over a given range. Each model reported in this section is run on 5 differently selected subsets of data and trained with a different random seed. The results are reported as the mean and standard deviation of results obtained.

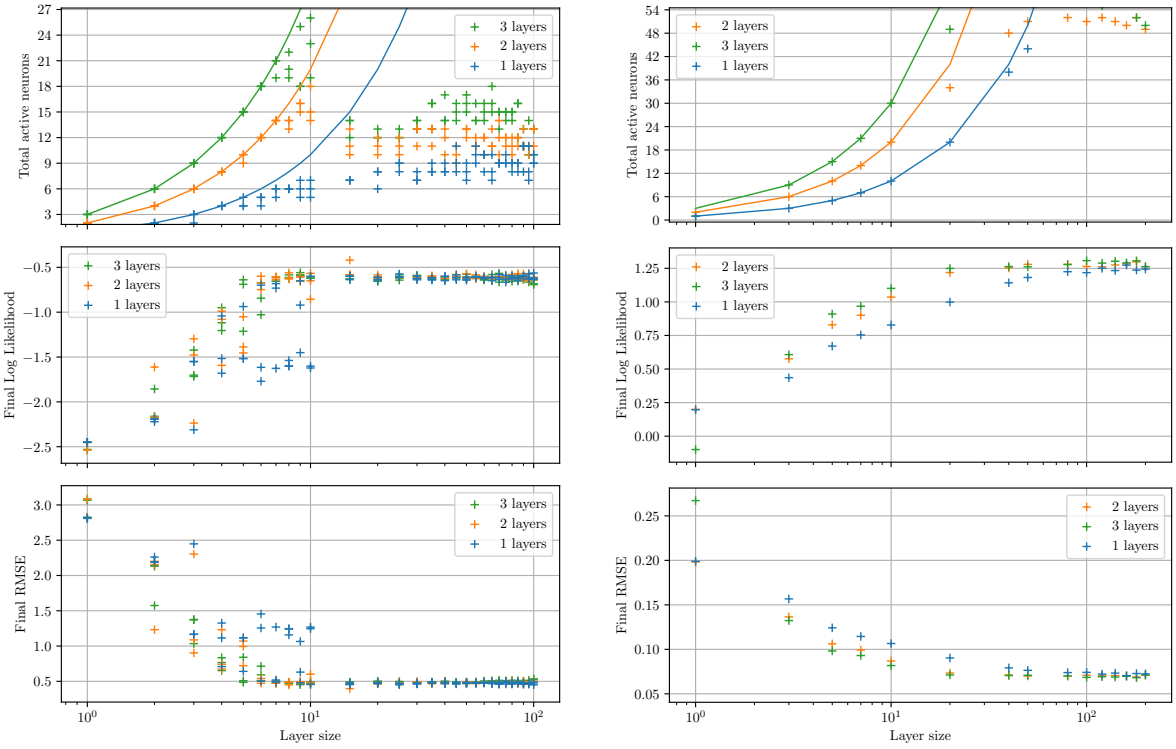
These tests are done in two sections, investigation into appropriate kernels, and an investigation into the other training settings of the model. It was ensured that the parameters not tested in each experiment were sufficiently good. They may not be optimal




