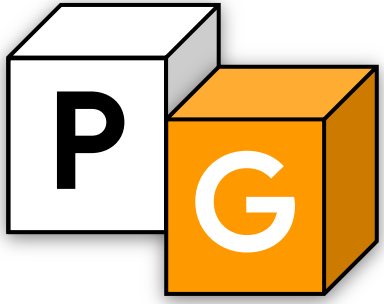
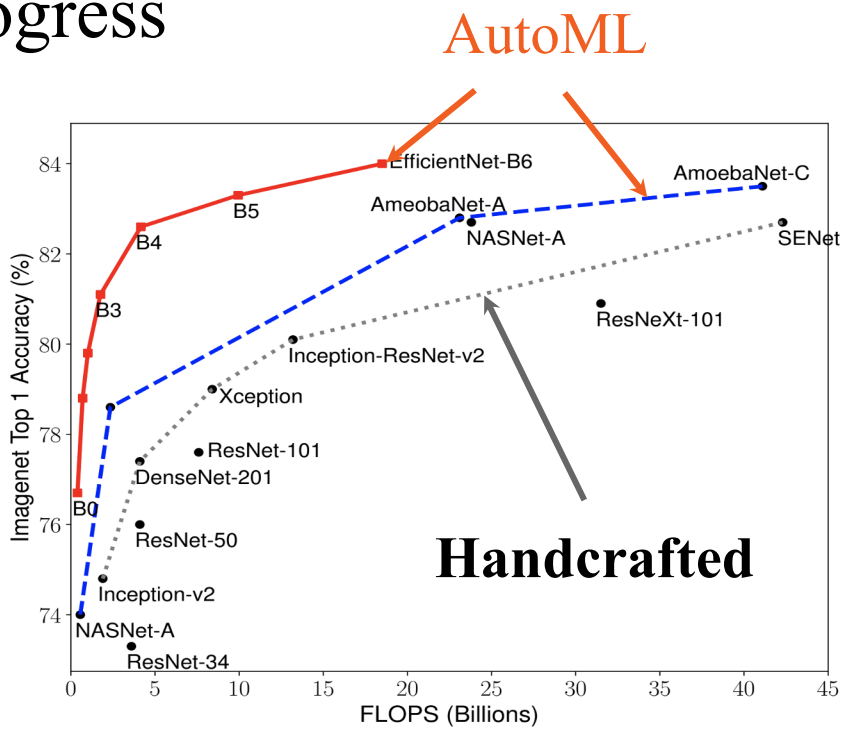


PyGlove: Symbolic Programming for AutoML

Daiyi Peng, Xuanyi Dong, Esteban Real, Mingxing Tan, Yifeng Lu,
Hanxiao Liu, Gabriel Bender, Adam Kraft, Chen Liang, Quoc V. Le

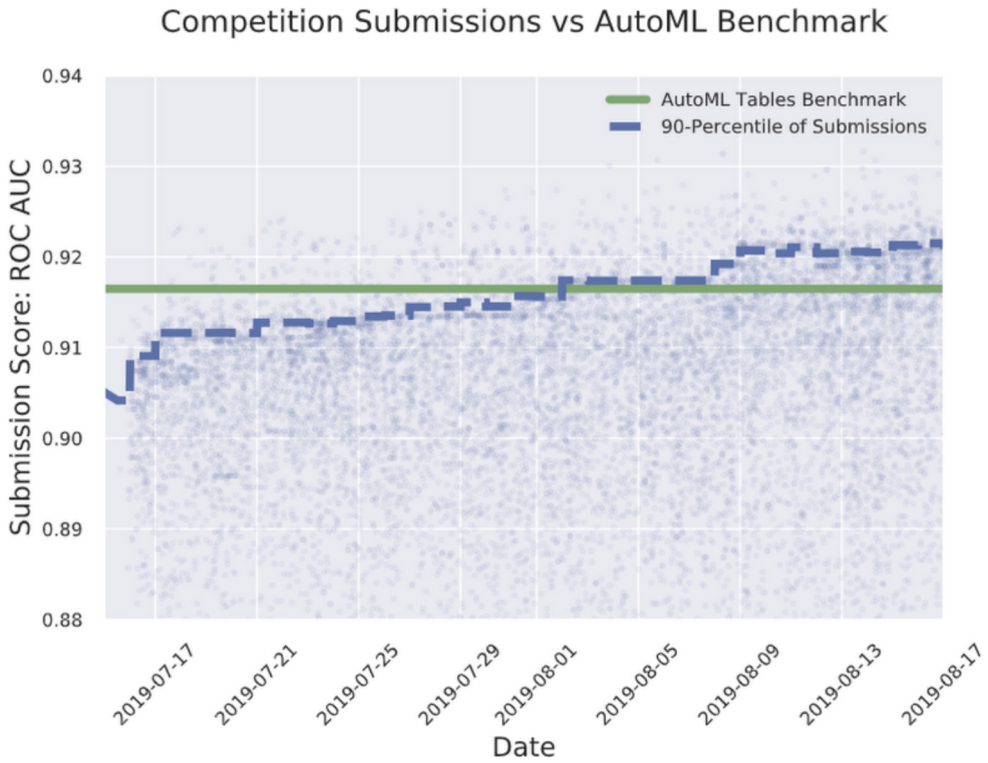
Google Research

AutoML: the progress



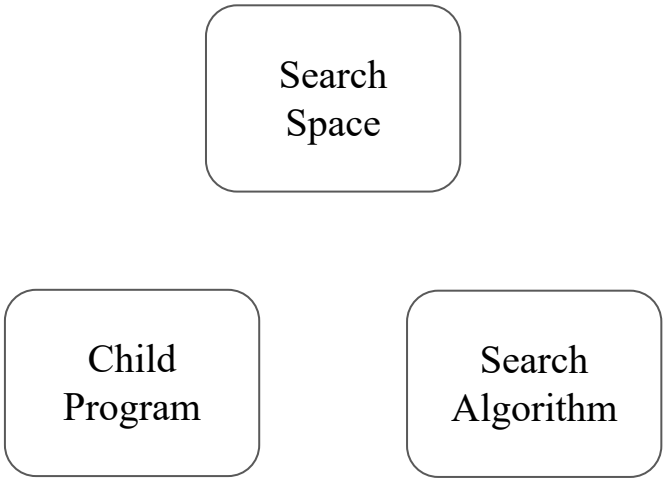
AutoML helps human experts in pushing the state-of-the-art results on ImageNet, with more accurate and more efficient model architectures (e.g., Tan & Le, ICML, 2019)

AutoML: the progress

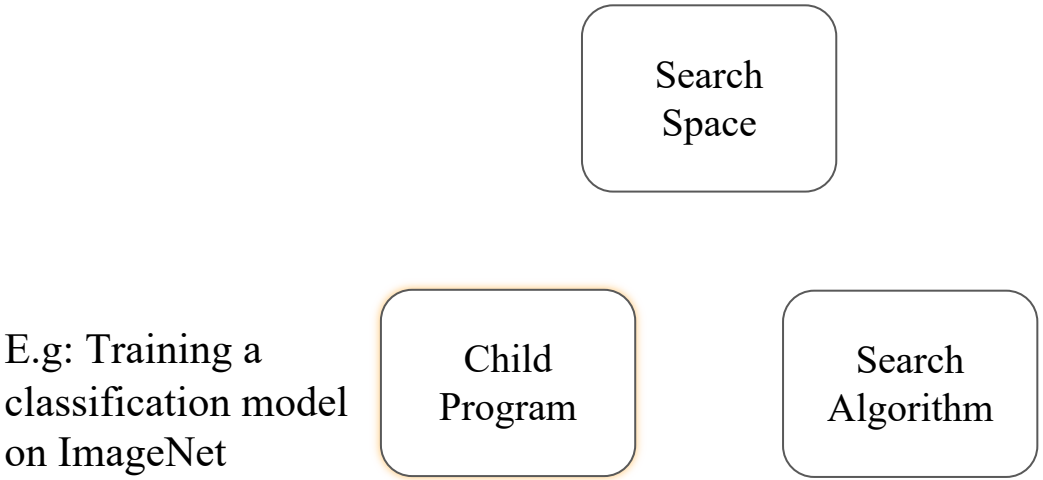


Google AutoML Tables beats >90% of daily submissions for approximately the first two weeks of the Kaggle Days competition. (Cloud Next ’19)

