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 λ 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 (λ, 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.

Preprint. Work in progress.

While there are several approaches to BO, those based on Gaussian processes (GP) [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 φ_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. $x_t = \mathrm{argmax}_{x \in \mathcal{X}} \varphi_t(x)$. 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 unevaluated value $f(x')$ when we have already evaluated $f(x)$. Secondly, we need a method to maximise φ_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 φ_t via off the shelf branch-and-bound or gradient based methods. However, when each $x \in \mathcal{X}$ 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 [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 [40]. Since RL is a fundamentally more difficult problem than optimisation [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.

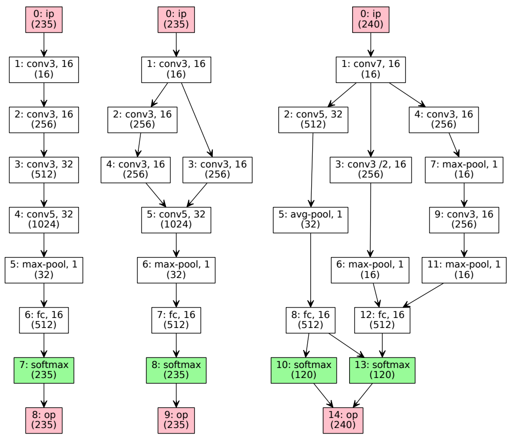
None of the above methods have been designed with a focus on the expense of evaluating a neural network, with an emphasis on being judicious in selecting which architecture to try next. Bayesian optimisation (BO) techniques, which use introspective Bayesian models to carefully determine future evaluations, 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 [2, 15, 25, 37, 41], they can only optimise among feed forward structures, 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 $\mathcal{X}$ of neural network architectures. When we evaluate f at $x \in \mathcal{X}$, 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 $x_\star = \mathrm{argmax}_{\mathcal{X}} f(x)$ is the optimal architecture, and x_t is the architecture evaluated at time t, we want $f(x_\star) - \max_{t \leq n} f(x_t)$ 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 $\mathcal{X}$, and is characterised by a mean function $\mu : \mathcal{X} \to \mathbb{R}$ and a (covariance) kernel $\kappa : \mathcal{X}^2 \to \mathbb{R}$. Given n observations $\mathcal{D}_n = \{(x_i, y_i)\}_{i=1}^n$, where



(a) (b) (c)

Figure 1: An illustration of some CNN architectures. In each layer, i: indexes the layer, followed by the label (e.g `conv3`), and then the number of units (e.g. number of filters). The input and output layers are pink while the decision (`softmax`) layers are green.

From Section 3: The layer mass is denoted in parentheses. The following are the normalised and unnormalised distances $d, \bar{d}$. All self distances are 0, i.e. $d(\mathcal{G},\mathcal{G}) = \bar{d}(\mathcal{G},\mathcal{G}) = 0$. Unnormalised: $d(\mathrm{a},\mathrm{b}) = 175.1$, $d(\mathrm{a},\mathrm{c}) = 1479.3$, $d(\mathrm{b},\mathrm{c}) = 1621.4$. Normalised: $\bar{d}(\mathrm{a},\mathrm{b}) = 0.0286$, $\bar{d}(\mathrm{a},\mathrm{c}) = 0.2395$, $\bar{d}(\mathrm{b},\mathrm{c}) = 0.2625$.

$x_i \in \mathcal{X}, y_i = f(x_i) + \epsilon_i \in \mathbb{R}$, and $\epsilon_i \sim \mathcal{N}(0,\eta^2)$, the posterior process $f|\mathcal{D}_n$ is also a GP with mean μ_n and covariance κ_n. Denote $Y \in \mathbb{R}^n$ with $Y_i = y_i$, $k, k' \in \mathbb{R}^n$ with $k_i = \kappa(x, x_i), k'_i = \kappa(x', x_i)$, and $K \in \mathbb{R}^{n\times n}$ with $K_{i,j} = \kappa(x_i, x_j)$. Then, μ_n, κ_n can be computed via,

$$\mu_n(x) = k^\top (K + \eta^2 I)^{-1} Y, \qquad \kappa_n(x, x') = \kappa(x, x') - k^\top (K + \eta^2 I)^{-1} k'. \tag{1}$$

For more background on GPs, we refer readers to Rasmussen and Williams [32]. When tasked with optimising a function f over a domain $\mathcal{X}$, BO models f as a sample from a GP. At time t, we have already evaluated f at points $\{x_i\}_{i=1}^{t-1}$ and obtained observations $\{y_i\}_{i=1}^{t-1}$. To determine the next point for evaluation x_t, we first use the posterior GP to define an *acquisition function* $\varphi_t : \mathcal{X} \to \mathbb{R}$, which measures the utility of evaluating f at any $x \in \mathcal{X}$ according to the posterior. We then maximise the acquisition $x_t = \mathrm{argmax}_{\mathcal{X}}\, \varphi_t(x)$, and evaluate f at x_t. The expected improvement acquisition [28],

$$\varphi_t(x) = \mathbb{E}\big[\max\{0, f(x) - \tau_{t-1}\}\big|\{(x_i, y_i)\}_{i=1}^{t-1}\big], \tag{2}$$

measures the expected improvement over the current maximum value according to the posterior GP. Here $\tau_{t-1} = \mathrm{argmax}_{i\leq t-1} f(x_i)$ denotes the current best value. This expectation can be computed in closed form for GPs. We use EI in this work, but the ideas apply just as well to other acquisitions [3].

GP/BO in the context of architecture search: Intuitively, $\kappa(x, x')$ is a measure of similarity between x and x'. If $\kappa(x, x')$ is large, then $f(x)$ and $f(x')$ are highly correlated. Hence, the GP effectively imposes a smoothness condition on $f : \mathcal{X} \to \mathbb{R}$; i.e. since networks a and b in Fig. 1 are similar, they are likely to have similar cross validation performance. In BO, when selecting the next point, we balance between *exploitation*, choosing points that we believe will have high f value, and *exploration*, choosing points that we do not know much about so that we do not get stuck at a bad optimum. For example, if we have already evaluated $f(\mathrm{a})$, then exploration incentivises us to choose c over b since we can reasonably gauge $f(\mathrm{b})$ from $f(\mathrm{a})$. On the other hand, if $f(\mathrm{a})$ has high value, then exploitation incentivises choosing b, as it is more likely to be the optimum than c.

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 $\mathcal{G} = (\mathcal{L}, \mathcal{E})$ 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 $\ell\ell(u)$ which denotes the type of operations performed at the layer. For instance, in Fig. 1a, $\ell\ell(1) =$ `conv3`, $\ell\ell(5) =$ `max-pool` denote a 3×3 convolution and a max-pooling operation. The attribute ℓu 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 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 $\ell\ell(u_{\text{ip}}) = \texttt{ip}$ and $\ell\ell(u_{\text{op}}) = \texttt{op}$. 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 $d : \mathcal{X}^2 \to \mathbb{R}_+$ 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 $\ell m : \mathcal{L} \to \mathbb{R}_+$ 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 $\ell u(u)$ 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 $\ell m(u_{\text{ip}}) = \ell m(u_{\text{op}}) = \zeta \sum_{u \in \mathcal{PL}} \ell m(u)$ where $\mathcal{PL}$ 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 $\mathcal{DL}$ are decision layers, $\forall\, u \in \mathcal{DL}, \ell m(u) = \frac{\zeta}{|\mathcal{DL}|} \sum_{u \in \mathcal{PL}} \ell m(u)$. In all our experiments, we use $\zeta = 0.1$. In Fig. 1, the layer masses for each layer are shown in parantheses.

Path lengths from/to u_{ip}/u_{op}: In a neural network $\mathcal{G}$, a path from u to v is a sequence of layers $u_1, \dots, u_s$ where $u_1 = u$, $u_s = v$ and $(u_i, u_{i+1}) \in \mathcal{E}$ 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 $u \in \mathcal{L}$, let $\delta^{\text{sp}}_{\text{op}}(u), \delta^{\text{lp}}_{\text{op}}(u), \delta^{\text{rw}}_{\text{op}}(u)$ denote the length of the shortest, longest and random walk paths from u to the output u_{op}. Similarly, let $\delta^{\text{sp}}_{\text{ip}}(u), \delta^{\text{lp}}_{\text{ip}}(u), \delta^{\text{rw}}_{\text{ip}}(u)$ denote the corresponding lengths

	conv3	conv5	max-pool	avg-pool	fc
conv3	0	0.2	∞	∞	∞
conv5	0.2	0	∞	∞	∞
max-pool	∞	∞	0	0.25	∞
avg-pool	∞	∞	0.25	0	∞
fc	∞	∞	∞	∞	0

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.

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 $t \in \{\text{ip,op}\}$, $\delta_t^s(u)$ can be computed for all $u \in \mathcal{L}$, in $\mathcal{O}(|\mathcal{E}|)$ time (see Appendix A.3 for details).

We are now ready to describe OTMANN. Given two networks $\mathcal{G}_1 = (\mathcal{L}_1, \mathcal{E}_1), \mathcal{G}_2 = (\mathcal{L}_2, \mathcal{E}_2)$ 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 $i \in \mathcal{G}_1$ and $j \in \mathcal{G}_2$. The OTMANN distance is computed by solving the following optimisation problem.

$$\begin{aligned} &\underset{Z}{\text{minimise}} \quad \phi_{\text{lmm}}(Z) + \phi_{\text{nas}}(Z) + \nu_{\text{str}}\phi_{\text{str}}(Z) \\ &\text{subject to} \sum_{j \in \mathcal{L}_2} Z_{ij} \le \ell m(i), \ \sum_{i \in \mathcal{L}_1} Z_{ij} \le \ell m(j), \ \forall i, j \end{aligned} \tag{3}$$

The label mismatch term ϕ_{lmm}, penalises matching masses that have different labels, while the structural term ϕ_{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 ϕ_{nas}. $\nu_{\text{str}} > 0$ 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 ϕ_{lmm}, ϕ_{nas}, and ϕ_{str}.

Label mismatch penalty ϕ_{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 $M(\mathtt{x}, \mathtt{y})$ denotes the penalty for transporting a unit mass from a layer with label x to a layer with label y. We then construct a matrix $C_{\text{lmm}} \in \mathbb{R}^{n_1 \times n_2}$ with $C_{\text{lmm}}(i,j) = M(\ell\ell(i), \ell\ell(j))$ corresponding to the mislabel cost for matching unit mass from each layer $i \in \mathcal{L}_1$ to each layer $j \in \mathcal{L}_2$. We then set $\phi_{\text{lmm}}(Z) = \langle Z, C_{\text{lmm}} \rangle = \sum_{i \in \mathcal{L}_1, j \in \mathcal{L}_2} Z(i,j)C(i,j)$ to be the sum of all matchings from $\mathcal{L}_1$ to $\mathcal{L}_2$ 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 ∞, while for similar layers, we choose a value less than 1.

Non-assignment penalty ϕ_{nas}: We set this to be the amount of mass that is unassigned in both networks, i.e. $\phi_{\text{nas}}(Z) = \sum_{i \in \mathcal{L}_1} \left(\ell m(i) - \sum_{j \in \mathcal{L}_2} Z_{ij}\right) + \sum_{j \in \mathcal{L}_2} \left(\ell m(j) - \sum_{i \in \mathcal{L}_1} Z_{ij}\right)$. 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, $M(\mathtt{x}, \mathtt{y}) \ll 1$ and for disparate layers $M(\mathtt{x}, \mathtt{y}) \gg 1$. 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 ν_{str} for ϕ_{lmm} and ϕ_{nas} – it is simple to specify reasonable values for $M(\mathtt{x}, \mathtt{y})$ from an understanding of their functionality.

Structural penalty ϕ_{str}: We define a matrix $C_{\text{str}} \in \mathbb{R}^{n_1 \times n_2}$ where $C_{\text{str}}(i,j)$ is small if layers $i \in \mathcal{L}_1$ and $j \in \mathcal{L}_2$ are at structurally similar positions in their respective networks. We then set $\phi_{\text{str}}(Z) = \langle Z, C_{\text{str}} \rangle$. For $i \in \mathcal{L}_1, \ j \in \mathcal{L}_2$, we let $C_{\text{str}}(i,j) = \frac{1}{6}\sum_{s \in \{\text{sp, lp, rw}\}} \sum_{t \in \{\text{ip,op}\}} |\delta_t^s(i) - \delta_t^s(j)|$ be the average of all path length differences, where δ_t^s are the path lengths defined previously. We define ϕ_{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 $\mathcal{O}(|\mathcal{E}|)$ 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 [44]. OT is a well studied problem with several efficient solvers [30]. Our theorem below, shows that the solution of (3) is a distance.

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

Operation	Description
dec_single	Pick a layer at random and decrease the number of units by $1/8$.
dec_en_masse	Pick several layers at random and decrease the number of units by $1/8$ for all of them.
inc_single	Pick a layer at random and increase the number of units by $1/8$.
inc_en_masse	Pick several layers at random and increase the number of units by $1/8$ for all of them.
dup_path	Pick a random path $u_1, \ldots, u_k$, duplicate $u_2, \ldots, u_{k-1}$ and connect them to u_1 and u_k.
remove_layer	Pick a layer at random and remove it. Connect the layer's parents to its children if necessary.
skip	Randomly pick layers u, v where u is topologically before v. Add (u, v) to $\mathcal{E}$.
swap_label	Randomly pick a layer and change its label.
wedge_layer	Randomly remove an edge (u, v) from $\mathcal{E}$. Create a new layer w and add $(u, w), (w, v)$ to $\mathcal{E}$.

