Exploring the Utility of Developer Exhaust

Jian Zhang, Max Lam, Stephanie Wang, Paroma Varma,
Luigi Nardi, Kunle Olukotun, Christopher Ré
Stanford University
{zjian,maxlam,steph17,paroma,lnardi,kunle}@stanford.edu,chrismre@cs.stanford.edu

ABSTRACT

Using machine learning to analyze data often results in *developer exhaust* – code, logs, or metadata that do not define the learning algorithm but are byproducts of the data analytics pipeline. We study how the rich information present in developer exhaust can be used to approximately solve otherwise complex tasks. Specifically, we focus on using log data associated with training deep learning models to perform model search by *predicting* performance metrics for untrained models. Instead of designing a different model for each performance metric, we present two preliminary methods that rely only on information present in logs to predict these characteristics for different architectures. We introduce (i) a nearest neighbor approach with a hand-crafted edit distance metric to compare model architectures and (ii) a more generalizable, end-to-end approach that trains an LSTM using model architectures and associated logs to predict performance metrics of interest. We perform model search optimizing for best validation accuracy, degree of overfitting, and best validation accuracy given a constraint on training time. Our approaches can predict validation accuracy within 1.37% error on average, while the baseline achieves 4.13% by using the performance of a trained model with the closest number of layers. When choosing the best performing model given constraints on training time, our approaches select the top-3 models that overlap with the true top-3 models 82% of the time, while the baseline only achieves this 54% of the time. Our preliminary experiments hold promise for how developer exhaust can help *learn* models that can approximate various complex tasks efficiently.

ACM Reference Format:
Jian Zhang, Max Lam, Stephanie Wang, Paroma Varma, Luigi Nardi, Kunle Olukotun, Christopher Ré. 2018. Exploring the Utility of Developer Exhaust. In *DEEM'18: International Workshop on Data Management for End-to-End Machine Learning, June 15, 2018, Houston, TX, USA.*

1 INTRODUCTION

The recent popularity of data analytics applications has led to a proliferation of tools that can analyze large amounts of data. While these frameworks and libraries are computationally efficient, they have also made procedures like preprocessing data [16, 23], labeling data [20], and analyzing model behavior [1] more *systematic*. For example, Numpy [23] and Pandas [16] help data scientists explore their data, Snorkel [20] and Fonduer [24] allow users to write programmatic labeling functions to efficiently label large amounts of training data, and TensorBoard in Tensorflow [1] lets users visually inspect how well their deep learning models are performing. Since such capabilities are not directly associated with improving the final prediction from these models, they can be viewed as *developer exhaust* – metadata or code associated with the data analysis pipeline that are not part of a learning algorithm but byproducts of developers building models to analyze their data.

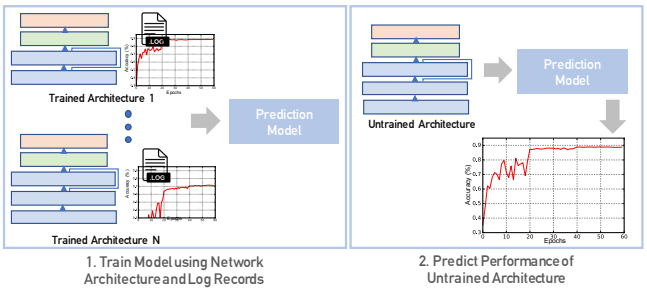

Figure 1: Our method predicts the validation performance of untrained architectures by utilizing information present in logs of pre-trained architectures.

Developer exhausts, such as log data generated while training a model or programmatic functions to preprocess data, are usually semi-structured. Therefore, it is possible to systematically study such forms of exhaust to aid and simplify otherwise complex tasks. For example, our previous work on labeling training data statically analyzed programmatic labeling functions and used the information present in *the code* users wrote to significantly reduce the sample complexity of a subsequent learning model [22]. In this paper, we focus on the logs generated by deep learning models during the training process. In the simplest case, these logs consist of training and validation accuracy recorded every n training epochs, which visualization platforms like TensorBoard use to provide insight into how a model is performing. We explore how information present in logs helps us build generalizable models that can approximate complex tasks like model search and go beyond optimizing for validation accuracy without additional user effort.

