# Manas: Mining Software Repositories to Assist AutoML

Giang Nguyen
gnguyen@iastate.edu
Dept. of Computer Science, Iowa State University
Ames, IA, USA

Md Johirul Islam
jhislam@amazom.com
Amazon Inc
Austin, TX, USA

Rangeet Pan
rangeet@iastate.edu
Dept. of Computer Science, Iowa State University
Ames, IA, USA

Hridesh Rajan
hridesh@iastate.edu
Dept. of Computer Science, Iowa State University
Ames, IA, USA

## ABSTRACT

Today deep learning is widely used for building software. A software engineering problem with deep learning is that finding an appropriate convolutional neural network (CNN) model for the task can be a challenge for developers. Recent work on AutoML, more precisely neural architecture search (NAS), embodied by tools like *Auto-Keras* aims to solve this problem by essentially viewing it as a search problem where the starting point is a default CNN model, and mutation of this CNN model allows exploration of the space of CNN models to find a CNN model that will work best for the problem. These works have had significant success in producing high-accuracy CNN models. There are two problems, however. First, NAS can be very costly, often taking several hours to complete. Second, CNN models produced by NAS can be very complex that makes it harder to understand them and costlier to train them. We propose a novel approach for NAS, where instead of starting from a default CNN model, the initial model is selected from a repository of models extracted from *GitHub*. The intuition being that developers solving a similar problem may have developed a better starting point compared to the default model. We also analyze common layer patterns of CNN models in the wild to understand changes that the developers make to improve their models. Our approach uses commonly occurring changes as mutation operators in NAS. We have extended *Auto-Keras* to implement our approach. Our evaluation using 8 top voted problems from *Kaggle* for tasks including image classification and image regression shows that given the same search time, without loss of accuracy, *Manas* produces models with **42.9%** to **99.6%** fewer number of parameters than *Auto-Keras*' models. Benchmarked on GPU, *Manas*' models train **30.3%** to **641.6%** faster than *Auto-Keras*' models.

## CCS CONCEPTS

• **Software and its engineering** → **Search-based software engineering**; • **Computing methodologies** → **Machine learning**.


*ICSE 2022, May 21âĂŞ29, 2022, Pittsburgh, PA, USA*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## KEYWORDS

Deep Learning, AutoML, Mining Software Repositories (MSR)

**ACM Reference Format:**
Giang Nguyen, Md Johirul Islam, Rangeet Pan, and Hridesh Rajan. 2021. Manas: Mining Software Repositories to Assist AutoML. In *Proceedings of The 44th International Conference on Software Engineering (ICSE 2022).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

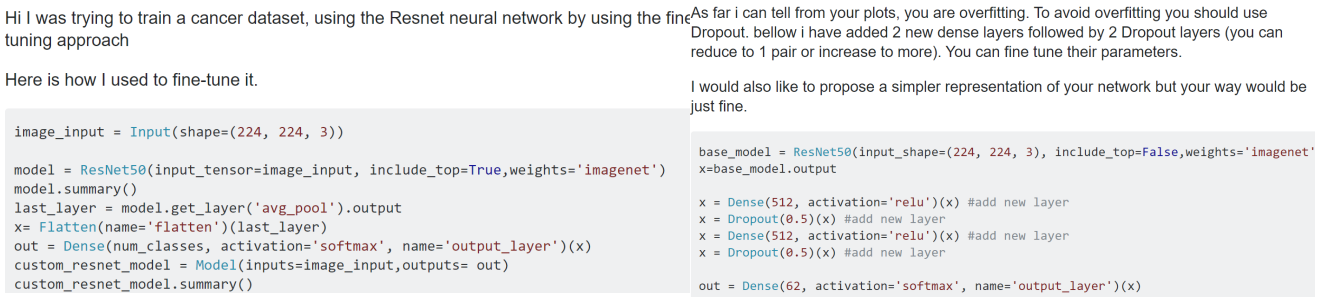
## 1 INTRODUCTION

An increasingly larger number of software systems today are including deep learning. Deep learning uses a convolutional neural network (CNN) model, essentially a graph with weighted edges and nodes that are weighted functions, to convert inputs to the output. As more software systems incorporate deep learning, more software developers have to design and train CNN models. Designing a CNN model is very difficult, and developers often struggle leading to bugs. Model bugs are frequent bugs in CNN programs [24, 25].

Recent work on neural architecture search (NAS) aims to solve this problem [53]. NAS techniques start from a collection of default CNN models and search for a suitable model for the problem. The search space is defined by the collection of default models and a collection of mutation operators that are used to modify CNN models to create new candidates. NAS techniques have been implemented in industrial-strength tools such as *Auto-Keras* [26]. NAS techniques face two problems. First, NAS can be very costly, e.g., *Auto-Keras* takes 8-12 hours on high performance machines to search for models with reasonable accuracy (90+%). The accuracy drops rapidly if the search time is reduced. Second, CNN models produced by NAS can be very complex that makes it harder to understand them for maintenance, and costlier to train and retrain them.

We introduce *Manas* (Mining Assisted Neural Architecture Search) to alleviate the limitations of NAS. The fundamental intuition behind *Manas* is that mining and using the hand-developed models that are available in open-source repositories as default models or starting point of search can help NAS leverage human developer efforts. *Manas* applies this intuition in two ways. First, hand-developed models are mined to search for a better starting point for NAS. Second, the change patterns of the hand-developed models are mined to identify more suitable mutation operators for NAS.

We have realized *Manas* by extending *Auto-Keras*, the state-of-the-art NAS framework. *Auto-Keras* is open source and outperforms state-of-the-art methods like SEAS [13], NASBOT [27] making it a suitable baseline [26]. Some key technical contributions in *Manas* include *model matching*, a technique for matching the problem that




**(a) Problem** **(b) Solution**

**Figure 1: An example from *StackOverflow* showing the necessity to change model architecture**

the developer intends to solve with the hand-developed models mined from repositories, *model adaptation*, a technique for adapting the mined model to the problem context, *model transformation*, a technique for adapting the mined model for further improving metric values, and *training adaptation* that leverages mined parameter values from the repositories to change the optimizer.

To evaluate *Manas*, we use the top-8 problems from diverse domains obtained from Kaggle for machine learning tasks including image classification and image regression. Our evaluation shows that given the same amount of searching time, *Manas* generates simpler neural architectures than *Auto-Keras* without losing accuracy. In terms of the models' size, *Manas*' models have **42.9%** to **99.6%** fewer numbers of parameters than *Auto-Keras*' models. We observed up to **641.6%** faster training speed when training models produced by *Manas* as compared to those produced by *Auto-Keras*.

Our main contributions are the following:

- We have proposed a novel approach for NAS that leverages software repository mining.
- We have proposed methods to identify the suitable models by analyzing user's intents and adapting models.
- We have utilized the common patterns to transform mined models to improve the performance of these models in terms of error rate, MSE, model complexity, and training time.
- We have implemented these ideas in a SOTA NAS framework, *Auto-Keras* [26]. **Our artifact is available here [3].**

The paper is organized as follows: §2 presents a motivating example, §3 presents preliminaries and problem statement, §4 describes the *Manas* approach for NAS, §6 describes the limitations and threats to validity of *Manas*, §7 describes related work, and §8 presents concludes

## 2 MOTIVATION

Deep Learning has received much attention in both academia and industry. Therefore, many deep learning libraries and tools are created for supporting a large number of deep learning developers. Although these libraries and tools make deep learning more accessible, there are still many challenges. One of the challenging problems is constructing an appropriate CNN model architecture, which also has been shown as a frequent bug in CNN programs by Islam *et al.* [24]. For instance, Figure 1a shows a query [1] posted on *StackOverflow* where a developer is unable to find an appropriate CNN model for their intent. In particular, the question discusses the difficulty that the ResNet architecture does not give the result as the developer expected. In response, an expert suggests changing the CNN model. Figure 1b shows the solution of the expert for the question of the developer. The expert suggested that the developer should add dropout layers to minimize overfitting.

Neural architecture search (NAS) aims to solve this problem [53]. NAS takes the training data as an input to automatically define the neural network for that data. Moreover, NAS is able to tune the hyperparameter of the searched neural network. There are both commercial and open-source realizations of NAS. For example, a developer can pay about $20 per hour to use Google's AutoML. *Auto-Keras* is an AutoML system using NAS [26] created as an open-source alternative. *Auto-Keras* returns outstanding results compared with state-of-the-art handmade models on CIFAR10 [28], MNIST [29], and FASHION [49] datasets. *Auto-Keras* is shown to outperform state-of-the-art methods like SEAS [13], NASBOT [27].