Table 2: 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.

Theorem 1. *Let $d(\mathcal{G}_1, \mathcal{G}_2)$ be the solution of (3) for networks $\mathcal{G}_1, \mathcal{G}_2$. Under mild regularity conditions on M, $d(\cdot, \cdot)$ is a pseudo-distance. That is, for all networks $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3$, it satisfies, $d(\mathcal{G}_1, \mathcal{G}_2) \geq 0$, $d(\mathcal{G}_1, \mathcal{G}_2) = d(\mathcal{G}_2, \mathcal{G}_1)$, $d(\mathcal{G}_1, \mathcal{G}_1) = 0$ and $d(\mathcal{G}_1, \mathcal{G}_3) \leq d(\mathcal{G}_1, \mathcal{G}_2) + d(\mathcal{G}_2, \mathcal{G}_3)$.*

For what follows, define $\bar{d}(\mathcal{G}_1, \mathcal{G}_2) = d(\mathcal{G}_1, \mathcal{G}_2)/(tm(\mathcal{G}_1) + tm(\mathcal{G}_2))$ where $tm(\mathcal{G}_i) = \sum_{u \in \mathcal{L}_i} \ell m(u)$ is the total mass of a network. Note that $\bar{d} \leq 1$. While $\bar{d}$ does not satisfy the triangle inequality, it provides a useful measure of dissimilarity normalised by the amount of computation. Our experience suggests that d puts more emphasis on the amount of computation at the layers over structure and vice versa for $\bar{d}$. Therefore, it is prudent to combine both quantities in any downstream application. The caption in Fig. 1 gives $d, \bar{d}$ values for the examples in that figure when $\nu_{\text{str}} = 0.5$.

We conclude this section with a couple of remarks. First, OTMANN shares similarities with Wasserstein (earth mover's) distances which also have an OT formulation. However, it is not a Wasserstein distance itself—in particular, the supports of the masses and the cost matrices change depending on the two networks being compared. Second, while there has been prior work for defining various distances and kernels on graphs, we cannot use them in BO because neural networks have additional complex properties in addition to graphical structure, such as the type of operations performed at each layer, the number of neurons, etc. The above work either define the distance/kernel between vertices or assume the same vertex (layer) set [9, 20, 26, 35, 46], none of which apply in our setting. While some methods do allow different vertex sets [45], they cannot handle layer masses and layer similarities. Moreover, the computation of the above distances are more expensive than OTMANN. Hence, these methods cannot be directly plugged into BO framework for architecture search.

In Appendix A, we provide additional material on OTMANN. This includes the proof of Theorem 1, a discussion on some design choices, and implementation details such as the computation of the path lengths. Moreover, we provide illustrations to demonstrate that OTMANN is a meaningful distance for architecture search. For example, a t-SNE embedding places similar architectures close to each other. Further, scatter plots showing the validation error vs distance on real datasets demonstrate that networks with small distance tend to perform similarly on the problem.

4 NASBOT

We now describe NASBOT, our BO algorithm for neural architecture search. Recall that in order to realise the BO scheme outlined in Section 2.1, we need to specify (a) a kernel κ for neural architectures and (b) a method to optimise the acquisition φ_t over these architectures. Due to space constraints, we will only describe the key ideas and defer all details to Appendix B.

As described previously, we will use a negative exponentiated distance for κ. Precisely, $\kappa = \alpha e^{-\beta d} + \bar{\alpha} d^{-\bar{\beta}\bar{d}}$, where $d, \bar{d}$ are the OTMANN distance and its normalised version. We mention that while this has the form of popular kernels, we do not know yet if it is in fact a kernel. In our several experiments, we did not encounter an instance where the eigenvalues of the kernel matrix were negative. If any case, there are several methods to circumvent this issue in kernel methods [39].

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 φ_t on those networks. Then we generate a set of N_{mut} mutations of this pool as follows. First, we stochastically select N_{mut} candidates from the set of networks already evaluated such that those with higher φ_t 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

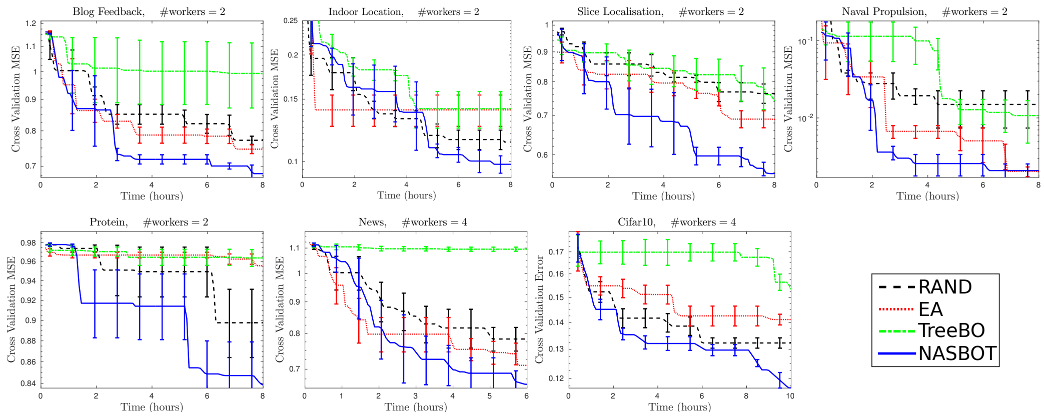


Figure 2: *Cross validation results:* In all figures, the x axis is time. The y axis is the mean squared error (MSE) in the first 6 figures and the classification error in the last. Lower is better in all cases. The title of each figure states the dataset and the number of parallel workers (GPUs). All figures were averaged over at least 5 independent runs of each method. Error bars indicate one standard error.

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_{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 φ_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.

Method	Blog $(60K, 281)$	Indoor $(21K, 529)$	Slice $(54K, 385)$	Naval $(12K, 17)$	Protein $(46K, 9)$	News $(40K, 61)$	Cifar10 $(60K, 1K)$	Cifar10 $150K$ iters
RAND	0.780 ± 0.034	**0.115 ±0.023**	0.758 ± 0.041	0.0103 ± 0.002	0.948 ± 0.024	**0.762 ±0.013**	0.1342 ± 0.002	0.0914 ± 0.008
EA	0.806 ± 0.040	0.147 ± 0.010	0.733 ± 0.041	**0.0079 ±0.004**	1.010 ± 0.038	**0.758 ±0.038**	0.1411 ± 0.002	0.0915 ± 0.010
TreeBO	0.928 ± 0.053	0.168 ± 0.023	0.759 ± 0.079	0.0102 ± 0.002	0.998 ± 0.007	0.866 ± 0.085	0.1533 ± 0.004	0.1121 ± 0.004
NASBOT	**0.731 ±0.029**	**0.117 ±0.008**	**0.615 ±0.044**	**0.0075 ±0.002**	**0.902 ±0.033**	**0.752 ±0.024**	**0.1209 ±0.003**	**0.0869 ±0.004**

Table 3: The first row gives the number of samples N and the dimensionality D of each dataset in the form (N, D). The subsequent rows show the regression MSE or classification error (lower is better) on the *test set* for each method. The last column is for Cifar10 where we took the best models found by each method in 24K iterations and trained it for $120K$ iterations. When we trained the VGG-19 architecture using our training procedure, we got test errors 0.1718 (60K iterations) and 0.1018 (150K iterations).

Experimental Set up: Each method is executed in an asynchronously parallel set up of 2-4 GPUs, That is, it can evaluate multiple models in parallel, with each model on a single GPU. When the evaluation of one model finishes, the methods can incorporate the result and immediately re-deploy the next job without waiting for the others to finish. For the blog, indoor, slice, naval and protein datasets we use 2 GeForce GTX 970 (4GB) GPUs and a computational budget of 8 hours for each method. For the news popularity dataset we use 4 GeForce GTX 980 (6GB) GPUs with a budget of 6 hours and for Cifar10 we use 4 K80 (12GB) GPUs with a budget of 10 hours. For the regression datasets, we train each model with stochastic gradient descent (SGD) with a fixed step size of 10^{-5}, a batch size of 256 for 20K batch iterations. For Cifar10, we start with a step size of 10^{-2}, and reduce it gradually. We train in batches of 32 images for 60K batch iterations. The methods evaluate between 70-120 networks depending on the size of the networks chosen and the number of GPUs.

Results: Fig. 2 plots the best validation score for each method against time. In Table 3, we present the results on the test set with the best model chosen on the basis of validation set performance. On the Cifar10 dataset, we also trained the best models for longer ($150K$ iterations). These results are in the last column of Table 3. We see that NASBOT is the most consistent of all methods. The average time taken by NASBOT to determine the next architecture to evaluate was 46.13s. For RAND, EA, and TreeBO this was 26.43s, 0.19s, and 7.83s respectively. The time taken to train and validate models was on the order of 10-40 minutes depending on the model size. Fig. 2 includes this time taken to determine the next point. Like many BO algorithms, while NASBOT's selection criterion is time consuming, it pays off when evaluations are expensive. In Appendices B and C, we provide additional details on the experiment set up and conduct synthetic ablation studies by holding out different components of the NASBOT framework. We also illustrate some of the best architectures found—on many datasets, common features were long skip connections and multiple decision layers.

Finally, we note that while our Cifar10 experiments fall short of the current state of the art [22, 23, 50], the amount of computation in these work is several orders of magnitude more than ours (both the computation invested to train a single model and the number of models trained). Further, they use constrained spaces specialised for CNNs, while NASBOT is deployed in a very general model space. We believe that our results can also be improved by employing enhanced training techniques such as image whitening, image flipping, drop out, etc. For example, using our training procedure on the VGG-19 architecture [34] yielded a test set error of 0.1018 after $150K$ iterations. However, VGG-19 is known to do significantly better on Cifar10. That said, we believe our results are encouraging and lay out the premise for BO for neural architectures.

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. 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.

Acknolwedgements

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

[1] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

[2] James Bergstra, Daniel Yamins, and David Daniel Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. 2013.

[3] Eric Brochu, Vlad M. Cora, and Nando de Freitas. A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning. *CoRR*, 2010.

[4] Krisztian Buza. Feedback prediction for blogs. In *Data analysis, machine learning and knowledge discovery*, pages 145–152. Springer, 2014.

[5] Andrea Coraddu, Luca Oneto, Aessandro Ghio, Stefano Savio, Davide Anguita, and Massimo Figari. Machine learning approaches for improving condition-based maintenance of naval propulsion plants. *Proceedings of the Institution of Mechanical Engineers, Part M: Journal of Engineering for the Maritime Environment*, 230(1):136–153, 2016.

[6] Corinna Cortes, Xavi Gonzalvo, Vitaly Kuznetsov, Mehryar Mohri, and Scott Yang. Adanet: Adaptive structural learning of artificial neural networks. *arXiv preprint arXiv:1607.01097*, 2016.

[7] Kelwin Fernandes, Pedro Vinagre, and Paulo Cortez. A proactive intelligent decision support system for predicting the popularity of online news. In *Portuguese Conference on Artificial Intelligence*, pages 535–546. Springer, 2015.

[8] Dario Floreano, Peter Dürr, and Claudio Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1(1):47–62, 2008.

[9] Xinbo Gao, Bing Xiao, Dacheng Tao, and Xuelong Li. A survey of graph edit distance. *Pattern Analysis and applications*, 13(1):113–129, 2010.

[10] David Ginsbourger, Janis Janusevskis, and Rodolphe Le Riche. Dealing with asynchronicity in parallel gaussian process based global optimization. In *4th International Conference of the ERCIM WG on computing & statistics (ERCIM'11)*, 2011.

[11] Franz Graf, Hans-Peter Kriegel, Matthias Schubert, Sebastian Pölsterl, and Alexander Cavallaro. 2d image registration in ct images using radial image descriptors. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 607–614. Springer, 2011.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[13] Gao Huang, Zhuang Liu, Kilian Q Weinberger, and Laurens van der Maaten. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017.

[14] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *LION*, 2011.

[15] Rodolphe Jenatton, Cedric Archambeau, Javier González, and Matthias Seeger. Bayesian optimization with tree-structured dependencies. In *International Conference on Machine Learning*, pages 1655–1664, 2017.

[16] Nan Jiang, Akshay Krishnamurthy, Alekh Agarwal, John Langford, and Robert E Schapire. Contextual decision processes with low bellman rank are pac-learnable. *arXiv preprint arXiv:1610.09512*, 2016.

[17] Kirthevasan Kandasamy, Gautam Dasarathy, Jeff Schneider, and Barnabas Poczos. Multi-fidelity bayesian optimisation with continuous approximations. *arXiv preprint arXiv:1703.06240*, 2017.

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

[19] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast bayesian optimization of machine learning hyperparameters on large datasets. *arXiv preprint arXiv:1605.07079*, 2016.

[20] Risi Imre Kondor and John Lafferty. Diffusion kernels on graphs and other discrete input spaces. In *ICML*, volume 2, pages 315–322, 2002.

[21] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images, 2009.

[22] Chenxi Liu, Barret Zoph, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. *arXiv preprint arXiv:1712.00559*, 2017.

[23] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. *arXiv preprint arXiv:1711.00436*, 2017.

[24] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[25] Hector Mendoza, Aaron Klein, Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. Towards automatically-tuned neural networks. In *Workshop on Automatic Machine Learning*, pages 58–65, 2016.

[26] Bruno T Messmer and Horst Bunke. A new algorithm for error-tolerant subgraph isomorphism detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):493–504, 1998.

[27] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Olivier Francon, Bala Raju, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. Evolving deep neural networks. *arXiv preprint arXiv:1703.00548*, 2017.

