Neural Architecture Search: Foundations and Trends

Colin White
Abacus.AI
colin@abacus.ai

Debadeepta Dey
Microsoft Research
dedey@microsoft.com

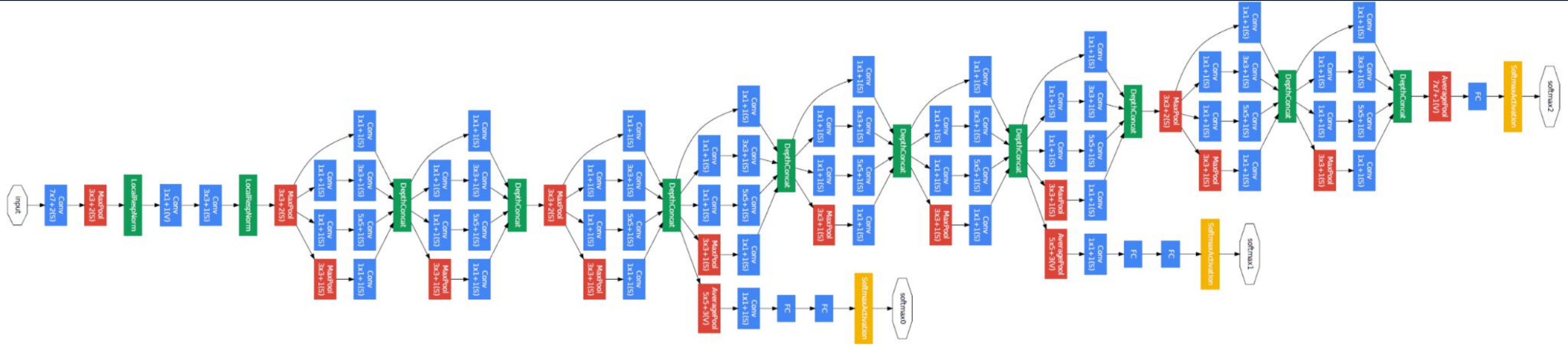
Slides (with hyperlinks): https://crwhite.ml/

Machine learning: a story of automation

```python
from torchvision.models import resnet50, ResNet50_Weights

# Best available weights (currently alias for IMAGENET1K_V2)
# Note that these weights may change across versions
resnet50(weights=ResNet50_Weights.DEFAULT)
```

Neural architecture search

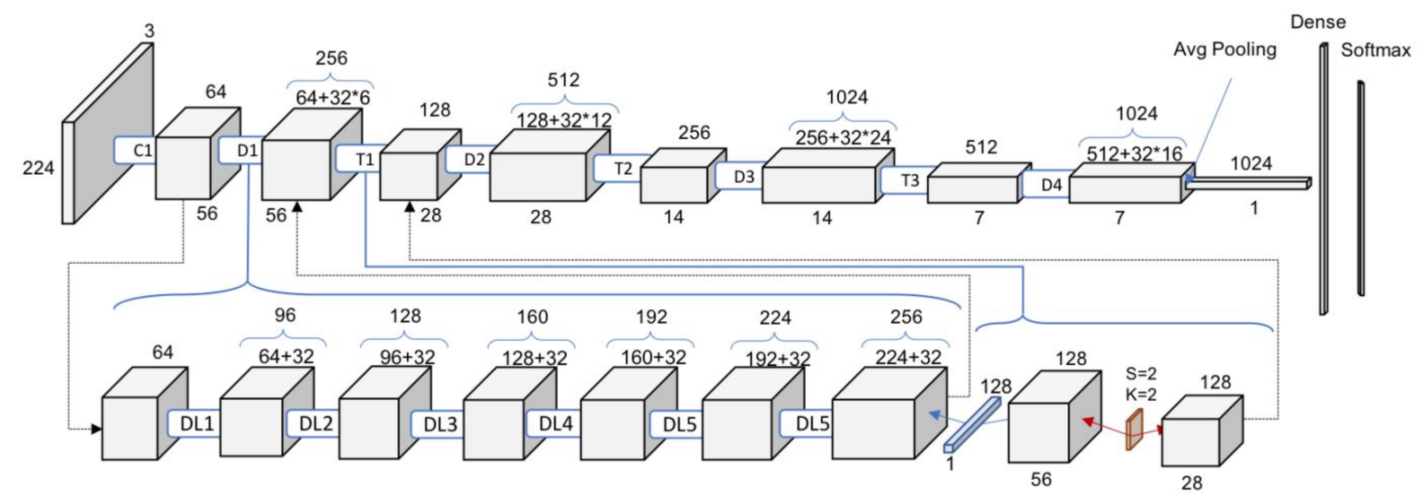


GoogLeNet (2014)

Architectures are getting increasingly more specialized and complex

Source: [GoogLeNet (2014)]

Neural architecture search

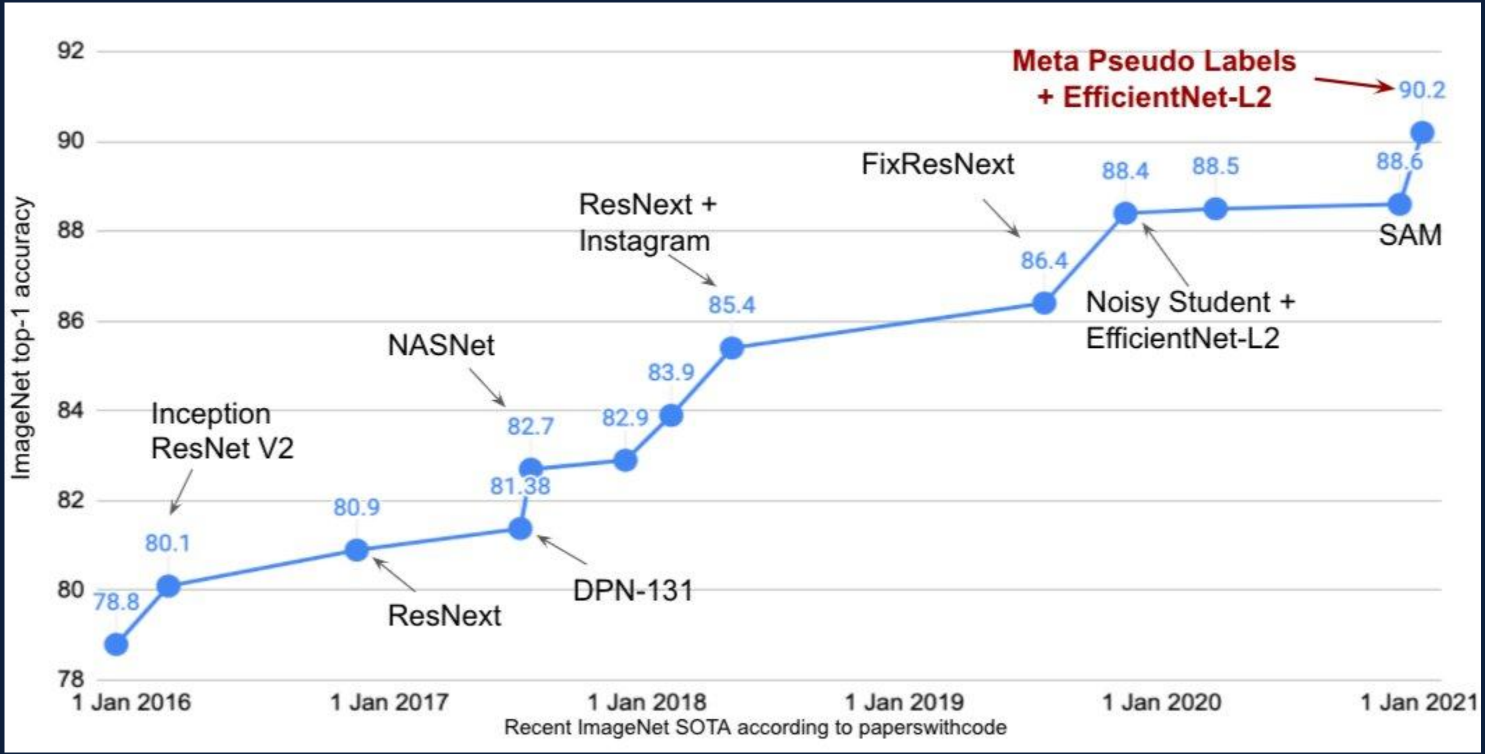


DenseNet (2016)

Architectures are getting increasingly more specialized and complex

Source: [DenseNet (2016)]

Neural architecture search



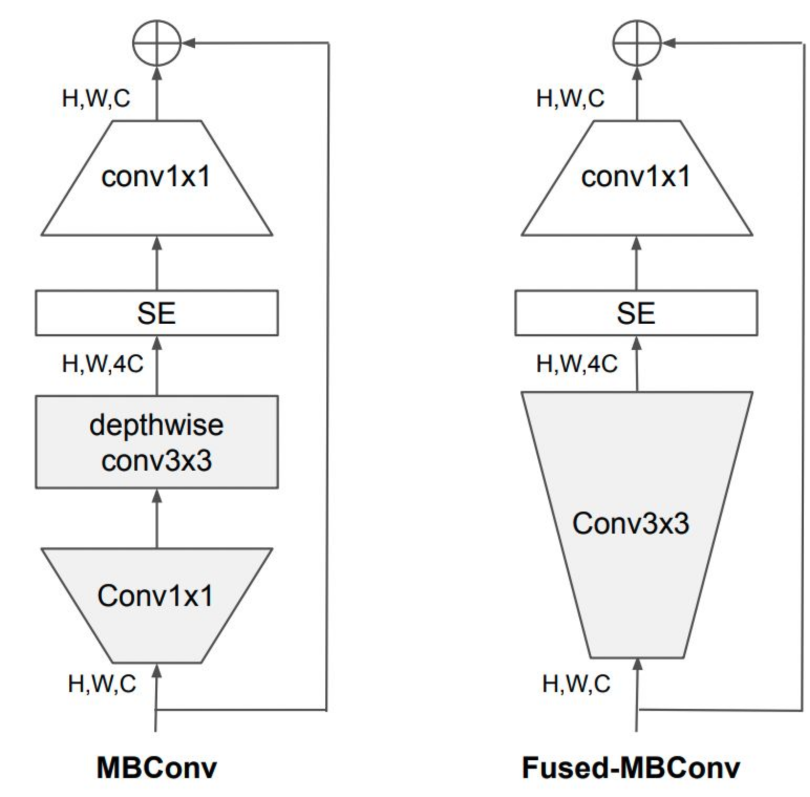
Searched models are replacing **human-designed** models

Source: https://twitter.com/quocleix/status/1349443438698143744/

Human + Neural architecture search

Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	224×224	32	1
2	MBConv1, k3x3	112×112	16	1
3	MBConv6, k3x3	112×112	24	2
4	MBConv6, k5x5	56×56	40	2
5	MBConv6, k3x3	28×28	80	3
6	MBConv6, k5x5	14×14	112	3
7	MBConv6, k5x5	14×14	192	4
8	MBConv6, k3x3	7×7	320	1
9	Conv1x1 & Pooling & FC	7×7	1280	1

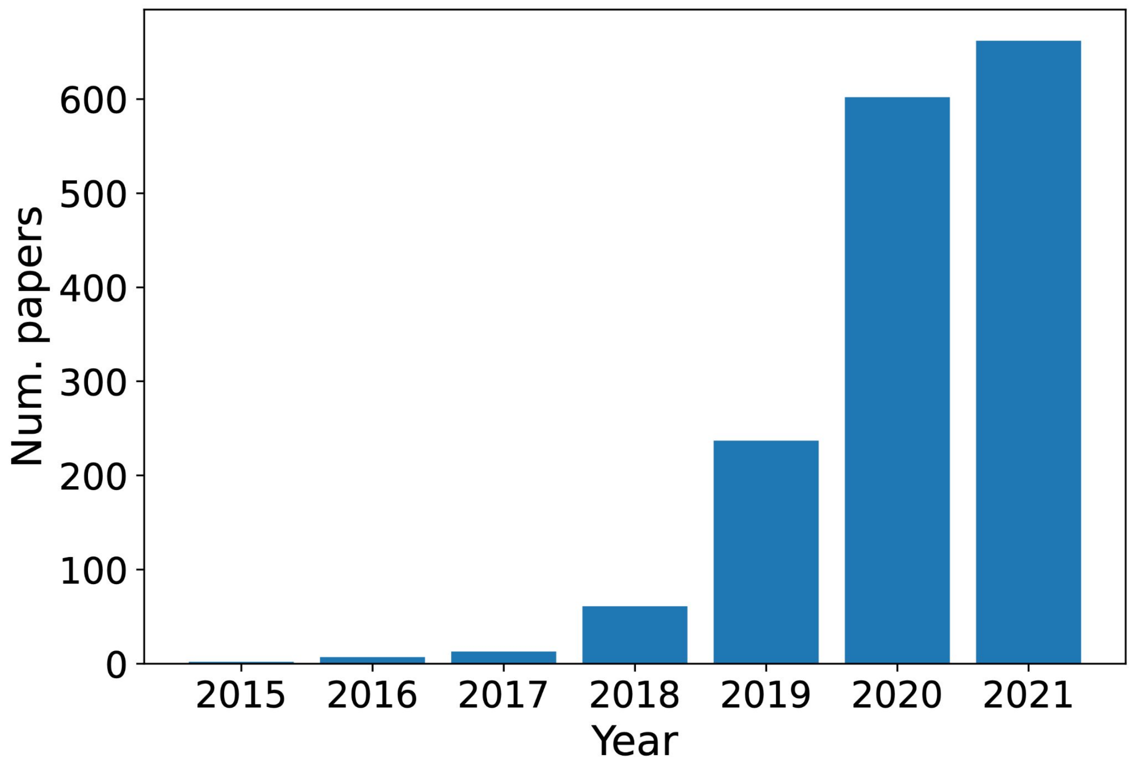
Stage	Operator	Stride	#Channels	#Layers
0	Conv3x3	2	24	1
1	Fused-MBConv1, k3x3	1	24	2
2	Fused-MBConv4, k3x3	2	48	4
3	Fused-MBConv4, k3x3	2	64	4
4	MBConv4, k3x3, SE0.25	2	128	6
5	MBConv6, k3x3, SE0.25	1	160	9
6	MBConv6, k3x3, SE0.25	2	256	15
7	Conv1x1 & Pooling & FC	-	1280	1



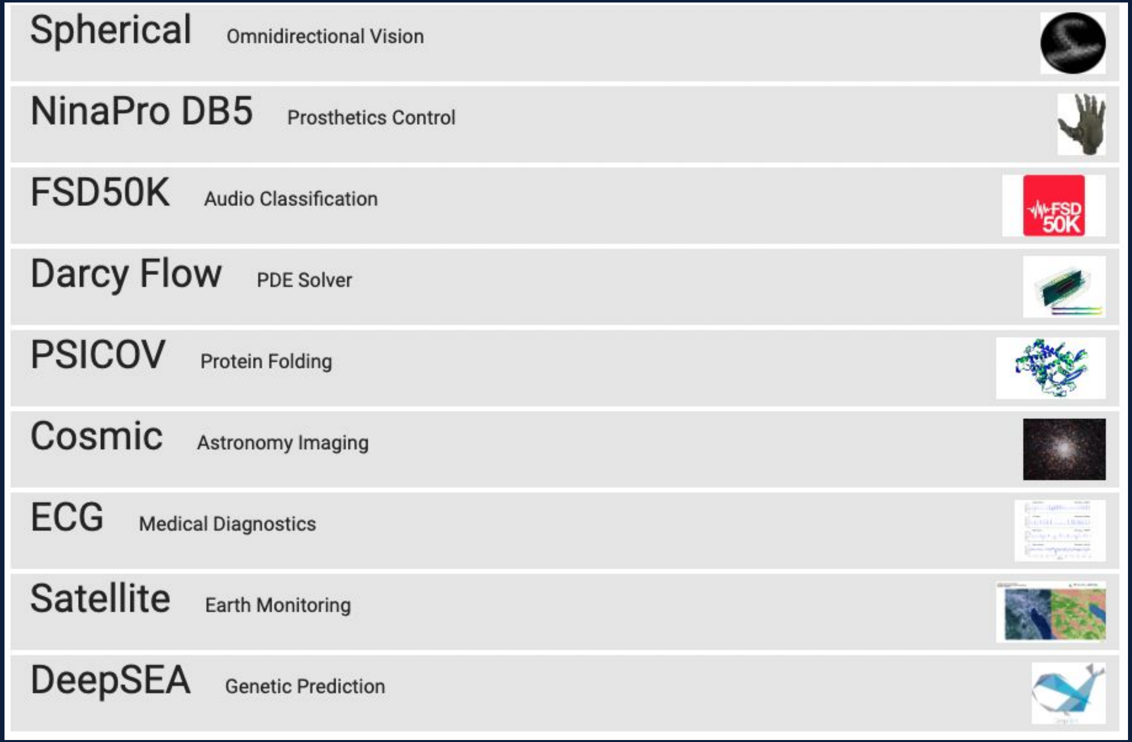
Source: EfficientNet (2019), EfficientNetV2 (2021)

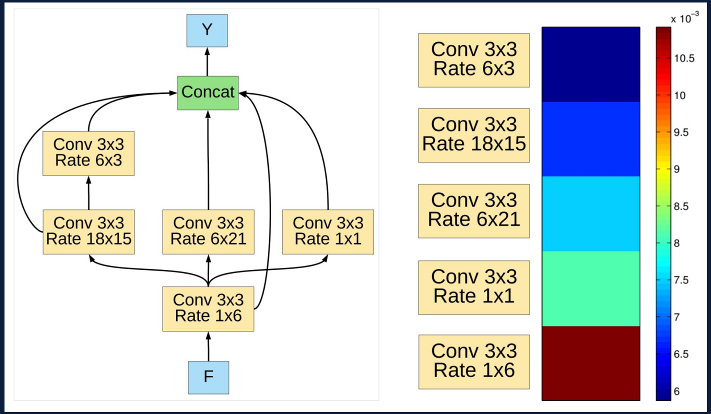
Neural architecture search

NAS: the process of **automating** the design of **neural architectures** for a given dataset.



New Datasets



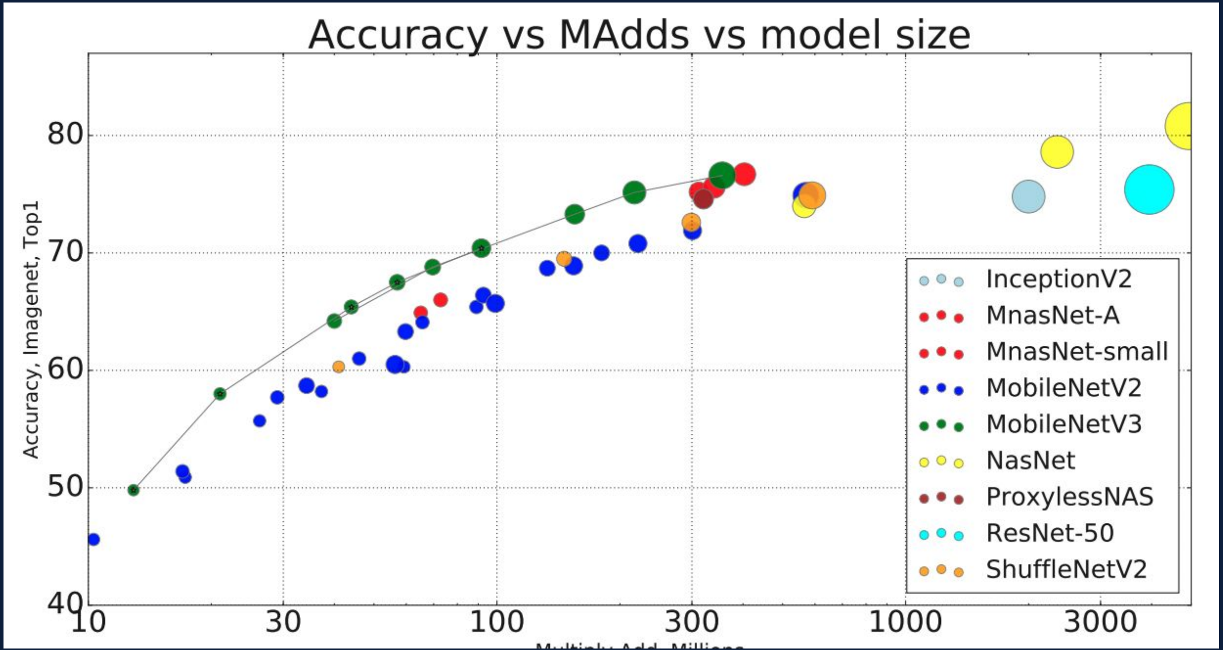


Graph neural networks . . . . .
Generative adversarial network
Dense prediction tasks . . . . .
Adversarial robustness . . . . .
Self-supervised learning for NAS

Source: NAS-Bench-360 (2021)

Chen et al. 2018

Fitting Models on Edge Devices



Input	Operator	exp size	$\#out$	SE	NL	s
$224^2 \times 3$	conv2d	-	16	-	HS	2
$112^2 \times 16$	bneck, 3x3	16	16	-	RE	1
$112^2 \times 16$	bneck, 3x3	64	24	-	RE	2
$56^2 \times 24$	bneck, 3x3	72	24	-	RE	1
$56^2 \times 24$	bneck, 5x5	72	40	✓	RE	2
$28^2 \times 40$	bneck, 5x5	120	40	✓	RE	1
$28^2 \times 40$	bneck, 5x5	120	40	✓	RE	1
$28^2 \times 40$	bneck, 3x3	240	80	-	HS	2
$14^2 \times 80$	bneck, 3x3	200	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	184	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	184	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	480	112	✓	HS	1
$14^2 \times 112$	bneck, 3x3	672	112	✓	HS	1
$14^2 \times 112$	bneck, 5x5	672	160	✓	HS	2
$7^2 \times 160$	bneck, 5x5	960	160	✓	HS	1
$7^2 \times 160$	bneck, 5x5	960	160	✓	HS	1
$7^2 \times 160$	conv2d, 1x1	-	960	-	HS	1
$7^2 \times 960$	pool, 7x7	-	-	-	-	1
$1^2 \times 960$	conv2d 1x1, NBN	-	1280	-	HS	1
$1^2 \times 1280$	conv2d 1x1, NBN	-	k	-	-	1

Input	Operator	exp size	$\#out$	SE	NL	s
$224^2 \times 3$	conv2d, 3x3	-	16	-	HS	2
$112^2 \times 16$	bneck, 3x3	16	16	✓	RE	2
$56^2 \times 16$	bneck, 3x3	72	24	-	RE	2
$28^2 \times 24$	bneck, 3x3	88	24	-	RE	1
$28^2 \times 24$	bneck, 5x5	96	40	✓	HS	2
$14^2 \times 40$	bneck, 5x5	240	40	✓	HS	1
$14^2 \times 40$	bneck, 5x5	240	40	✓	HS	1
$14^2 \times 40$	bneck, 5x5	120	48	✓	HS	1
$14^2 \times 48$	bneck, 5x5	144	48	✓	HS	1
$14^2 \times 48$	bneck, 5x5	288	96	✓	HS	2
$7^2 \times 96$	bneck, 5x5	576	96	✓	HS	1
$7^2 \times 96$	bneck, 5x5	576	96	✓	HS	1
$7^2 \times 96$	conv2d, 1x1	-	576	✓	HS	1
$7^2 \times 576$	pool, 7x7	-	-	-	-	1
$1^2 \times 576$	conv2d 1x1, NBN	-	1024	-	HS	1
$1^2 \times 1024$	conv2d 1x1, NBN	-	k	-	-	1

The best and most efficient architectures today are found automatically

Source: [MobileNetV3 (2019)]

Motivation - Summary

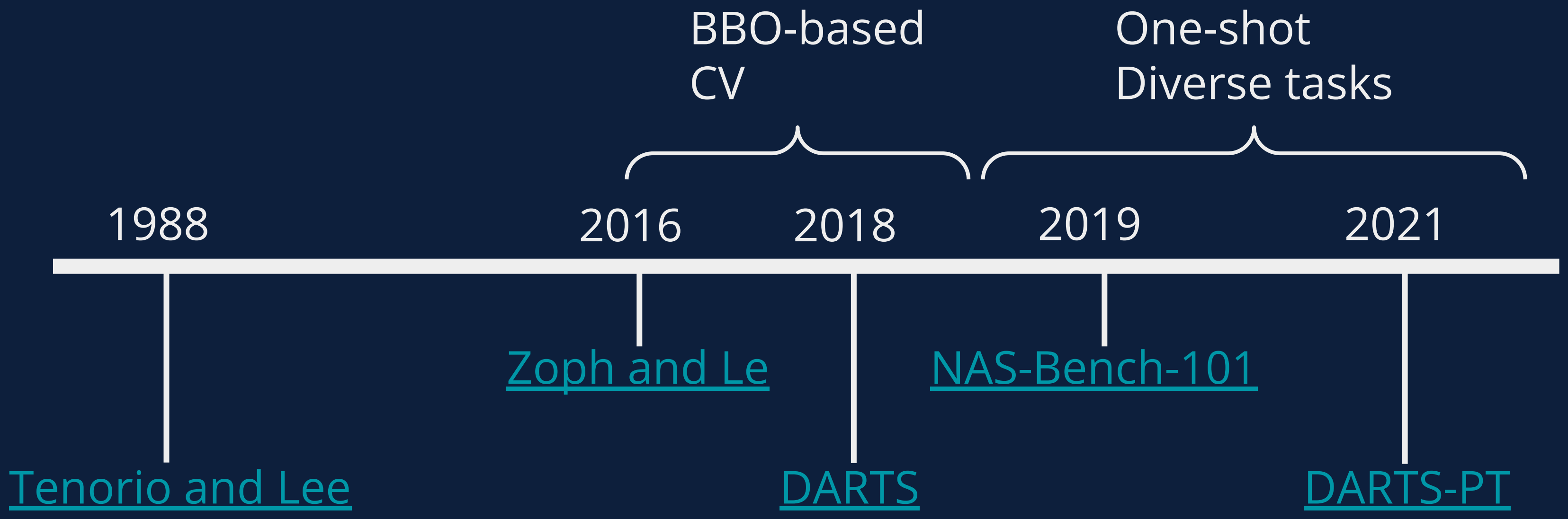
- Widely-used benchmarks
- New datasets
- Constrained / multi-objective problems