NAS has two limitations. First, it can be very costly. For example, *Auto-Keras* consumes 2,300% [41] more GPU computation time compared to using handmade model. Second, NAS often produces very complex models that are hard to understand and time-consuming to train. To illustrate, we used *Auto-Keras* on another dataset *Blood Cell* [34] from *Kaggle*. The model created by *Auto-Keras* for *Blood Cell* problem has more than 2.3 million learnable parameters and more than 7 weight layers. The searched CNN models are constructed based on the architecture of the large default CNN models; thus, the models produced by *Auto-Keras* are often really large. Smaller CNN models train faster and save more energy [22]. Han *et al.* have shown that reducing the number of parameters of deep learning models can reduce the training time by 3× to 4×, and energy comsumption by 3× to 7× [19]. The rest of this work describes our approach *Manas* that addresses these limitations. As an example, for the *Blood Cell* dataset, *Manas* produces a model that decreases the error rate by 47.1%, the model depth by 14.3%, the model width by 87.0%, and increases the training speed by 56.9% compared with *Auto-Keras*'s model. The model produced by *Manas* has 6 layers and 0.3 million parameters (neurons), whereas the model produced by *Auto-Keras* has 7 layers, 2.3 million parameters.

## 3 PRELIMINARIES AND PROBLEM STATEMENT

**Preliminaries (NAS):** We define NAS like it was defined in *Auto-Keras*' papers [26]. Given a search space $\mathcal{S}^*$ and the input data

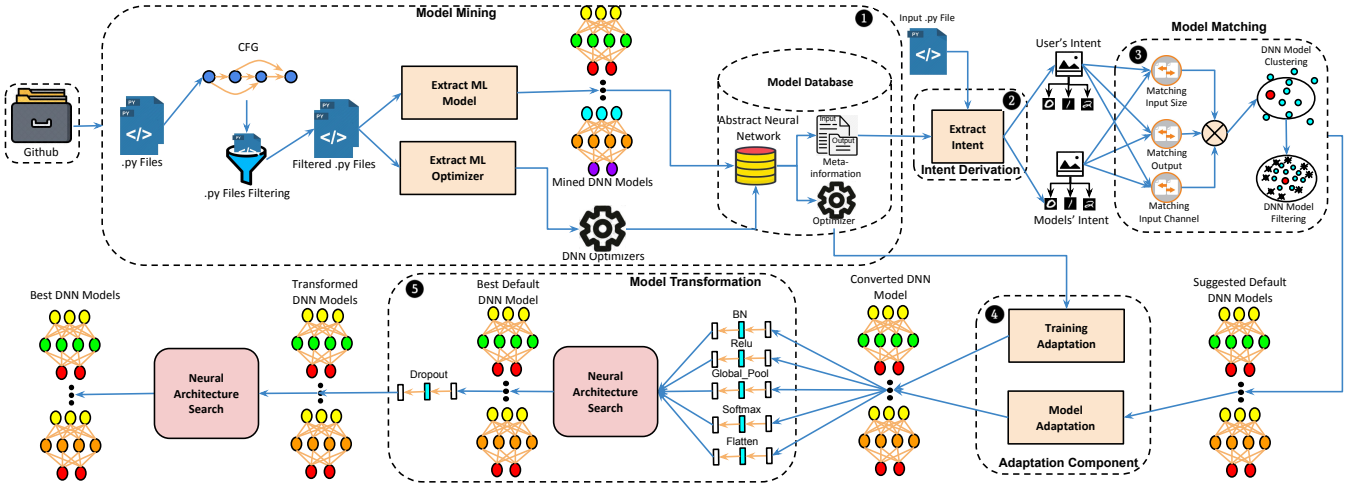



**Figure 2: An Overview of *Manas*. Two inputs are mined models from repository (left-top), and user's initial file (middle-top).**

$\mathcal{D}$ split into $\mathcal{D}_{train}$ and $\mathcal{D}_{val}$, the goal is to find an optimal neural architecture $\mathcal{N}^* \in \mathcal{S}^*$, which achieves the lowest value of the cost function in dataset $\mathcal{D}$. Search space $\mathcal{S}^*$ covers all the neural architectures created from default neural networks.

$$\mathcal{N}^* = \arg\min_{\mathcal{N}^* \in \mathcal{S}^*} Cost(\mathcal{N}'(\theta^*), \mathcal{D}_{val}) \quad (1)$$

$$\theta^* = \arg\min_{\theta} \mathcal{L}(\mathcal{N}'(\theta), \mathcal{D}_{train}) \quad (2)$$

Where *Cost* and $\mathcal{L}$ are the cost function and lost function, $\theta$ is the learned parameter of $\mathcal{N}'$.

**Problem statement:** This work aims to utilize the neural networks from open source repositories to optimize the neural architecture search. We extract the intents of both input dataset and mined neural networks to determine better starting points (initial models) for NAS. Given a neural architecture space $\mathcal{S}$ containing all neural networks, the goal is to find optimal initial architecture $\mathcal{N} \in \mathcal{S}$ for NAS. The optimal initial architectures $\mathcal{N}$ support NAS to find out the optimal neural network $\mathcal{N}^*$ faster. Based on the Equations 1 and 2, we define $\mathcal{N}'$ as follows:

$$\mathcal{N}' = \arg\min_{\mathcal{N} \in \mathcal{S}} \mathcal{L}(\mathcal{N}, \mathcal{D}_{train}) \quad (3)$$

$\mathcal{N}'$ is the neural network with lowest error among the initial architectures.

## 4 MANAS

The Figure 2 shows the overview of *Manas*. *Manas* has five major components that we describe below.

❶ To initialize *Manas* for NAS, the model database must be populated by mining models from open source repositories. Currently, *Manas* collects high quality models from *GitHub* by extracting Python files from *Keras* projects. These projects are selected using certain filtering criteria to ensure code quality. Then, API usage is used to filter Python files to those that contain models. Finally, both the models and the values for optimizer are extracted to store in the model database. This database is constructed once and should be updated periodically as new models are added frequently.

❷ The intent of both users and models are identified to select the good models from the database. The user's intent is identified by analyzing the user's dataset while models are analyzed to infer the models' intent.

❸ Model matching matches the user's intent with the models to obtain a good starting point. It selects the modes which have the closest intent with the user by using the model clustering approach. In case that there are too many models, the model filtering approach will be applied to reduce the number of models.

❹ The selected models and optimizer values are adapted into *Manas* by adaptation system.

❺ The selected models are transformed by model transformation based on related state-of-the-art papers and common layer API patterns of mined models. The transformation can enhance the performance of the models in terms of errors and training speed.

### 4.1 Model Mining

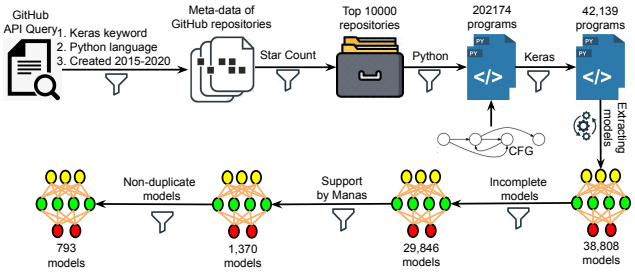



**Figure 3: Model mining process.**

In order to extract CNN models and their optimizers from source code repositories, we build a source code analyzer based on the

control flow graph (CFG) ❶. Figure 3 shows the overview of model mining process.

***GitHub* repositories' meta-data collection:** We collect the meta-data of *GitHub* repositories by using a *GitHub* API query [1]. Meta-data contains basic information of a repository such as authors, repository's name, etc. The query allows us to obtain the meta-data of the *GitHub* repositories with three filtering criteria including Python programming language, containing *Keras* keyword, and created date between 2015-01-01 and 2020-12-31. From the meta-data, we obtain *GitHub* URLs of the top 10,000 repositories with the most star count to ensure the quality of the models [7]. The URLs help us to access the repositories to collect the CNN models.

**Keras programs collection:** We obtain 202,174 Python programs from collected repositories; however, only 42,139 programs use *Keras* API. In particular, we use CFG to analyze their import statements of Python programs to only collect which one import *Keras* API.