Model search, or selecting the correct model architecture, has traditionally been used to select the best performing model for tasks like image recognition and natural language processing [2, 3, 6, 15, 21, 25]. These methods train candidate architectures to convergence to find the best model, therefore taking up to several hundreds of hours to find the best model architecture [21, 25]. We explore how we can *predict* the performance of different model

DEEM'18, June 15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-5828-6/18/06…$15.00
https://doi.org/10.1145/3209889.3209895

architectures by using the information in logs of previously trained models. As a preliminary study, we present two methods to predict validation performance of various model architectures in the context of convolutional neural networks; moreover, we show how these methods can be easily adapted to predict other characteristics, like performance given training time constraints and optimizing for models that overfit the least.

Our first approach to utilize log data is inspired by methods like Paleo [19], which can model the computational performance of neural network model architectures. However, instead of modeling each layer individually, we rely on an edit distance-based, model-space-specific featurization to measure the similarity of model architectures and select nearest neighbors of a model. Our second approach aims at an end-to-end system performance prediction system that does not rely on manual featurization. Inspired by the recent trend in model performance prediction [5, 12], we design a model similar to Peephole [4] and ENAS [18], training a long short term memory (LSTM) model [9] to predict model performance. However, instead of *training* different model architectures on the fly, we simply input logs and architectures of pre-trained models to the LSTM to approximate the performance of untrained models.

Besides predicting the validation accuracy, we also demonstrate how other information in developer exhaust can be easily incorporated to go beyond traditional model search for predicting best performance. As a first step, we show how *without training or building new models*, we can adjust our LSTM to predict overfitting. We extend our approach to predict which models perform the best under training time constraints. We hope this modeling approach can generalize to predicting other computational costs like memory and network usage by utilizing system logs and *learning* model behavior instead of modeling it explicitly.

Our preliminary results show the nearest neighbor and LSTM approaches can approximate the best validation accuracy achieved by an untrained model with 1.37% and 1.54% error in accuracy, respectively; the baseline, which approximates the performance of an untrained model with a trained model with the closest number of layers, achieves an average error of 4.13%. We also apply our approach to select top performing models under training time constraints; in 82% of the cases, the top-3 models predicted with our nearest neighbor approach overlap with the true top-3 models, while the baseline can only achieve this 54% of the time.

While our exploration of using developer exhaust for predicting model performance is preliminary, it holds promise to how the seemingly simple information present in logs can aid other complex tasks. Going beyond training and validation accuracy, we could also use system logs to predict other characteristics about neural architectures. Utilizing developer exhaust can also help train comprehensive models that can learn to predict the performance and computational costs of different programs, which do not have to be restricted to neural networks.

2 METHODOLOGY

In our preliminary exploration, we focus on a constrained model space (Section 2.1) and explore how the information present in developer exhaust can help predict a wide variety of characteristics for deep learning models. First, we introduce a similarity metric to compare model architectures and use the nearest-neighbor approach to predict untrained model performance (Section 2.2). To move past hand-crafted metrics, we design a simple LSTM-based end-to-end model that uses model architecture hyperparameters and logs as input (Section 2.3).

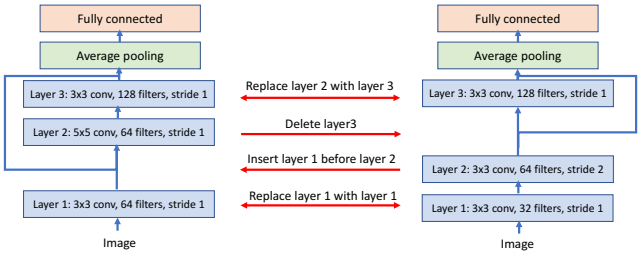

Figure 2: Transformation from one model to another model in our model space. The model $\mathcal{M}_s$ (left) can be transformed to model $\mathcal{M}_t$ (right) using 2 replacement, 1 deletion and 1 insertion operations. The total penalty of this transformation is the sum of penalties induced by the 4 operations.