- Democratizing deep learning
  - Latest techniques are just a few GPU-hours

NAS: A History

Studied since at least the late 1980s

Resurgence in late 2016



NAS: Basic Definition

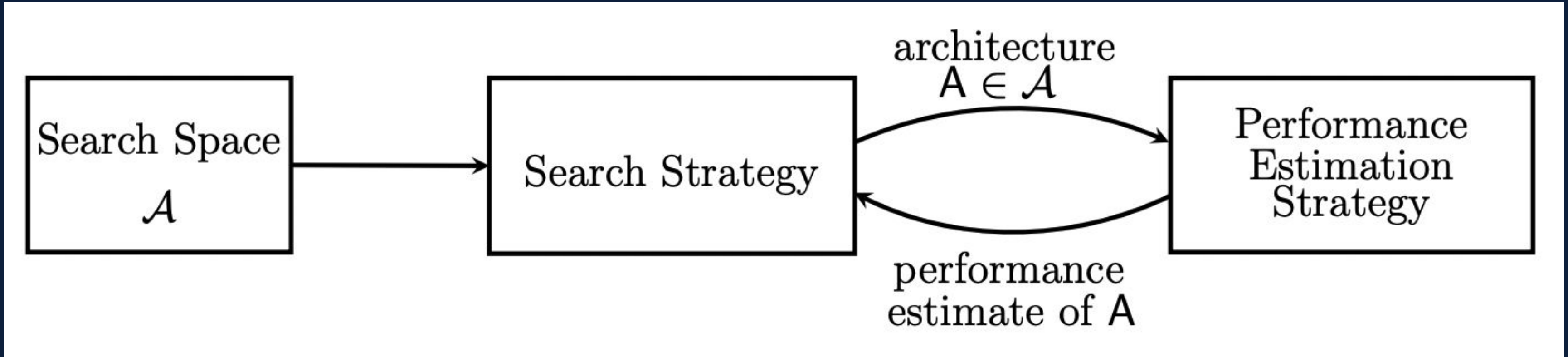
- Define a search space ***A,***

$$
\begin{aligned}
\min_{a \in \mathcal{A}} \quad & \mathcal{L}_{\text{val}}\left(w^*(a), a\right) \\
\text{s.t.} \quad & w^*(a) = \operatorname{argmin}_w \; \mathcal{L}_{\text{train}}\left(w, a\right)
\end{aligned}
$$

Three Pillars of NAS

- Search space
- Search strategy
- Performance estimation strategy

Coupled, for one-shot methods

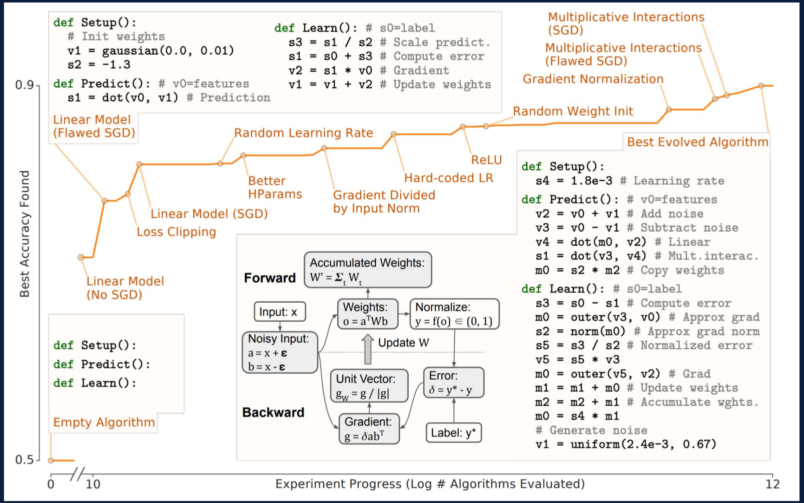


Elsken et al. (2018)

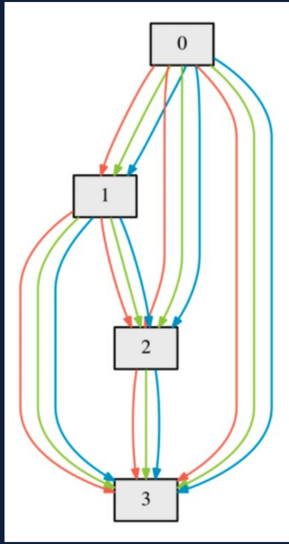
Roadmap - Part 1

- Motivation and introduction
- Search spaces
  - Macro
  - Cell-based
  - Hierarchical
  - Encodings
- Black-box optimization methods
  - Baselines
  - Bayesian optimization
  - Evolution
- Performance prediction

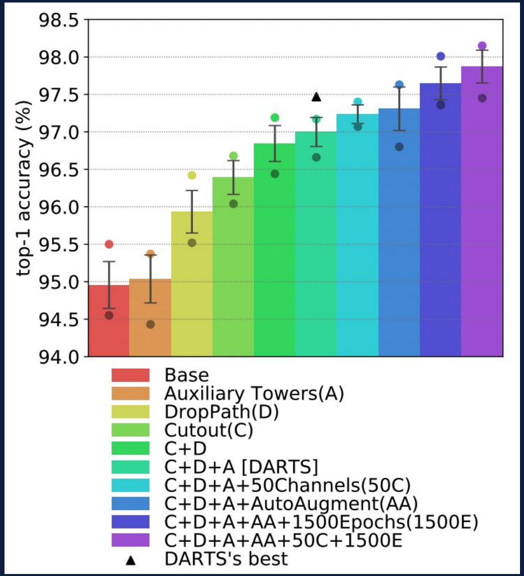
Search Spaces: Exploration vs. Exploitation



[AutoML-Zero (2020)]



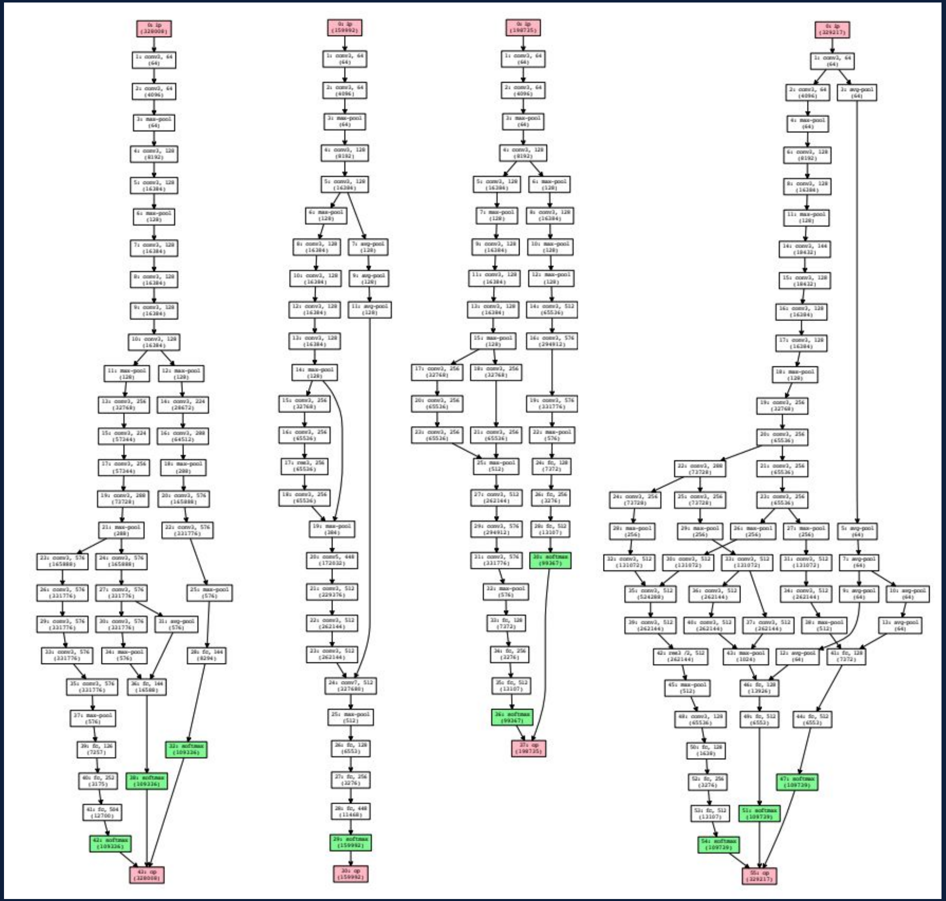
[DARTS (2018)]



[Yang et al. (2019)]

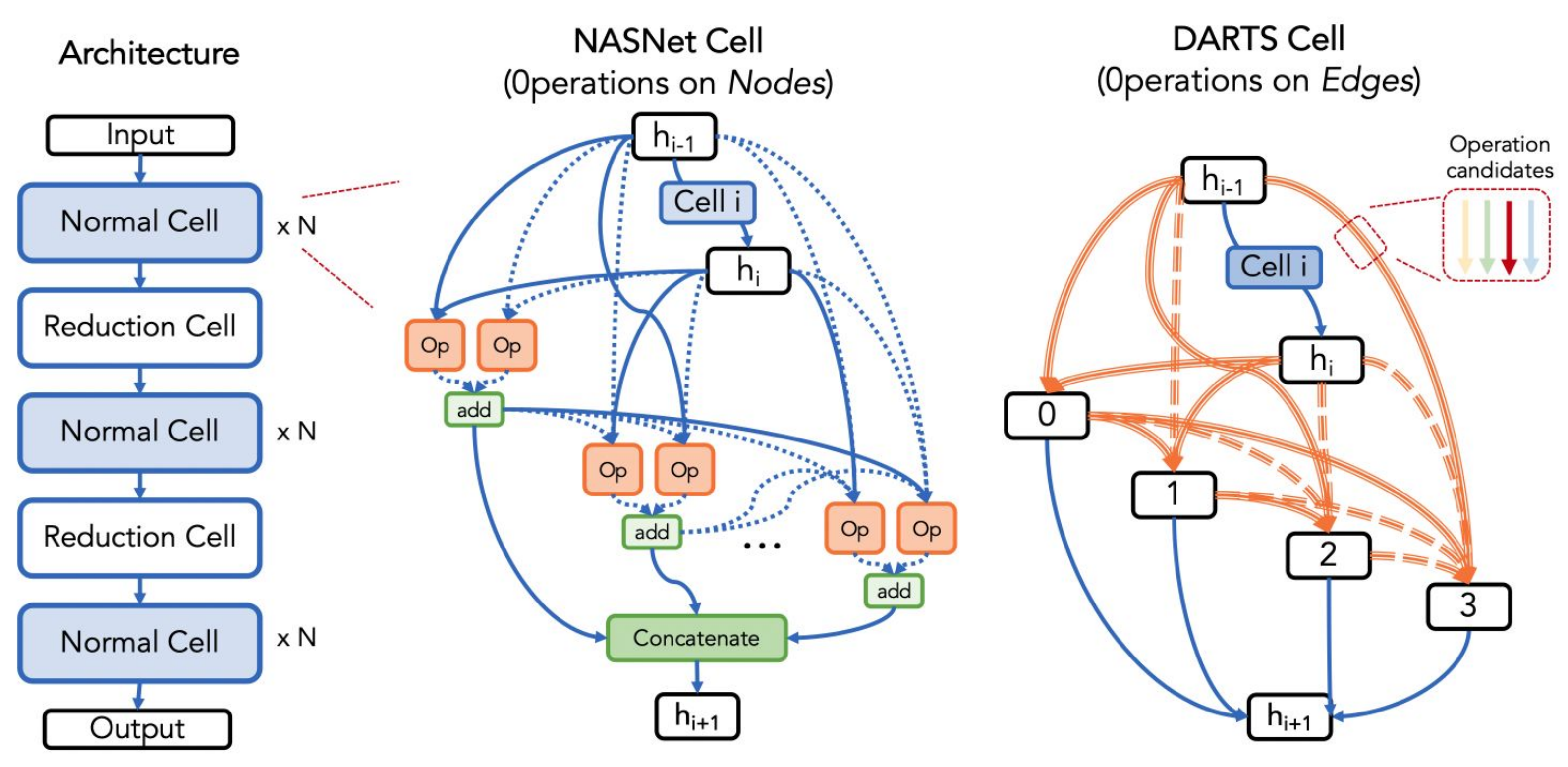
Macro Search Spaces

- Define a set of operations
- Iteratively add more nodes



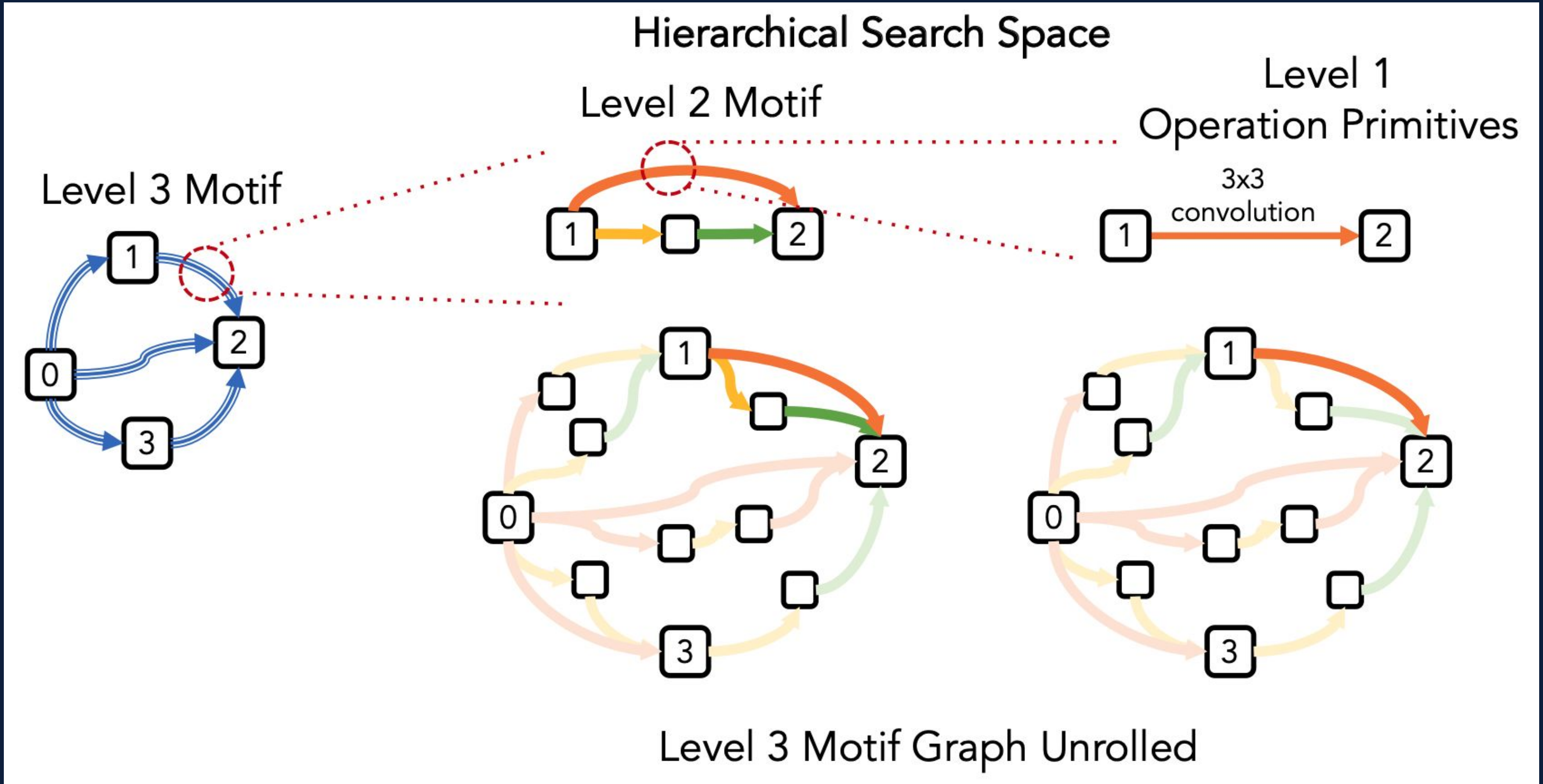
(NASBOT 2018)

Cell-based search spaces



[NASNet (2017)] [DARTS (2018)]

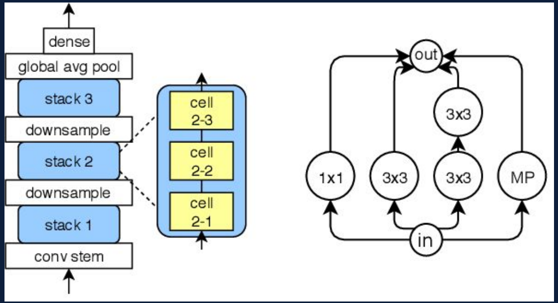
Hierarchical Search Spaces

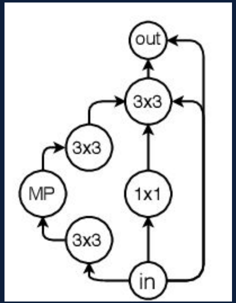


Liu et al. (2017) Ru et al. (2020)

NAS-Bench-101

- Size 423k
- Used to **simulate** NAS experiments
- Allows for more principled research
  - Fixed training pipeline
  - Can run many trials






```
# Load the data from file (this will take some time)
nasbench = api.NASBench('/path/to/nasbench.tfrecord')

# Create an Inception-like module (5x5 convolution replaced with two 3x3
# convolutions).
model_spec = api.ModelSpec(
    # Adjacency matrix of the module
    matrix=[[0, 1, 1, 1, 0, 1, 0],    # input layer
            [0, 0, 0, 0, 0, 0, 1],    # 1x1 conv
            [0, 0, 0, 0, 0, 0, 1],    # 3x3 conv
            [0, 0, 0, 0, 1, 0, 0],    # 5x5 conv (replaced by two 3x3's)
            [0, 0, 0, 0, 0, 0, 1],    # 5x5 conv (replaced by two 3x3's)
            [0, 0, 0, 0, 0, 0, 1],    # 3x3 max-pool
            [0, 0, 0, 0, 0, 0, 0]],   # output layer
    # Operations at the vertices of the module, matches order of matrix
    ops=[INPUT, CONV1X1, CONV3X3, CONV3X3, CONV3X3, MAXPOOL3X3, OUTPUT])

# Query this model from dataset, returns a dictionary containing the metrics
# associated with this model.
data = nasbench.query(model_spec)
```

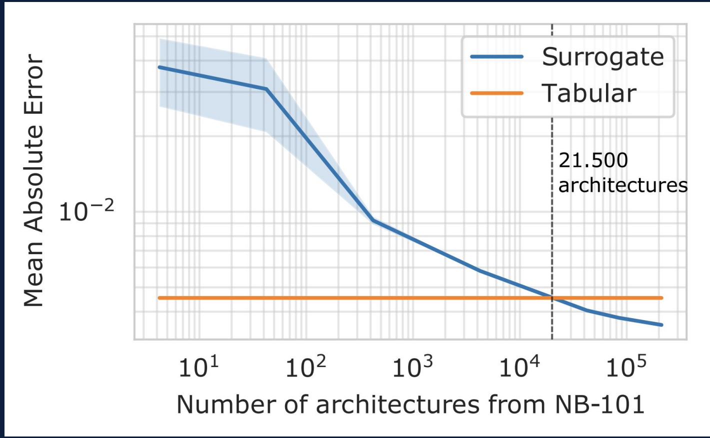
[NAS-Bench-101 (2019)]

NAS Benchmarks

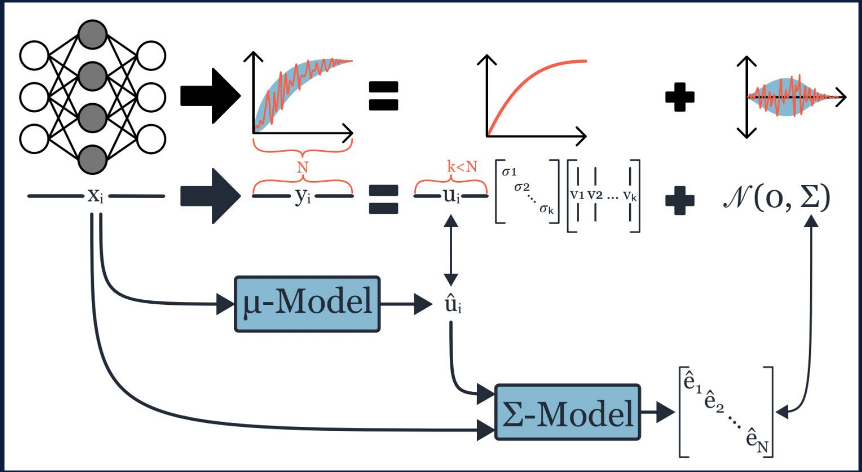
		Queryable						
Benchmark	**Size**	**Tab.**	**Surr.**	**LCs**	**Macro**	**One-Shot**	**Task**	**#Tasks**
NAS-Bench-101	423k	✓				✗	Image class.	1
NATS-Bench-TSS (NAS-Bench-201)	6k	✓		✓		✓	Image class.	3
NATS-Bench-SSS	32k	✓		✓	✓	✓	Image class.	3
NAS-Bench-NLP	$> 10^{53}$			✓		✗	NLP	1
NAS-Bench-1Shot1	364k	✓				✓	Image class.	1
Surr-NAS-Bench-DARTS (NAS-Bench-301)	10^{18}		✓			✓	Image class.	1
Surr-NAS-Bench-FBNet	10^{21}		✓			✗	Image class.	1
NAS-Bench-ASR	8k	✓			✓	✓	ASR	1
TransNAS-Bench-101-Micro	4k	✓		✓		✓	Var. CV	7
TransNAS-Bench-101-Macro	3k	✓		✓	✓	✗	Var. CV	7
NAS-Bench-111	423k		✓	✓		✗	Image class.	1
NAS-Bench-311	10^{18}		✓	✓		✓	Image class.	1
NAS-Bench-NLP11	$> 10^{53}$		✓	✓		✗	NLP	1
NAS-Bench-MR	10^{23}		✓		✓	✗	Var. CV	9
NAS-Bench-360	Var.				✓	✓	Var.	30
NAS-Bench-Macro	6k	✓			✓	✗	Image class.	1
HW-NAS-Bench-201	6k	✓		✓		✓	Image class.	3
HW-NAS-Bench-FBNet	10^{21}					✗	Image class.	1

Surrogate NAS Benchmarks

- Surr-NAS-Bench-DARTS (NAS-Bench-301)
- Surr-NAS-Bench-FBNet
- NAS-Bench-111
- NAS-Bench-311
- NAS-Bench-NLP11



[Surr-NAS-Bench (2020)]

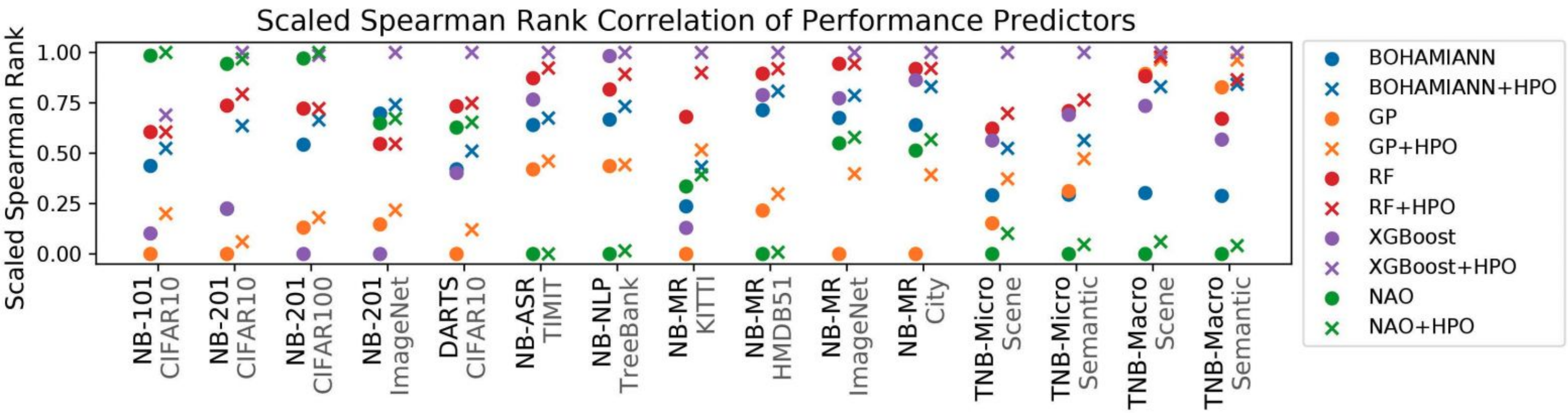


[NAS-Bench-x11 (2021)]

NAS-Bench-Suite (25 tasks)

	NAS Algorithms					Performance Predictors				
	RS	RE	BANANAS	LS	NPENAS	GP	BOHAM.	RF	XGB	NAO
Avg.Rank, 101&201	4.50	3.00	3.50	**1.50**	2.50	4.67	2.83	2.17	4.17	**1.17**
Avg. Rank, non-101&201	3.06	**2.11**	2.83	3.13	3.87	4.08	3.06	**1.33**	2.46	4.08

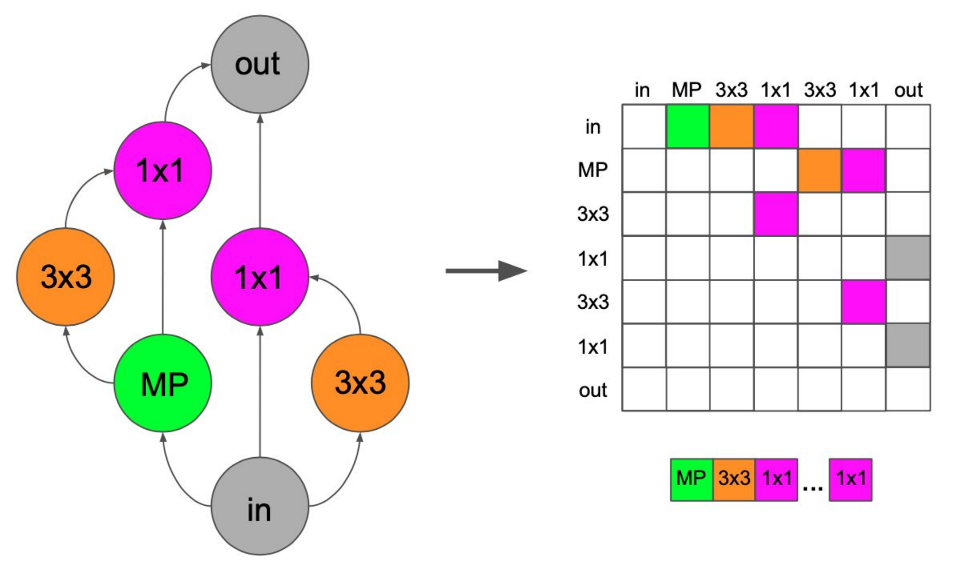
❖ Conclusions drawn from just the popular NAS-Bench-101 and NAS-Bench-201 can be misleading!