(a) Boston Housing




(b) Concrete

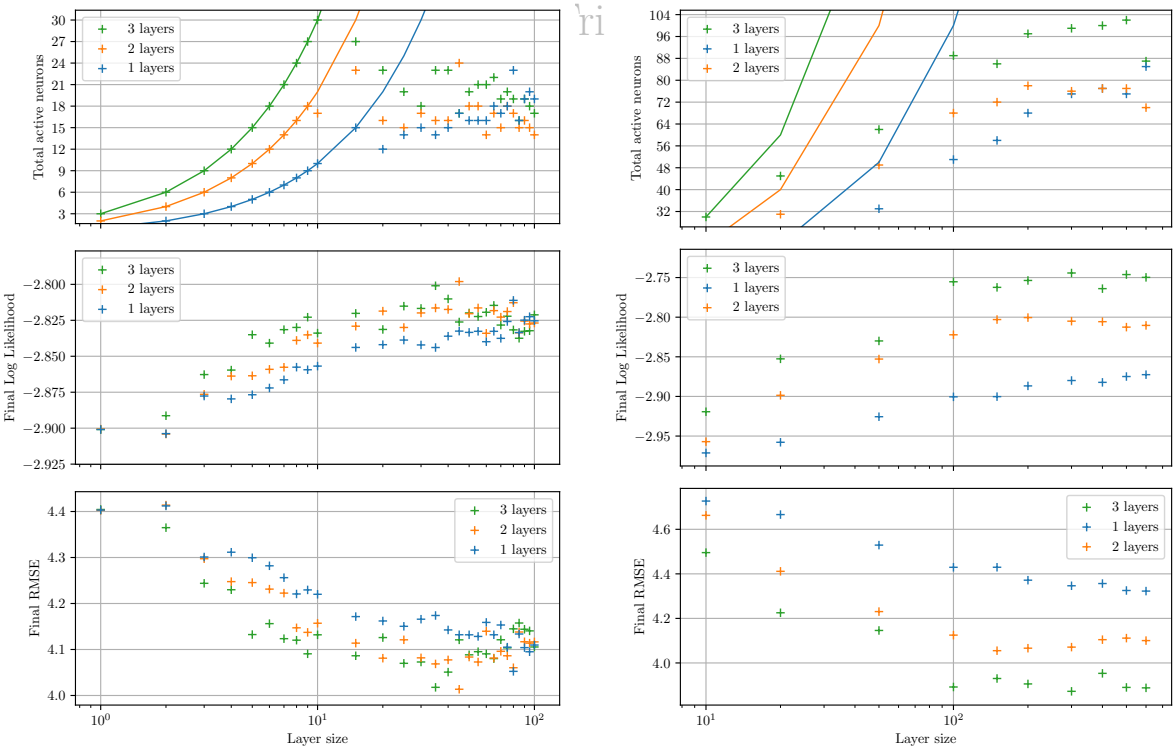



(c) Energy




(d) Kinematics 8nm

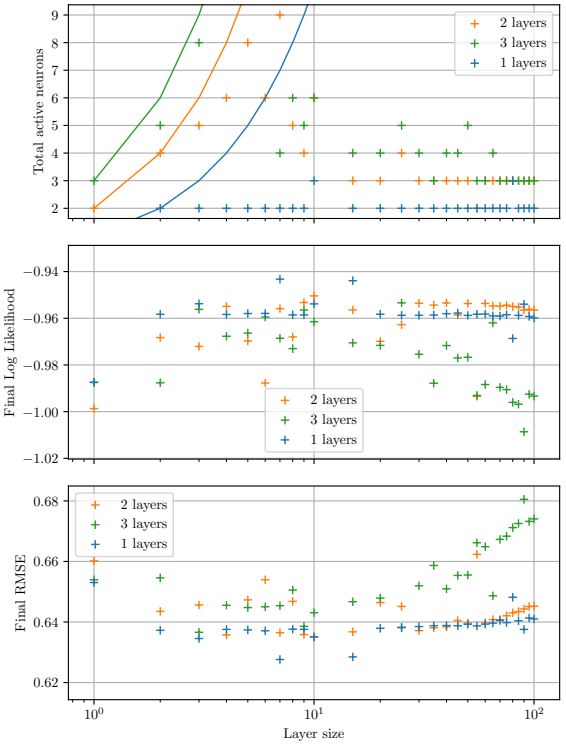
Fig. 5.4 Plots detailing the effect of pruning in V BNNs on various datasets for a range of architectures. Upper plots show the number of units active in the network, defined by the average KL of units input weights. Solid lines show the total number of units available in the network. Middle plots show the average log likelihood over the last 20 optimisation steps of the network. Lower plots show the average RMSE error over the last 20 optimisation steps of the network.



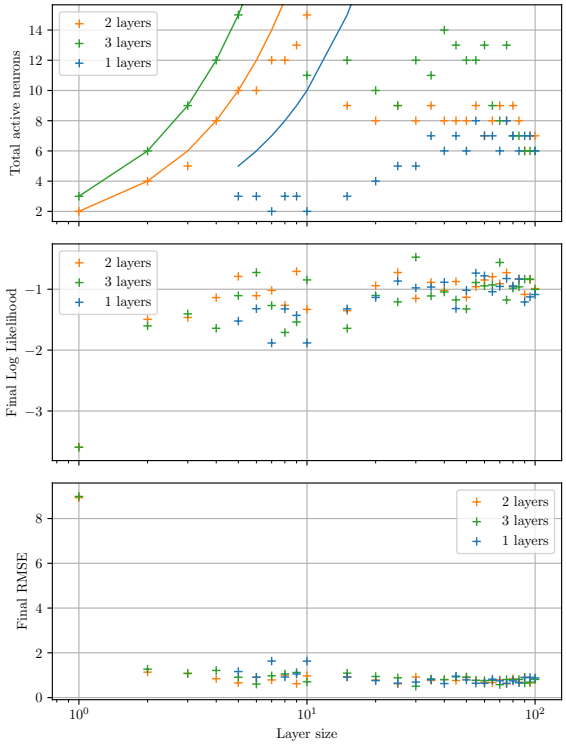

(a) Power Plant




(b) Protein Tertiary Structure




(c) Wine Quality




(d) Yacht

Fig. 5.5 Plots detailing the effect of pruning in V BNNs on various datasets for a range of architectures. Upper plots show the number of units active in the network, defined by the average KL of units input weights. Solid lines show the total number of units available in the network. Middle plots show the average log likelihood over the last 20 optimisation steps of the network. Lower plots show the average RMSE error over the last 20 optimisation steps of the network.

however and so this pair of experiment implicitly assumes that the results of each are reasonably independent of the other settings.

The models are optimised with the L-BFGS-B algorithm, using the implementation from the python package SciPy.

### 5.4.1 Kernel investigation

First we investigate the appropriate form of kernel to use. A non linear EQ kernel is always placed on the input. In addition a combination of some, all, or none of the following are also used, added together: a liner kernel on the inputs $\mathbf{x}$, a linear kernel on the outputs $\mathbf{y}_{1:i}$ and a non-linear kernel on the outputs.

Table 5.1 shows the results of fitting these models to a training set of half the dataset, randomly selected. Table **??** shows the results of fitting these models to a training set consisting of 10 points from the dataset, randomly selected. Both are then validated on the remaining data and the log likelihood reported. The second table, the low data case, is of particular interest for the architecture search case as we are looking to make good prediction about model performance from just a few samples.

The results of this investigation are inconclusive. When using a large amount of data there seems to be a preference for using all the possible kernel option, however this still not close to unanimous. In the low data case, the differences in mean are often overshadowed by the standard deviation of the results. The kernel chosen to use in experiments therefore is a non-linear kernel on the inputs and a linear kernel on the outputs as it appears to overall be the best fitting, but not clearly.

### 5.4.2 Model fitting hyper-parameters

We now look at the additional properties of the GPAR model that can be applied. These are described in more depth in the methodology section.