2.1 Model Space

We focus on a class of models similar to residual networks (ResNets) [7] that are widely used for image classification; we assume that both the models associated with the developer exhausts, and the models we predict performance for are within this model class. These models consist of linearly-connected convolutional layers, each of which is followed by a batch normalization layer [10] and a ReLu layer [17]. Skip connections exist among the convolutional layers and an average pooling and a fully connected layer are placed after the convolutional layers to generate predictions.

Each convolution layer in a model is parameterized by hyparameters like the number of filters, the stride of convolution operations, and the existence of skip connections whose output is combined with the output of the layer. Formally, the i^{th} convolution layer, $f(\boldsymbol{W}_i, \boldsymbol{\theta}_i) : \mathbb{R}^m \mapsto \mathbb{R}^n$, is a function mapping from the m dimensional input vector to n dimensional output vector[1]. $\boldsymbol{W}_i$ is the trainable parameter of the i^{th} layer while $\boldsymbol{\theta}_i$ specifies the hyperparameter of this layer. Therefore, a model with c convolution layers in our model space is fully specified by $\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_c$.

2.2 Nearest Neighbor Approach

Intuitively, we expect similar model architectures to demonstrate similar performance. Motivated by the edit distance used for string comparison [14], we introduce an edit distance metric to measure similarity among model architectures; this metric is the basis of our nearest neighbor approach. The edit distance is the minimum penalty induced by deletion, insertion and replacement operations required to transform the original model to the target model. Since the models in our model space differ only in terms of their convolution layers (we do not take into account the skip connections explicitly in our preliminary investigation), the following transformation operations are sufficient to convert an existing model $\mathcal{M}_s$ to a new model $\mathcal{M}_t$:

[1]The input and output are tensors, we vectorize it for simplicity.

- **Replacement:** We replace a convolution layer $f(\boldsymbol{W}_i^s, \boldsymbol{\theta}_i^s)$ from model $\mathcal{M}_s$ with a convolution layer $f(\boldsymbol{W}_j^t, \boldsymbol{\theta}_j^t)$ from model $\mathcal{M}_t$. The hyperparameters defining these two layers, $\boldsymbol{\theta}_i^s$ and $\boldsymbol{\theta}_j^t$, can be the same or different.
- **Insertion:** We insert a convolution layer $f(\boldsymbol{W}_j^t, \boldsymbol{\theta}_j^t)$ from model $\mathcal{M}_t$ into model $\mathcal{M}_s$.
- **Deletion:** We delete a convolution layer $f(\boldsymbol{W}_i^s, \boldsymbol{\theta}_i^s)$ from model $\mathcal{M}_s$.

We associate a penalty with each of the transformation operations to find the *minimum total penalty, i.e. the edit distance between $\mathcal{M}_s$ and $\mathcal{M}_t$*. Algorithm 1 describes the procedure to compute the edit distance between $\mathcal{M}_s$ and $\mathcal{M}_t$. We assume that models $\mathcal{M}_s$ and $\mathcal{M}_t$ have p and q convolution layers, respectively. The function EditDistance(i, j) returns the edit distance between i^{th}-p^{th} convolution layers in $\mathcal{M}_s$ and j^{th}-q^{th} convolution layers in $\mathcal{M}_t$. For example, transforming the 3-layer $\mathcal{M}_s$ into $\mathcal{M}_t$ shown in Figure 2 requires the following operations: (1) Replace layer 1 in $\mathcal{M}_s$ with layer 1 in $\mathcal{M}_t$; (2) insert layer 2 from $\mathcal{M}_t$ before layer 2 in $\mathcal{M}_s$; (3) delete layer 3 in $\mathcal{M}_s$; (4) replace layer 3 in $\mathcal{M}_s$ with layer 3 in $\mathcal{M}_t$.

