Neural Architecture Search with Bayesian Optimisation and Optimal Transport

Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabás Póczos, Eric P Xing
Machine Learning Department, Carnegie Mellon University
{kandasamy, willie, schneide, bapoczos, epxing}@cs.cmu.edu

Abstract
Bayesian Optimisation (BO) refers to a class of methods for global optimisation of a function $f$ which is only accessible via point evaluations. It is typically used in settings where $f$ is expensive to evaluate. A common use case for BO in machine learning is model selection, where it is not possible to analytically model the generalisation performance of a statistical model, and we resort to noisy and expensive training and validation procedures to choose the best model. Conventional BO methods have focused on Euclidean and categorical domains, which, in the context of model selection, only permits tuning scalar hyper-parameters of machine learning algorithms. However, with the surge of interest in deep learning, there is an increasing demand to tune neural network architectures. In this work, we develop NASBOT, a Gaussian process based BO framework for neural architecture search. To accomplish this, we develop a distance metric in the space of neural network architectures which can be computed efficiently via an optimal transport program. This distance might be of independent interest to the deep learning community as it may find applications outside of BO. We demonstrate that NASBOT outperforms other alternatives for architecture search in several cross validation based model selection tasks on multi-layer perceptrons and convolutional neural networks.

1 Introduction
In many real world problems, we are required to sequentially evaluate a noisy black-box function $f$ with the goal of finding its optimum in some domain $\mathcal{X}$. Typically, each evaluation is expensive in such applications, and we need to keep the number of evaluations to a minimum. Bayesian optimisation (BO) refers to an approach for global optimisation that is popularly used in such settings. It uses Bayesian models for $f$ to infer function values at unexplored regions and guide the selection of points for future evaluations. BO has been successfully applied for many optimisation problems in optimal policy search, industrial design, and scientific experimentation. That said, the quintessential use case for BO in machine learning is model selection [14, 37]. For instance, consider selecting the regularisation parameter $\lambda$ and kernel bandwidth $h$ for an SVM. We can set this up as a zeroth order optimisation problem where our domain is a two dimensional space of $(\lambda, h)$ values, and each function evaluation trains the SVM on a training set, and computes the accuracy on a validation set. The goal is to find the model, i.e. hyper-parameters, with the highest validation accuracy.

The majority of the BO literature has focused on settings where the domain $\mathcal{X}$ is either Euclidean or categorical. This suffices for many tasks, such as the SVM example above. However, with recent successes in deep learning, neural networks are increasingly becoming the method of choice for many machine learning applications. A number of recent work have designed novel neural network architectures to significantly outperform the previous state of the art [12, 13, 34, 42]. This motivates studying model selection over the space of neural architectures to optimise for generalisation performance. A critical challenge in this endeavour is that evaluating a network via train and validation procedures is very expensive. This paper proposes a BO framework for this problem.
While there are several approaches to BO, those based on Gaussian processes (GP) \cite{32} are most common in the BO literature. In its most unadorned form, a BO algorithm operates sequentially, starting at time 0 with a GP prior for \( f \); at time \( t \), it incorporates results of evaluations from \( 1, \ldots, t-1 \) in the form of a posterior for \( f \). It then uses this posterior to construct an acquisition function \( \varphi_t \), where \( \varphi_t(x) \) is a measure of the value of evaluating \( f \) at \( x \) at time \( t \) if our goal is to maximise \( f \). Accordingly, it chooses to evaluate \( f \) at the maximiser of the acquisition, i.e. \( x_t = \arg\max_{x \in X} \varphi_t(x) \).

There are two key ingredients to realising this plan for GP based BO. First, we need to quantify the similarity between two points \( x, x' \) in the domain in the form of a kernel \( \kappa(x, x') \). The kernel is needed to define the GP, which allows us to reason about an unobserved value \( f(x') \) when we have already evaluated \( f(x) \). Secondly, we need a method to maximise \( \varphi_t \).

These two steps are fairly straightforward in conventional domains. For example, in Euclidean spaces, we can use one of many popular kernels such as Gaussian, Laplacian, or Matérn; we can maximise \( \varphi_t \) via off the shelf branch-and-bound or gradient based methods. However, when each \( x \in X \) is a neural network architecture, this is not the case. Hence, our challenges in this work are two-fold. First, we need to quantify (dis)similarity between two networks. Intuitively, in Fig. 1, network 1a is more similar to network 1b, than it is to 1c. Secondly, we need to be able to traverse the space of such networks to optimise the acquisition function. Our main contributions are as follows.

1. We develop a (pseudo-)distance for neural network architectures called OTMANN (Optimal Transport Metrics for Architectures of Neural Networks) that can be computed efficiently via an optimal transport program.
2. We develop a BO framework for optimising functions on neural network architectures called NASBOT (Neural Architecture Search with Bayesian Optimisation and Optimal Transport). This includes an evolutionary algorithm to optimise the acquisition function.
3. Empirically, we demonstrate that NASBOT outperforms other baselines on model selection tasks for multi-layer perceptrons (MLP) and convolutional neural networks (CNN). Our python implementations of OTMANN and NASBOT are available at github.com/kirthevasank/nasbot.

**Related Work:** Recently, there has been a surge of interest in methods for neural architecture search \cite{1, 6, 8, 18, 22, 23, 27, 29, 33, 38, 48–51}. We discuss them in detail in the Appendix due to space constraints. Broadly, they fall into two categories, based on either evolutionary algorithms (EA) or reinforcement learning (RL). EA provide a simple mechanism to explore the space of architectures by making a sequence of changes to networks that have already been evaluated. However, as we will discuss later, they are not ideally suited for optimising functions that are expensive to evaluate. While RL methods have seen recent success, architecture search is in essence an optimisation problem – find the network with the lowest validation error. There is no explicit need to maintain a notion of state and solve credit assignment \cite{40}. Since RL is a fundamentally more difficult problem than optimisation \cite{16}, these approaches need to try a very large number of architectures to find the optimum. This is not desirable, especially in computationally constrained settings.

None of the above methods have been designed with a focus on the expense of evaluating a neural network, with an emphasis on being judicious in selecting which architecture to try next. Bayesian optimisation (BO) techniques, which use introspective Bayesian models to carefully determine future evaluations, are well suited for expensive evaluations. BO usually consumes more computation to determine future points than other methods, but this pays dividends when the evaluations are very expensive. While there has been some work on BO for architectures \cite{2, 15, 25, 37, 41}, they can only provide a simple mechanism to explore the space of architectures, e.g. Fig. 1a, but not Figs. 1b, 1c. We compare NASBOT to one such method and demonstrate that feed forward structures are inadequate for many problems.

## 2 Set Up

Our goal is to maximise a function \( f \) defined on a space \( X \) of neural network architectures. When we evaluate \( f \) at \( x \in X \), we obtain a possibly noisy observation \( y \) of \( f(x) \). In the context of architecture search, \( f \) is the performance on a validation set after \( x \) is trained on the training set. If \( x_\star = \arg\max_{x \in X} f(x) \) is the optimal architecture, and \( x_t \) is the architecture evaluated at time \( t \), we want \( f(x_t) - \max_{x \leq n} f(x) \) to vanish fast as the number of evaluations \( n \to \infty \). We begin with a review of BO and then present a graph theoretic formalism for neural network architectures.

### 2.1 A brief review of Gaussian Process based Bayesian Optimisation

A GP is a random process defined on some domain \( X \), and is characterised by a mean function \( \mu : X \to \mathbb{R} \) and a (covariance) kernel \( \kappa : X^2 \to \mathbb{R} \). Given \( n \) observations \( D_n = \{(x_i, y_i)\}_{i=1}^n \), where
We will use the CNNs in Fig. 1 to illustrate the concepts. A neural network $G_{\ell_u}$ chooses $c$ from a set of candidates $X$ and directs it to another node $X'$. The attribute $\ell_u$ is used to choose the best output $f$ from the decision layers perform decision layers. For a classification task, the decision layers perform softmax operations and output the probabilities an input datum belongs to each class. For regression, the decision layers perform linear combinations of the outputs of the previous layers and output a single scalar. All networks are equipped with self-attention mechanisms, which allow the network to compute attention weights for each input feature and focus on the most relevant features.