**Models extraction:** In this work, since *Manas* only works with *Keras* CNN, we will explain how we extract CNN from *Keras* programs. We use CFG to extract a model from a deep learning program. We manually create a list of function calls used to build neural networks of *Keras* based on *Keras*' documentation [2]. Then, CFG examines every API call to collect the functions contained in the list and their connections. The collected functions represent the layers in the models. The connections between functions represent the layer connections in the models. The reason for using CFG is to collect complete models from programs even if they contain branches. For example, the CFG contains a convolutional block followed by a Dense block in the "if" branch and a skip connection in the "else" branch. If a convolutional block combines with a dense block or a convolutional block combines with a skip connection to be a complete model, we will extract those parts in the branches to collect the complete model. In case that there is a loop containing a part of a model in the CNN programs. If the number of iterations is available to extract that part completely, we will collect it to complete the model. If there is a method call in the program containing a part of the model, we will collect it to complete the model. For the other cases, when the parts in the branches cannot complete the model or cannot be extracted, we will ignore them. For instance, a part of the model in a loop whose numbers of iterations are unavailable cannot be extracted. After this step, we collected 38,808 models from 42,139 *Keras* programs. The number of models is less than the number of *Keras* programs because many programs does not contain models. We assume that a program contains a model if it has at least an API used to build a model.

Rather than mining only neural architectures, we also mine their optimizers that deep learning developers carefully select after spending manual efforts on retraining their models. Then, whenever a model is selected as an initial model, *Manas* trains the model with its optimizer instead of the default one. Optimizers are algorithms deciding how the parameters of the models change. Every optimizer has strengths and weaknesses; thus, it is necessary to choose a suitable optimizer. While models are the decisive factor to the performance of *Manas*, we note that it is wasteful if we cannot fully use these models. In other words, good models with wrong optimizers cannot produce a good performance. To obtain the optimizer, we use the same process of extracting CNN models by creating a list of functions related to the optimizer. After that, CFG analysis is used to obtain the API call, which contains the optimizer.

**Incomplete models detection:** A complete model includes an input layer, hidden layers, and an output layer [52]. The input layer is the first convolutional layer of the neural network, which is distinguished from the other convolutional layers based on its parameters. In particular, only the API call representing for input layer contains *input_shape* parameter. The output layer is the last linear layer of the CNN. Hidden layers are the layers between the input layer and the output layer, including convolutional layers, activation layers, and fully connected layers (linear layers). Therefore, if an extracted model does not have convolutional layers, activation layers, or linear layers, we will consider that the model is incomplete. By removing incomplete models, there are 29,846 models left.

**Supported by Manas:** As *Manas* currently supports a few kinds of layers that are convolutional layer, linear layer, batch normalization layer, concatenate layer, add layer, max pooling layer, dropout layer, Softmax layer, ReLU layer, flatten layer, and global pooling layer, we filter out the models containing unsupported layers. After this step, we have 1,370 models left.

**Model duplication detection:** We obtain 793 models after removing the duplicate ones. If two models have the same abstract neural network, there will be a duplicate model. We will explain in detail about the abstract neural network in Section 4.4.

## 4.2 Intent Derivation

Intent derivation ❷ is used to extract both user's intents and model's intents. The user's intent is extracted by analyzing the training data while we analyze the input layer and output layer of a model to acquire the intent of models.

**User's Intent and Model's Intent Derivation:** We obtain the user's intents by analyzing the dataset and obtain the model's intents by analyzing the input layer and the output layer. Taking an image classification as an example, we obtain the user's intents including input size and output channel from the image data. On the other hand, the model's intents also are the input size and the output channel of the model can be obtained by analyzing the the first convolutional layer and the last linear layer of a CNN model, which are the input layer and the output layer the model, respectively.

Example 1. *The Line 1 is an API call representing for the input layer of a CNN. We extract the value of the argument* input_shape *from the input layer to obtain the input size and the input channel, which are (120, 180) and 3, respectively. The output channel is obtained from the output layer 3. By extracting the first argument's value of the output layer, we obtain the value of the output channel.*

```
Conv2D('input_shape': [3, 120, 180], activation='relu')
...
Dense(10, activation='relu')
```

Since we currently focus on image problems like image classification and image regression, we use the input size, the input channel,

[1] "https://api.github.com/search/repositories?q=keras+language:python+created:yyyy-mm-dd"

and the output channel of image as intent of user and model. First, the input size includes the height and the width extracted from input data and models. Second, the input channel represents the number of primary colors in the image. Third, the output channel is the number of output categories of the data and models.

## 4.3 Model matching

Model matching ❸ is a ranking system used to find good models for a certain problem by using the intents. Instead of using constant default models, model matching finds the suitable models to uses them as default models for NAS.

**Model clustering:** Model clustering uses the intents of both the input dataset and mined neural networks to select appropriate initial architectures $N$ for NAS. First of all, *Manas* clusters the mined models based on the intents of the models and the input dataset. Secondly, in meta-feature (intent) space, our approach identifies closest clusters to the input dataset. Lastly, *Manas* uses all the models in the closest cluster to the dataset as the initial architectures for NAS. Formally, we determine the initial architectures $\mathcal{N}$ for dataset $D$ in Equation 3 as follows:

$$\mathcal{N} = \{C_k \mid n \in C_k, C_k \in C\} \tag{4}$$

$$C = \{\mathcal{G}(\Delta o_k, \Delta s_k) \mid \Delta i_k = 0, k \in [1, |\mathcal{S}|]\} \tag{5}$$

$$dist(n, \mathcal{D}) = \begin{cases} \min \Delta o \text{ if } \min \Delta i = 0 \\ \min \Delta s \text{ if } \min \Delta i = 0, \min \Delta o = 0 \end{cases} \tag{6}$$

In Equation 5, $C$ is a set of clusters of neural network detected by clustering algorithm G-means [18], which uses a statistical test to automatically decide the number of clusters. $\Delta o_k$, $\Delta i_k$, and $\Delta s_k$ are measured as follows:

$$\Delta i_k = |i - i_k| \tag{7}$$

$$\Delta o_k = |o - o_k| \tag{8}$$

$$\Delta s_k = \sqrt{(w - w_k)^2 + (h - h_k)^2} \tag{9}$$

Where $i$, $o$, $w$, and $h$ are the input channel, the number of output classes, the input width, and the input height of the dataset, respectively. Similarly, $i_k$, $o_k$, $w_k$, and $h_k$ are the input channel, the number of output classes, the input width, and the input height of a model k, respectively. The idea behind the clustering equations is that there are two types of the input channel, including 1 and 3. Therefore, we classify the neural networks that have the same input channel first. Then, we use ($\Delta o$, $\Delta s$) as the input of G-means to split the mined models into clusters. After that we identify the closest model $n$ to the input dataset like Equation 6. The closest model is identified based on the priority of $\Delta o$ and $\Delta s$ that $\Delta o$ takes precedence over $\Delta s$. We have tried to run our tool in different orders of priority $\Delta o$ and $\Delta s$; however, this order of priority gives us the best results. The closest cluster to the input dataset is the cluster which contains the closest model. Then, we select all the models in the closest cluster to the input dataset as the initial models $\mathcal{N}$ for NAS shown in Equation 4.

**Model filtering:** If the number of initial models found by the model clustering approach is too large, we use model filtering to filter some models to increase the performance of *Manas*. In the searching process, *Manas* trains all the default neural networks to select the best one. After that NAS is applied to tune the selected model. Thus, with a specific time budget, the more time *Manas* spends on trying the default models, the less time it spends on NAS for model searching. To filtering neural networks, we detect the equivalent architectures. We treat each neural architecture as a graph, whose trainable layers like convolutional layers or dense layers represent vertices, connections between two trainable layers represent edges, and channels of the outgoing trainable layers represent the weights of edges between two vertices. We use Cosine similarity to measure the similarity between two vertices. $M_{ij}$ is an element of the similarity matrix of vertex $v_i$ of a graph $G_A$ and vertex $v_j$ of a graph $G_B$, which is measured as follows:

$$M_{ij} = \frac{\sum_{k=1}^{n} w_{ik} w_{jk}}{\sqrt{\sum_{l=1}^{m} w_{il}^2} \sqrt{\sum_{n=1}^{k} w_{jl}^2}} \tag{10}$$

Suppose that $v_k$ is the common neighbor of vertex $v_i$ and vertex $v_j$, we have $w_{ik}$ and $w_{jk}$ are the weights edges $v_k v_i$ and $v_k v_j$, respectively. We also have $w_{il}$ or $w_{jl}$ are the weights of edges that incident with each of the vertices $v_i$ and $v_j$, respectively.

$$w_{pq} = \begin{cases} \text{weight of the edge between } v_p \text{ and } v_q \text{ in a graph } G \\ 0 \text{ otherwise} \end{cases} \tag{11}$$

We create a similarity matrix $M$ for each pair of graphs and classify equal matrices into the same classes. In this work, we filter the models by only using the neural network whose graphs belong to the class having the highest number of matrices. In other words, we will only choose the most used architecture in the determined initial architecture for NAS. We consider the most frequent architecture to be more suitable than others because many deep learning developers repeatedly use these neural architectures. Non-trainable layers like Dropout or activation are not used as the factors to detect the similarities of the neural architectures because we want to utilize the various usages of these layers to increase the performance of *Manas*. Particularly, different models can have the same graph structure because of the differences in non-trainable layers; therefore, a graph can refer to many different neural architectures.