This operation sequence induces minimal total penalty, i.e. the edit distance between the two models. This edit distance can be expanded recursively as

$$\text{EditDistance}(1,1) = \text{Penalty}_{\text{replace}}(1,1) + \text{EditDistance}(2,2)$$

where the first term is from replacement operation (1) and the second is the edit distance between the second to third convolution layers in $\mathcal{M}_s$ and the second to third convolution layers in $\mathcal{M}_t$. We utilize this recursive structure and implement the edit distance calculation efficiently using dynamic programming. Note that in Algorithm 1, the penalty functions $\text{Penalty}_{\text{replace}}$, $\text{Penalty}_{\text{insert}}$ and $\text{Penalty}_{\text{delete}}$ depend on $\boldsymbol{\theta}_i^s$ and $\boldsymbol{\theta}_j^t$, the hyperparameters of the layers i and j in $\mathcal{M}_s$ and $\mathcal{M}_t$, respectively. For different layer pairs, they may result in different penalties as discussed in Appendix B.

Algorithm 1 Edit distance between Model Architectures

```
Input: p, q (the number of conv layers in M_s and M_t)
function EditDistance(i, j)
    if i > p and j > q then
        return 0
    else if i > p then
        return EditDistance(i, j + 1) + Penalty_insert(i, j)
    else if j > q then
        return EditDistance(i + 1, j) + Penalty_delete(i, j)
    end if
    D_replace = EditDistance(i + 1, j + 1) + Penalty_replace(i, j)
    D_insert = EditDistance(i, j + 1) + Penalty_insert(i, j)
    D_delete = EditDistance(i + 1, j) + Penalty_delete(i, j)
    return min(D_replace, D_insert, D_delete)
end function
```

Given an untrained model architecture in our model space, we search for the K-nearest neighbor models in the existing logs with respect to the edit distance. We perform a weighted average of the validation accuracy curves of the K-nearest neighbor models to predict the validation accuracy curve of the untrained model. To encode our intuition that more similar architectures have more similar performance, we use the inverse of edit distance as the weights. We evaluate the nearest neighbor approaches in Section 3.

2.3 LSTM-Based End-to-End Approach

To study how developer exhaust can be utilized without the need for handcrafted penalty functions, we present a more general method that can take as input the log data and the model architecture and output the validation metric at the required training epoch. In deep learning, model architectures can be described as computational graphs, and these can be serialized into sequential representations, e.g. using topological ordering of the nodes in the graphs. As a popular modeling tool for sequential representations, LSTM networks are a natural fit for encoding the model architectures without architecture featurization.

In our end-to-end approach, we design a LSTM-based regressor to predict performance characteristics of an untrained model. We first use a distributional embedding to encode each node in the serialized computational graphs. This sequence of node embedding representations are piped into a single-layer LSTM. We extract the last output of the LSTM and pass it through a three-layer multiple layer perceptron (MLP) [8]. We use the scalar output of the MLP to perform least square regression of the target values. More specifically, our LSTM-based model uses a 40 dimensional embedding and a 100 dimensional single-layer LSTM to encode the model architecture. The MLP component uses 3 fully connected layers with 100, 10, and 1 dimensional output respectively. Each of these fully connected layers is followed by ReLu activation to enhance the model with non-linearity. For the model space we described in Section 2.1, convolutional layers with different attribute values are encoded using different embedding vectors. In this case, the convolutional layers are linearly connected. Thus, we can feed the representation of the architecture, the sequence of embedding vectors, directly into the LSTM without computation graph serialization.

3 EVALUATION

To evaluate methods in predicting performance of untrained models with existing log information, we first compare our nearest neighbor and LSTM-based approach to two baselines in predicting the best validation accuracy of untrained models. The first baseline predicts the performance of the untrained model using the validation accuracy curve of a randomly selected model from our model space, while the second baseline predicts with the validation accuracy curve using a randomly selected model with the closest number of convolutional layers. By simply changing the input information to our two approaches, we can also predict the degree of overfitting, or the discrepancy of best training and test accuracy. Finally, going beyond accuracy information in the logs, we also utilize the training time of different model architectures from logs to select the models that perform the best under training time constraints.