### 2.2 A Mathematical Formalism for Neural Networks

Our formalism will view a neural network as a graph whose vertices are the layers of the network. We will use the CNNs in Fig. 1 to illustrate the concepts. A neural network $G = (\mathcal{L}, \mathcal{E})$ is defined by a set of layers $\mathcal{L}$ and directed edges $\mathcal{E}$. An edge $(u, v) \in \mathcal{E}$ is a ordered pair of layers. In Fig. 1, the layers are depicted by rectangles and the edges by arrows. A layer $u \in \mathcal{L}$ is equipped with a layer label $\ell_l(u)$ which denotes the type of operations performed at the layer. For instance, in Fig. 1a, $\ell_l(1) = \text{conv}3$, $\ell_l(5) = \text{max-pool}$ denote a $3 \times 3$ convolution and a max-pooling operation. The attribute $\ell_u$ denotes the number of computational units in a layer. In Fig. 1b, $\ell_u(5) = 32$ and $\ell_u(7) = 16$ are the number of convolutional filters and fully connected nodes.

In addition, each network has decision layers which are used to obtain the predictions of the network. For a classification task, the decision layers perform softmax operations and output the probabilities an input datum belongs to each class. For regression, the decision layers perform linear combinations of the outputs of the previous layers and output a single scalar. All networks are equipped with self-attention mechanisms, which allow the network to compute attention weights for each input feature and focus on the most relevant features.
have at least one decision layer. When a network has multiple decision layers, we average the output of each decision layer to obtain the final output. The decision layers are shown in green in Fig. 1. Finally, every network has a unique input layer $u_{ip}$ and output layer $u_{op}$ with labels $\ell(u_{ip}) = 1p$ and $\ell(u_{op}) = op$. It is instructive to think of the role of $u_{ip}$ as feeding a data point to the network and the role of $u_{op}$ as averaging the results of the decision layers. The input and output layers are shown in pink in Fig. 1. We refer to all layers that are not input, output or decision layers as processing layers.

The directed edges are to be interpreted as follows. The output of each layer is fed to each of its children; so both layers 2 and 3 in Fig. 1b take the output of layer 1 as input. When a layer has multiple parents, the inputs are concatenated; so layer 5 sees an input of $16 + 16$ filtered channels coming in from layers 3 and 4. Finally, we mention that neural networks are also characterised by the values of the weights/parameters between layers. In architecture search, we typically do not consider these weights. Instead, an algorithm will (somewhat ideally) assume access to an optimisation oracle that can minimise the loss function on the training set and find the optimal weights.

We next describe a distance $d : \mathcal{X}^2 \rightarrow \mathbb{R}_+$ for neural architectures. Recall that our eventual goal is a kernel for the GP; given a distance $d$, we will aim for $\kappa(x, x') = e^{-\beta d(x, x')^p}$, where $\beta, p \in \mathbb{R}_+$, as the kernel. Many popular kernels take this form. For e.g. when $\mathcal{X} \subset \mathbb{R}^n$ and $d$ is the $L^2$ norm, $p = 1, 2$ correspond to the Laplacian and Gaussian kernels respectively.

3 The OTMANN Distance

To motivate this distance, note that the performance of a neural network is determined by the amount of computation at each layer, the types of these operations, and how the layers are connected. A meaningful distance should account for these factors. To that end, OTMANN is defined as the minimum of a matching scheme which attempts to match the computation at the layers of one network to the layers of the other. We incur penalties for matching layers with different types of operations or those at structurally different positions. We will find a matching that minimises these penalties, and the total penalty at the minimum will give rise to a distance. We first describe two concepts, layer masses and path lengths, which we will use to define OTMANN.

Layer masses: The layer masses $\ell m : L \rightarrow \mathbb{R}_+$ will be the quantity that we match between the layers of two networks when comparing them. $\ell m(u)$ quantifies the significance of layer $u$. For processing layers, $\ell m(u)$ will represent the amount of computation carried out by layer $u$ and is computed via the product of $\ell(u)$ and the number of incoming units. For example, in Fig. 1b, $\ell m(5) = 32 \times (16 + 16)$ as there are 16 filtered channels each coming from layers 3 and 4 respectively. As there is no computation at the input and output layers, we cannot define the layer mass directly as we did for the processing layers. Therefore, we use $\ell m(u_{ip}) = \ell m(u_{op}) = \zeta \sum_{u \in PC} \ell m(u)$ where $PC$ denotes the set of processing layers, and $\zeta \in (0, 1)$ is a parameter to be determined. Intuitively, we are using an amount of mass that is proportional to the amount of computation in the processing layers. Similarly, the decision layers occupy a significant role in the architecture as they directly influence the output. While there is computation being performed at these layers, this might be problem dependent – there is more computation performed at the softmax layer in a 10 class classification problem than in a 2 class problem. Furthermore, we found that setting the layer mass for decisions layers based on computation underestimates their contribution to the network. Following the same intuition as we did for the input/output layers, we assign an amount of mass proportional to the mass in the processing layers. Since the outputs of the decision layers are averaged, we distribute the mass among all decision layers; that is, if $DL$ are decision layers, $\forall u \in DL$, $\ell m(u) = \frac{\zeta}{|DL|} \sum_{u \in PC} \ell m(u)$. In all our experiments, we use $\zeta = 0.1$. In Fig. 1, the layer masses for each layer are shown in parentheses.

Path lengths from/to $u_{ip}$: In a neural network $G$, a path from $u$ to $v$ is a sequence of layers $u_1, \ldots, u_s$ where $u_1 = u$, $u_s = v$ and $(u_i, u_{i+1}) \in E$ for all $i \leq s - 1$. The length of this path is the number of hops from one node to another in order to get from $u$ to $v$. For example, in Fig. 1c, $(2, 5, 8, 13)$ is a path from layer 2 to 13 of length 3. Let the shortest (longest) path length from $u$ to $v$ be the smallest (largest) number of hops from one node to another among all paths from $u$ to $v$. Additionally, define the random walk path length as the expected number of hops to get from $u$ to $v$ if, from any layer we hop to one of its children chosen uniformly at random. For example, in Fig. 1c, the shortest, longest and random walk path lengths from layer 1 to layer 14 are 5, 7, and 5.67 respectively. For any $u \in L$, let $\delta_{ip}^p(u), \delta_{op}^p(u), \delta_{op}^w(u)$ denote the length of the shortest, longest and random walk paths from $u$ to the output $u_{op}$. Similarly, let $\delta_{ip}^p(u), \delta_{ip}^w(u)$ denote the corresponding lengths
for walks from the input $u_{ip}$ to $u$. As the layers of a neural network can be topologically ordered\(^1\), the above path lengths are well defined and finite. Further, for any $s \in \{\text{sp, lp, rw}\}$ and $t \in \{\text{ip, op}\}$, $\delta^s_t(u)$ can be computed for all $u \in L$, in $O(|E|)$ time (see Appendix A.3 for details).

We are now ready to describe OTMANN. Given two networks $G_1 = (L_1, E_1), G_2 = (L_2, E_2)$ with $n_1, n_2$ layers respectively, we will attempt to match the layer masses in both networks. We let $Z \in \mathbb{R}^{n_1 \times n_2}$ be such that $Z(i, j)$ denotes the amount of mass matched between layer $i \in G_1$ and $j \in G_2$. The OTMANN distance is computed by solving the following optimisation problem.

\[
\begin{align*}
\text{minimise} & \quad \phi_{lmm}(Z) + \phi_{nas}(Z) + \nu_{str} \phi_{str}(Z) \\
\text{subject to} & \quad \sum_{j \in L_2} Z_{ij} \leq \ell m(i), \sum_{i \in L_1} Z_{ij} \leq \ell m(j), \forall i, j
\end{align*}
\]

The label mismatch term $\phi_{lmm}$ penalises matching masses that have different labels, while the structural term $\phi_{str}$ penalises matching masses at structurally different positions with respect to each other. If we choose not to match any mass in either network, we incur a non-assignment penalty $\phi_{nas}$. $\nu_{str} > 0$ determines the trade-off between the structural and other terms. The inequality constraints ensure that we do not over assign the masses in a layer. We now describe $\phi_{lmm}, \phi_{nas}, \text{ and } \phi_{str}$.