NAS-Bench-Suite (2022) https://github.com/automl/naslib

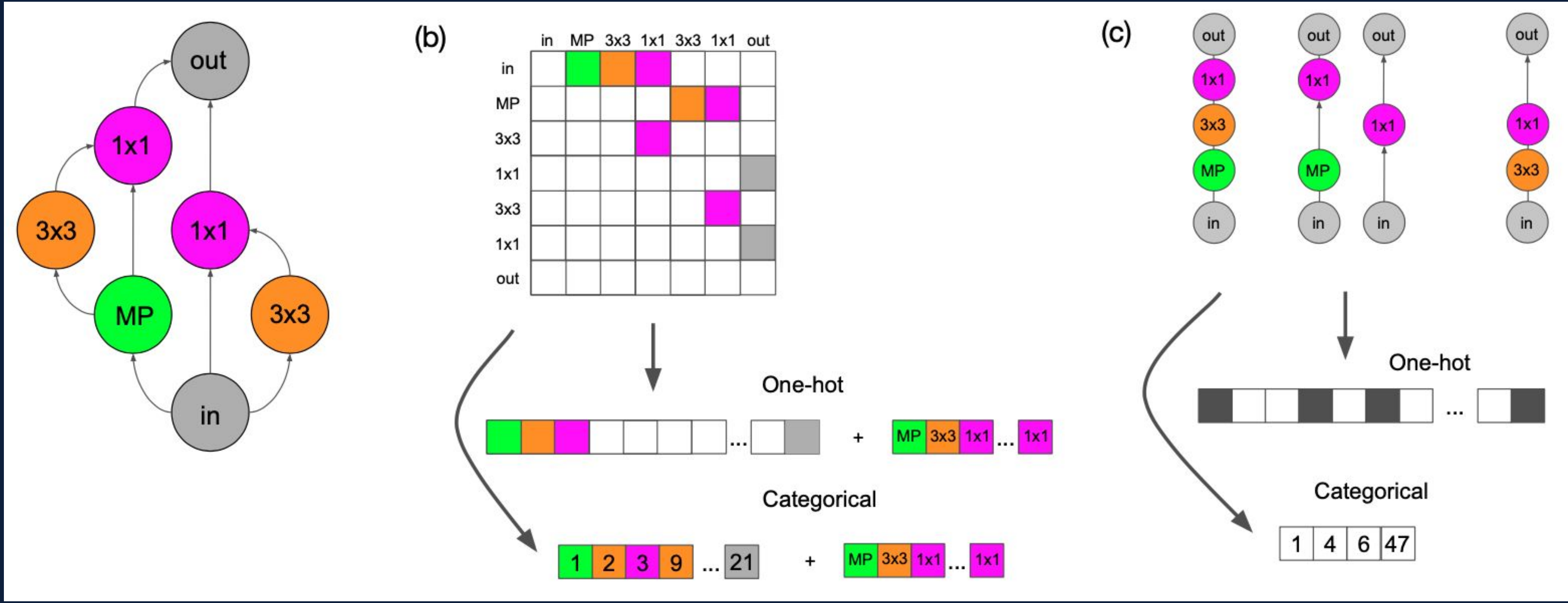
Encodings for NAS

Most NAS algorithms search over DAG-based architectures, which must be encoded into a tensor



White et al. (2020)

Encodings for NAS



NAS encoding-dependent subroutines

Many NAS algos can be composed from three subroutines

- Sample random architecture
- Perturb architecture
- Train predictor model

Algorithm 1 BANANAS

Input: Search space A, dataset D, parameters t_0, T, M, c, x, acquisition function ϕ, function $f(a)$ returning validation error of a after training.

1. Draw t_0 architectures $a_0, \dots, a_{t_0}$ uniformly at random from A and train them on D.
2. For t from t_0 to T,
   i. Train an ensemble of neural predictors on $\{(a_0, f(a_0)), \dots, (a_t, f(a_t))\}$ using the path encoding to represent each architecture.
   ii. Generate a set of c candidate architectures from A by randomly mutating the x architectures a from $\{a_0, \dots, a_t\}$ that have the lowest value of $f(a)$.
   iii. For each candidate architecture a, evaluate the acquisition function $\phi(a)$.
   iv. Denote a_{t+1} as the candidate architecture with minimum $\phi(a)$, and evaluate $f(a_{t+1})$.

Output: $a^* = \text{argmin}_{t=0,\dots,T} f(a_t)$.

Algorithm 1 Aging Evolution

$population \leftarrow$ empty queue ▷ The population.
$history \leftarrow \varnothing$ ▷ Will contain all models.
while $|population| < P$ **do** ▷ Initialize population.
  $model.arch \leftarrow$ RANDOMARCHITECTURE()
  $model.accuracy \leftarrow$ TRAINANDEVAL($model.arch$)
  add $model$ to right of $population$
  add $model$ to history
end while
while $|history| < C$ **do** ▷ Evolve for C cycles.
  $sample \leftarrow \varnothing$ ▷ Parent candidates.
  while $|sample| < S$ **do**
    $candidate \leftarrow$ random element from $population$
      ▷ The element stays in the $population$.
    add $candidate$ to $sample$
  end while
  $parent \leftarrow$ highest-accuracy model in $sample$
  $child.arch \leftarrow$ MUTATE($parent.arch$)
  $child.accuracy \leftarrow$ TRAINANDEVAL($child.arch$)
  add $child$ to right of $population$
  add $child$ to $history$
  remove $dead$ from left of $population$ ▷ Oldest.
  discard $dead$
end while
return highest-accuracy model in $history$

Algorithm 1 Bayesian Optimized Neural Architecture Search (BONAS). $\mathcal{A}$ is the given search space, N is the number of initial architectures, k is the ratio of GCN / BLR update times.

initialize random N fully-trained architectures: $\mathcal{D} = \{(A_i, X_i, t_i)\}_{i=1}^N$ from search space $\mathcal{A}$;
initial training of GCN using $\mathcal{D}$ with proposed loss;
replace the final layer of GCN with BLR;
initialize Sampler;
repeat
  for iteration$= 1, 2, \dots, k$ **do**
    sample candidate pool $\mathcal{C}$ from $\mathcal{A}$;
    for each candidate m in $\mathcal{C}$ **do**
      embed m using GCN;
      compute μ and σ^2 in (4) and (5) using BLR;
      compute expected improvement (EI) in (2);
    end for
    $M \leftarrow$ candidate with the highest EI score;
    fully train M to obtain its actual performance;
    add M and its actual performance to $\mathcal{D}$;
    update BLR using the enlarged $\mathcal{D}$;
    update Sampler;
  end for
  retrain GCN using the enlarged $\mathcal{D}$ with proposed loss;
until stop criterion satisfy.

White et al. (2020)

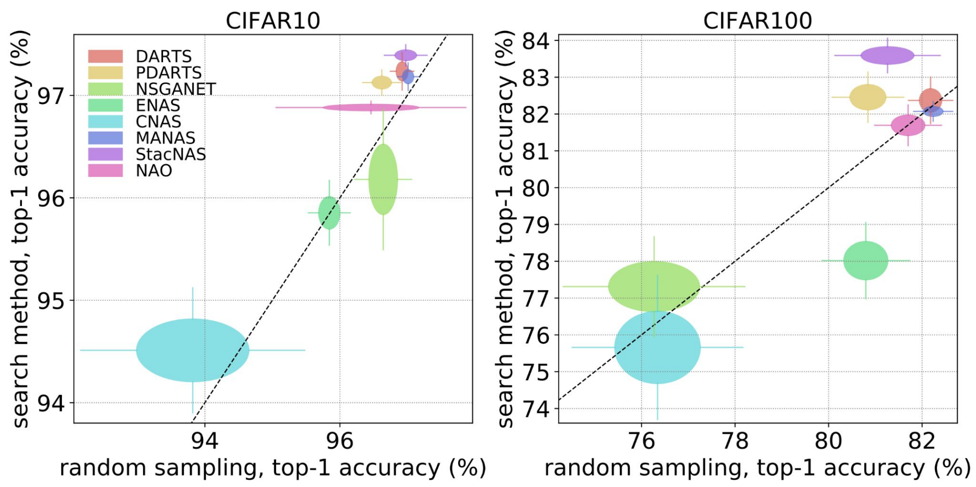
Roadmap - Part 1

- Motivation and introduction
- Search spaces
  - Macro
  - Cell-based
  - Hierarchical
  - Encodings
- Black-box optimization methods
  - Baselines
  - Bayesian optimization
  - Evolution
- Performance prediction

Random Search & Random Sampling

Random search is surprisingly competitive [Li and Talwalkar, 2019], [Yang et al., 2019], [Sciuto et al., 2020]

Random sampling: performance of a randomly drawn architecture.



Yang et al. (2019)

Local Search

- Five lines of code
- Performs surprisingly well on popular benchmarks

Algorithm 1 Local search

Input: Search space A, objective function ℓ, neighborhood function N

1. Pick an architecture $v_1 \in A$ uniformly at random
2. Evaluate $\ell(v_1)$; denote a dummy variable $\ell(v_0) = \infty$; set $i = 1$
3. While $\ell(v_i) < \ell(v_{i-1})$:
   i. Evaluate $\ell(u)$ for all $u \in N(v_i)$
   ii. Set $v_{i+1} = \text{argmin}_{u \in N(v_i)} \ell(u)$; set $i = i + 1$

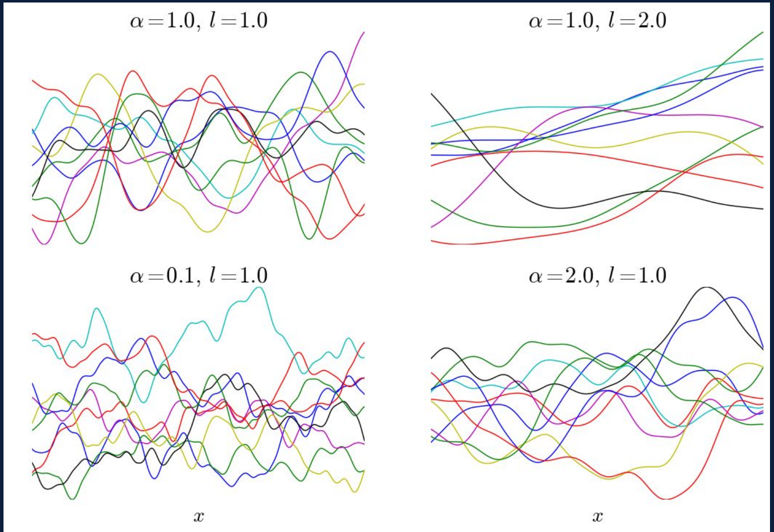
Output: Architecture v_i

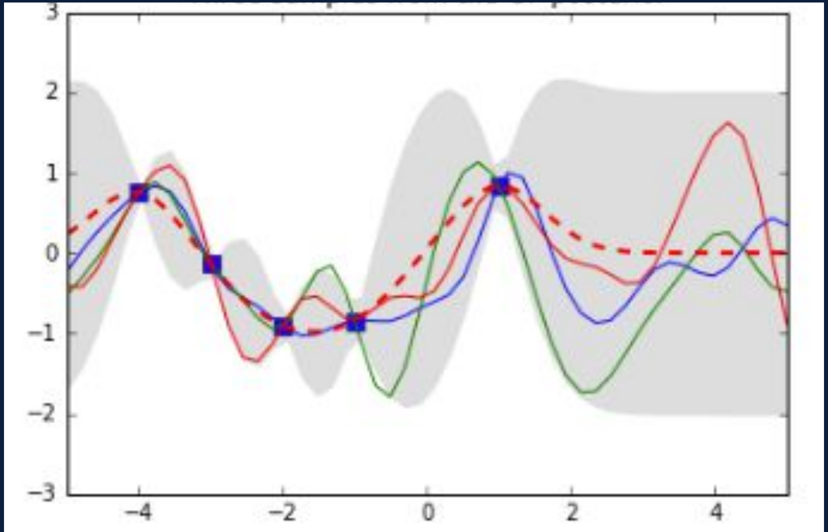
[Ottellander et al. 2020]

[White et al. 2020]

Bayesian optimization

- [NASBOT, 2018], [Auto-Keras, 2018], [NASBOWL, 2020]





Source: http://keyonvafa.com/gp-tutorial/, https://katbailey.github.io/post/gaussian-processes-for-dummies/

“BO + Neural Predictor” Framework

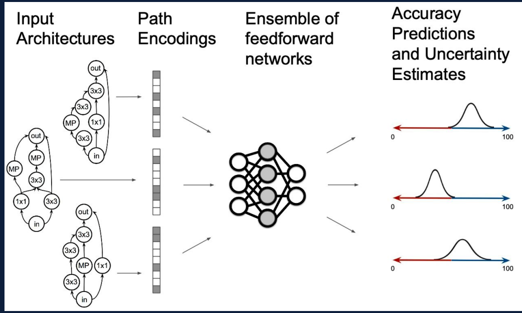
[NASGBO, 2019], [BONAS, 2019], [BANANAS, 2021]

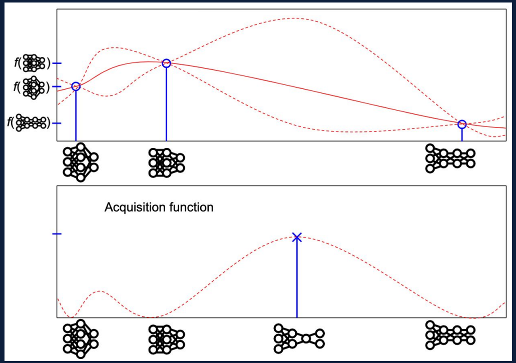
Algorithm 1 BANANAS

Input: Search space A, dataset D, parameters t_0, T, M, c, x, acquisition function ϕ, function $f(a)$ returning validation error of a after training.

1. Draw t_0 architectures $a_0, \ldots, a_{t_0}$ uniformly at random from A and train them on D.
2. For t from t_0 to T,
   - i. Train an ensemble of meta neural networks on $\{(a_0, f(a_0)), \ldots, (a_t, f(a_t))\}$.
   - ii. Generate a set of c candidate architectures from A by randomly mutating the x architectures a from $\{a_0, \ldots, a_t\}$ that have the lowest value of $f(a)$.
   - iii. For each candidate architecture a, evaluate the acquisition function $\phi(a)$.
   - iv. Denote a_{t+1} as the candidate architecture with minimum $\phi(a)$, and evaluate $f(a_{t+1})$.

Output: $a^* = \text{argmin}_{t=0,\ldots,T} f(a_t)$.





Train 10 arch.’s each iteration

“BO + Neural Predictor” Components

Algorithm 1 BANANAS

Input: Search space A, dataset D, parameters t_0, T, M, c, x, acquisition function ϕ, function $f(a)$ returning validation error of a after training.

1. Draw t_0 architectures $a_0, \dots, a_{t_0}$ uniformly at random from A and train them on D.
2. For t from t_0 to T,
   i. Train an ensemble of meta neural networks on $\{(a_0, f(a_0)), \dots, (a_t, f(a_t))\}$.
   ii. Generate a set of c candidate architectures from A by randomly mutating the x architectures a from $\{a_0, \dots, a_t\}$ that have the lowest value of $f(a)$.
   iii. For each candidate architecture a, evaluate the acquisition function $\phi(a)$.
   iv. Denote a_{t+1} as the candidate architecture with minimum $\phi(a)$, and evaluate $f(a_{t+1})$.

Output: $a^* = \text{argmin}_{t=0,\dots,T} f(a_t)$.

- Architecture encoding
- Uncertainty calibration
- Neural predictor architecture
- Acquisition function
- Acquisition optimization strategy

BANANAS

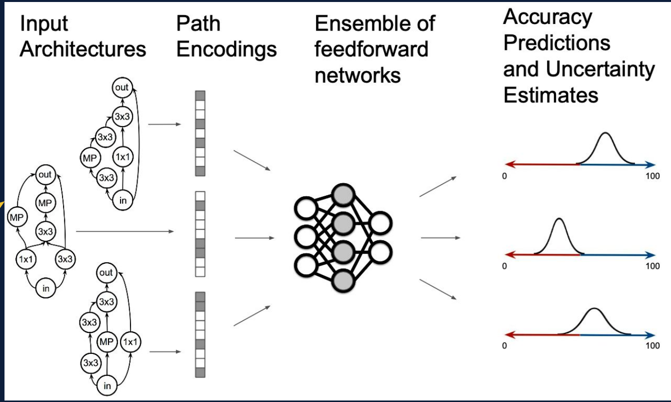
[BANANAS, 2021]

Algorithm 1 BANANAS

Input: Search space A, dataset D, parameters t_0, T, M, c, x, acquisition function ϕ, function $f(a)$ returning validation error of a after training.

1. Draw t_0 architectures $a_0, \ldots, a_{t_0}$ uniformly at random from A and train them on D.
2. For t from t_0 to T,
   i. Train an ensemble of meta neural networks on $\{(a_0, f(a_0)), \ldots, (a_t, f(a_t))\}$.
   ii. Generate a set of c candidate architectures from A by randomly mutating the x architectures a from $\{a_0, \ldots, a_t\}$ that have the lowest value of $f(a)$.
   iii. For each candidate architecture a, evaluate the acquisition function $\phi(a)$.
   iv. Denote a_{t+1} as the candidate architecture with minimum $\phi(a)$, and evaluate $f(a_{t+1})$.

Output: $a^* = \text{argmin}_{t=0,\ldots,T} f(a_t)$.



Path encoding, ensemble

Small mutations

Independent Thompson Sampling

Evolution

Algorithm 1 General Evolutionary Algorithm

Input: Initial population of architecture data $\mathcal{D}_0 = (a_i, y_i)_{i=1}^{N_0}$, objective function f, total number of evolution steps T

Output: The optimal architecture a_T^*

for $t = 1, \ldots, T$ **do**

Sample a set of parent architectures $\mathcal{S}_{parents} = \{(a_j, y_j)\}_{j=1}^{N_p}$ from the population $\mathcal{D}_{t-1}$

Generate children architectures by mutating parent architectures and evaluate their performance to obtain $\mathcal{S}_{children} = \{(a_k, y_k)\}_{k=1}^{N_c}$

Update the population with $\mathcal{S}_{children}$ to obtain $\mathcal{D}_t$

end for

Decisions: sampling the initial population, selecting the parents, generating the offspring

Real et al. (2018)

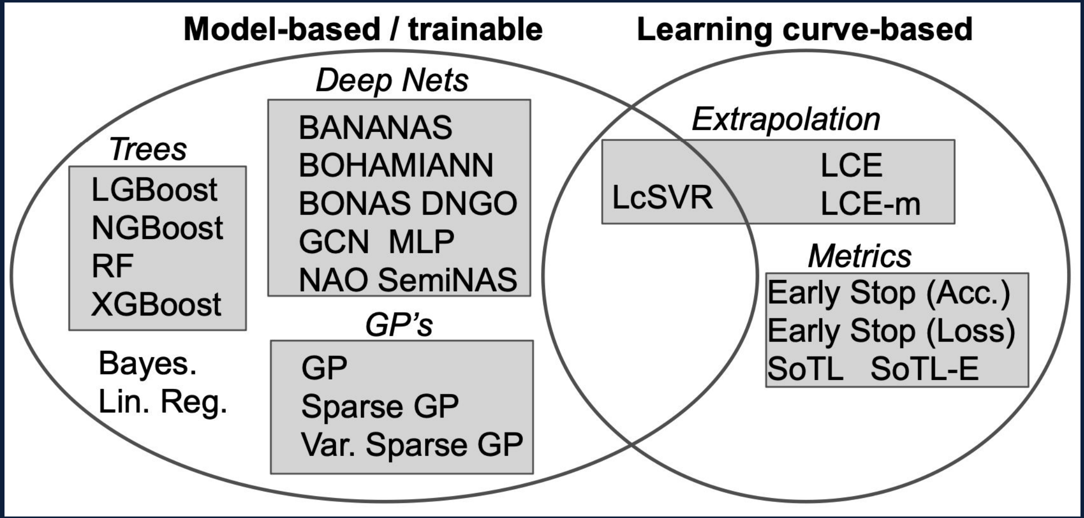
Roadmap - Part 1

- Motivation and introduction
- Search spaces
  - Macro
  - Cell-based
  - Hierarchical
  - Encodings
- Black-box optimization methods
  - Baselines
  - Bayesian optimization
  - Evolution
- Performance prediction

Performance Predictors

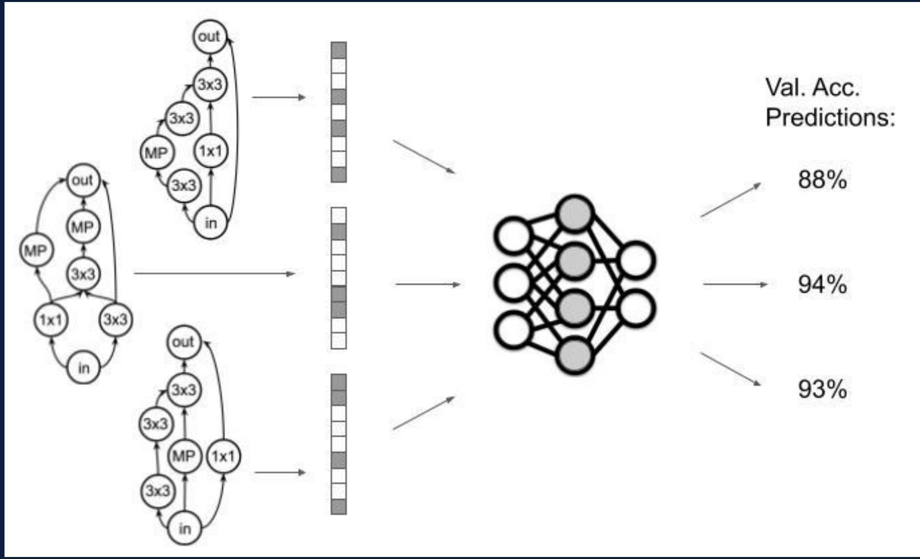
Any technique which predicts the (relative) accuracy of an architecture, without fully training it.

- ***Initialization***: performs any necessary pre-computation
- ***Query***: take any architecture as input, and output predicted accuracy



Model-Based Predictors

- Supervised learning - regression
  - X - the architecture encoding (e.g. one-hot adjacency matrix)
  - Y - validation accuracy of trained architecture



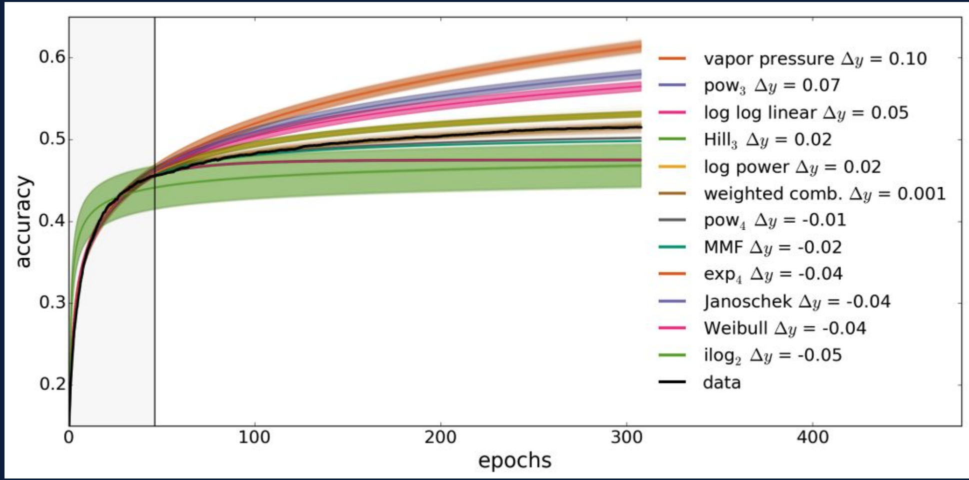
[White et al. 2019]

- Gaussian processes [Kandasamy et al. 2018], [Jin et al. 2018]
- Boosted trees [Luo et al. 2020], [Siems et al. 2020]
- GNNs [Shi et al. 2019], [Wen et al. 2019]
- Specialized encodings [White et al. 2019], [Ning et al. 2020]

High init time, **low query time**

Learning curve based predictors

- Learning curve extrapolation
  - Fit partial learning curve to parametric model [Domhan et al. 2015]
  - Bayesian NN [Klein et al. 2017]
- Training statistics
  - Early stopping (val acc) [Elsken et al. 2018]
  - Sum of training losses [Ru et al. 2020]



[Elsken et al. 2018]

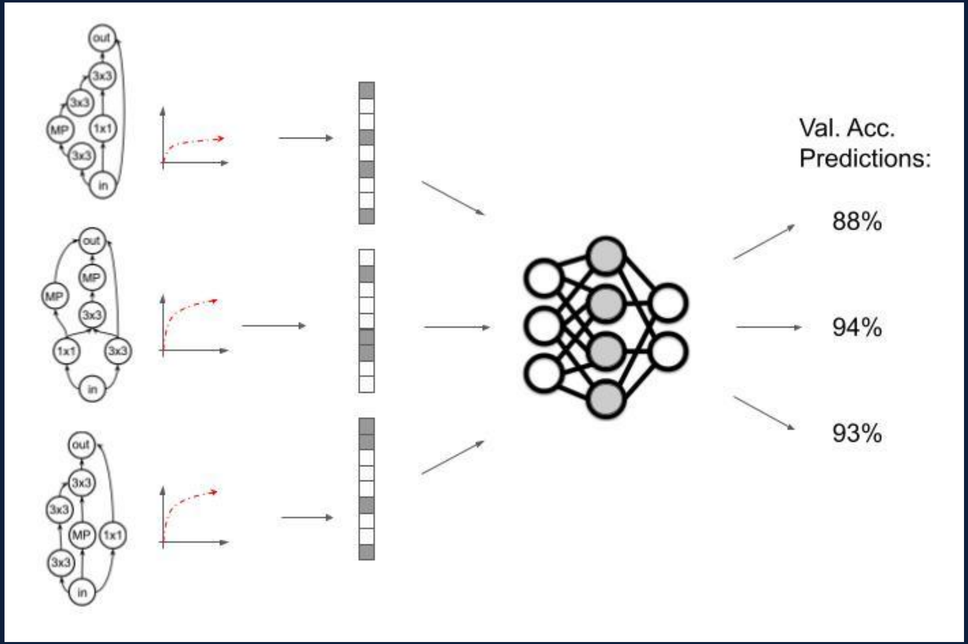
No init time, **high query time**

Hybrid model-based + LC predictors

Train a model, using partial learning curve + hyperparams, to predict final accuracy

- First and second derivatives as features, SVR [Baker et al. 2017]
- Full LC as features, Bayesian NN [Klein et al. 2017]

High init time, high query time

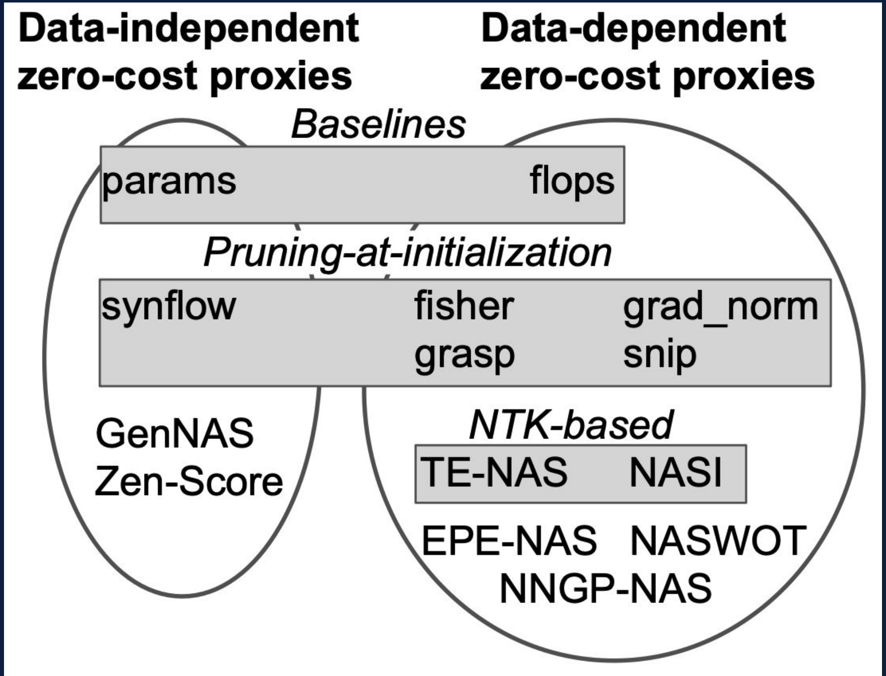


"Zero-cost" proxies

Compute a statistic of an architecture in 3-5 seconds

- Jacobian covariance [Mellor et al. 2020]
- Synaptic Flow [Abdelfattah et al. 2021]
  - SNIP [Lee et al. 2018]

Low init time, low query time



[Abdelfattah et al. 2021]

$$\texttt{snip} : \mathcal{S}_p(\theta) = \left| \frac{\partial \mathcal{L}}{\partial \theta} \odot \theta \right|, \quad \texttt{grasp} : \mathcal{S}_p(\theta) = -(H \frac{\partial \mathcal{L}}{\partial \theta}) \odot \theta, \quad \texttt{synflow} : \mathcal{S}_p(\theta) = \frac{\partial \mathcal{L}}{\partial \theta} \odot \theta$$

OMNI: The Omnipotent Predictor

Algorithm 1 OMNI predictor

Input: Search space A, dataset D, initialization time budget B_{init}, query time budget B_{query}.