[28] J.B. Mockus and L.J. Mockus. Bayesian approach to global optimization and application to multiobjective and constrained problems. *Journal of Optimization Theory and Applications*, 1991.

[29] Renato Negrinho and Geoff Gordon. Deeparchitect: Automatically designing and training deep architectures. *arXiv preprint arXiv:1704.08792*, 2017.

[30] Gabriel Peyré and Marco Cuturi. *Computational Optimal Transport*. Available online, 2017.

[31] PS Rana. Physicochemical properties of protein tertiary structure data set, 2013.

[32] C.E. Rasmussen and C.K.I. Williams. *Gaussian Processes for Machine Learning*. Adaptative computation and machine learning series. University Press Group Limited, 2006.

[33] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Quoc Le, and Alex Kurakin. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*, 2017.

[34] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[35] Alexander J Smola and Risi Kondor. Kernels and regularization on graphs. In *Learning theory and kernel machines*, pages 144–158. Springer, 2003.

[36] Edward Snelson and Zoubin Ghahramani. Sparse gaussian processes using pseudo-inputs. In *Advances in neural information processing systems*, pages 1257–1264, 2006.

[37] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical Bayesian Optimization of Machine Learning Algorithms. In *Advances in Neural Information Processing Systems*, 2012.

[38] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[39] Dougal J Sutherland. *Scalable, Active and Flexible Learning on Distributions*. PhD thesis, Carnegie Mellon University Pittsburgh, PA, 2015.

[40] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[41] Kevin Swersky, David Duvenaud, Jasper Snoek, Frank Hutter, and Michael A Osborne. Raiders of the lost architecture: Kernels for bayesian optimization in conditional parameter spaces. *arXiv preprint arXiv:1409.4011*, 2014.

[42] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[43] Joaquín Torres-Sospedra, Raúl Montoliu, Adolfo Martínez-Usó, Joan P Avariento, Tomás J Arnau, Mauri Benedito-Bordonau, and Joaquín Huerta. Ujiindoorloc: A new multi-building and multi-floor database for wlan fingerprint-based indoor localization problems. In *Indoor Positioning and Indoor Navigation (IPIN), 2014 International Conference on*, pages 261–270. IEEE, 2014.

[44] Cédric Villani. *Optimal transport: old and new*, volume 338. Springer Science & Business Media, 2008.

[45] S Vichy N Vishwanathan, Nicol N Schraudolph, Risi Kondor, and Karsten M Borgwardt. Graph kernels. *Journal of Machine Learning Research*, 11(Apr):1201–1242, 2010.

[46] Walter D Wallis, Peter Shoubridge, M Kraetz, and D Ray. Graph distances using graph union. *Pattern Recognition Letters*, 22(6-7):701–704, 2001.

[47] Andrew Wilson and Hannes Nickisch. Kernel interpolation for scalable structured gaussian processes (kiss-gp). In *International Conference on Machine Learning*, pages 1775–1784, 2015.

[48] Lingxi Xie and Alan Yuille. Genetic cnn. *arXiv preprint arXiv:1703.01513*, 2017.

[49] Zhao Zhong, Junjie Yan, and Cheng-Lin Liu. Practical network blocks design with q-learning. *arXiv preprint arXiv:1708.05552*, 2017.

[50] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[51] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. *arXiv preprint arXiv:1707.07012*, 2017.

A Additional Details on OTMANN

A.1 Optimal Transport Reformulation

We begin with a review optimal transport. Throughout this section, $\langle \cdot, \cdot \rangle$ denotes the Frobenius dot product. $\mathbf{1}_n, \mathbf{0}_n \in \mathbb{R}^n$ 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 $\mathbf{1}_{n_1}^\top y_1 = \mathbf{1}_{n_2}^\top y_2$. Let $C \in \mathbb{R}_+^{n_1 \times n_2}$. The following optimisation problem,

$$\begin{aligned} &\underset{Z}{\text{minimise}} && \langle Z, C \rangle \\ &\text{subject to} && Z > 0, \;\; Z\mathbf{1}_{n_2} = y_1, \;\; Z^\top \mathbf{1}_{n_1} = y_2. \end{aligned} \tag{4}$$

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 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 $\mathcal{G}_1, \mathcal{G}_2$, for $i = 1, 2$, let $tm(\mathcal{G}_i) = \sum_{u \in \mathcal{L}_i} \ell m(u)$ denote the total mass in $\mathcal{G}_i$, and $\bar{n}_i = n_i + 1$ where $n_i = |\mathcal{L}_i|$. $y_1 = [\{\ell m(u)\}_{u \in \mathcal{L}_1}, tm(\mathcal{G}_2)] \in \mathbb{R}^{\bar{n}_1}$ will be the supplies in our OT problem, and $y_2 = [\{\ell m(u)\}_{u \in \mathcal{L}_2}, tm(\mathcal{G}_1)] \in \mathbb{R}^{\bar{n}_2}$ will be the demands. To define the cost matrix, we augment the mislabel and structural penalty matrices $C_{\text{lmm}}, C_{\text{str}}$ with an additional row and column of zeros; i.e. $C'_{\text{lmm}} = [C_{\text{lmm}}\, \mathbf{0}_{n_1}; \mathbf{0}_{\bar{n}_2}^\top] \in \mathbb{R}^{\bar{n}_1 \times \bar{n}_2}$; C'_{str} is defined similarly. Let $C'_{\text{nas}} = [\mathbf{0}_{n_1, n_2}\, \mathbf{1}_{n_1}; \mathbf{1}_{n_2}^\top\, 0] \in \mathbb{R}^{\bar{n}_1 \times \bar{n}_2}$. We will show that (3) is equivalent to the following OT program.

$$\begin{aligned} &\underset{Z'}{\text{minimise}} && \langle Z', C' \rangle \\ &\text{subject to} && Z'\mathbf{1}_{\bar{n}_2} = y_1, \quad Z'^\top \mathbf{1}_{\bar{n}_1} = y_2. \end{aligned} \tag{5}$$

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 ∞'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 $Z' \in \mathbb{R}^{\bar{n}_1 \times \bar{n}_2}$ be such that its first $n_1 \times n_2$ block is Z and, $Z_{\bar{n}_1 j} = \sum_{i=1}^{n_1} Z_{ij}$, $Z_{i\bar{n}_2} = \sum_{j=1}^{n_2} Z_{ij}$, 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 $\sum_j Z'_{\bar{n}_1 j} Z'_{ij} = \sum_j \ell m(j) - \sum_{ij} Z_{ij} + Z_{\bar{n}_1, \bar{n}_2} = tm(\mathcal{G}_2)$. We then have, $Z'\mathbf{1}_{\bar{n}_2} = y_1$ Similarly, we can show $Z'^\top \mathbf{1}_{\bar{n}_1} = y_2$. Therefore, Z' is feasible for (5). We see that the objectives are equal via simple calculations,

$$\begin{aligned} \langle Z', C' \rangle &= \langle Z', C'_{\text{lmm}} + C'_{\text{str}} \rangle + \langle Z', C'_{\text{nas}} \rangle \\ &= \langle Z, C_{\text{lmm}} + C_{\text{str}} \rangle + \sum_{j=1}^{n_2} Z'_{ij} + \sum_{i=1}^{n_1} Z'_{ij} \\ &= \langle Z, C_{\text{lmm}} \rangle + \langle Z, C_{\text{str}} \rangle + \sum_{i \in \mathcal{L}_1} \big(\ell m(i) - \sum_{j \in \mathcal{L}_2} Z_{ij}\big) + \sum_{j \in \mathcal{L}_2} \big(\ell m(j) - \sum_{i \in \mathcal{L}_1} Z_{ij}\big). \end{aligned} \tag{6}$$

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. $\square$

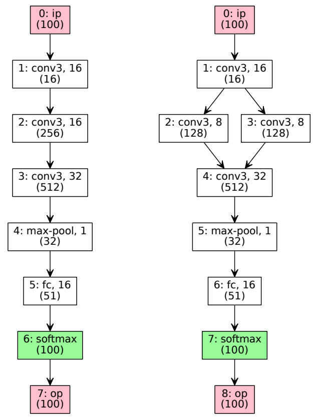


Figure 3: An example of 2 CNNs which have $d = \bar{d} = 0$ distance. The OT solution matches the mass in each layer in the network on the left to the layer horizontally opposite to it on the right with 0 cost. For layer 2 on the left, its mass is mapped to layers 2 and 3 on the left. However, while the descriptor of these networks is different, their functional behaviour is the same.

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* $\mathtt{x}$, $\mathtt{y}$, $\mathtt{z}$ *we have* $M(\mathtt{x},\mathtt{z}) \leq M(\mathtt{x},\mathtt{y}) + M(\mathtt{y},\mathtt{z})$. *Let $d(\mathcal{G}_1, \mathcal{G}_2)$ be the solution of* (3) *for networks $\mathcal{G}_1, \mathcal{G}_2$. Then $d(\cdot,\cdot)$ is a pseudo-distance. That is, for all networks $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3$, it satisfies, $d(\mathcal{G}_1,\mathcal{G}_2) > 0$, $d(\mathcal{G}_1,\mathcal{G}_2) = d(\mathcal{G}_2,\mathcal{G}_1)$, $d(\mathcal{G}_1,\mathcal{G}_1) = 0$ and $d(\mathcal{G}_1,\mathcal{G}_3) \leq d(\mathcal{G}_1,\mathcal{G}_2) + d(\mathcal{G}_2,\mathcal{G}_3)$.*

Some remarks are in order. First, observe that while $d(\cdot,\cdot)$ is a pseudo-distance, it is not a distance; i.e. $d(\mathcal{G}_1,\mathcal{G}_2) = 0 \not\Rightarrow \mathcal{G}_1 = \mathcal{G}_2$. 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 $d(\mathcal{G}_2,\mathcal{G}_1)$ is $C'^\top$ if the cost matrix for $d(\mathcal{G}_1,\mathcal{G}_2)$ is C and $\langle Z', C'\rangle = \langle Z'^\top, C'^\top\rangle$ for all Z'. We also have $d(\mathcal{G}_1,\mathcal{G}_1) = 0$ 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 $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3$ be given and m_1, m_2, m_3 be their total masses. Let the solutions to $d(\mathcal{G}_1,\mathcal{G}_2)$ and $d(\mathcal{G}_2,\mathcal{G}_3)$ be $P \in \mathbb{R}^{\bar{n}_1\times\bar{n}_2}$ and $Q \in \mathbb{R}^{\bar{n}_2\times\bar{n}_3}$ respectively. When solving (5), we see that adding extra mass to the non-assignment layers does not change the objective, as an optimiser can transport mass between the two layers with 0 cost. Hence, we can assume w.l.o.g that (5) was solved with $y_i = \left[\{\ell m(u)\}_{u\in\mathcal{L}_i}, \left(\sum_{j\in\{1,2,3\}} tm(\mathcal{G}_j) - tm(\mathcal{G}_i)\right)\right] \in \mathbb{R}^{\bar{n}_i}$ for $i = 1,2,3$, when computing the distances $d(\mathcal{G}_1,\mathcal{G}_2)$, $d(\mathcal{G}_1,\mathcal{G}_3)$, $d(\mathcal{G}_2,\mathcal{G}_3)$; i.e. the total mass was $m_1+m_2+m_3$ for all three pairs. We can similarly assume that P, Q account for this extra mass, i.e. $P_{\bar{n}_1\bar{n}_2}$ and $Q_{\bar{n}_2\bar{n}_3}$ have been increased by m_3 and m_1 respectively from their solutions in (5).

To apply the gluing lemma, let $S = P\mathrm{diag}(1/y_2)Q \in \mathbb{R}^{\bar{n}_1\times\bar{n}_3}$, where $\mathrm{diag}(1/y_2)$ is a diagonal matrix whose $(j,j)^{\text{th}}$ element is $1/(y_2)_j$ (note $y_2 > 0$). We see that S is feasible for (5) when computing $d(\mathcal{G}_1,\mathcal{G}_3)$,

$$R\mathbf{1}_{\bar{n}_3} = P\mathrm{diag}(1/y_2)Q\mathbf{1}_{\bar{n}_3} = P\mathrm{diag}(1/y_2)y_2 = P\mathbf{1}_{\bar{n}_2} = y_1.$$

Similarly, $R^\top\mathbf{1}_{\bar{n}_1} = y_3$. Now, let U', V', W' be the cost matrices C' in (5) when computing $d(\mathcal{G}_1,\mathcal{G}_2)$, $d(\mathcal{G}_2,\mathcal{G}_3)$, and $d(\mathcal{G}_1,\mathcal{G}_3)$ respectively. We will use the following technical lemma whose proof is given below.