**Label mismatch penalty $\phi_{lmm}$**: We begin with a label penalty matrix $M \in \mathbb{R}^{L \times L}$ where $L$ is the number of all label types and $M(x, y)$ denotes the penalty for transporting a unit mass from a layer with label $y$ to a layer with label $y$. We then construct a matrix $C_{lmm} \in \mathbb{R}^{n_1 \times n_2}$ with $C_{lmm}(i, j) = M(\ell(i), \ell(j))$ corresponding to the mislabel cost for matching unit mass from each layer $i \in L_1$ to each layer $j \in L_2$. We then set $\phi_{lmm}(Z) = \sum_{i \in L_1, j \in L_2} Z(i, j) C(i, j)$ to be the sum of all matchings from $L_1$ to $L_2$ weighted by the label penalty terms. This matrix $M$, illustrated in Table 1, is a parameter that needs to be specified for OTMANN. They can be specified with an intuitive understanding of the functionality of the layers; e.g. many values in $M$ are $\infty$, while for similar layers, we choose a value less than 1.

**Non-assignment penalty $\phi_{nas}$**: We set this to be the amount of mass that is unassigned in both networks, i.e. $\phi_{nas}(Z) = \sum_{i \in L_1} \ell m(i) - \sum_{j \in L_2} Z_{ij} + \sum_{j \in L_2} \ell m(j) - \sum_{i \in L_1} Z_{ij}$. This essentially implies that the cost for not assigning unit mass is 1. The costs in Table 1 are defined relative to this. For similar layers $x, y$, $M(x, y) = 1$ and for disparate layers $M(x, y) > 1$. That is, we would rather match conv3 to conv5 than not assign it, provided the structural penalty for doing so is small; conversely, we would rather not assign a conv3, than assign it to fc. This also explains why we did not use a trade-off parameter like $\nu_{str}$ for $\phi_{lmm}$ and $\phi_{nas}$—it is simple to specify reasonable values for $M(x, y)$ from an understanding of their functionality.

**Structural penalty $\phi_{str}$**: We define a matrix $C_{str} \in \mathbb{R}^{n_1 \times n_2}$ where $C_{str}(i, j)$ is small if layers $i \in L_1$ and $j \in L_2$ are at structurally similar positions in their respective networks. We then set $\phi_{str}(Z) = (Z, C_{str})$. For $i \in L_1, j \in L_2$, we let $C_{str}(i, j) = \frac{1}{n} \sum_{s \in \{\text{sp, lp, rw}\}} \sum_{t \in \{\text{ip, op}\}} |\delta^s_t(i) - \delta^s_t(j)|$ be the average of all path length differences, where $\delta^s_t$ are the path lengths defined previously. We define $\phi_{str}$ in terms of the shortest/longest/random-walk path lengths from/to the input/output, because they capture various notions of information flow in a neural network; a layer’s input is influenced by the paths the data takes before reaching the layer and its output influences all layers it passes through before reaching the decision layers. If the path lengths are similar for two layers, they are likely to be at similar structural positions. Further, this form allows us to solve (3) efficiently via an OT program and prove distance properties about the solution. If we need to compute pairwise distances for several networks, as is the case in BO, the path lengths can be pre-computed in $O(|E|)$ time, and used to construct $C_{str}$ for two networks at the moment of computing the distance between them.

This completes the description of our matching program. In Appendix A, we prove that (3) can be formulated as an Optimal Transport (OT) program \cite{OT}. OT is a well studied problem with several efficient solvers \cite{sinkhorn}. Our theorem below, shows that the solution of (3) is a distance.

\(^1\)A topological ordering is an ordering of the layers $u_1, \ldots, u_{|E|}$ such that $u$ comes before $v$ if $(u, v) \in E$. 

<table>
<thead>
<tr>
<th>conv3</th>
<th>conv5</th>
<th>max-pool</th>
<th>avg-pool</th>
<th>fc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.2</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>0.2</td>
<td>0</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>0</td>
</tr>
<tr>
<td>$\infty$</td>
<td>$\infty$</td>
<td>0</td>
<td>0.25</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$\infty$</td>
<td>$\infty$</td>
<td>0.25</td>
<td>0</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1: An example label mismatch cost matrix $M$. There is zero cost for matching identical layers, < 1 cost for similar layers, and infinite cost for disparate layers.
We use an evolutionary algorithm (EA) approach to optimise the acquisition function (2). For this, we begin with an initial pool of networks and evaluate the acquisition $\varphi_t$ on those networks. Then we generate a set of $N_{\text{mut}}$ mutations of this pool as follows. First, we stochastically select $N_{\text{mut}}$ candidates from the set of networks already evaluated such that those with higher $\varphi$ values are more likely to be selected than those with lower values. Then we modify each candidate, to produce a new architecture. These modifications, described in Table 2, might change the architecture either by...
increasing or decreasing the number of computational units in a layer, by adding or deleting layers, or by changing the connectivity of existing layers. Finally, we evaluate the acquisition on this $N_{\text{mut}}$ mutations, add it to the initial pool, and repeat for the prescribed number of steps. While EA works fine for cheap functions, such as the acquisition $\varphi_t$ which is analytically available, it is not suitable when evaluations are expensive, such as training a neural network. This is because EA selects points for future evaluations that are already close to points that have been evaluated, and is hence inefficient at exploring the space. In our experiments, we compare NASBOT to the same EA scheme used to optimise the acquisition and demonstrate the former outperforms the latter.

We conclude this section by observing that this framework for NASBOT/OTMANN has additional flexibility to what has been described. Suppose one wishes to also tune over drop-out probabilities, regularisation penalties and batch normalisation at each layer. These can be treated as part of the layer label, and we can design an augmented label penalty matrix $M$ which accounts for these considerations. If one wishes to jointly tune other scalar hyper-parameters (e.g. learning rate), they can use an existing kernel for euclidean spaces and define the GP over the joint architecture + hyper-parameter space via a product kernel. BO methods for early stopping in iterative training procedures [17, 19] can be easily incorporated by defining a fidelity space. Using a line of work in scalable GPs [36, 47], one can apply our methods to challenging problems which might require trying a very large number ($\sim 100K$) of architectures. These extensions will enable deploying NASBOT in large scale settings, but are tangential to our goal of introducing a BO method for architecture search.

5 Experiments

Methods: We compare NASBOT to the following baselines. RAND: random search; EA (Evolutionary algorithm): the same EA procedure described above. TreeBO [15]: a BO method which only searches over feed forward structures. Random search is a natural baseline to compare optimisation methods. However, unlike in Euclidean spaces, there is no natural way to randomly explore the space of architectures. Our RAND implementation, operates in exactly the same way as NASBOT, except that the EA procedure is fed a random sample from $\text{Unif}(0, 1)$ instead of the GP acquisition each time it evaluates an architecture. Hence, RAND is effectively picking a random network from the same space explored by NASBOT; neither method has an unfair advantage because it considers a different space. While there are other methods for architecture search, their implementations are highly nontrivial and are not made available.

Datasets: We use the following datasets: blog feedback [4], indoor location [43], slice localisation [11], naval propulsion [5], protein tertiary structure [31], news popularity [7], Cifar10 [21]. The first six are regression problems for which we use MLPs. The last is a classification task on images for which we use CNNs. Table 3 gives the size and dimensionality of each dataset. For the first 6 datasets, we use a $0.6 – 0.2 – 0.2$ train-validation-test split and normalised the input and output to have zero mean and unit variance. Hence, a constant predictor will have a mean squared error of approximately 1. For Cifar10 we use $40K$ for training and $10K$ each for validation and testing.
We described NASBOT, a BO framework for neural architecture search. NASBOT finds better architectures for MLPs and CNNs more efficiently than other baselines on several datasets. A key contribution of this work is the efficiently computable OTMANN distance for neural network architectures, which may be of independent interest as it might find applications outside of BO. Our code for NASBOT and OTMANN will be made available.

## 6 Conclusion

We described NASBOT, a BO framework for neural architecture search. NASBOT finds better architectures for MLPs and CNNs more efficiently than other baselines on several datasets.
Acknowledgements

We would like to thank Guru Guruganesh and Dougal Sutherland for the insightful discussions. This research is partly funded by DOE grant DESC0011114. KK is supported by a Facebook fellowship and a Siebel scholarship.

References


A Additional Details on OTMANN

A.1 Optimal Transport Reformulation

We begin with a review of optimal transport. Throughout this section, \( \langle \cdot, \cdot \rangle \) denotes the Frobenius dot product. \( 1_n, 0_n \in \mathbb{R}^n \) denote a vector of ones and zeros respectively.

