Resource-Aware Deep Learning Model Exploration on GPUs

Tong Shu

DL-GPU Workshop 2023
Day 2

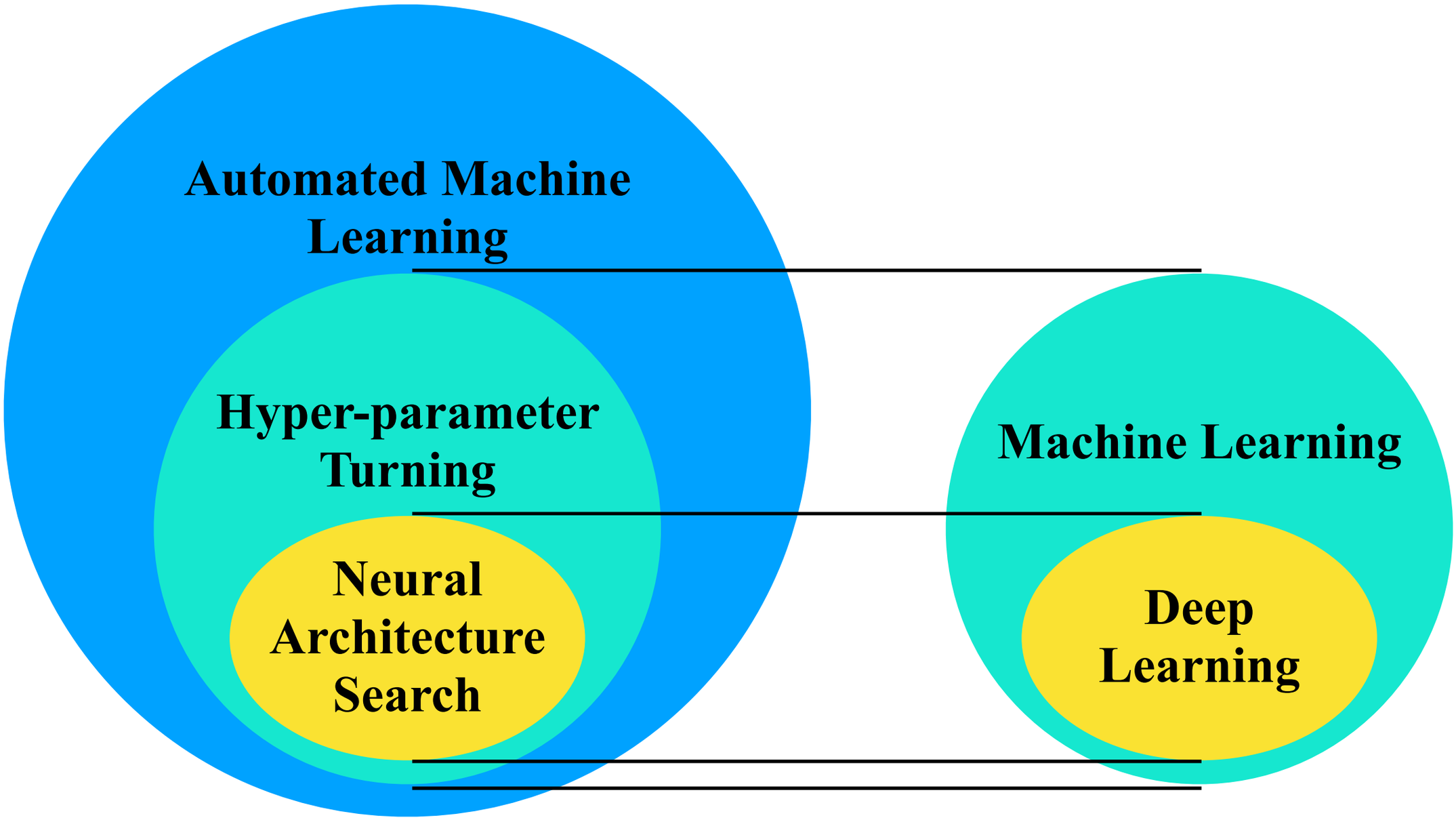
07/25/2023

Outline

- Introduction to neural architecture search (NAS)
- Hands-on exercises on NAS in Neural Network Intelligence
- Introduction to resource-aware neural architecture search
- Latency prediction of deep learning (DL) inference

Neural Architecture Search

AutoML => HPT => NAS



AutoML => HPT => NAS (Cont.)