Lemma 4. *For all $i \in \mathcal{L}_1$, $j \in \mathcal{L}_2$, $k \in \mathcal{L}_3$, we have $W'_{ik} \leq U'_{ij} + V'_{jk}$.*

Applying Lemma 4 yields the triangle inequality.

$$\begin{aligned} d(\mathcal{G}_1, \mathcal{G}_3) \le \langle R, W' \rangle &= \sum_{i \in \mathcal{L}_1, k \in \mathcal{L}_3} W'_{ik} \sum_{j \in \mathcal{L}_2} \frac{P_{ij} Q_{jk}}{(y_2)_j} \le \sum_{i,j,k} (U'_{ij} + V'_{jk}) \frac{P_{ij} Q_{jk}}{(y_2)_j} \\ &= \sum_{ij} \frac{U'_{ij} P_{ij}}{(y_2)_j} \sum_k Q_{jk} + \sum_{jk} \frac{V'_{jk} Q_{jk}}{(y_2)_j} \sum_k P_{ij} \\ &= \sum_{ij} U'_{ij} P_{ij} + \sum_{jk} V'_{jk} Q_{jk} = d(\mathcal{G}_1, \mathcal{G}_2) + d(\mathcal{G}_2, \mathcal{G}_3) \end{aligned}$$

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

Proof of Lemma 4. Let $W' = W'_{\text{lmm}} + W'_{\text{str}} + W'_{\text{nas}}$ be the decomposition into the label mismatch, structural and non-assignment parts of the cost matrices; define similar quantities $U'_{\text{lmm}}, U'_{\text{str}}, U'_{\text{nas}}, V'_{\text{lmm}}, V'_{\text{str}}, V'_{\text{nas}}$ for U', V'. Noting $a \le b+c$ and $d \le e+f$ implies $a+d \le b+e+c+f$, it is sufficient to show the triangle inquality for each component individually. For the label mismatch term, $(W'_{\text{lmm}})_{ik} \le (U'_{\text{lmm}})_{ij} + (V'_{\text{lmm}})_{jk}$ follows directly from the conditions on M by setting $\mathtt{x} = \ell\ell(i)$, $\mathtt{y} = \ell\ell(j)$, $\mathtt{z} = \ell\ell(k)$, where i, j, k are indexing in $\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3$ respectively.

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

Finally, for the structural terms we note that W'_{str} can be written as $W'_{\text{str}} = \sum_t W'^{(t)}$ as can $U'^{(t)}, T'^{(t)}$. Here t indexes over the choices for the types of distances considered, i.e. $t \in \{\text{sp}, \text{lp}, \text{rw}\} \times \{\text{ip}, \text{op}\}$. It is sufficient to show $(W'^{(t)})_{ik} \le (U'^{(t)})_{ij} + (T'^{(t)})_{jk}$. This inequality takes the form,

$$|\delta^{(t)}_{1i} - \delta^{(t)}_{3k}| \le |\delta^{(t)}_{1i} - \delta^{(t)}_{2j}| + |\delta^{(t)}_{2j} - \delta^{(t)}_{3k}|.$$

Where $\delta^{(t)}_{g\ell}$ 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 $\delta^{\text{rw}}_{\text{op}}(u)$ for all $u \in \mathcal{L}$

Require: $\mathcal{G} = (\mathcal{L}, \mathcal{E})$, $\mathcal{L}$ is topologically sorted in S.
1: $\delta^{\text{rw}}_{\text{op}}(u_{\text{op}}) = 0$, $\delta^{\text{rw}}_{\text{op}}(u) = \mathtt{nan}\ \forall u \ne u_{\text{op}}$.
2: **while** S is not empty **do**
3: $\quad u \leftarrow \text{pop\_last}(S)$
4: $\quad \Delta \leftarrow \{\delta^{\text{rw}}_{\text{op}}(c) : c \in \text{children}(u)\}$
5: $\quad \delta^{\text{rw}}_{\text{op}}(u) \leftarrow 1 + \text{average}(\Delta)$
6: **end while**
7: **Return** $\delta^{\text{rw}}_{\text{op}}$.

	c3	c5	c7	mp	ap	fc	sm
c3	0	0.2	0.3				
c5	0.2	0	0.2				
c7	0.3	0.2	0				
mp				0	0.25		
ap				0.25	0		
fc						0	
sm							0

Table 4: The label mismatch cost matrix M we used in our CNN experiments. $M(\mathtt{x},\mathtt{y})$ 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 ∞ cost. We have not shown the `ip` and `op` layers, but they are similar to the `fc` column, 0 in the diagonal and ∞ elsewhere.

	re	cr	<rec>	lg	ta	lin
re	0	.1	.1	.25	.25	
cr	.1	0	.1	.25	.25	
<rec>	.1	.1	0	.25	.25	
lg	.25	.25	.25	0	.1	
ta	.25	.25	.25	.1	0	
lin						0

Table 5: The label mismatch cost matrix M we used in our MLP experiments. The labels abbreviated are `relu`, `crelu`, `<rec>`, `logistic`, `tanh`, and `linear` in order. `<rec>` is place-holder for any other rectifier such as `leaky-relu`, `softplus`, `elu`. A blank indicates ∞ cost. The design here was simple. Each label gets 0 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 ∞ elsewhere.

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

For $\delta_{\mathrm{ip}}^{\mathrm{rw}}$ we make the following changes to Algorithm 1. In step 1, we set $\delta_{\mathrm{ip}}^{\mathrm{rw}}(u_{\mathrm{ip}}) = 0$, in step 3, we $\mathrm{pop\_first}$ and Δ in step 4 is computed using the parents. $\delta_{\mathrm{ip}}^{\mathrm{sp}}, \delta_{\mathrm{ip}}^{\mathrm{lp}}$ 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`, `softmax`, `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 $M(\mathtt{res}k, \mathtt{conv}j) = 0.9 \times M(\mathtt{conv}k, \mathtt{conv}j) + 0.1 \times 1$; 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`, `crelu`, `leaky-relu`, `softplus`, `elu`, `logistic`, `tanh`, `linear`, `ip`, `op`. Here the first seven are common non-linear activations; `relu`, `crelu`, `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 $0.1 \times \ell m(u) \times \langle$#-incoming-channels$\rangle$ 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_{str}. First, for $i \in \mathcal{L}_1,\ j \in \mathcal{L}_2$, we let $C_{\text{str}}^{\text{all}}(i,j) = \frac{1}{6}\sum_{s\in\{\text{sp, lp, rw}\}}\sum_{t\in\{\text{ip,op}\}}|\delta_t^s(i) - \delta_t^s(j)|$ be the average of *all* path length differences; i.e. $C_{\text{str}}^{\text{all}}$ 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 by zeroing 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 $C_{\text{str}}^{\text{all}}, C_{\text{str}}^{\text{rec}}, C_{\text{str}}^{\text{sig}}$ 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 noramlised 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 validaiton 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,

$$\kappa(\cdot,\cdot) = \alpha e^{-\sum_i \beta_i d_i^p(\cdot,\cdot)} + \bar{\alpha} e^{-\sum_i \bar{\beta}_i \bar{d}_i^{\bar{p}}(\cdot,\cdot)}. \tag{7}$$

Here, $d_i, \bar{d}_i$, are the OTMANN distance and its normalised counterpart developed in Section 3, computed with different values for $\nu_{\text{str}} \in \{\nu_{\text{str},i}\}_i$. $\beta_i, \bar{\beta}_i$ manage the relative contributions of $d_i, \bar{d}_i$, while $(\alpha, \bar{\alpha})$ 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 $d_i, \bar{d}_i$. In the experiments we report, we used $\{\nu_{\text{str},i}\}_i = \{0.1, 0.2, 0.4, 0.8\}$, $p = 1$ and $\bar{p} = 2$. 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; $\alpha, \bar{\alpha}, \{(\beta_i, \bar{\beta}_i)\}_{i=1}^4$ and the GP noise variance η^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

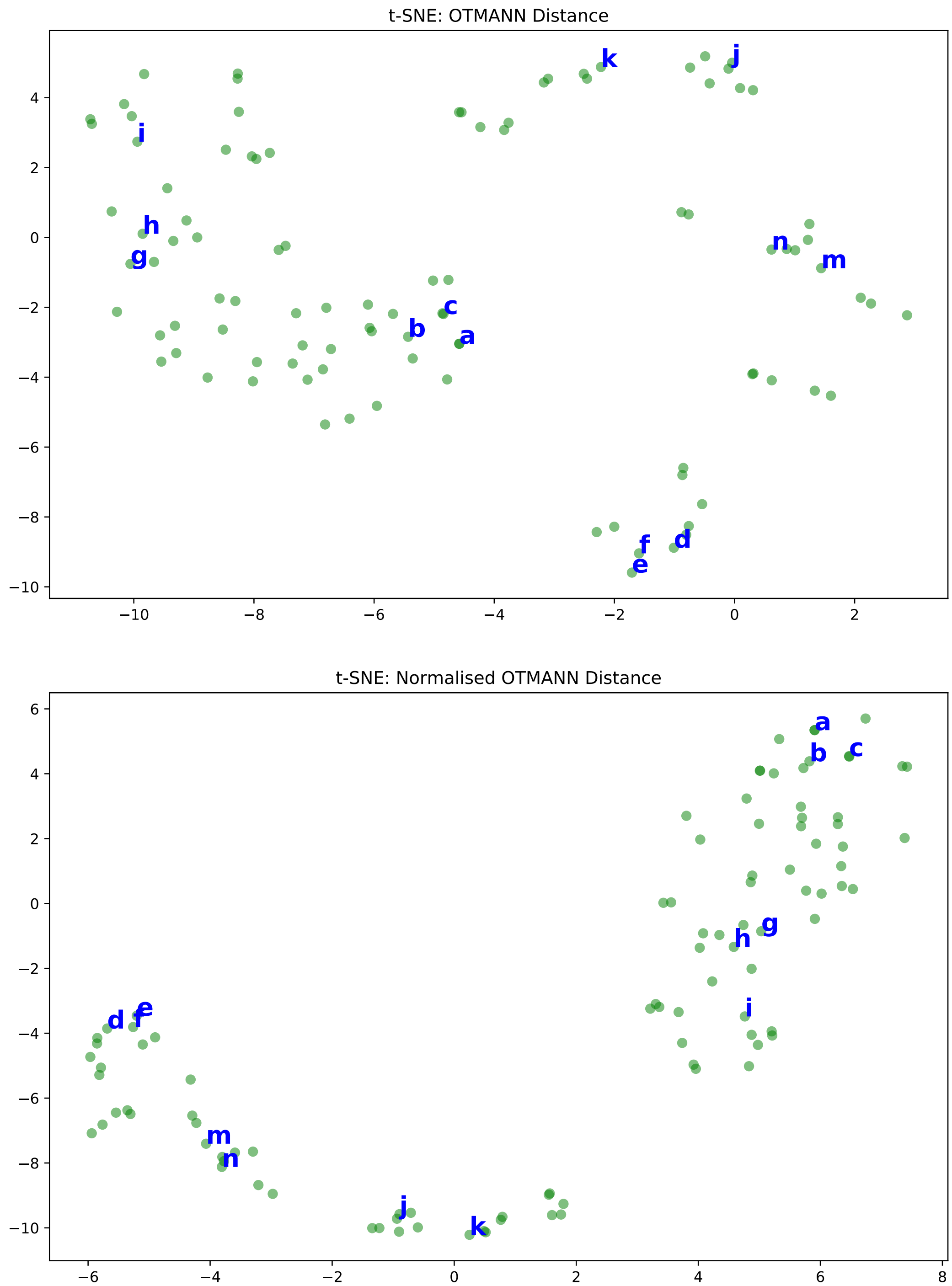


Figure 4: Two dimensional t-SNE embeddings of 100 randomly generated CNN architectures based on the OTMANN distance (top) and its normalised version (bottom). Some networks have been indexed a-n in the figures; these network architectures are illustrated in Figure 5. Networks that are similar are embedded close to each other indicating that the OTMANN induces a meaningful topology among neural network architectures.

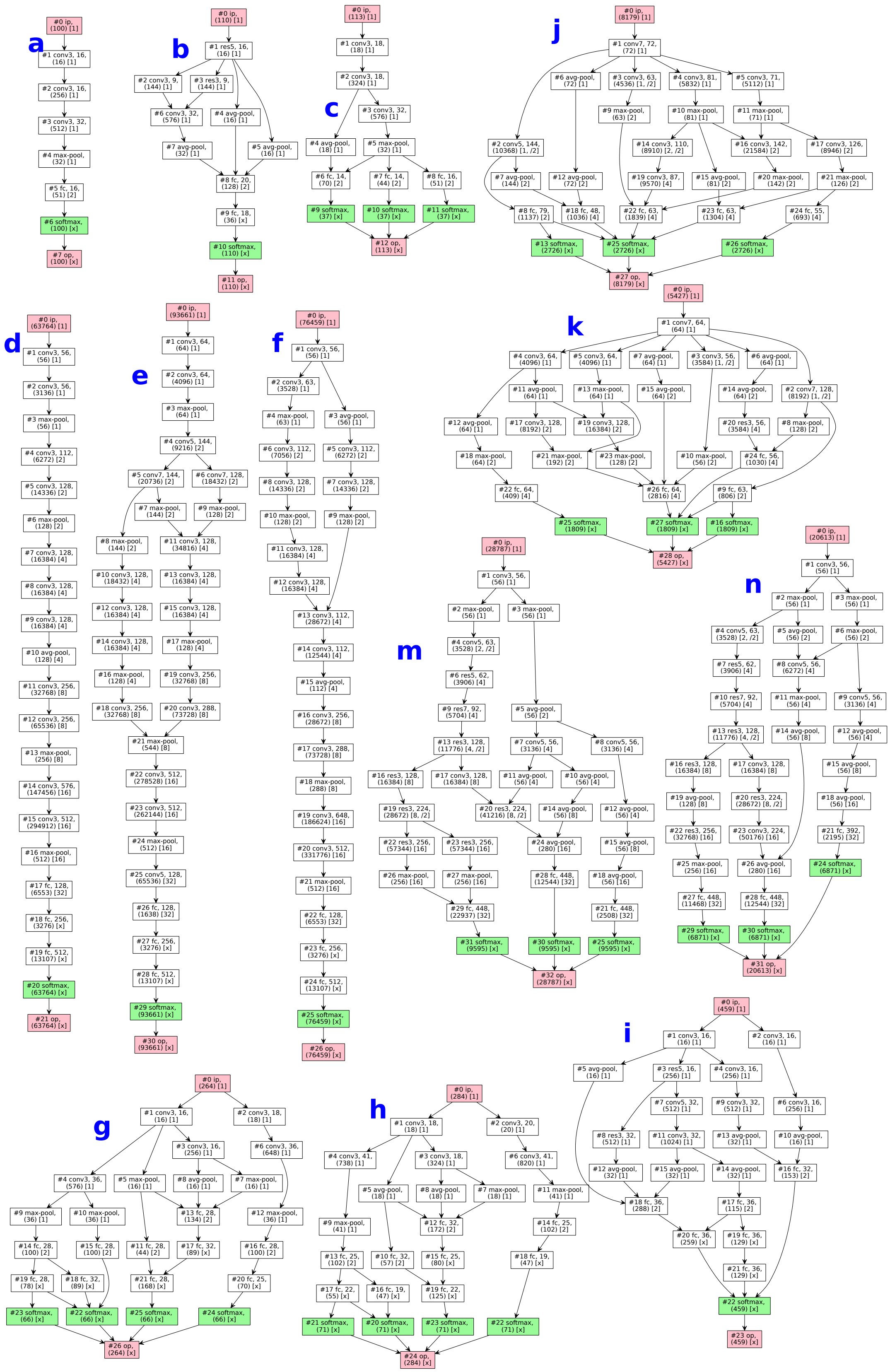


Figure 5: Illustrations of the nextworks indexed a-n in Figure 4.