## 4.4 Adaptation System

The adaptation system ❹ is used to adapt deep learning models and optimizers from the database into *Manas*. We store the extracted CNN models in a database as an abstract neural network (AbNN), which is an abstract representation of the neural network. This representation has the structure as a network where the nodes are API calls, and the connections are the order between API calls. We use this representation to adapt models and their optimizers into *Manas*. From each node, we obtain the name of layer and its parameters, which can be converted into an API call. Figure 4 presents an example of AbNN built from an mined model. Notice that if an activation parameter is implemented as an argument inside a layer, we consider that the activation is a separate layer.

## 4.5 Model Transformation

Even though the mined models are from the top star *GitHub* repositories, they are not perfect. Therefore, in step ❻, we transform the initial architectures by modifying or adding the batch normalization (BN) layer, the flatten layer, activation layers, the global average pooling (GAP) layer, and the dropout layer to optimize NAS in

```
Conv2D(64, kernel_size = (3, 3),activation='relu',
    input_shape=(3, 120, 180))
MaxPooling2D(pool_size=(2,2))
Conv2D(32, activation='relu')
MaxPooling2D(pool_size=(2,2))
Dropout(0.25)
Flatten()
Dense(20, activation='softmax')
SGD(lr=0.01, decay=1e−6)
```

**(a) Extracted model**

```
{'func': 'Conv2D', 'input_shape': [3, 120, 180], 'arg2
    ': 64, 'kernel_size': [3, 3]}
{'func': 'relu'}
{'func': 'MaxPool2d', 'pool_size': [2, 2]}
{'func': 'Conv2D', 'arg1': 64, 'arg2': 32}
{'func': 'relu'}
{'func': 'MaxPool2d', 'pool_size': [2, 2]}
{'func': 'Dropout', 'arg1': 0.25}
{'func': 'Flatten'}
{'func': 'linear', 'arg1': 128, 'arg2': 20}
{'func': 'softmax'}

{'func': 'SGD', 'lr': 0.01, 'decay': 1e−06}
```

**(b) Abstract Neural Network**

**Figure 4: Building an abstraction of neural network from extracted model**

terms of speed, errors, and the number of parameters. We choose these layers to modify the default models because of two reasons. Firstly, we have found many common patterns related to these layers. Secondly, many recent studies have shown the effectiveness of these layers on increasing the performance of deep learning models. We have created a set of pre-defined rules to transform networks based on related state-of-the-art papers and common function calls' patterns from mined models. *Manas* analyzes these architectures to determine whether the network satisfies the pre-defined rules. If these rules are satisfied, the pre-defined model transformations will be applied. The pre-defined model transformations support NAS to ignore transformations offered by our approach and focus on the other transformations. The model transformations do not work for *Auto-Keras*' initial neural architectures because those default models have already included these transformations.

```
(0): Conv2d(3, 32, ...)

(1): Tanh()
(2): MaxPool2d(kernel=2, stride=2, ...)

(3): Conv2d(32, 32, ...)

(4): Tanh()
(5): MaxPool2d(kernel=2, stride=2, ...)

(6): Flatten()
(7): Linear(in=32, out=32, ...)
(8): ReLU()

(9): Linear(in=32, out=2, ...)
(10): Softmax()
```

**(a) Original model**

```
(0): Conv2d(3, 32, ...)
(1): BatchNorm2d(32, ...)
(2): ReLU()
(3): MaxPool2d(kernel=2, stride=2, ...)
(4): Dropout2d(p=0.5)
(5): Conv2d(32, 32, ...)
(6): BatchNorm2d(32, ...)
(7): ReLU()
(8): MaxPool2d(kernel=2, stride=2, ...)
(9): Dropout2d(p=0.5)
(10): GlobalAvgPool2d()
(11): Linear(in=32, out=32, ...)
(12): ReLU()
(13): Dropout2d(p=0.25)
(14): Linear(in=32, out=2, ...)
(15): Softmax()
```

**(b) Transformed model**

**Figure 5: Original CNN model vs transformed CNN model**

**Batch normalization layer constraint:** We add a new batch normalization layer between the convolutional layer and the activation layer to increase training speed [23]. Using BN means that we modify the activations to normalize the input layer to decrease the training time. Many well-known neural architectures like ResNet [20], DenseNet [21], EfficientNet [44] use BN to increase the training speed. BN is also popular in optimizing NAS [12, 14, 53].

Example 2. *According to Figure 5a, the original model has a convolutional layer connects to a ReLU layer, which is an activation function layer. Thus, in Figure 5b, following the batch normalization layer constraint, Manas adds a new BN layer between the convolutional layer and the ReLU layer like following example.*

```
(0): Conv2d(3, 32, ...)
(1): ReLU()
```

**(a) Original model**

```
(0): Conv2d(3, 32, ...)
(1): BatchNorm2d(32, ...)
(2): ReLU()
```

**(b) Transformed model**

**Figure 6: Example of batch normalization layer constraint**

**Global average pooling layer constraint:** We use GAP to reshape the data into the correct format for fully connected layers to prevent overfitting [30]. Moreover, we use the mined models as default models for NAS; thus, the original input size of the initial models and the input size of the dataset can be different, which can cause a shape mismatch bug. However, using GAP can solve this problem since it does not care about the input shape.

Example 3. *According to Figure 5a, the origin model used the flatten layer to pass the feature map through the CNN. Therefore, in Figure 5b, following the constraint about GAP layer, Manas transforms flatten into GAP like following example.*

```
Flatten()
Linear(in=32, out=32, ...)
```

**(a) Original model**

```
GlobalAvgPool2d()
Linear(in=32, out=32, ...)
```

**(b) Transformed model**

**Figure 7: Example of global average pooling layer constraint**

**Activation layer constraint:** We investigate the patterns of usages of activation functions used in the mined model. We have found 3218 hidden layers used in 793 models, where ReLU is used 2946 times, accounting for 94.55%. Therefore, we replace the current activation layers of convolutional layers with ReLU.

Example 4. *According to Figure 5a, the origin model uses Tanh for the convolutional layer. Thus, following the constraint of the activation layer, Manas transforms Tanh to ReLU like following example like following example.*

```
(3): Conv2d(32, 32, ...)
(4): Tanh()
```

**(a) Original model**

```
(5): Conv2d(32, 32, ...)
(6): BatchNorm2d(32, ...)
(7): ReLU()
```

**(b) Transformed model**

**Figure 8: Example of activation layer constraint**

**Dropout layer constraint:** We investigate the frequency of dropout [42] layers and their drop rates used in the mined model. Out of 806 times that Dropout is used in hidden layers, the drop rate of 0.25 is used 385 times, which accounts for 47.8%. Out of 753 times that dropout is used in fully connected layers, the drop rate of 0.5 is used 529 times, which accounts for 70.3%. Thus, We add Dropout with a drop rate of 0.25 to the hidden layers and Dropout with a drop rate of 0.5 fully connected layers.

Example 5. *According to Figure 5a, the origin model does not use the dropout layer. Therefore, in Figure 5b, following dropout layer constraint, Manas adds dropout layers into a convolutional layer with*

*0.25 drop rate and a dropout layer into the fully connected layer with 0.5 drop rate like following example.*

```
Conv2d(32, 32, ...)
ReLU()
MaxPool2d(kernel=2, stride=2, ...)
```

**(a) Original model**

```
Conv2d(32, 32, ...)
ReLU()
MaxPool2d(kernel=2, stride=2, ...)
Dropout2d(p=0.5)
```

**(b) Transformed model**

**Figure 9: Example of dropout layer constraint**

All the transformations except Dropout transformation are applied simultaneously to the mined models before searching. After the best model is selected from candidate models, dropout transformation is applied to the best model. Since adding dropout does not always improve the model's errors, we utilize NAS to identify whether using dropout is good or bad.