- **Markov structure.** This can be varied from no structure down to a Markov length of 1.
- **Tying input scales** This can either be active or not.
- **Joint training** This can either be active or not.

The results here are again inconclusive. No set of settings appears to be particularly dominant although the one trend that does appear as one might expect is that performing joint optimisation is better than not. In the low data case it also appear that tying the scales of the kernels on the input helps. Overall a markov length of 1 was chosen, as

it speeds up training, and the input scales were tied with joint optimisation of the log likelihood performed.

## 5.5 Architecture search experiments

The results of section 5.4 specify the parameters of the surrogate model to be used in the Bayesian Optimisation procedure, and the results from section 5.3 define a search space to be search over. Given the cost of training BNNs, instead of performing these search in an online fashion, the results from section 5.3 are used as a surrogate dataset for training new BNNs. To sample an architecture, the algorithm draws the corresponding results from the pre trained results. While this may introduce some bias into the results due to only a single random dataset being used, there are no better options due to the cost of training BNNs on the limited resources available.

All results in this section are run on 16 different random seeds to provide a good statistical estimate for the properties of each combination of settings. Fully reproducible code and settings can be found at https://github.com/MJHutchinson/GPAR_Architecture_Search

### 5.5.1 Final performance architecture search

#### On architecture

**[* Ablation study to be run *]**

#### On synthetic functions

**[* Ablation study to be run *]**

### 5.5.2 Multi-output search

#### On architecture

The data used here are 3 outputs taken at 1,000, 4,000 and 30,000 or 50,000 optimisation steps (dataset dependant, the final output) from the results of section 5.3. Each value is the average of the previous 20 optimisation steps results to smooth the noise in objective function that appears in BNN training.

4 search methods are run on this search space. 3 Bayesian Optimisation GPAR methods, using the Expected Improvement, Probability of Improvement and Upper Confidence

| Kernal settings | | | Dataset | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Linear kernal on inputs | Linear kernal on outputs | Non-linear kernal on outputs | bostonHousing | concrete | energy | kin8nm | power-plant | wine-quality-red | yacht |
| False | False | False | -2.64 ± 0.351 | -1.86 ± 0.429 | -0.649 ± 0.2 | 1.37 ± 0.304 | -0.437 ± 0.299 | -0.874 ± 0.246 | -51 ± 98.7 |
| | | True | -2.42 ± 0.269 | -1.75 ± 0.374 | -0.49 ± 0.211 | 2.87 ± 0.409 | 0.167 ± 0.235 | -0.554 ± 0.168 | -48.9 ± 97.7 |
| | True | False | -2.79 ± 0.248 | -1.82 ± 0.457 | -0.529 ± 0.222 | 2.97 ± 0.4 | 0.0153 ± 0.245 | -0.611 ± 0.169 | -49.1 ± 96.8 |
| | | True | -2.56 ± 0.3 | -1.75 ± 0.354 | -0.391 ± 0.207 | **3.12 ± 0.432** | 0.196 ± 0.212 | -0.57 ± 0.187 | -52.7 ± 7.68 |
| True | False | False | -2.64 ± 0.352 | -1.86 ± 0.429 | -0.644 ± 0.203 | 1.4 ± 0.31 | -0.388 ± 0.306 | -0.797 ± 0.213 | -50.5 ± 98.3 |
| | | True | **-2.41 ± 0.265** | **-1.74 ± 0.37** | -0.489 ± 0.212 | 2.88 ± 0.412 | 0.197 ± 0.26 | -0.529 ± 0.16 | -48.7 ± 97.2 |
| | True | False | -2.78 ± 0.303 | -1.82 ± 0.458 | -0.527 ± 0.222 | 2.95 ± 0.431 | 0.14 ± 0.13 | -0.6 ± 0.174 | -48.7 ± 96.8 |
| | | True | -2.56 ± 0.284 | -1.77 ± 0.39 | **-0.387 ± 0.208** | 3.1 ± 0.44 | **0.245 ± 0.223** | **-0.531 ± 0.161** | **-48.1 ± 96.9** |

Table 5.1 The per data-point validation log likelihood of fitting various combinations of kernels to the performance characteristics of BNNs trained on various datasets. Training sets are 50% of samples randomly selected. Validation set is the remaining data. Each experiment run 5 times with different seeds. The mean and standard deviation of the results are reported.

| Kernal settings | | | Dataset | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Linear kernal on inputs | Linear kernal on outputs | Non-linear kernal on outputs | bostonHousing | concrete | energy | kin8nm | power-plant | wine-quality-red | yacht |
| False | False | False | -123 ± 242 | -23.1 ± 25.9 | -185 ± 386 | 1.49 ± 2.43 | 0.233 ± 2.48 | -19.9 ± 27.4 | -3.93E+03 ± 6.83E+03 |
| | | True | -124 ± 255 | -22.9 ± 27.3 | **-96.6 ± 136** | 3.19 ± 1.25 | 1.27 ± 1.48 | -19.9 ± 27.8 | -3.25E+03 ± 5.05E+03 |
| | True | False | -118 ± 234 | **-21.2 ± 23.6** | -580 ± 586 | 2.8 ± 1.37 | 1.37 ± 2.23 | -33.7 ± 37.9 | -4.02E+03 ± 6.92E+03 |
| | | True | -120 ± 241 | -36.2 ± 32.9 | -457 ± 451 | 1.11 ± 4.54 | **1.71 ± 1.75** | -33.5 ± 37.9 | -4.24E+03 ± 7.37E+03 |
| True | False | False | -123 ± 241 | -23 ± 26 | -169 ± 350 | 1.1 ± 2.12 | 0.53 ± 1.96 | -19.8 ± 27.4 | -3.92E+03 ± 6.8E+03 |
| | | True | **-105 ± 210** | -24 ± 29.3 | -144 ± 207 | 2.78 ± 1.47 | 1.31 ± 1.5 | -19.7 ± 27.9 | **-3.18E+03 ± 5.09E+03** |
| | True | False | -121 ± 242 | -21.3 ± 24.4 | -379 ± 531 | 2.97 ± 1.35 | 1.34 ± 2.21 | -19.8 ± 27.4 | -4E+03 ± 6.88E+03 |
| | | True | -120 ± 243 | -24.6 ± 24.6 | -210 ± 296 | **3.22 ± 1.14** | 1.63 ± 1.57 | **-19.6 ± 27.8** | -4.14E+03 ± 7.16E+03 |