AutoML: the process



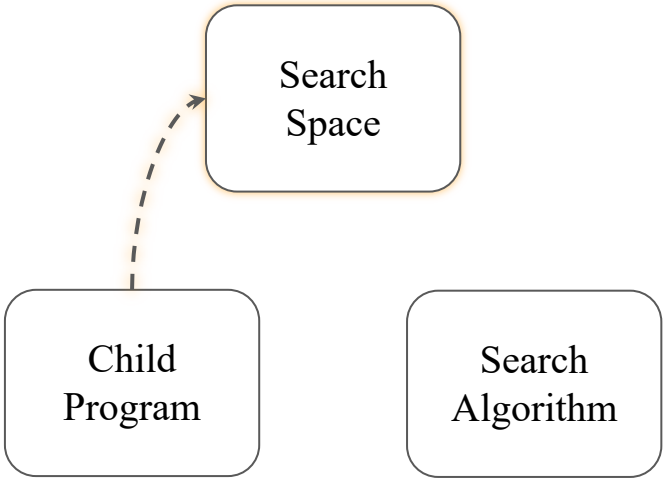
AutoML: the process



E.g: Training a classification model on ImageNet

AutoML: the process

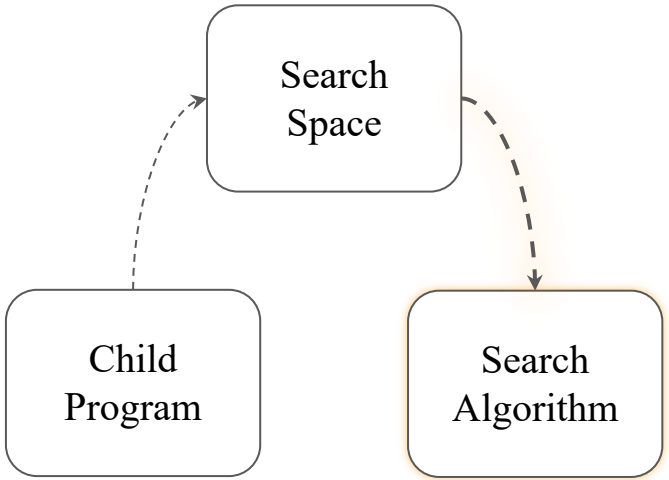
E.g: A list of operations to try in a residual block of the model



E.g: Training a classification model on ImageNet

AutoML: the process

E.g: A list of operations to try in a residual block of the model

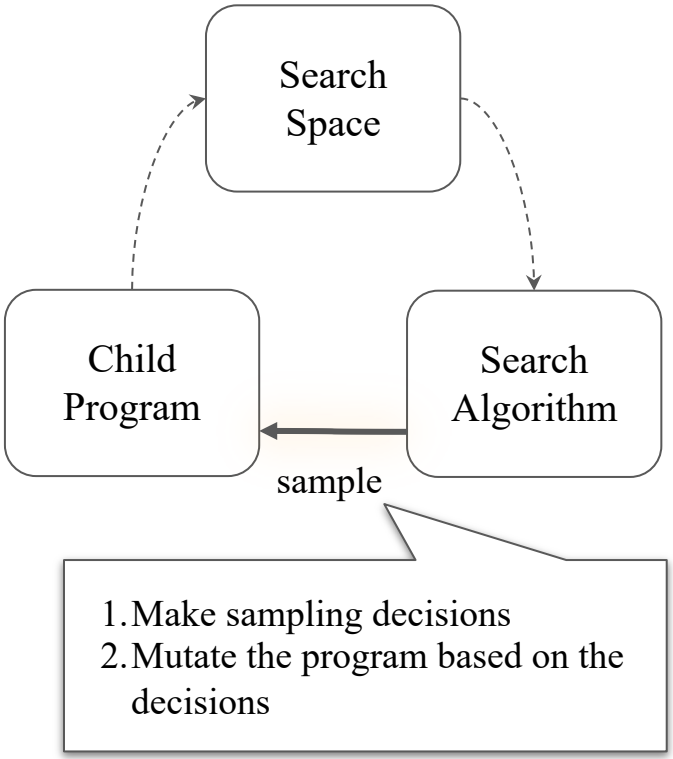


E.g: Training a classification model on ImageNet

E.g: A reinforcement learning algorithm

AutoML: the process

E.g: A list of operations to try in a residual block of the model

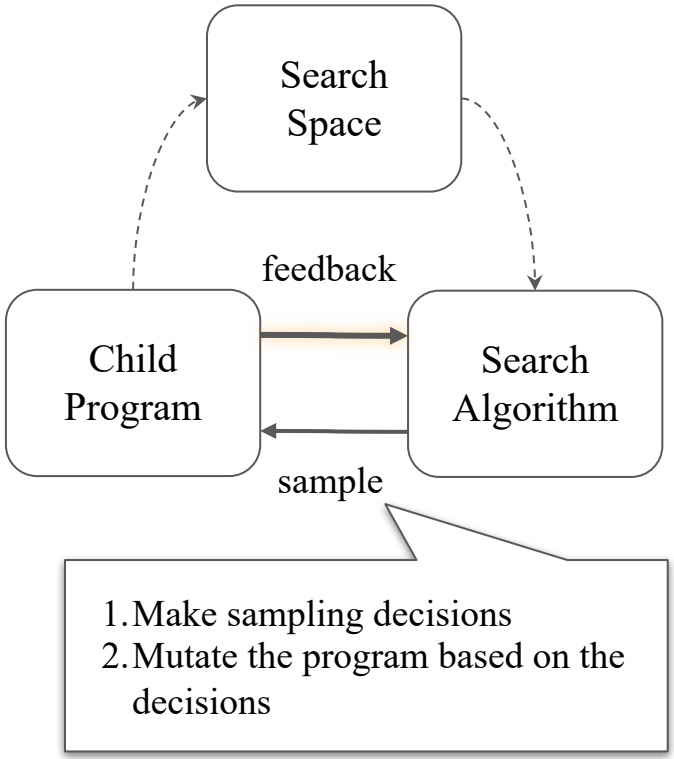


E.g: Training a classification model on ImageNet

E.g: A reinforcement learning algorithm

AutoML: the process

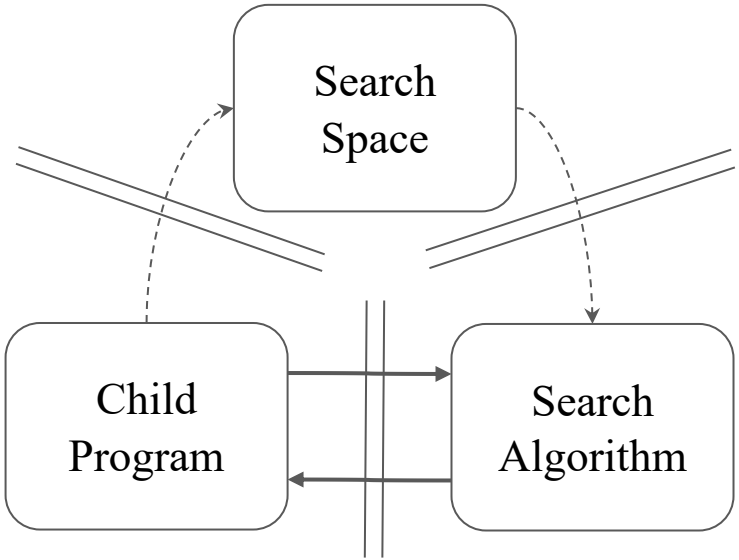
E.g: A list of operations to try in a residual block of the model



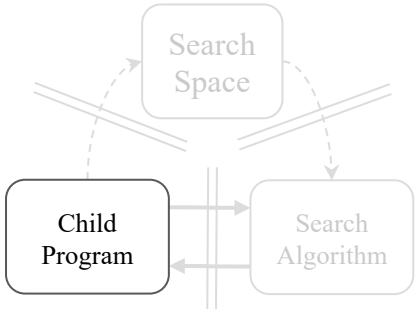
E.g: Training a classification model on ImageNet

E.g: A reinforcement learning algorithm

AutoML: programming challenges



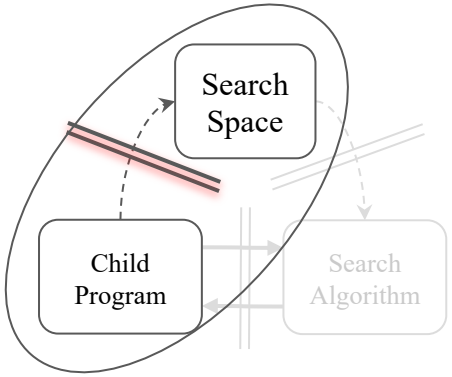
Example 1: coupling between child program and search space




```
class ResidualBlock:
  def call(self, inputs):
    op = Conv(...)
    return Add(
      inputs, op(inputs))
```