## 5 EVALUATION

We implement *Manas* by extending *Auto-Keras* [26]. All experiments use Python 3.6, with 16GB GPU Tesla V100. In these experiments, we use 8 datasets for image classification and image regression, which are obtained from *Kaggle* based on the vote count. The efficiency and effectiveness of *Manas* are evaluated in four aspects. Firstly, we evaluate the metric values including error rate and MSE that is lower the better, model complexity, and training speed of *Manas* by comparing it with *Auto-Keras*. Secondly, we evaluate the model matching approach's efficiency and effectiveness. Lastly, we evaluate the efficiency and effectiveness of model transformation and training adaptation approaches.

Since many models for different *Kaggle* datasets have already been published on *GitHub*, it is possible that some *Kaggle* models of the testing dataset have already been included in our mined models. To avoid this problem, for each dataset, we have compared the default model of *Manas* with the top 5 lowest error rate models from *Kaggle*. *Manas* default model is not one of *Kaggle*'s models. Moreover, since most of the models from *Kaggle* are published as Jupyter Notebooks, we only mine the model written as Python files.

To evaluate the performance of our method, we use **8** different image datasets collected from *Kaggle*: *Blood Cell* [34], *Breast Cancer* [35], *Flower* [32], *Intel Image Classification* (IIC) [6], *Malaria* [4], *MNIST: Ham* [31], *Sign Language Digits* (SD) [33], and *Sign Language* (SL) [45]. These dataset are often used for image classification task; however, we treat prediction targets them as numerical values for image regression task. We only use image datasets because *Auto-Keras* only apply NAS for image data. If a dataset is already separated into training data and validation data, we will directly use it. However, for the dataset that does not have separate training data and validation data, we divide it into two subsets, 80% of randomly selected images are used for training and the remaining images for validation.

### 5.1 Results

*5.1.1 RQ1: How efficient is Manas?* To evaluate the efficiency of *Manas*, we run both *Manas* and *Auto-Keras* on 8 datasets for image classification and image regression. We vary the search time from 2 hours to 20 hours, which are described in Figure 10. Table 1 shows the error rate, MSE, depth, number of parameters, and speed of the best models of *Manas* and *Auto-Keras*. By the best model, we mean one that has the lowest error rate or MSE after search timeout. By comparing *Manas* and *Auto-Keras*, two conclusions can be drawn.

First of all, Table 1 shows that *Manas* produces models, which has lower error rate or MSE compared with *Auto-Keras*' models. The lower errors of *Manas* compared with *Auto-Keras* indicates that using mined models as the starting points for NAS can produce better models. Notably, *Manas* outperforms *Auto-Keras* by achieving **17.6%** lower error rate on average. For some problems like *IIC* and *MNIST: HAM* for image classification task, the decrease of the error rate of *Manas* compared with *Auto-Keras* is small because the models created by *Manas* and *Auto-Keras* have reached the limit of error rate. Therefore, a small decrease in error rate is a major improvement. We use the error rate as the main evaluation metric to clearly point out this improvement of *Manas* to *Auto-Keras*.

Secondly, *Manas* achieves lower errors with less complicated models compared to *Auto-Keras*. Using mined models on NAS significantly decreases the complexity of the produced models. On average, models generated by *Manas* are **75.7%** less deep and **93.0%** less wide compared to the models generated by *Auto-Keras* for image classification task. Similarly, *Manas* creates models with **74.4%** less deep and **89.1%** less wide compare to *Auto-Keras*'s models for image regression task. Simpler DNN models train faster and save more energy [22] than the complex ones. Han *et al.* have shown that reducing the number of parameters of deep learning models can reduce the training time by 3× to 4×, and energy comsumption by 3× to 7× [19]. Table 1 also shows that on average, *Manas*' models run faster than *Auto-Keras*' model **247.1%** and **66.3%** in image classification and image regression, respectively.

*5.1.2 RQ2: How efficient are model transformation and optimizers modification?* We create an ablation study to observe the efficiency of the model transformation and modifying optimizers via mining separately.

**Ablation Study:**

- *Original Manas* (OM) represents mined models + no transformation + no optimizer + NAS.
- *Transformed Manas* (TM) represents mined models + transformations + no optimizer + NAS.
- *Manas* (MN) represents for mined models + transformations + optimizers + NAS.
- *Auto-Keras* (AK) represents NAS.

We observe the error values of *Original Manas*, *Transformed Manas*, *Manas*, and *Auto-Keras* by executing them on 8 datasets for 20 hours for both image classification and image regression. To evaluate the efficiency of the model transformation, we compare *Original Manas* and *Transformed Manas*. We compare *Transformed Manas* and *Manas* to evaluate the efficiency of optimizers modification. To evaluate the combination of all methods, we compare *Manas* and *Auto-Keras*.

Notice that we have applied the model transformation on *Auto-Keras*' models; however, this did not lead to any improvement because those default models use BN layers or Dropout layers appropriately. Figure 10b does not have the *Original Manas* series because the best models of *Original Manas* and *Transformed Manas*

**Table 1: *Manas* Classification & Regression Results**

| Data | Image Classification | | | | | | | | Image Regression | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Error rate (%) | | Depth (layers) | | Param # (million) | | Speed (epoch/min) | | MSE | | Depth (layers) | | Param # (million) | | Speed (epoch/min) | |
| | AK | MN | AK | MN | AK | MN | AK | MN | AK | MN | AK | MN | AK | MN | AK | MN |
| Blood Cell | 18.9 | 10.0 (↓ 47.1%) | 7 | 6 (↓ 14.3%) | 2.3 | 0.3 (↓ 87.0%) | 5.1 | 8.0 (↑ 56.9%) | 0.16 | 0.11 (↓ **31.3 %**) | 21 | 9 (↓ 57.1%) | 11.2 | 0.1 (↓ 99.1%) | 2.7 | 8.4 (↑ 211.1%) |
| Breast Cancer | 6.9 | 6.9 (↓ 0.0%) | 21 | 8 (↓ 61.9%) | 11.2 | 0.5 (↓ 95.5%) | 0.2 | 0.5 (↑ 150%) | 0.05 | 0.06 (↑ 20.0%) | 26 | 7 (↓ 73.1%) | 11.4 | 0.04 (↓**99.6%**) | 0.2 | 0.6 (↑ 200.0%) |
| Flower | 14.6 | 12.8 (↓ 12.3%) | 121 | 11 (↓ **90.9%**) | 7.0 | 1.5 (↑ 78.7%) | 2.7 | 3.8 (↑ 40.7%) | 0.64 | 0.65 (↓ 1.5%) | 21 | 9 (↓ 57.1%) | 11.2 | 0.4 (↓ 96.4%) | 3.3 | 4.3 (↑ 30.3%) |
| IIC | 8.9 | 8.6 (↓ 3.4%) | 27 | 7 (↓ 74.1%) | 15.5 | 0.5 (↓ 96.8%) | 0.7 | 2.6 (↑ 73.1%) | 0.58 | 0.59 (↑ 1.7%) | 122 | 5 (↓ **95.9%**) | 7.0 | 0.3 (↓ 95.7%) | 0.7 | 2.4 (↑ 242.9%) |
| Malaria | 2.3 | 1.6 (↓ 30.4%) | 30 | 7 (↓ 76.7%) | 20.3 | 0.3 (↓ **98.5%**) | 1.7 | 4.3 (↑ 152.9%) | 0.02 | 0.02 (↓ 0%) | 21 | 8 (↓ 61.9%) | 11.2 | 0.4 (↓ 96.4%) | 2.4 | 4.7 (↑ 95.8%) |
| MNIST: Ham | 20.6 | 20.3 (↓ 1.5%) | 7 | 6 (↓ 14.3%) | 0.9 | 1.6 (↑ 77.8%) | 2.2 | 1.4 (↓ 36.4%) | 0.81 | 0.73 (↓ 9.9%) | 26 | 13 (↓ 50.0%) | 11.9 | 0.4 (↓ 96.6%) | 1.1 | 5.2 (↑ **372.7%**) |
| SD | 0.5 | 0.0 (↓ **100.0%**) | 22 | 11 (↓ 50.0%) | 11.3 | 0.7 (↓ 93.8%) | 8.9 | 66.0 (↑**641.6%**) | 0.05 | 0.1 (↑ 100.0%) | 21 | 12 (↓ 42.9%) | 11.2 | 0.4 (↓ 96.4%) | 48.4 | 78.2 (↑ 61.6%) |
| SL | 0.0 | 0.0 (↓ 0.0%) | 22 | 6 (↓ 72.7%) | 11.3 | 0.3 (↓ 97.3%) | 5.5 | 8.0 (↑ 45.5%) | 0 | 0 (↓ 0.0%) | 27 | 10 (↓ 63.0%) | 12.6 | 7.2 (↓ 42.90%) | 5.0 | 2.6 (↓ 48.0%) |
| Avg | 9.1 | 7.5 (↓**17.6%**) | 32.1 | 7.8 (↓**75.7%**) | 10.0 | 0.7 (↓**93.0%**) | 3.4 | 11.8 (↑**247.1%**) | 0.3 | 0.3 (↓**0.0%**) | 35.6 | 9.1 (↓**74.4%**) | 11.0 | 1.2 (↓**89.1%**) | 8.0 | 13.3 (↑ **66.3%**) |