Table 5.2 The per data-point validation log likelihood of fitting various combinations of kernels to the performance characteristics of BNNs trained on various datasets. Training sets are 10 samples randomly selected. Validation set is the remaining data. Each experiment run 5 times with different seeds. The mean and standard deviation of the results are reported.

| Markov length | Model settings: Input scales tied | Joint trained | Dataset: bostonHousing | concrete | energy | kin8nm | power-plant | wine-quality-red | yacht |
|---|---|---|---|---|---|---|---|---|---|
| False | False | False | -2.64 ± 0.351 | -1.86 ± 0.429 | -0.649 ± 0.2 | 1.37 ± 0.304 | -0.437 ± 0.299 | -0.874 ± 0.246 | -51 ± 98.7 |
| | | True | **-2.42 ± 0.269** | **-1.75 ± 0.374** | -0.49 ± 0.211 | 2.87 ± 0.409 | 0.167 ± 0.235 | -0.554 ± 0.168 | -48.9 ± 97.7 |
| | True | False | -2.79 ± 0.248 | -1.82 ± 0.457 | -0.529 ± 0.222 | 2.97 ± 0.4 | 0.0153 ± 0.245 | -0.611 ± 0.169 | -49.1 ± 96.8 |
| | | True | -2.56 ± 0.3 | **-1.75 ± 0.354** | **-0.391 ± 0.207** | **3.12 ± 0.432** | 0.196 ± 0.212 | -0.57 ± 0.187 | -5.27 ± 7.68 |
| True | False | False | -2.64 ± 0.352 | -1.86 ± 0.429 | -0.644 ± 0.203 | 1.4 ± 0.31 | -0.388 ± 0.306 | -0.797 ± 0.213 | -50.5 ± 98.3 |
| | | True | **-2.41 ± 0.265** | **-1.74 ± 0.37** | -0.489 ± 0.212 | 2.88 ± 0.412 | 0.197 ± 0.26 | **-0.529 ± 0.16** | -48.7 ± 97.2 |
| | True | False | -2.78 ± 0.303 | -1.82 ± 0.458 | -0.527 ± 0.222 | 2.95 ± 0.431 | 0.14 ± 0.13 | -0.6 ± 0.174 | -48.7 ± 96.8 |
| | | True | -2.56 ± 0.284 | -1.77 ± 0.39 | -0.387 ± 0.208 | **3.1 ± 0.44** | **0.245 ± 0.223** | **-0.531 ± 0.161** | -48.1 ± 96.9 |

Table 5.3 The per data-point validation log likelihood of fitting various combinations of model hyper-parameter to the performance characteristics of BNNs trained on various datasets. Training sets are 50% of the samples randomly selected. Validation set is the remaining data. Each experiment run 5 times with different seeds. The mean and standard deviation of the results are reported.

| Markov length | Model settings: Input scales tied | Joint trained | Dataset: bostonHousing | concrete | energy | kin8nm | power-plant | wine-quality-red | yacht |
|---|---|---|---|---|---|---|---|---|---|
| False | False | False | -123 ± 242 | -23.1 ± 25.9 | -185 ± 386 | 1.49 ± 2.43 | 0.233 ± 2.48 | -19.9 ± 27.4 | -3.93E+03 ± 6.83E+03 |
| | | True | -124 ± 255 | -22.9 ± 27.3 | -96.6 ± 136 | 3.19 ± 1.25 | 1.27 ± 1.48 | -19.9 ± 27.8 | -3.25E+03 ± 5.05E+03 |
| | True | False | -118 ± 234 | -21.2 ± 23.6 | -580 ± 586 | 2.8 ± 1.37 | 1.37 ± 2.23 | -33.7 ± 37.9 | -4.02E+03 ± 6.92E+03 |
| | | True | -120 ± 241 | -36.2 ± 32.9 | -457 ± 451 | 1.11 ± 4.54 | 1.71 ± 1.75 | -33.5 ± 37.9 | -4.24E+03 ± 7.37E+03 |
| True | False | False | -123 ± 241 | -23 ± 26 | -169 ± 350 | 1.1 ± 2.12 | 0.53 ± 1.96 | -19.8 ± 27.4 | -3.92E+03 ± 6.8E+03 |
| | | True | -105 ± 210 | -24 ± 29.3 | -144 ± 207 | 2.78 ± 1.47 | 1.31 ± 1.5 | -19.7 ± 27.9 | -3.18E+03 ± 5.09E+03 |
| | True | False | -121 ± 242 | -21.3 ± 24.4 | -379 ± 531 | 2.97 ± 1.35 | 1.34 ± 2.21 | -19.8 ± 27.4 | -4E+03 ± 6.88E+03 |
| | | True | -120 ± 243 | -24.6 ± 24.6 | -210 ± 296 | 3.22 ± 1.14 | 1.63 ± 1.57 | -19.6 ± 27.8 | -4.14E+03 ± 7.16E+03 |

Table 5.4 The per data-point validation log likelihood of fitting various combinations of model hyper-parameter to the performance characteristics of BNNs trained on various datasets. Training sets are 10 samples randomly selected. Validation set is the remaining data. Each experiment run 5 times with different seeds. The mean and standard deviation of the results are reported.

