Learning Systems 2018:
Lecture **15** – AutoML

Crescat scientia; vita excolatur

Ian Foster and Rick Stevens
Argonne National Laboratory
The University of Chicago

AutoML

- Sweeping through many machine learning methods
  - TPOT
  - Auto-sklearn
  - Data Robot
- Hyper Parameter Optimization
  - Talos
  - CANDLE/rmbo/hyperop
- Neural Architecture Search
  - NAS
  - NASnet
  - PNAS/SMBO
  - ENAS

What are we trying to do?

- The prediction task, where the goal is to find a model that provides a solution for the task
- The hyperparameter optimization task, where the goal is to find the best model (with least effort) for the prediction task
- The hyperparameter optimization task optimization task, where the goal is to find the best approach to best approach to finding the best model for the prediction task

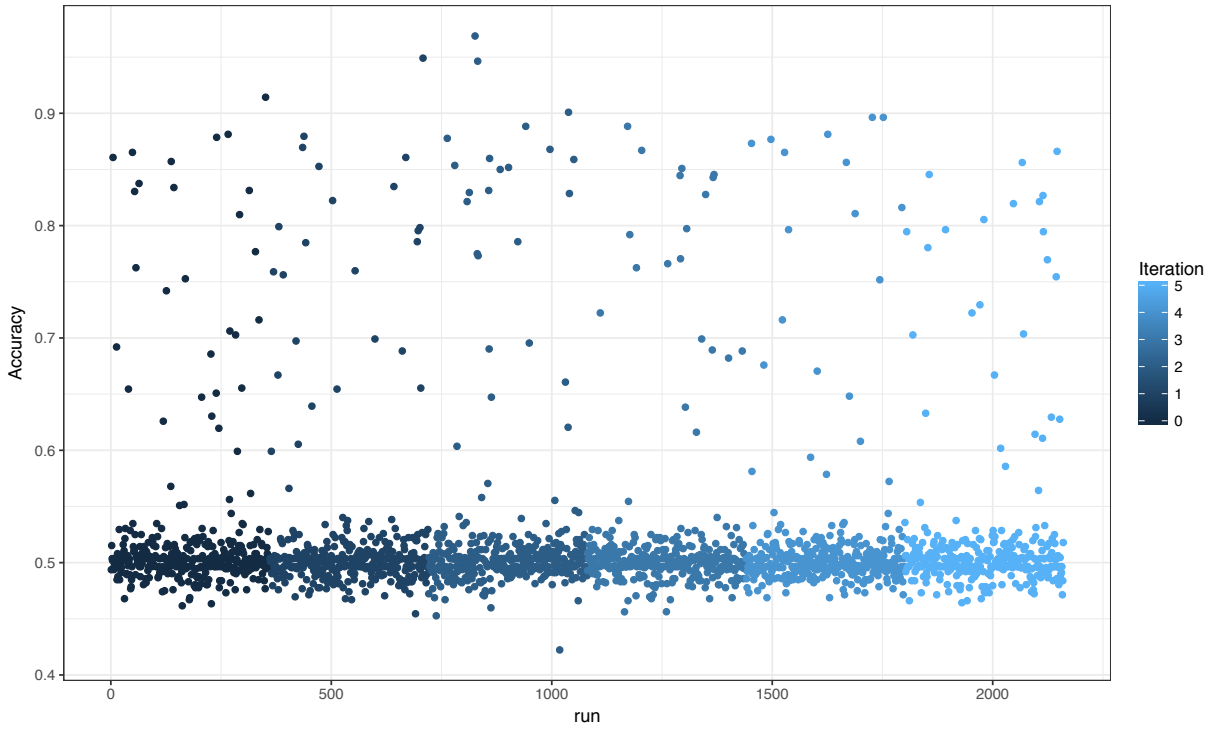
What are we trying to do?

- Prediction Optimum
- Result Entropy

Prediction optimum is where we have a model that is both precise and generalized. Result entropy is where the entropy is as close to zero (minimal) as possible. Result entropy can be understood as a measure of similarity between between all the results within a result set (one round of going through n permutations). The ideal scenario is where the prediction optimum is 1, which is 100% prediction performance and 100% generality, and the resulting entropy is 0. This means that no matter what we do within the hyperparameter space, we only get the perfect result every time.

Example CANDLE Hyperparameter Search

Scatter Plot of Sampling HP Space vs Accuracy
Matched Pair AutoEncoder (NT3, max accuracy 98%)

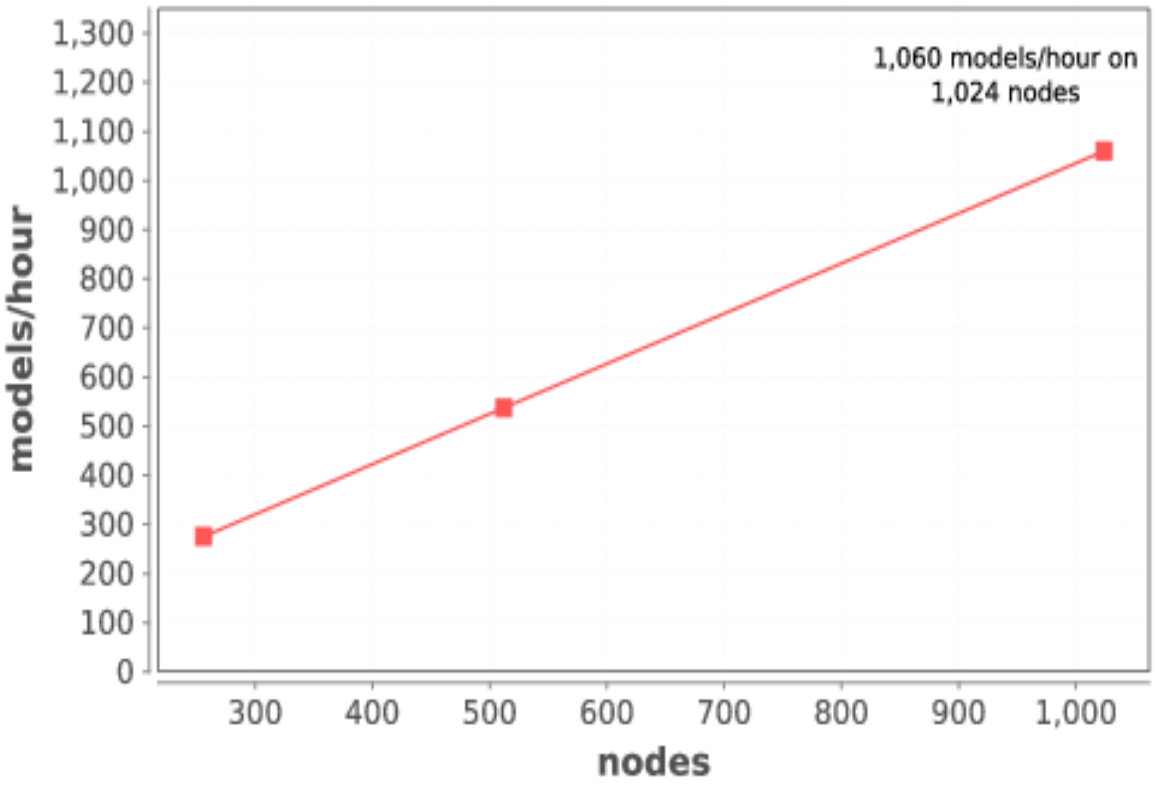



```
makeDiscreteParam("batch_size", values = c(16
, 32, 64, 128, 256, 512)),
  makeIntegerParam("epochs", lower = 5, upper
 = 500),
  makeDiscreteParam("activation", values = c(
"softmax", "elu", "softplus", "softsign", "re
lu", "tanh", "sigmoid", "hard_sigmoid", "line
ar")),
  makeDiscreteParam("dense", values = c("500
100 50", "1000 500 100 50", "2000 1000 500
100 50", "2000 1000 1000 500 100 50", "2000
1000 1000 1000 500 100 50")),
  makeDiscreteParam("optimizer", values = c("
adam", "sgd", "rmsprop", "adagrad", "adadelta
","adamax","nadam")),
  makeNumericParam("drop", lower = 0, upper =
 0.9),
  makeNumericParam("learning_rate", lower = 0
.00001, upper = 0.1),
  makeDiscreteParam("conv", values = c("50 50
50 50 50 1", "25 25 25 25 25 1", "64 32 16 32
64 1", "100 100 100 100 100 1", "32 20 16 32
10 1"))
```

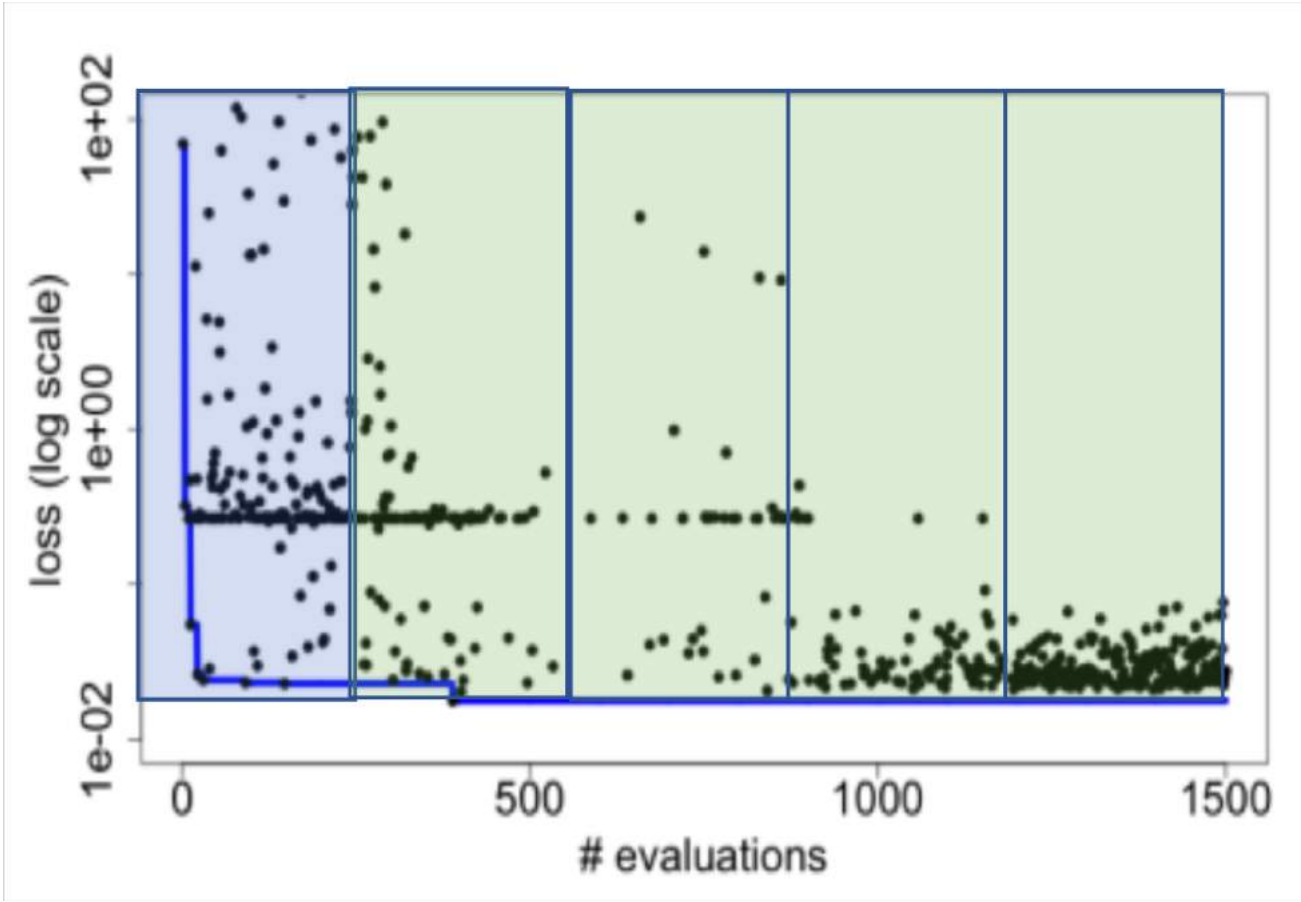
ALCF – Theta, DGX1, OLCF – SummitDev, Titan, NERSC – Cori

Scaling Hyperparameter Search

Scaling models/hour on Titan

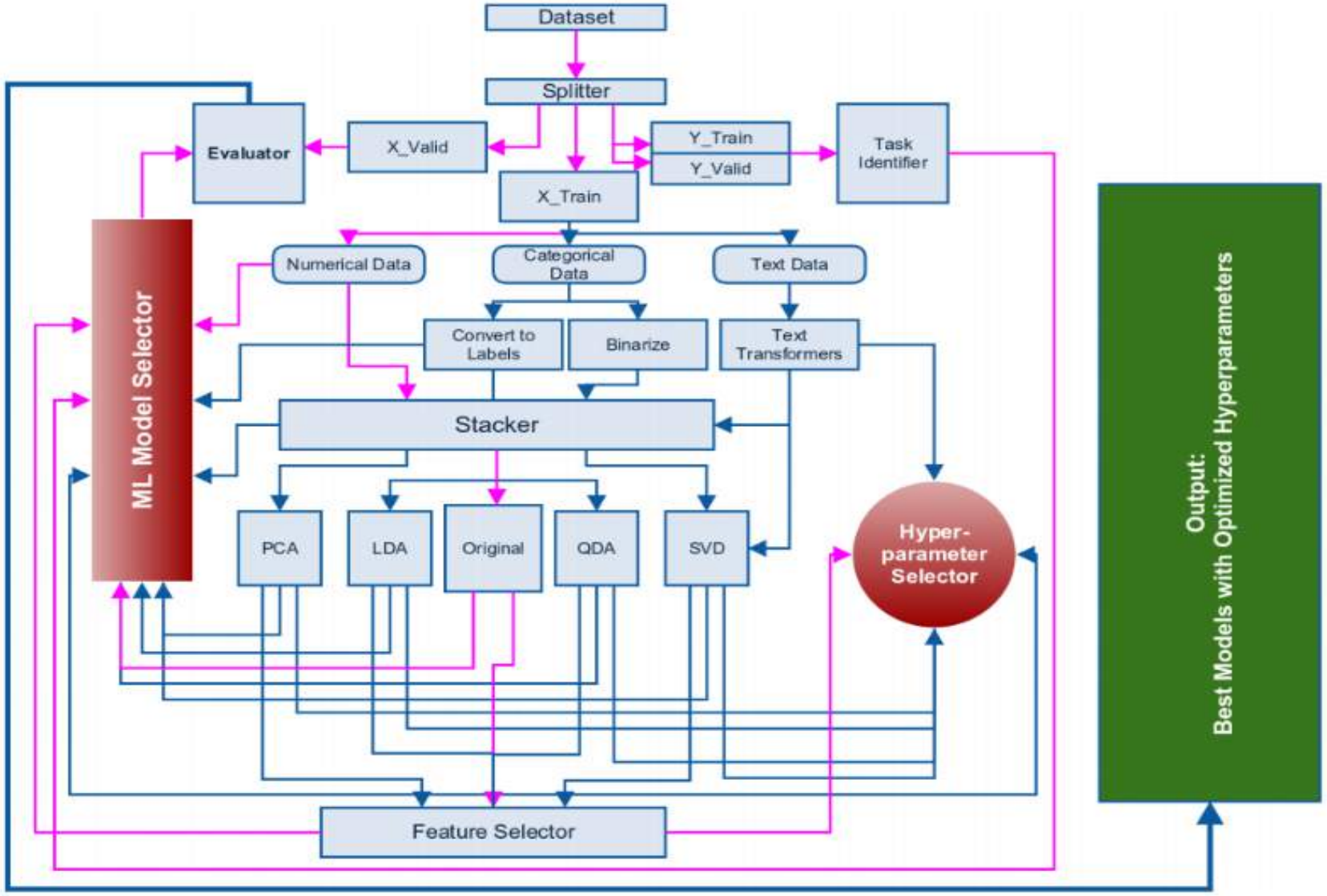


Candle provides easy access to ~ 100 petaflop



Approaches for Traditional ML Methods

Workfow from Kaggle Winner



auto-sklearn

auto-sklearn is an automated machine learning toolkit and a drop-in replacement for a scikit-learn estimator:

```
>>> import autosklearn.classification
>>> cls = autosklearn.classification.AutoSklearnClassifier()
>>> cls.fit(X_train, y_train)
>>> predictions = cls.predict(X_test)
```

auto-sklearn frees a machine learning user from algorithm selection and hyperparameter tuning. It leverages recent advantages in *Bayesian optimization*, *meta-learning* and *ensemble construction*. Learn more about the technology behind *auto-sklearn* by reading our paper published at NIPS 2015 .

Example

```
>>> import autosklearn.classification
>>> import sklearn.model_selection
>>> import sklearn.datasets
>>> import sklearn.metrics
>>> X, y = sklearn.datasets.load_digits(return_X_y=True)
>>> X_train, X_test, y_train, y_test = \
        sklearn.model_selection.train_test_split(X, y, random_state=1)
>>> automl = autosklearn.classification.AutoSklearnClassifier()
>>> automl.fit(X_train, y_train)
>>> y_hat = automl.predict(X_test)
>>> print("Accuracy score", sklearn.metrics.accuracy_score(y_test, y_hat))
```

This will run for one hour and should result in an accuracy above 0.98.

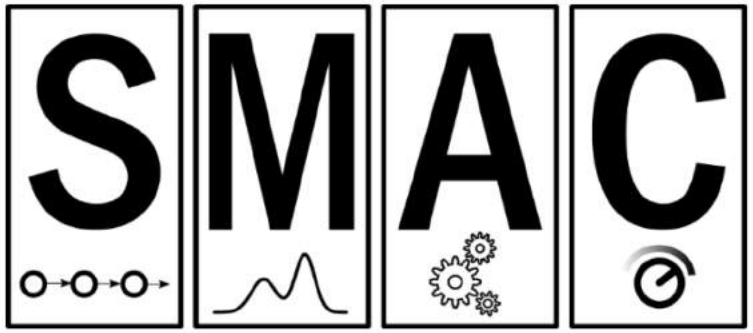
Manual

- Installation
- Manual
- APIs
- Extending auto-sklearn

University of Freiburg

AutoML ...

provides methods and processes to make Machine Learning available for non-Machine Learning experts, to improve efficiency of Machine Learning and to accelerate research on Machine Learning. Machine learning (ML) has achieved considerable successes in recent years and an ever-growing number of disciplines rely on it. However, this success crucially relies on human machine learning experts to perform manual tasks. As the complexity of these tasks is often beyond non-ML-experts, the rapid growth of machine learning applications has created a demand for off-the-shelf machine learning methods that can be used easily and without expert knowledge. We call the resulting research area that targets progressive automation of machine learning AutoML.



SMAC

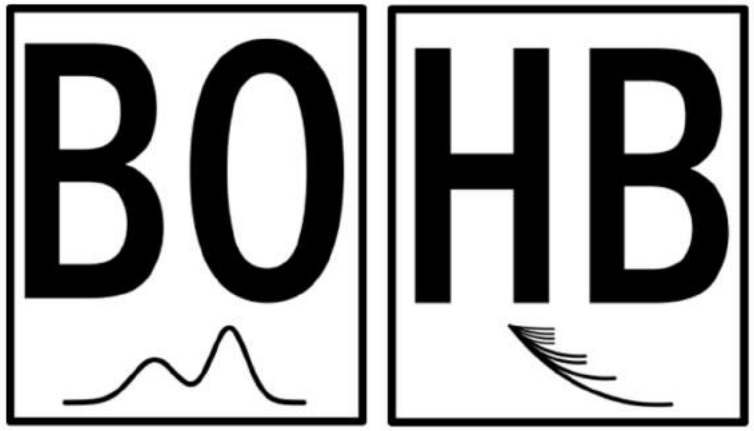
Sequential Model-based Algorithm Configuration is a state-of-the-art tool to optimize the performance of your algorithm by determining a well-performing parameter setting.