A class in the child program

Example 1: coupling between child program and search space



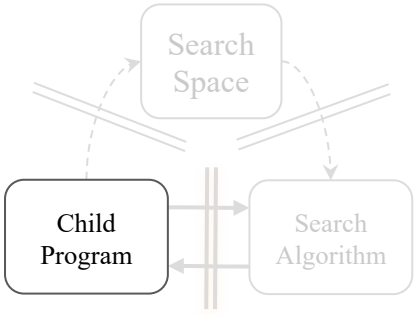

```
class ResidualBlock:
  def call(self, inputs):
    op = Conv(...)
    return Add(
      inputs, op(inputs))
```

A class in the child program

```
class SearchableResidualBlock:
  def call(self, inputs, hps):
    if hps.op_type == 'conv':
      op = Conv(...)
    elif hps.op_type == 'dense':
      op = Dense(...)
    elif ...
    return Add(inputs, op(inputs))
```

A class that couples with a search space

Example 2: coupling in Efficient NAS methods

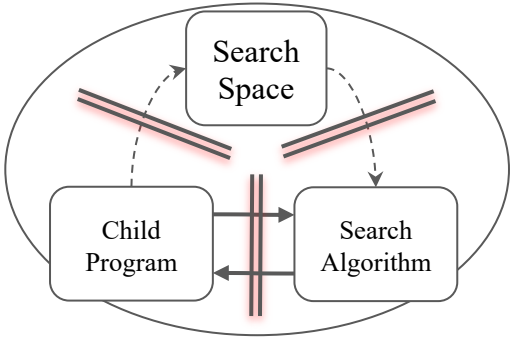



```
class MobileModel(object):

  def call(self, inputs):
    …
    layer = MBConv()
    ...
    return Sequential([layer, ...])
```

A class in the child program

Example 2: coupling in Efficient NAS methods




```
class MobileModel(object):

  def call(self, inputs):
    layer = MBConv()
    ...
    return Sequential([layer, ...])
```

A class in the child program

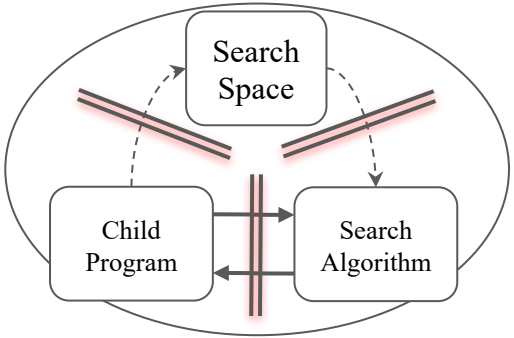
ENAS

```
class EnasMobileModel(object):

  def call(self, inputs, hps):
    ...
    layer = Switch([
      MBConv((3, 3), 3)
      MBConv((5, 5), 3)
      MBConv((7, 7), 3)
      ...
    ], selected=hps.op_choice0)
    ...
    return Sequential([layer, ...])
```

A class that couples with a search algorithm

Example 2: coupling in Efficient NAS methods




```
class MobileModel(object):

  def call(self, inputs):
    layer = MBConv()
    ...
    return Sequential(
        [layer, ...])
```

A class in the child program

DARTS

```
class DartsMobileModel(object):

  def call(self, inputs, hps):
    layer = WeightedSum([
      MBConv((3, 3), 3)
      MBConv((5, 5), 3)
      MBConv((7, 7), 3)
      ...
    ], weights=hps.op_weights0)
    ...
    return Sequential(
        [layer, ...])
```

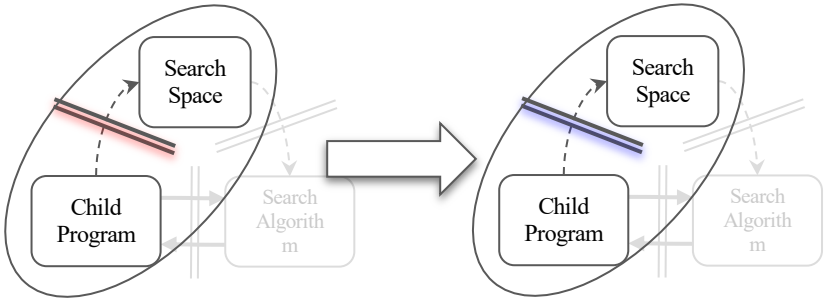
```
class EnasMobileModel(object):

  uts, hps):
    3)
    3)
    3)
  op_choice0)
  l(
  )
```

h a search algorithm

The fluidity of couplings

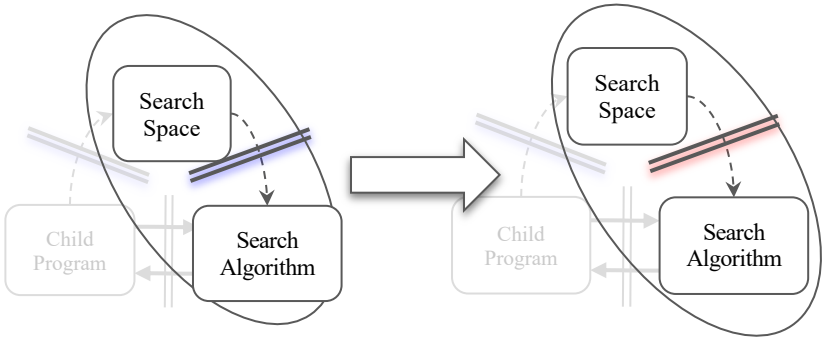
Change search space



Search space A with a blackbox search algorithm

Search space B with a blackbox search algorithm

Change search algorithm

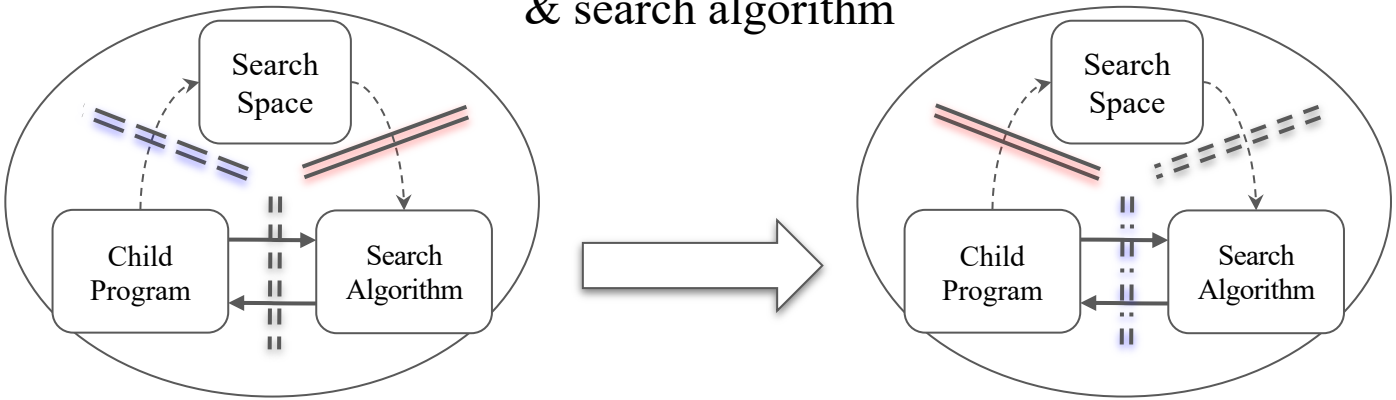


Blackbox search algorithm X on any search space

Blackbox search algorithm Y on any search space

The fluidity of couplings

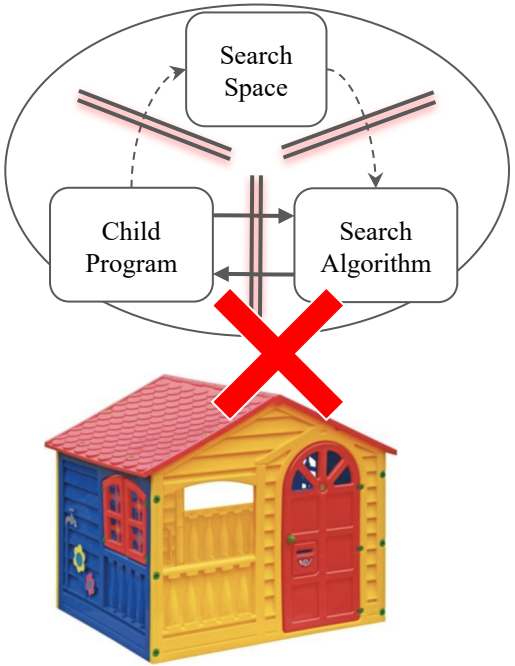
Change search space
& search algorithm



Efficient NAS algorithm Y
on search space B

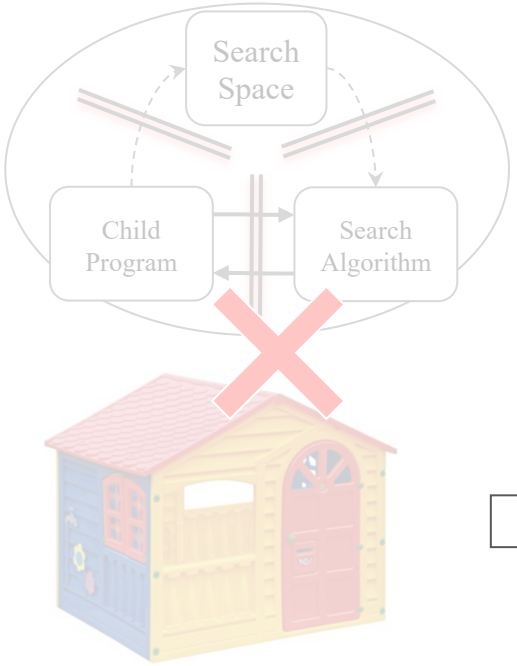
Efficient NAS algorithm X
on search space A

What if?