Bound. The final search method is Random Search as an ablative baseline. At each iteration, the algorithm picks 4 new points to investigate, in the random search case fully training the network, and in the case of the GPAR models, advancing the training by one output stage. 4 choices were used to speed up the algorithm, as sampling a single point at a time proved too slow to run.

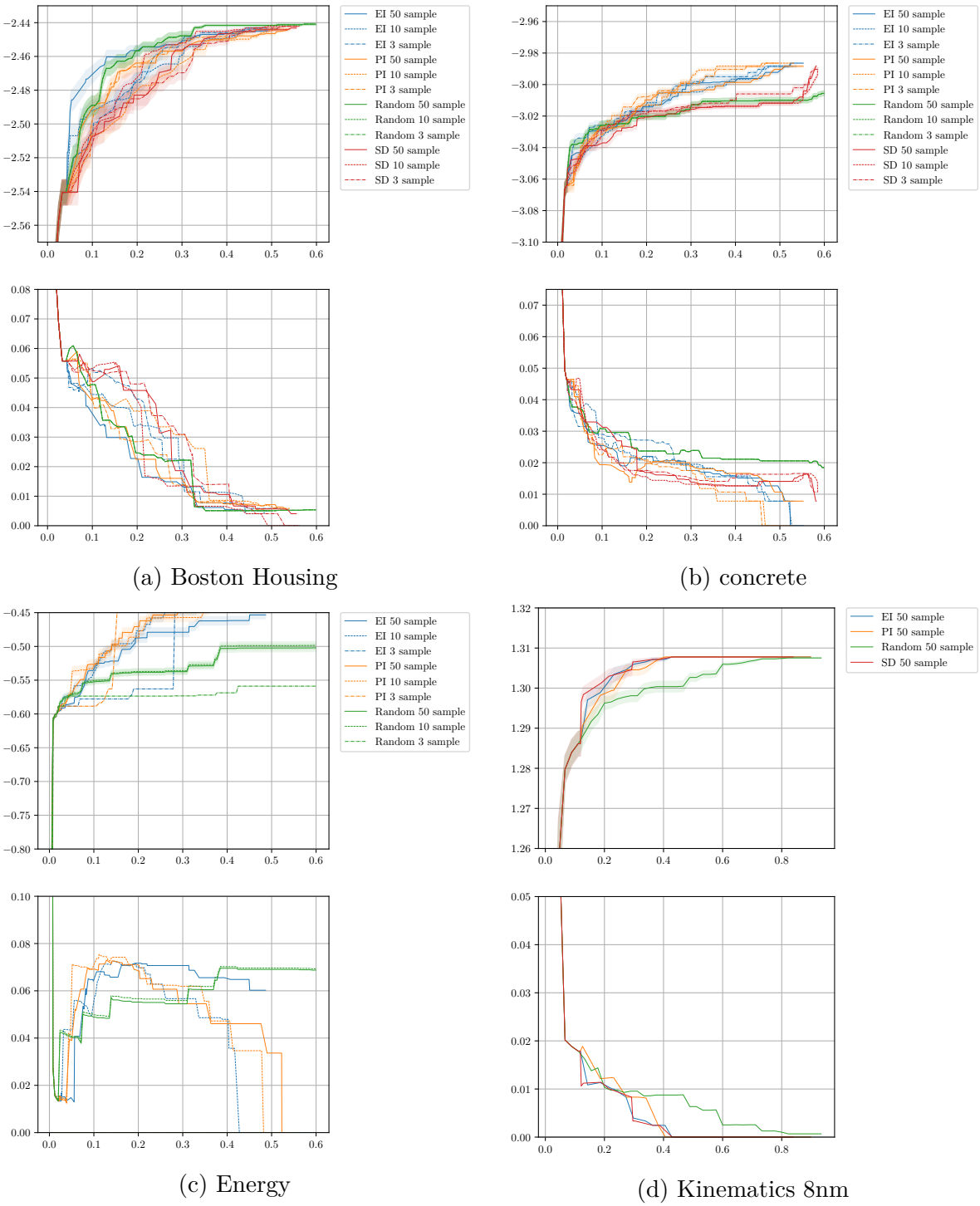
Also investigated in these plots are the effects of deliberately under-sampling when estimating the posterior of the Gaussian Process function. Since multiple points are being tested at each iteration, we want to diversify the points selected, else it is likely we will end up testing multiple points in a very similar location at the maxima of the acquisition function. By using low cardinality independent sets of samples drawn from the model to estimate the posterior function, we hope to inject some stochastic noise in to he maxima of the acquisition function, and spread out the next points to sample among a group of the best options.

The progressions of the search are plotted in terms of the fraction of the total points in the search space that have been sampled.

The results of these searches are seen in figures 5.6 and 5.7. In general the results of these searches are promising. After the initial random search, the GPAR methods pull away from random search, both in terms of the mean best value found, and the standard deviation of the best value found, implying that the GPAR methods also more consistently find better solutions.