In the table, Avg, AK and MN represent average, *Auto-Keras*, and *Manas*, respectively. In each cell, ↓ and ↑ represent a decrease percentage and an increase percentage, respectively.
In each evaluation metric, the bold value show the best percentage change of *Manas* compared to *Auto-Keras*. -

are the same. In other words, the model already contains all the transformation constraints, so there is no transformation applied to the model. Similarly, Figure 10e does not have the *Transformed Manas* series because the optimizers of the best model of *Transformed Manas* and *Manas* are the same. The selected model's optimizer of this problem is not available; therefore, we use the default optimizer of *Auto-Keras* for this model. From the results are shown in Figure 10, we draw four observations.

First of all, model transformation increases the performance of *Manas* in terms of errors and converge time. Figures 10a, 10c, 10f, 10g show that *Transformed Manas* achieve lower errors than *Original Manas* most of the time. The activation layers, GAP layer, and the dropout layers contribute to the better errors of *Transformed Manas* compared to *Original Manas*. For example, the dropout layers can prevent overfitting, which decreases the errors. The model transformation also helps *Manas* to converge faster. As we can observe Figures 10a, 10c, 10f, 10g, *Original Manas* has higher errors compared to *Transformed Manas* in the first 2 hours of training, which indicate the *Transformed Manas* converge faster than *Original Manas*. *Transformed Manas* has a fast converge speed thanks to BN.

Secondly, mined optimizers help *Manas* to reduce errors and converge faster. In most of problems, the best models of *Manas* have lower errors than the best models of *Transformed Manas*. Moreover, *Manas* can converge easier with the mined optimizers. For instance, the Figures 10d, 10f, 10g shows that *Transformed Manas* has trouble converging. After 8 hours of searching, *Transformed Manas* cannot find out better models while *Manas* succeeds. *Transformed Manas* has trouble in converging since it does not have an suitable optimizer for those problems like *Manas*.

Thirdly, *Auto-Keras* may have lower errors than *Manas* in the first few hours; however, in the last hours, *Manas* often finds out better models than *Auto-Keras*. For example, Figures 10b, 10c, 10d, and 10m show that *Auto-Keras* has the errors at the beginning; however, as time gone we have not seen any improvement in *Auto-Keras* results. The reason for this problem is that *Auto-Keras* starts with very complicated models that take a lot of time to train; therefore, the number of models is searched by *Auto-Keras* are small, which decreases the chance to find out good models of *Auto-Keras*. *Manas* starts with simple models, which trains faster. Therefore, *Manas* may not be off to a good start, but in the end, it still outperforms *Auto-Keras*. The observation shows the benefit of using simple mined models in NAS.

Lastly, Figure 10 indicates that *Manas* obtains lower error rates than *Auto-Keras* in almost different time periods. As we can observe in Figures 10a and 10i, *Manas* always outperforms *Auto-Keras* in terms of error rate in the 20-hours of searching. Figure 10 shows that the longer searching time sometimes gives worse models. During the searching process, the NAS estimates the errors of searched models and selects the best one. However, the estimation may not be accurate, leading to an incorrect choice. This problem of NAS indicates the importance of using a simple model. Simple models can train faster that increases the chance for NAS to search for more models. Since the estimation of NAS can be incorrect; thus, if we increase the number of searched model, we can increase the chance to obtain better models. For example, after 8-hours of searching, NAS goes wrong with *IIC* problem when it produces worse models than before. However, when we keep searching for new models, NAS gradually fixes the problem to obtains a good model.

**Table 2: Efficiency of Model Matching**

| Data | Blood Cell | | Breast Cancer | | Flower | | IIC | | Malaria | | Ham | | SD | | SL | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Task | IC | IM | IC | IM | IC | IM | IC | IM | IC | IM | IC | IM | IC | IM | IC | IM |
| Total | 793 | | | | | | | | | | | | | | | |
| MC | 5 | 5 | 5 | 7 | 18 | 6 | 11 | 46 | 5 | 7 | 6 | 11 | 69 | 4 | 5 | 5 |
| MF | - | - | - | - | - | - | - | 12 | - | - | - | - | 12 | - | - | - |

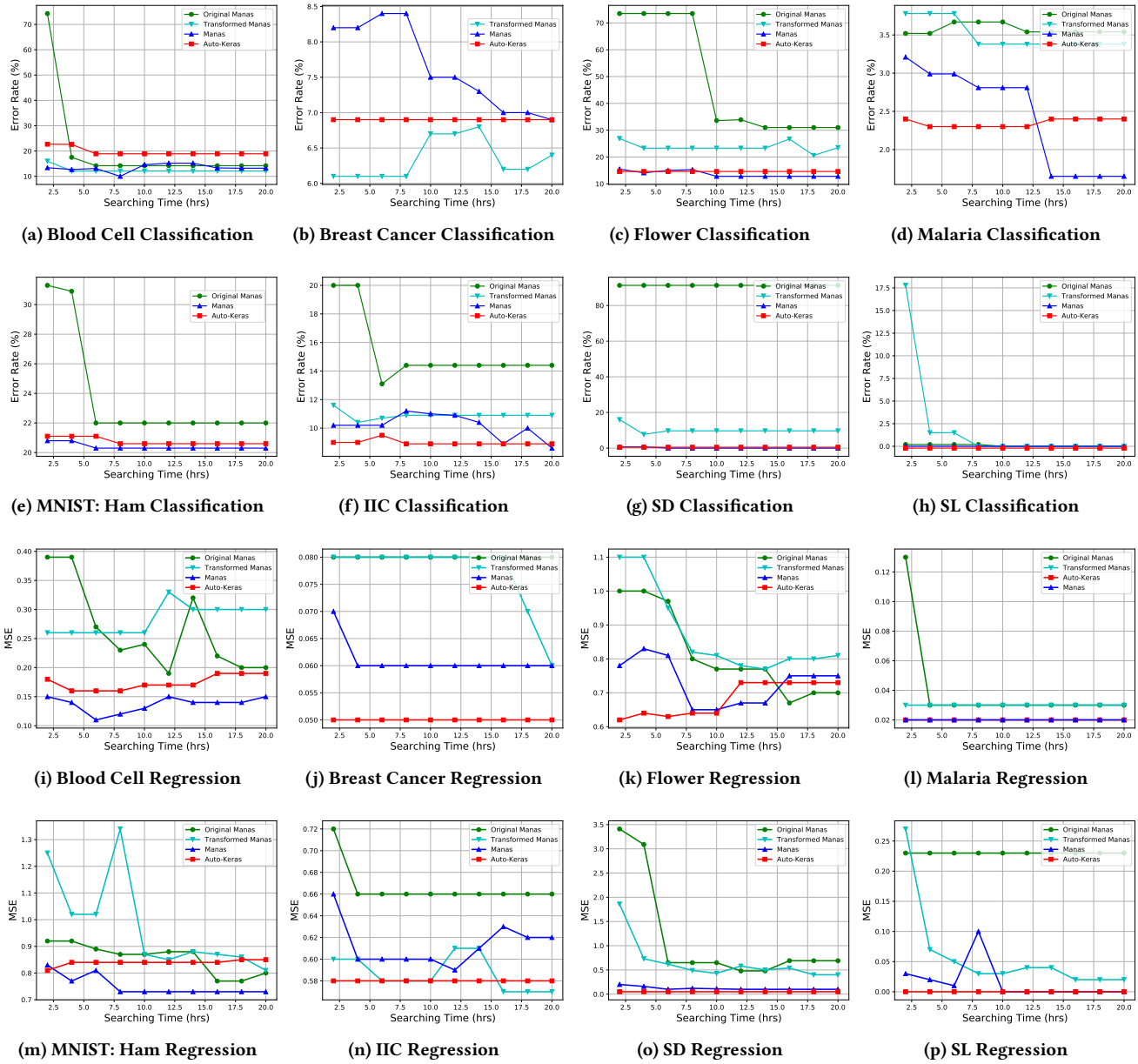
In each cell, IC, IM, MM, and MF represent image classification, image regression, model clustering and model filtering, respectively. The unit of all the data in the table is the number of models.