Initialization():

- $\mathcal{D}_{\text{train}} \leftarrow \emptyset$
- While $t < B_{\text{init}}$
  - Draw an architecture a randomly from A
  - Train a to completion to compute val. accuracy y_a
  - $\mathcal{D}_{\text{train}} \leftarrow \mathcal{D}_{\text{train}} \cup \{(a, y_a)\}$
- Train an NGBoost model m to predict the final val. accuracy of architectures from $\mathcal{D}_{\text{train}}$, using the architecture encoding, SoTL-E, and Jacob. cov. as input features.

Query(architecture a_{test}):

- While $t < B_{\text{query}}$, train a_{test}
- Compute SoTL-E using the partial learning curve, and compute Jacob. cov., and the arch. encoding of a_{test}
- Predict val. acc. of a_{test} using m and the above features.

[White et al. (2021)]

Thanks! Questions?

- Search spaces
  - Macro
  - Cell-based
  - Hierarchical
  - Encodings
- Black-box optimization methods
  - Baselines
  - Bayesian optimization
  - Evolution
- Performance prediction

colin@abacus.ai

Slides (with hyperlinks): https://crwhite.ml/

Microsoft

Neural Architecture Search : Part 2

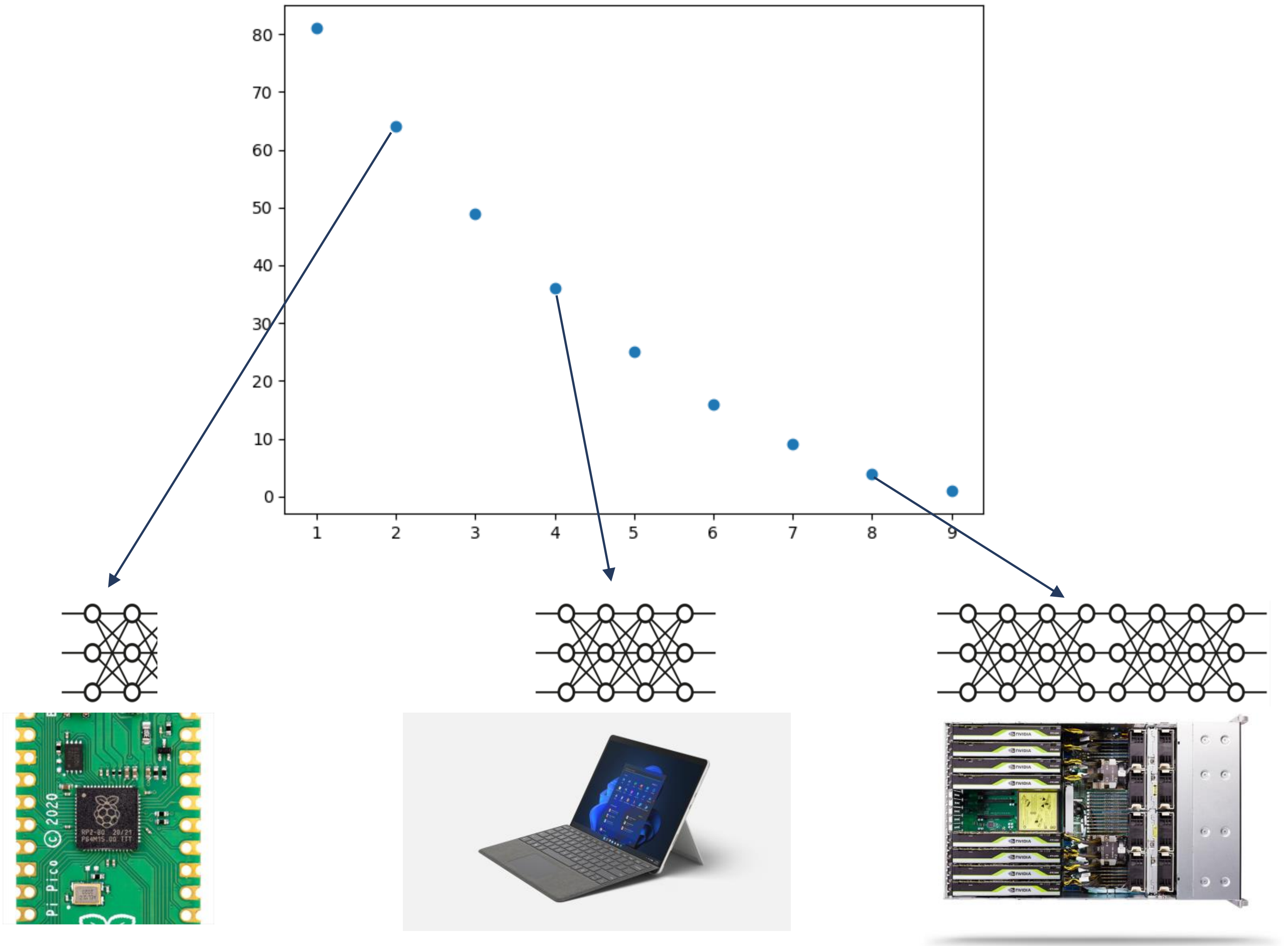
Debadeepta Dey, Microsoft Research (dedey@microsoft.com)
Colin White, Abacus.AI (colin@abacus.ai)

Roadmap

- The Power of Pareto-Frontiers
- Weight-sharing Methods
  - ENAS
  - OFA
- Recent Transformer-based Search Spaces
- Petridish
- Reproducibility, Fair Comparison, Best Practices
- Open Problems

The Power of Pareto-Frontiers

The Power of Pareto-Frontier: Varying compute ability

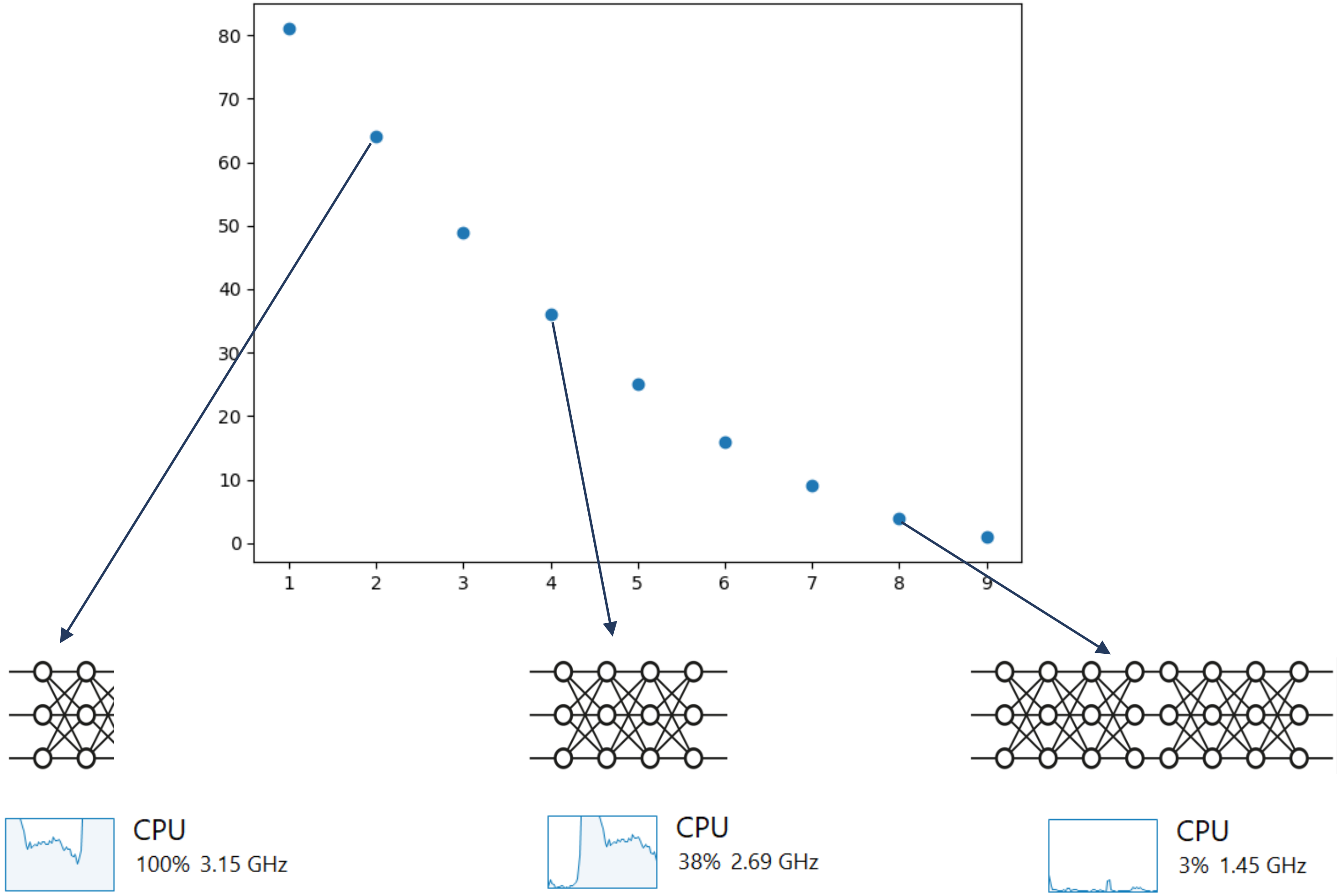


IoT Edge

Laptop

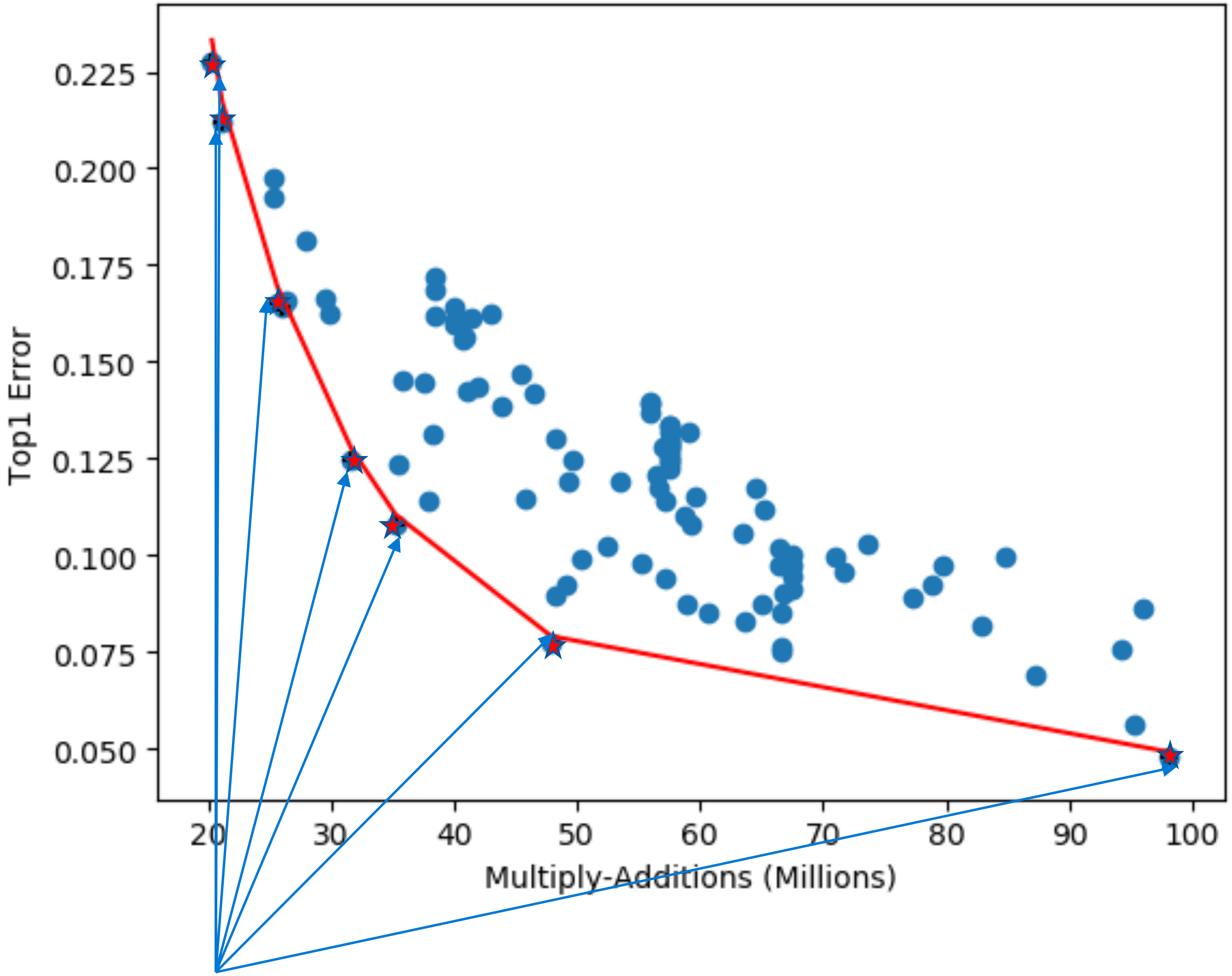
Server

The Power of Pareto-Frontier: Dynamic device load



Pareto-frontiers are *generalization* of model compression!

Search Once, Deploy Everywhere!



Train models on the frontier!

Pareto-Frontier Search Methods

- The vast majority of methods in current NAS literature do *NOT* output pareto-frontiers!
- Combining multiple objectives via scalarization does *NOT* output pareto-frontiers!
- Can we leverage single-objective search methods and turn them into pareto-frontier output methods?

[Bag of Baselines for Multi-objective Joint Neural Architecture Search and Hyperparameter Optimization]

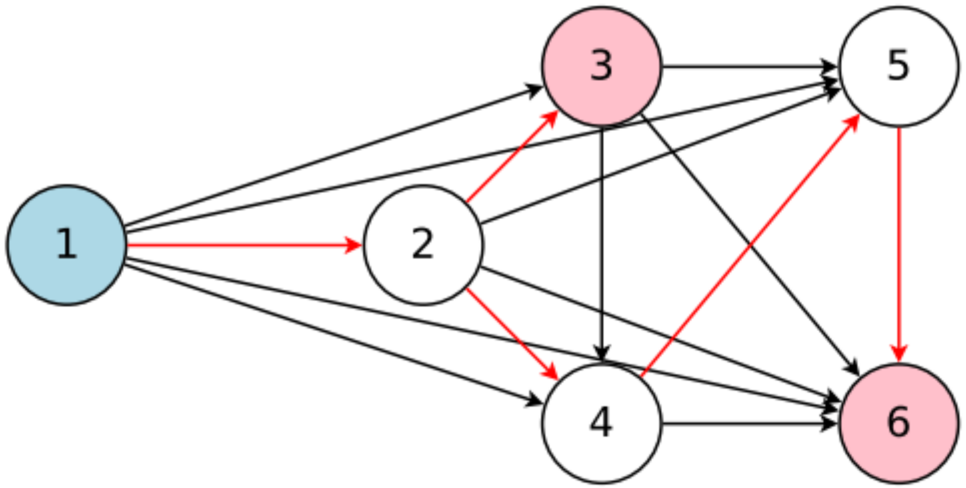
Guerrero-Viu et al., AutoML Workshop at ICML 2021

Weight-sharing Methods

Efficient Neural Architecture Search
via
Parameter Sharing
Pham et al, ICML 2018

ENAS

ENAS



"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."

- Uses a single Nvidia 1080Ti GPU!
  - Search < 16 hours!
  - Compared to NAS via RL, 1000x reduction in search time!

Diagram credit:
ENAS ICML 2018

Microsoft

Please attend NAS 2 for weight-sharing in-depth!

Efficient Neural Architecture Search
by Tejaswini Pedapati, Martin Wistuba

The growing interest in the automation of deep learning has led to the development of a wide variety of automated methods for Neural Architecture Search. However, initial neural architecture algorithms were computationally intensive and took several GPU days. Training a candidate network is the most expensive step of the search. Rather than training each candidate network from scratch, the next few algorithms proposed parameter sharing amongst the candidate networks. But these parameter-sharing algorithms had their own drawbacks. In this tutorial, we would give an overview of some of the one-shot algorithms, their drawbacks, and how to combat them. Later advancements accelerated the search by training fewer candidates using techniques such as zero-shot, few-shot, and transfer learning. Just by using some characteristics of a randomly initialized neural network, some search algorithms were able to find a well-performing model. Rather than searching from scratch, some methods leveraged transfer learning. In this tutorial, we cover several of these flavors of algorithms that expedited the Neural Architecture Search.

[Once-for-All: Train One Network and Specialize it for Efficient Deployment
Cai et al., ICLR 2020]

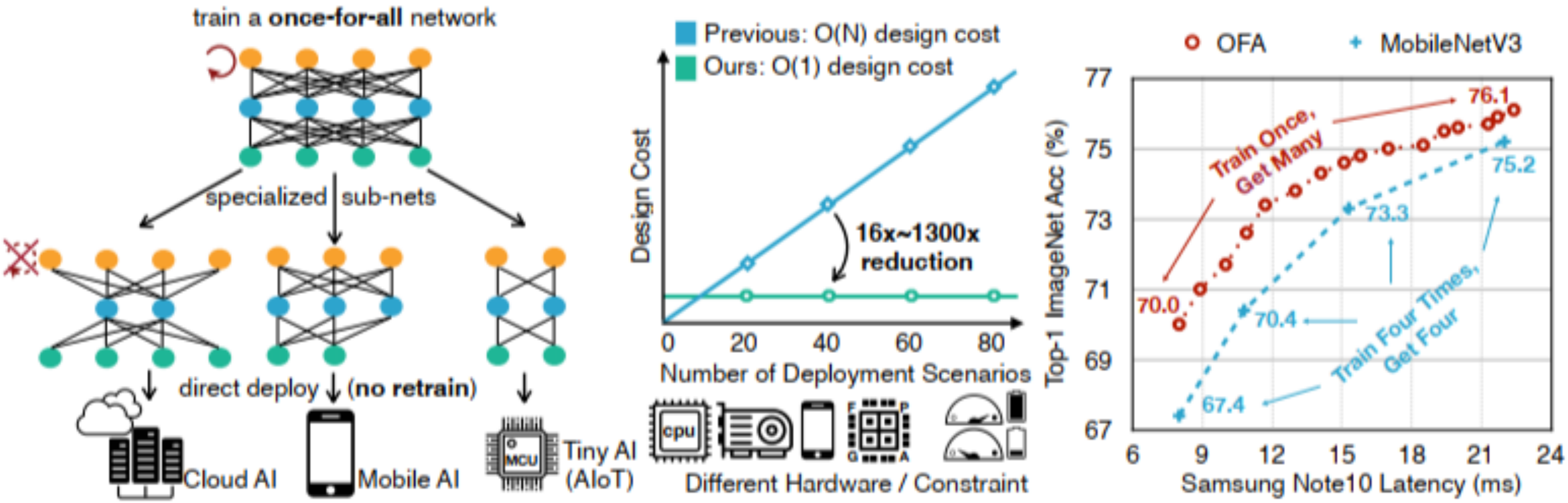


Diagram credit: OFA ICLR 2020

Phase 1: Train supergraph

$$\min_{W_o} \sum_{arch_i} \mathcal{L}_{val}\big(C(W_o, arch_i)\big)$$

- Want to find weights such that every subgraph is competitive wrt the subgraph being independently trained!
- Exponentially many subgraphs!
  - Infeasible to enumerate and train each separately. ☹
- Can randomly sample a few each step and update shared weights (remember ENAS!)
  - Updates interfere with each other leading to reduced performance ☹
- Solution: Train the biggest and progressively shrink down!

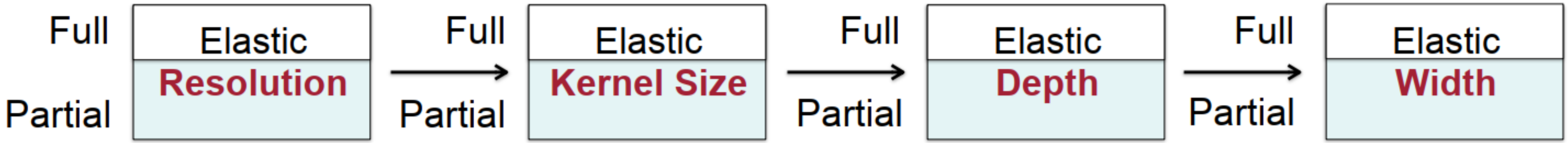
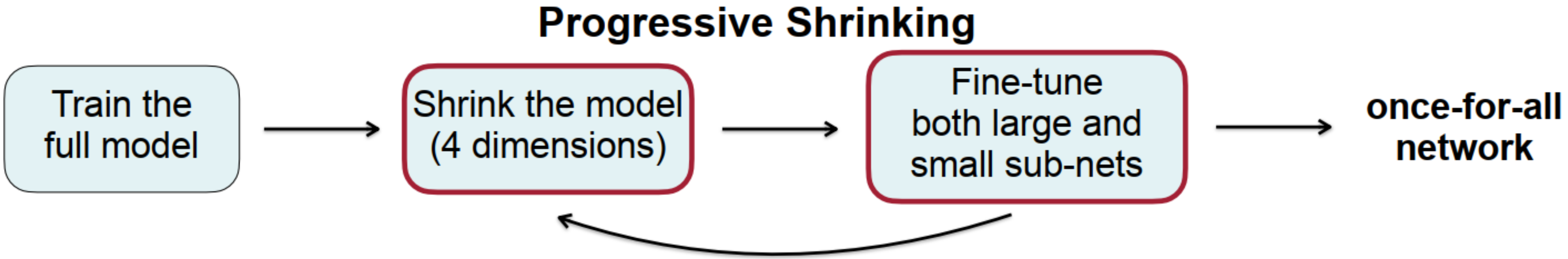
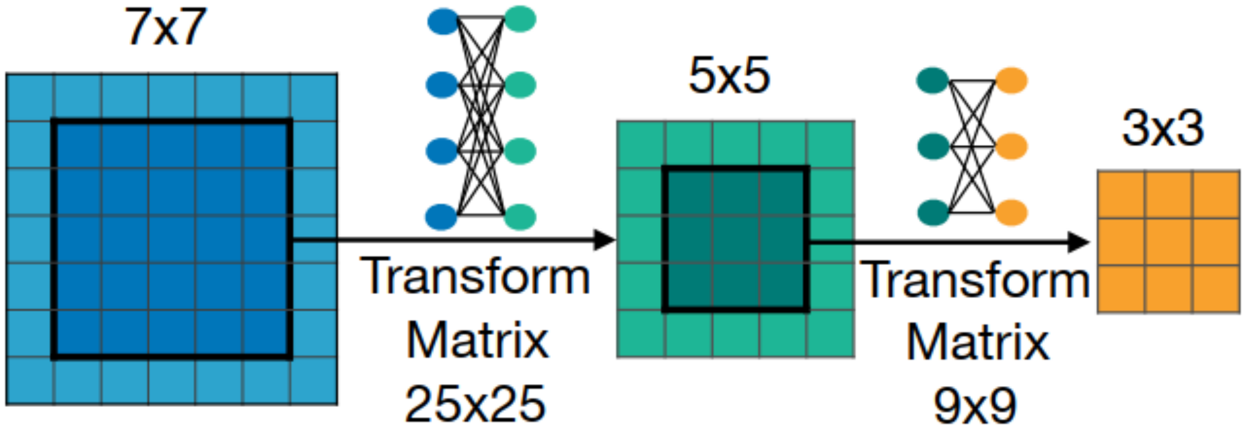


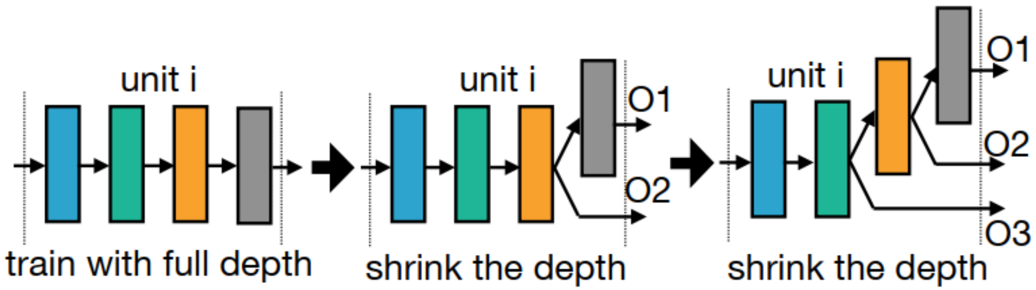
Diagram credit: OFA ICLR 2020

Phase 1: Train supergraph





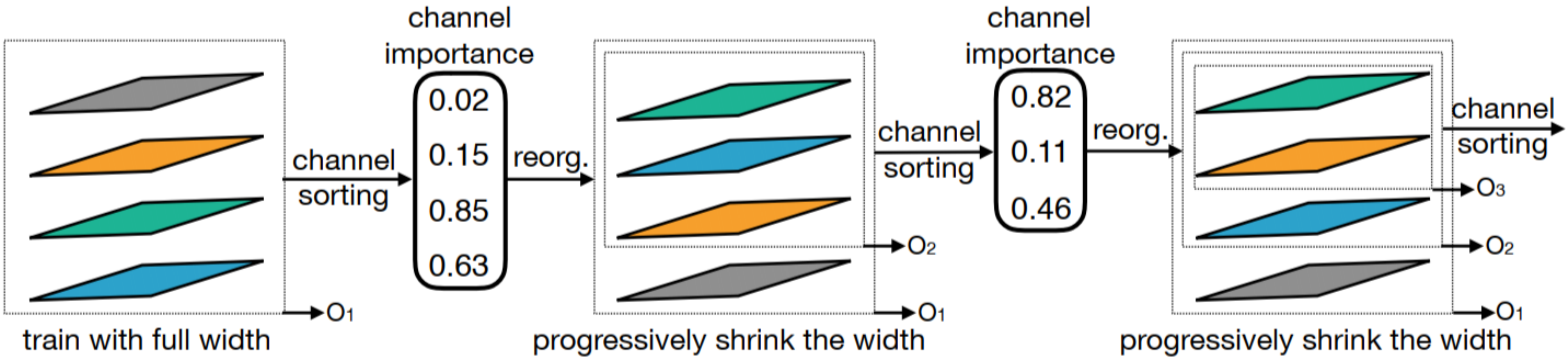
Kernel Shrinking



Depth Shrinking

Diagram credit: OFA ICLR 2020

Phase 1: Train supergraph



Width shrinking

- Throughout kernel, depth and width shrinking sample different input resolutions.
- Important detail: Use distillation to guide training of smaller architectures!
- Phase 1 cost: 1200 GPU hours (~3 days with 16 GPUs)

Diagram credit: OFA ICLR 2020

Phase 2: Train regressors

- Sample 16k different architectures – input image sizes and measure accuracy on validation set to generate (architecture, accuracy) tuples.
  - Train small NN to predict accuracy given architecture as input.
- Do same on each target platform to get (architecture, latency) tuples.
  - Train small NN to predict latency given architecture as input.