In a **couple of cases** however, we see that random search outperforms the Bayesian optimisation methods, for example on the Boston Housing set. EI is competitive, but random search pulls ahead further into the search. An explanation from this comes from re-examining the figures 5.4 and 5.5 . In problems where random search is competitive, we see that the results of BNN training are significantly more noisy than in other problems (e.g. in the Boston Housing problem). This excessive noise on the data will make model fitting and therefore optimisation tricky due to the inseparability of the effect of the number of layers, making random search more competitive.

**On synthetic functions**



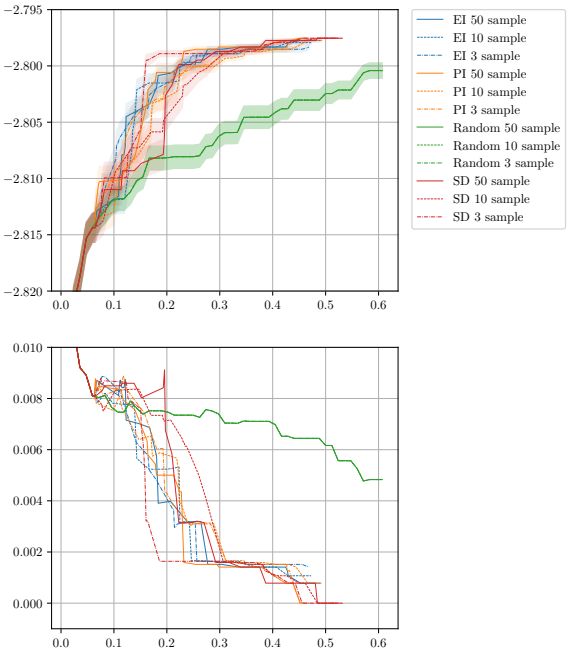

(a) Boston Housing

(b) concrete

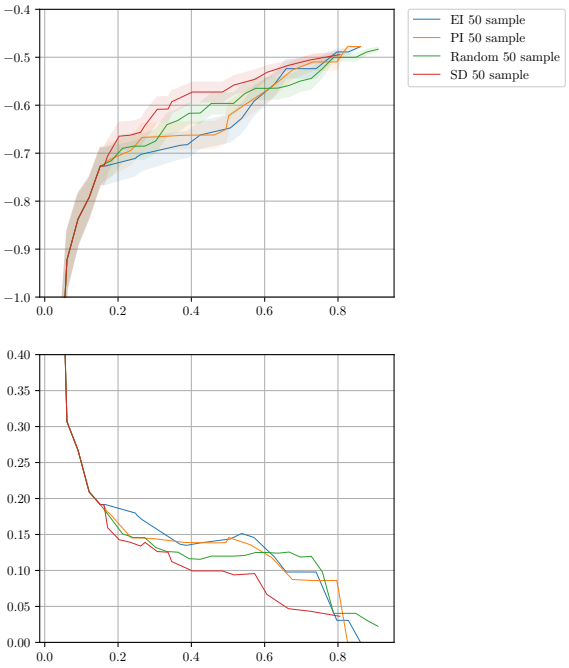
(c) Energy

(d) Kinematics 8nm

Fig. 5.6 The search efficiency of searching the architecture space of MLP BNNs on various datasets, utilising the GPAR method with 3 checkpoints of training monitored. Acquisition function used are Expected Improvement (EI), Probability of Improvement (PI), Upper Confidence Bound (SD) and random search. 50, 10, and 3 samples were tried for estimating the mean and variance of the GP posterior function.




(a) Power Plant




(b) Yacht

Fig. 5.7 The search efficiency of searching the architecture space of MLP BNNs on various datasets, utilising the GPAR method with 3 checkpoints of training monitored. Acquisition function used are Expected Improvement (EI), Probability of Improvement (PI), Upper Confidence Bound (SD) and random search. 50, 10, and 3 samples were tried for estimating the mean and variance of the GP posterior function.

# Chapter 6

# Conclusion

Presented here are two main contributions.

First a detailed study of the effects of various hyper-parameters on the performance of MLP structure BNNs, and identification of systematic trends. Also investigated are the detail of the balance made between the cost terms in the ELBO loss, and its effect on the performance of networks, and on the pruning of units in networks.

Second is a new method for searching for optimal MLP structure networks in Bayesian Neural Networks, a task currently untackled. This method robustly outperforms the random search on the majority of tasks. [**Does it beat out GP search?**]

## 6.1 Future study

This thesis

is thesis the correct terminology here?

represents a first step into architecture search for BNNs. Two directions of future work present as interesting.

First a continuation of this specific search method for architecture search, or for hyper-parameter optimisation in general. The ability to incorporate information from snapshots of time during training to allow for early stopping is a reasonably uncommon property in search mechanisms, and this method presented here is entirely novel.

Several immediate improvements could be investigated. The current acquisition functions used are inherently greedy algorithms. Trading off for some additional exploration may help the search rapidly identify good regions to explore. Brochu et al. (2010) investigates methods for altering the exploration-exploitation balance in the acquisition functions used in this project. Ginsbourger et al. (2008) introduces methodology to do parallel processing

EI Bayesian optimisation. Instead of the stochastic noise introduced in this project to diversify the selected search points, this work introduces mutli-step optimal acquisition functions that may provide significant gains in performance of this model. Hennig and Schuler CHRISTIANSCHULER (2012) proposes a new acquisition function attempts to minimise the entropy of the predictive models posterior. The objective of this is to optimally find the *loction* of the maxima, not nessicaraliy explore it. Application of this to the search space may speed up the process of finding optimal architectures

The other line of investigation is to apply some of the more modern architecture search techniques to BNNs. Recent efforts have scaled BNNs to the size of some larger CNN models, and so automated search becomes even more important at this scale. It is not immediately clear how to apply network morphisms or one-shot modelling to BNNs, but the results of this would be intriguing to investigate.