A review of Optimal Transport [44]: Let \( y_1 \in \mathbb{R}^{n_1}, y_2 \in \mathbb{R}^{n_2} \) be such that \( 1_{n_1}^T y_1 = 1_{n_2}^T y_2 \). Let \( C \in \mathbb{R}^{n_1 \times n_2} \). The following optimisation problem,

\[
\begin{align*}
\text{minimise} & \quad \langle Z, C \rangle \\
\text{subject to} & \quad Z > 0, \quad Z 1_{n_2} = y_1, \quad Z^T 1_{n_1} = y_2.
\end{align*}
\]

is called an optimal transport program. One interpretation of this set up is that \( y_1 \) denotes the supplies at \( n_1 \) warehouses, \( y_2 \) denotes the demands at \( n_2 \) retail stores, \( C_{ij} \) denotes the cost of transporting a unit mass of supplies from warehouse \( i \) to store \( j \) and \( Z_{ij} \) denotes the mass of material transported from \( i \) to \( j \). The program attempts to find a transportation plan which minimises the total cost of transportation \( \langle Z, C \rangle \).

**OT formulation of (3):** We now describe the OT formulation of the OTMANN distance. In addition to providing an efficient way to solve (3), the OT formulation will allow us to prove the metric properties of the solution. When computing the distance between \( G_1, G_2 \), for \( i = 1, 2 \), let \( tm(G_1) = \sum_{u \in \mathcal{L}_i} \ell m(u) \) denote the total mass in \( G_1 \), and \( \bar{n}_i = n_i + 1 \) where \( n_i = |\mathcal{L}_i| \). \( y_1 = \{ (\ell m(u), tm(G_2)) \}_{u \in \mathcal{L}_1} \in \mathbb{R}^{n_1} \) will be the supplies in our OT problem, and \( y_2 = \{ (\ell m(u), tm(G_1)) \}_{u \in \mathcal{L}_2} \in \mathbb{R}^{n_2} \) will be the demands. To define the cost matrix, we augment the mislabel and structural penalty matrices \( C_{\text{imm}}, C_{\text{str}} \) with an additional row and column of zeros; i.e. \( C'_{\text{imm}} = [0, 1_{n_1 \times n_2}^T] \) \( C'_{\text{str}} \) is defined similarly. Let \( C'_{\text{nas}} = [0, 1_{n_1 \times n_2}^T, 0] \in \mathbb{R}^{n_1 \times n_2} \). We will show that (3) is equivalent to the following OT program.

\[
\begin{align*}
\text{minimise} & \quad \langle Z', C' \rangle \\
\text{subject to} & \quad Z' 1_{n_2} = y_1, \quad Z'^T 1_{\bar{n}_1} = y_2.
\end{align*}
\]

One interpretation of (5) is that the last row/column appended to the cost matrices serve as a non-assignment layer and that the cost for transporting unit mass to this layer from all other layers is 1. The costs for mislabelling was defined relative to this non-assignment cost. The costs for similar layers is much smaller than 1; therefore, the optimiser is incentivised to transport mass among similar layers rather than not assign it provided that the structural penalty is not too large. Correspondingly, the cost for very disparate layers is much larger so that we would never match, say, a convolutional layer with a pooling layer. In fact, the \( \infty \)'s in Table 1 can be replaced by any value larger than 2 and the solution will be the same. The following theorem shows that (3) and (5) are equivalent.

**Theorem 2.** Problems (3) and (5) are equivalent, in that they both have the same minimum and we can recover the solution of one from the other.

**Proof.** We will show that there exists a bijection between feasible points in both problems with the same value for the objective. First let \( Z \in \mathbb{R}^{n_1 \times n_2} \) be a feasible point for (3). Let \( Z' \in \mathbb{R}^{n_1 \times n_2} \) be such that its first \( n_1 \times n_2 \) block is \( Z \) and, \( Z_{\bar{n}_1,j} = \sum_{i=1}^{n_2} Z_{ij} \), \( Z_{\bar{n}_1,\bar{n}_2} = \sum_{j=1}^{n_2} Z_{ij} \), and \( Z_{\bar{n}_1,\bar{n}_2} = \sum_{ij} Z_{ij} \). Then, for all \( i \leq n_1, \sum_{j} Z'_{ij} = \ell m(j) \) and \( \sum_{j} Z'_{\bar{n}_1,j} = \sum_{j} \ell m(j) - \sum_{ij} Z_{ij} + Z_{\bar{n}_1,\bar{n}_2} = tm(G_2) \). We then have, \( Z' 1_{n_2} = y_1 \) Similarly, we can show \( Z'^T 1_{\bar{n}_1} = y_2 \). Therefore, \( Z' \) is feasible for (5). We see that the objectives are equal via simple calculations,

\[
\begin{align*}
\langle Z', C' \rangle & = \langle Z', C'_{\text{imm}} + C'_{\text{str}} \rangle + \langle Z', C'_{\text{nas}} \rangle \\
& = \langle Z, C_{\text{imm}} + C_{\text{str}} \rangle + \sum_{j=1}^{n_2} Z'_{ij} + \sum_{i=1}^{n_1} Z'_{ij} \\
& = \langle Z, C_{\text{imm}} \rangle + \langle Z, C_{\text{str}} \rangle + \sum_{i \in \mathcal{L}_1} (\ell m(i) - \sum_{j \in \mathcal{L}_2} Z_{ij}) + \sum_{j \in \mathcal{L}_2} (\ell m(j) - \sum_{i \in \mathcal{L}_1} Z_{ij}).
\end{align*}
\]

The converse also follows via a straightforward argument. For given \( Z' \) that is feasible for (5), we let \( Z \) be the first \( n_1 \times n_2 \) block. By the equality constraints and non-negativity of \( Z', Z \) is feasible for (3). By reversing the argument in (6) we see that the objectives are also equal. \( \square \)
A.2 Distance Properties of OTMANN

The following theorem shows that the solution of (3) is a pseudo-distance. This is a formal version of Theorem 1 in the main text.

**Theorem 3.** Assume that the mislabel cost matrix $M$ satisfies the triangle inequality; i.e. for all labels $x, y, z$ we have $M(x, z) \leq M(x, y) + M(y, z)$. Let $d(G_1, G_2)$ be the solution of (3) for networks $G_1, G_2$. Then $d(\cdot, \cdot)$ is a pseudo-distance. That is, for all networks $G_1, G_2, G_3$, it satisfies, $d(G_1, G_2) > 0$, $d(G_1, G_2) = d(G_2, G_1)$, $d(G_1, G_1) = 0$ and $d(G_1, G_3) \leq d(G_1, G_2) + d(G_2, G_3)$.

Some remarks are in order. First, observe that while $d(\cdot, \cdot)$ is a pseudo-distance, it is not a distance; i.e. $d(G_1, G_2) = 0 \not\Rightarrow G_1 = G_2$. For example, while the networks in Figure 3 have different descriptors according to our formalism in Section 2.2, their distance is 0. However, it is not hard to see that their functionality is the same – in both cases, the output of layer 1 is passed through 16 conv3 filters and then fed to a layer with 32 conv3 filters – and hence, this property is desirable in this example. It is not yet clear however, if the topology induced by our metric equates two functionally dissimilar networks. We leave it to future work to study equivalence classes induced by the OTMANN distance. Second, despite the OT formulation, this is not a Wasserstein distance. In particular, the supports of the masses and the cost matrices change depending on the two networks being compared.

**Proof of Theorem 3.** We will use the OT formulation (5) in this proof. The first three properties are straightforward. Non-negativity follows from non-negativity of $Z', C'$ in (5). It is symmetric since the cost matrix for $d(G_2, G_1)$ is $C'^T$ if the cost matrix for $d(G_1, G_2)$ is $C$ and $(Z', C') = (Z'^T, C'^T)$ for all $Z'$. We also have $d(G_1, G_1) = 0$ since, then, $C'$ has a zero diagonal.