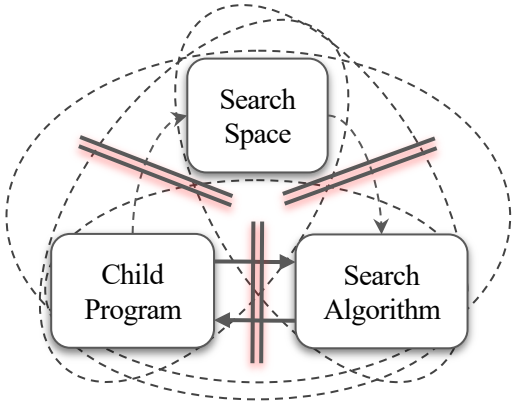


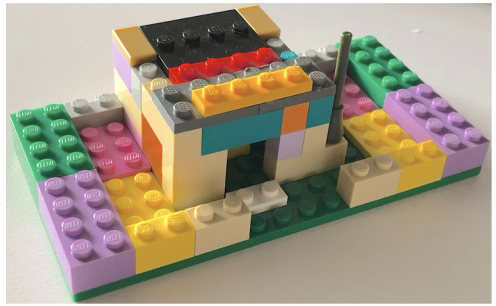
Fixed coupling

What if?



Fixed coupling





Dynamic coupling

LEGO image credit: Daniel Peng

Symbolic Programming for AutoML

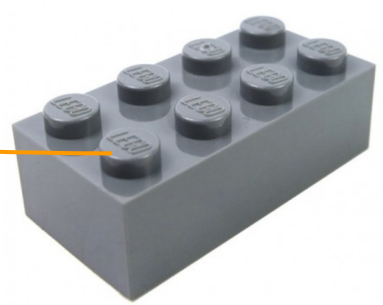
Symbolize: Make regular program symbolically programmable

Symbolize existing classes

```
Conv = symbolize(
    tf.keras.layers.Conv2D)
```

Hyper-parameters are like the studs of a LEGO brick

```
@symbolize
class Trainer(object):
  def __init__(
    self, model, optimizer):
    ...
  def train(self):
    return trainer_impl(
      Self.optimizer,
self.model)
```



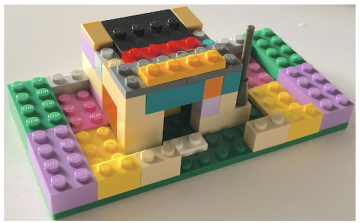
Symbolize new classes

Symbolizing classes

Symbolic objects are mutable

```
Trainer(
  model=Stacked(
    op=Conv(8, (3, 3)),
    repeats=2),
  optimizer=Adam(2e-2)
)
```

Program parts are not only compositional, but also ...

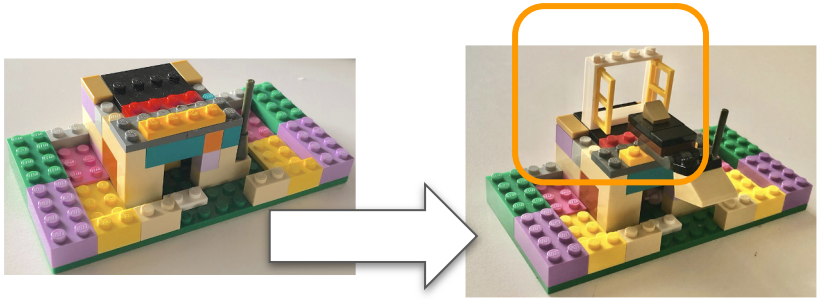


Symbolic objects are mutable

```
Trainer(
  model=Stacked(
    op=Conv(8, (3, 3)),
    repeats=2),
  optimizer=Adam(2e-2)
)
```

```
Trainer(
  model=Stacked(
    op=MaxPool((3, 3)),
    repeats=2),
  optimizer=Adam(2e-2)
)
```

Program parts are not only compositional, but also can be modified programmatically.



Programming interfaces are provided for symbolic manipulation

```
def swap(k, v, parent):
  if isinstance(v, Conv):
    return MaxPool(v.kernel)
  return v

trainer.clone().rebind(swap)
```

Clone trainer and replace all the Conv layers into MaxPools

From a static program to a search space

```
Trainer(
  model=Stacked(
    op=Conv(8, (3, 3)),
    repeats=3),
  optimizer=Adam(2e-2)
)
```

```
Trainer(
  model=Stacked(
    op=oneof([
      Identity(),
      MaxPool((3, 3)),
      Conv(oneof([4, 8]), (3, 3))]),
    repeats=3)),
  optimizer=
    oneof([
      Adam(2e-2),
      RMSProp(floatv(1e-6, 1e-3))])
)
```

A static child program

A search space

From a static program to a search space

```
trainer = Trainer(
  model=Stacked(
    op=Conv(8, (3, 3)),
    repeats=3),
  optimizer=Adam(2e-2)
)
```

A static child program

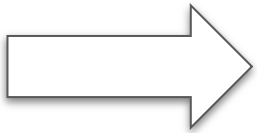
```
hyper_trainer = Trainer(
  model=Stacked(
    op=oneof([
      Identity(),
      MaxPool((3, 3)),
      Conv(oneof([4, 8]), (3, 3))]),
    repeats=3)),
  optimizer=
    oneof([
      Adam(2e-2),
      RMSProp(floatv(1e-6, 1e-3))])
)
```

A search space

From a static program to a search space

```
class SearchableStacked:
  def call(self, inputs, hps):
    if hps.op == 'identity':
       op = Identity()
    elif hps.op == 'max_pool':
       op = MaxPool(...)
    elif hp.op == 'Conv':
       op = Conv(...)
    return [op] * [illegible]repeats
```

A class that blends the child program and the search space



```
hyper_trainer = Trainer(
  model=Stacked(
    op=oneof([
      Identity(),
      MaxPool((3, 3)),
      Conv(oneof([4, 8]), (3, 3))]),
    repeats=3)),
  optimizer=
    oneof([
      Adam(2e-2),
      RMSProp(floatv(1e-6, 1e-3))])
)
```

Search space via composition

Search expressed as a for-loop

```
for trainer, feedback in sample(
    search_space=hyper_trainer,
    algorithm=PPO()):
  reward = trainer.train()
  feedback(reward)
```

Search as a feedback loop with sampled child programs

Search expressed as a for-loop

```
for trainer, feedback in sample(
    search_space=hyper_trainer,
    algorithm=PPO()):
  reward = trainer.train()
  feedback(reward)
```