Read More »

Auto-Sklearn

Auto-Sklearn is an automated machine learning toolkit to automatically determine a well-performing machine learning pipeline. It is a drop-in replacement for a scikit-learn estimator

Read More »



BOHB

BOHB combines the benefits of both Bayesian Optimization and HyperBand, in order to achieve the best of both worlds: strong anytime performance and fast convergence to optimal configurations.

Read More »

University of Freiburg

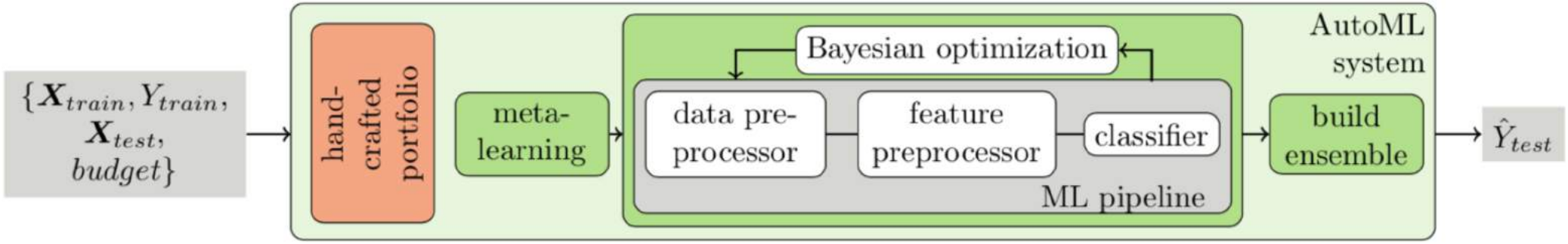


Figure 1: Our pipeline submitted to the ChaLearn Automatic Machine Learning Challenge (2014-2016)

Posh auto-sklearn

University of Freiburg

PoSH Auto-sklearn

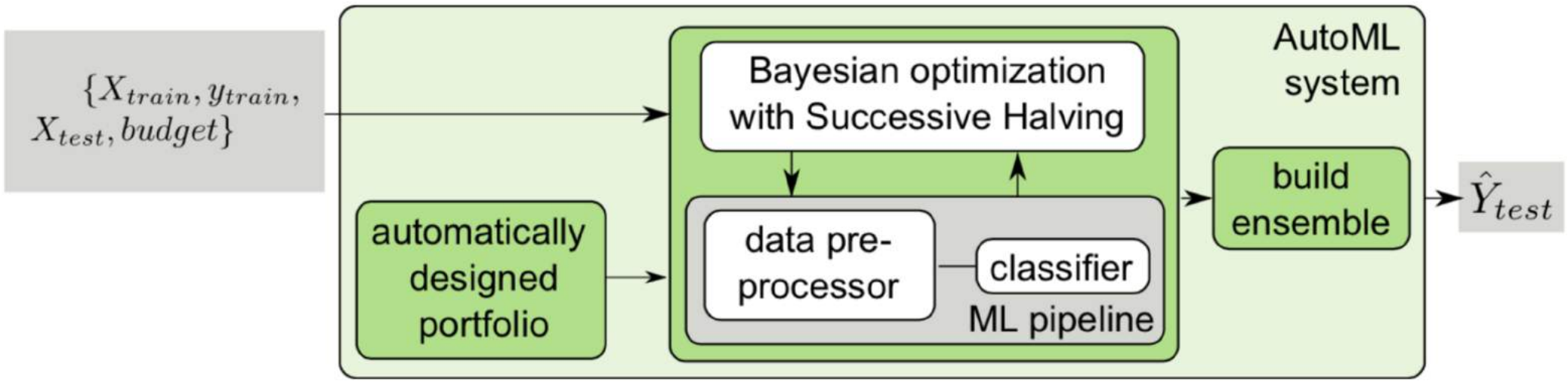


Figure 2: Our pipeline submitted to the AutoML challenge 2018

Posh auto-sklearn

Name	Values	Default
Splitting criterion	{gini, entropy}	gini
Min #samples per split	$[2, 20]$	2
Min #samples per leaf	$[1, 20]$	1
Bootstrap	{True, False}	True
Max % of features	$10^{[0,1]}$	0.5

Table 3: Configuration space of the random forest.

Name	Values	Default	Log
C	$[2^{-5}, 2^{15}]$	1	✓
gamma	$[2^{-15}, 2^{3}]$	0.1	✓
shrinking	True, False	True	

Table 4: Configuration space of the support vector machine.

Posh auto-sklearn

Name	Values	Default	Log
Loss	{hinge, log, modified huber, squared hinge, perceptron}	log	
Penalty	{l1, l2, elastic net}	l2	
α	$[1, 20]$	1	
l1 ratio	$[1e{-}9, 1e{-}1]$	0.15	✓
tolerance	$[1e{-}5, 1e{-}1]$	$1e{-}4$	✓
epsilon	$[1e{-}5, 1e{-}1]$	$1e{-}4$	✓
learning rate schedule	{optimal, invscaling, constant}	invscaling	
eta_0	$[1e{-}7, 1e{-}1]$	$1e{-}2$	✓
power_t	$[1e{-}5, 1]$	0.5	
average	False, True	False	

Table 5: Configuration space of the linear classifier trained by stochastic gradient descent. l1 ratio is only active for the elastic net penalty, epsilon only for the modified huber loss function, and power_t only for the invscaling learning rate schedule.

Posh auto-sklearn

Name	Values	Default	Log
Max depth	[1, 10]	3	
Learning rate	[0.01, 1]	0.1	✓
Booster	GBTree, DART	GBTree	
Subsample	[0.01, 1.0]	1.0	
Min child weight	$[1e^{-10}, 20]$	1	
Sample type	uniform, weighted	uniform	
Normalization type	tree, forest	tree	
Dropout rate	$[1e-10, 1-1e-10]$	0.5	

Table 6: Configuration space of Extreme Gradient Boosting (Chen and Guestrin, 2016). Hyperparameters in the lower half are only active if the *Dropout Additive Regression Trees (DART)*-Booster (Vinayak and Gilad-Bachrach, 2015) is chosen.

Posh auto-sklearn

Name	Values	Default	Log
Imputation strategy	{mean, median, most frequent}	mean	
One Hot Encoding (OHE)	{On, Off}	On	
OHE use minimum fraction	{On, Off}	On	
OHE mimimum fraction	[0.0001, 0.5]	0.01	✓
Rescaling strategy	{Standardize, None, MinMax, Normalize, Quantile Transformer, Percentile MinMax}	Standardize	
# Quantiles	[10, 2000]	1000	
Quantile distribution	{uniform, normal}	uniform	
Lower percentile	[0.001, 0.3]	0.25	
Upper percentile	[0.7, 0.999]	0.75	

Table 7: Configuration space of the preprocessing steps.

Posh auto-sklearn

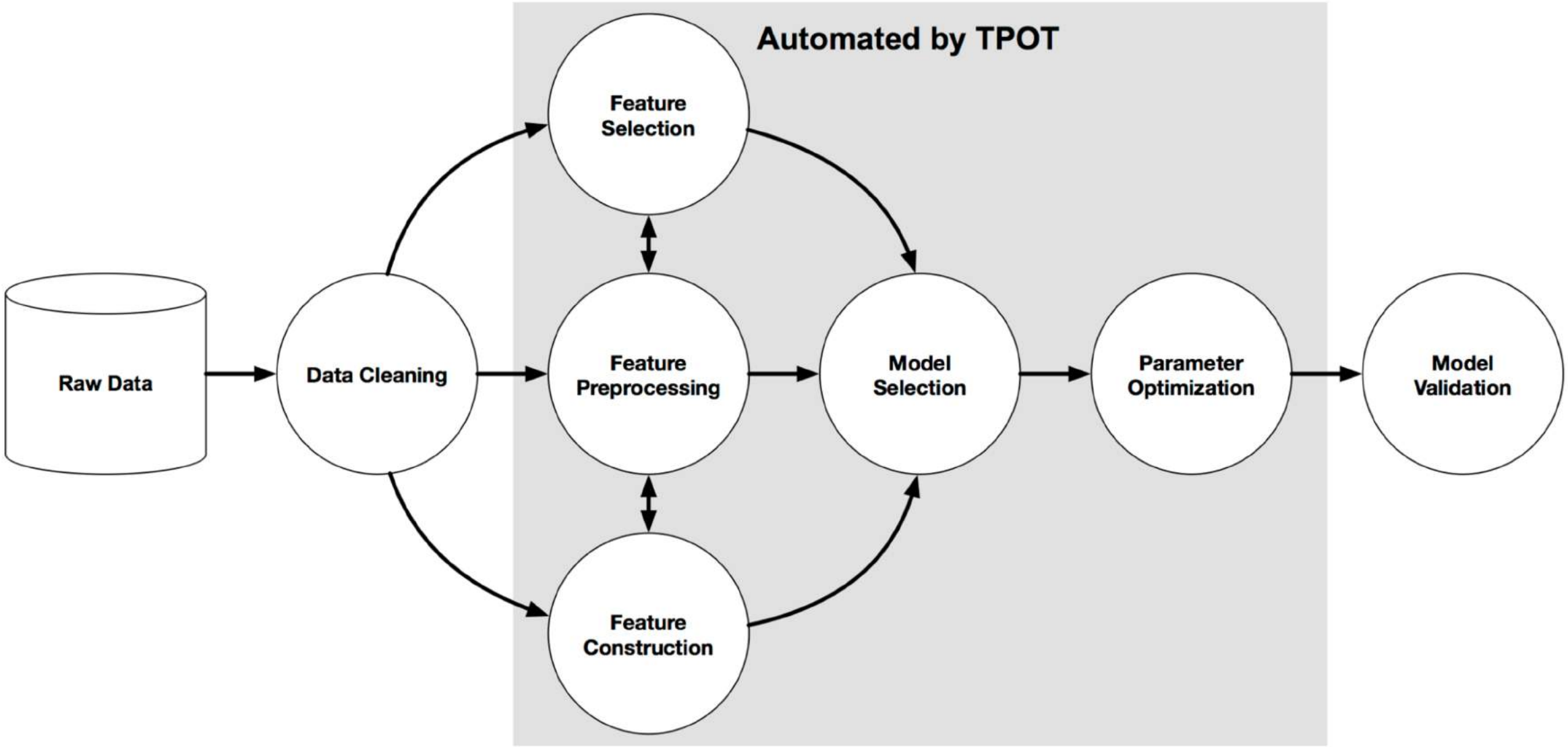
A.4. Additional Measures for Robustness

We implemented the following fallbacks and preprocessing steps in *PoSH Auto-sklearn* which were not relevant for the competition in the end:

- We first saved a dummy predictions consisting of predictions of the majority class to disk. In case something would crashes afterwards, constantly predicting the majority class gives a score of at least 0.0, while not producing a single predictions is punished with a score of -1.0.
- If the dataset had less than 1000 data points, we reverted to cross-validation instead of successive halving.
- For datasets with more than 45.000 data points, we capped the number of training points at 30.000 to retain decent computational complexity. 45.000 was used as a threshold as we used $\frac{2}{3}$ of the data for training only (30.000 data points).
- We prepared a backup solution using Extremely Randomized Trees (Geurts et al., 2006) for the case that no configuration from our portfolio completed in the first iteration (smallest budget) of successive halving.

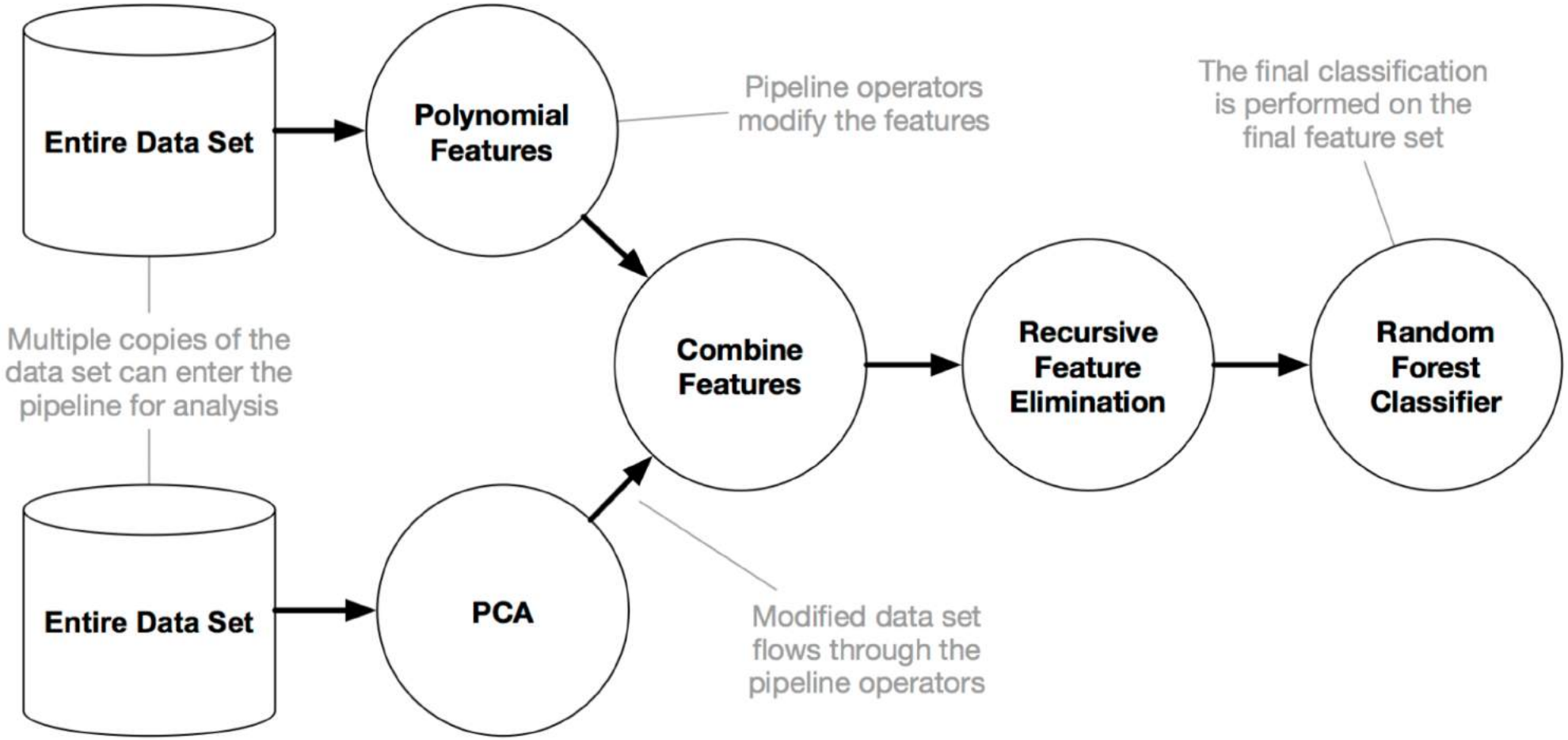
Posh auto-sklearn

TPOT will automate the most tedious part of machine learning by intelligently exploring thousands of possible pipelines to find the best one for your data.



An example Machine Learning pipeline

Once TPOT is finished searching (or you get tired of waiting), it provides you with the Python code for the best pipeline it found so you can tinker with the pipeline from there.



TPOT is built on top of scikit-learn, so all of the code it generates should look familiar... if you're familiar with scikit-learn, anyway.

TPOT is still under active development and we encourage you to check back on this repository regularly for updates.

For further information about TPOT, please see the project documentation.

README.md

Master status: build passing | build passing | health 95% | coverage 96%

Development status: build passing | build passing | health 95% | coverage 96%

Package information: python 2.7 | python 3.6 | license LGPL v3 | pypi package 0.9.5

Consider TPOT your **Data Science Assistant**. TPOT is a Python Automated Machine Learning tool that optimizes machine learning pipelines using genetic programming.

```
from tpot import TPOTClassifier
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

digits = load_digits()
X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target,
                                                    train_size=0.75, test_size=0.25)

tpot = TPOTClassifier(generations=5, population_size=50, verbosity=2, n_jobs=-1)
tpot.fit(X_train, y_train)
```

learning pipelines using genetic programming.

```
from tpot import TPOTClassifier
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

digits = load_digits()
X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target,
                                                    train_size=0.75, test_size=0.25)

tpot = TPOTClassifier(generations=5, population_size=50, verbosity=2, n_jobs=-1)
tpot.fit(X_train, y_train)
```

```
Optimization Progress:  33%|███▍       | 100/300 [01:02<09:07,  2.74s/pipeline]
Generation 1 - Current best internal CV score: 0.9644750872792087
Optimization Progress:  50%|█████      | 150/300 [01:35<05:41,  2.27s/pipeline]
Generation 2 - Current best internal CV score: 0.9681323584103183
Optimization Progress:  67%|██████▋    | 200/300 [01:59<01:59,  1.19s/pipeline]
Generation 3 - Current best internal CV score: 0.9718282518620386
Optimization Progress:  83%|████████▎  | 250/300 [02:23<00:41,  1.21pi
Generation 4 - Current best internal CV score: 0.9756538552070356

Generation 5 - Current best internal CV score: 0.9756538552070356

Best pipeline: KNeighborsClassifier(input_matrix, KNeighborsClassifier__n_neighbors=10, KNeighborsClassifier__p=DEFAU
LT, KNeighborsClassifier__weights=distance)
```

~99% accuracy on MNIST out of the box

```
print(tpot.score(X_test, y_test))
```

```
0.995555555556
```

TPOT will automate the most tedious part of machine learning by intelligently exploring thousands of possible pipelines to find the best one for your data.

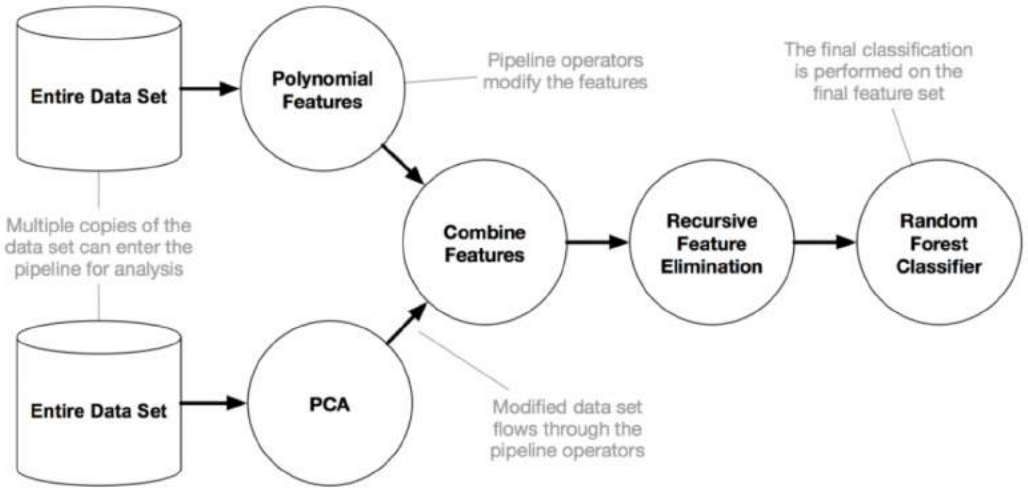
TPOT

The Tree-Based Pipeline Optimization Tool (TPOT) was one of the very first AutoML methods and open-source software packages developed for the data science community. TPOT was developed by Dr. Randal Olson while a postdoctoral student with Dr. Jason H. Moore at the Computational Genetics Laboratory of the University of Pennsylvania and is still being extended and supported by this team.