- Phase 2 cost: ~40 GPU hours

Search

- Simple! Use evolutionary search/RL/random search against the simulators (regressors from Phase 2)
- Search cost: a few minutes on a laptop!

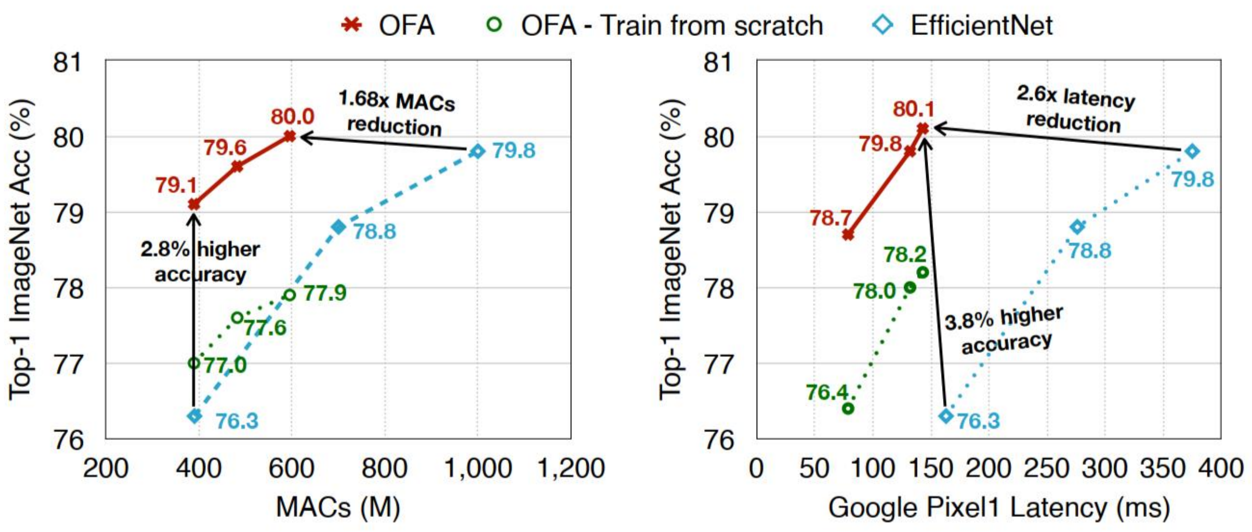
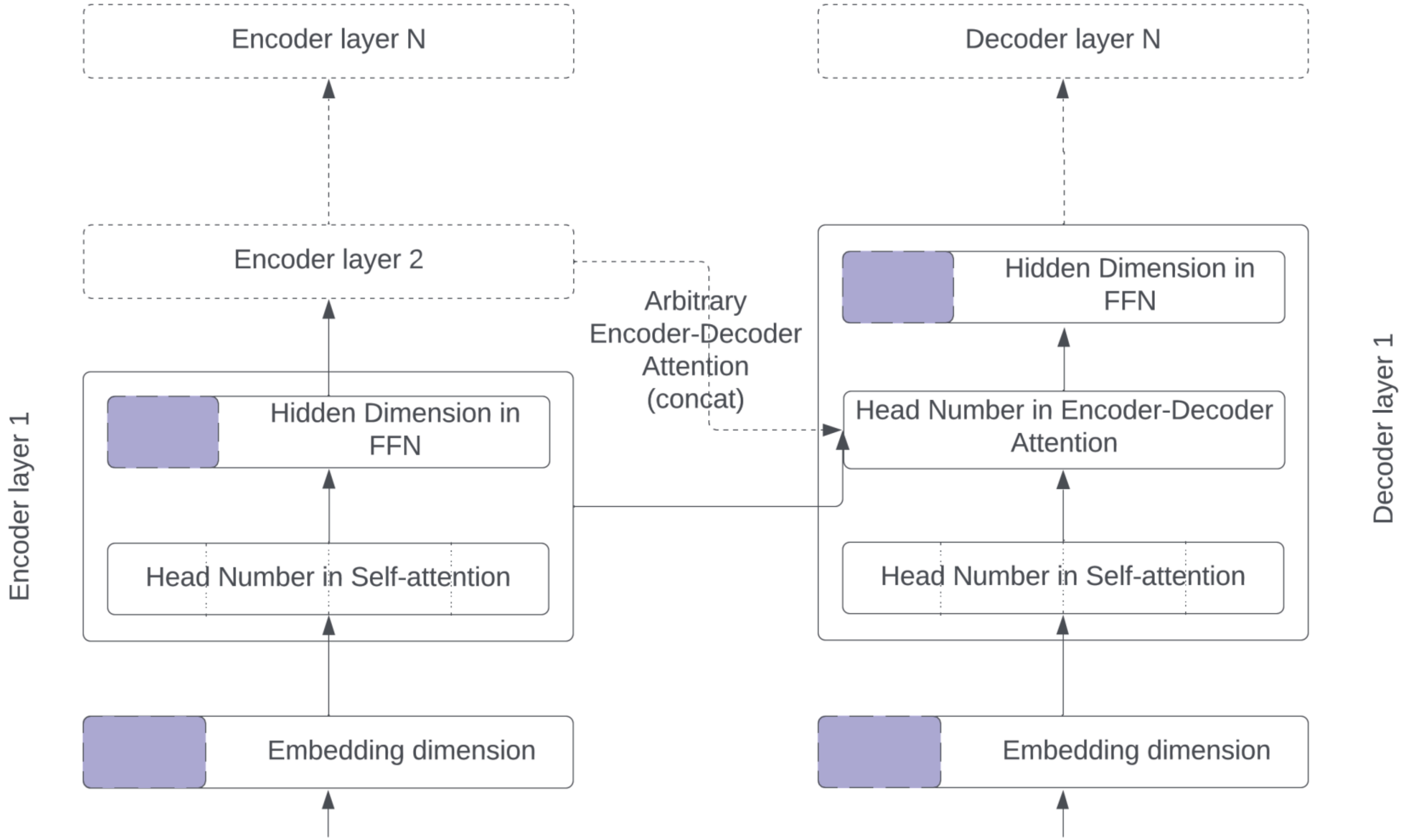


Figure 9: OFA achieves 80.0% top1 accuracy with 595M MACs and 80.1% top1 accuracy with 143ms Pixel1 latency, setting a new SOTA ImageNet top1 accuracy on the mobile setting.

Diagram credit: OFA ICLR 2020

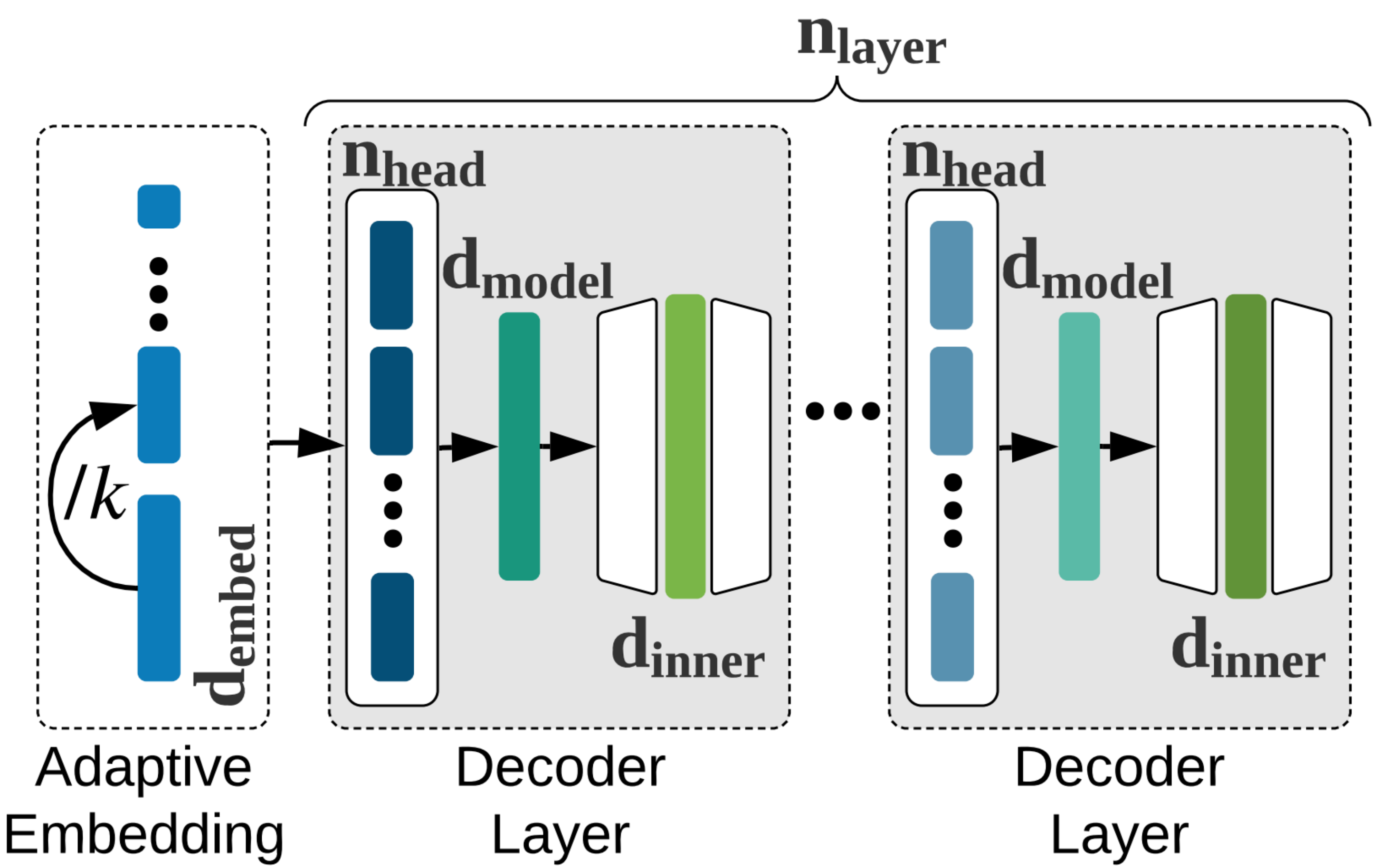
Recent Transformer-based Search Spaces

HAT: Hardware-Aware Transformers for Efficient Natural Language Processing



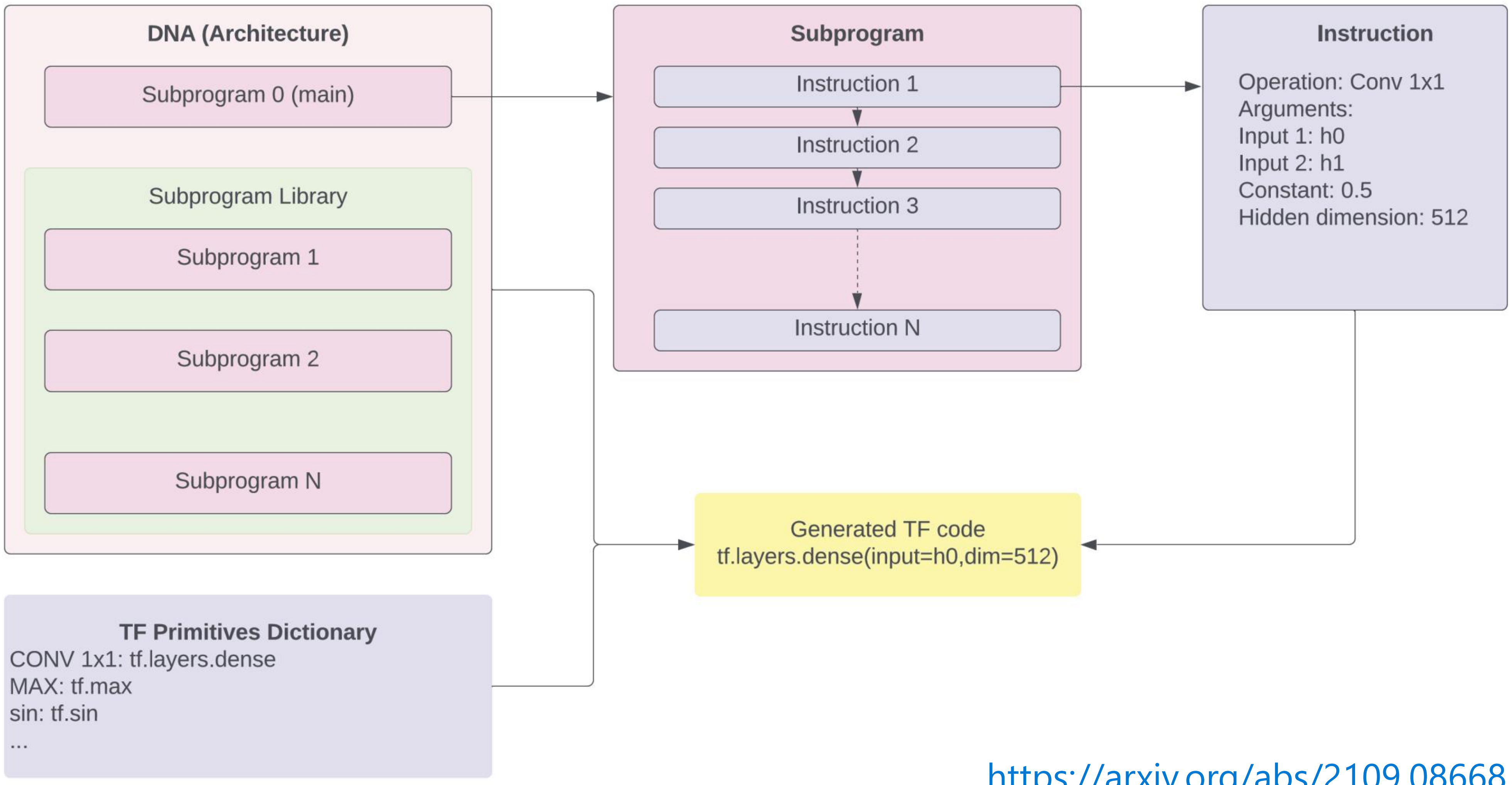
https://arxiv.org/abs/2005.14187

LiteTransformerSearch: Training-free On-device Search for Efficient Autoregressive Language Models



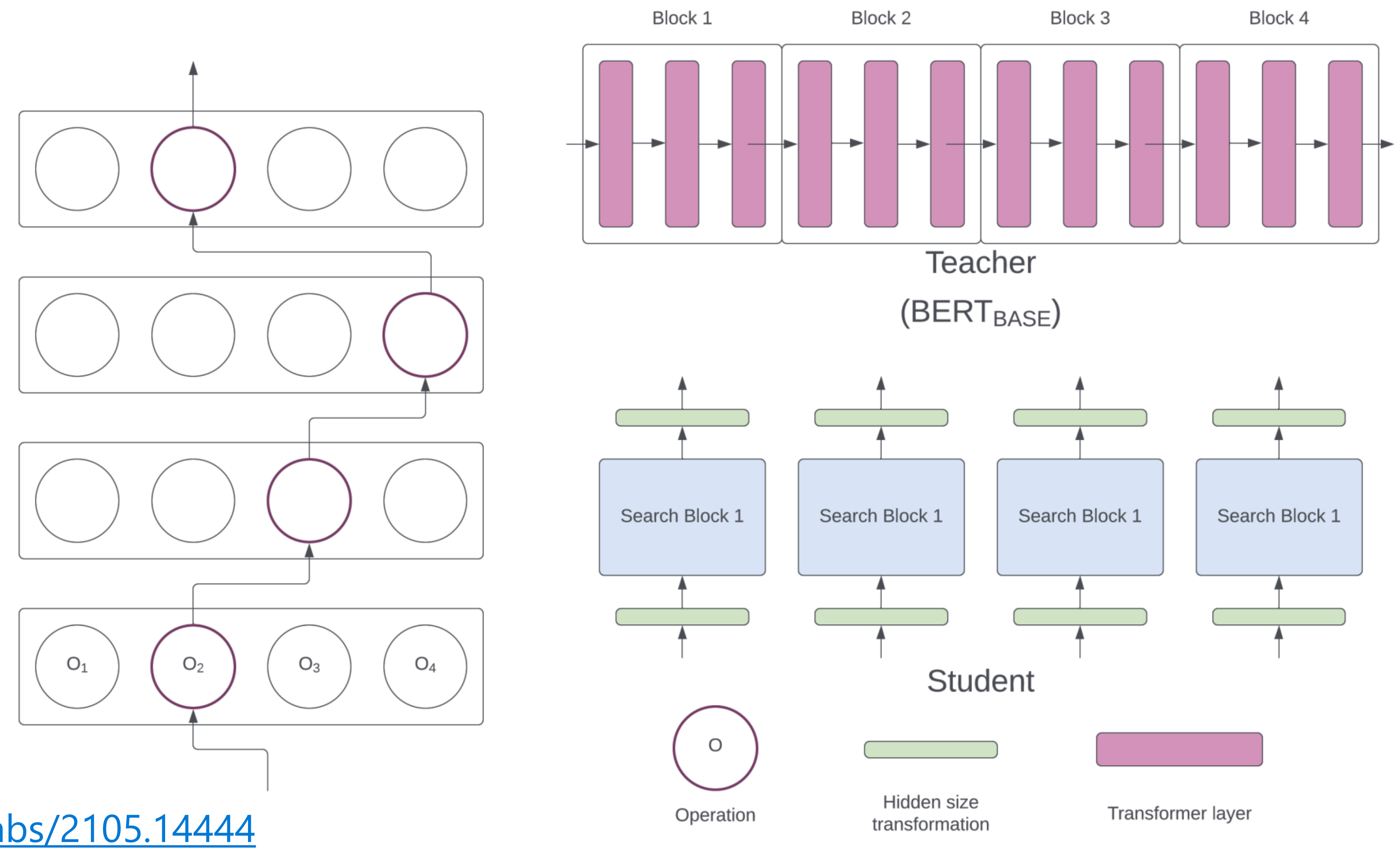
https://arxiv.org/pdf/2203.02094.pdf

Primer: Searching for Efficient Transformers for Language Modeling



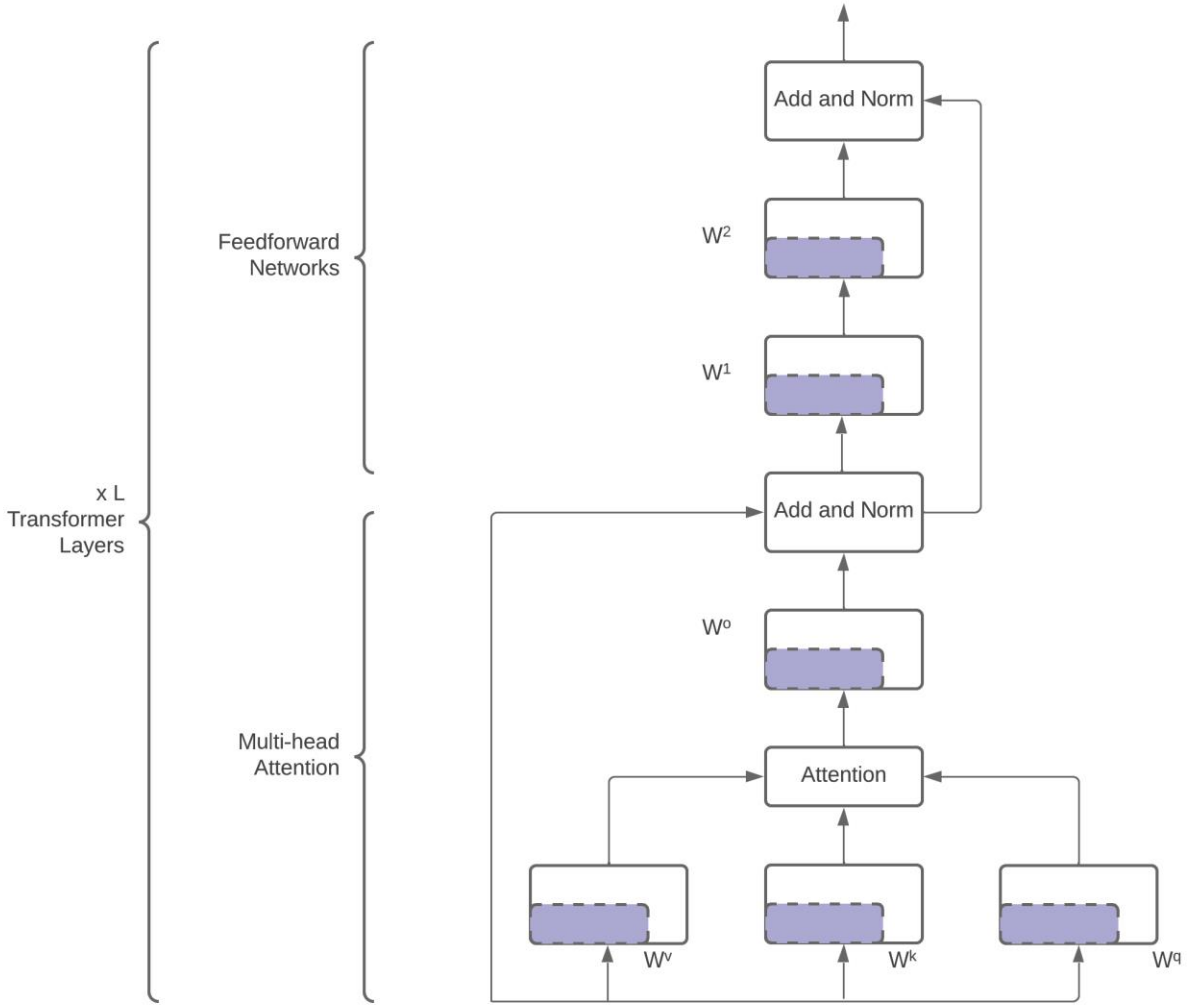
https://arxiv.org/abs/2109.08668

NAS-BERT: Task-Agnostic and Adaptive-Size BERT Compression with Neural Architecture Search