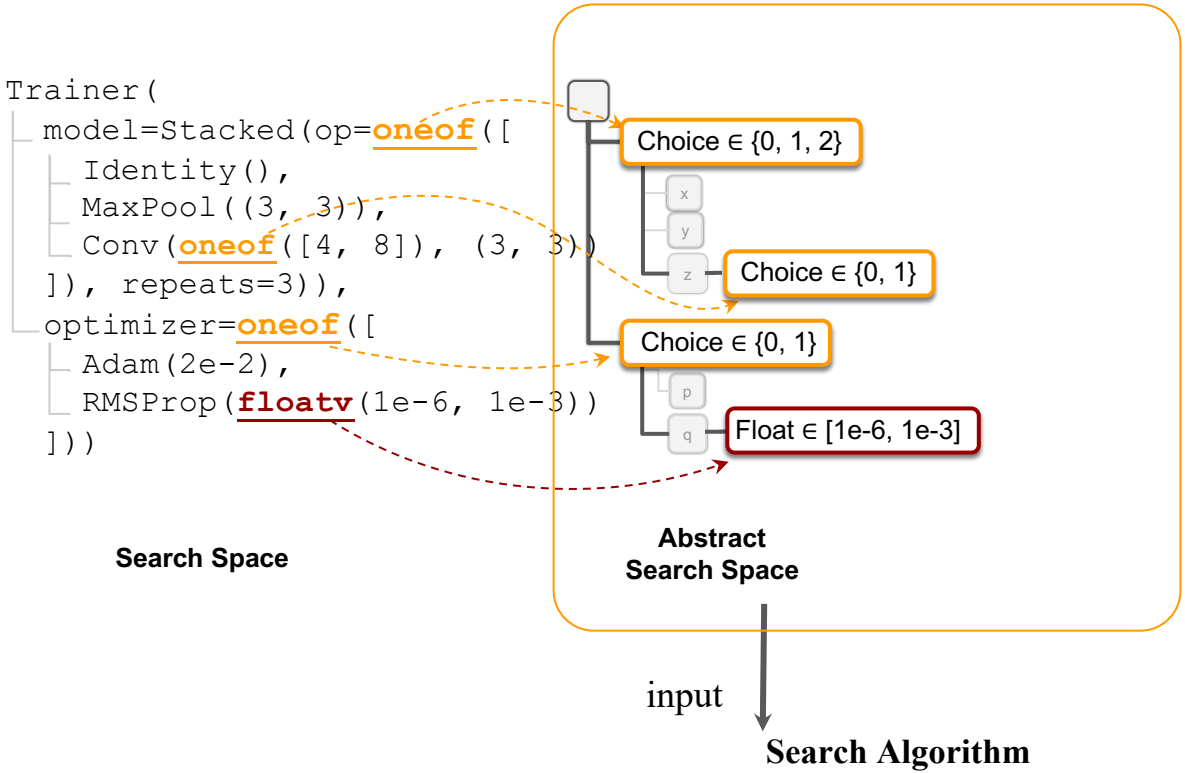
Search as a feedback loop with sampled child programs

How sample works (1)

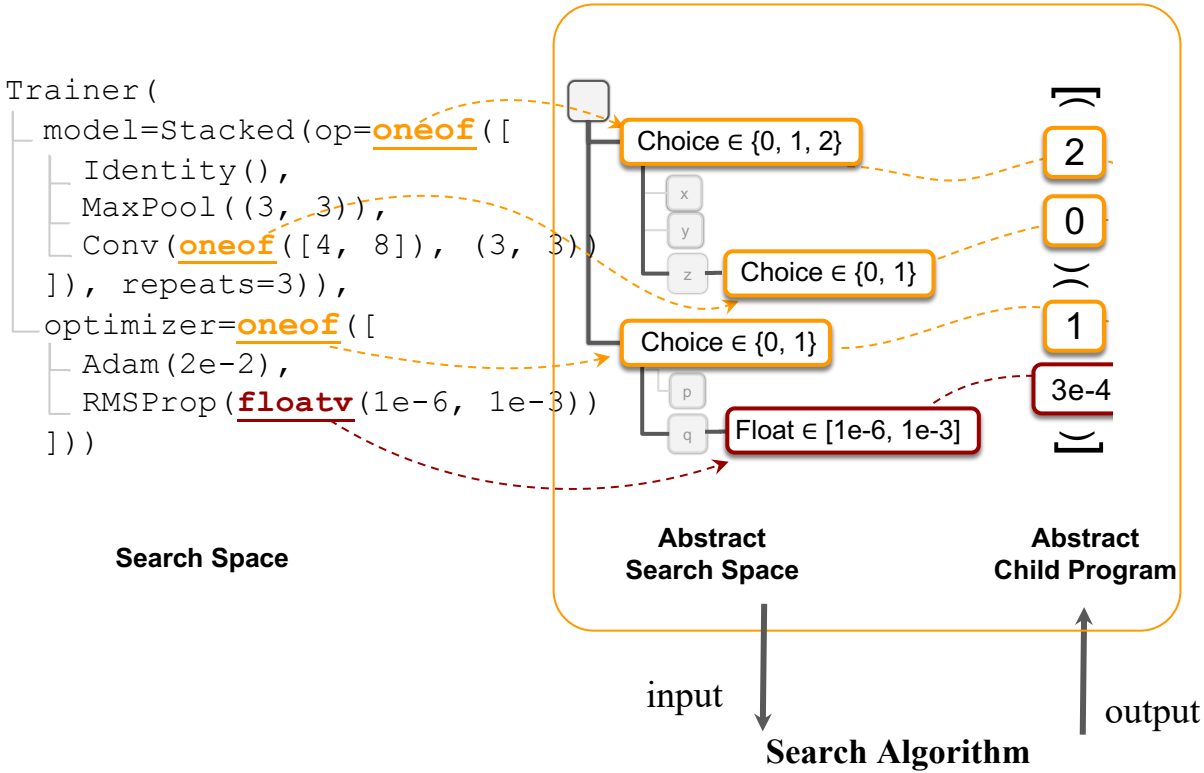
```
Trainer(
  model=Stacked(op=oneof([
    Identity(),
    MaxPool((3, 3)),
    Conv(oneof([4, 8]), (3, 3))
  ]), repeats=3)),
  optimizer=oneof([
    Adam(2e-2),
    RMSProp(floatv(1e-6, 1e-3))
  ]))
```

Search Space

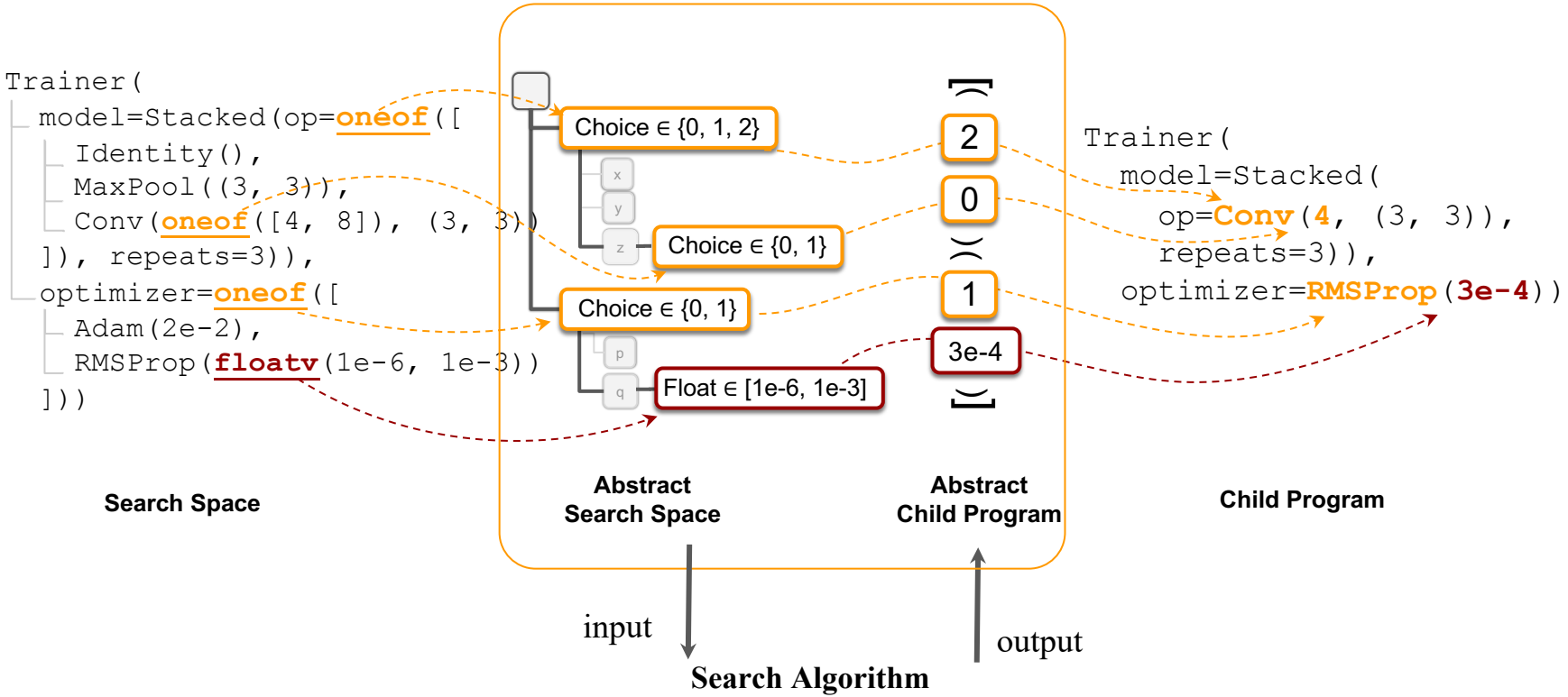
How sample works (2)