To prove the triangle inequality, we will use a gluing lemma, similar to what is used in the proof of Wasserstein distances [30]. Let $G_1, G_2, G_3$ be given and $m_1, m_2, m_3$ be their total masses. Let the solutions to $d(G_1, G_2)$ and $d(G_2, G_3)$ be $P \in \mathbb{R}^{n_1 \times n_2}$ and $Q \in \mathbb{R}^{n_2 \times n_3}$ respectively. When solving (5), we see that adding extra mass to the non-assignment layers does not change the objective, as one optimiser can transport mass between the two layers with 0 cost. Hence, we can assume w.l.o.g that (5) was solved with $y_i = \{tm(u)\}_{u \in L_i}, \left(\sum_{j \in \{1, 2, 3\}} tm(G_j) - tm(G_i)\right) \in \mathbb{R}_+^n, \text{ for } i = 1, 2, 3$, when computing the distances $d(G_1, G_2)$, $d(G_1, G_3)$, $d(G_2, G_3)$; i.e. the total mass was $m_1 + m_2 + m_3$ for all pairs. We can similarly assume that $P, Q$ account for this extra mass, i.e. $P_{n_1 n_2}$ and $Q_{n_2 n_3}$ have been increased by $m_3$ and $m_1$ respectively from their solutions in (5).

To apply the gluing lemma, let $S = P \text{diag}(1/y_2)Q \in \mathbb{R}^{n_1 \times n_3}$, where $\text{diag}(1/y_2)$ is a diagonal matrix whose $(j, j)^{th}$ element is $1/(y_2)_j$ (note $y_2 > 0$). We see that $S$ is feasible for (5) when computing $d(G_1, G_3)$,

$$R_{n_3} = P \text{diag}(1/y_2)Q 1_{n_3} = P \text{diag}(1/y_2)y_2 = P 1_{n_2} = y_1.$$ 

Similarly, $R_{1} = y_1$. Now, let $U', V', W'$ be the cost matrices $C'$ in (5) when computing $d(G_1, G_3), d(G_2, G_3)$, and $d(G_1, G_3)$ respectively. We will use the following technical lemma whose proof is given below.

**Lemma 4.** For all $i \in L_1, j \in L_2, k \in L_3$, we have $W'_{ik} \leq U'_{ij} + V'_{jk}$.
Applying Lemma 4 yields the triangle inequality:
\[
d(G_1, G_3) \leq \langle R, W' \rangle = \sum_{i, j, k \in E} W'_{ik} \sum_{j \in E_3} P_{ij} Q_{jk} \leq \sum_{i, j, k} (U'_{ij} + V'_{jk}) P_{ij} Q_{jk}
\]
\[
= \sum_{ij} U'_{ij} P_{ij} + \sum_{jk} V'_{jk} Q_{jk} = d(G_1, G_2) + d(G_2, G_3)
\]

The first step uses the fact that \(d(G_1, G_3)\) is the minimum of all feasible solutions and the third step uses Lemma 4. The fourth step rearranges terms and the fifth step uses \(P^\top 1_{\bar{n}_1} = Q 1_{\bar{n}_3} = y_2\).

**Proof of Lemma 4.** Let \(W' = W'_{\text{lin}} + W'_{\text{str}} + W'_{\text{nas}}\) be the decomposition into the label mismatch, structural and non-assignment parts of the cost matrices; define similar quantities \(U'_{\text{lin}}, U'_{\text{str}}, U'_{\text{nas}}, V'_{\text{lin}}, V'_{\text{str}}, V'_{\text{nas}}\) for \(U', V'\). Noting \(a \leq b+c\) and \(d \leq e+f\) implies \(a+d \leq b+c+e+f\), it is sufficient to show the triangle inequality for each component individually. For the label mismatch term, \((W'_{\text{lin}})_{ik} \leq (U'_{\text{lin}})_{ij} + (V'_{\text{lin}})_{jk}\) follows directly from the conditions on \(M\) by setting \(x = \ell(i), y = \ell(j), z = \ell(k)\), where \(i, j, k\) are indexing in \(L_1, L_2, L_3\) respectively.

For the non-assignment terms, when \((W'_{\text{nas}})_{ik} = 0\) the claim is true trivially. \((W'_{\text{nas}})_{ik} = 1\), either when \((i = \bar{n}_1, k \leq n_3)\) or \((i \leq n_1, k = \bar{n}_3)\). In the former case, when \(j \leq n_2, (U'_{\text{nas}})_{ik} = 1\) and when \(j = \bar{n}_2, (V'_{\text{nas}})_{ik} = 1\) as \(k \leq n_3\). We therefore have, \((W'_{\text{nas}})_{ik} = (U'_{\text{nas}})_{ij} + (V'_{\text{nas}})_{jk} = 1\). A similar argument shows equality for the \((i \leq n_1, k = \bar{n}_3)\) case as well.

Finally, for the structural terms we note that \(W'_{\text{str}}\) can be written as \(W'_{\text{str}} = \sum_{t} W'\{t\}\) as can \(U'\{t\}, V'\{t\}\). Here \(t\) indexes over the choices for the types of distances considered, i.e. \(t \in \{\text{sp, lp, rw}\} \times \{\text{ip, op}\}\). It is sufficient to show \((W'\{t\})_{ik} \leq (U'\{t\})_{ij} + (V'\{t\})_{jk}\). This inequality takes the form
\[
|\delta_{1i}^{(t)} - \delta_{3k}^{(t)}| \leq |\delta_{1i}^{(t)} - \delta_{2j}^{(t)}| + |\delta_{2j}^{(t)} - \delta_{3k}^{(t)}|.
\]

Where \(\delta_{gt}^{(t)}\) refers to distance type \(t\) in network \(g\) for layer \(s\). The above is simply the triangle inequality for real numbers. This concludes the proof of Lemma 4.

**A.3 Implementation & Design Choices**

**Masses on the decision & input/output layers:** It is natural to ask why one needs to model the mass in the decision and input/output layers. For example, a seemingly natural choice is to use 0 for these layers. Using 0 mass, is a reasonable strategy if we were to allow only one decision layer. However, when there are multiple decision layers, consider comparing the following two networks: the first has a feed forward MLP with non-linear layers, the second is the same network but with an additional linear decision layer \(u\), with one edge from \(u_{ip}\) to \(u\) and an edge from \(u\) to \(u_{op}\). This latter models the function as a linear + non-linear term which might be suitable for some problems unlike modeling it only as a non-linear term. If we do not add layer masses for the input/output/decision layers, then the distance between both networks would be 0 - as there will be equal mass in the FF part for both networks and they can be matched with 0 cost.

**Algorithm 1:** Compute \(\delta_{\text{op}}(u)\) for all \(u \in L\)

**Require:** \(G = (L, E)\), \(L\) is topologically sorted in \(S\).
1: \(\delta_{\text{op}}(u_{\text{op}}) = 0, \delta_{\text{op}}(u) = \text{n\text{a\text{n}}} \forall u \neq u_{\text{op}}\).
2: \textbf{while} \(S\) is not empty \textbf{do}
3: \quad \(u \leftarrow \text{pop, last}(S)\)
4: \quad \(\Delta \leftarrow \{\delta_{\text{op}}(c) : c \in \text{children}(u)\}\)
5: \quad \(\delta_{\text{op}}(u) \leftarrow 1 + \text{average}(\Delta)\)
6: \textbf{end while}
7: \textbf{Return} \(\delta_{\text{op}}\).
Table 4: The label mismatch cost matrix $M$ we used in our CNN experiments. $M(x, y)$ denotes the penalty for transporting a unit mass from a layer with label $x$ to a layer with label $y$. The labels abbreviated are conv3, conv5, conv7, max-pool, avg-pool, fc, and softmax in order. A blank indicates $\infty$ cost. We have not shown the ip and op layers, but they are similar to the fc column, 0 in the diagonal and $\infty$ elsewhere.

Table 5: The label mismatch cost matrix $M$ we used in our MLP experiments. The labels abbreviated are relu, elu, leaky-relu, softplus, linear, and logistic in order. <rec> is place-holder for any other rectifier such as leaky-relu, softplus, relu. A blank indicates $\infty$ cost. A rectifier gets 0.1 cost with itself. A rectifier gets 0.1 cost with another rectifier and 0.25 with a sigmoid, vice versa for all sigmoids. The rest of the costs are infinity. We have not shown the ip and op, but they are similar to the lin column, 0 in the diagonal and $\infty$ elsewhere.

Computing path lengths $\delta^j_i$: Algorithm 1 computes all path lengths in $O(|E|)$ time. Note that topological sort of a connected digraph also takes $O(|E|)$ time. The topological sorting ensures that $\delta_{op}^{rw}$ is always computed for the children in step 4. For $\delta_{op}, \delta_{ip}$ we would replace the averaging of $\Delta$ in step 5 with the minimum and maximum of $\Delta$ respectively.