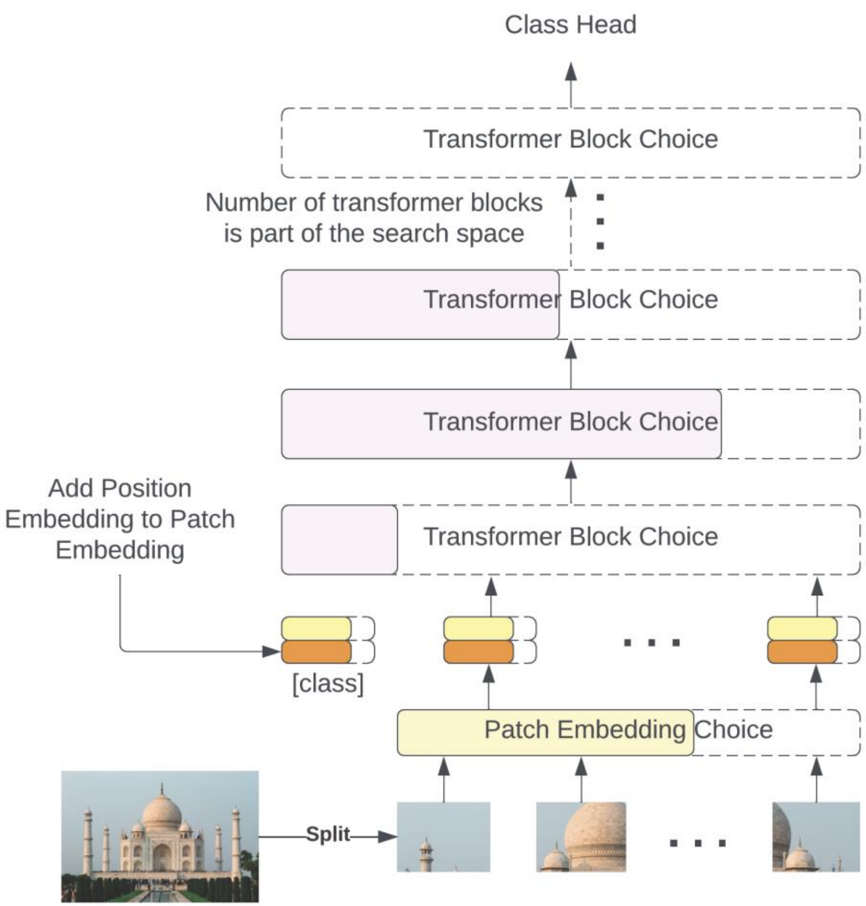
https://arxiv.org/abs/2105.14444

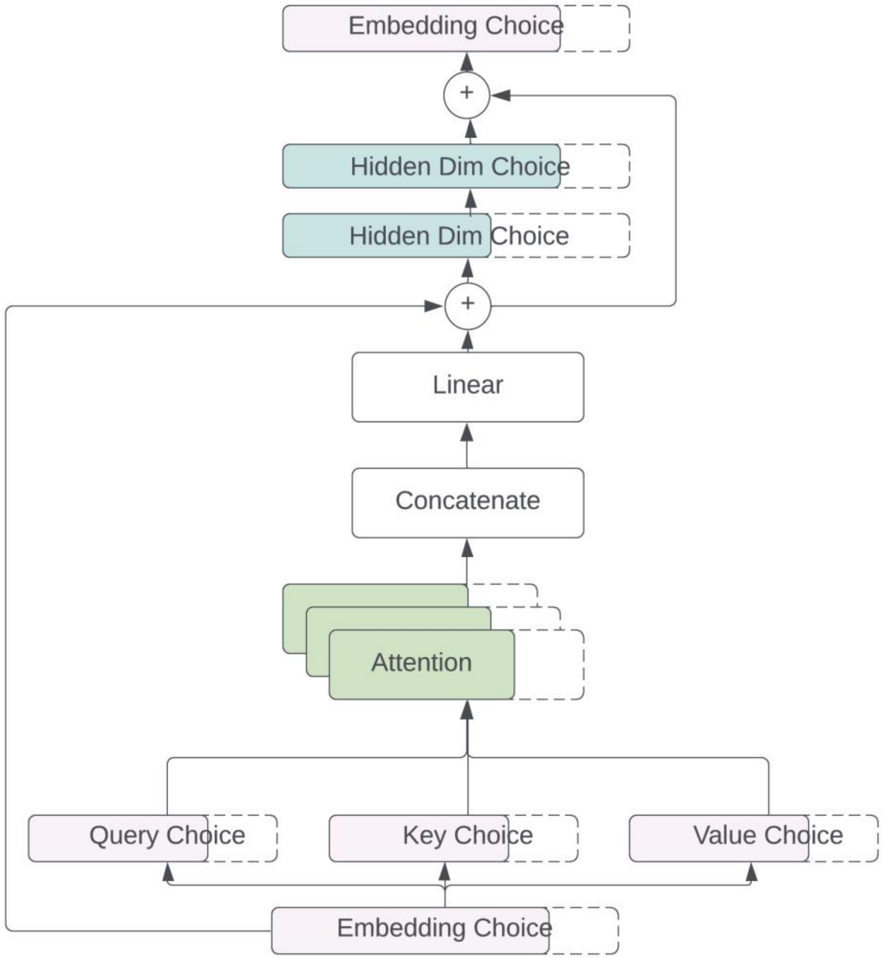
AutoTinyBERT: Automatic Hyper-parameter Optimization for Efficient Pre-trained Language Models



https://aclanthology.org/2021.acl-long.400.pdf

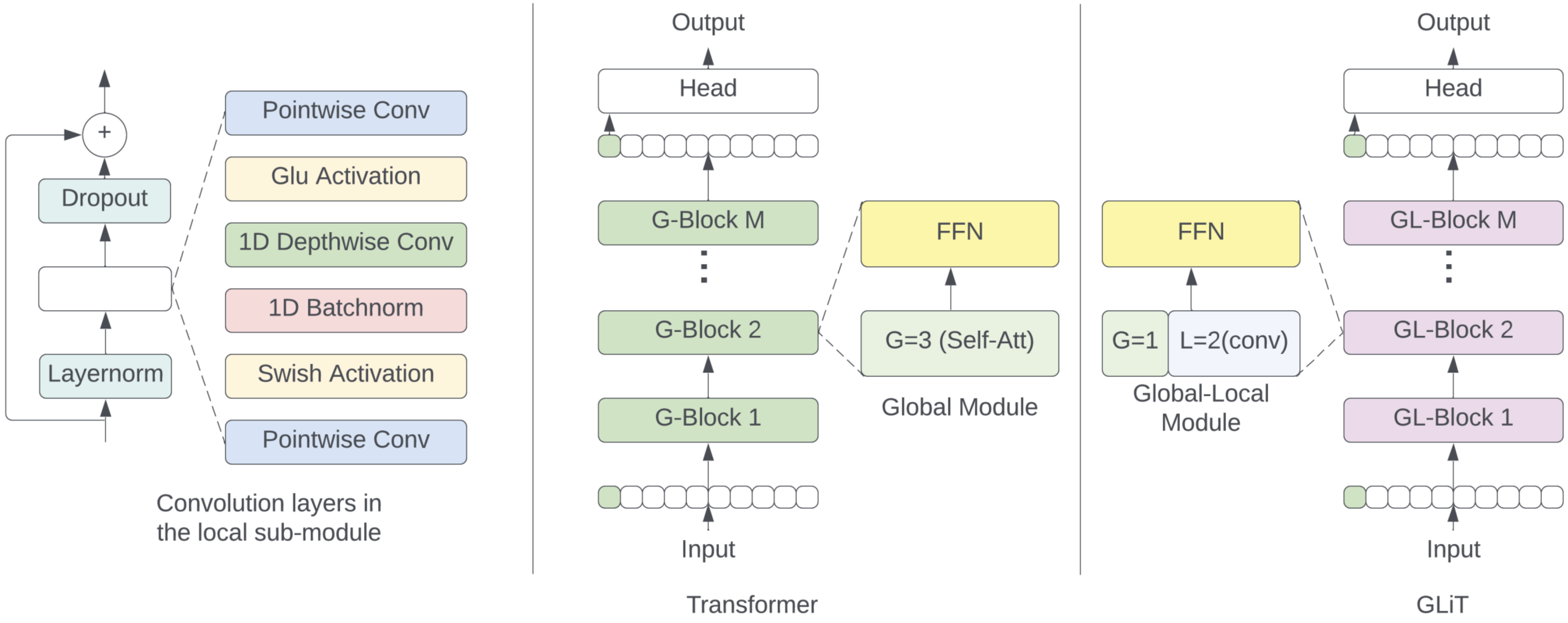
AutoFormer: Searching Transformers for Visual Recognition





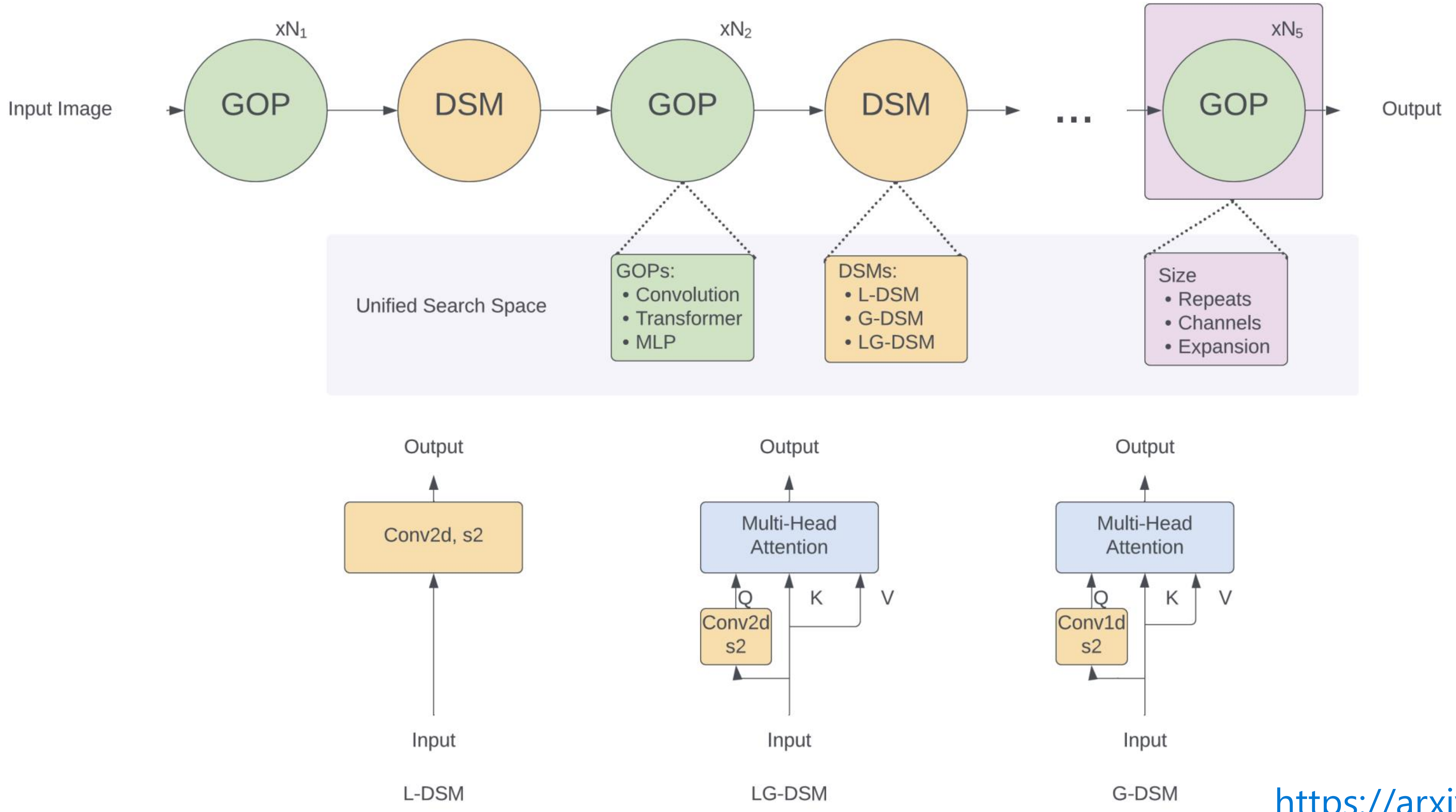
https://arxiv.org/abs/2107.00651

GLiT: Neural Architecture Search for Global and Local Image Transformer



https://arxiv.org/abs/2107.02960

UniNet: Unified Architecture Search with Convolution, Transformer, and MLP



https://arxiv.org/abs/2110.04035

Petridish: Efficient Forward Architecture Search
Hu et al, NeuRIPS 2019

Petridish overview

- Warm start
  - Inspired by gradient boosting.
- Expand the search tree
  - Focus on the most cost-effective ones.
  - Directly search the pareto-frontier.
- Predict performance.
  - Utilizing model initialization to select children to train.

The Cascade-Correlation Learning Architecture

Scott E. Fahlman and **Christian Lebiere**
August 29, 1991
CMU-CS-90-100

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Cascade-Correlation is a new architecture and supervised learning algorithm for artificial neural networks. Instead of just adjusting the weights in a network of fixed topology, Cascade-Correlation begins with a minimal network, then automatically trains and adds new hidden units one by one, creating a multi-layer structure. Once a new hidden unit has been added to the network, its input-side weights are frozen. This unit then becomes a permanent feature-detector in the network, available for producing outputs or for creating other, more complex feature detectors. The Cascade-Correlation architecture has several advantages over existing algorithms: it learns very quickly, the network determines its own size and topology, it retains the structures it has built even if the training set changes, and it requires no back-propagation of error signals through the connections of the network.

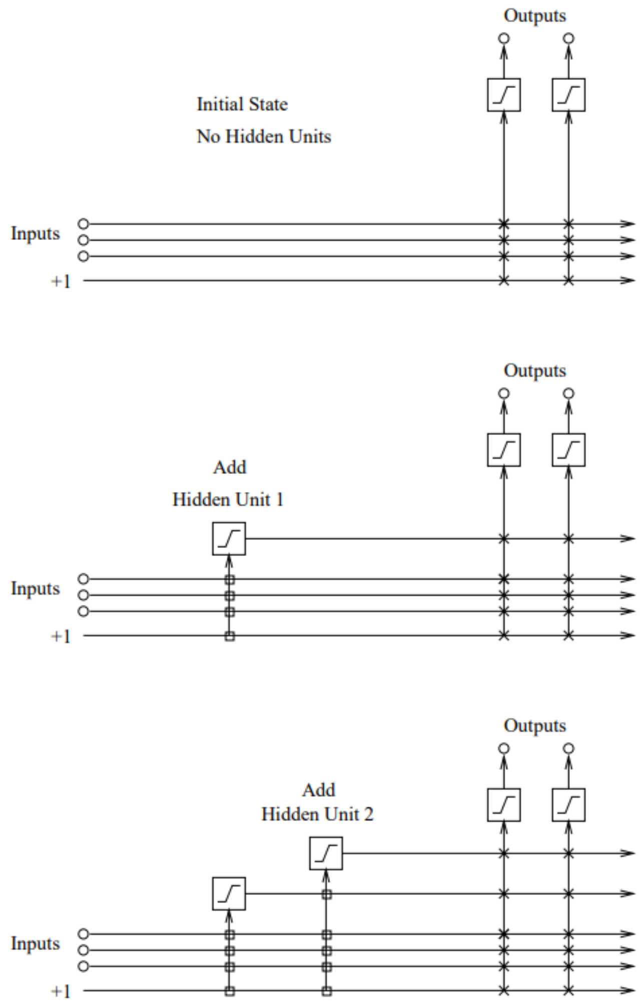
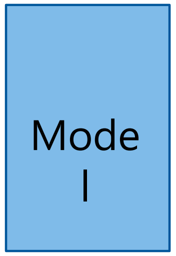
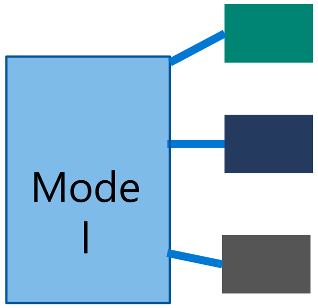


Figure 1: The Cascade architecture, initial state and after adding two hidden units. The vertical lines sum all incoming activation. Boxed connections are frozen, X connections are trained repeatedly.

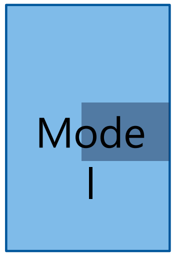
Incremental Training



Phase 0
Original model

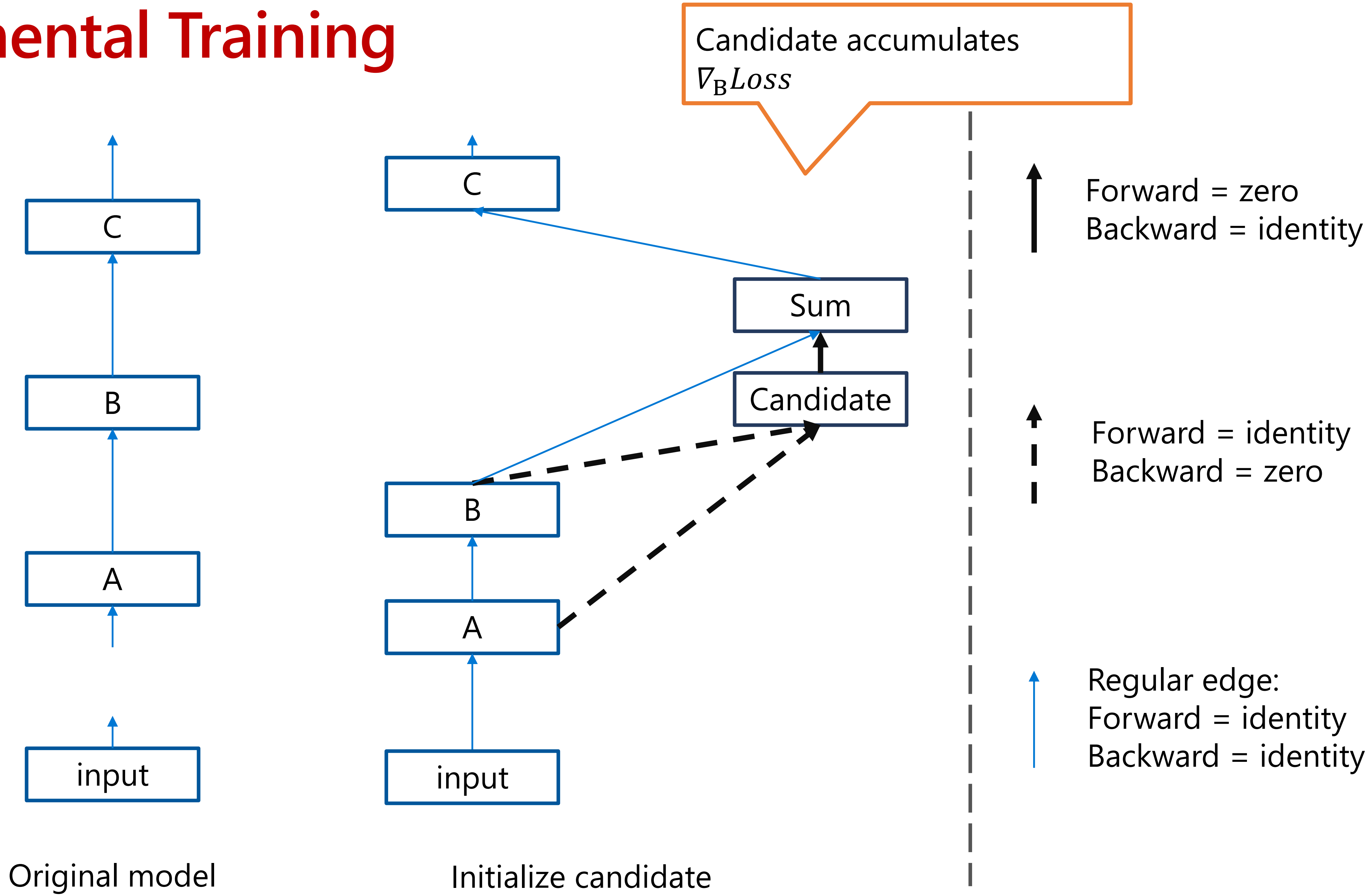


Phase 1
Initialize candidates, but do not allow candidates to affect the original model.

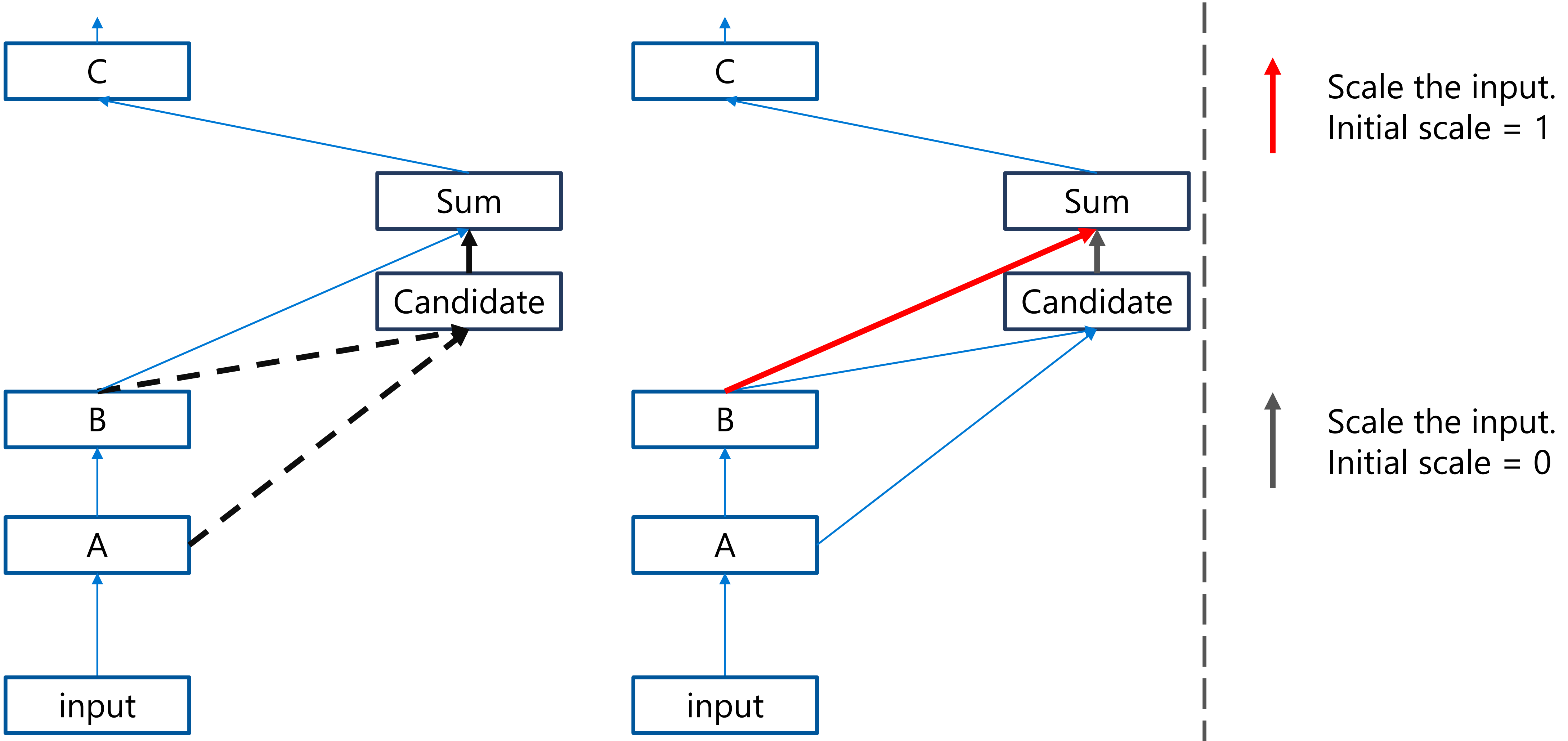


Phase 2
Officially add an candidate to model. Now the candidate can affect the original.

Incremental Training



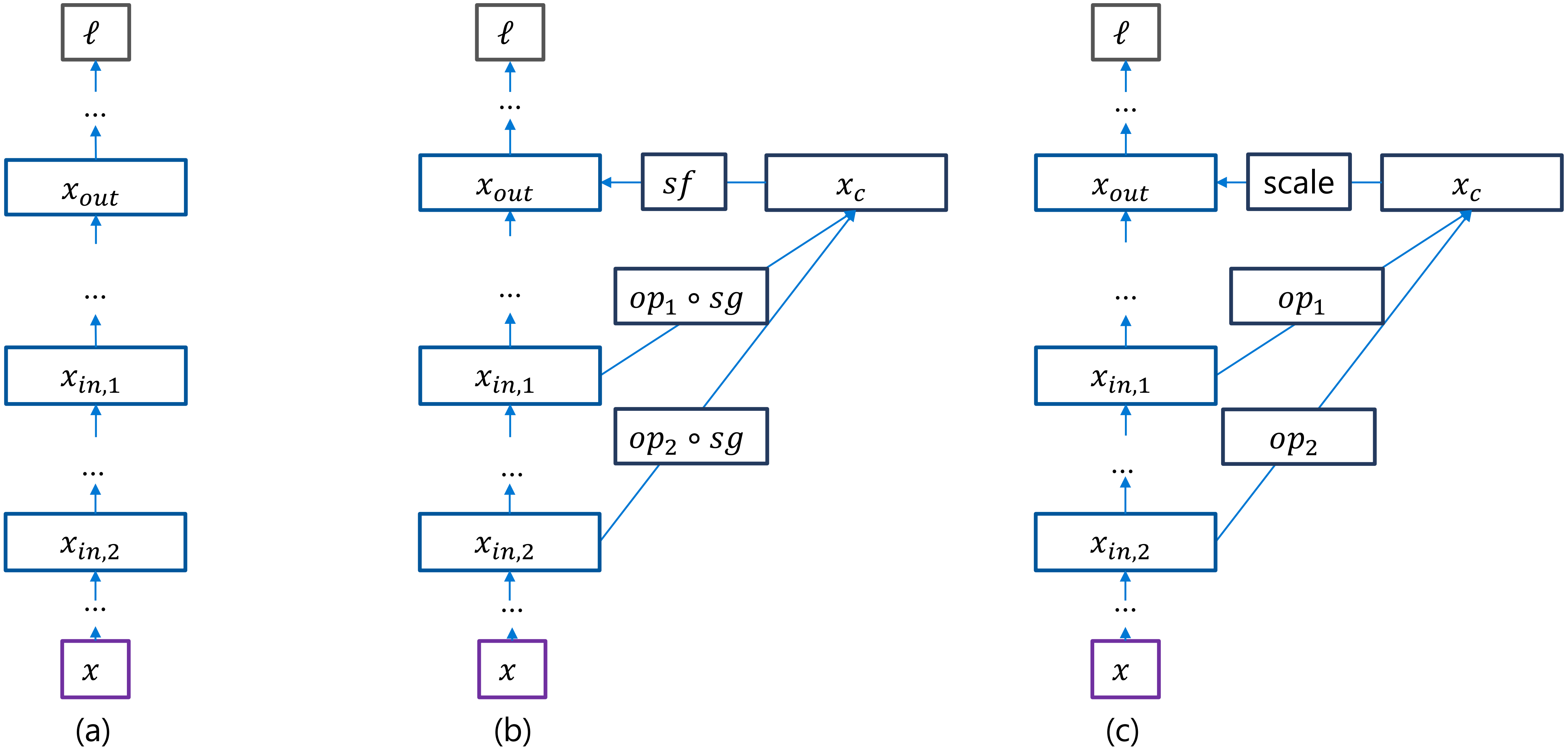
Incremental Training



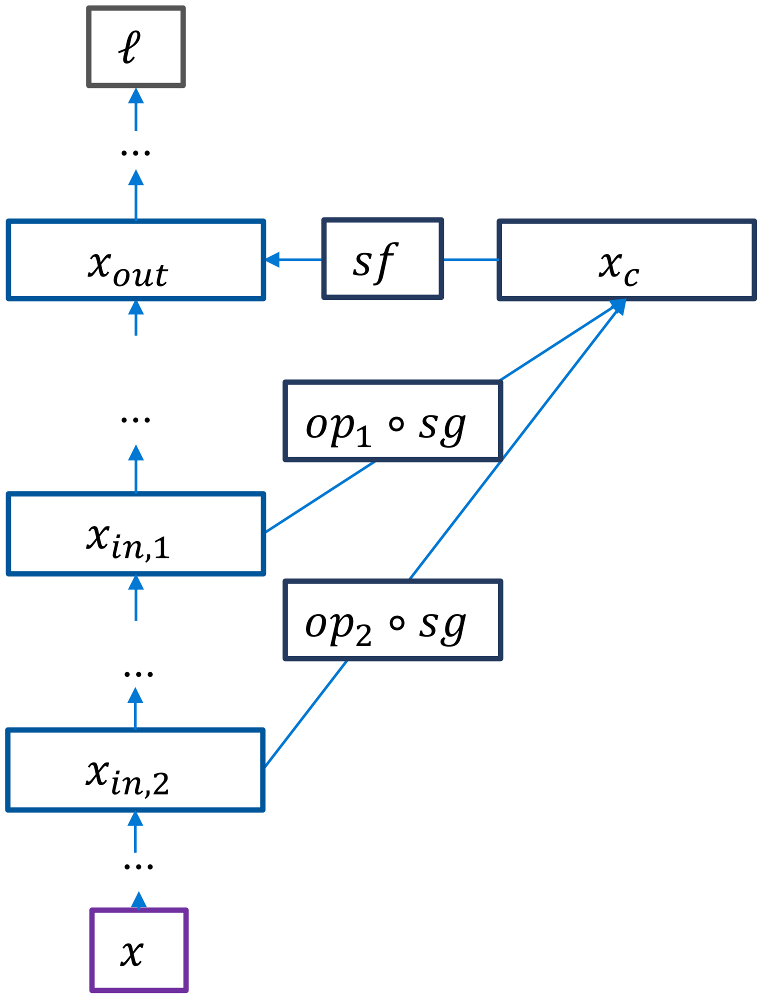
Initialize candidate

Officially add candidate to model

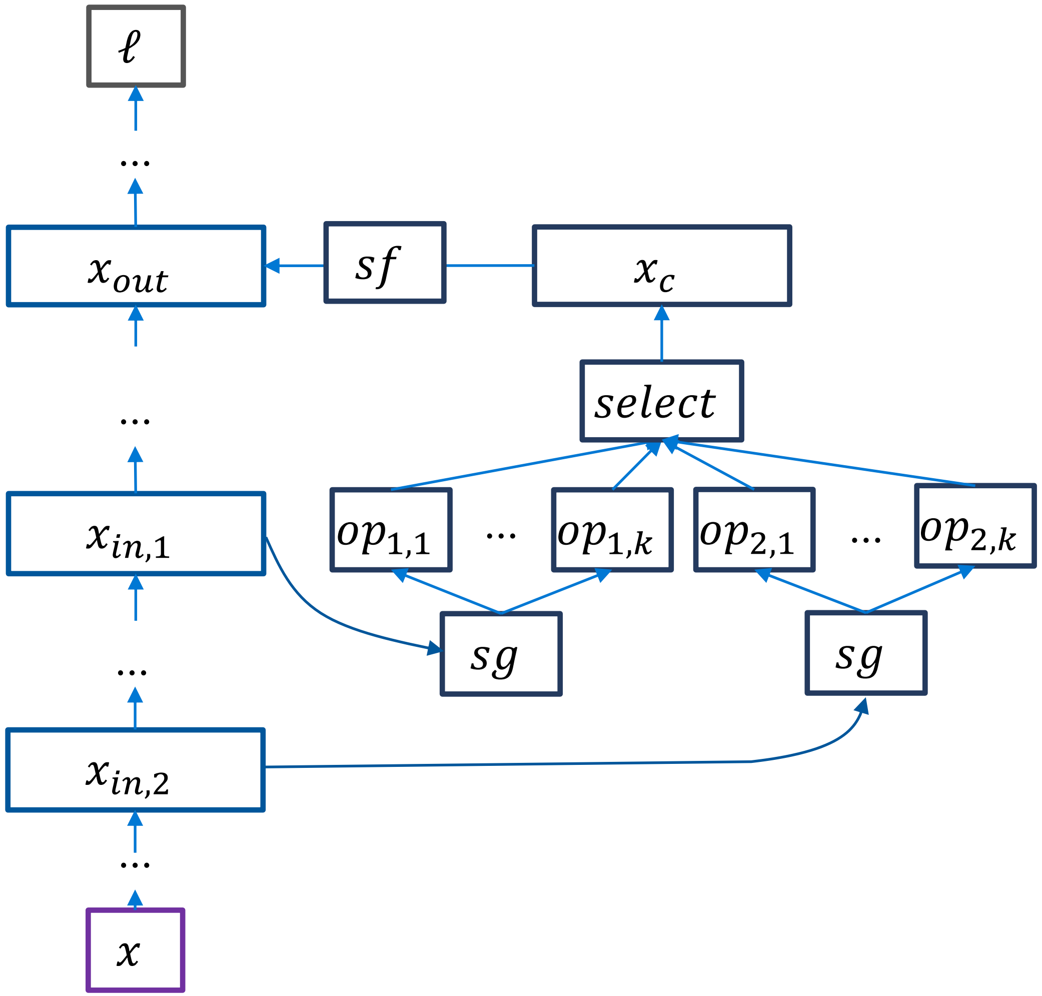
Incremental Training (Summary)