# References


Adam, G. and Lorraine, J. (2019). Understanding Neural Architecture Search Techniques. Technical report.

Baker, B., Gupta, O., Naik, N., and Raskar, R. (2016). Designing Neural Network Architectures using Reinforcement Learning.

Baker, B., Gupta, O., Raskar, R., and Naik, N. (2017). Accelerating Neural Architecture Search using Performance Prediction.

Bender, G., Kindermans, P.-J., Zoph, B., Vasudevan, V., and Le, Q. (2018). Understanding and Simplifying One-Shot Architecture Search. Technical report.

Bergstra, J. (2010). Algorithms for Hyper-Parameter Optimization. In *NIPS*, pages 1–9.

Bergstra, J., Yamins, D., and Cox, D. D. (2013). Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures.

Blundell, C., Cornebise, J., Kavukcuoglu, K., and Wierstra, D. (2015). Weight Uncertainty in Neural Networks.

Brochu, E., Cora, V. M., and De Freitas, N. (2010). A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning. Technical report.

Brock, A., Lim, T., Ritchie, J. M., and Weston, N. (2017). SMASH: One-Shot Model Architecture Search through HyperNetworks.

Bui, T., Hernández-Lobato, D., Hernandez-Lobato, J., Li, Y., and Turner, R. (2016). Deep gaussian processes for regression using approximate expectation propagation. In *International Conference on Machine Learning*, pages 1472–1481.

Cai, H., Zhu, L., and Han, S. (2018). ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware.

Cortes, C., Gonzalvo, X., Kuznetsov, V., Mohri, M., and Yang, S. (2016). AdaNet: Adaptive Structural Learning of Artificial Neural Networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 874–883. JMLR. org.

Cox, D. D. and John, S. (1992). A statistical method for global optimization. In *[Proceedings] 1992 IEEE International Conference on Systems, Man, and Cybernetics*, pages 1241–1246. IEEE.

Domhan, T., Springenberg, J. T., and Hutter, F. (2015). Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *IJCAI International Joint Conference on Artificial Intelligence*, volume 2015-Janua, pages 3460–3468.

Duvenaud, D. (2014). *Automatic model construction with Gaussian processes*. PhD thesis, University of Cambridge.

Elsken, T., Metzen, J.-H., and Hutter, F. (2017). Simple And Efficient Architecture Search for Convolutional Neural Networks.

Elsken, T., Metzen, J. H., and Hutter, F. (2018). Neural Architecture Search: A Survey.

Elsken, T., Metzen, J. H., and Hutter, F. (2019). Efficient Multi-objective Neural Architecture Search via Lamarckian Evolution.

Floreano, D., Dürr, P., and Mattiussi, C. (2008). Neuroevolution: From architectures to learning. *Evolutionary Intelligence*, 1(1):47–62.

Fusi, N., Sheth, R., and Elibol, M. (2018). Probabilistic matrix factorization for automated machine learning. In *Advances in Neural Information Processing Systems*, pages 3352–3361.

Ginsbourger, D., Le Riche, R., and Carraro, L. (2008). A Multi-points Criterion for Deterministic Parallel Global Optimization based on Gaussian Processes A Multi-points Criterion for Deterministic Parallel Global Optimization based on Gaussian Processes A Multi-points Criterion for Deterministic Parallel Global Optimization based on Gaussian Processes. Technical report.

Graves, A. (2011). Practical variational inference for neural networks. In *Advances in neural information processing systems*, pages 2348–2356.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2016-Decem, pages 770–778.

Hennig, P. and Schuler CHRISTIANSCHULER, C. J. (2012). Entropy Search for Information-Efficient Global Optimization. Technical report.

Hinton, G., Hinton, G., and Van Camp, D. (1993). Keeping Neural Networks Simple by Minimizing the Description Length of the Weights. *IN PROC. OF THE 6TH ANN. ACM CONF. ON COMPUTATIONAL LEARNING THEORY*, pages 5—-13.

Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. (2017). Densely connected convolutional networks. Technical report.

Hutter, F., Kotthoff, L., and Vanschoren, J. (2018). *Automatic machine learning: methods, systems, challenges*.

Jenatton, R., Archambeau, C., González, J., and Seeger, M. (2017). Bayesian Optimization with Tree-structured Dependencies. In *International Conference on Machine Learning*, pages 1655–1664.

Jin, H., Song, Q., and Hu, X. (2018). Auto-Keras: An Efficient Neural Architecture Search System.

Jones, D. R., Schonlau, M., and Welch, W. J. (1998). Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492.

Jozefowicz, R., Zaremba, W., and Sutskever, I. (2015). An Empirical Exploration of Recurrent Network Architectures. Technical Report 3.

Kandasamy, K., Neiswanger, W., Schneider, J., Poczos, B., and Xing, E. (2018). Neural architecture search with bayesian optimisation and optimal transport. *arXiv preprint arXiv:1802.07191*.

Kingma, D. P., Salimans, T., and Welling, M. (2015). Variational Dropout and the Local Reparameterization Trick.

Kitano, H. (1990). Designing neural networks using genetic algorithms with graph generation system. *Complex systems*, 4(4):461–476.

Klein, A., Falkner, S., Bartels, S., Hennig, P., and Hutter, F. (2016). Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets.