The goal of TPOT is to automate the building of ML pipelines by combining a flexible expression tree representation of pipelines with stochastic search algorithms such as genetic programming. TPOT makes use of the Python-based scikit-learn library as its ML menu.

Several peer-reviewed papers have been published on TPOT. Our first paper in 2016 won a best paper award at the EvoStar computer science conference. Our second paper in 2016 won a best paper award at the GECCO computer science conference. We showed in a 2017 paper presented at the GECCO conference how TPOT could be adapted to the analysis of big data from genetic studies of common human diseases. This paper was nominated for a best paper award. Please contact us for reprints of these papers and others. These can also be found on arXiv.

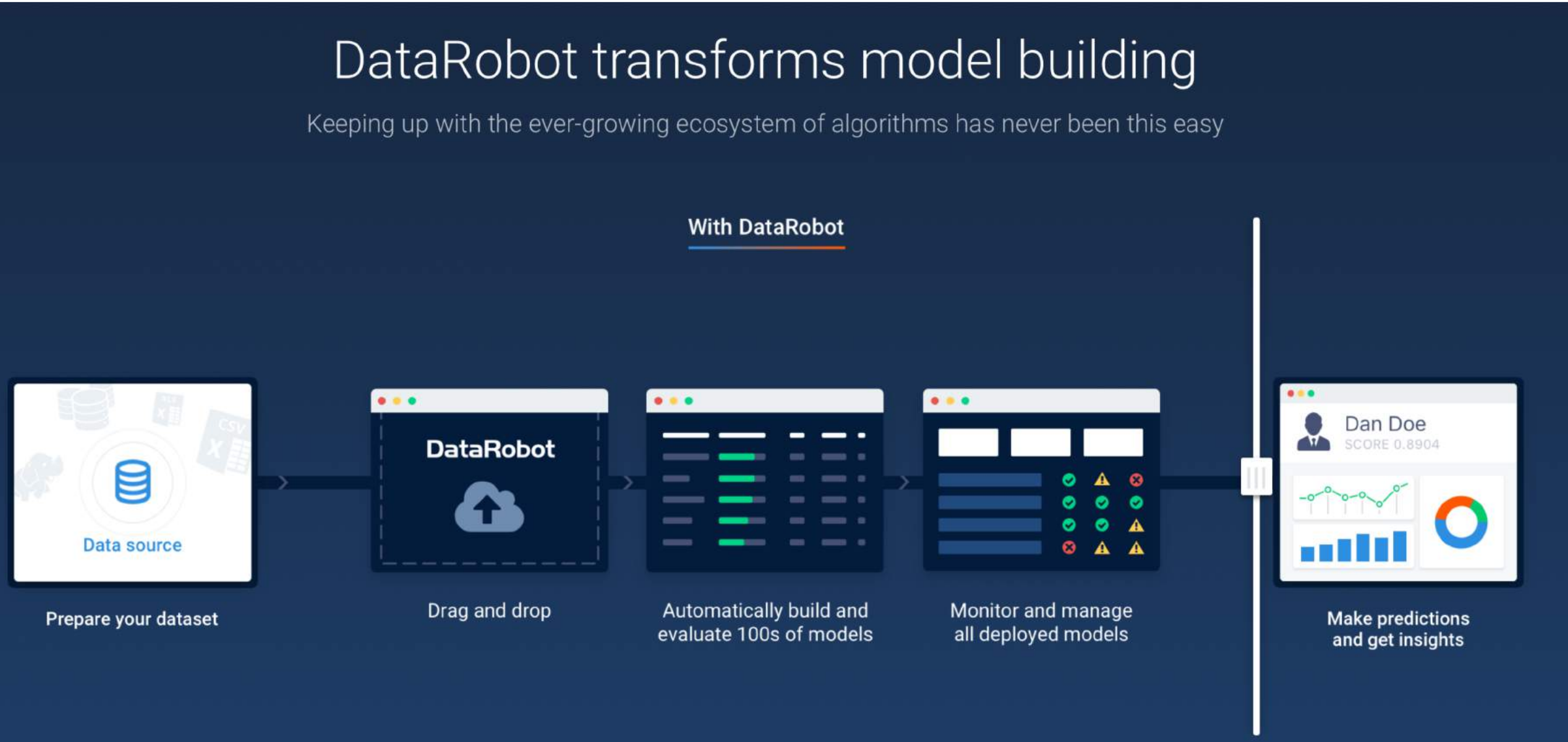
The TPOT software is open-source, programmed in Python, and available on GitHub.



TPOT — -bash — bash — Homebrew — 193×37

```
(tpot) cree:TPOT stevens$ tpot data/mnist.csv -is , -target class -o tpot_exported_pipeline.py -g 5 -p 20 -cv 5 -s 42 -v 2

TPOT settings:
CHECKPOINT_FOLDER    =     None
CONFIG_FILE          =     None
CROSSOVER_RATE       =     0.1
EARLY_STOP           =     None
GENERATIONS          =     5
INPUT_FILE           =     data/mnist.csv
INPUT_SEPARATOR      =     ,
MAX_EVAL_MINS        =     5
MAX_TIME_MINS        =     None
MEMORY               =     None
MUTATION_RATE        =     0.9
NUM_CV_FOLDS         =     5
NUM_JOBS             =     1
OFFSPRING_SIZE       =     20
OUTPUT_FILE          =     tpot_exported_pipeline.py
POPULATION_SIZE      =     20
RANDOM_STATE         =     42
SCORING_FN           =     accuracy
SUBSAMPLE            =     1.0
TARGET_NAME          =     class
TPOT_MODE            =     classification
VERBOSITY            =     2

Generation 1 - Current best internal CV score: 0.9526475883008712
Generation 2 - Current best internal CV score: 0.9526475883008712
Generation 3 - Current best internal CV score: 0.9532378704030924
Generation 4 - Current best internal CV score: 0.9616759659341992
Generation 5 - Current best internal CV score: 0.9616759659341992

Best pipeline: RandomForestClassifier(ZeroCount(input_matrix), bootstrap=True, criterion=entropy, max_features=0.05, min_samples_leaf=4, min_samples_split=3, n_estimators=100)

Training score: 0.9616759659341992
Holdout score: 0.9596571428571429
(tpot) cree:TPOT stevens$
```

https://www.datarobot.com/product/



Hyperparameter Optimization

(sometimes also called “metaparameter” optimization)

https://github.com/jhfjhfj1/autokeras

Hyperparameter Scanning and Optimization for Keras Tweet

build passing coverage 79%

Talos is a solution that helps finding hyperparameter configurations for Keras models. To perform hyperparameter optimization with Talos, there is no need to learn any new syntax, or change anything in the way Keras models are created. Keras functionality is fully exposed, and any parameter can be included in the scans.

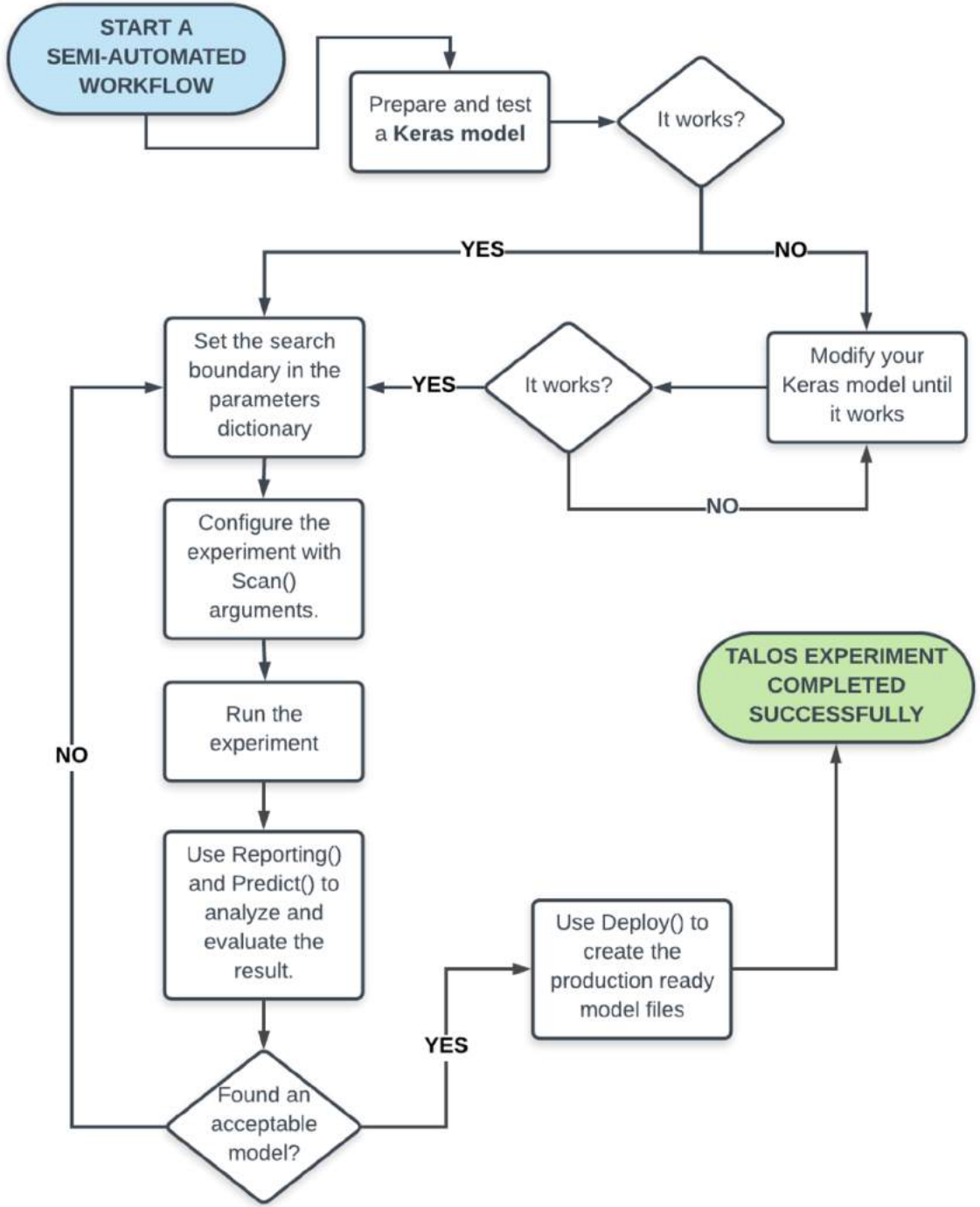
See a brief | concise | comprehensive example Notebook

Read the User Manual

Read a Report on Hyperparameter Optimization with Keras

Read the Roadmap

Install `pip install talos`


START A SEMI-AUTOMATED WORKFLOW
Prepare and test a **Keras model**
It works?
YES
NO
Modify your Keras model until it works
It works?
YES
Set the search boundary in the parameters dictionary
NO
Configure the experiment with Scan() arguments.
Run the experiment
Use Reporting() and Predict() to analyze and evaluate the result.
Found an acceptable model?
NO
YES
Use Deploy() to create the production ready model files
TALOS EXPERIMENT COMPLETED SUCCESSFULLY

How to use

Let's consider an example of a simple Keras model:

```
model = Sequential()
model.add(Dense(8, input_dim=x_train.shape[1], activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(y_train.shape[1], activation='softmax'))

model.compile(optimizer='adam',
              loss=categorical_crossentropy,
              metrics=['acc'])

out = model.fit(x_train, y_train,
                batch_size=20,
                epochs=200,
                verbose=0,
                validation_data=[x_val, y_val])
```

Autokeras

To prepare the model for a talos scan, we simply replace the parameters we want to include in the scans with references to our parameter dictionary (example of dictionary provided below). The below example code complete here.

```
def iris_model(x_train, y_train, x_val, y_val, params):

    model = Sequential()
    model.add(Dense(params['first_neuron'], input_dim=x_train.shape[1], activation=params['activation']))
    model.add(Dropout(params['dropout']))
    model.add(Dense(y_train.shape[1], activation=params['last_activation']))

    model.compile(optimizer=params['optimizer'],
                  loss=params['losses'],
                  metrics=['acc'])

    out = model.fit(x_train, y_train,
                    batch_size=params['batch_size'],
                    epochs=params['epochs'],
                    verbose=0,
                    validation_data=[x_val, y_val])

    return out, model
```

Autokeras

As you can see, the only thing that changed, is the values that we provide for the parameters. We then pass the parameters with a dictionary:

As you can see, the only thing that changed, is the values that we provide for the parameters. We then pass the parameters with a dictionary:

```
p = {'lr': (2, 10, 30),
     'first_neuron':[4, 8, 16, 32, 64, 128],
     'hidden_layers':[2,3,4,5,6],
     'batch_size': [2, 3, 4],
     'epochs': [300],
     'dropout': (0, 0.40, 10),
     'weight_regulizer':[None],
     'emb_output_dims': [None],
     'optimizer': [Adam, Nadam],
     'losses': [categorical_crossentropy, logcosh],
     'activation':[relu, elu],
     'last_activation': [softmax]}
```

Autokeras

The above example is a simple indication of what is possible. Any parameter that Keras accepts, can be included in the dictionary format.

Talos accepts lists with values, and tuples (start, end, n). Learning rate is normalized to 1 so that for each optimizer, lr=1 is the default Keras setting. Once this is all done, we can run the scan:

```
h = ta.Scan(x, y,
            params=p,
            dataset_name='first_test',
            experiment_no='2',
            model=iris_model,
            grid_downsample=0.5)
```

Autokeras

Hyperparameter Optimization for Keras

- DeepReplay — Hyperparameter Visualization
- Hyperas — Keras Wrapper for Hyperopt
- Kopt — Another Hyperopt Based Optimizer
- Talos — Hyperparameter Optimization for Keras

There is also the KerasClassifier inside sklearn for grid search.

Other Hyperparameter Optimization Solutions

- auto_ml — Automated machine learning
- BTB — Bayesian Tuning and Bandits
- Chocolate — Decentralized Hyperparameter Optimization
- Cornell-MOE — parallel Bayesian optimization algorithms
- deap — Evolutionary Algorithm Optimization
- devol — Evolutionary Algorithm optimization
- GPyOpt — Gaussian Process Optimization
- H20 — Automatic Machine Learning
- HORD — Deterministic RBF Surrogates

Neural Architecture Search

LITERATURE ON NEURAL ARCHITECTURE SEARCH

The following list considers papers related to neural architecture search. It is by no means a complete list. If you miss a paper on this list, please let us know.

- Architecture Search (and Hyperparameter Optimization):
  - Automatically Evolving CNN Architectures Based on Blocks (Sun et al. 2018)
    https://arxiv.org/abs/1810.11875
  - Training Frankenstein's Creature to Stack: HyperTree Architecture Search (Hundt et al. 2018)
    https://arxiv.org/abs/1810.11714
  - Fast Neural Architecture Search of Compact Semantic Segmentation Models
    via Auxiliary Cells (Nekrasov et al. 2018)
    https://arxiv.org/abs/1810.10804
  - Automatic Configuration of Deep Neural Networks with Parallel Efficient Global Optimization (van Stein et al. 2018)
    https://arxiv.org/abs/1810.05526
  - Graph Hypernetworks for Neural Architecture Search (Zhang et al. 2018)
    https://arxiv.org/abs/1810.05749
  - Gradient Based Evolution to Optimize the Structure of Convolutional Neural Networks (Mitschke et al. 2018)
    https://ieeexplore.ieee.org/document/8451394
  - Neural Architecture Optimization (Luo et al. 2018)
    https://arxiv.org/abs/1808.07233
  - Exploring Shared Structures and Hierarchies for Multiple NLP Tasks (Chen et al. 2018)
    https://arxiv.org/abs/1808.07658
  - Neural Architecture Search: A Survey (Elsken et al. 2018)
    https://arxiv.org/abs/1808.05377
  - BlockQNN: Efficient Block-wise Neural Network Architecture Generation (Zhong et al. 2018)
    https://arxiv.org/abs/1808.05584
  - Automatically Designing CNN Architectures Using Genetic Algorithm for Image Classification (Sunet al. 2018)
    https://arxiv.org/abs/1808.03818
  - Reinforced Evolutionary Neural Architecture Search (Chen et al. 2018)
    https://arxiv.org/abs/1808.00193

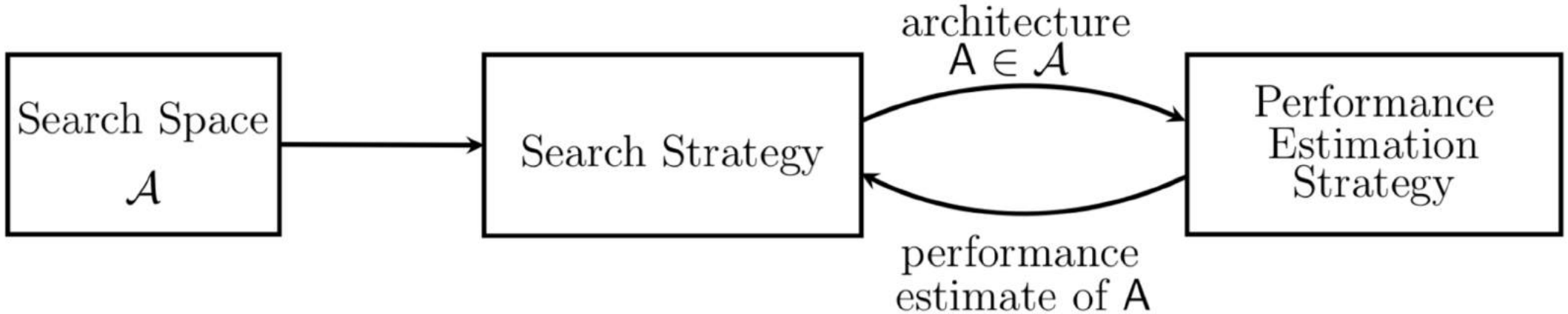


Figure 1: Abstract illustration of Neural Architecture Search methods. A search strategy selects an architecture A from a predefined search space $\mathcal{A}$. The architecture is passed to a performance estimation strategy, which returns the estimated performance of A to the search strategy.

- **Search Strategy.** The search strategy details how to explore the search space. It encompasses the classical exploration-exploitation trade-off since, on the one hand, it is desirable to find well-performing architectures quickly, while on the other hand, premature convergence to a region of suboptimal architectures should be avoided.

- **Performance Estimation Strategy.** The objective of NAS is typically to find architectures that achieve high predictive performance on unseen data. *Performance Estimation* refers to the process of estimating this performance: the simplest option is to perform a standard training and validation of the architecture on data, but this is unfortunately computationally expensive and limits the number of architectures that can be explored. Much recent research therefore focuses on developing methods that reduce the cost of these performance estimations.