3.1 Experimental Setup

Model Space. Based on the model space described in Section 2.1, we generate models with 8 to 18 convolution layers with the following properties:

- Number of filters from {24, 36, 48, 64};
- Filter kernel size with width and height sampled from {3, 5, 7};

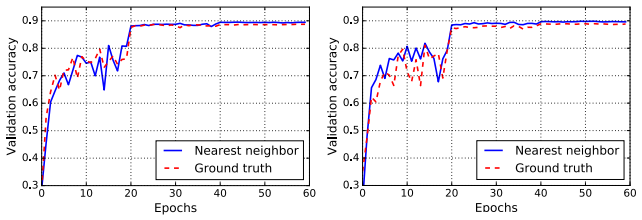

Figure 3: Ground truth vs. predicted validation accuracy using nearest-neighbor approach.

- Stride for convolution operation from $\{1, 2\}$ for layers at 1/3 and 2/3 of the convolution layer chain (rest set to stride 1);
- Number of skip connections from $\{0, 1, 2, 3\}$ to merge with the output of each convolution layer, with the source layer of the skip connection randomly selected.

Evaluation Protocol. We use the CIFAR10 image classification dataset with 40K training and 10K validation samples for our evaluation [13]. We use the training logs associated with 122 randomly selected models from our model space. These models were trained for 60 epochs using standard SGD with momentum 0.9, initial learning rate 0.05, and the learning rate dropping by a factor of 10 after 20 and 40 epochs. To support hyperparameter tuning for our LSTM-based approach, we split the models into training, validation and test set with a ratio of $3:1:1$. We use the training set as the existing repository of logs, while the validation set is for grid search of dropout rate, L2 regularizer and learning rate of Adam optimizer [11]. For the baselines and nearest neighbor approach, we use the union of training and validation set as the existing repository and predict over the test set. We refer to Appendix B for details on penalty function design for the nearest neighbor approach. Note the number of sampled models is substantially smaller than the size of the model space, and we aim to generalize with limited training samples in predicting the performance of untrained models.

3.2 Results

To quantitatively evaluate our nearest neighbor and LSTM approaches, we report the comparison to the baselines for predicting performance charateristics. In our experiments, we bootstrap from our small number of models, and collect metrics over 10 runs with different model set splits. We report the average prediction error with its standard deviation from across all runs in Table 1. We run our nearest neighbor approach with 1, 3, 5 nearest neighbor trained models. As shown in Table 1, our approach with 5 nearest neighbor models performs the best, with a 1.37% average error in predicting the best validation accuracy for a model. It outperforms the two baselines by 3.54% and 2.76%, respectively. Figure 3 shows examples of how the predicted validation accuracy of an untrained model compares to the ground truth validation accuracy. We include a detailed discussion in Appendix A.

Our end-to-end LSTM model achieves 1.54% average error in predicting test accuracy without requiring any hand-crafted comparison metrics. By simply switching the input information, we also evaluate our two methods on predicting the degree of overfitting, the discrepancy between best training and best test accuracy. Our LSTM-based model achieves the lowest average error of 0.67%. These experiments validate that by utilizing existing training logs, our two proposed approaches provide effective ways to predict performance characteristics of untrained models in the same model space. The performance of the LSTM model shows how we could build end-to-end systems to predict performance metrics without hand-crafting a featurization specific to a certain model space.

Going beyond accuracy information in the logs, we also demonstrate how recorded runtime can help predict the best test accuracy of an untrained model under training time budgets. In this task, we first predict the average running time of each epoch using the recorded runtime in the logs and estimate the number of epochs under the budget. The final prediction is achieved by retrieving the best accuracy before that epoch from the predicted validation accuracy curve. We then use the 3-nearest neighbor approach to predict the average epoch time and validation accuracy curve. For each model set split, we rank the test set models with predicted best accuracy under the time budget. With 50 runs using different model set splits, we report how often the predicted set of top-3 model overlaps with the ground truth top-3 models. With a budget of 10k and 20k seconds, the predicted and true top-3 models overlap in 74% and 82% of the runs, respectively; at the same time, the two baselines only achieve 24%, 42% for the 10k budget and 40%, 54% for the 20k budget. This demonstrates how information in logs can help with model search with computational constraints.