How sample works (3)



How sample works (4)



Supporting Efficient NAS using rewrite

```
hyper_trainer = Trainer(
  model=MobileNet(
    layers=[
       oneof([
         MBConv((3, 3), 3),
         MBConv((5, 5), 3),
         MBConv((7, 7), 3),
         ...
       ]),
       ...
    ]),
    ...
)
```

A regular search space

Supporting Efficient NAS using rewrite

```
hyper_trainer = Trainer(
  model=MobileNet(
    layers=[
      oneof([
        MBConv((3, 3), 3),
        MBConv((5, 5), 3),
        MBConv((7, 7), 3),
        ...
      ]),
      ...
    ]),
    ...
)
```

A regular search space

```
super_trainer = SuperNetworkTrainer(
  model=MobileNet(
    layers=[
      Switch([
        MBConv((3, 3), 3),
        MBConv((5, 5), 3),
        MBConv((7, 7), 3),
        ...
      ], index=algo.next_decision()),
      ...
    ]),
    ...
)
```

A super-program required by the efficient NAS algorithm

Supporting Efficient NAS using rewrite

```
class EnasMobileModel(object):

  def call(self, inputs, hps):
    …
    layer = Switch([
      MBConv((3, 3), 3)
      MBConv((5, 5), 3)
      MBConv((7, 7), 3)
      …
    ], selected=hps.op_choice0)
    …
    return Sequential([layer, ...])
```

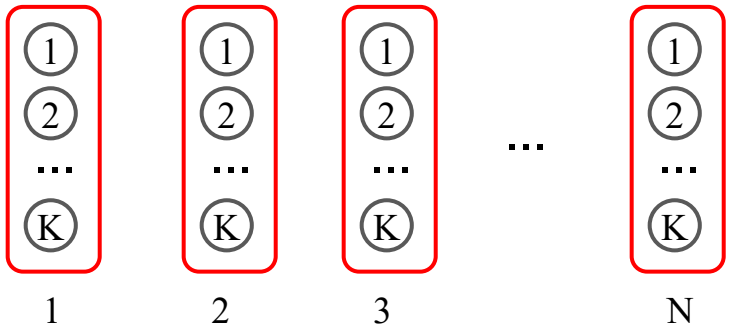
```
super_trainer = SuperNetworkTrainer(
  model=MobileNet(
    layers=[
      Switch([
        MBConv((3, 3), 3),
        MBConv((5, 5), 3),
        MBConv((7, 7), 3),
        ...
      ], index=algo.next_decision()),
      ...
    ]),
    ...
)
```

A class that blends the child program, search space and search algorithm together

A super-program required by the efficient NAS algorithm

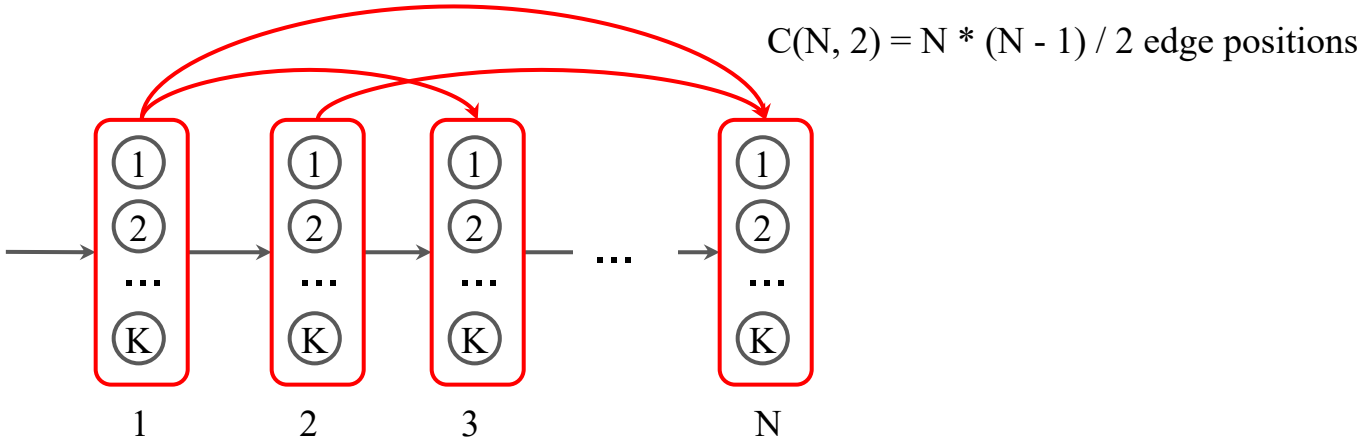
Case Study

Expressing complex search spaces (1)



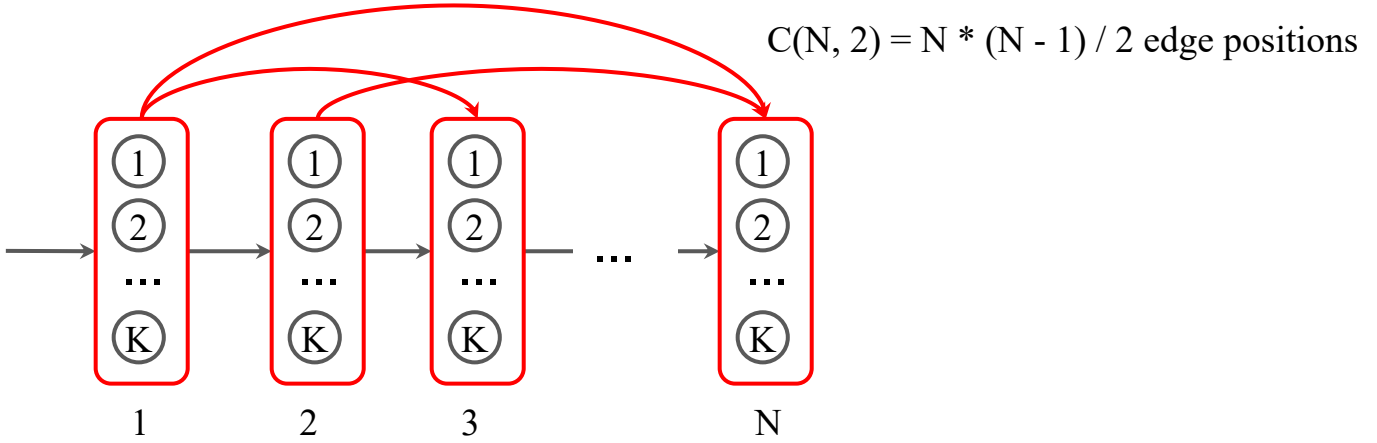
NAS-Bench-101

Expressing complex search spaces (1)



NAS-Bench-101

Expressing complex search spaces (1)

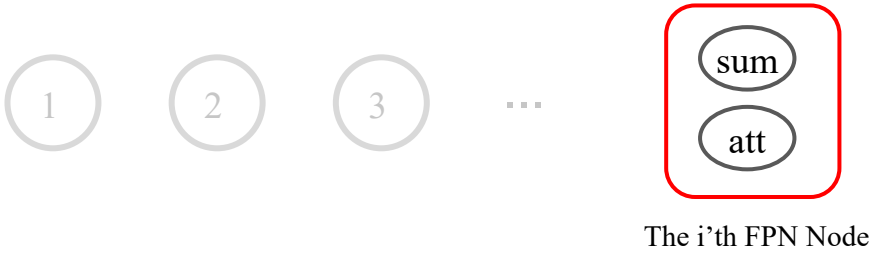



```
ModelSpec(
  nodes=[oneof(range(K))]*N,
  edges=[oneof([0, 1])]*N*(N-1)/2)
```

Independent sampling

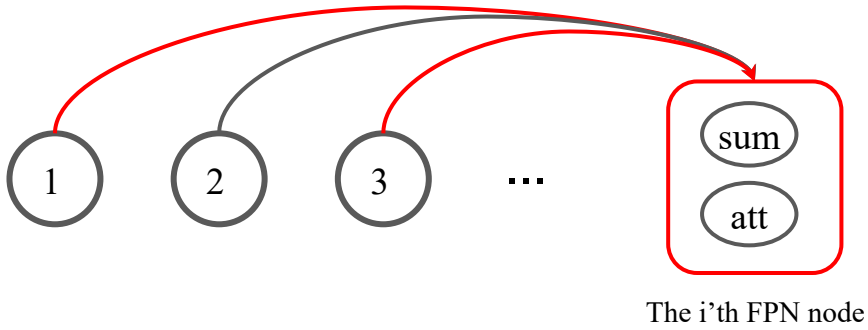
NAS-Bench-101

Expressing complex search spaces (2)