For $\delta_{ip}^{rw}$ we make the following changes to Algorithm 1. In step 1, we set $\delta_{ip}^{rw}(u_{ip}) = 0$, in step 3, we pop_first and $\Delta$ in step 4 is computed using the parents. $\delta_{ip}, \delta_{ip}^{op}$ are computed with the same procedure but by replacing the averaging with minimum or maximum as above.

Label Penalty Matrices: The label penalty matrices used in our NASBOT implementation, described below, satisfy the triangle inequality condition in Theorem 3.

CNNs: Table 4 shows the label penalty matrix $M$ for used in our CNN experiments with labels conv3, conv5, conv7, max-pool, avg-pool, sofmax, ip, op. conv$k$ denotes a $k \times k$ convolution while avg-pool and max-pool are pooling operations. In addition, we also use res3, res5, res7 layers which are inspired by ResNets. A res$k$ uses 2 concatenated conv$k$ layers but the input to the first layer is added to the output of the second layer before the relu activation – See Figure 2 in He et al. [12]. The layer mass for res$k$ layers is twice that of a conv$k$ layer. The costs for the res in the label penalty matrix is the same as the conv block. The cost between a res$k$ and conv$j$ is $M(\text{res$k$}, \text{conv}j) = 0.9 \times M(\text{conv$k$}, \text{conv}j) + 0.1 \times 1$; i.e. we are using a convex combination of the conv costs and the non-assignment cost. The intuition is that a res$k$ is similar to conv$k$ block except for the residual addition.

MLPs: Table 5 shows the label penalty matrix $M$ for used in our MLP experiments with labels relu, elu, leaky-relu, softplus, elu, logistic, tanh, linear, ip, op. Here the first seven are common non-linear activations; relu, elu, leaky-relu, softplus, elu rectifiers while logistic and tanh are sigmoidal activations.

Other details: Our implementation of OTMANN differs from what is described in the main text in two ways. First, in our CNN experiments, for a fc layer $u$, we use $0.1 \times \ell m(u) \times (# \text{incoming-channels})$ as the mass, i.e. we multiply it by 0.1 from what is described in the main text. This is because, in the convolutional and pooling channels, each unit is an image where as in the fc layers each unit is a scalar. One could, in principle, account for the image sizes at the various layers when computing the layer masses, but this also has the added complication of depending on the size of the input image which varies from problem to problem. Our approach is simpler and yields reasonable results.
Secondly, we use a slightly different form for $C_{\text{str}}$. First, for $i \in L_1$, $j \in L_2$, we let $C_{\text{all}}(i, j) = \frac{1}{2} \sum_{s \in \{\text{sp}, \text{lp, rw}\}} \sum_{t \in \{\text{ip,op}\}} |\delta_{t}(i) - \delta_{t}(j)|$ be the average of all path length differences; i.e. $C_{\text{all}}$ captures the path length differences when considering all layers. For CNNs, we similarly construct matrices $C_{\text{str}}^\text{conv}$, $C_{\text{str}}^\text{pool}$, $C_{\text{str}}^\text{fc}$, except they only consider the convolutional, pooling and fully connected layers respectively in the path lengths. For $C_{\text{str}}^\text{conv}$, the distances to the output (from the input) can be computed byzeroing outgoing (incoming) edges to layers that are not convolutional. We can similarly construct $C_{\text{str}}^\text{pool}$ and $C_{\text{str}}^\text{fc}$ only counting the pooling and fully connected layers. Our final cost matrix for the structural penalty is the average of these four matrices, $C_{\text{str}} = (C_{\text{str}}^\text{all} + C_{\text{str}}^\text{conv} + C_{\text{str}}^\text{pool} + C_{\text{str}}^\text{fc})/4$.

For MLPs, we adopt a similar strategy by computing matrices $C_{\text{str}}^\text{all}$, $C_{\text{str}}^\text{rec}$, $C_{\text{str}}^\text{all}$ with all layers, only rectifiers, and only sigmoidal layers and let $C_{\text{str}} = (C_{\text{str}}^\text{all} + C_{\text{str}}^\text{rec} + C_{\text{str}}^\text{sig})/3$. The intuition is that by considering certain types of layers, we are accounting for different types of information flow due to different operations.

### A.4 Some Illustrations of the OTMANN Distance

We illustrate that OTMANN computes reasonable distances on neural network architectures via a two-dimensional t-SNE visualisation [24] of the network architectures based. Given a distance matrix between $m$ objects, t-SNE embeds them in a $d$ dimensional space so that objects with small distances are placed closer to those that have larger distances. Figure 4 shows the t-SNE embedding using the OTMANN distance and its normalised version. We have indexed 13 networks in both figures in a-n and displayed their architectures in Figure 5. Similar networks are placed close to each other indicating that OTMANN induces a meaningful topology among neural network architectures.

Next, we show that the distances induced by OTMANN are correlated with validation error performance. In Figure 6 we provide the following scatter plot for networks trained in our experiments for the Indoor, Naval and Slice datasets. Each point in the figure is for pair of networks. The $x$-axis is the OTMANN distance between the pair and the $y$-axis is the difference in the validation error on the dataset. In each figure we used 300 networks giving rise to $45K$ pairwise points in each scatter plot. As the figure indicates, when the distance is small the difference in performance is close to 0. However, as the distance increases, the points are more scattered. Intuitively, one should expect that while networks that are far apart could perform similarly or differently, similar networks should perform similarly. Hence, OTMANN induces a useful topology in the space of architectures that is smooth for validation performance on real world datasets. This demonstrates that it can be incorporated in a BO framework to optimise a network based on its validation error.

### B Implementation of NASBOT

Here, we describe our BO framework for NASBOT in full detail.

#### B.1 The Kernel

As described in the main text, we use a negative exponentiated distance as our kernel. Precisely, we use,

$$k(\cdot, \cdot) = \alpha e^{-\sum_{i} \beta_i d^p_i(\cdot, \cdot)} + \tilde{\alpha} e^{-\sum_{i} \tilde{\beta}_i \tilde{d}^p_i(\cdot, \cdot)}.$$  

(7)

Here, $d_i, \tilde{d}_i$, are the OTMANN distance and its normalised counterpart developed in Section 3, computed with different values for $\nu_{\text{str}} \in \{\nu_{\text{str}, i}\}_i$, $\beta_i, \tilde{\beta}_i$ manage the relative contributions of $d_i, \tilde{d}_i$, while $(\alpha, \tilde{\alpha})$ manage the contributions of each kernel in the sum. An ensemble approach of the above form, instead of trying to pick a single best value, ensures that NASBOT accounts for the different topologies induced by the different distances $d_i, \tilde{d}_i$. In the experiments we report, we used $\{\nu_{\text{str}, i}\}_i = \{0.1, 0.2, 0.4, 0.8\}$, $p = 1$ and $\tilde{p} = 2$. Our experience suggests that NASBOT was not particularly sensitive to these choices expect when we used only very large or only very small values in $\{\nu_{\text{str}, i}\}_i$.

NASBOT, as described above has 11 hyper-parameters of its own: $\alpha, \tilde{\alpha}, \{(\beta_i, \tilde{\beta}_i)\}_i = 1$ and the GP noise variance $\eta^2$. While maximising the GP marginal likelihood is a common approach to pick hyper-parameters, this might cause over-fitting when there are many of them. Further, as training large neural networks is typically expensive, we have to content with few observations for the GP
Figure 4: Two dimensional t-SNE embeddings of 100 randomly generated CNN architectures based on the OTMANN distance (top) and its normalised version (bottom). Some networks have been indexed a-n in the figures; these network architectures are illustrated in Figure 5. Networks that are similar are embedded close to each other indicating that the OTMANN induces a meaningful topology among neural network architectures.
Figure 5: Illustrations of the networks indexed a-n in Figure 4.
Figure 6: Each point in the scatter plot indicates the log distance between two architectures (x axis) and the difference in the validation error (y axis), on the Indoor, Naval and Slice datasets. We used 300 networks, giving rise to $\sim 45K$ pairwise points. On all datasets, when the distance is small, so is the difference in the validation error. As the distance increases, there is more variance in the validation error difference. Intuitively, one should expect that while networks that are far apart could perform similarly or differently, networks with small distance should perform similarly.