Incremental Training (Choice of Candidates)

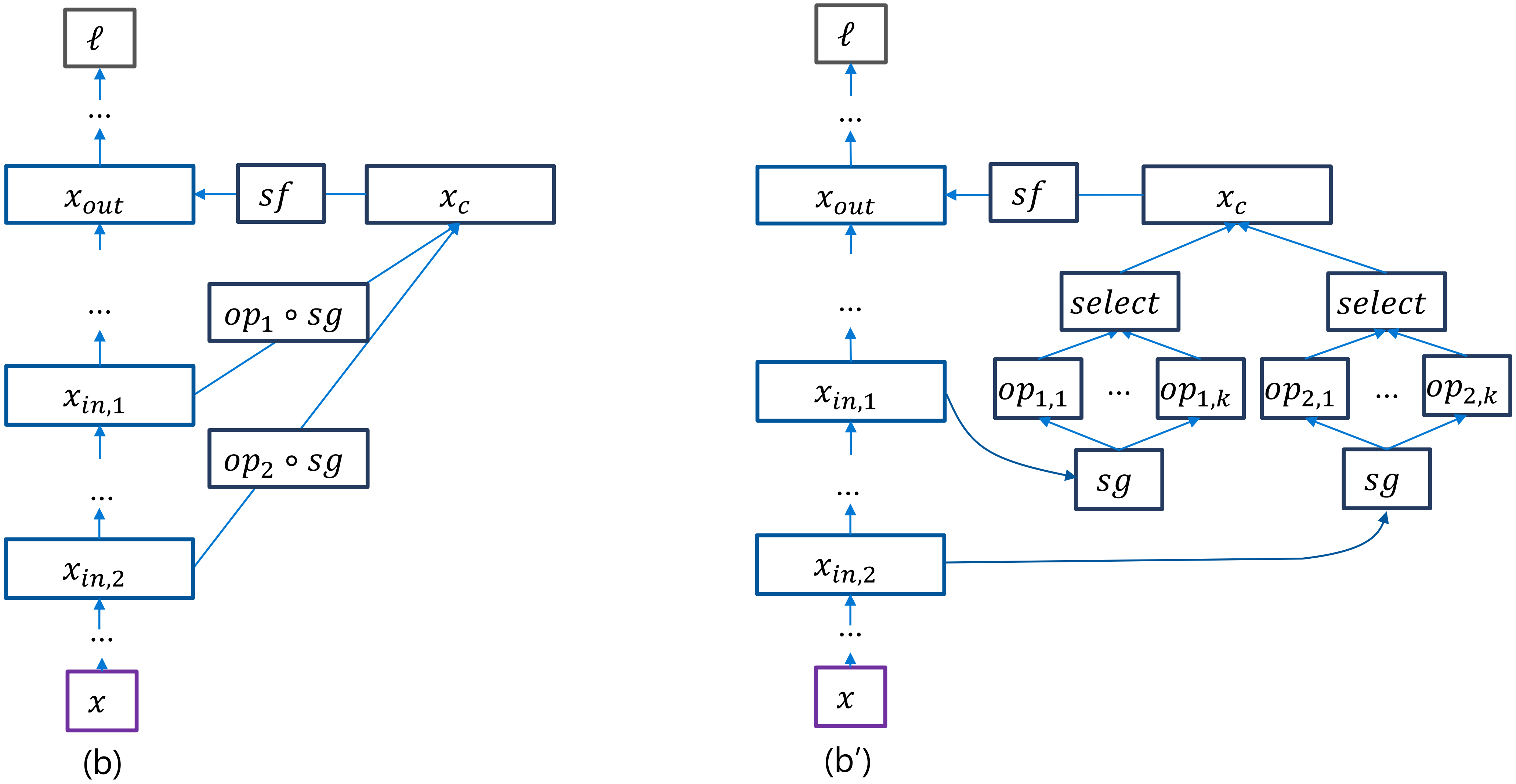


(b)



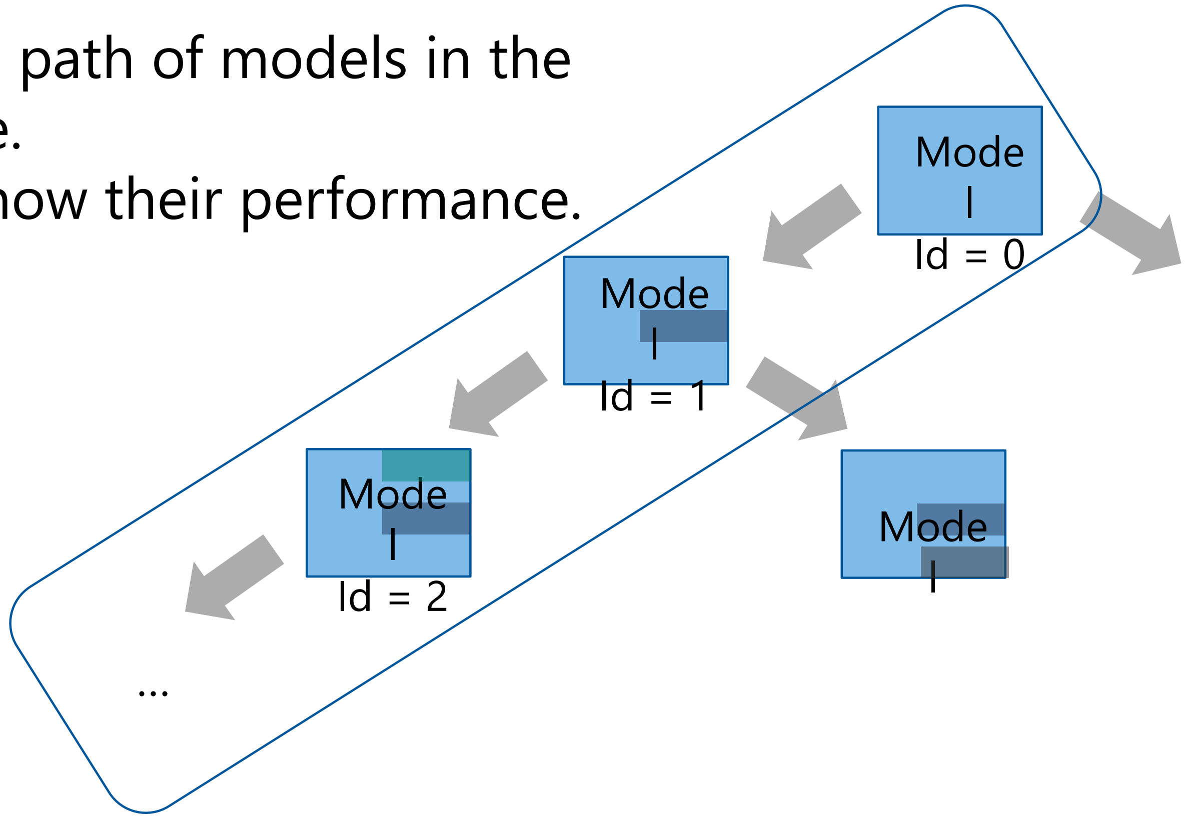
(b′)

Incremental Training (Choice of Candidates)



Incremental training during search

Consider a path of models in the search tree.
Want to know their performance.



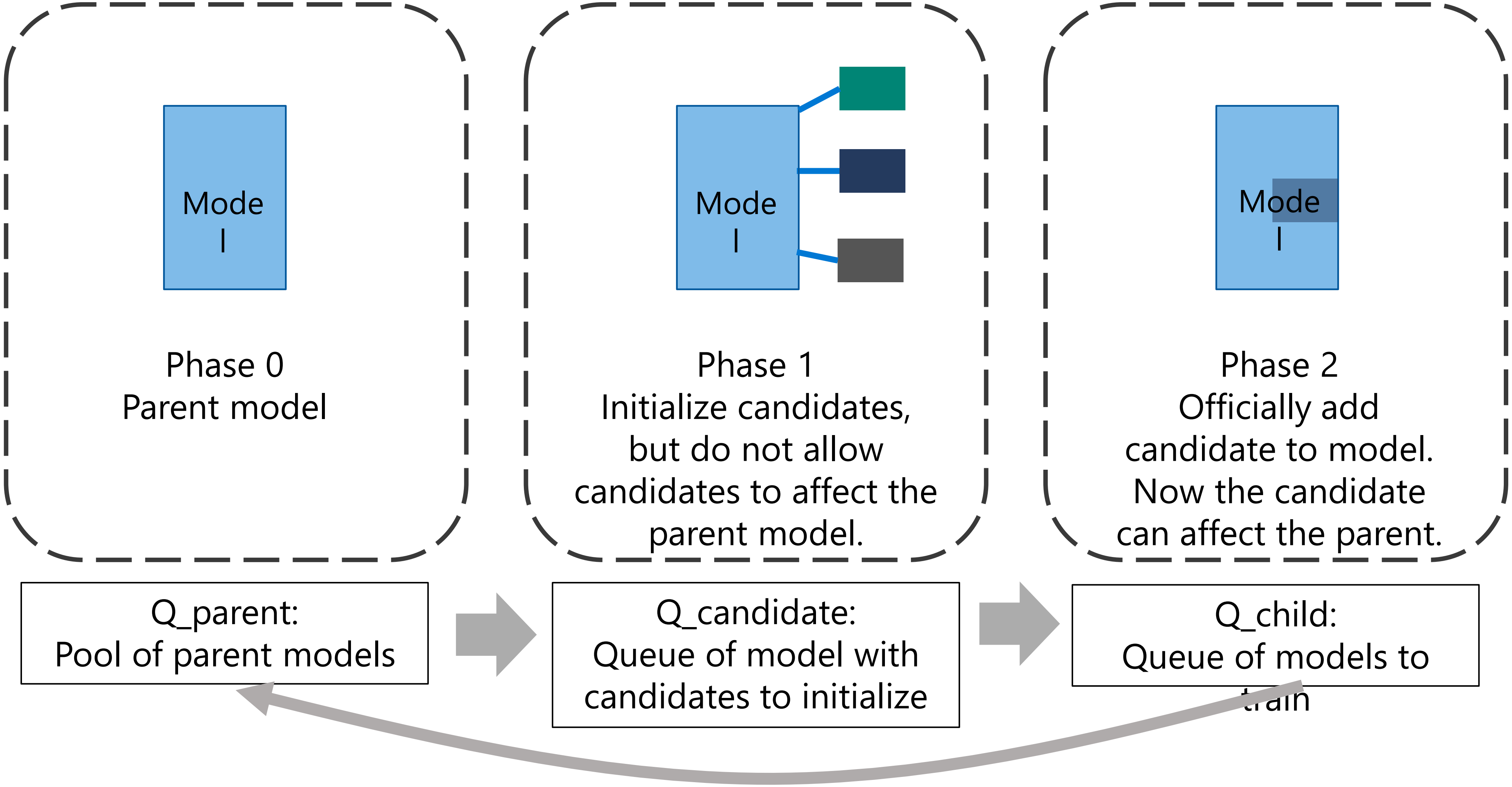
Option 1 (From-scratch) :

- Train models independently.
- 300 epochs per model

Option 2 (Incremental) :

- Start from parent; initialize children
- 40 epochs per model

Search on distributed systems

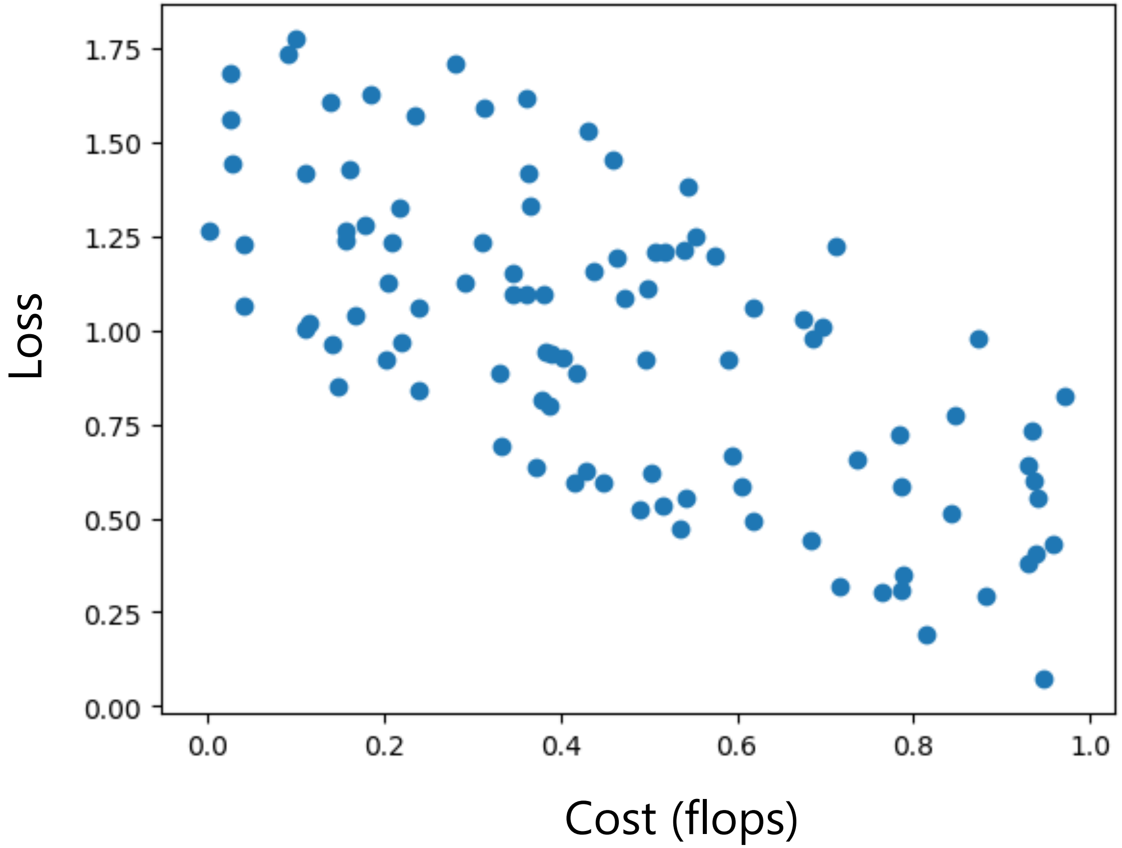


Search on distributed systems

- Q_parent: explore-exploit a diverse set of good models to extend.
- Q_candidate: initialize promising candidates
- Q_children: train promising children

- How do we know a model is good?

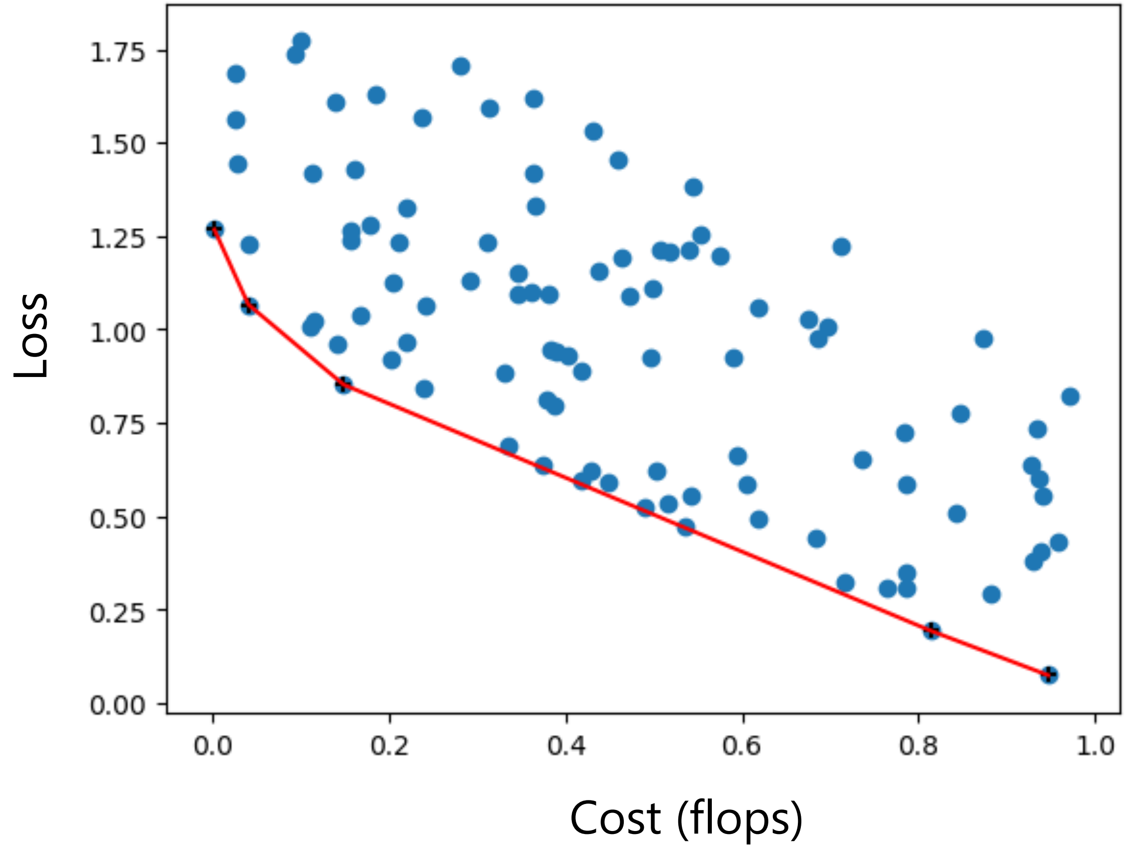
Expanding the Most Cost-efficient Models



This figure is for
illustration only

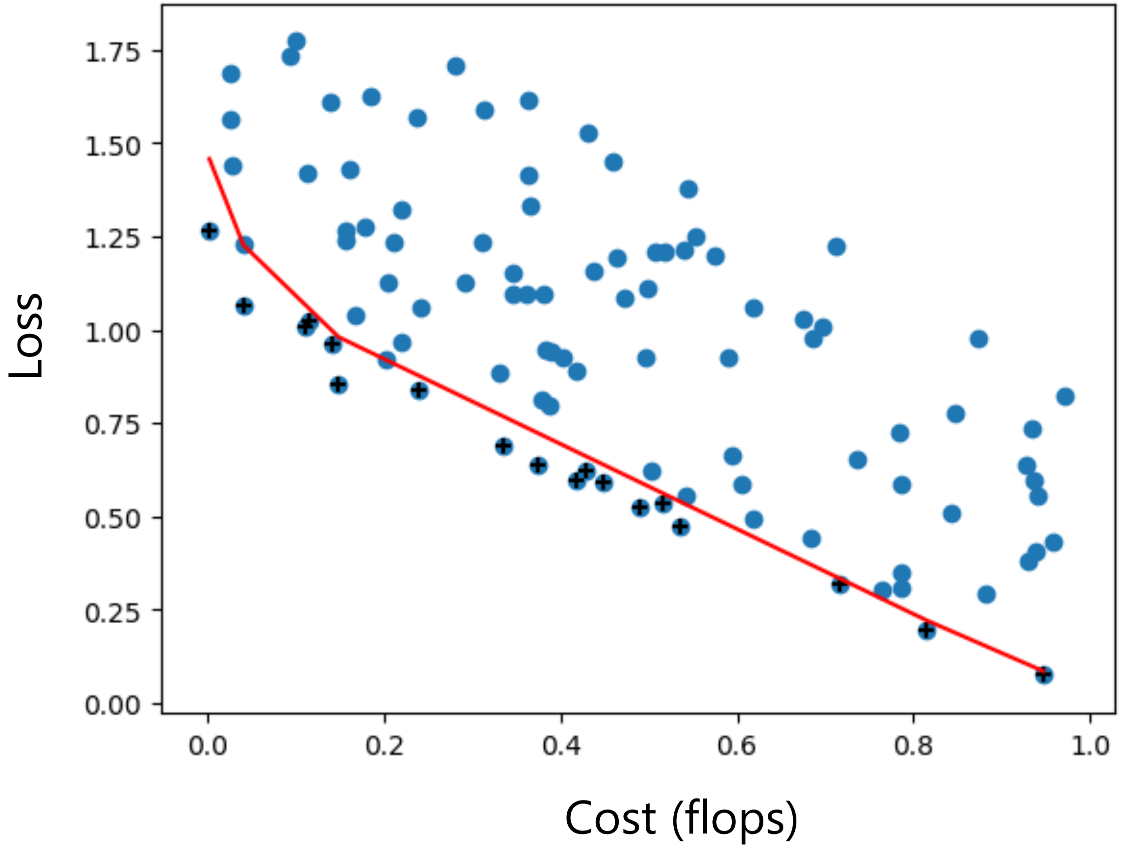
Expanding the Most Cost-efficient Models

- Convex hull



Expanding the Most Cost-efficient Models

- Epsilon- convex hull



Key advantage:

Method naturally produces a 'gallery' of models which are nearly-optimal for every serving time budget need.

This is critical to production serving needs.