- Automated machine learning (AutoML)
  - The process of automating the tasks of applying machine learning to real-world problems
  - Includes every stage from beginning with a raw dataset to building a machine learning model ready for deployment

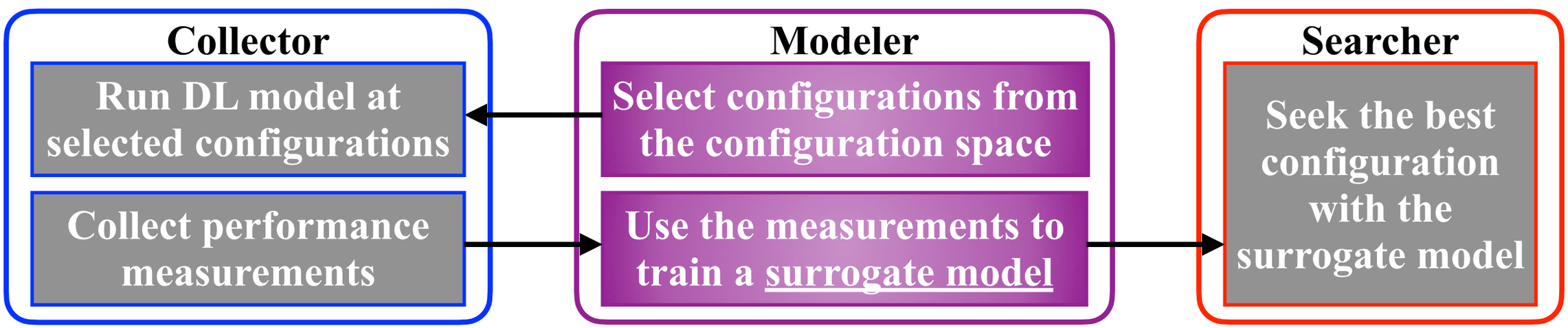
- Hyperparameter tuning (HPT) is the problem of choosing a set of optimal hyperparameters for a learning algorithm.
  - Hyperparameter: a parameter whose value is used to control the learning process

- Neural architecture search (NAS) aims to design a neural architecture that achieves the best possible performance using limited computing resources in an automated way with minimal human intervention.

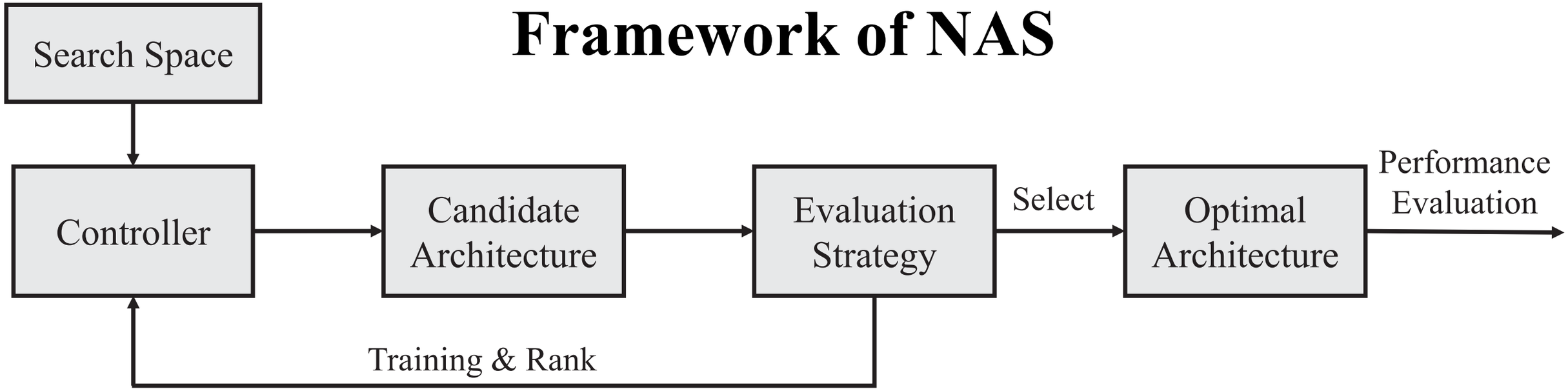
Hyperparameter Tuning

- Model-free auto-tuners
  - Evolutionary algorithms: e.g., genetic algorithm

- Empirical model-based auto-tuners
  - Configuration space
  - Auto-tuning algorithms
  - Performance measurement