B.2 Optimising the Acquisition

We use an evolutionary algorithm (EA) approach to optimise the acquisition function (2). We begin with an initial pool of networks and evaluate the acquisition $\varphi_t$ on those networks. Then we generate a set of $N_{\text{mut}}$ mutations of this pool as follows. First, we stochastically select $N_{\text{mut}}$ candidates from the set of networks already evaluated such that those with higher $\varphi_t$ values are more likely to be selected than those with lower values. Then we apply a mutation operator to each candidate, to produce a modified architecture. Finally, we evaluate the acquisition on this $N_{\text{mut}}$ mutations, add it to the initial pool, and repeat for the prescribed number of steps.

**Mutation Operator:** To describe the mutation operator, we will first define a library of modifications to a neural network. These modifications, described in Table 6, might change the architecture either by increasing or decreasing the number of computational units in a layer, by adding or deleting layers, or by changing the connectivity of existing layers. They provide a simple mechanism to explore the space of architectures that are close to a given architecture. The *one-step mutation operator* takes a given network and applies one of the modifications in Table 6 picked at random to produce a new network. The *k-step mutation operator* takes a given network, and repeatedly applies the one-step operator $k$ times — the new network will have undergone $k$ changes from the original one. One can also define a compound operator, which picks the number of steps probabilistically. In our implementation of NASBOT, we used such a compound operator with probabilities $(0.5, 0.25, 0.125, 0.075, 0.05)$; i.e. it chooses a one-step operator with probability 0.5, a 4-step operator with probability 0.075, etc.

**Sampling strategy:** The sampling strategy for EA is as follows. Let $\{z_i\}_i$, where $z_i \in \mathcal{X}$ be the points evaluated so far. We sample $N_{\text{mut}}$ new points from a distribution $\pi$ where $\pi(z_i) \propto \exp(g(z_i)/\sigma)$. Here $g$ is the function to be optimised (for NASBOT, $\varphi_t$ at time $t$). $\sigma$ is the standard deviation of all previous evaluations. As the probability for large $g$ values is higher, they are more likely to get selected. $\sigma$ provides normalisation to account for different ranges of function values.
<table>
<thead>
<tr>
<th><strong>Operation</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>dec_single</td>
<td>Pick a layer at random and decrease the number of units by 1/8.</td>
</tr>
<tr>
<td>dec_en_masse</td>
<td>First topologically order the networks, randomly pick 1/8 of the layers (in order) and decrease the number of units by 1/8. For networks with eight layers or fewer pick a 1/4 of the layers (instead of 1/8) and for those with four layers or fewer pick 1/2.</td>
</tr>
<tr>
<td>inc_single</td>
<td>Pick a layer at random and increase the number of units by 1/8.</td>
</tr>
<tr>
<td>inc_en_masse</td>
<td>Choose a large sub set of layers, as for dec_en_masse, and increase the number of units by 1/8.</td>
</tr>
<tr>
<td>dup_path</td>
<td>This modifier duplicates a random path in the network. Randomly pick a node $u_1$ and then pick one of its children $u_2$ randomly. Keep repeating to generate a path $u_1, u_2, \ldots, u_{k-1}, u_k$ until you decide to stop randomly. Create duplicate layers $\tilde{u}<em>2, \ldots, \tilde{u}</em>{k-1}$ where $\tilde{u}_i = u_i$ for $i = 2, \ldots, k - 1$. Add these layers along with new edges $(u_1, \tilde{u}<em>2), (\tilde{u}</em>{k-1}, u_k)$, and $(\tilde{u}<em>j, \tilde{u}</em>{j+1})$ for $j = 2, \ldots, k - 2$.</td>
</tr>
<tr>
<td>remove_layer</td>
<td>Picks a layer at random and removes it. If this layer was the only child (parent) of any of its parents (children) $u$, then adds an edge from $u$ (one of its parents) to one of its children $u$.</td>
</tr>
<tr>
<td>skip</td>
<td>Randomly picks layers $u, v$ where $u$ is topologically before $v$ and $(u, v) \notin \mathcal{E}$. Add $(u, v)$ to $\mathcal{E}$.</td>
</tr>
<tr>
<td>swap_label</td>
<td>Randomly pick a layer and change its label.</td>
</tr>
<tr>
<td>wedge_layer</td>
<td>Randomly pick any edge $(u, v) \in \mathcal{E}$. Create a new layer $w$ with a random label $\ell(w)$. Remove $(u, v)$ from $\mathcal{E}$ and add $(u, w), (w, v)$. If applicable, set the number of units $\ell(u(w))$ to be $(\ell(u) + \ell(v))/2$.</td>
</tr>
</tbody>
</table>

Table 6: Descriptions of modifiers to transform one network to another. The first four change the number of units in the layers but do not change the architecture, while the last five change the architecture.

Since our candidate selection scheme at each step favours networks that have high acquisition value, our EA scheme is more likely to search at regions that are known to have high acquisition. The stochasticity in this selection scheme and the fact that we could take multiple steps in the mutation operation ensures that we still sufficiently explore the space. Since an evaluation of $\varphi_t$ is cheap, we can use many EA steps to explore several architectures and optimise $\varphi_t$.

**Other details:** The EA procedure is also initialised with the same initial pools in Figures 20, 21. In our NASBOT implementation, we increase the total number of EA evaluations $n_{\text{EA}}$ at rate $O(\sqrt{t})$ where $t$ is the current time step in NASBOT. We set $N_{\text{mut}}$ to be $O(\sqrt{n_{\text{EA}}})$. Hence, initially we are only considering a small neighborhood around the initial pool, but as we proceed along BO, we expand to a larger region, and also spend more effort to optimise $\varphi_t$.

**Considerations when performing modifications:** The modifications in Table 6 is straightforward in MLPs. But in CNNs one needs to ensure that the image sizes are the same when concatenating them as an input to a layer. This is because strides can shrink the size of the image. When we perform a modification we check if this condition is violated and if so, disallow that modification. When a skip modifier attempts to add a connection from a layer with a large image size to one with a smaller one, we add avg-pool1 layers at stride 2 so that the connection can be made (this can be seen, e.g. in the second network in Fig. 8).

**B.3 Other Implementation Details**

**Initialisation:** We initialise NASBOT (and other methods) with an initial pool of 10 networks. These networks are illustrated in Fig. 20 for CNNs and Fig. 21 for MLPs at the end of the document. These are the same networks used to initialise the EA procedure to optimise the acquisition. All initial networks have feed forward structure. For the CNNs, the first 3 networks have structure similar to the VGG nets [34] and the remaining have blocked feed forward structures as in He et al. [12]. We also use blocked structures for the MLPs with the layer labels decided arbitrarily.

**Domain:** For NASBOT, and other methods, we impose the following constraints on the search space. If the EA modifier (explained below) generates a network that violates these constraints, we simply skip it.

- Maximum number of layers: 60
• Maximum mass: \(10^8\)
• Maximum in/out degree: 5
• Maximum number of edges: 200
• Maximum number of units per layer: 1024
• Minimum number of units per layer: 8

Layer types: We use the layer types detailed in Appendix A.3 for both CNNs and MLPs. For CNNs, all pooling operations are done at stride 2. For convolutional layers, we use either stride 1 or 2 (specified in the illustrations). For all layers in a CNN we use \texttt{relu} activations.

Parallel BO: We use a parallelised experimental set up where multiple models can be evaluated in parallel. We handle parallel BO via the hallucination technique in Ginsbourger et al. [10].

Finally, we emphasise that many of the above choices were made arbitrarily, and we were able to get NASBOT working efficiently with our first choice for these parameters/specifications. Note that many end-to-end systems require specification of such choices.

C Addendum to Experiments

C.1 Baselines

RAND: Our RAND implementation, operates in exactly the same way as NASBOT, except that the EA procedure (Sec. B.2) is fed a random sample from \texttt{Unif}(0, 1) instead of the GP acquisition each time it evaluates an architecture. That is, we follow the same schedule for \(n_{\text{EA}}\) and \(N_{\text{mut}}\) as we did for NASBOT. Hence RAND has the opportunity to explore the same space as NASBOT, but picks the next evaluation randomly from this space.

EA: This is as described in Appendix B except that we fix \(N_{\text{mut}} = 10\) all the time. In our experiments where we used a budget based on time, it was difficult to predict the total number of evaluations so as to set \(N_{\text{mut}}\) in perhaps a more intelligent way.