Search on CIFAR10

Petridish on macro search space

Petridish on cell search space

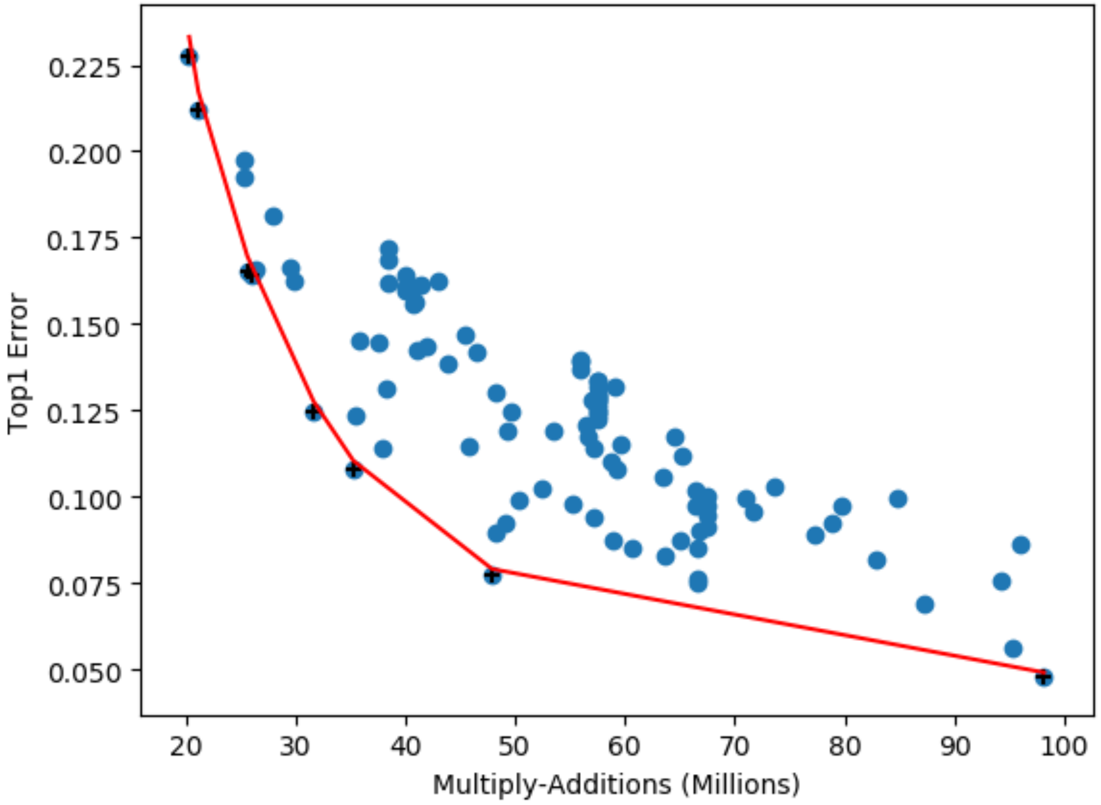
Method	# params (mil.)	Search (GPU-Days)	Test Error (%)
Zoph & Le (2017)†	7.1	1680+	4.47
Zoph & Le (2017) + more filters†	37.4	1680+	3.65
Real et al. (2017)†	5.4	2500	5.4
ENAS macro (Pham et al., 2018)†	21.3	0.32	4.23
ENAS macro + more filters†	38	0.32	3.87
Lemonade I (Elsken et al., 2018a)	8.9	56	3.37
Petridish initial model ($N = 6$, $F = 32$)	0.4	–	4.6
Petridish initial model ($N = 12$, $F = 64$)	3.1	–	3.06 ± 0.12
Petridish macro	2.2	5	2.83 | 2.85 ± 0.12
NasNet-A (Zoph et al., 2018)	3.3	1800	2.65
AmoebaNet-B (Real et al., 2018)	2.8	3150	2.55 ± 0.05
PNAS (Liu et al., 2017)†	3.2	225	3.41 ± 0.09
ENAS cell (Pham et al., 2018)	4.6	0.45	2.89
Lemonade II (Elsken et al., 2018a)	3.98	56	3.50
DARTS (Liu et al., 2019)	3.4	4	2.76 ± 0.09
SNAS (Xie et al., 2019)	2.8	1.5	2.85 ± 0.02
Luo et al. (2018)†	3.3	0.4	3.53
PARSEC (Casale et al., 2019)	3.7	1	2.81 ± 0.03
DARTS random (Liu et al., 2019)	3.1	–	3.29 ± 0.15
16 Random Models in Petridish space	2.27 ± 0.15	–	3.32 ± 0.15
Petridish cell w/o feature selection	2.50 ± 0.28	–	3.26 ± 0.10
Petridish cell	2.5	5	2.61 | 2.87 ± 0.13
Petridish cell more filters (F=37)	3.2	5	2.51 | 2.75 ± 0.21

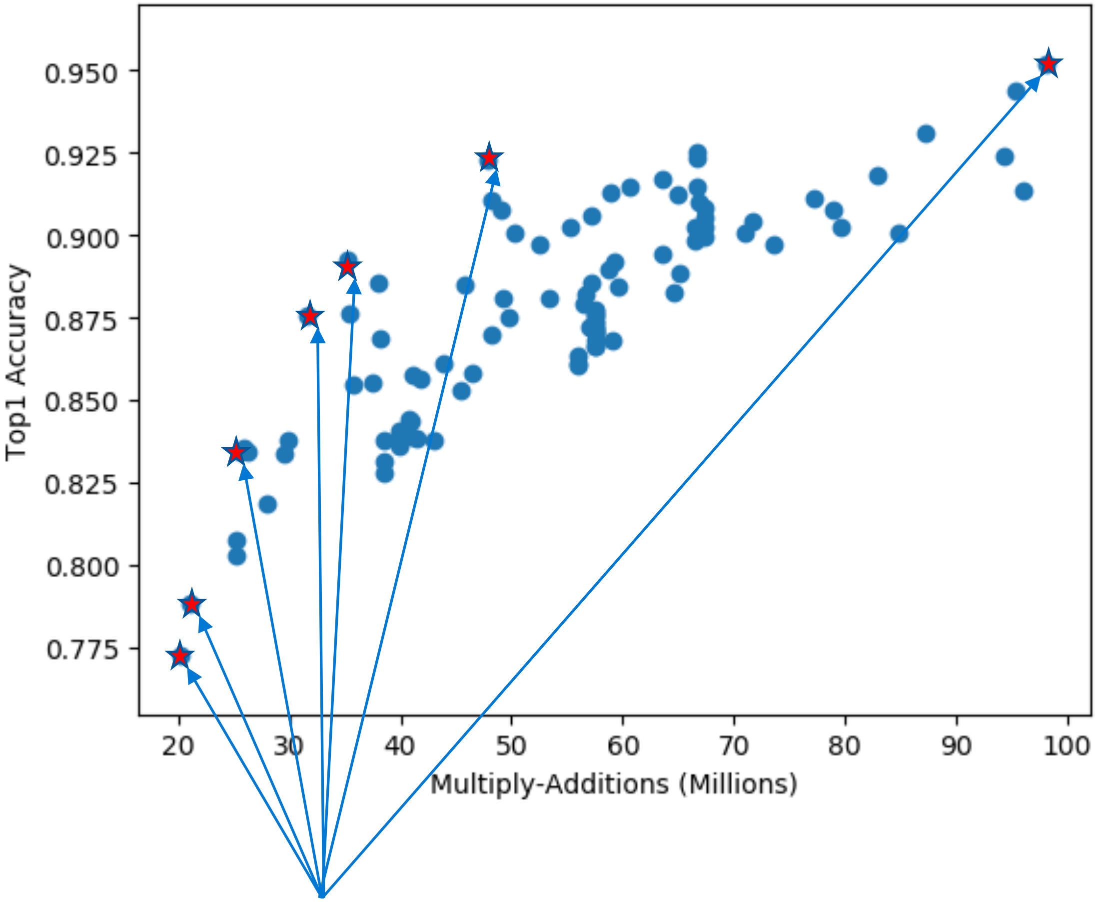
Transfer to ImageNet

Method	# params (mil.)	# multi-add (mil.)	Search (GPU-Days)	top-1 Test Error (%)
Inception-v1 (Szegedy et al., 2015)	6.6	1448	–	30.2
MobileNetV2 (Sandler et al., 2018)	6.9	585	–	28.0
NASNet-A (Zoph et al., 2017)	5.3	564	1800	26.0
AmoebaNet-A (Real et al., 2018)	5.1	555	3150	25.5
PNAS (Liu et al., 2017a)	5.1	588	225	25.8
DARTS (Liu et al., 2019)	4.9	595	4	26.9
SNAS (Xie et al., 2019)	4.3	522	1.6	27.3
Proxyless (Han Cai, 2019)†	7.1	465	8.3	24.9
Path-level (Cai et al., 2018)†	–	588	8.3	25.5
PARSEC (Casale et al., 2019)	5.6	–	1	26.0
Petridish macro (N=6,F=44)	4.3	511	5	28.5 | 28.7 ± 0.15
Petridish cell (N=6,F=44)	4.8	598	5	26.0 | 26.3 ± 0.20

No domain-knowledge injection in architecture design at all!

Search Once, Deploy Everywhere!





Example search on CIFAR10

Train models on the frontier with every orthogonal trick!
(data augmentation, distillation….)

Reproducibility, Fair Comparison and Best Practices!

Difficult to compare approaches!

- Search spaces and datasets.
- Training routine used.
  - Does it have all the tips and tricks?
- Hardware-Software used.
  - TPU vs. GPU vs. driver version vs. cuda version vs. framework.
- Stochasticity in training on GPUs.

NAS Evaluation is Frustratingly Hard, Yang et al., ICLR 2020
Random Search and Reproducibility for Neural Architecture Search, Li and Talwalker, UAI 2020

Benchmarks Help!

- NASBench-101
  - Cannot evaluate weight-sharing, DARTS-like search spaces.
- NASBench-201
  - Uses different search space than 101.
- NASBench-1Shot1
  - Leverages 101 to make it amenable for weight-sharing.
- NASBench-301
  - 60,000 models sampled from DARTS search space trained on CIFAR10 to train surrogate model.
- NASBench-NLP
  - 14k RNN architectures trained on Penn Tree Bank.
- NASBench-ASR
  - 8k architectures trained on TIMIT audio dataset for speech recognition.

Benchmarks Help!

- NAS-HPO-Bench-II
  - 4K cell-based CNNs with different learning rates, batch sizes.
- HW-NAS-Bench
  - Evaluate NAS-Bench-201 and FBNet search spaces on 6 devices (edge devices, FPGA, ASIC).
- On Network Design Spaces for Visual Recognition
  - Over 100k architectures evaluated on CIFAR-10 from different search spaces.
- NAS-Bench-Suite
  - Collection of NAS Benchmarks through unified interface.
- NAS-Bench-360
  - 10 diverse tasks which are not just traditional vision tasks.

Checklist *Before* Starting Project

The NAS Best Practices Checklist (version 1.0.1, Nov. 1st, 2021)
by Marius Lindauer and Frank Hutter

Best practices for releasing code

For all experiments you report, check if you released:

- ☐ Code for the training pipeline used to evaluate the final architectures
- ☐ Code for the search space
- ☐ The hyperparameters used for the final evaluation pipeline, as well as random seeds
- ☐ Code for your NAS method
- ☐ Hyperparameters for your NAS method, as well as random seeds

Note that the easiest way to satisfy the first three of these is to use *existing* NAS benchmarks, rather than changing them or introducing new ones.

https://www.automl.org/nas_checklist.pdf

Journal of Machine Learning Research 21 (2020) 1-18 Submitted 1/20; Revised 11/20; Published 11/20

Best Practices for Scientific Research on Neural Architecture Search

Marius Lindauer LINDAUER@TNT.UNI-HANNOVER.DE
Leibniz University of Hannover
Hannover, 30167, Germany

Frank Hutter FH@CS.UNI-FREIBURG.DE
University of Freiburg & Bosch Center for Artificial Intelligence
Freiburg im Breisgau, 79110, Germany

https://www.jmlr.org/papers/volume21/20-056/20-056.pdf

NAS Frameworks

https://github.com/automl/NASLib

https://github.com/microsoft/archai

NAS Frameworks

aw_nas: A Modularized and Extensible NAS Framework

Maintained by NICS-EFC Lab (Tsinghua University) and Novauto Technology Co. Ltd. (Beijing China).

https://github.com/walkerning/aw_nas

PyGlove: Manipulating Python Programs

https://github.com/google/pyglove

Is NAS solved?

No fully general solution yet but useful successes!

Still, lots of domain knowledge injection into the process.

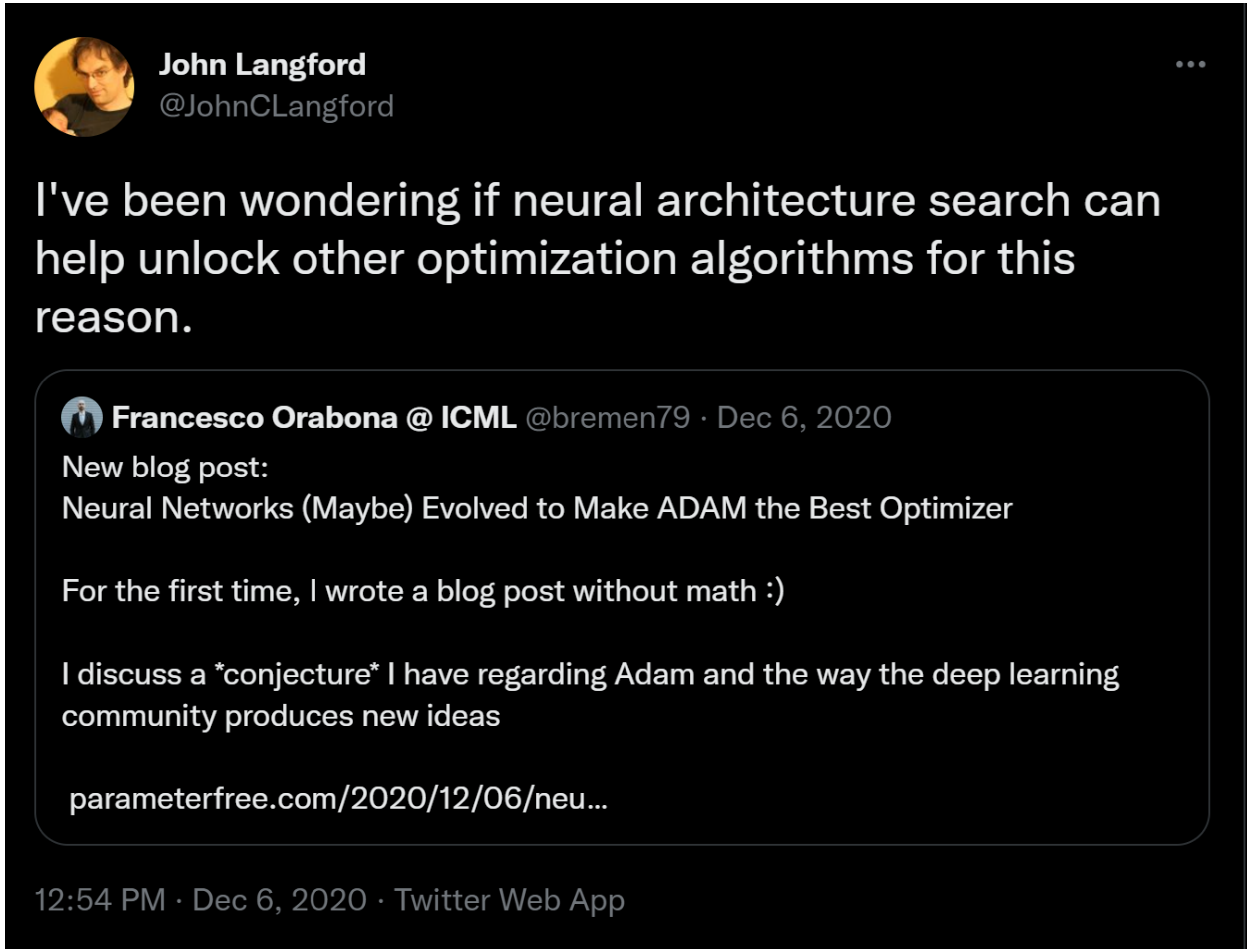
Tricks and tips needed for vision datasets are completely different from language or speech datasets (to be SOTA).

Need diverse benchmarks/tasks and rigorous reporting.

Hyperparameters are set to magic constants.

Open Problem 1: Optimizers and Learning Rate Schedules

- Are we handicapped by current optimizers?
- Rank of architectures in NAS-Bench-101 varies drastically on switching optimizer!
  - HW: Try this on other benchmarks!

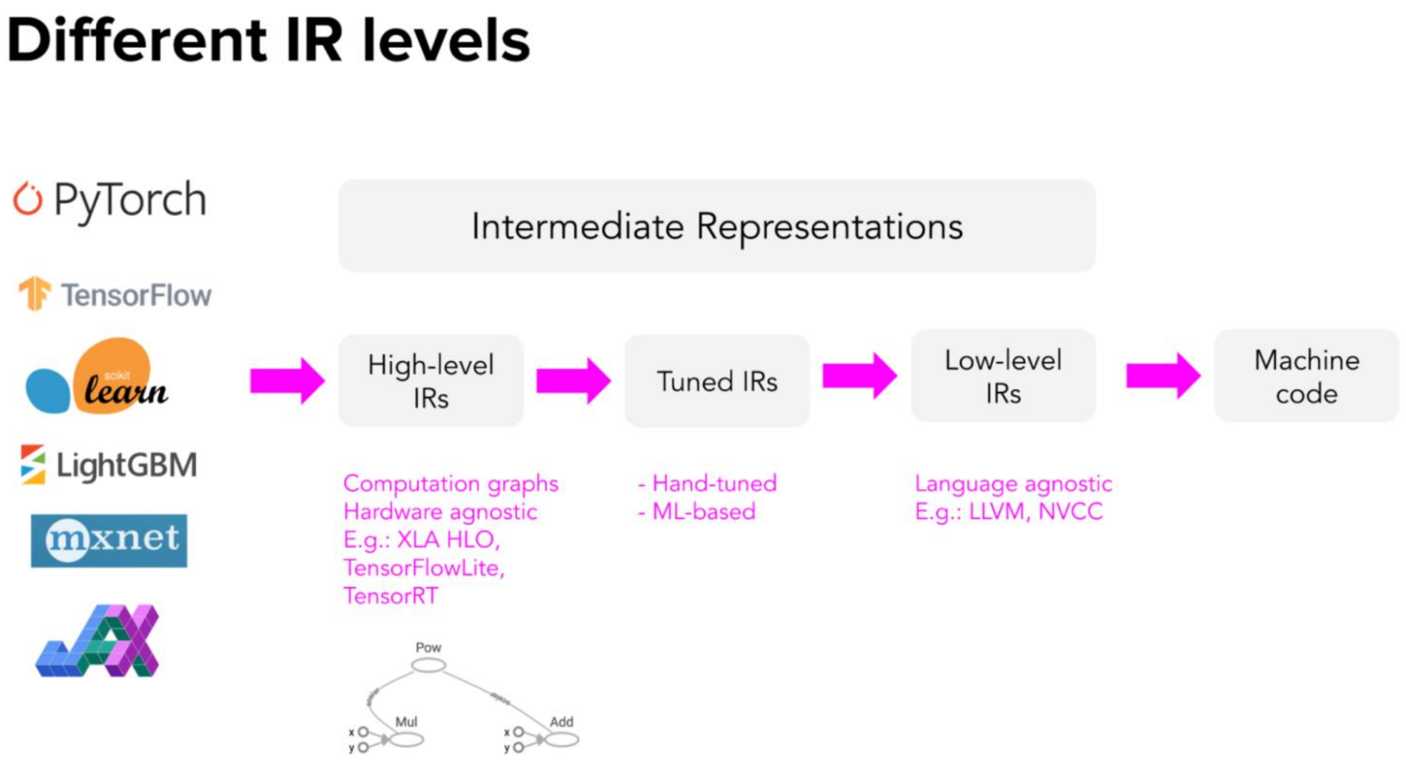


Tweet Dec 2020

Open Problem 2: Deep Learning Compilers!

"Pattern matching" for common manually designed architectures!

Compilers in the inner search loop will detect and optimize operator combinations that are commonly used!



Source: huyenchip

Questions?

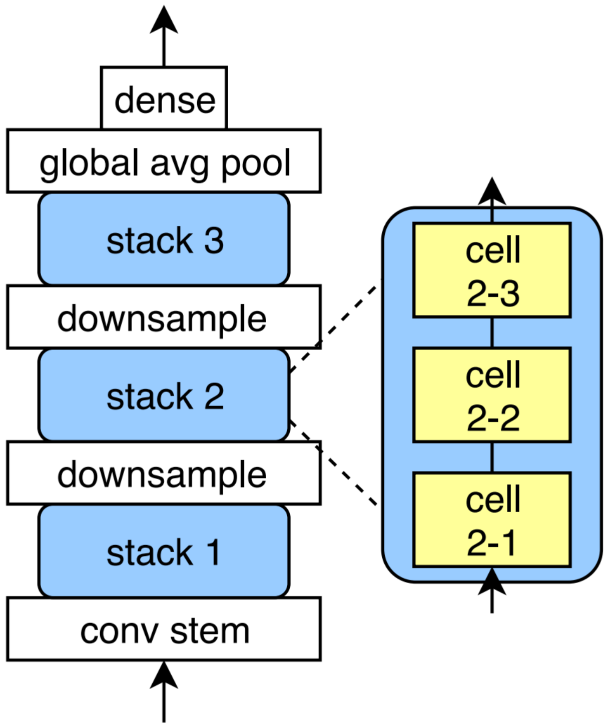
Appendix

Benchmarks to the Rescue: Tabular

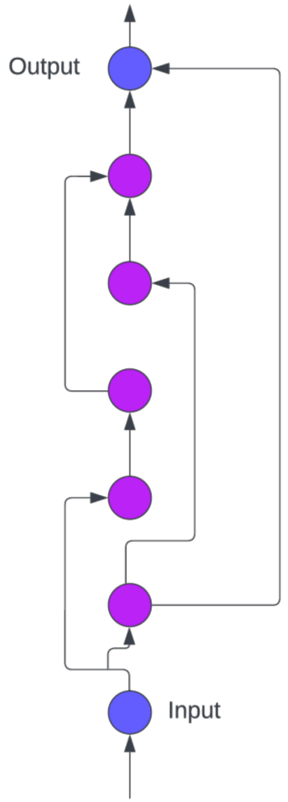
Arch ID	Training Error	Validation Error	Training Duration	Validation Duration	Test Error
0000001	0.15	0.18	632	10	0.21
0000002	0.33	0.41	515	8	0.46
0000003	0.28	0.22	585	11	0.23
…	…	…	…	…	…

Train *every* architecture in the search space.
Save all logs and data in a table.
NAS can be run on a laptop!

NAS-Bench-101



Skeleton

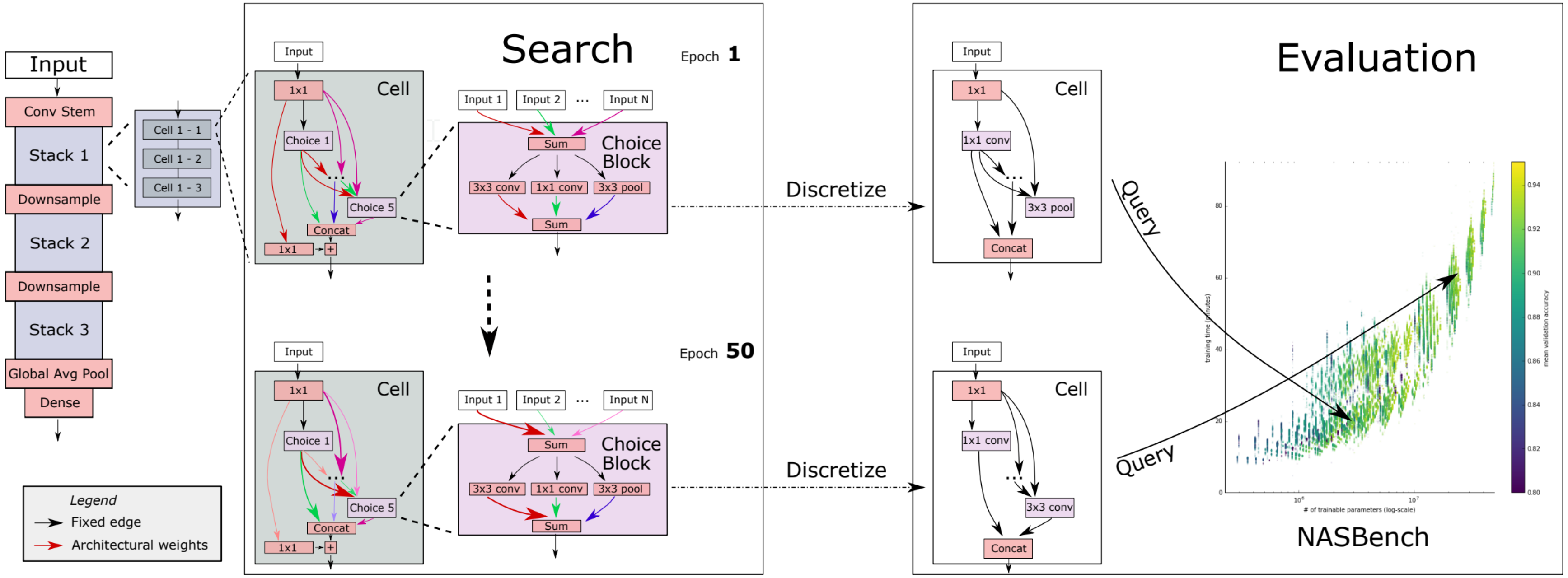


Example Cell

- Maximum 7 nodes per cell.
- Nodes are one of 3 operators.
  - 3x3 conv
  - 1x1 conv
  - 3x3 max pool
- Edges are tensors.
- Max edges 9 in a cell.
- 423k unique architectures.
- Trained on CIFAR10.
  - 4,12,36,108 epochs

NAS-Bench-101: Towards Reproducible Neural Architecture Search, Ying et al., ICML 2019

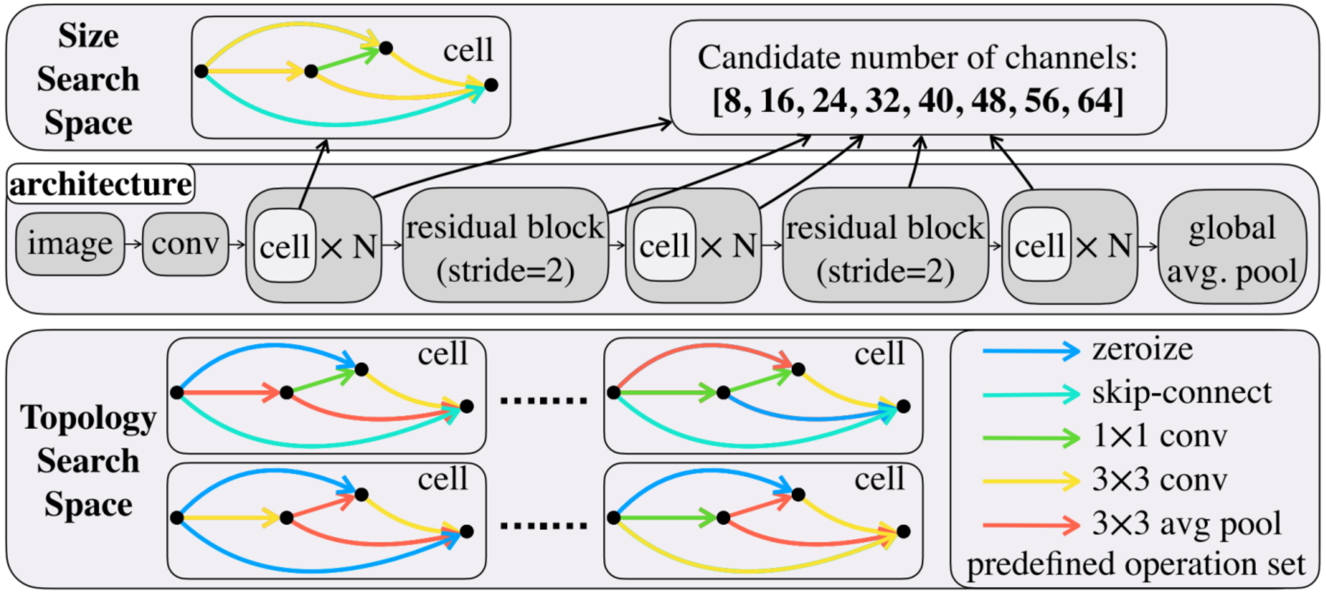
NAS-Bench-1Shot1



- Not possible to evaluate one-shot (weight-sharing) methods on NAS-Bench-101.
- Search space does not contain all possible cells (edges restricted <=9).
- NAS-Bench-1Shot1 defines a new search space.
  - Reuses 101 to allow for one-shot methods to be evaluated.

NAS-Bench-1Shot1: Benchmarking and Dissecting One-shot Neural Architecture Search,
Zela et al., ICLR 2020

NATS-Bench (NAS-Bench-201)



- One-shot compatible.
- Two search spaces:
  - Topological space: 6.5k unique
  - Size space: 32.8k unique
- 3 datasets:
  - CIFAR10
  - CIFAR100
  - ImageNet16-120
- Trained with 12, 20, 90 epochs.

NATS-Bench: Benchmarking NAS Algorithms for Architecture Topology and Size, Dong et al., TPAMI 2021