Method	Err. best test acc.	Err. degree of overfitting
Baseline 1	4.91% + 1.06%	0.93% + 0.18%
Baseline 2	4.13% + 1.25%	0.96% + 0.22%
1-Nearest Neighbor	1.78% + 0.33%	0.90% + 0.18%
3-Nearest Neighbor	1.39% + 0.30%	0.79% + 0.15%
5-Nearest Neighbor	**1.37% + 0.27%**	0.73% + 0.11%
LSTM	1.54% + 0.43%	**0.67% + 0.07%**

Table 1: The average absolute error for predicting the best validation accuracy and degree of overfitting of an untrained model. We report the average error and standard deviation over 10 runs.

4 CONCLUSION

We discuss two preliminary methods of using information present in logs generated while training deep learning models to predict the validation accuracy of untrained models with different architectures. The nearest-neighbor based approach allows users to see which models were selected to predict performance while the LSTM-based method should generalize better to different model architectures. We are continuing to explore how we can modify the edit distance metric to be robust to more significant differences in model architecture, and the variations in the model the LSTM can handle. Moreover, while we utilized logs that we generated ourselves for the experiments, we want to explore how well it extends to logs that *other developers* generate by looking at training logs from open source models. Using developer exhaust like log data to aid a complex problem like model search holds promise about the utility of information in data analysis byproducts like metadata, and logs.

Acknowledgement. We gratefully acknowledge the support of DARPA under No. FA87501720095, NIH under No. U54EB020405, ONR under No. N000141712266, NSF under No. 1563078, the National Science Foundation (NSF) Graduate Research Fellowship under No. DGE-114747, Joseph W. and Hon Mai Goodman Stanford Graduate Fellowship, and members of the Stanford DAWN project: Intel, Microsoft, Teradata, VMware, Google and NEC. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views, policies, or endorsements, either expressed or implied, of DARPA, DOE, NIH, ONR, or the U.S. Government.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Tal Ben-Nun and Torsten Hoefler. 2018. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. *arXiv preprint arXiv:1802.09941* (2018).

[3] Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. 2017. SMASH: one-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344* (2017).

[4] Boyang Deng, Junjie Yan, and Dahua Lin. 2017. Peephole: Predicting network performance before training. *arXiv preprint arXiv:1712.03351* (2017).

[5] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. 2015. Speeding Up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curves.. In *IJCAI*, Vol. 15. 3460–8.

[6] Thomas Elsken, Jan-Hendrik Metzen, and Frank Hutter. 2017. Simple and efficient architecture search for convolutional neural networks. *arXiv preprint arXiv:1711.04528* (2017).

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

[8] Geoffrey E Hinton. 1987. Learning translation invariant recognition in a massively parallel networks. In *International Conference on Parallel Architectures and Languages Europe*. Springer, 1–13.

[9] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[10] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*. 448–456.

[11] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[12] Aaron Klein, Stefan Falkner, Jost Tobias Springenberg, and Frank Hutter. 2016. Learning curve prediction with Bayesian neural networks. (2016).

[13] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. 2014. The CIFAR-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html* (2014).

[14] Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. 707–710.

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

[16] Wes McKinney et al. 2010. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, Vol. 445. Austin, TX, 51–56.

[17] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*. 807–814.

[18] Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. 2018. Efficient Neural Architecture Search via Parameter Sharing. *arXiv preprint arXiv:1802.03268* (2018).

[19] Hang Qi, Evan R Sparks, and Ameet Talwalkar. 2016. Paleo: A performance model for deep neural networks. (2016).

[20] Alexander Ratner, Stephen H Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. 2017. Snorkel: Rapid training data creation with weak supervision. *arXiv preprint arXiv:1711.10160* (2017).

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

[22] Paroma Varma, Bryan D He, Payal Bajaj, Nishith Khandwala, Imon Banerjee, Daniel Rubin, and Christopher Ré. 2017. Inferring Generative Model Structure with Static Analysis. In *Advances in Neural Information Processing Systems*. 239–249.

[23] Stéfan van der Walt, S Chris Colbert, and Gael Varoquaux. 2011. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering* 13, 2 (2011), 22–30.