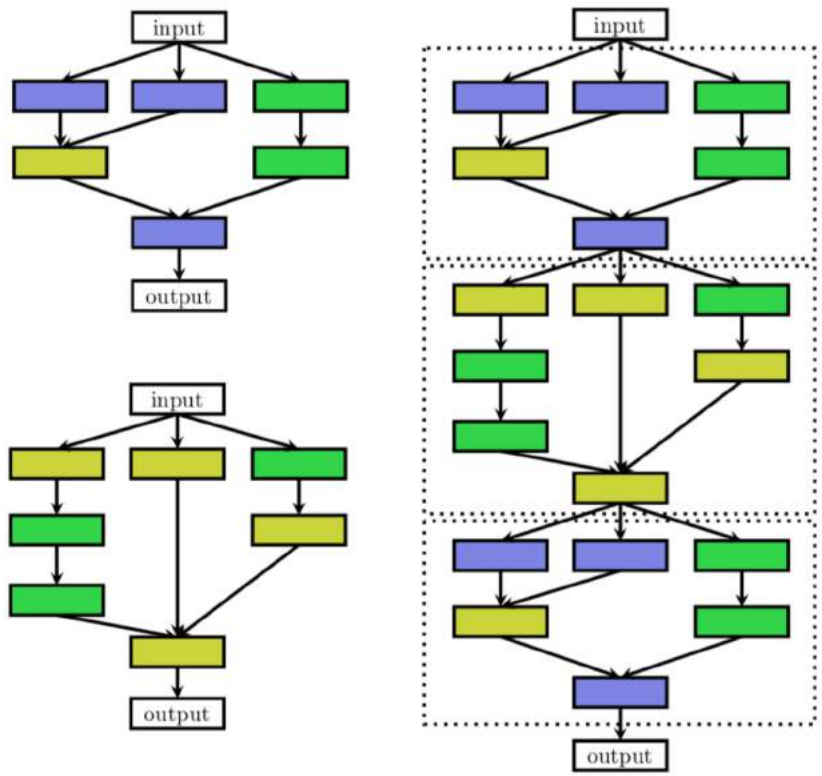


Figure 3: Illustration of the cell search space. Left: Two different cells, e.g., a normal cell (top) and a reduction cell (bottom) (Zoph et al., 2018). Right: an architecture built by stacking the cells sequentially. Note that cells can also be combined in a more complex manner, such as in multi-branch spaces, by simply replacing layers with cells.

1. The size of the search space is drastically reduced since cells can be comparably small. For example, Zoph et al. (2018) estimate a seven-times speed-up compared to their previous work (Zoph and Le, 2017) while achieving better performance.

2. Cells can more easily be transferred to other datasets by adapting the number of cells used within a model. Indeed, Zoph et al. (2018) transfer cells optimized on CIFAR-10 to ImageNet and achieve state-of-the-art performance.

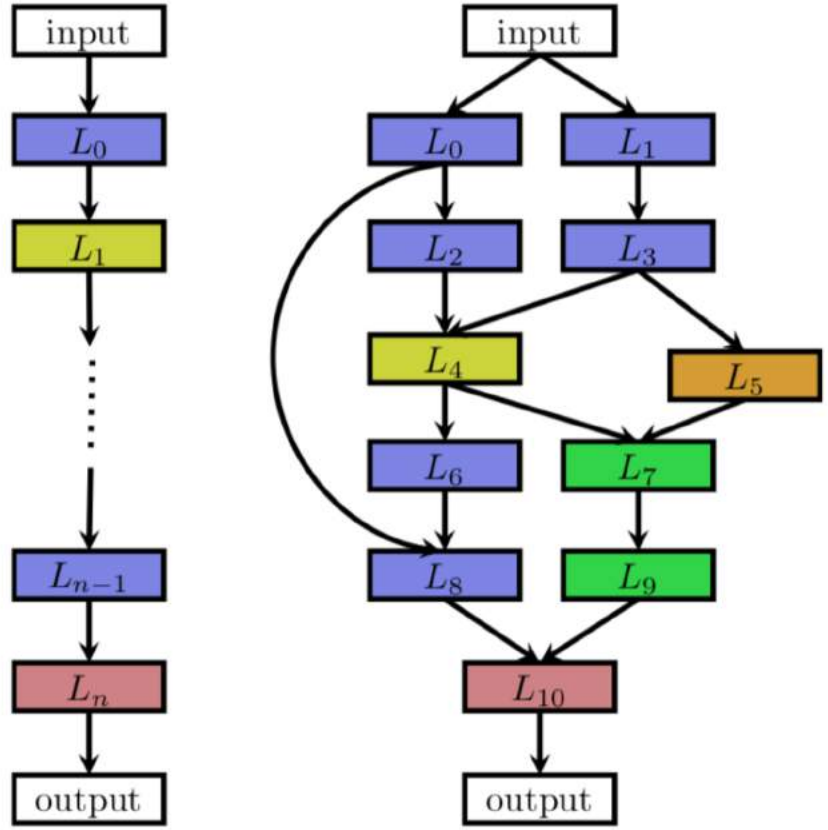


Figure 2: An illustration of different architecture spaces. Each node in the graphs corresponds to a layer in a neural network, e.g., a convolutional or pooling layer. Different layer types are visualized by different colors. An edge from layer L_i to layer L_j denotes that L_i receives the output of L_j as input. Left: an element of a chain-structured space. Right: an element of a more complex search space with additional layer types and multiple branches and skip connections.

NAS with RL

NEURAL ARCHITECTURE SEARCH WITH REINFORCEMENT LEARNING

Barret Zoph,* **Quoc V. Le**
Google Brain
{barretzoph,qvl}@google.com

ABSTRACT

Neural networks are powerful and flexible models that work well for many difficult learning tasks in image, speech and natural language understanding. Despite their success, neural networks are still hard to design. In this paper, we use a recurrent network to generate the model descriptions of neural networks and train this RNN with reinforcement learning to maximize the expected accuracy of the generated architectures on a validation set. On the CIFAR-10 dataset, our method, starting from scratch, can design a novel network architecture that rivals the best human-invented architecture in terms of test set accuracy. Our CIFAR-10 model achieves a test error rate of 3.65, which is 0.09 percent better and 1.05x faster than the previous state-of-the-art model that used a similar architectural scheme. On the Penn Treebank dataset, our model can compose a novel recurrent cell that outperforms the widely-used LSTM cell, and other state-of-the-art baselines. Our cell achieves a test set perplexity of 62.4 on the Penn Treebank, which is 3.6 perplexity better than the previous state-of-the-art model. The cell can also be transferred to the character language modeling task on PTB and achieves a state-of-the-art perplexity of 1.214.

1 INTRODUCTION

The last few years have seen much success of deep neural networks in many challenging applications, such as speech recognition (Hinton et al., 2012), image recognition (LeCun et al., 1998;

NEURAL ARCHITECTURE SEARCH WITH REINFORCEMENT LEARNING

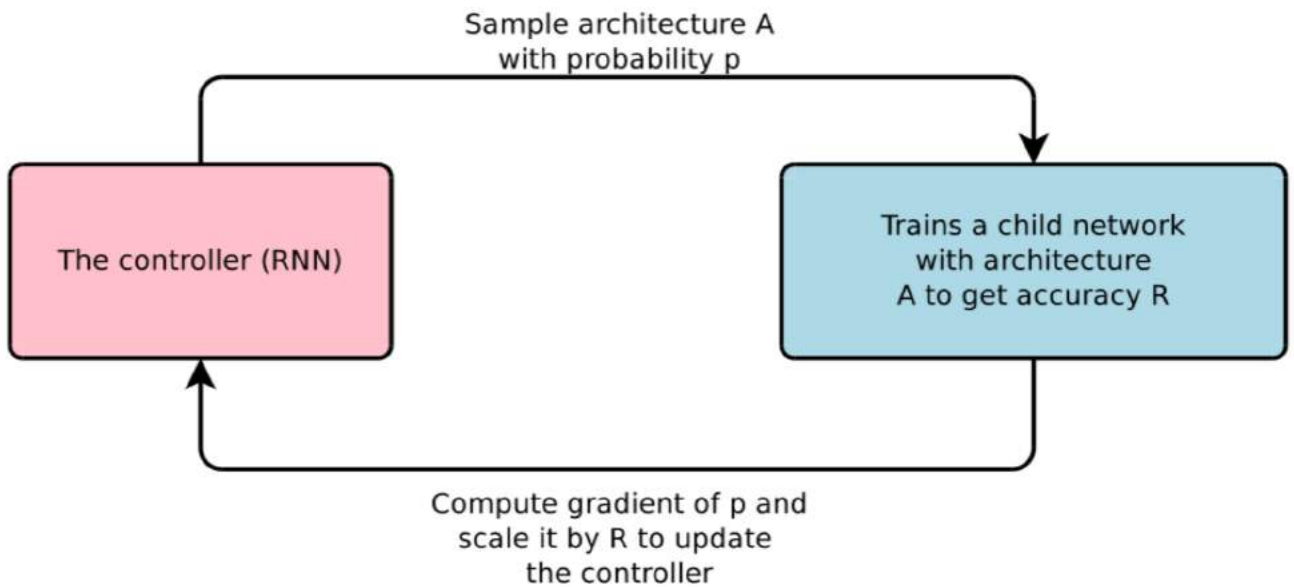


Figure 1: An overview of Neural Architecture Search.

NEURAL ARCHITECTURE SEARCH WITH REINFORCEMENT LEARNING

In Neural Architecture Search, we use a controller to generate architectural hyperparameters of neural networks. To be flexible, the controller is implemented as a recurrent neural network. Let's suppose we would like to predict feedforward neural networks with only convolutional layers, we can use the controller to generate their hyperparameters as a sequence of tokens:

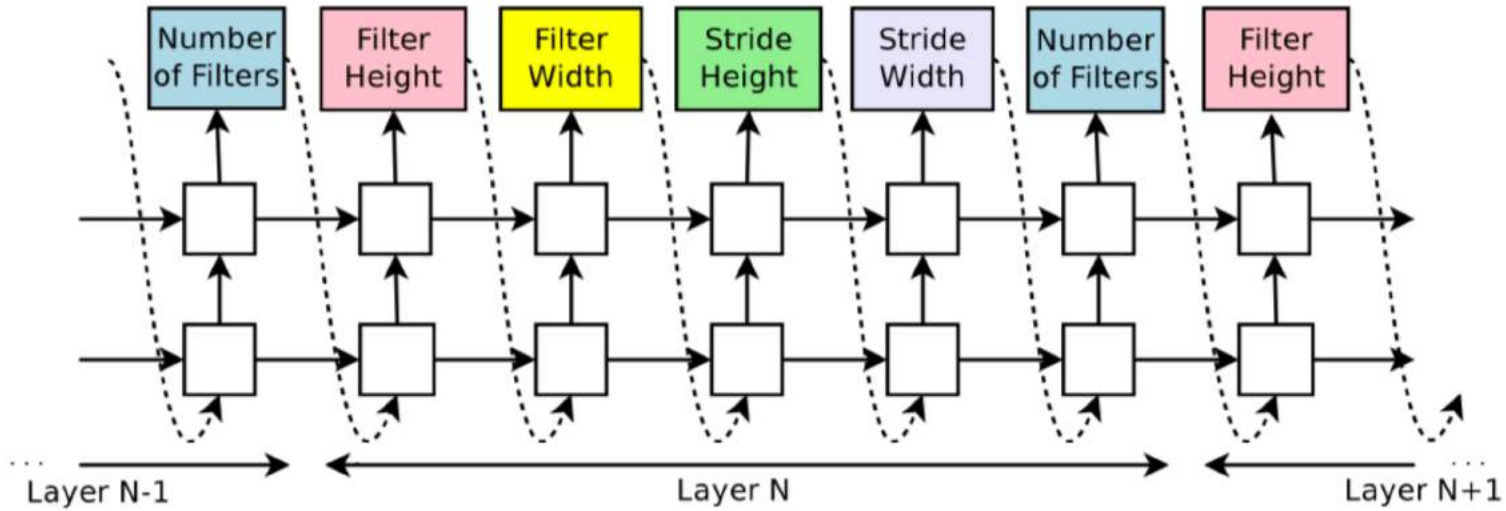


Figure 2: How our controller recurrent neural network samples a simple convolutional network. It predicts filter height, filter width, stride height, stride width, and number of filters for one layer and repeats. Every prediction is carried out by a softmax classifier and then fed into the next time step as input.

NEURAL ARCHITECTURE SEARCH WITH REINFORCEMENT LEARNING

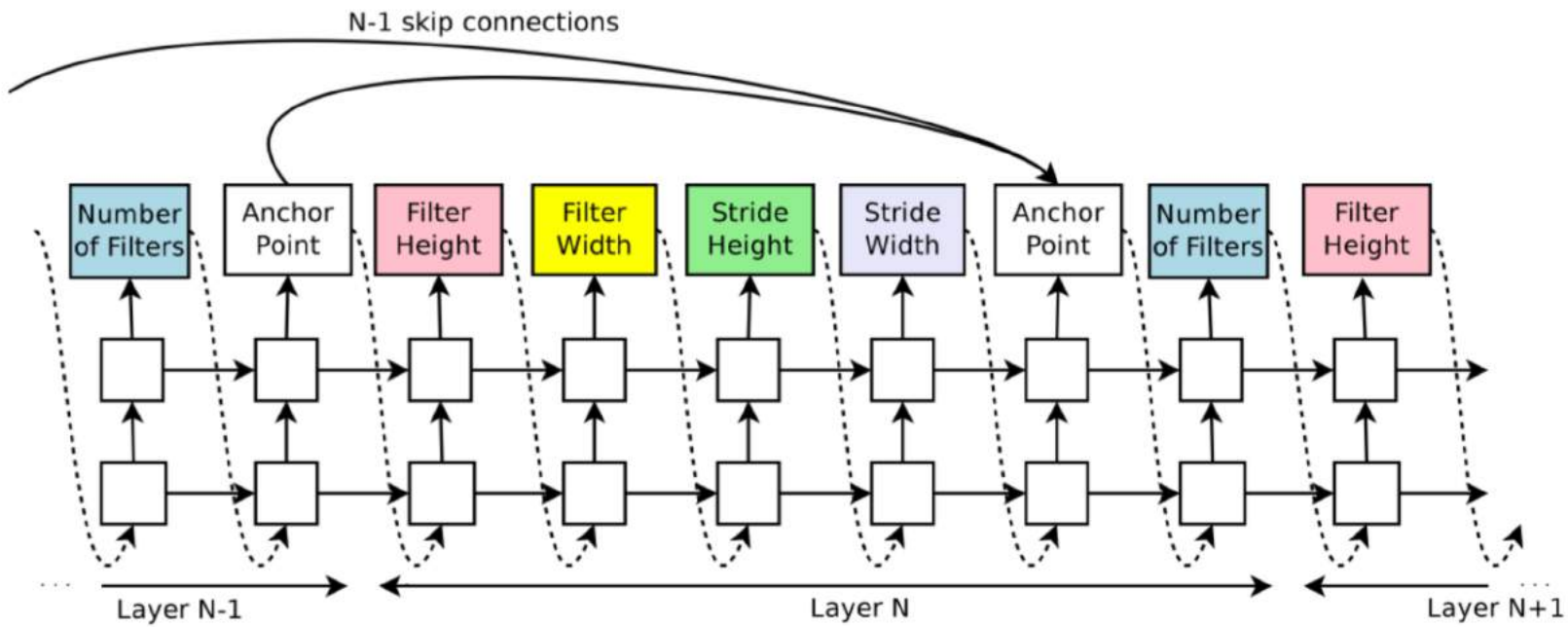


Figure 4: The controller uses anchor points, and set-selection attention to form skip connections.

NEURAL ARCHITECTURE SEARCH WITH REINFORCEMENT LEARNING

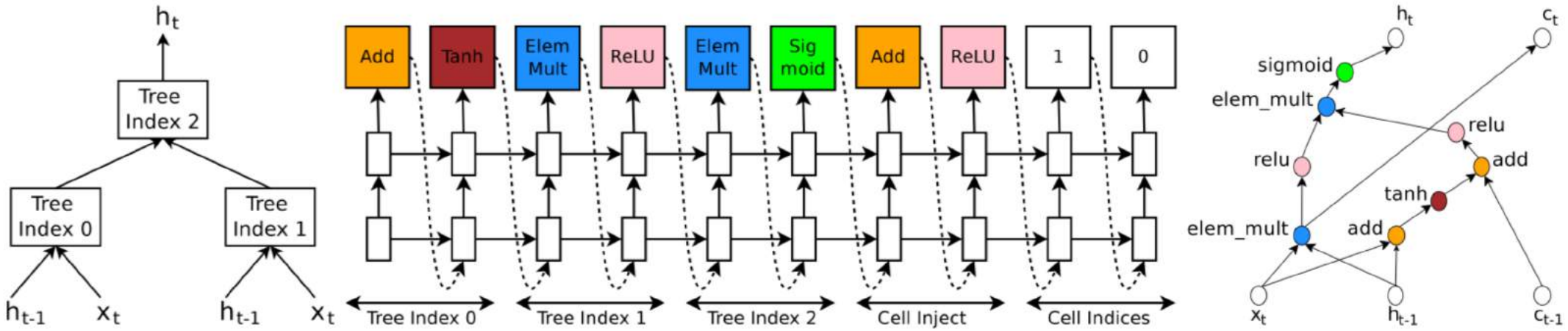


Figure 5: An example of a recurrent cell constructed from a tree that has two leaf nodes (base 2) and one internal node. Left: the tree that defines the computation steps to be predicted by controller. Center: an example set of predictions made by the controller for each computation step in the tree. Right: the computation graph of the recurrent cell constructed from example predictions of the controller.

Model	Depth	Parameters	Error rate (%)
Network in Network (Lin et al., 2013)	-	-	8.81
All-CNN (Springenberg et al., 2014)	-	-	7.25
Deeply Supervised Net (Lee et al., 2015)	-	-	7.97
Highway Network (Srivastava et al., 2015)	-	-	7.72
Scalable Bayesian Optimization (Snoek et al., 2015)	-	-	6.37
FractalNet (Larsson et al., 2016)	21	38.6M	5.22
with Dropout/Drop-path	21	38.6M	4.60
ResNet (He et al., 2016a)	110	1.7M	6.61
ResNet (reported by Huang et al. (2016c))	110	1.7M	6.41
ResNet with Stochastic Depth (Huang et al., 2016c)	110	1.7M	5.23
	1202	10.2M	4.91
Wide ResNet (Zagoruyko & Komodakis, 2016)	16	11.0M	4.81
	28	36.5M	4.17
ResNet (pre-activation) (He et al., 2016b)	164	1.7M	5.46
	1001	10.2M	4.62
DenseNet ($L = 40, k = 12$) Huang et al. (2016a)	40	1.0M	5.24
DenseNet($L = 100, k = 12$) Huang et al. (2016a)	100	7.0M	4.10
DenseNet ($L = 100, k = 24$) Huang et al. (2016a)	100	27.2M	3.74
DenseNet-BC ($L = 100, k = 40$) Huang et al. (2016b)	190	25.6M	3.46
Neural Architecture Search v1 no stride or pooling	15	4.2M	5.50
Neural Architecture Search v2 predicting strides	20	2.5M	6.01
Neural Architecture Search v3 max pooling	39	7.1M	4.47
Neural Architecture Search v3 max pooling + more filters	39	37.4M	3.65

Table 1: Performance of Neural Architecture Search and other state-of-the-art models on CIFAR-10.