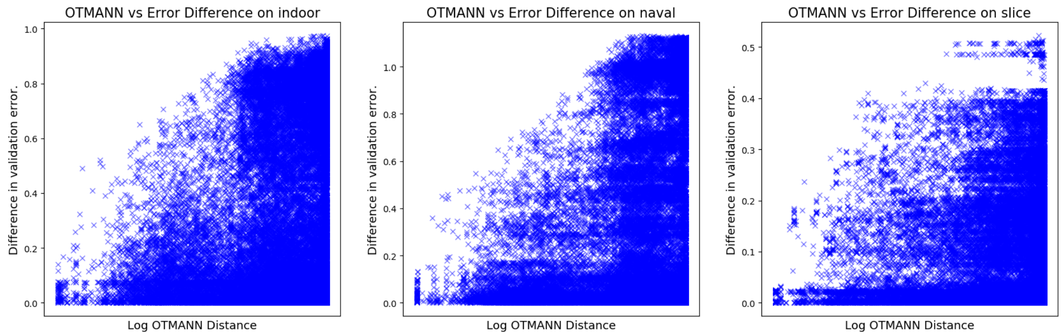


Figure 6: Each point in the scatter plot indicates the log distance between two architectures (x axis) and the difference in the validation error (y axis), on the Indoor, Naval and Slice datasets. We used 300 networks, giving rise to $\sim 45K$ pairwise points. On all datasets, when the distance is small, 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.

in practical settings. Our solution is to start with a (uniform) prior over these hyper-parameters and sample hyper-parameter values from the posterior under the GP likelihood [37], which we found to be robust. While it is possible to treat ν_{str} itself as a hyper-parameter of the kernel, this will require us to re-compute all pairwise distances of networks that have already been evaluated each time we change the hyper-parameters. On the other hand, with the above approach, we can compute and store distances for different $\nu_{\text{str},i}$ values whenever a new network is evaluated, and then compute the kernel cheaply for different values of $\alpha, \bar{\alpha}, \{(\beta_i, \bar{\beta}_i)\}_i$.

B.2 Optimising the Acquisition

We use a evolutionary algorithm (EA) approach to optimise the acquisition function (2). We begin with an initial pool of networks and evaluate the acquisition φ_t on those networks. Then we generate a set of N_{mut} mutations of this pool as follows. First, we stochastically select N_{mut} candidates from the set of networks already evaluated such that those with higher φ_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_{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. Typical implementations of EA in Euclidean spaces define the mutation operator via a Gaussian (or other) perturbation of a chosen candidate. It is instructive to think of the probabilities for each step in our scheme above as being analogous to the width of the Gaussian chosen for perturbation.

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_{mut} new points from a distribution π where $\pi(z_i) \propto \exp(g(z_i)/\sigma)$. Here g is the function to be optimised (for NASBOT, φ_t at time t). σ is the standard deviation of all previous evaluations. As the probability for large g values is higher, they are more likely to get selected. σ provides normalisation to account for different ranges of function values.

Operation	Description
dec_single	Pick a layer at random and decrease the number of units by $1/8$.
dec_en_masse	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 $1/2$.
inc_single	Pick a layer at random and increase the number of units by $1/8$.
inc_en_masse	Choose a large sub set of layers, as for dec_en_masse, and increase the number of units by $1/8$.
dup_path	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 $\tilde{u}_2, \ldots, \tilde{u}_{k-1}$ where $\tilde{u}_i = u_i$ for $i = 2, \ldots, k-1$. Add these layers along with new edges $(u_1, \tilde{u}_2)$, $(\tilde{u}_{k-1}, u_k)$, and $(\tilde{u}_j, \tilde{u}_{j+1})$ for $j = 2, \ldots, k-2$.
remove_layer	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).
skip	Randomly picks layers u, v where u is topologically before v and $(u, v) \notin \mathcal{E}$. Add (u, v) to $\mathcal{E}$.
swap_label	Randomly pick a layer and change its label.
wedge_layer	Randomly pick any edge $(u, v) \in \mathcal{E}$. Create a new layer w with a random label $\ell\ell(w)$. Remove (u, v) from $\mathcal{E}$ and add $(u, w), (w, v)$. If applicable, set the number of units $\ell u(w)$ to be $(\ell u(u) + \ell u(v))/2$.

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 φ_t is cheap, we can use many EA steps to explore several architectures and optimise φ_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_{EA} at rate $\mathcal{O}(\sqrt{t})$ where t is the current time step in NASBOT. We set N_{mut} to be $\mathcal{O}(\sqrt{n_{\mathsf{EA}}})$. 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 φ_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-pool` layers at stride 2 so that the connection can be made (this can be seen, for 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 `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 $\mathrm{Unif}(0, 1)$ instead of the GP acquisition each time it evaluates an architecture. That is, we follow the same schedule for n_{EA} and N_{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_{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.

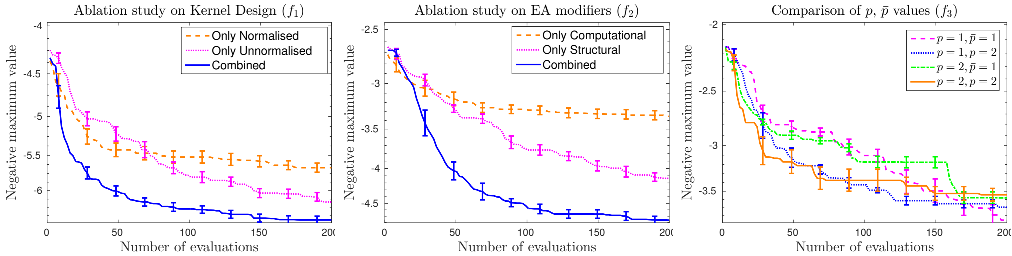
Finally, we experiment with different choices for p and $\bar{p}$ in (7). As the figures indicate, the performance was not particularly sensitive to these choices.

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, $\mathrm{deg_i}$ is the average in degree the layers, $\mathrm{deg_o}$ is the average out degree, δ is the shortest distance from u_{ip} to u_{op}, str is the average stride in CNNS, $\mathrm{frac\_conv3}$ is the fraction of layers that are `conv3`, $\mathrm{frac\_sigmoid}$ is the fraction of layers that are sigmoidal.

$$
\begin{aligned}
f_0 &= \exp(-0.001 * |\mathrm{am} - 1000|) + \exp(-0.5 * |\mathrm{deg_i} - 5|) + \exp(-0.5 * |\mathrm{deg_o} - 5|) + \\
&\quad \exp(-0.1 * |\delta - 5|) + \exp(-0.1 * ||\mathcal{L}| - 30|) + \exp(-0.05 * ||\mathcal{E}| - 100|) \\
f_1 &= f_0 + \exp(-3 * |\mathrm{str} - 1.5|) + \exp(-0.3 * ||\mathcal{L}| - 50|) + \\
&\quad \exp(-0.001 * |\mathrm{am} - 500|) + \mathrm{frac\_conv3} \\
f_2 &= f_0 + \exp(-0.001 * |\mathrm{am} - 2000|) + \exp(-0.1 * ||\mathcal{E}| - 50|) + \mathrm{frac\_sigmoid} \\
f_3 &= f_0 + \mathrm{frac\_sigmoid}
\end{aligned}
$$

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].



(a) (b) (c)

Figure 7: We compare NASBOT for different design choices in our framework. (a): Comparison of NASBOT using only the normalised distance $e^{-\beta\bar{d}}$, only the unnormalised distance $d^{-\beta d}$, and the combination $e^{-\beta d}+e^{-\bar{\beta}\bar{d}}$. (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.

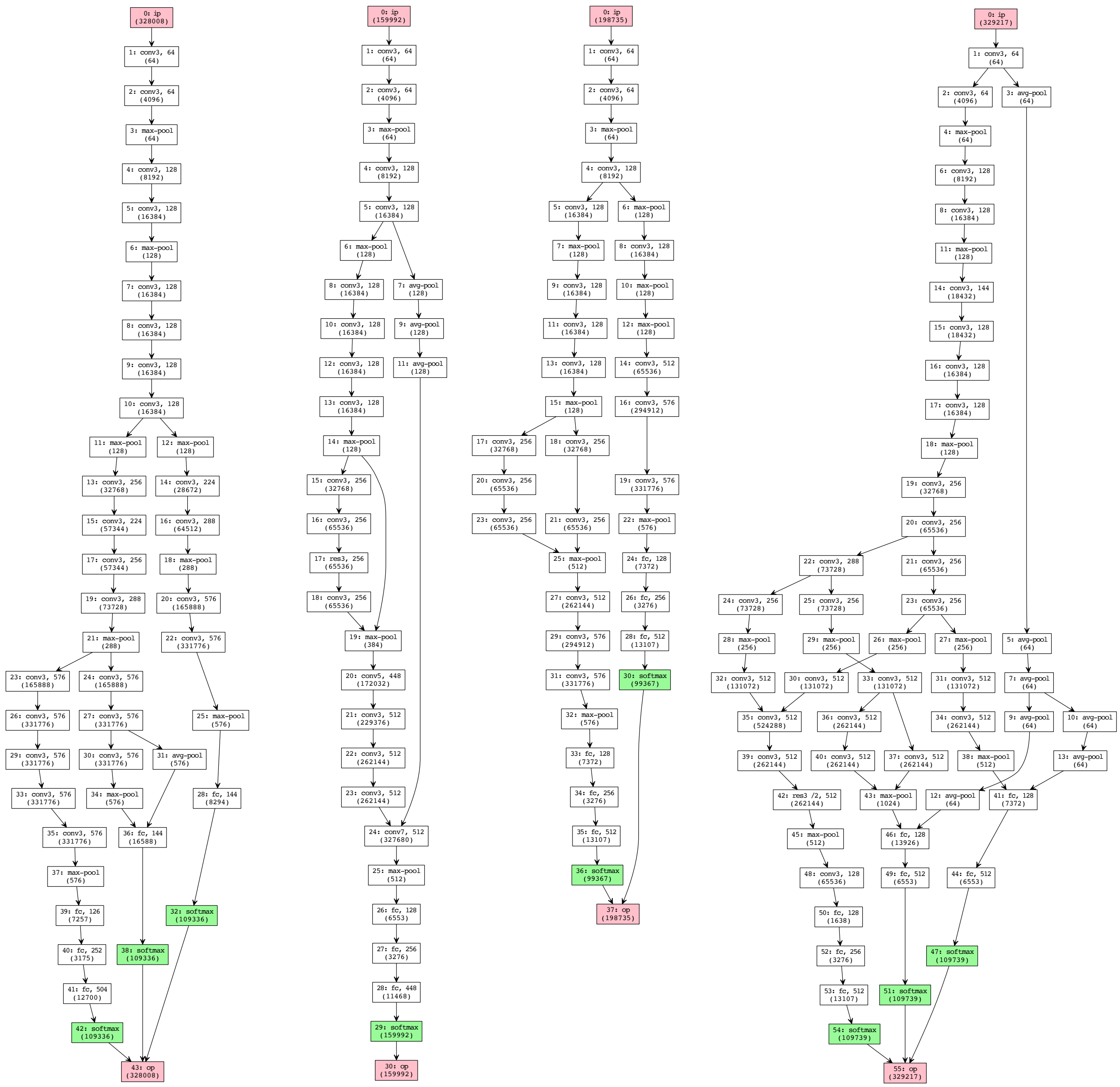


Figure 8: Optimal network architectures found with NASBOT on Cifar10 data.