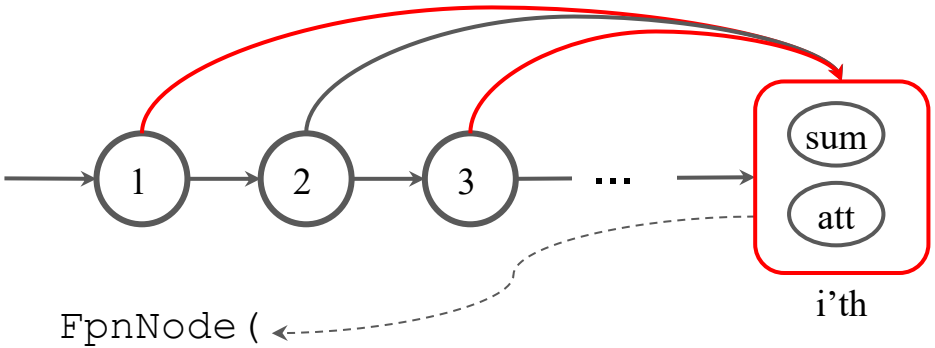
NAS-FPN: a FPN (Feature Pyramid) node

Expressing complex search spaces (2)



NAS-FPN: a FPN (Feature Pyramid) node

Expressing complex search spaces (2)

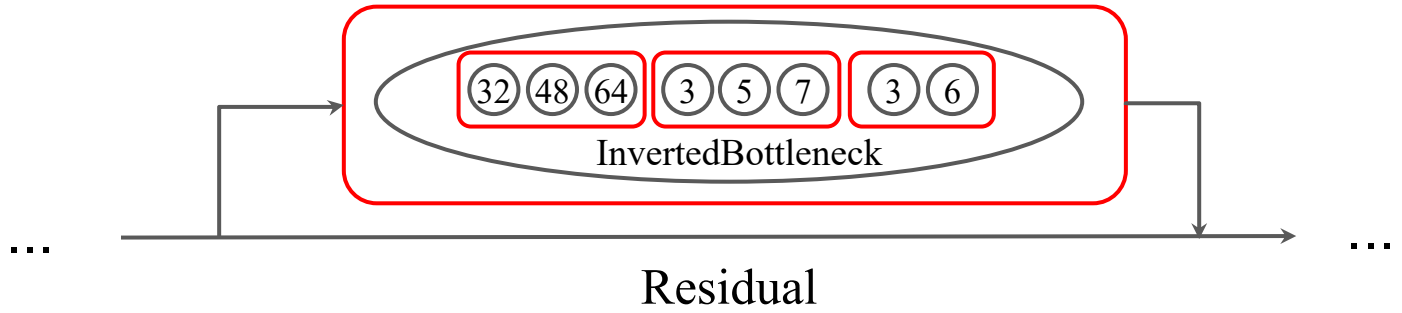



```
FpnNode(
  type=oneof(['sum', 'attention']),
  level=3,
  input_offsets=manyof(
      2, range(NUM_PRE_NODES),
      distinct=True,
      sorted=True))
```

Multiple choices with constraints.

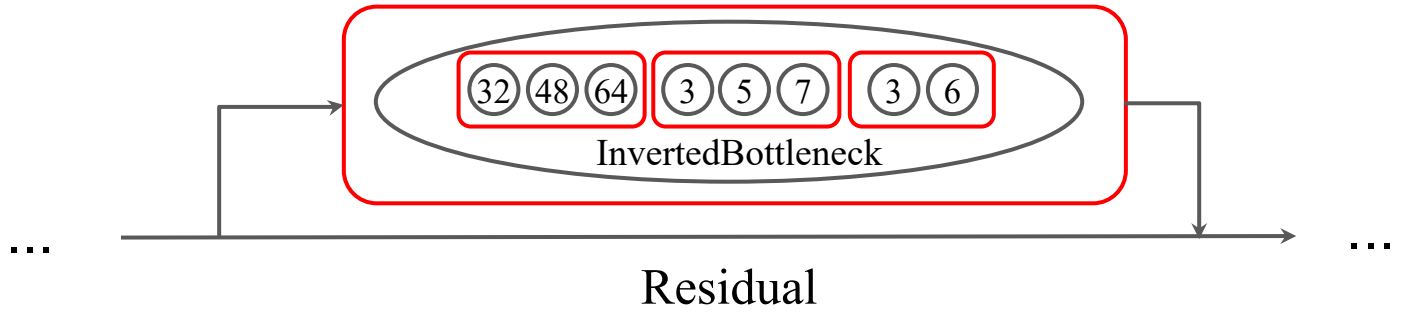
NAS-FPN: a FPN (Feature Pyramid) node

Expressing complex search spaces



TuNAS: A residual layer

Expressing complex search spaces

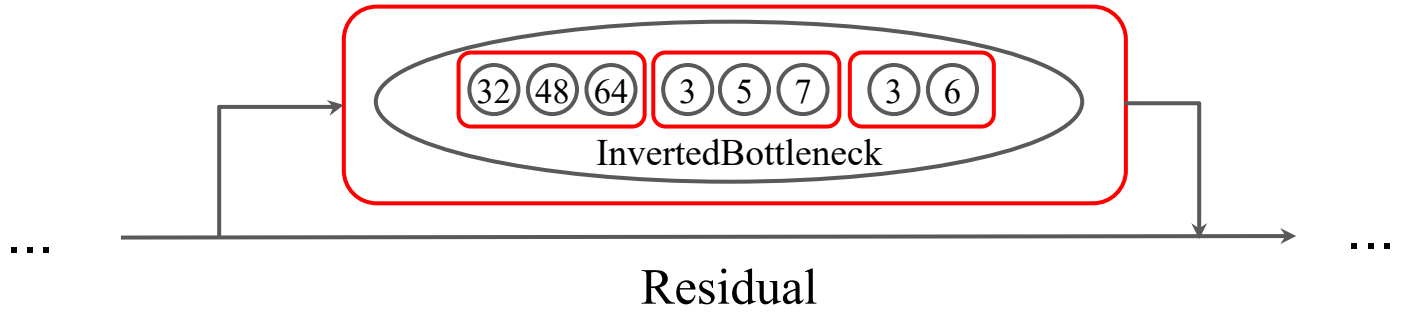



```
Residual(oneof([
  InvertedBottleneck(
    filters=oneof([32, 48, 64]),
    kernel=oneof([3, 5, 7]),
    expansion=oneof([3, 6])),
  Zero()]))
```

Active only when the InvertedBottleneck is selected

TuNAS: A residual layer

Expressing complex search spaces




```
Residual(oneof([
  InvertedBottleneck(
    filters=oneof([32, 48, 64]),
    kernel=oneof([3, 5, 7]),
    expansion=oneof([3, 6])),
  Zero()]))
```

Active only when the InvertedBottleneck is selected

Downgrades the Residual to Identity when selected

TuNAS: A residual layer

Create search space programmatically

```
def relax_filters(k, v, parent):
  if isinstance(parent, Conv):
    if k == 'filters':
      return oneof([v//2, v, v*2])
  return v

hyper_trainer = trainer.clone().rebind(relax_filters)
```

Creating an architectural search space by relaxing the filters of all Conv layers in current model into a set of options.

Create search space programmatically

```
def relax_filters(k, v, parent):
  if isinstance(parent, Conv):
    if k == 'filters':
      return oneof([v//2, v, v*2])
  return v

hyper_trainer = trainer.clone().rebind(relax_filters)
```

Creating an architectural search space by relaxing the filters of all Conv layers in current model into a set of options.