Kushner, H. J. (1964). A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise. *Journal of Basic Engineering*, 86(1):97–106.

Li, L. and Talwalkar, A. (2019). Random Search and Reproducibility for Neural Architecture Search.

Liu, C., Zoph, B., Neumann, M., Shlens, J., Hua, W., Li, L.-J., Fei-Fei, L., Yuille, A., Huang, J., and Murphy, K. (2018a). Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 19–34.

Liu, H., Simonyan, K., Vinyals, O., Fernando, C., and Kavukcuoglu, K. (2017). Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*.

Liu, H., Simonyan, K., and Yang, Y. (2018b). Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*.

Mendoza, H., Klein, A., Feurer, M., Springenberg, J. T., and Hutter, F. (2016). Towards automatically-tuned neural networks. In *Workshop on Automatic Machine Learning*, pages 58–65.

Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Shahrzad, H., Navruzyan, A., Duffy, N., and Others (2019). Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pages 293–312. Elsevier.

Mockus, J., Tiesis, V., and Zilinskas, A. (1978). Toward global optimization. *The Application of Bayesian Methods for Seeking the Extremum*, 2:117–128.

Negrinho, R. and Gordon, G. (2017). Deeparchitect: Automatically designing and training deep architectures. *arXiv preprint arXiv:1704.08792*.

Peter Angeline, J., Gregory Saunders, M., and Jordan Pollack, P. (1994). An evolutionary algorithm that constructs recurrent neural networks. Technical Report 1.

Pham, H., Guan, M. Y., Zoph, B., Le, Q. V., and Dean, J. (2018). Efficient Neural Architecture Search via Parameter Sharing.

Real, E., Aggarwal, A., Huang, Y., and Le, Q. V. (2018). Regularized Evolution for Image Classifier Architecture Search.

Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y. L., Tan, J., Le, Q., and Kurakin, A. (2017). Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*.

Requeima, J., Tebbutt, W., Bruinsma, W., and Turner, R. E. (2018). The Gaussian Process Autoregressive Regression Model (GPAR).

Russo, D. J., Van Roy, B., Kazerouni, A., Osband, I., and Wen, Z. (2018). A Tutorial on Thompson Sampling. *Foundations and Trends® in Machine Learning*, 11(1):1–96.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal Policy Optimization Algorithms.

Sciuto, C., Yu, K., Jaggi, M., Musat, C., and Salzmann, M. (2019). Evaluating the Search Phase of Neural Architecture Search.

Shridhar, K., Laumann, F., and Liwicki, M. (2018). Uncertainty Estimations by Softplus normalization in Bayesian Convolutional Neural Networks with Variational Inference.

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959.

Stanley, K. O., Clune, J., Lehman, J., and Miikkulainen, R. (2018a). Designing neural networks through neuroevolution. *Nature Machine Intelligence*, 1(1):24–35.

Stanley, K. O., Clune, J., Lehman, J., and Miikkulainen, R. (2018b). Designing neural networks through neuroevolution. *Nature Machine Intelligence*, 1(1):24–35.

Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127.

Suganuma, M., Ozay, M., and Okatani, T. (2018). Exploiting the Potential of Standard Convolutional Autoencoders for Image Restoration by Evolutionary Search.

Swersky, K., Duvenaud, D., Snoek, J., Hutter, F., and Osborne, M. A. (2014a). Raiders of the lost architecture: Kernels for Bayesian optimization in conditional parameter spaces. *arXiv preprint arXiv:1409.4011*.

Swersky, K., Snoek, J., and Adams, R. P. (2014b). Freeze-Thaw Bayesian Optimization.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2016). Rethinking the Inception Architecture for Computer Vision. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2016-Decem, pages 2818–2826.

Thompson, W. R. (1933). On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples. *Biometrika*, 25(3/4):285.

Wang, L., Zhao, Y., Jinnai, Y., Tian, Y., and Fonseca, R. (2019). AlphaX: eXploring Neural Architectures with Deep Neural Networks and Monte Carlo Tree Search.

Wei, T., Wang, C., Rui, Y., and Chen, C. W. (2016). Network Morphism.

Williams, C. K. I. and Rasmussen, C. E. (2006). *Gaussian processes for machine learning*, volume 2. MIT Press Cambridge, MA.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256.

Wistuba, M. (2017). Finding Competitive Network Architectures Within a Day Using UCT.

Wu, A., Nowozin, S., Meeds, E., Turner, R. E., Hernández-Lobato, J. M., and Gaunt, A. L. (2018). Fixing variational bayes: Deterministic variational inference for bayesian neural networks. *arXiv preprint arXiv:1810.03958*.

Xie, L. and Yuille, A. (2017). Genetic cnn. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1379–1388.

Xie, S., Zheng, H., Liu, C., and Lin, L. (2018). SNAS: Stochastic Neural Architecture Search.

Zela, A., Klein, A., Falkner, S., and Hutter, F. (2018). Towards Automated Deep Learning: Efficient Joint Neural Architecture and Hyperparameter Search.

Zhang, Y., Bahadori, M. T., Su, H., and Sun, J. (2016). FLASH: fast Bayesian optimization for data analytic pipelines. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2065–2074. ACM.

Zhong, Z., Yan, J., and Liu, C.-L. (2017). Practical network blocks design with q-learning. *arXiv preprint arXiv:1708.05552*.

Zoph, B. and Le, Q. V. (2016). Neural Architecture Search with Reinforcement Learning. *arXiv preprint arXiv:1611.01578*.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. (2018). Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710.

Can I make the text size much smaller here?? Or do references not count towards page limits and word counts. Saves about a page per size drop