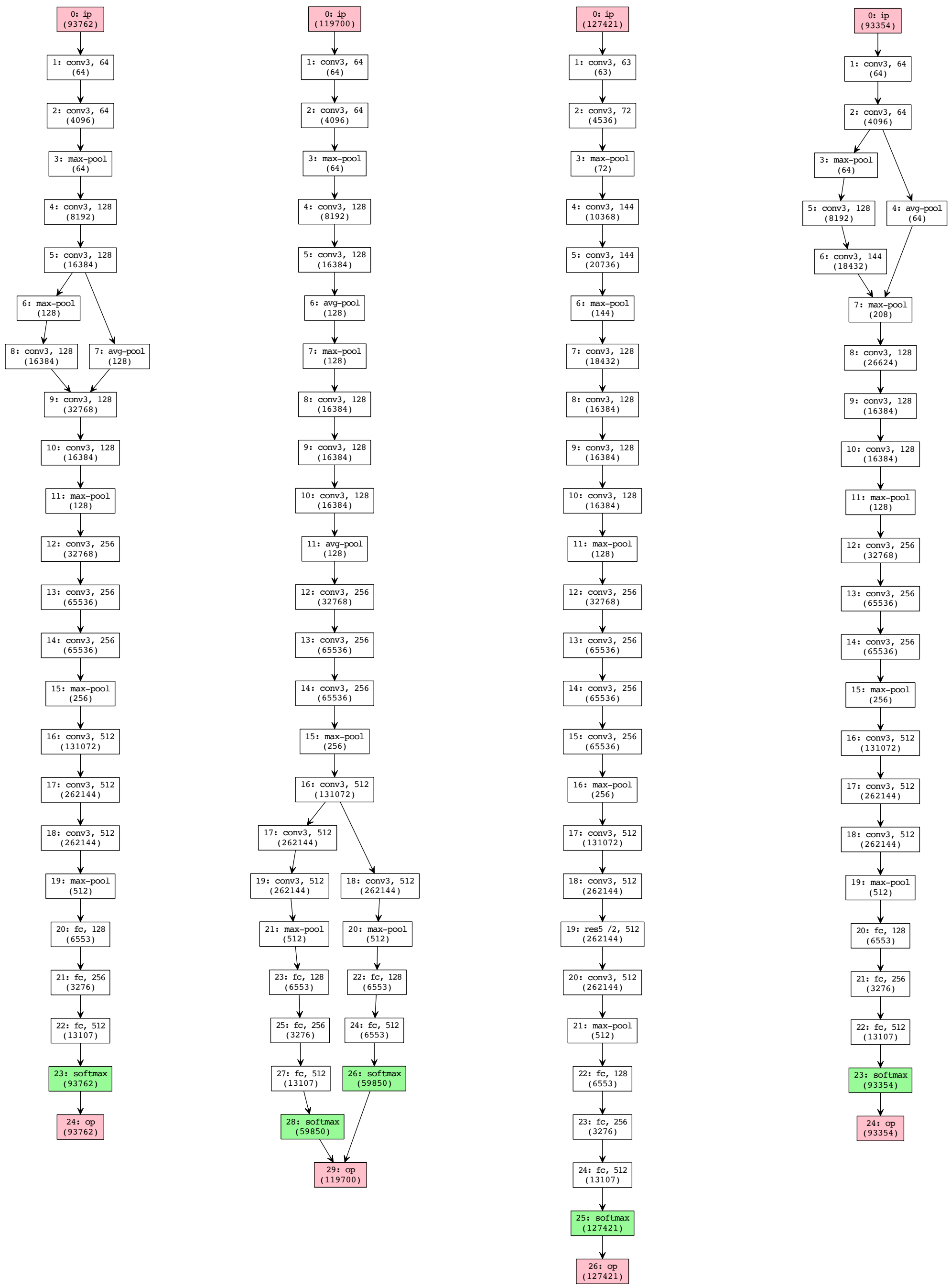


Figure 9: Optimal network architectures found with EA on Cifar10 data.

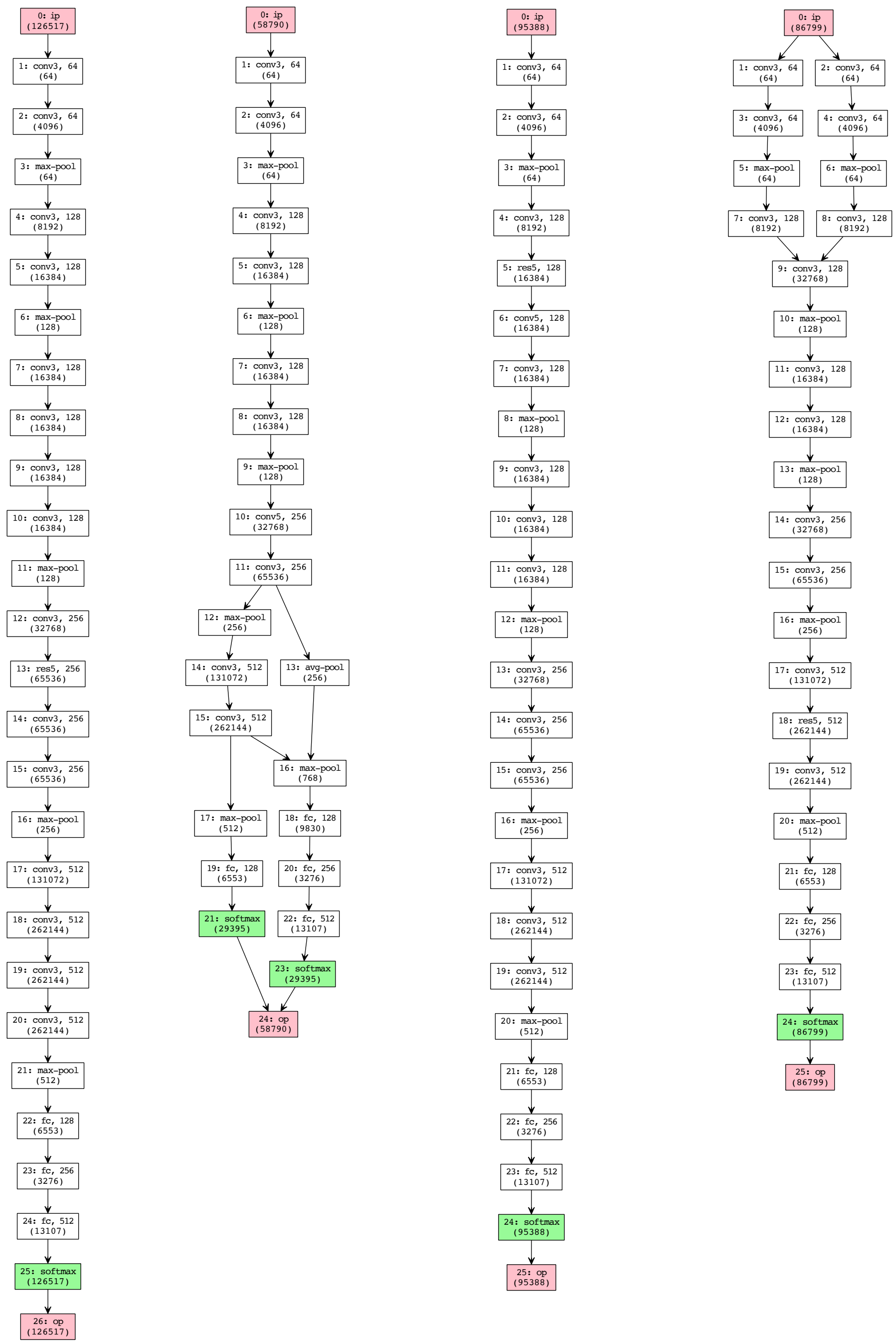


Figure 10: Optimal network architectures found with RAND on Cifar10 data.

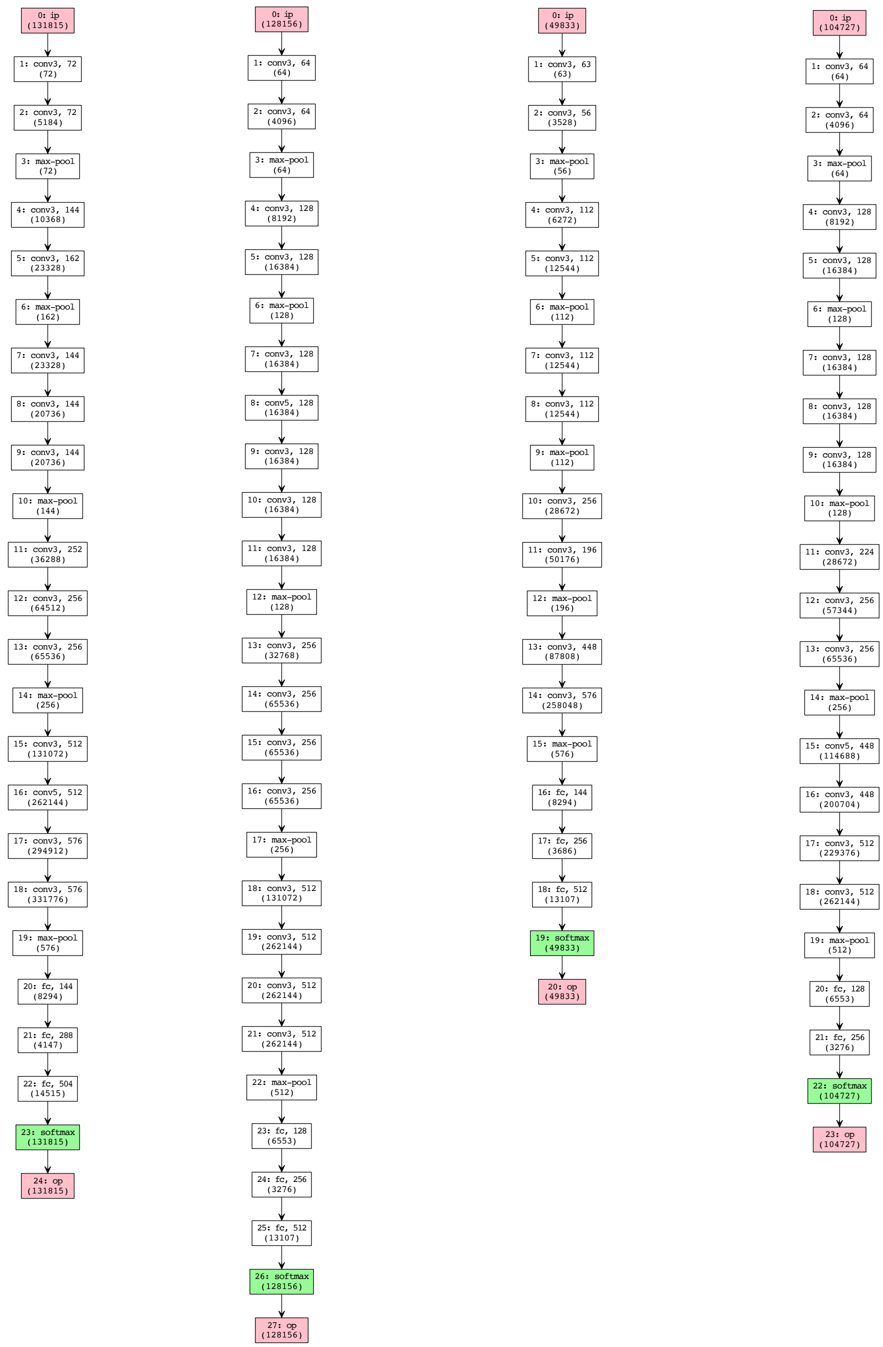


Figure 11: Optimal network architectures found with TreeBO on Cifar10 data.

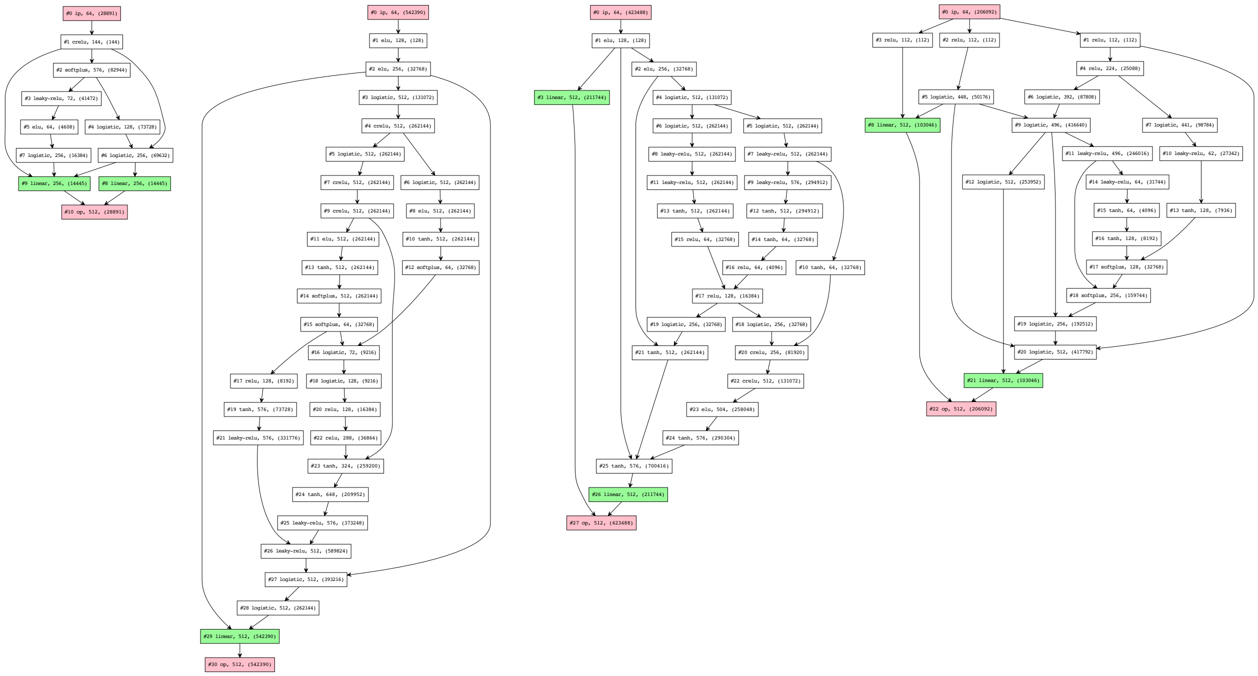


Figure 12: Optimal network architectures found with NASBOTon Indoor data.