**Table 3: *Manas* vs *Auto-Keras* for Well-known Problems**

| Data | Error Rate (%) | | Depth (layers) | | Param # (million) | | Speed (epoch/min) | |
|---|---|---|---|---|---|---|---|---|
| | AK | MN | AK | MN | AK | MN | AK | MN |
| CIFAR10 | 7 | 7.7 (↑ 10.0%) | 21 | 10 (↓ 52.4%) | 19.4 | 0.8 (↓ 95.9%) | 1 | 2.9 (↑ 190.0%) |
| Fashion | 5.2 | 11(↑ 111.5%) | 22 | 9 (↓ 59.1%) | 14.7 | 1.8 (↓ 87.8%) | 2.1 | 1.8 (↓ 14.3%) |
| MNIST | 0.5 | 0.7 (↑ 40.0%) | 23 | 6 (↓ 73.9%) | 19.3 | 2.9 (↓ 85.0%) | 1.3 | 3.1 (↑ 138.5%) |
| Avg | 4.2 | 6.5 (↑ 54.8%) | 22 | 8.3 (↓ 62.3%) | 17.8 | 1.8 (↓ 89.9%) | 1.5 | 2.6 (↑ 73.3%) |

In each cell, Avg, AK and MN represent average, *Auto-Keras*, and *Manas*, respectively.

*5.1.3 RQ3: How efficient is model matching?* The goal of model matching is not only selecting good default models for *Manas* but also reducing the number of default models. From Table 1, we observe the efficiency of model matching, when *Manas* can outperform *Auto-Keras* in many different perspectives. Table 2 shows that




(a) Blood Cell Classification (b) Breast Cancer Classification (c) Flower Classification (d) Malaria Classification

(e) MNIST: Ham Classification (f) IIC Classification (g) SD Classification (h) SL Classification

(i) Blood Cell Regression (j) Breast Cancer Regression (k) Flower Regression (l) Malaria Regression

(m) MNIST: Ham Regression (n) IIC Regression (o) SD Regression (p) SL Regression

**Figure 10: Error Rate and MSE of *Auto-Keras*, *Original Manas*, *Transformed Manas*, and *Manas* over time**

model matching, including model clustering and model filtering, can significantly reduce the number of default models for *Manas*. We use all 793 mined models as input for each problem, which may take few days to complete training. By using model matching, we decrease the number of default models by 99% on average. Taking *SD* in image classification as an example, after using DNN clustering, there are remaining 69 default models. Therefore, we use DNN filtering on this dataset to reduce the number of models of *SD* from 69 models to 12 models, which eliminates 82.6% number of the DNN model.

*5.1.4 RQ4: How efficient is Manas for well-known problems?* We also evaluate *Manas* with the well-known datasets like FASHION, MNIST, or CIFAR10 for the image classification task. Table 3 shows the error rate, depth, number of parameters, and speed of the best models of *Manas* and *Auto-Keras*. As can be observed, *Auto-Keras* achieve better error rates on these datasets than *Manas*; however, *Auto-Keras* produces larger models to achieve these error rates while *Manas* uses much smaller models to get close to the error rates of *Auto-Keras* in CIFAR10 and MNIST problems. Particularly, *Manas*' models are 62.3% shorter and 89.9% than *Auto-Keras* âĂŹ

models on average, which increases the training speed of *Manas*' models by 73.3% compared to *Auto-Keras* âĂŹ models. *Auto-Keras* achieves better error rates than *Manas* since *Auto-Keras* uses ResNet and DenseNet as initial models. These neural networks are well-tuned to achieve outstanding results on these datasets, which once again shows that using good initial architecture optimizes NAS.

## 6 LIMITATIONS AND THREATS TO VALIDITY

**Limitations:** In this work, *Manas* directly derives the intent from the image dataset, which is limited to the image classification and image regression problems. As such, *Manas* could be extended with other meta-features that suit other machine learning problems, but in this work, we have focused on image classification and image regression. *Manas* can only work with few kinds of layers since it only uses the layers that *Auto-Keras* supports. This limitation can decrease the performance of *Manas* because if *Manas* supported more layers, it can mine more kinds of models from the software repositories.

**Internal Validity:** We have tried our best to obtain the results of *Manas* and *Auto-Keras* on as many datasets as possible. Because of the time limit, *Manas* is currently evaluated on 8 datasets. All the source code, trained models, datasets, and evaluation data are public for reproduction to mitigate these threats.

**External Validity:** First of all, *Manas* only focuses on image classification/regression problems. We rely on meta-features to find good starting points for NAS; therefore, one possible threat is that meta-features (intents) do not work for other types of problems. However, Auto-Sklearn [16] and Auto-Sklearn 2.0 [15] mitigate this threat by showing that using meta-features can increase the performance of AutoML systems in terms of training speed and accuracy on structured datasets. Secondly, *Manas* only focuses on CNN. Thus, another threat is that the model transformation approach does not work for other types of models. Nevertheless, Cambronero *et al.* [9] propose AMS showing that using unspecified complementary and functionally related API components can improve the performance of AutoML systems for classical models such as Linear Regression or Random Forest. The difference between *Manas* and AMS is that AMS applies these transformations to search space while *Manas* applies these transformations to default models.

## 7 RELATED WORK

**Neural architecture search:** NAS is a technique for automatically finding appropriate neural architectures which can outperform most of the hand-designed neural networks. Specifically, NAS needs a the training dataset as the input to create a powerful neural architecture. There are many different approaches for a NAS system; however, most of them have three main components which are search space, search strategy, and optimization strategy. The search space represents the search boundary of a NAS system limiting what kinds of neural network can be searched and optimized. For instance, Baker *et al.* [5] use the convolutional architecture with pooling, linear transformations as a search space. Around the same time, Zoph *et al.* [53] use a similar search space; however, the authors use more skip connection for the search space. The search strategy is used to search appropriate models in a defined search space. There are many approaches to search models such as reinforcement learning [5, 8, 51, 53, 54] or evolutionary algorithms [40, 43, 50]. This optimization strategy supports NAS to guide the network search process. The optimization strategy evaluates a searched model with training data without training these models. Recently, many methods are proposed to optimize NAS [27, 36, 39]. In our work, *Manas* mines the neural networks from repositories to enhance the power of NAS by supporting it to have a better starting point.

**AutoML:** AutoML is a process for constructing an appropriate model architecture for a specific problem automatically. Many features that AutoML can provide to deep learning users, such as automated data augmentation, automated hyperparameter tuning, or automated model selection. A lot of AutoML systems have been created like Auto-WEKA [46] on top of WEKA [17, 48], Auto-Sklearn [16] on top of Scikit-learn, which support deep learning user to automate tuning hyperparameter and model selection. Some other AutoML systems can support deep learning users to automate optimizing the full ML pipeline. For instance, TPOT [37, 38] uses evolutionary programming to optimize ML software. However, a main disadvantage of these systems are very slow because of the high GPU computation requirement. Recently, *Auto-Keras* is created to handle this problem, which has implemented network morphism [11, 47] to reduce the searching time of NAS. Network morphism is a technique to morph a neural architecture without changing its functionality. Nevertheless, even though *Auto-Keras* apply network morphism technique, it still takes a lot of GPU computation. Our approach uses DNN model mining and common layer patterns to enhance the performance of AutoML system.