Model	Parameters	Test Perplexity
Mikolov & Zweig (2012) - KN-5	2M‡	141.2
Mikolov & Zweig (2012) - KN5 + cache	2M‡	125.7
Mikolov & Zweig (2012) - RNN	6M‡	124.7
Mikolov & Zweig (2012) - RNN-LDA	7M‡	113.7
Mikolov & Zweig (2012) - RNN-LDA + KN-5 + cache	9M‡	92.0
Pascanu et al. (2013) - Deep RNN	6M	107.5
Cheng et al. (2014) - Sum-Prod Net	5M‡	100.0
Zaremba et al. (2014) - LSTM (medium)	20M	82.7
Zaremba et al. (2014) - LSTM (large)	66M	78.4
Gal (2015) - Variational LSTM (medium, untied)	20M	79.7
Gal (2015) - Variational LSTM (medium, untied, MC)	20M	78.6
Gal (2015) - Variational LSTM (large, untied)	66M	75.2
Gal (2015) - Variational LSTM (large, untied, MC)	66M	73.4
Kim et al. (2015) - CharCNN	19M	78.9
Press & Wolf (2016) - Variational LSTM, shared embeddings	51M	73.2
Merity et al. (2016) - Zoneout + Variational LSTM (medium)	20M	80.6
Merity et al. (2016) - Pointer Sentinel-LSTM (medium)	21M	70.9
Inan et al. (2016) - VD-LSTM + REAL (large)	51M	68.5
Zilly et al. (2016) - Variational RHN, shared embeddings	24M	66.0
Neural Architecture Search with base 8	32M	67.9
Neural Architecture Search with base 8 and shared embeddings	25M	64.0
Neural Architecture Search with base 8 and shared embeddings	54M	62.4

Table 2: Single model perplexity on the test set of the Penn Treebank language modeling task. Parameter numbers with ‡ are estimates with reference to Merity et al. (2016).

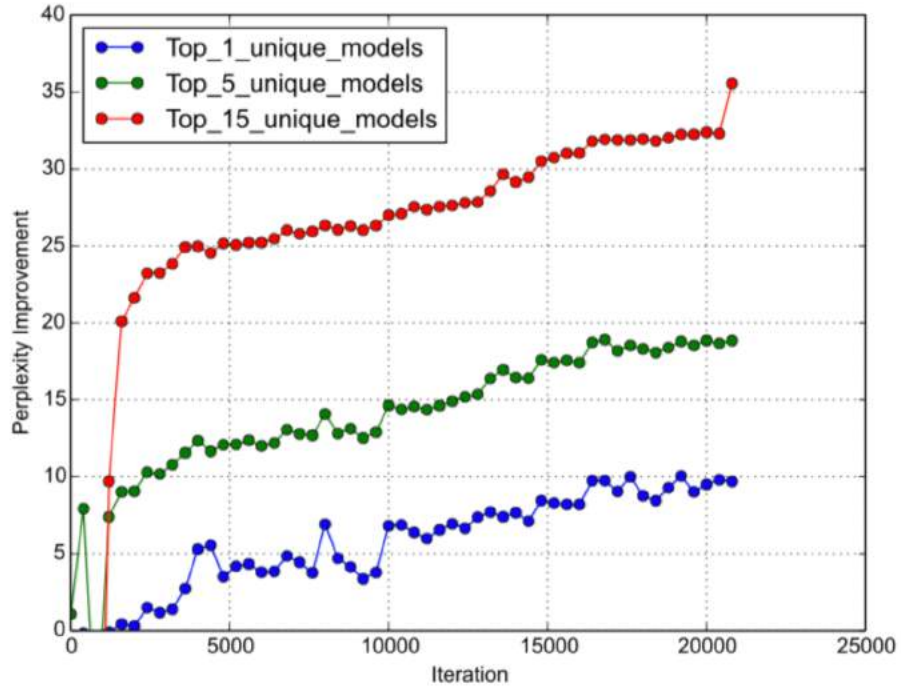


Figure 6: Improvement of Neural Architecture Search over random search over time. We plot the difference between the average of the top k models our controller finds vs. random search every 400 models run.

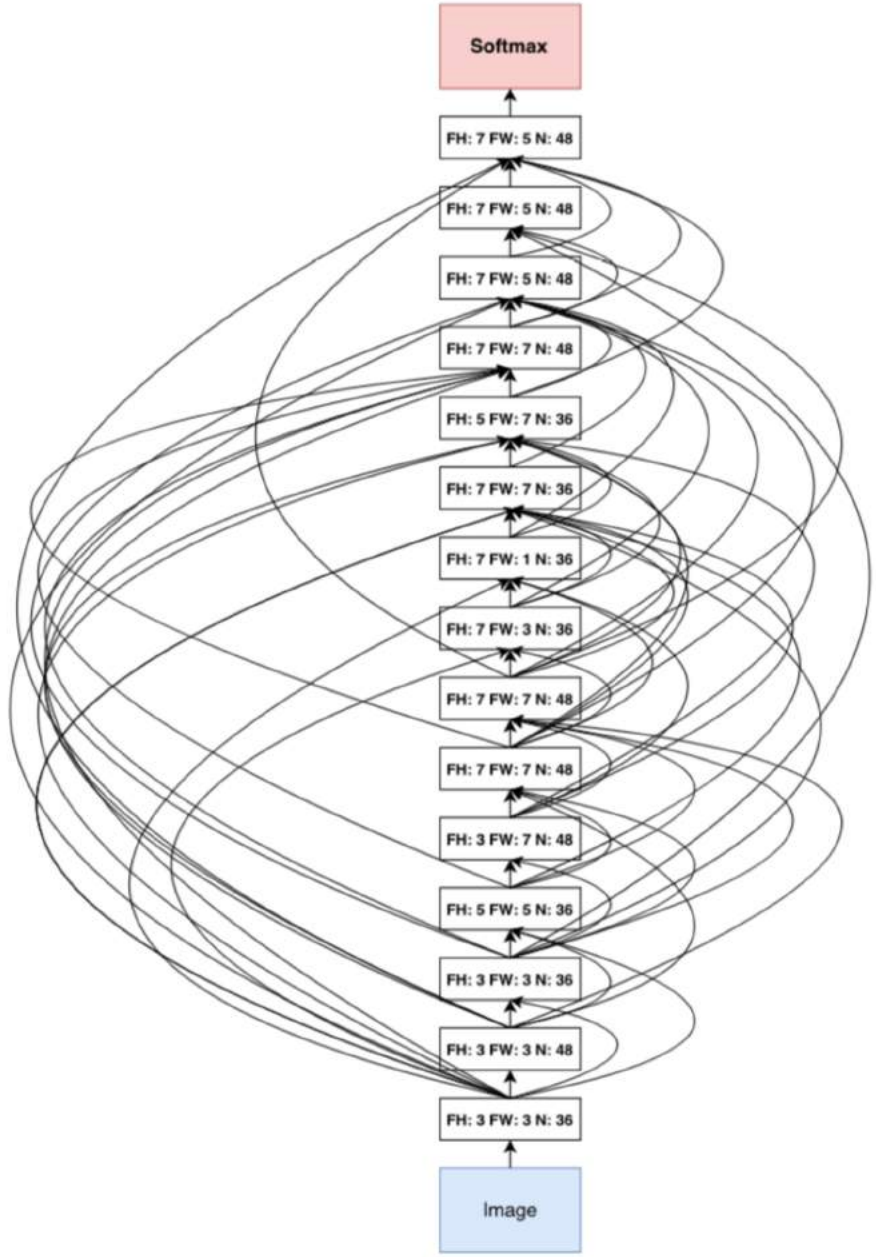


Figure 7: Convolutional architecture discovered by our method, when the search space does not have strides or pooling layers. FH is filter height, FW is filter width and N is number of filters. Note that the skip connections are not residual connections. If one layer has many input layers then all input layers are concatenated in the depth dimension.

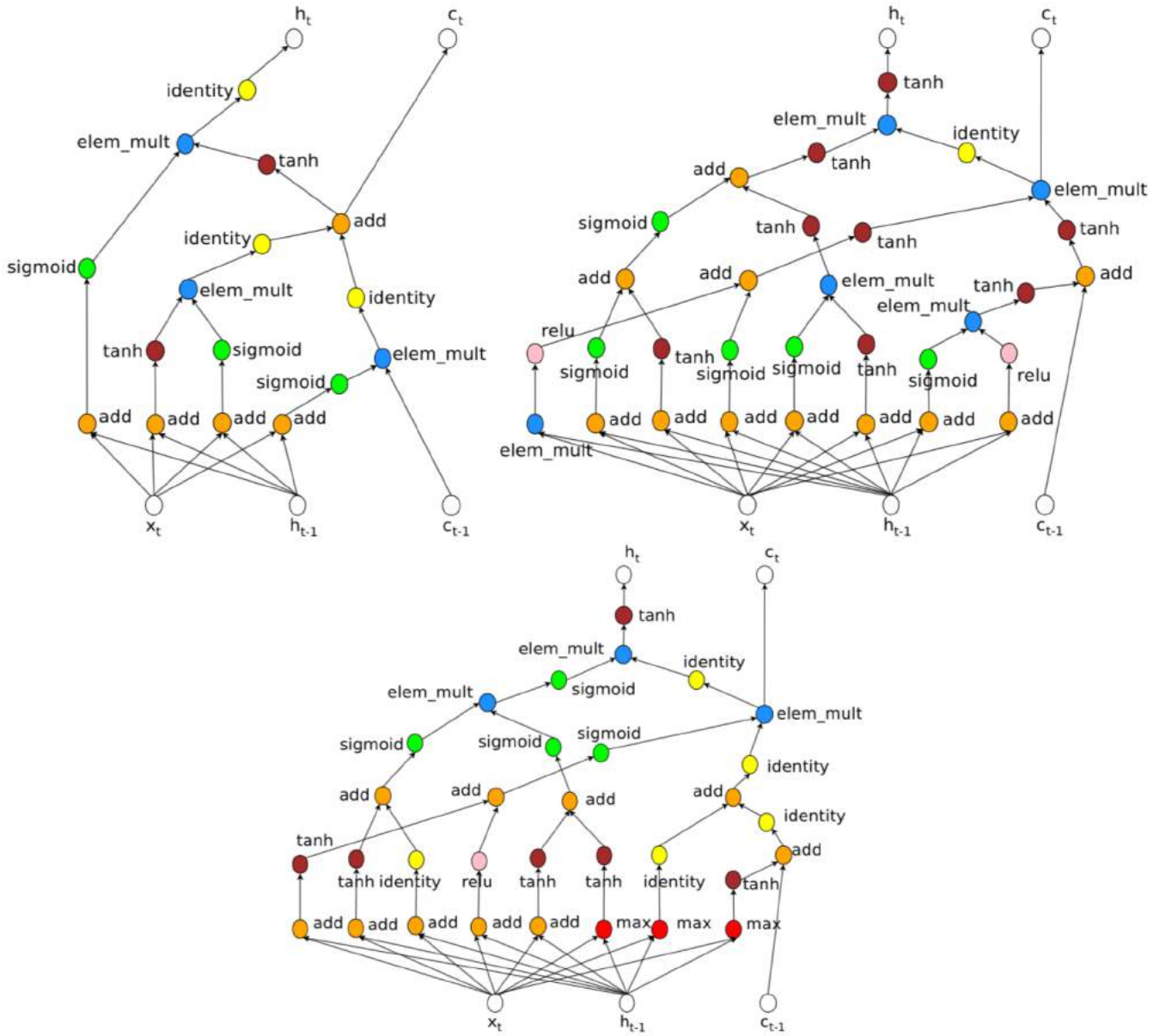


Figure 8: A comparison of the original LSTM cell vs. two good cells our model found. Top left: LSTM cell. Top right: Cell found by our model when the search space does not include max and sin. Bottom: Cell found by our model when the search space includes max and sin (the controller did not choose to use the sin function).

NASNet

Learning Transferable Architectures for Scalable Image Recognition

Barret Zoph
Google Brain
barretzoph@google.com

Vijay Vasudevan
Google Brain
vrv@google.com

Jonathon Shlens
Google Brain
shlens@google.com

Quoc V. Le
Google Brain
qvl@google.com

Abstract

Developing neural network image classification models often requires significant architecture engineering. In this paper, we study a method to learn the model architectures directly on the dataset of interest. As this approach is expensive when the dataset is large, we propose to search for an architectural building block on a small dataset and then transfer the block to a larger dataset. The key contribution of this work is the design of a new search space (which we call the "NASNet search space") which enables transferability. In our experiments, we search for the best convolutional layer (or "cell") on the CIFAR-10 dataset and then apply this cell to the ImageNet dataset by stacking together more copies of this cell, each with their own parameters to design a convolutional architecture, which we name a "NASNet architecture". We also introduce a new regu-

1. Introduction

Developing neural network image classification models often requires significant *architecture engineering*. Starting from the seminal work of [32] on using convolutional architectures [17, 34] for ImageNet [11] classification, successive advancements through architecture engineering have achieved impressive results [53, 59, 20, 60, 58, 68].

In this paper, we study a new paradigm of designing convolutional architectures and describe a scalable method to optimize convolutional architectures on a dataset of interest, for instance the ImageNet classification dataset. Our approach is inspired by the recently proposed Neural Architecture Search (NAS) framework [71], which uses a reinforcement learning search method to optimize architecture configurations. Applying NAS, or any other search methods, directly to a large dataset, such as the ImageNet dataset, is however computationally expensive. We there-

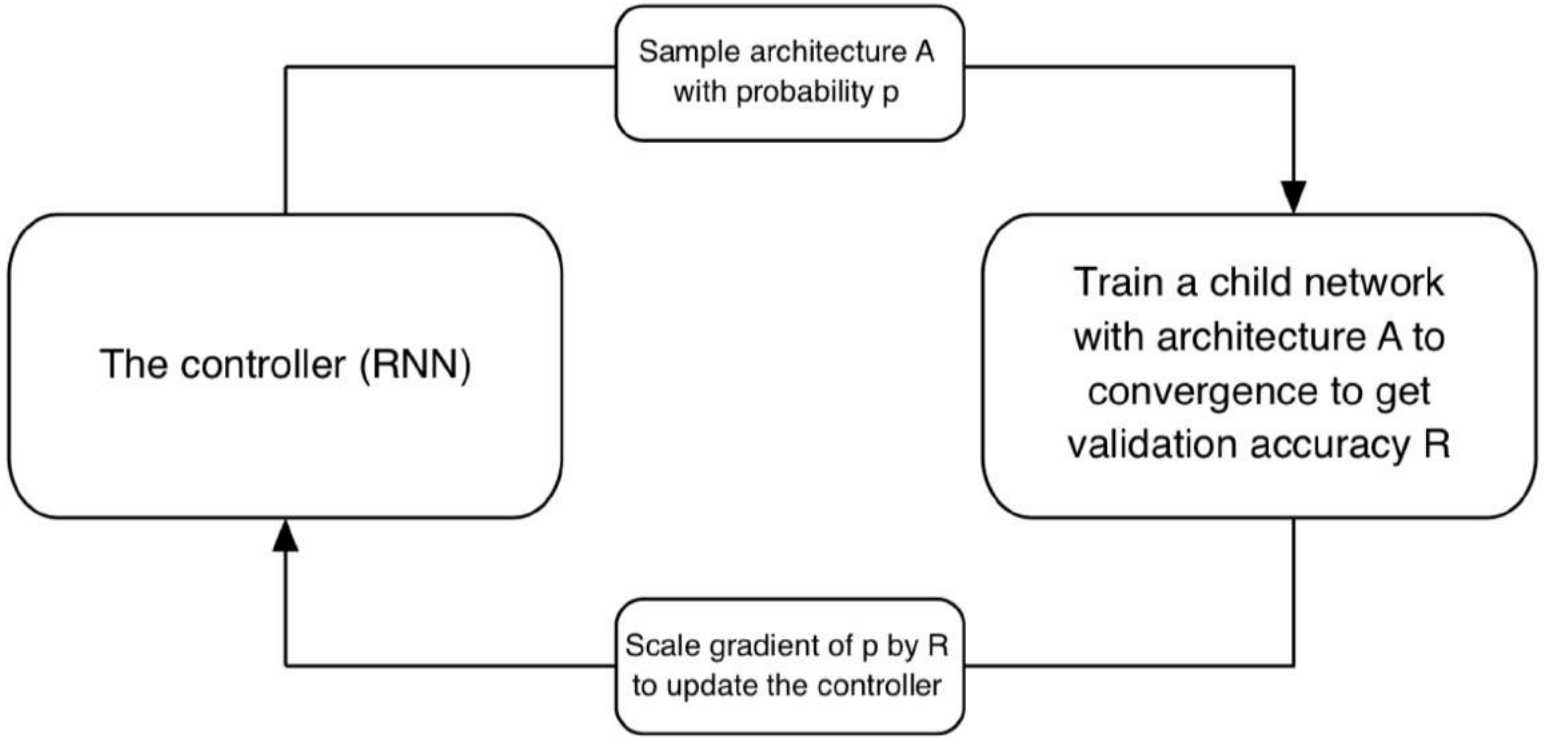


Figure 1. Overview of Neural Architecture Search [71]. A controller RNN predicts architecture A from a search space with probability p. A child network with architecture A is trained to convergence achieving accuracy R. Scale the gradients of p by R to update the RNN controller.

Learning Transferable Architectures for Scalable Image Recognition

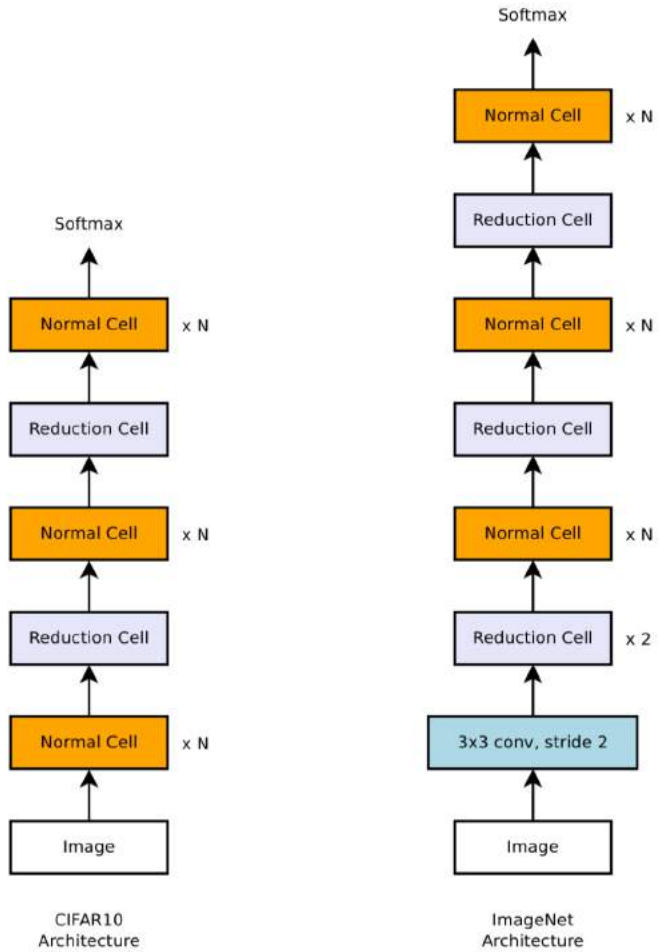


Figure 2. Scalable architectures for image classification consist of two repeated motifs termed *Normal Cell* and *Reduction Cell*. This diagram highlights the model architecture for CIFAR-10 and ImageNet. The choice for the number of times the Normal Cells that gets stacked between reduction cells, N, can vary in our experi-

Note the replication

x N

Learning Transferable Architectures for Scalable Image Recognition

In steps 3 and 4, the controller RNN selects an operation to apply to the hidden states. We collected the following set of operations based on their prevalence in the CNN literature:

- identity
- 1x7 then 7x1 convolution
- 3x3 average pooling
- 5x5 max pooling
- 1x1 convolution
- 3x3 depthwise-separable conv
- 7x7 depthwise-separable conv
- 1x3 then 3x1 convolution
- 3x3 dilated convolution
- 3x3 max pooling
- 7x7 max pooling
- 3x3 convolution
- 5x5 depthwise-seperable conv

Learning Transferable Architectures for Scalable Image Recognition

Finally, our work makes use of the reinforcement learning proposal in NAS [71]; however, it is also possible to use random search to search for architectures in the NASNet search space. In random search, instead of sampling the decisions from the softmax classifiers in the controller RNN, we can sample the decisions from the uniform distribution. In our experiments, we find that random search is slightly worse than reinforcement learning on the CIFAR- 10 dataset.