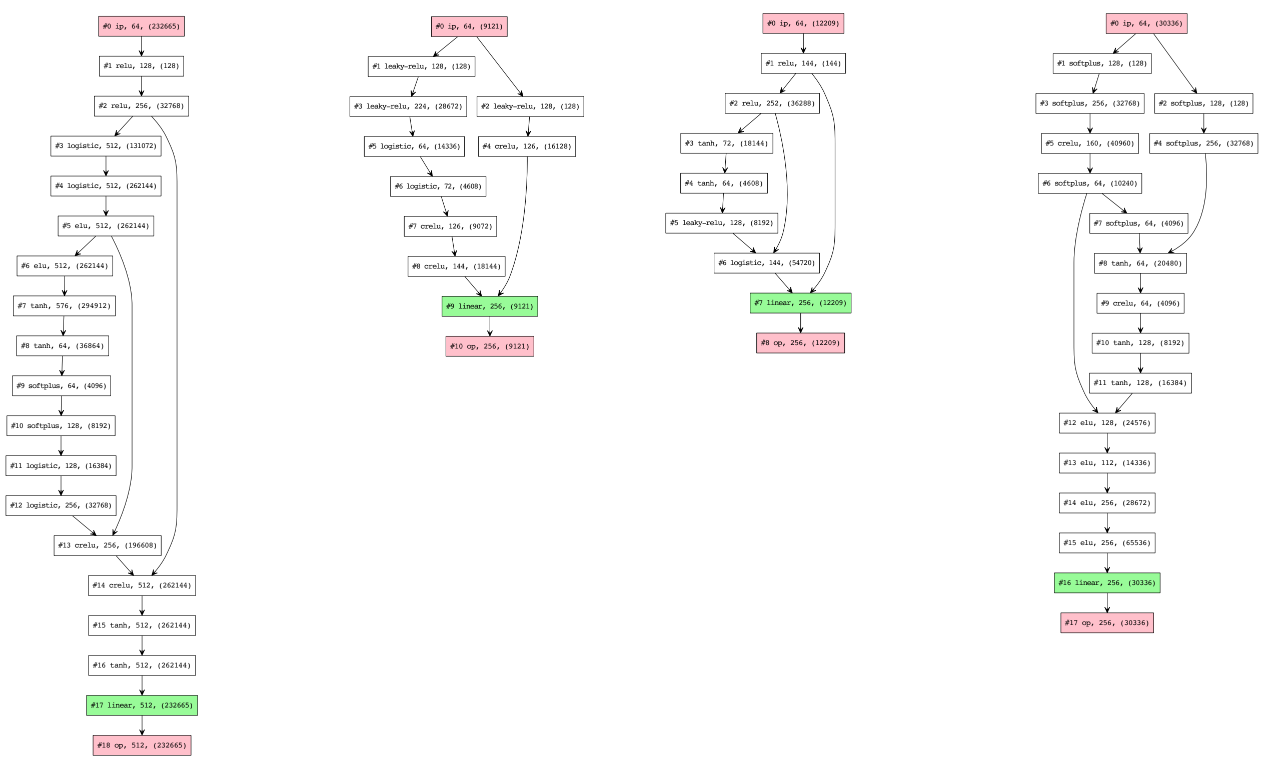


Figure 13: Optimal network architectures found with EA on Indoor data.

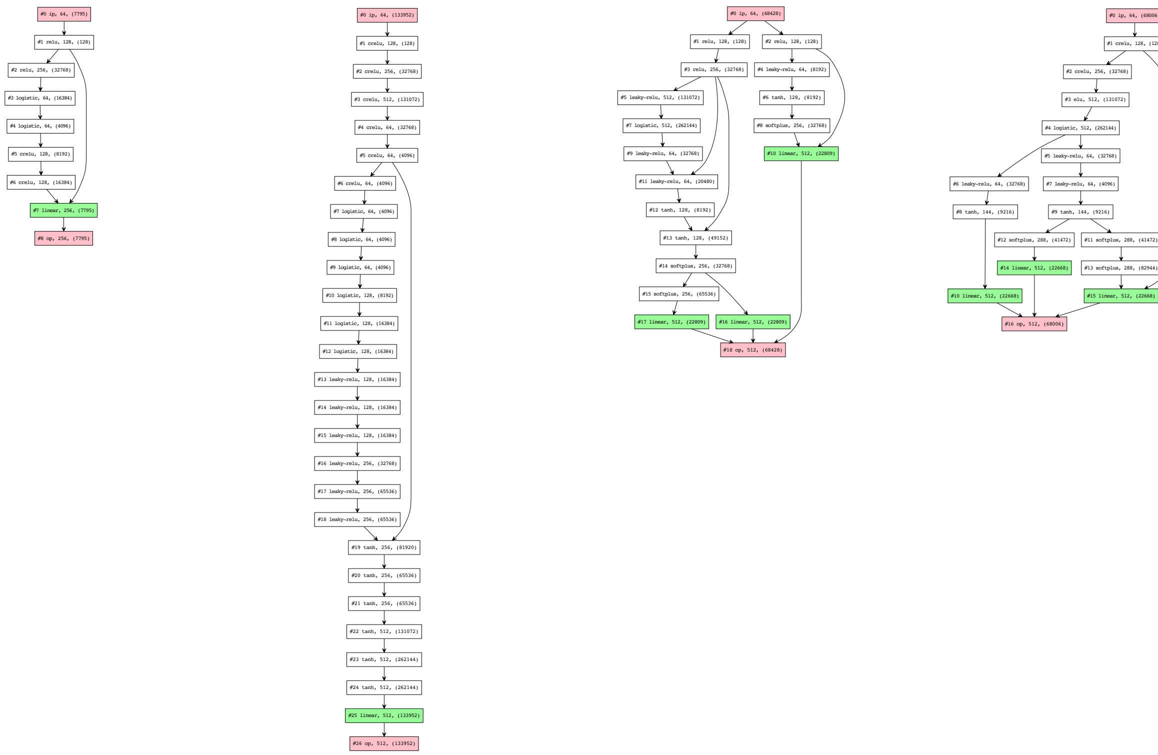


Figure 14: Optimal network architectures found with RAND on Indoor data.

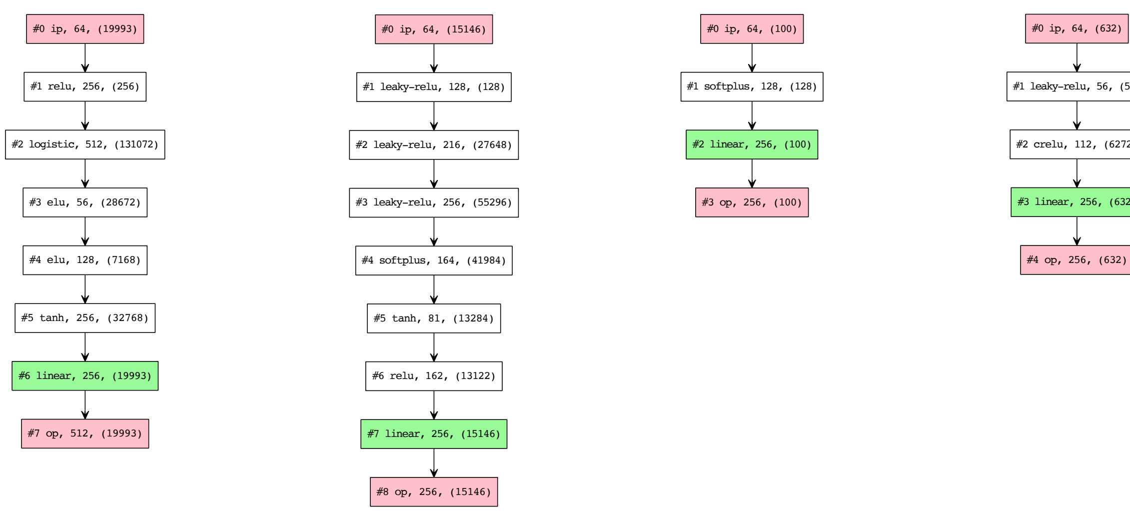


Figure 15: Optimal network architectures found with TreeBO on Indoor data.

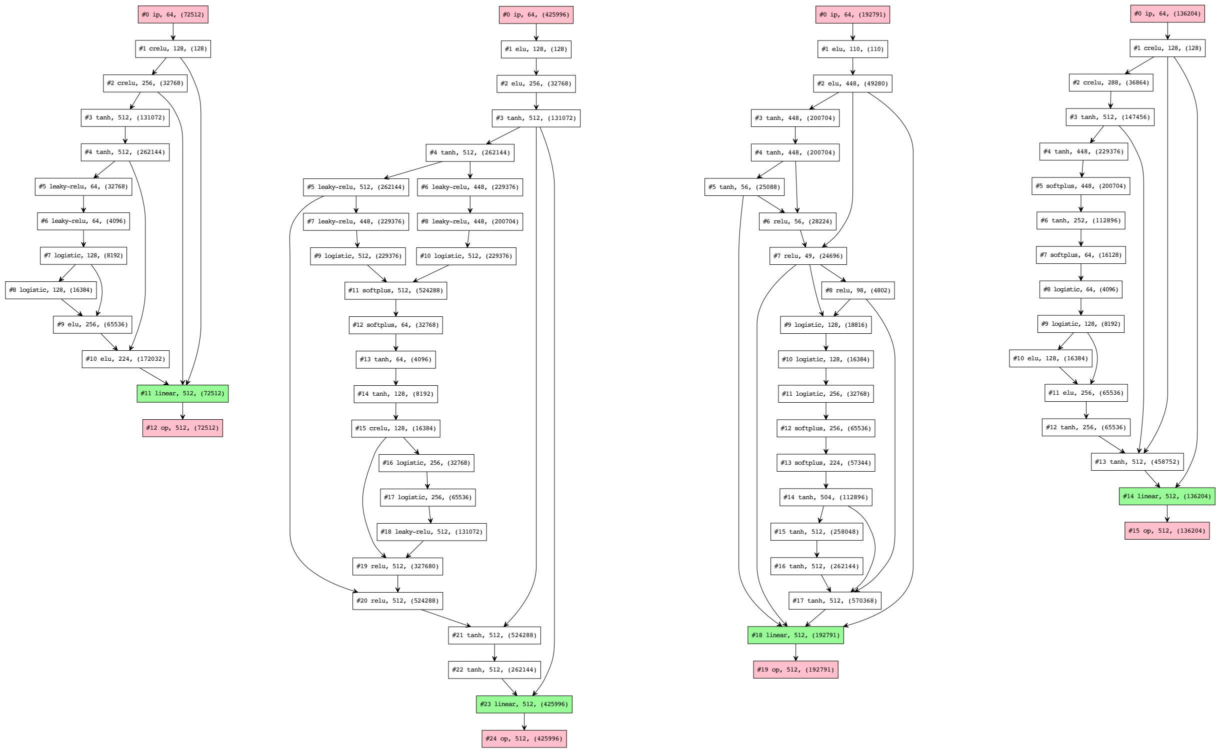


Figure 16: Optimal network architectures found with NASBOTon Slice data.

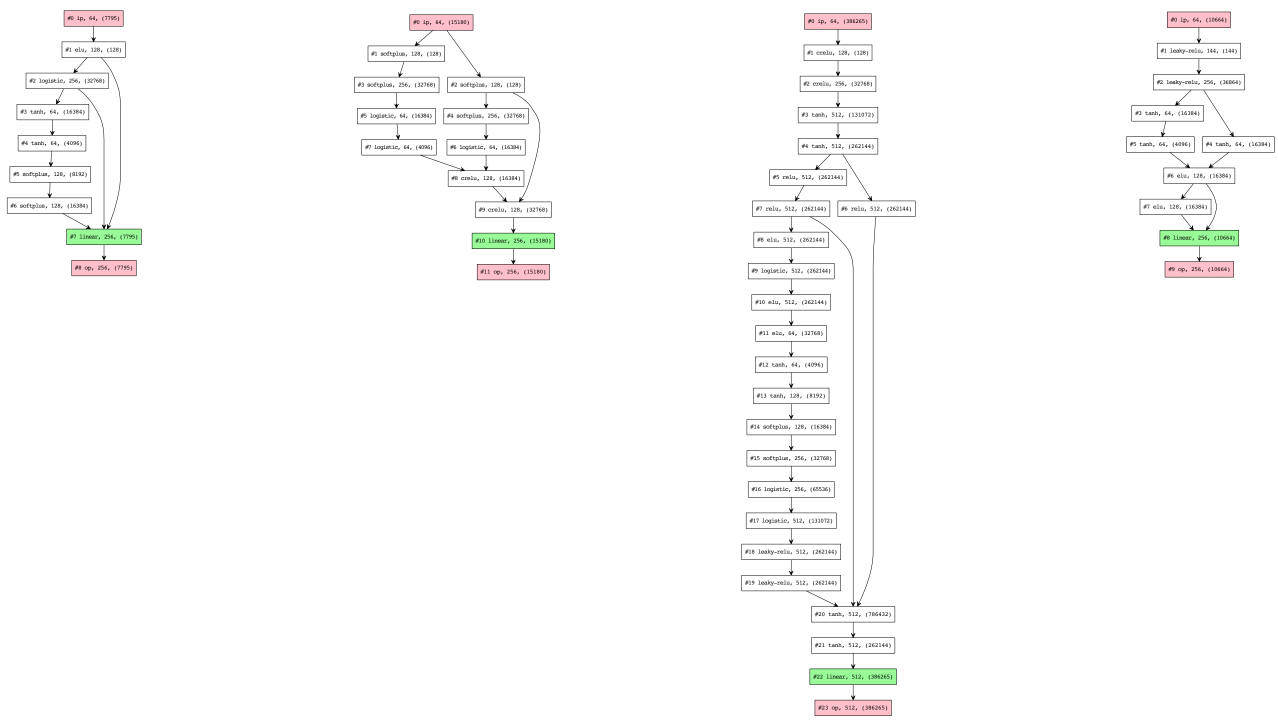


Figure 17: Optimal network architectures found with EA on Slice data.