**Mining software repositories:** Cambronero *et al.* [10] proposed AL, a system that leverages existing machine learning code from repositories to synthesize final pipelines. AL can generate ML pipelines for a wide range of problems without any manual selection. Cambronero *et al.* [9] also proposed AMS, which automated generates new search space for AutoML systems by utilizing source code repositories. The new search space is created based on an input ML pipeline, which increases the performance of AutoML systems. However, these only operate classical machine learning models, whereas *Manas* works with neural networks.

**Meta-features:** Auto-Sklearn [16] uses 38 meta-features of structured datasets to find a better starting point. Feurer *et al.* [15] proposes Auto-Sklearn 2.0, which reduces the number of meta-features to 3, including the number of data points, the number of features, and the number of classes. The reasons for the reduction are that good meta-features are time-consuming and memory-consuming to generate. We also do not know which meta-features work best for which problem. Unlike Auto-Sklearn and Auto-Sklearn 2.0, *Manas* uses meta-features to find a better starting point for NAS, which works for neural networks. Moreover, *Manas* also proposes the meta-features, which helps NAS find a better starting point for image datasets.

## 8 CONCLUSION

We present *Manas*, a technique for NAS, which uses the mining technique to assist NAS. The key idea of *Manas* is to mine models from repositories to enhance NAS. In particular, we use CNN models mined from software repositories as the default model of

NAS. From a large number of models, we use the model matching approach to find good models for a problem. We also apply some transformations for those models to enhance their performances. With better default models, *Manas* can increase NAS's performance, which leads to better CNN models as search results. Our experiment shows that *Manas* can produce better CNN models in error rate and MSE, model complexity, and training speed than *Auto-Keras*. Future work will involve extending *Manas* to problems beyond those tackled in this paper, such as video classification. The model search can also be further improved to other kinds of layers.

## REFERENCES


[1] Anonymized. 2015. *Resnet network doesn't work as expected.* https://stackoverflow.com/questions/49226447/resnet-network-doesnt-work-as-expected

[2] Anonymized. 2021. Keras documentation. https://keras.io/api/

[3] Anonymized. 2021. Manas artifact. https://github.com/tess100766/Manas

[4] Arunava. 2018. Malaria Cell Images Dataset. https://www.kaggle.com/iarunava/cell-images-for-detecting-malaria

[5] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. 2017. Designing Neural Network Architectures using Reinforcement Learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=S1c2cvqee

[6] Puneet Bansal. 2018. *Intel Image Classification.* https://www.kaggle.com/puneet6060/intel-image-classification

[7] Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Understanding the factors that impact the popularity of GitHub repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 334–344.

[8] Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. 2018. Efficient architecture search by network transformation. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

[9] José P Cambronero, Jürgen Cito, and Martin C Rinard. 2020. Ams: Generating automl search spaces from weak specifications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 763–774.

[10] José P Cambronero and Martin C Rinard. 2019. AL: autogenerating supervised learning programs. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–28.

[11] Tianqi Chen, Ian J. Goodfellow, and Jonathon Shlens. 2016. Net2Net: Accelerating Learning via Knowledge Transfer. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1511.05641

[12] Yukang Chen, Tong Yang, Xiangyu Zhang, GAOFENG MENG, Xinyu Xiao, and Jian Sun. 2019. DetNAS: Backbone Search for Object Detection. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/file/228b25587479f2fc7570428e8bcbabdc-Paper.pdf

[13] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2018. Simple and efficient architecture search for Convolutional Neural Networks. https://openreview.net/forum?id=SySaJ0xCZ

[14] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Efficient Multi-Objective Neural Architecture Search via Lamarckian Evolution. In *International Conference on Learning Representations*. https://openreview.net/forum?id=ByME42AqK7

[15] Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. 2020. Auto-sklearn 2.0: The next generation. *arXiv preprint arXiv:2007.04074* (2020).

[16] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and robust automated machine learning. In *Advances in neural information processing systems*. 2962–2970.

[17] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. 2009. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter* 11, 1 (2009), 10–18.

[18] Greg Hamerly and Charles Elkan. 2004. Learning the k in k-means. In *Advances in neural information processing systems*. 281–288.

[19] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1510.00149

[20] 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.

[21] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708.

[22] Forrest N Iandola, Matthew W Moskewicz, Khalid Ashraf, and Kurt Keutzer. 2016. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2592–2600.

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

[24] Md Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A Comprehensive Study on Deep Learning Bug Characteristics. In *ESEC/FSE'19: The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) (ESEC/FSE 2019)*.

[25] Md Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing Deep Neural Networks: Fix Patterns and Challenges. In *ICSE'20: The 42nd International Conference on Software Engineering* (Seoul, South Korea).

[26] Haifeng Jin, Qingquan Song, and Xia Hu. 2019. Auto-Keras: An Efficient Neural Architecture Search System. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 1946–1956.

[27] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric P Xing. 2018. Neural Architecture Search with Bayesian Optimisation and Optimal Transport. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2018/file/f33ba15effa5c10e873bf3842afb46a6-Paper.pdf

[28] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. *Learning multiple layers of features from tiny images.* Technical Report. Citeseer.

[29] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[30] Min Lin, Qiang Chen, and Shuicheng Yan. 2014. Network In Network. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1312.4400

[31] Kevin Mader. 2018. Skin Cancer MNIST: HAM10000. https://www.kaggle.com/kmader/skin-cancer-mnist-ham10000

[32] Alexander Mamaev. 2017. Flowers Recognition. https://www.kaggle.com/alxmamaev/flowers-recognition

[33] Arda Mavi. 2017. Sign Language Digits Dataset. https://www.kaggle.com/ardamavi/sign-language-digits-dataset

[34] Paul Mooney. 2017. Blood Cell Images. https://www.kaggle.com/paultimothymooney/blood-cells

[35] Paul Mooney. 2017. *Breast Histopathology Images.* https://www.kaggle.com/paultimothymooney/breast-histopathology-images

[36] Niv Nayman, Asaf Noy, Tal Ridnik, Itamar Friedman, Rong Jin, and Lihi Zelnik. 2019. XNAS: Neural Architecture Search with Expert Advice. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/file/00e26af6ac3b1c1c49d7c3d79c60d000-Paper.pdf

[37] Randal S Olson, Nathan Bartley, Ryan J Urbanowicz, and Jason H Moore. 2016. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. ACM, 485–492.

[38] Randal S Olson, Ryan J Urbanowicz, Peter C Andrews, Nicole A Lavender, Jason H Moore, et al. 2016. Automating biomedical data science through tree-based pipeline optimization. In *European Conference on the Applications of Evolutionary Computation*. Springer, 123–137.

[39] Junran Peng, Ming Sun, ZHAO-XIANG ZHANG, Tieniu Tan, and Junjie Yan. 2019. Efficient Neural Architecture Transformation Search in Channel-Level for Object Detection. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/file/3aaa3db6a8983226601cac5dde15a26b-Paper.pdf

[40] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. 2017. Large-scale evolution of image classifiers. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2902–2911.

[41] Adrian Rosebrock. 2019. Auto-Keras and AutoML: A Getting Started Guide. https://www.pyimagesearch.com/2019/01/07/auto-keras-and-automl-a-getting-started-guide

[42] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.

[43] Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao. 2017. A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, 497–504.

[44] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*. PMLR, 6105–6114.

[45] Tecperson. 2017. *Sign Language MNIST*. https://www.kaggle.com/datamunge/sign-language-mnist

[46] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 847–855.

[47] Tao Wei, Changhu Wang, Yong Rui, and Chang Wen Chen. 2016. Network morphism. In *International Conference on Machine Learning*. 564–572.

[48] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. 2016. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.

[49] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747* (2017).

[50] Lingxi Xie and Alan Yuille. 2017. Genetic cnn. In *Proceedings of the IEEE International Conference on Computer Vision*. 1379–1388.

[51] Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. 2018. Practical block-wise neural network architecture generation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2423–2432.

[52] Runjie Zhu, Xinhui Tu, and Jimmy Xiangji Huang. 2020. Chapter seven - Deep learning on information retrieval and its applications. In *Deep Learning for Data Analytics*, Himansu Das, Chittaranjan Pradhan, and Nilanjan Dey (Eds.). Academic Press, 125–153. https://doi.org/10.1016/B978-0-12-819764-6.00008-9

[53] Barret Zoph and Quoc V. Le. 2017. Neural Architecture Search with Reinforcement Learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=r1Ue8Hcxg

[54] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 8697–8710.