Framework of NAS



- NAS begins with a set of predefined operation sets and uses a controller to obtain a large number of candidate neural architectures based on the search space created by these operation sets.
- The candidate architectures are then trained on the training set and ranked according to their accuracy on the validation set.
- The ranking information of the candidate neural architecture is then used as feedback information to adjust the search strategy, enabling a set of new candidate neural architectures to be obtained.
- When the termination condition is reached, the search process is terminated to select the best neural structure.
- The chosen neural architecture then conducts performance evaluation on the test set.

NAS Optimization 1

- Global search space
  - Searches all necessary components of the neural architecture
  - Disadvantage: extremely large search space

- Modular search space
  - Treats the neural architecture as a stack of multiple different types of modules.
  - Advantage: the search task is simplified to only one or more modules of different types.

NAS Optimization 2

- Discrete search strategy
  - Discretely modifying an operation to change the neural architecture
  - Disadvantage: cannot use the gradient strategy to quickly adjust the neural architecture

- Continuous search strategy.
  - Continuously relaxes the structural parameters of the neural architecture.
  - Advantage: gradient-optimized

NAS Optimization 3

- Search from scratch
  - The model is built from scratch until the final neural architecture is generated.
  - Disadvantage: ignore the existing neural architecture design experience

- Neural architecture recycling
  - Takes the existing artificially designed high-performance neural architecture as a starting point and uses the NAS method to perform network transformations on them to improve their performance.
  - Advantage: utilize the existing excellent neural architecture

NAS Optimization 4

- Fully trained
  - Train each candidate neural architecture from scratch until convergence.
  - Disadvantage: if the network structures of the subsequent and previous neural architectures are similar, this relationship cannot be fully utilized.

- Incomplete training
  - Use the shared structure between candidate architectures or performance prediction
  - Advantage: speed up the relative performance ranking of candidate architectures

State-of-the-Art NAS Systems

- NNI (Neural Network Intelligence)
  - https://nni.readthedocs.io/en/stable/
  - Retiarii: A Deep Learning Exploratory-Training Framework [OSDI'20]
  - https://www.youtube.com/watch?v=SiGgAkt8YYg

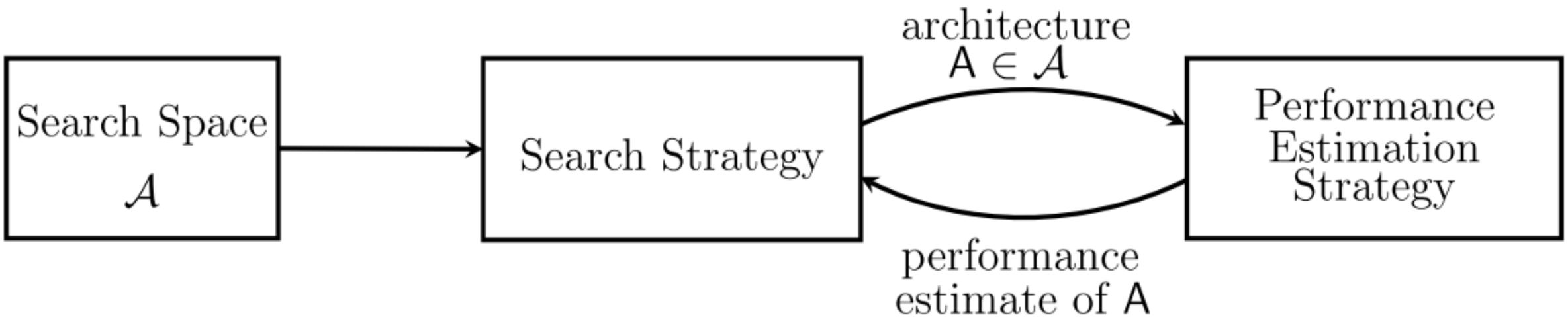
- DeepHyper
  - https://deephyper.readthedocs.io/en/latest/
  - Scalable Reinforcement-Learning-Based Neural Architecture Search for Cancer Deep Learning Research [SC'19]
  - AgEBO-Tabular: Joint Neural Architecture and Hyperparameter Search with Autotuned Data-Parallel Training for Tabular Data [SC'20]
  - Recurrent Neural Network Architecture Search for Geophysical Emulation [SC'21]
  - https://www.youtube.com/watch?v=XOywdjPMZq8
    - From 15 minutes to 55 minutes