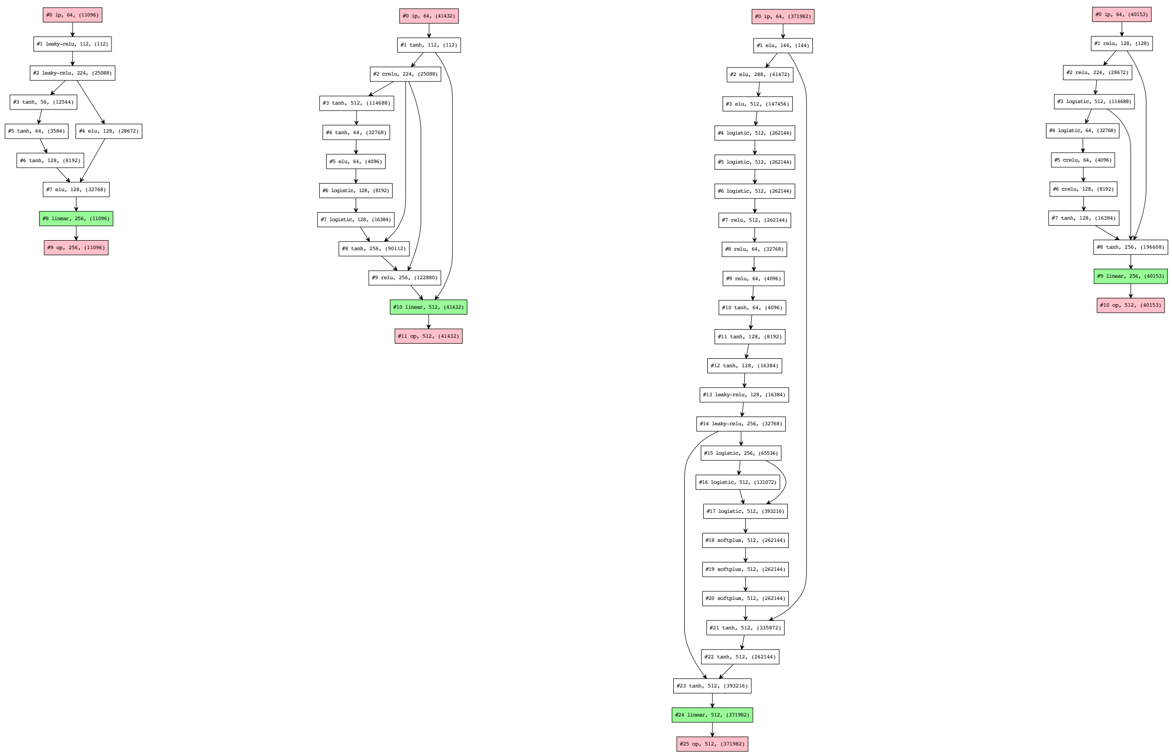


Figure 18: Optimal network architectures found with RAND on Slice data.

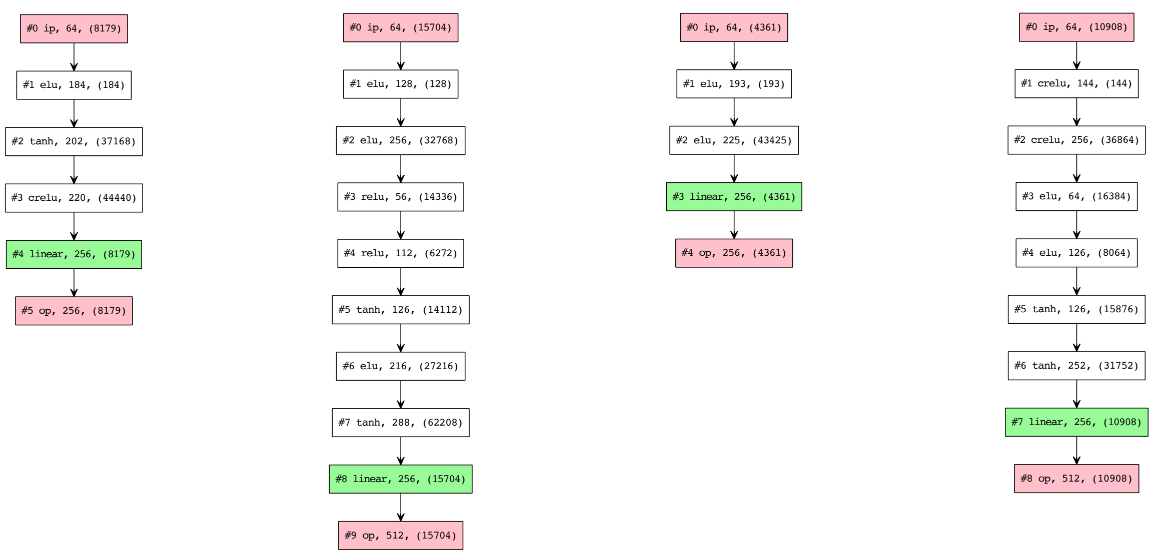


Figure 19: Optimal network architectures found with TreeBO on Slice data.

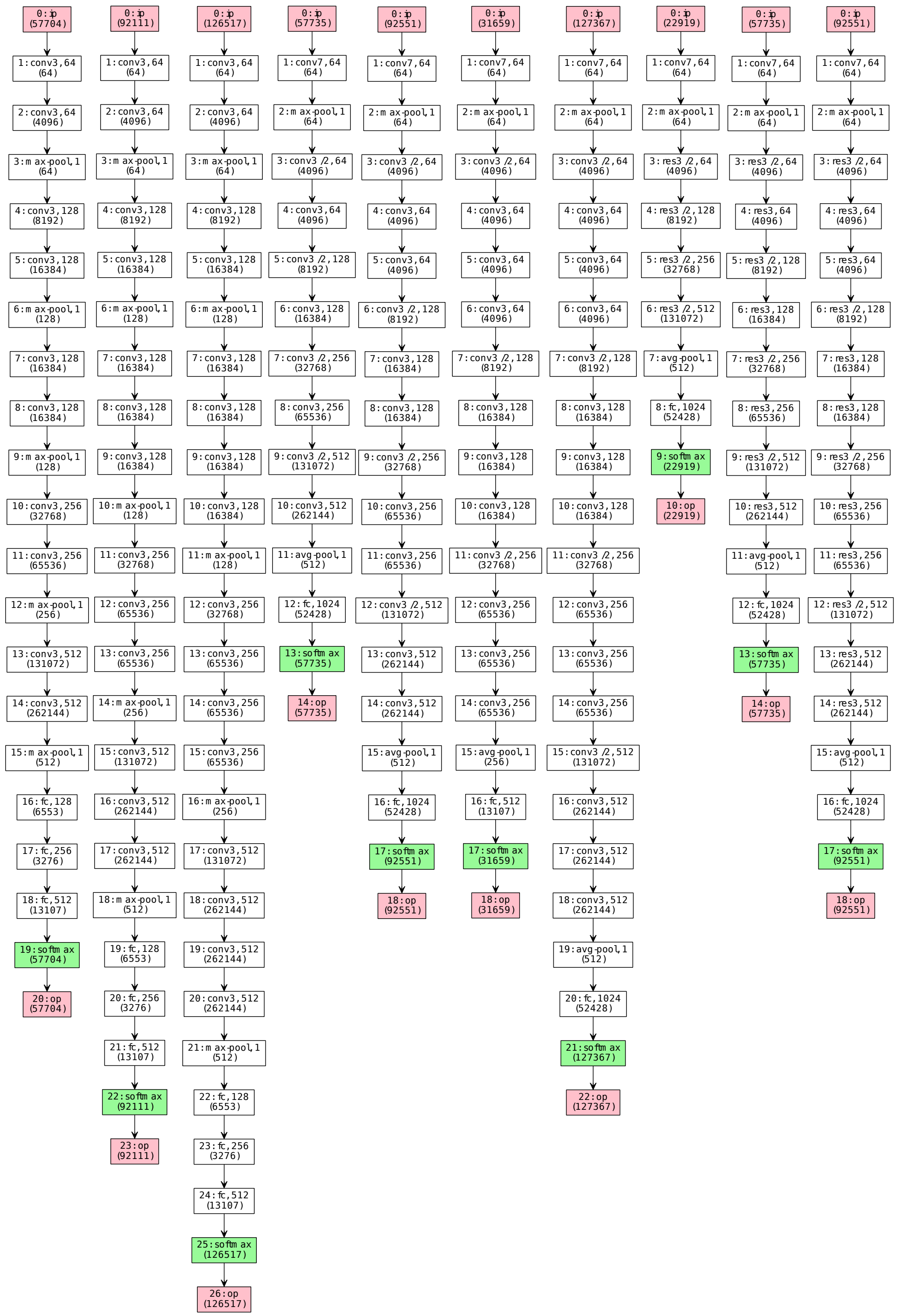


Figure 20: Initial pool of CNN network architectures. The first 3 networks have structure similar to the VGG nets [34] and the remaining have blocked feed forward structures as in He et al. [12].

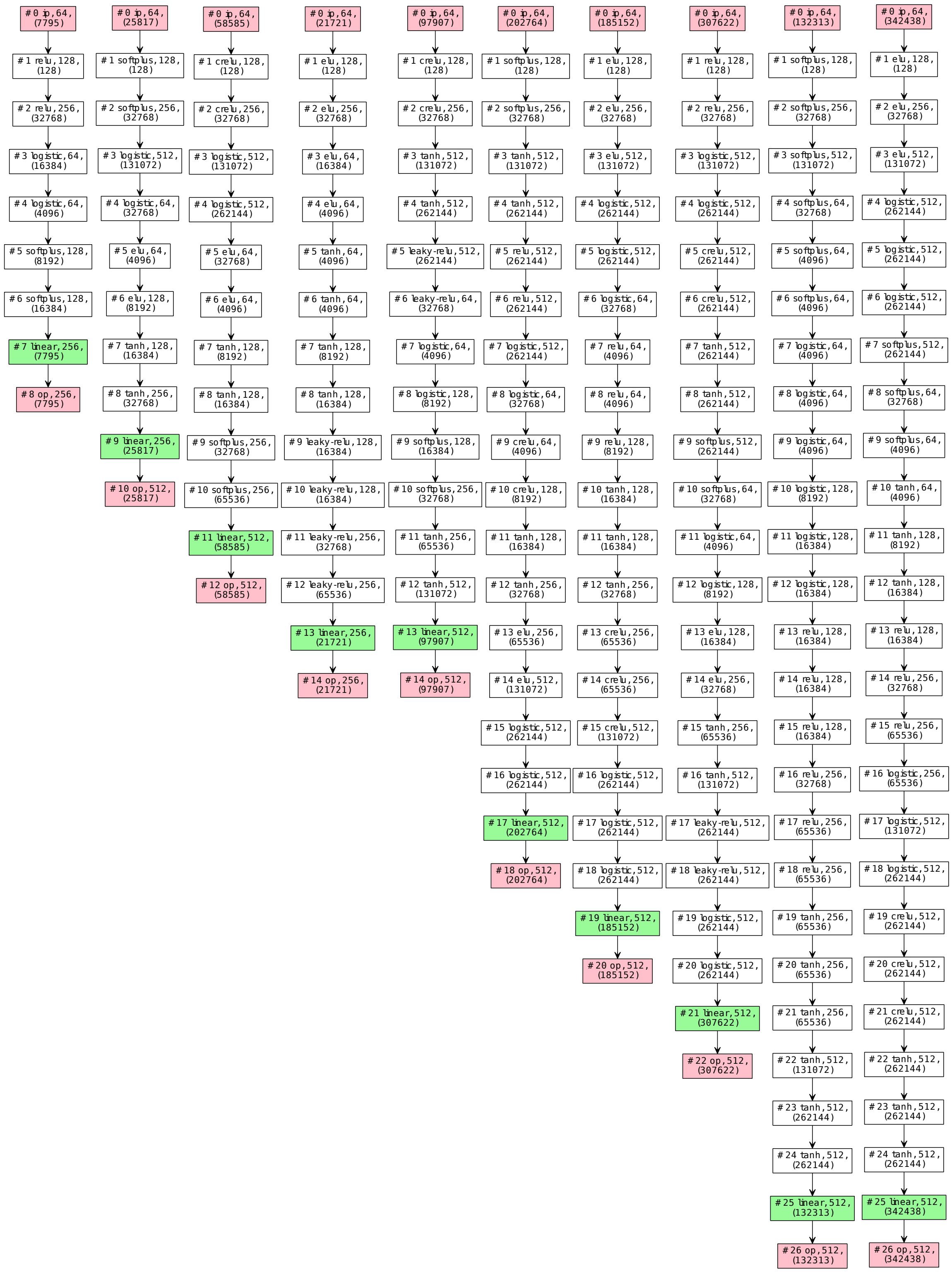


Figure 21: Initial pool of MLP network architectures.