[24] Sen Wu, Luke Hsiao, Xiao Cheng, Braden Hancock, Theodoros Rekatsinas, Philip Levis, and Christopher Ré. 2017. Fonduer: Knowledge Base Construction from Richly Formatted Data. *arXiv preprint arXiv:1703.05028* (2017).

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

A VISUALIZATION OF PREDICTED TEST ACCURACY

We first show examples of how well the test performance of a nearest neighbor model from existing logs can predict the performance of an untrained model. As shown in Figure 4(a) and (b), the edit distance metric described in Section 2.2 successfully selects pre-trained models that have performance similar to the untrained model. However, Figure 4(c) shows an example where the metric fails – this is because the untrained model has stride 2 convolution layers while the nearest neighbor trained model does not.

B PENALTY FUNCTIONS FOR EDIT DISTANCE

In Table 2, we specify the value of the penalty functions used in Algorithm 1. For the replacement operation, the discrepancy of each hyperparameter of layer i in one model and layer j in the other model contributes to the penalty. Motivated by the fact that models with and without stride 2 convolution often have very different validation performance (Figure 4(c)), we have a strong penalty for replacement using a layer with different strides. This strong penalty can avoid, in some cases, matching models with and without stride 2 layers as nearest neighbors. To have a balanced penalty across different operations, if layer i and layer j have the same stride, the penalty of insertion and deletion is 2.0, which matches the penalty of replacement for layers with the same stride. Inspired by the intuition that models with different number of layers typically have very different validation performance, we enforce a strong deletion and insertion penalty even when layer i and layer j have the same stride parameter. For models without stride 2, this strong penalty prevents two models with very different number of layers from being nearest neighbors.

$\text{PENALTY}_{\text{replace}}(i, j)$	$\mathbb{1}$(layer i and layer j have different # of filters) $+\mathbb{1}$(layer i and layer j have different kernel size) $+\infty \cdot \mathbb{1}$(layer i and layer j have different stride)
$\text{PENALTY}_{\text{insert}}(i, j)$	$2.0 \cdot \mathbb{1}$(layer i and layer j have different stride) $+\infty \cdot \mathbb{1}$(layer i and layer j have the same strides)
$\text{PENALTY}_{\text{delete}}(i, j)$	$2.0 \cdot \mathbb{1}$(layer i and layer j have different strides) $+\infty \cdot \mathbb{1}$(layer i and layer j have the same strides)

Table 2: The value of penalty functions as a function of the hyperparameters of the convolutional layers.

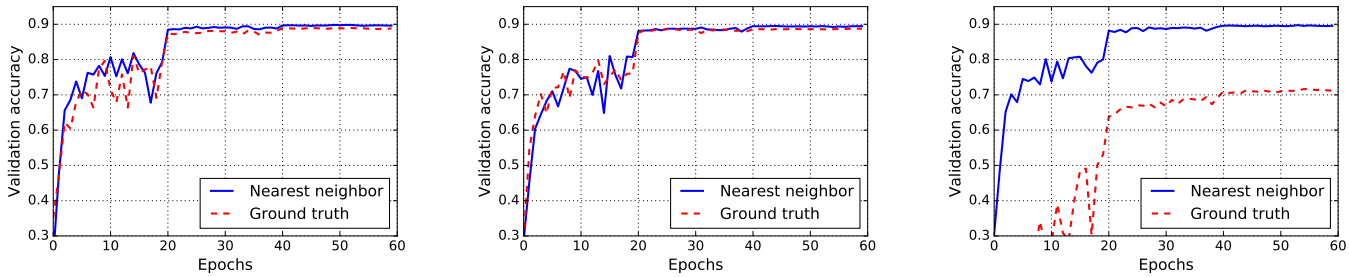

Figure 4: Predicted validation accuracy curve and the ground truth performance of example models. Given a new model, the prediction is the validation accuracy curve of the nearest existing model with respect to the edit distance. For the two example models in (a) and (b), the nearest neighbor model demonstrates similar validation accuracy curve to the examples. In (c), the nearest neighbor model demonstrates different validation accuracy curve to the given example model.