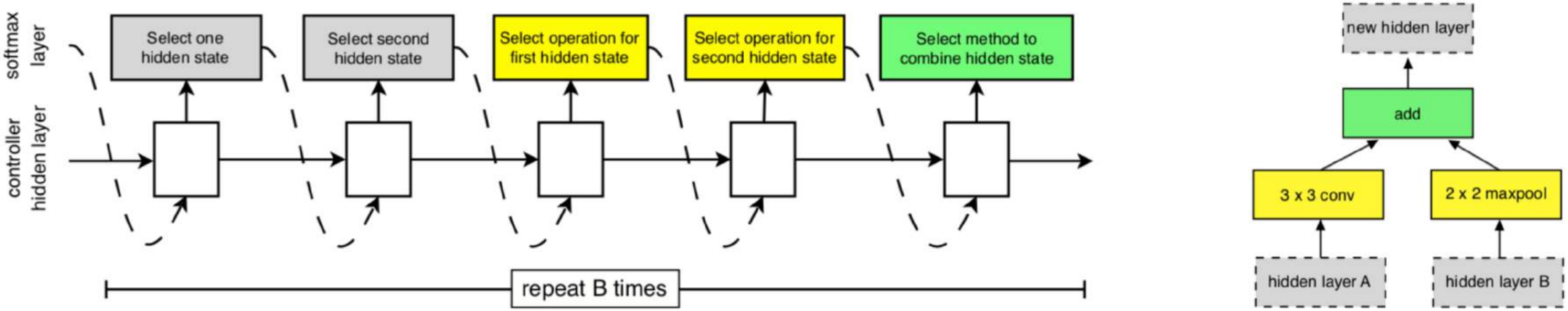


Figure 3. Controller model architecture for recursively constructing one block of a convolutional cell. Each block requires selecting 5 discrete parameters, each of which corresponds to the output of a softmax layer. Example constructed block shown on right. A convolutional cell contains B blocks, hence the controller contains $5B$ softmax layers for predicting the architecture of a convolutional cell. In our experiments, the number of blocks B is 5.

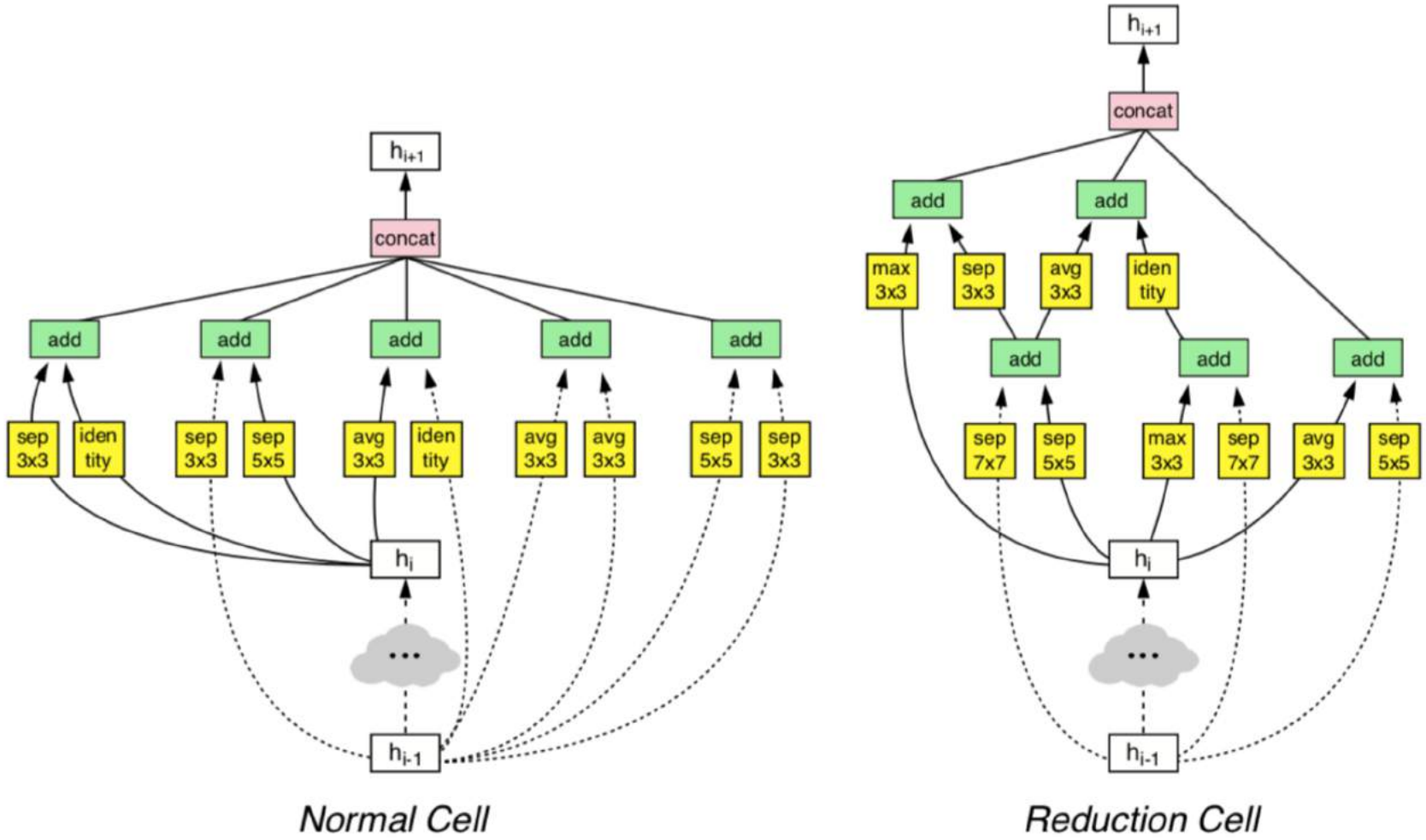


Figure 4. Architecture of the best convolutional cells (NASNet-A) with $B = 5$ blocks identified with CIFAR-10 . The input (white) is the hidden state from previous activations (or input image). The output (pink) is the result of a concatenation operation across all resulting branches. Each convolutional cell is the result of B blocks. A single block is corresponds to two primitive operations (yellow) and a combination operation (green). Note that colors correspond to operations in Figure 3.

Learning Transferable Architectures for Scalable Image Recognition

model	depth	# params	error rate (%)
DenseNet ($L = 40, k = 12$) [26]	40	1.0M	5.24
DenseNet($L = 100, k = 12$) [26]	100	7.0M	4.10
DenseNet ($L = 100, k = 24$) [26]	100	27.2M	3.74
DenseNet-BC ($L = 100, k = 40$) [26]	190	25.6M	3.46
Shake-Shake 26 2x32d [18]	26	2.9M	3.55
Shake-Shake 26 2x96d [18]	26	26.2M	2.86
Shake-Shake 26 2x96d + cutout [12]	26	26.2M	2.56
NAS v3 [71]	39	7.1M	4.47
NAS v3 [71]	39	37.4M	3.65
NASNet-A (6 @ 768)	-	3.3M	3.41
NASNet-A (6 @ 768) + cutout	-	3.3M	2.65
NASNet-A (7 @ 2304)	-	27.6M	2.97
NASNet-A (7 @ 2304) + cutout	-	27.6M	2.40
NASNet-B (4 @ 1152)	-	2.6M	3.73
NASNet-C (4 @ 640)	-	3.1M	3.59

Table 1. Performance of Neural Architecture Search and other state-of-the-art models on CIFAR-10. All results for NASNet are the mean accuracy across 5 runs.

Learning Transferable Architectures for Scalable Image Recognition

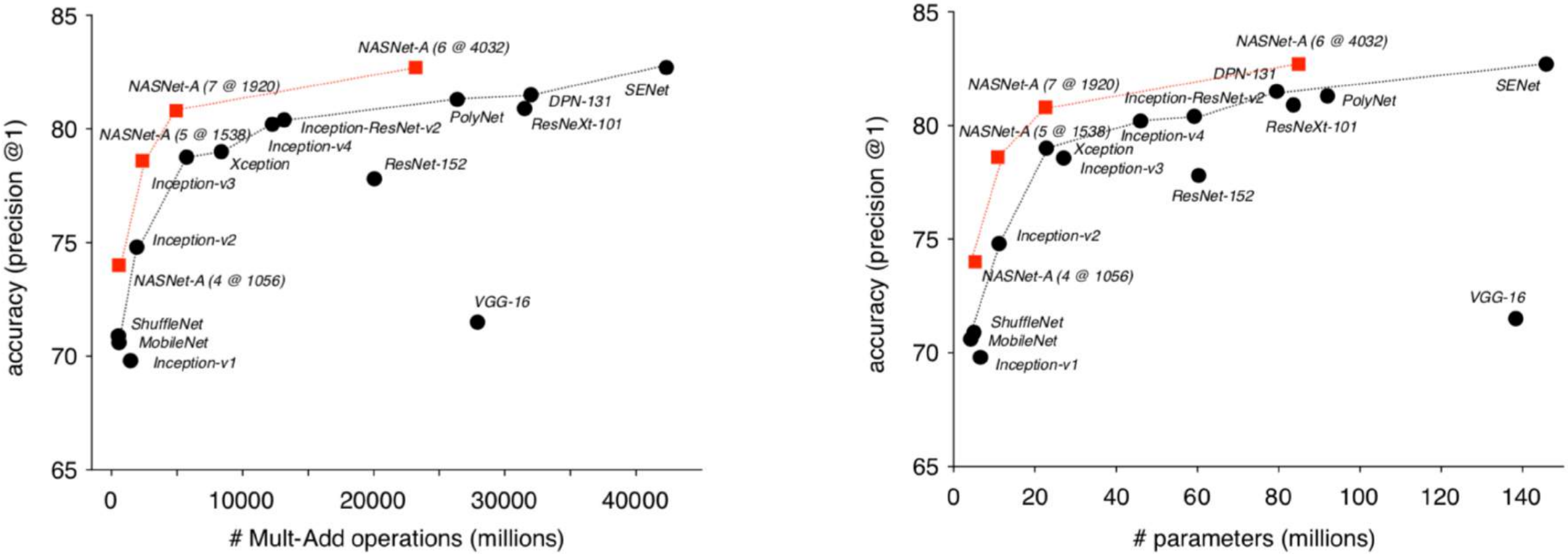


Figure 5. Accuracy versus computational demand (left) and number of parameters (right) across top performing published CNN architectures on ImageNet 2012 ILSVRC challenge prediction task. Computational demand is measured in the number of floating-point multiply-add operations to process a single image. Black circles indicate previously published results and red squares highlight our proposed models.

Learning Transferable Architectures for Scalable Image Recognition

Model	image size	# parameters	Mult-Adds	Top 1 Acc. (%)	Top 5 Acc. (%)
Inception V2 [29]	224×224	11.2 M	1.94 B	74.8	92.2
NASNet-A (5 @ 1538)	**299×299**	**10.9 M**	**2.35 B**	**78.6**	**94.2**
Inception V3 [60]	299×299	23.8 M	5.72 B	78.8	94.4
Xception [9]	299×299	22.8 M	8.38 B	79.0	94.5
Inception ResNet V2 [58]	299×299	55.8 M	13.2 B	80.1	95.1
NASNet-A (7 @ 1920)	**299×299**	**22.6 M**	**4.93 B**	**80.8**	**95.3**
ResNeXt-101 (64 x 4d) [68]	320×320	83.6 M	31.5 B	80.9	95.6
PolyNet [69]	331×331	92 M	34.7 B	81.3	95.8
DPN-131 [8]	320×320	79.5 M	32.0 B	81.5	95.8
SENet [25]	**320×320**	**145.8 M**	**42.3 B**	**82.7**	**96.2**
NASNet-A (6 @ 4032)	**331×331**	**88.9 M**	**23.8 B**	**82.7**	**96.2**

Table 2. Performance of architecture search and other published state-of-the-art models on ImageNet classification. Mult-Adds indicate the number of composite multiply-accumulate operations for a single image. Note that the composite multiple-accumulate operations are calculated for the image size reported in the table. Model size for [25] calculated from open-source implementation.

Learning Transferable Architectures for Scalable Image Recognition

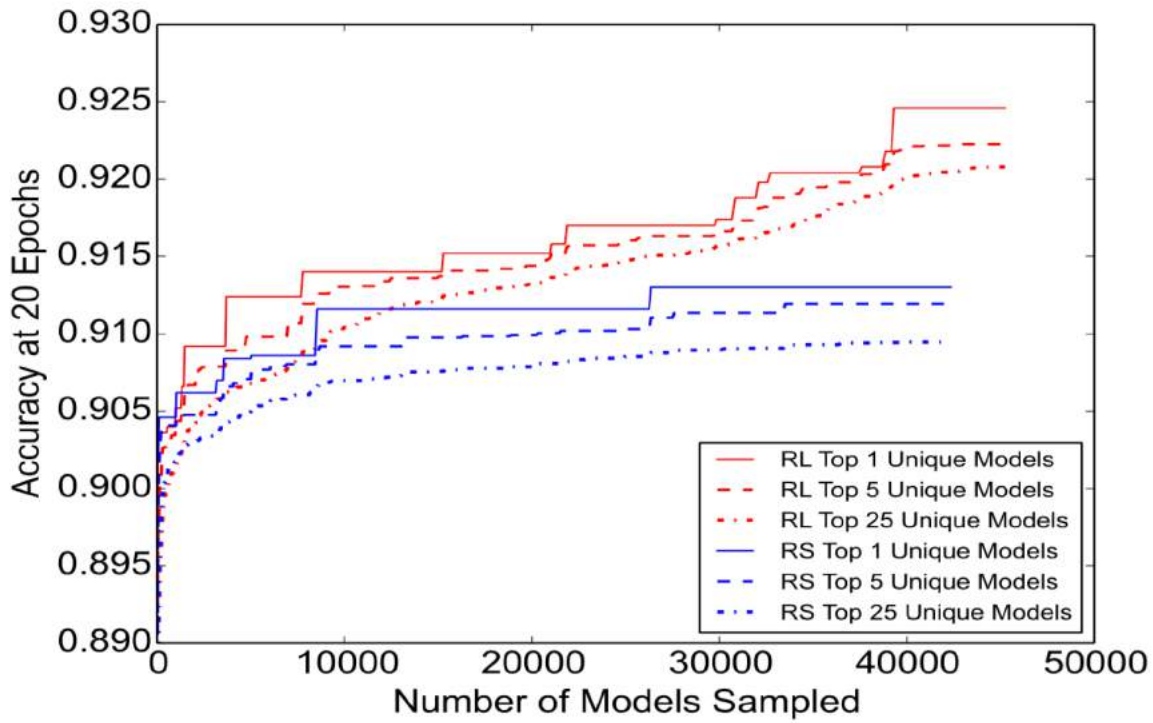


Figure 6. Comparing the efficiency of random search (RS) to reinforcement learning (RL) for learning neural architectures. The x-axis measures the total number of model architectures sampled, and the y-axis is the validation performance on CIFAR-10 after 20 epochs of training.

Progressive NAS

Progressive Neural Architecture Search

Chenxi Liu[1⋆], Barret Zoph[2], Maxim Neumann[2], Jonathon Shlens[2], Wei Hua[2], Li-Jia Li[2], Li Fei-Fei[2,3], Alan Yuille[1], Jonathan Huang[2], and Kevin Murphy[2]

[1] Johns Hopkins University
[2] Google AI
[3] Stanford University

Abstract. We propose a new method for learning the structure of convolutional neural networks (CNNs) that is more efficient than recent state-of-the-art methods based on reinforcement learning and evolutionary algorithms. Our approach uses a sequential model-based optimization (SMBO) strategy, in which we search for structures in order of increasing complexity, while simultaneously learning a surrogate model to guide the search through structure space. Direct comparison under the same search space shows that our method is up to 5 times more efficient than the RL method of Zoph et al. (2018) in terms of number of models evaluated, and 8 times faster in terms of total compute. The structures we discover in this way achieve state of the art classification accuracies on CIFAR-10 and ImageNet.

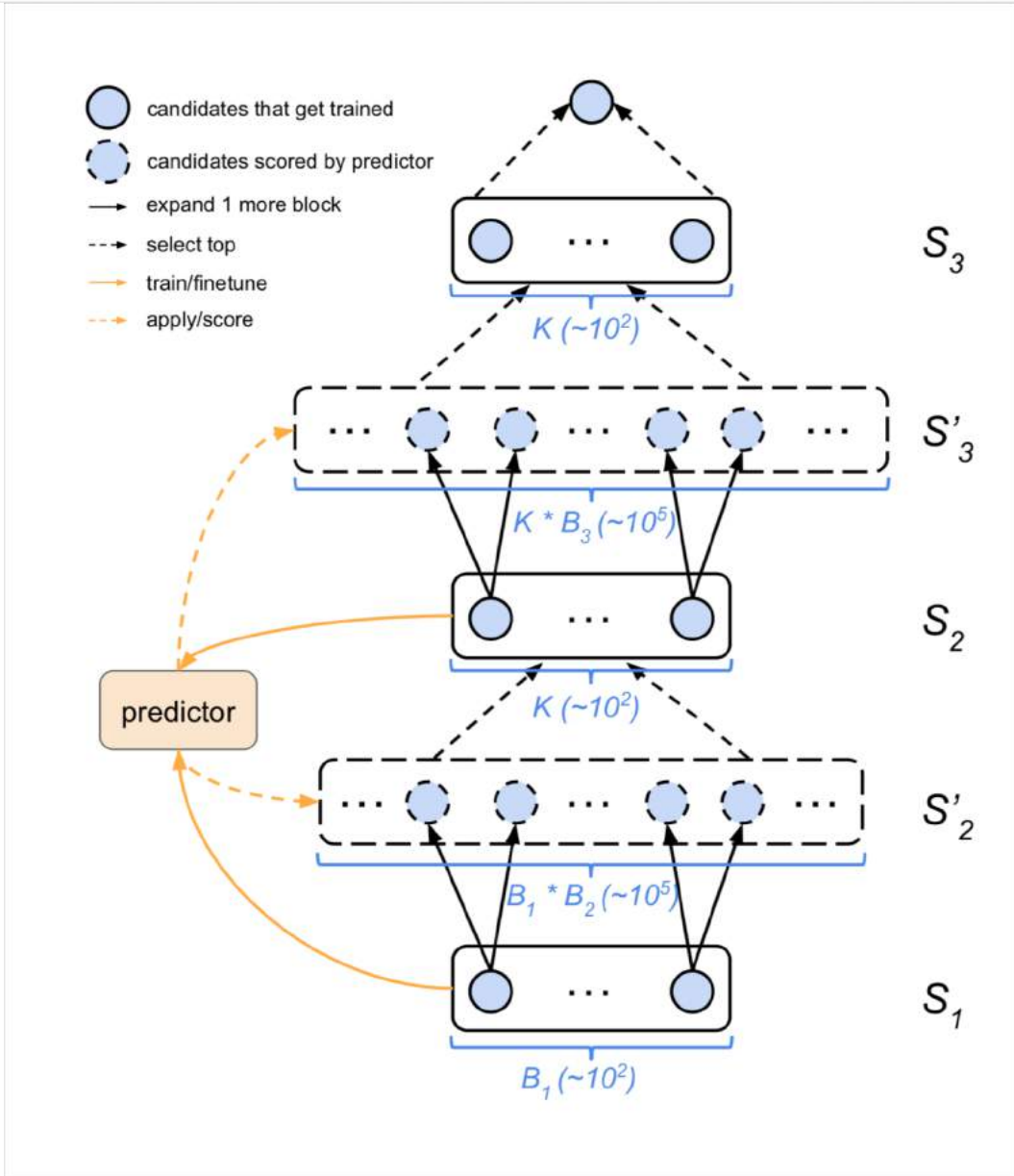
1 Introduction

There has been a lot of recent interest in automatically learning good neural net architectures. Some of this work is summarized in Section 2, but at a high level, current techniques usually fall into one of two categories: evolutionary algorithms (see e.g. [28,24,35]) or reinforcement learning (see e.g., [40,41,39,5,2]). When using evolutionary algorithms (EA), each neural network structure is en-

PNAS .. Progressive Neural Architecture Search

The main contribution of this work is to show how we can accelerate the search for good CNN structures by using progressive search through the space of increasingly complex graphs, combined with a learned prediction function to efficiently identify the most promising models to explore. The resulting models achieve the same level of performance as previous work but with a fraction of the computational cost.