Switch between search algorithms

```
for trainer, feedback in sample(
    search_space=hyper_trainer,
    algorithm=RandomSearch()):
  reward = trainer.train()
  feedback(reward)
```

Switch between search algorithms

```
for trainer, feedback in sample(
    search_space=hyper_trainer,
    algorithm=RandomSearch()):
  reward = trainer.train()
  feedback(reward)
```

```
for trainer, feedback in sample(
    search_space=hyper_trainer,
    algorithm=Bayesian()):
  reward = trainer.train()
  feedback(reward)
```

RS → Bayesian

Switch between search algorithms

```
for trainer, feedback in sample(
    search_space=hyper_trainer,
    algorithm=RandomSearch()):
  reward = trainer.train()
  feedback(reward)
```

```
for trainer, feedback in sample(
    search_space=hyper_trainer,
    algorithm=Bayesian()):
  reward = trainer.train()
  feedback(reward)
```

RS → Bayesian

Bayesian → TuNAS

```
tunas.rewrite(hyper_trainer,Reinforce()).search()
```

Exploring 3 search spaces and 3 search algorithms

#	Search space	Search algorithm	**Lines of codes**	Search cost	Train cost	Test accuracy	# of MAdds
1	$(static)$	N/A	N/A	N/A	1	73.1	300M
2	$(static) \rightarrow \mathcal{S}_1$	RS	**+23**	25	1	73.7 ($\uparrow$ 0.6)	299M
3	$\mathcal{S}_1$	RS $\rightarrow$ Bayesian	**+1**	25	1	73.9 ($\uparrow$ 0.8)	305M
4	$\mathcal{S}_1$	Bayesian $\rightarrow$ TuNAS	**+1**	1	1	74.2 ($\uparrow$ 1.1)	301M
5	$(static) \rightarrow \mathcal{S}_2$	TuNAS	**+10**	1	1	73.3 ($\uparrow$ 0.2)	307M
6	$\mathcal{S}_1, \mathcal{S}_2 \rightarrow \mathcal{S}_3$	TuNAS	**+1**	2	1	73.8 ($\uparrow$ 0.7)	303M

It requires only a few lines of change to iterate on both search spaces and search algorithms

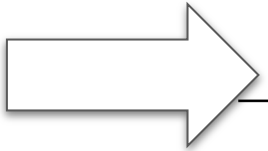
Expressing complex search flows

```
for trainer, feedback in sample(
    search_space=hyper_trainer,
    algorithm=PPO(),
    partition_fn=None):
  reward = trainer.train()
  feedback(reward)
```

Regular search flow

Expressing complex search flows

```
for trainer, feedback in sample(
    search_space=hyper_trainer,
    algorithm=PPO(),
    partition_fn=None):
  reward = trainer.train()
  feedback(reward)
```



Search type	for-loop pattern
Joint	$\texttt{for}(x, f_x) : \ldots$
Separate	$\texttt{for}(x_1, f_{x1}) : \ldots$ $\texttt{for}(x_2, f_{x2}) : \ldots$
Factorized	$\texttt{for}(x_1, f_{x1}) :$ $\quad \texttt{for}(x_2, f_{x2}) : \ldots$

Regular search flow

Complex search flow

Expressing complex search flows: An example

```
def factorized_search(search_space):
  for edge_space, ops_feedback in pg.sample(
      search_space, RegularizedEvolution(),
      trials=300, partition_fn=lambda v: v.hints == OP_HINT):
    rewards = []
    for example, edges_feedback in pg.sample(
        edge_space, RegularizedEvolution(), trials=20):
      reward = nasbench.get_reward(example)
      edges_feedback(reward)
      rewards.append(reward)
    ops_feedback(top5_average(rewards))
```

Outer loop search for nodes, whose examples are sub-spaces for edges

Inner loop search for edges on fixed nodes

Factorized	for (x_1, f_{x1}) : for (x_2, f_{x2}) : ...

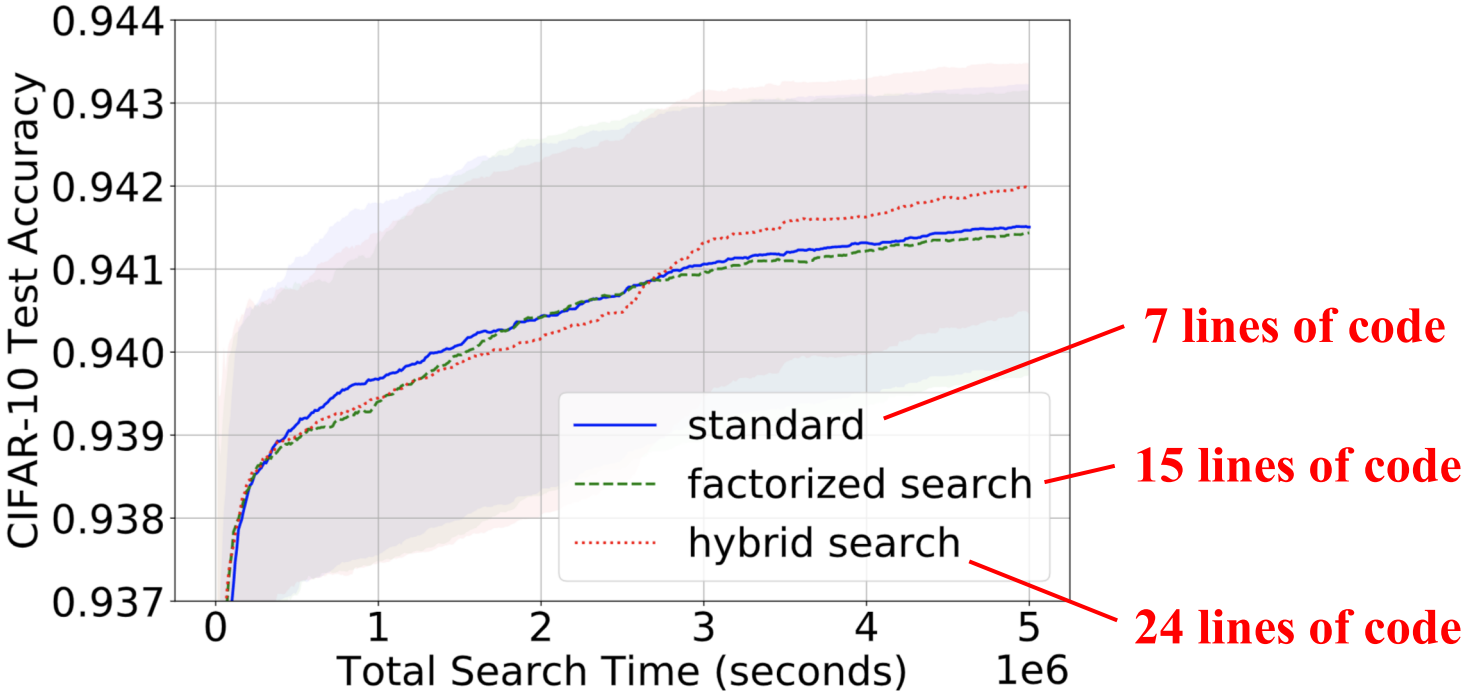
Expressing complex search flows: An example

```
def factorized_search(search_space):
  for edge_space, ops_feedback in pg.sample(
      search_space, RegularizedEvolution(),
      trials=300, partition_fn=lambda v: v.hints == OP_HINT):
    rewards = []
    for example, edges_feedback in pg.sample(
        edge_space, RegularizedEvolution(), trials=20):
      reward = nasbench.get_reward(example)
      edges_feedback(reward)
      rewards.append(reward)
    ops_feedback(top5_average(rewards))
```

Allows the search algorithm of the outer loop to see only the node sub-space

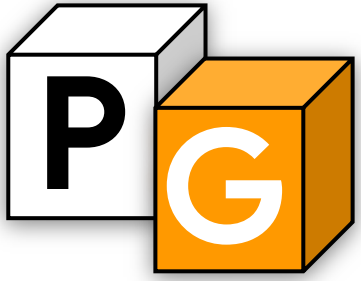
Factorized	for (x_1, f_{x1}) : for (x_2, f_{x2}) : ...

Exploring 3 search flows on NAS-Bench-101

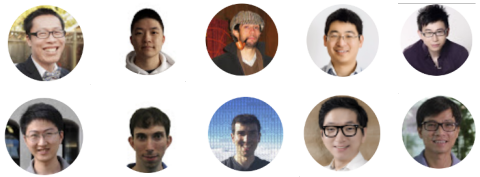


Search performances (mean) and their standard deviations with 500 runs.

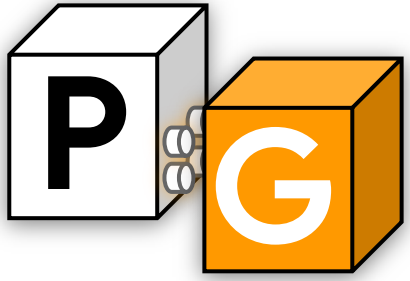
AutoML needs Symbolic Programming



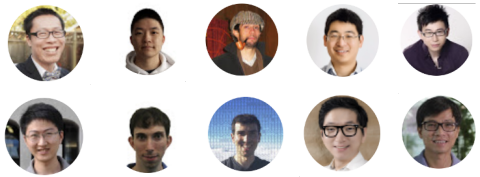
Thank you!



Google Research



Thank you!



Google Research