TreeBO: As the implementation from Jenatton et al. [15] was not made available, we wrote our own. It differs from the version described in the paper in a few ways. We do not tune for a regularisation penalty and step size as they do to keep it line with the rest of our experimental set up. We set the depth of the network to 60 as we allowed 60 layers for the other methods. We also check for the other constraints given in Appendix B before evaluating a network. The original paper uses a tree structured kernel which can allow for efficient inference with a large number of samples. For simplicity, we construct the entire kernel matrix and perform standard GP inference. The result of the inference is the same, and the number of GP samples was always below 120 in our experiments so a sophisticated procedure was not necessary.

C.2 Details on Training

In all methods, for each proposed network architecture, we trained the network on the train data set, and periodically evaluated its performance on the validation data set. For MLP experiments, we optimised network parameters using stochastic gradient descent with a fixed step size of \(10^{-5}\) and a batch size of 256 for 20,000 iterations. We computed the validation set MSE every 100 iterations; from this we returned the minimum MSE that was achieved. For CNN experiments, we optimised network parameters using stochastic gradient descent with a batch size of 32. We started with a learning rate of 0.01 and reduced it gradually. We also used batch normalisation and trained the model for 60,000 batch iterations. We computed the validation set classification error every 4000 iterations; from this we returned the minimum classification error that was achieved.

After each method returned an optimal neural network architecture, we again trained each optimal network architecture on the train data set, periodically evaluated its performance on the validation data set, and finally computed the MSE or classification error on the test data set. For MLP experiments, we used the same optimisation procedure as above; we then computed the test set MSE at the iteration where the network achieved the minimum validation set MSE. For CNN experiments, we used the same optimisation procedure as above, except here the optimal network architecture was trained...
for 120,000 iterations; we then computed the test set classification error at the iteration where the network achieved the minimum validation set classification error.

C.3 Optimal Network Architectures and Initial Pool

Here we illustrate and compare the optimal neural network architectures found by different methods. In Figures 8-11, we show some optimal network architectures found on the Cifar10 data by NASBOT, EA, RAND, and TreeBO, respectively. We also show some optimal network architectures found for these four methods on the Indoor data, in Figures 12-15, and on the Slice data, in Figures 16-19. A common feature among all optimal architectures found by NASBOT was the presence of long skip connections and multiple decision layers.

In Figure 21, we show the initial pool of MLP network architectures, and in Figure 20, we show the initial pool of CNN network architectures. On the Cifar10 dataset, VGG-19 was one of the networks in the initial pool. While all methods beat VGG-19 when trained for 24K iterations (the number of iterations we used when picking the model), TreeBO and RAND lose to VGG-19 (see Section 5 for details). This could be because the performance after shorter training periods may not exactly correlate with performance after longer training periods.

C.4 Ablation Studies and Design Choices

We conduct experiments comparing the various design choices in NASBOT. Due to computational constraints, we carry them out on synthetic functions.

In Figure 7a, we compare NASBOT using only the normalised distance, only the unnormalised distance, and the combined kernel as in (7). While the individual distances performs well, the combined form outperforms both.

Next, we modify our EA procedure to optimise the acquisition. We execute NASBOT using only the EA modifiers which change the computational units (first four modifiers in Table 6), then using the modifiers which only change the structure of the networks (bottom 5 in Table 6), and finally using all 9 modifiers, as used in all our experiments. The combined version outperforms the first two.

Finally, we experiment with different choices for $p$ and $\bar{p}$ in (7). As the figures indicate, the performance was not particularly sensitive to these choices.

Below we describe the three synthetic functions $f_1$, $f_2$, $f_3$ used in our synthetic experiments. $f_3$ applies for CNNs while $f_1$, $f_2$ apply for MLPs. Here $am$ denotes the average mass per layer, $deg_i$ is the average in degree the layers, $deg_o$ is the average out degree, $\delta$ is the shortest distance from $u_{ip}$ to $u_{op}$, $str$ is the average stride in CNNs, frac_conv3 is the fraction of layers that are conv3, frac_sigmoid is the fraction of layers that are sigmoidal.

\[
\begin{align*}
f_0 &= \exp(-0.001 \cdot |am - 1000|) + \exp(-0.5 \cdot |deg_i - 5|) + \exp(-0.5 \cdot |deg_o - 5|) + \exp(-0.1 \cdot |\delta - 5|) + \exp(-0.1 \cdot |\ell - 30|) + \exp(-0.05 \cdot |\ell| - 100) \\
f_1 &= f_0 + \exp(-3 \cdot |str - 1.5|) + \exp(-0.3 \cdot |\ell| - 50) + \frac{am - 500}{frac_{conv3}} \\
f_2 &= f_0 + \exp(-0.001 \cdot |am - 2000|) + \exp(-0.1 \cdot |\ell| - 50) + frac_{sigmoid} \\
f_3 &= f_0 + frac_{sigmoid}
\end{align*}
\]

D Additional Discussion on Related Work

Historically, evolutionary (genetic) algorithms (EA) have been the most common method used for designing architectures [8, 18, 23, 27, 33, 38, 48]. EA techniques are popular as they provide a simple mechanism to explore the space of architectures by making a sequence of changes to networks that have already been evaluated. However, as we will discuss later, EA algorithms, while conceptually and computationally simple, are typically not best suited for optimising functions that are expensive to evaluate. A related line of work first sets up a search space for architectures via incremental modifications, and then explores this space via random exploration, MCTS, or A* search [6, 22, 29].
Number of evaluations
0 50 100 150 200
Negative maximum value
-6
-5.5
-5
-4.5
-4
Ablation study on Kernel Design ($f_1$)
Only Normalised
Only Unnormalised
Combined

Number of evaluations
0 50 100 150 200
Negative maximum value
-4.5
-4
-3.5
-3
-2.5
Ablation study on EA modifiers ($f_2$)
Only Computational
Only Structural
Combined

Number of evaluations
0 50 100 150 200
Negative maximum value
-3.5
-3
-2.5
-2
Comparison of $p$, $\bar{p}$ values ($f_3$)
$p = 1$, $\bar{p} = 1$
$p = 1$, $\bar{p} = 2$
$p = 2$, $\bar{p} = 1$
$p = 2$, $\bar{p} = 2$

Figure 7: We compare NASBOT for different design choices in our framework. (a): Comparison of NASBOT using only the normalised distance $e^{-\beta d}$, only the unnormalised distance $d^{-\beta d}$, and the combination $e^{-\beta d} + e^{-\bar{\beta}d}$. (b): Comparison of NASBOT using only the EA modifiers which change the computational units (top 4 in Table 6), modifiers which only change the structure of the networks (bottom 5 in Table 6), and all 9 modifiers. (c): Comparison of NASBOT with different choices for $p$ and $\bar{p}$. In all figures, the $x$ axis is the number of evaluations and the $y$ axis is the negative maximum value (lower is better). All figures were produced by averaging over at least 10 runs.

Some of the methods above can only optimise among feed forward structures, e.g. Fig. 1a, but cannot handle spaces with arbitrarily structured networks, e.g. Figs. 1b, 1c.

The most successful recent architecture search methods that can handle arbitrary structures have adopted reinforcement learning (RL) [1, 49–51]. However, architecture search is in essence an optimisation problem – find the network with the highest function value. There is no explicit need to maintain a notion of state and solve the credit assignment problem in RL [40]. Since RL is fundamentally more difficult than optimisation [16], these methods typically need to try a very large number of architectures to find the optimum. This is not desirable, especially in computationally constrained settings.
Figure 8: Optimal network architectures found with NASBOT on Cifar10 data.
Figure 9: Optimal network architectures found with EA on Cifar10 data.
Figure 10: Optimal network architectures found with RAND on Cifar10 data.
Figure 11: Optimal network architectures found with TreeBO on Cifar10 data.
Figure 12: Optimal network architectures found with NASBOT on Indoor data.

Figure 13: Optimal network architectures found with EA on Indoor data.
Figure 14: Optimal network architectures found with RAND on Indoor data.

Figure 15: Optimal network architectures found with TreeBO on Indoor data.
Figure 16: Optimal network architectures found with NASBOT on Slice data.

Figure 17: Optimal network architectures found with EA on Slice data.
Figure 18: Optimal network architectures found with RAND on Slice data.

Figure 19: Optimal network architectures found with TreeBO on Slice data.
nets [34] and the remaining have blocked feed forward structures as in He et al. [12].

Figure 20: Initial pool of CNN network architectures. The first 3 networks have structure similar to the VGG nets [34] and the remaining have blocked feed forward structures as in He et al. [12].
Figure 21: Initial pool of MLP network architectures.