Progressive Neural Architecture Search



Progressive Neural Architecture Search

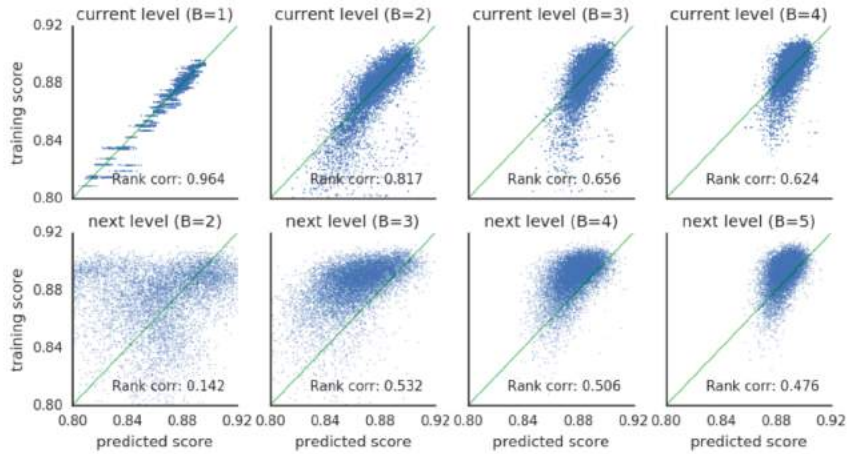


Fig. 3. Accuracy of MLP-ensemble predictor. Top row: true vs predicted accuracies on models from the training set over different trials. Bottom row: true vs predicted accuracies on models from the set of all unseen larger models. Denoted is the mean rank correlation from individual trials.

	$b=1$		$b=2$		$b=3$		$b=4$	
Method	$\hat{\rho}_1$	$\tilde{\rho}_2$	$\hat{\rho}_2$	$\tilde{\rho}_3$	$\hat{\rho}_3$	$\tilde{\rho}_4$	$\hat{\rho}_4$	$\tilde{\rho}_5$
MLP	0.938	0.113	0.857	0.450	0.714	0.469	0.641	0.444
RNN	0.970	**0.198**	**0.996**	0.424	0.693	0.401	**0.787**	0.413
MLP-ensemble	**0.975**	0.164	0.786	**0.532**	0.634	**0.504**	0.645	**0.468**
RNN-ensemble	0.972	0.164	0.906	0.418	**0.801**	0.465	0.579	0.424

Table 1. Spearman rank correlations of different predictors on the training set, $\hat{\rho}_b$, and when extrapolating to unseen larger models, $\tilde{\rho}_{b+1}$. See text for details.

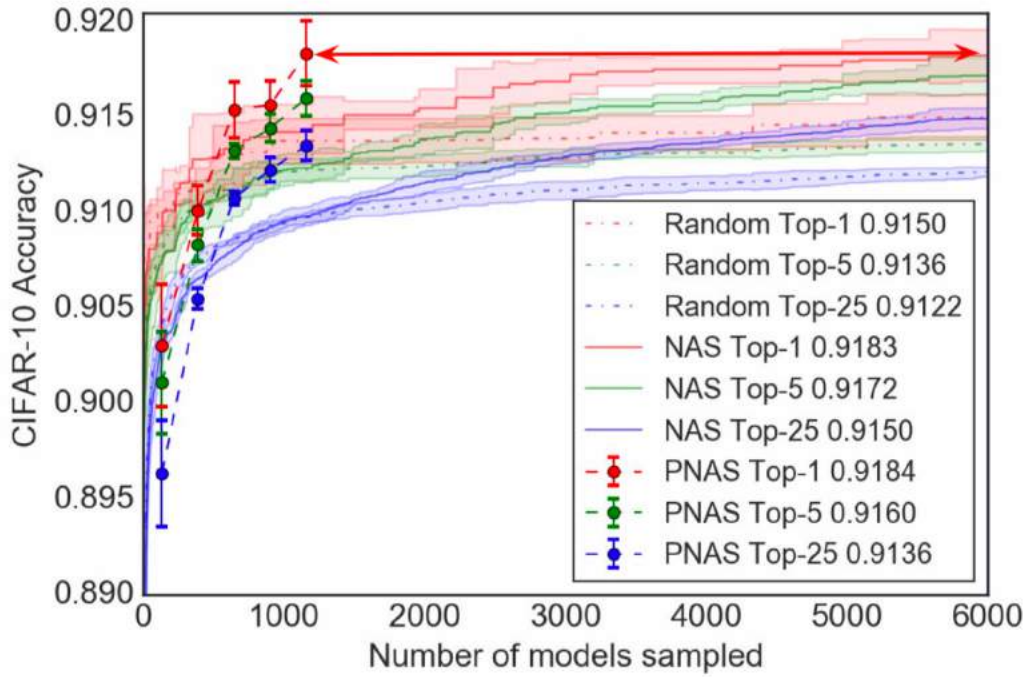


Fig. 4. Comparing the relative efficiency of NAS, PNAS and random search under the same search space. We plot mean accuracy (across 5 trials) on CIFAR-10 validation set of the top M models, for $M \in \{1, 5, 25\}$, found by each method vs number of models which are trained and evaluated. Each model is trained for 20 epochs. Error bars and the colored regions denote standard deviation of the mean.

B	Top	Accuracy	# PNAS	# NAS	Speedup (# models)	Speedup (# examples)
5	1	0.9183	1160	5808	5.0	8.2
5	5	0.9161	1160	4100	3.5	6.8
5	25	0.9136	1160	3654	3.2	6.4

Table 2. Relative efficiency of PNAS (using MLP-ensemble predictor) and NAS under the same search space. B is the size of the cell, "Top" is the number of top models we pick, "Accuracy" is their average validation accuracy, "# PNAS" is the number of models evaluated by PNAS, "# NAS" is the number of models evaluated by NAS to achieve the desired accuracy. Speedup measured by number of examples is greater than speedup in terms of number of models, because NAS has an additional reranking stage, that trains the top 250 models for 300 epochs each before picking the best one.

Model	B	N	F	Error	Params	M_1	E_1	M_2	E_2	Cost
NASNet-A [41]	5	6	32	3.41	3.3M	20000	0.9M	250	13.5M	21.4-29.3B
NASNet-B [41]	5	4	N/A	3.73	2.6M	20000	0.9M	250	13.5M	21.4-29.3B
NASNet-C [41]	5	4	N/A	3.59	3.1M	20000	0.9M	250	13.5M	21.4-29.3B
Hier-EA [21]	5	2	64	3.75±0.12	15.7M	7000	5.12M	0	0	35.8B[9]
AmoebaNet-B [27]	5	6	36	3.37±0.04	2.8M	27000	2.25M	100	27M	63.5B[10]
AmoebaNet-A [27]	5	6	36	3.34±0.06	3.2M	20000	1.13M	100	27M	25.2B[11]
PNASNet-5	5	3	48	3.41±0.09	3.2M	1160	0.9M	0	0	1.0B

Table 3. Performance of different CNNs on CIFAR test set. All model comparisons employ a comparable number of parameters and exclude cutout data augmentation [9]. "Error" is the top-1 misclassification rate on the CIFAR-10 test set. (Error rates have the form $\mu \pm \sigma$, where μ is the average over multiple trials and σ is the standard deviation. In PNAS we use 15 trials.) "Params" is the number of model parameters. "Cost" is the total number of examples processed through SGD ($M_1 E_1 + M_2 E_2$) before the architecture search terminates. The number of filters F for NASNet-{B, C} cannot be determined (hence N/A), and the actual E_1, E_2 may be larger than the values in this table (hence the range in cost), according to the original authors.

Efficient NAS

Efficient Neural Architecture Search via Parameter Sharing

Hieu Pham [*1 2] **Melody Y. Guan** [*3] **Barret Zoph** [1] **Quoc V. Le** [1] **Jeff Dean** [1]

Abstract

We propose *Efficient Neural Architecture Search* (ENAS), a fast and inexpensive approach for automatic model design. In ENAS, a controller discovers neural network architectures by searching for an optimal subgraph within a large computational graph. The controller is trained with policy gradient to select a subgraph that maximizes the expected reward on a validation set. Meanwhile the model corresponding to the selected subgraph is trained to minimize a canonical cross entropy loss. Sharing parameters among child models allows ENAS to deliver strong empirical performances, while using much fewer GPU-hours than existing automatic model design approaches, and notably, 1000x less expensive than standard Neural Architecture Search. On the Penn Treebank dataset, ENAS discovers a novel architecture that achieves a test perplexity of 55.8, establishing a new state-of-the-art among

spite its impressive empirical performance, NAS is computationally expensive and time consuming, *e.g.* Zoph et al. (2018) use 450 GPUs for 3-4 days (*i.e.* 32,400-43,200 GPU hours). Meanwhile, using less resources tends to produce less compelling results (Negrinho & Gordon, 2017; Baker et al., 2017a). We observe that the computational bottleneck of NAS is the training of each child model to convergence, only to measure its accuracy whilst throwing away all the trained weights.

The main contribution of this work is to improve the efficiency of NAS by *forcing all child models to share weights* to eschew training each child model from scratch to convergence. The idea has apparent complications, as different child models might utilize their weights differently, but was encouraged by previous work on transfer learning and multitask learning, which established that parameters learned for a particular model on a particular task can be used for other models on other tasks, with little to no modifications (Razavian et al., 2014; Zoph et al., 2016; Luong et al., 2016).

ENAS

Efficient Neural Architecture Search via Parameter Sharing

The main contribution of this work is to improve the efficiency of NAS by *forcing all child models to share weights* to eschew training each child model from scratch to convergence. The idea has apparent complications, as different child models might utilize their weights differently, but was encouraged by previous work on transfer learning and multitask learning, which established that parameters learned for a particular model on a particular task can be used for other models on other tasks, with little to no modifications (Razavian et al., 2014; Zoph et al., 2016; Luong et al., 2016).

Efficient Neural Architecture Search via Parameter Sharing

Efficient Neural Architecture Search via Parameter Sharing

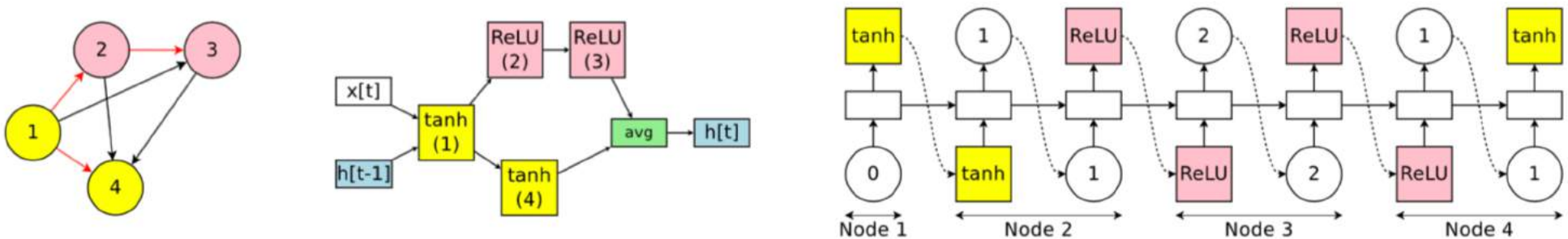


Figure 1. An example of a recurrent cell in our search space with 4 computational nodes. *Left:* The computational DAG that corresponds to the recurrent cell. The red edges represent the flow of information in the graph. *Middle:* The recurrent cell. *Right:* The outputs of the controller RNN that result in the cell in the middle and the DAG on the left. Note that nodes 3 and 4 are never sampled by the RNN, so their results are averaged and are treated as the cell's output.

2.3. Designing Convolutional Networks

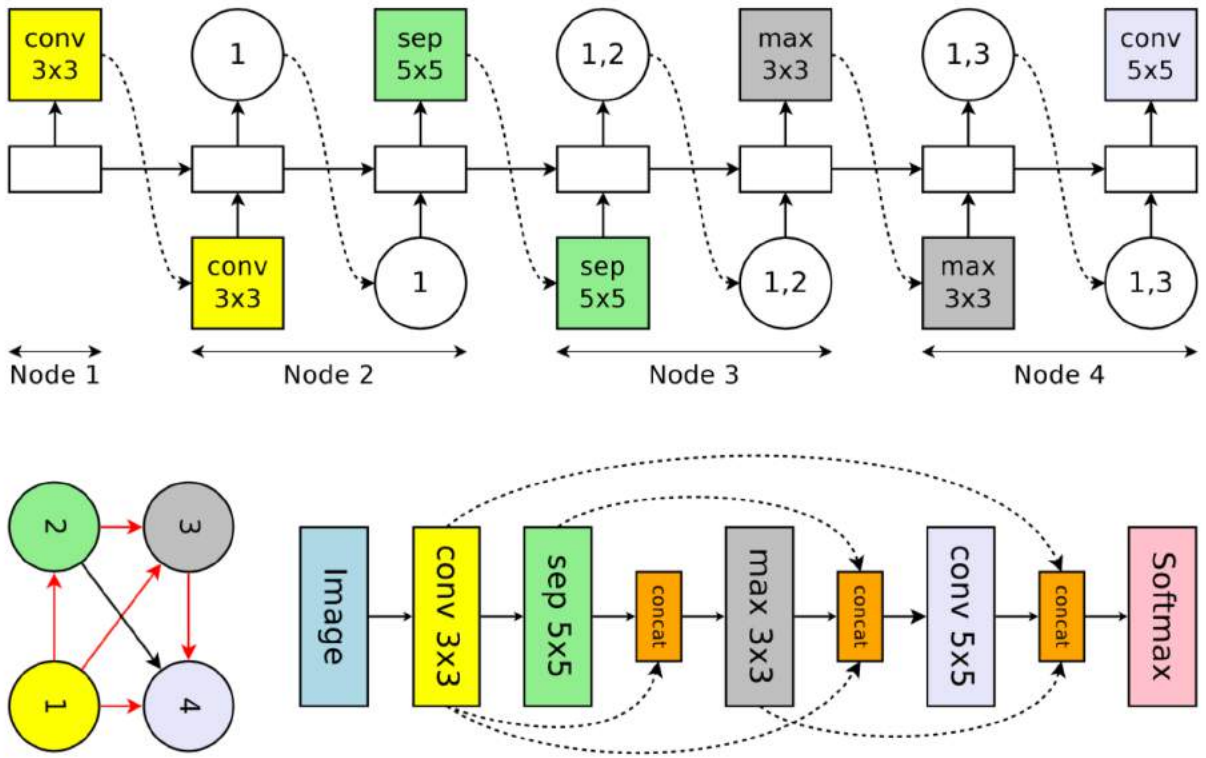


Figure 3. An example run of a recurrent cell in our search space with 4 computational nodes, which represent 4 layers in a convolutional network. *Top:* The output of the controller RNN. *Bottom Left:* The computational DAG corresponding to the network's architecture. Red arrows denote the active computational paths. *Bottom Right:* The complete network. Dotted arrows denote skip connections.

Efficient Neural Architecture Search via Parameter Sharing

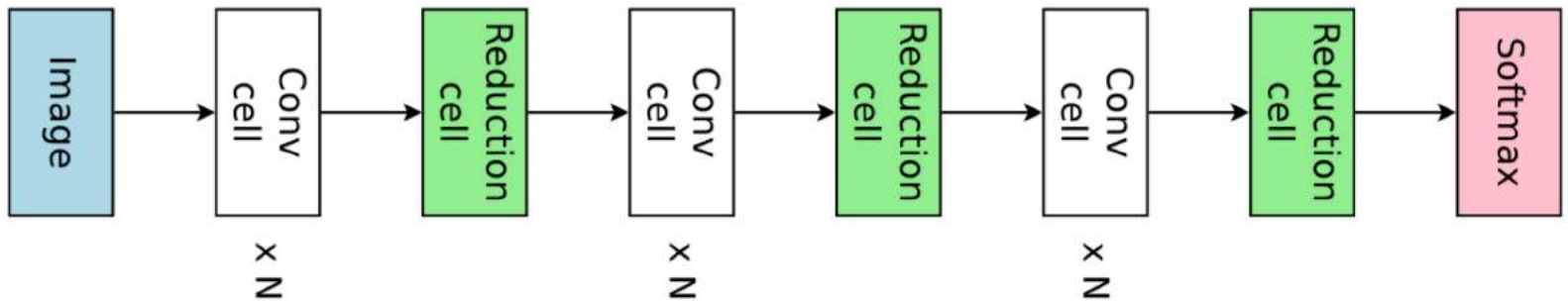


Figure 4. Connecting 3 blocks, each with N convolution cells and 1 reduction cell, to make the final network.

Efficient Neural Architecture Search via Parameter Sharing

Importantly, ENAS cell outperforms NAS (Zoph & Le, 2017) by more than 6 perplexity points, whilst the search process of ENAS, in terms of GPU hours, is more than 1000x faster.

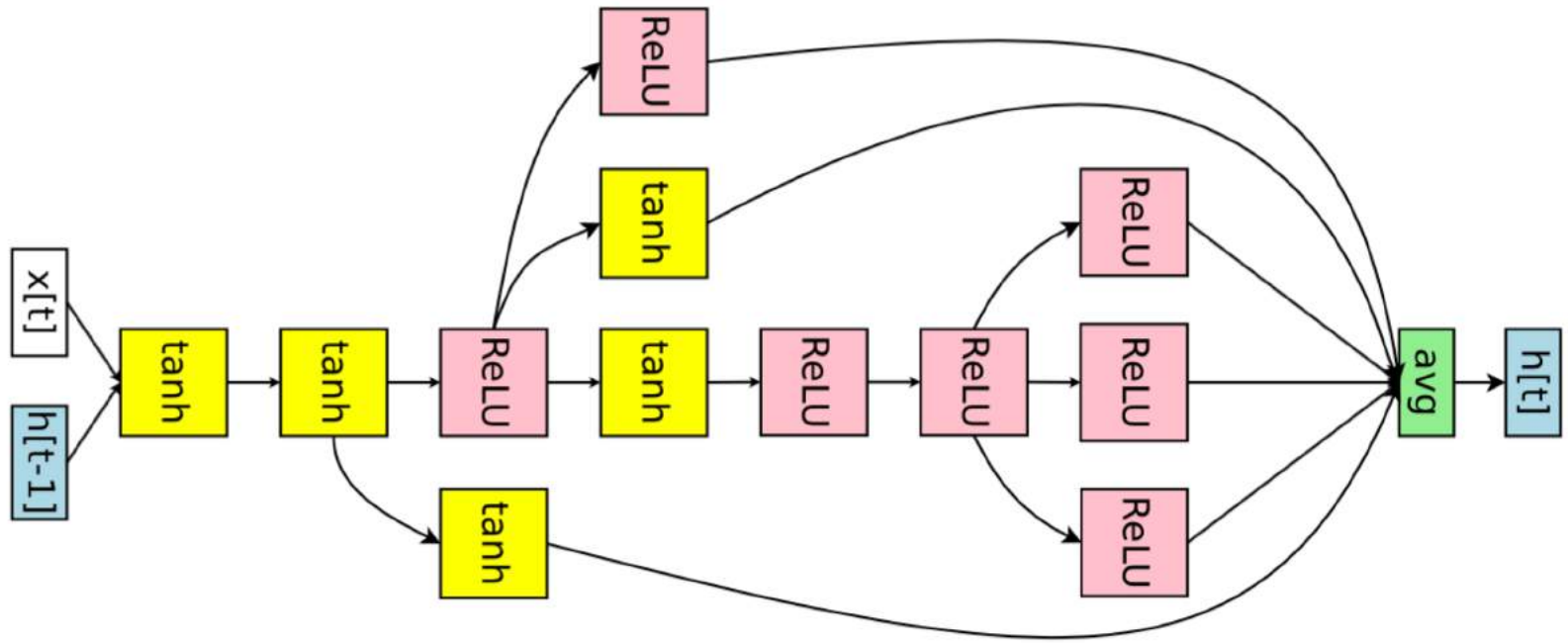


Figure 6. The RNN cell ENAS discovered for Penn Treebank.

Efficient Neural Architecture Search via Parameter Sharing

Architecture	Additional Techniques	Params (million)	Test PPL
LSTM (Zaremba et al., 2014)	Vanilla Dropout	66	78.4
LSTM (Gal & Ghahramani, 2016)	VD	66	75.2
LSTM (Inan et al., 2017)	VD, WT	51	68.5
LSTM (Melis et al., 2017)	Hyper-parameters Search	24	59.5
LSTM (Yang et al., 2018)	VD, WT, ℓ_2, AWD, MoC	22	57.6
LSTM (Merity et al., 2017)	VD, WT, ℓ_2, AWD	24	57.3
LSTM (Yang et al., 2018)	VD, WT, ℓ_2, AWD, MoS	**22**	**56.0**
RHN (Zilly et al., 2017)	VD, WT	24	66.0
NAS (Zoph & Le, 2017)	VD, WT	54	62.4
ENAS	VD, WT, ℓ_2	**24**	**55.8**

Table 1. Test perplexity on Penn Treebank of ENAS and other baselines. Abbreviations: RHN is *Recurrent Highway Network*, VD is *Variational Dropout*; WT is *Weight Tying*; ℓ_2 is *Weight Penalty*; AWD is *Averaged Weight Drop*; MoC is *Mixture of Contexts*; MoS is *Mixture of Softmaxes*.

Method	GPUs	Times (days)	Params (million)	Error (%)
DenseNet-BC (Huang et al., 2016)	–	–	25.6	3.46
DenseNet + Shake-Shake (Gastaldi, 2016)	–	–	26.2	2.86
DenseNet + CutOut (DeVries & Taylor, 2017)	–	–	26.2	**2.56**
Budgeted Super Nets (Veniat & Denoyer, 2017)	–	–	–	9.21
ConvFabrics (Saxena & Verbeek, 2016)	–	–	21.2	7.43
Macro NAS + Q-Learning (Baker et al., 2017a)	10	8-10	11.2	6.92
Net Transformation (Cai et al., 2018)	5	2	19.7	5.70
FractalNet (Larsson et al., 2017)	–	–	38.6	4.60
SMASH (Brock et al., 2018)	1	1.5	16.0	4.03
NAS (Zoph & Le, 2017)	800	21-28	7.1	4.47
NAS + more filters (Zoph & Le, 2017)	800	21-28	37.4	**3.65**
ENAS + macro search space	1	0.32	21.3	4.23
ENAS + macro search space + more channels	1	0.32	38.0	**3.87**
Hierarchical NAS (Liu et al., 2018)	200	1.5	61.3	3.63
Micro NAS + Q-Learning (Zhong et al., 2018)	32	3	–	3.60
Progressive NAS (Liu et al., 2017)	100	1.5	3.2	3.63
NASNet-A (Zoph et al., 2018)	450	3-4	3.3	3.41
NASNet-A + CutOut (Zoph et al., 2018)	450	3-4	3.3	**2.65**
ENAS + micro search space	1	0.45	4.6	3.54
ENAS + micro search space + CutOut	1	0.45	4.6	**2.89**

Table 2. Classification errors of ENAS and baselines on CIFAR-10. In this table, the first block presents DenseNet, one of the state-of-the-art architectures designed by human experts. The second block presents approaches that design the entire network. The last block presents techniques that design modular cells which are combined to build the final network.

Efficient Neural Architecture Search via Parameter Sharing

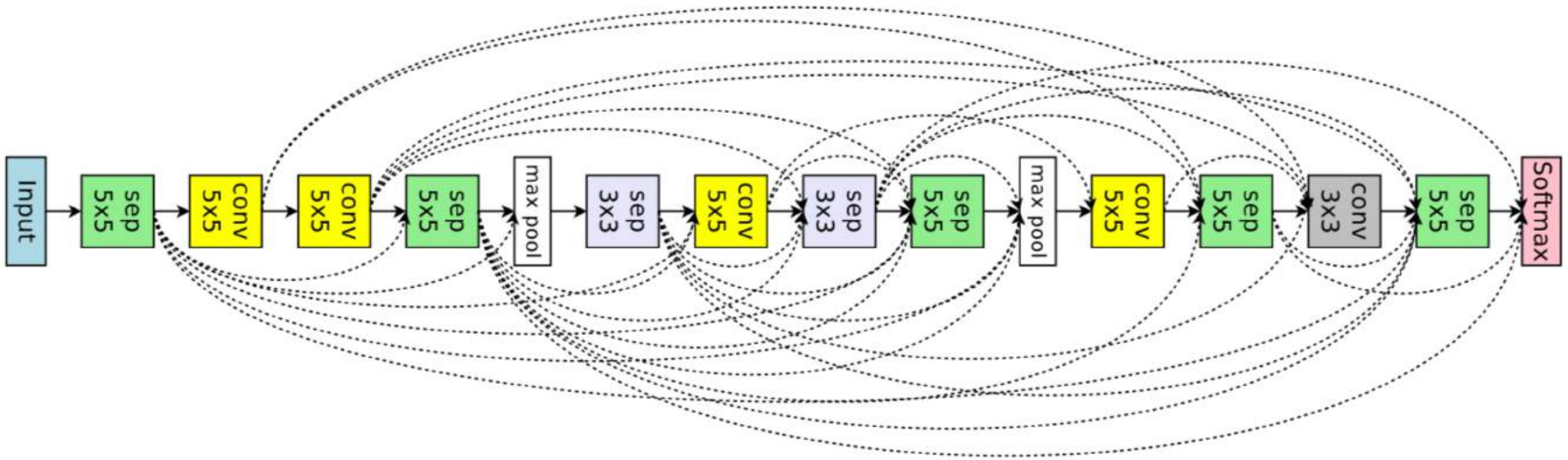


Figure 7. ENAS's discovered network from the macro search space for image classification.

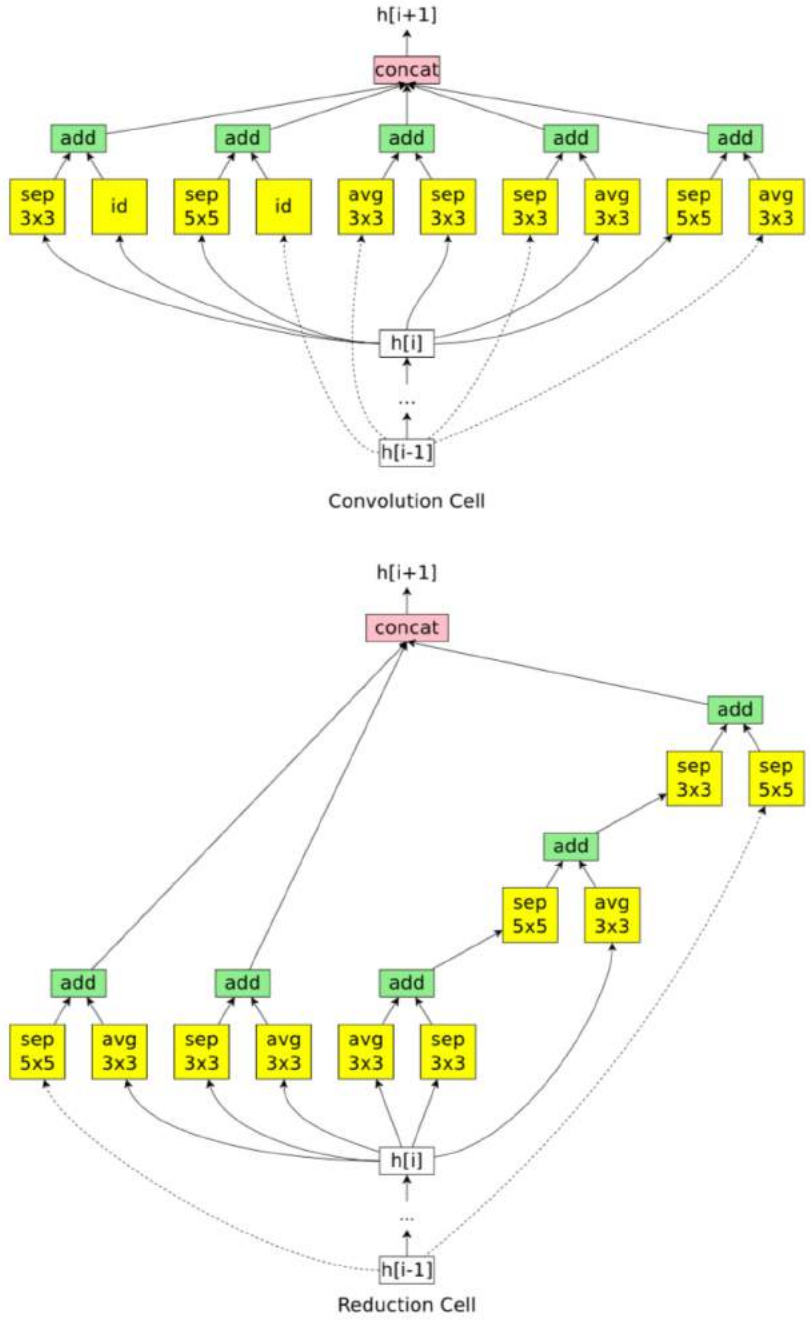
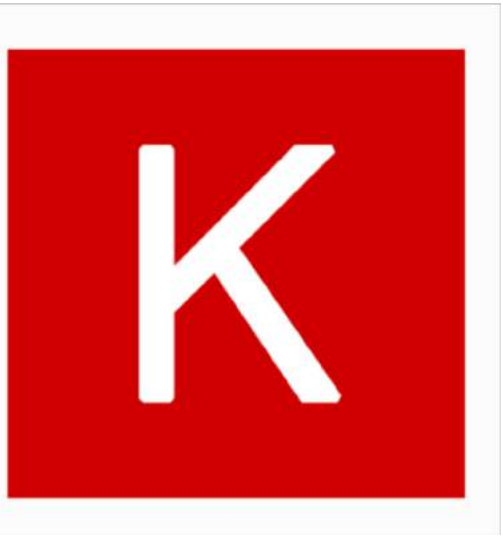


Figure 8. ENAS cells discovered in the micro search space.

Auto-Keras is an open source software library for automated machine learning (AutoML). It is developed by DATA Lab at Texas A&M University and community contributors. The ultimate goal of AutoML is to provide easily accessible deep learning tools to domain experts with limited data science or machine learning background. Auto-Keras provides functions to automatically search for architecture and hyperparameters of deep learning models.



Installation

To install the package, please use the `pip` installation as follows:

```
pip install autokeras
```

Note: currently, Auto-Keras is only compatible with: **Python 3.6**.

Example

Here is a short example of using the package.

```
import autokeras as ak

clf = ak.ImageClassifier()
clf.fit(x_train, y_train)
results = clf.predict(x_test)
```

melodyguan / enas

Watch 62 | Star 827 | Fork 194

Code | Issues 48 | Pull requests 5 | Projects 0 | Wiki | Insights

TensorFlow Code for paper "Efficient Neural Architecture Search via Parameter Sharing" https://arxiv.org/abs/1802.03268

27 commits | 2 branches | 0 releases | 2 contributors | Apache-2.0

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

hyhieu Update README.md — Latest commit d1a90ac on Jun 7

data/ptb	open source	7 months ago
img	added enas cell image	7 months ago
scripts	open source	7 months ago
src	fixed to update prev_c, prev_h in the controller	7 months ago
.gitignore	fixed syntax errors	7 months ago
LICENSE	added LICENSE	7 months ago
README.md	Update README.md	5 months ago

README.md

Efficient Neural Architecture Search via Parameter Sharing

Authors' implementation of "Efficient Neural Architecture Search via Parameter Sharing" (2018) in TensorFlow.

Includes code for CIFAR-10 image classification and Penn Tree Bank language modeling tasks.

Paper: https://arxiv.org/abs/1802.03268

Authors: Hieu Pham*, Melody Y. Guan*, Barret Zoph, Quoc V. Le, Jeff Dean

This is not an official Google product.

carpedm20 / ENAS-pytorch

Watch 80 | Star 1,264 | Fork 225

Code | Issues 21 | Pull requests 0 | Projects 0 | Wiki | Insights

PyTorch implementation of "Efficient Neural Architecture Search via Parameters Sharing"

pytorch | neural-architecture-search | google-brain

47 commits | 1 branch | 0 releases | 3 contributors | Apache-2.0

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

carpedm20 Merge pull request #35 from nkcr/patch-1 ... Latest commit ed2232d 2 days ago

assets	add best_rnn_epoch27.png	9 months ago
data	Partial initial cleanup, working towards PyTorch 0.4 bug in cell.	9 months ago
models	Moves the Node definition to utils.py (typo fix)	5 days ago
.gitignore	initial commit	9 months ago
LICENSE	update LICENSE	8 months ago
README.md	add link for official code	7 months ago
config.py	Adds a mode (single) and --dag_path	5 days ago
dag.json	json representation of the best dag from the paper	5 days ago
generate_gif.py	initial commit	9 months ago
main.py	Parses the "single" mode	5 days ago
requirements.txt	fix dimension and dropout bug during the test()	9 months ago
run.sh	add summary of relative reward per batch and more configs	9 months ago
tensorboard.py	initial commit	9 months ago
trainer.py	Adds parameters to use a single dag in single mode	5 days ago
utils.py	Logs when a dag is saved or loaded	5 days ago

README.md

Google’s CloudML AutoML

Cloud AutoML BETA

Train high-quality custom machine learning models with minimum effort and machine learning expertise.
Try AutoML Translation, Natural Language, and Vision, now in beta.

TRY FREE

CONTACT SALES

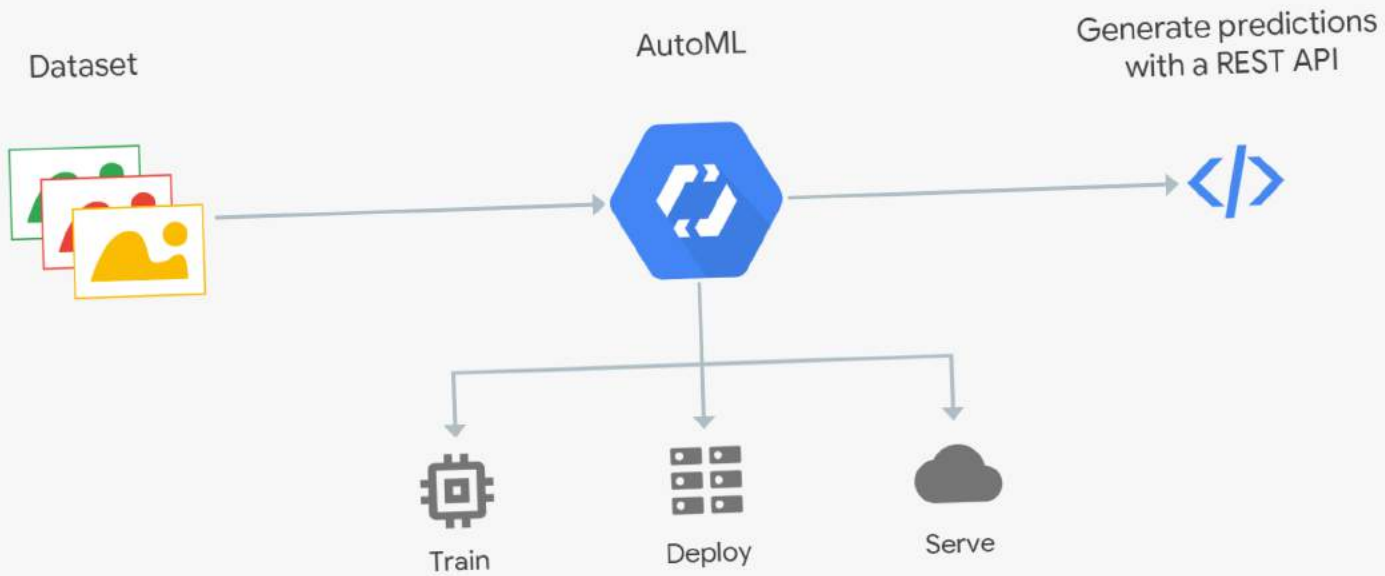
Train custom machine learning models

Cloud AutoML is a suite of machine learning products that enables developers with limited machine learning expertise to train high-quality models specific to their business needs, by leveraging Google's state-of-the-art transfer learning, and Neural Architecture Search technology.

State-of-the-art performance

Use Cloud AutoML to leverage Google's proprietary technology, which offers fast performance and accurate predictions. AutoML puts more than 10 years of Google Research technology in the hands of our users.

How AutoML works



Cloud AutoML Features

Train high-quality custom machine learning models with minimum effort and machine learning expertise.

Custom models

Train custom machine learning models that are specific to your business needs, with minimum effort and machine learning expertise.

Fully integrated

Cloud AutoML is fully integrated with other Google Cloud services, providing customers with a consistent method of access across the entire Google Cloud service line. Store your training data in Cloud Storage. To generate a prediction on your trained model, simply use the existing Vision API by adding a parameter for your custom model or use Cloud ML Engine's online prediction service.

Powered by Google's AutoML and Transfer Learning

Leverages Google state-of-the-art AutoML and Transfer Learning technology to produce high-quality models.

Integration with human labeling (AutoML Vision only)

For customers with images but no labels yet, we provide a team of in-house people that will review your custom instructions and classify your images accordingly. You will get high-quality training data, while your data remains private. You can use the human-labeled data seamlessly to train a custom model.