NAS in Neural Network Intelligence

NNI (Neural Network Intelligence)

- NNI is a lightweight but powerful toolkit to help users automate
  - Hyperparameter Optimization
  - **<u>Neural Architecture Search</u>**
  - Model Compression
  - Feature Engineering

NAS Components of NNI



- Model search space (or search space, or model space)
  - A set of models from which the best model is explored/searched
- Exploration strategy (or search strategy)
  - The algorithm that is used to explore a model search space
- Model evaluator
  - Responsible for training a model and evaluating its performance

Construct Model Space

- NNI provides multi-level APIs for users to express model space.
  - Mutation Primitives: high-level APIs that are utilities to build blocks in search space (straightforward yet expressive enough).
  - Hyper-module Library: plug-and-play modules (e.g., the optimal activation function in AutoActivation)
  - Mutator: interface to customize new mutators for expressing more complicated model spaces.

APIs for Constructing Search Space

Name	Category	Brief Description
LayerChoice	Mutation Primitives	Select from some PyTorch modules
InputChoice	Mutation Primitives	Select from some inputs (tensors)
ValueChoice	Mutation Primitives	Select from some candidate values
Repeat	Mutation Primitives	Repeat a block by a variable number of times
Cell	Mutation Primitives	Cell structure popularly used in literature
NasBench101Cell	Mutation Primitives	Cell structure (variant) proposed by NAS-Bench-101
NasBench201Cell	Mutation Primitives	Cell structure (variant) proposed by NAS-Bench-201
AutoActivation	Hyper-modules	Library Searching for activation functions
Mutator	Mutator	Flexible mutations on graphs

Define a Model Space

- Model space
  - Defined by users to express a set of models that users want to explore
  - Contains potentially good-performing models.

- In NNI, a model space is defined with
  - A base model
  - Possible mutations on the base model

Define Base Model

- Defining a base model is almost the same as defining a PyTorch model.
- Replace the code import torch.nn as nn with import nni.retiarii.nn.pytorch as nn to use the wrapped PyTorch modules.

```
import torch
import torch.nn.functional as F
import nni.retiarii.nn.pytorch as nn
from nni.retiarii import model_wrapper

@model_wrapper
# this decorator should be put on the out most
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(self.conv2(x), 2)
        x = torch.flatten(self.dropout1(x), 1)
        x = self.fc2(self.dropout2(F.relu(self.fc1(x))))
        output = F.log_softmax(x, dim=1)
        return output
```

Define Model Mutations

- A base model is only one concrete model, but not a model space that includes many models.

- API and Primitives are provided for users to express how the base model can be mutated.

```
class DepthwiseSeparableConv(nn.Module):
    def __init__(self, in_ch, out_ch):
        super().__init__()
        self.depthwise = nn.Conv2d(in_ch, in_ch,
kernel_size=3, groups=in_ch)
        self.pointwise = nn.Conv2d(in_ch, out_ch,
kernel_size=1)

    def forward(self, x):
        return self.pointwise(self.depthwise(x))
```

```
@model_wrapper
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
-       self.conv2 = nn.Conv2d(32, 64, 3, 1)
+       self.conv2 = nn.LayerChoice([
+           nn.Conv2d(32, 64, 3, 1),
+           DepthwiseSeparableConv(32, 64)
+       ])
-       self.dropout1 = nn.Dropout(0.25)
+       self.dropout1 =
nn.Dropout(nn.ValueChoice([0.25, 0.5, 0.75]))
        self.dropout2 = nn.Dropout(0.5)
-       self.fc1 = nn.Linear(9216, 128)
-       self.fc2 = nn.Linear(128, 10)
+       feature = nn.ValueChoice([64, 128, 256])
+       self.fc1 = nn.Linear(9216, feature)
+       self.fc2 = nn.Linear(feature, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(self.conv2(x), 2)
        x = torch.flatten(self.dropout1(x), 1)
        x = self.fc2(self.dropout2(F.relu(self.fc1(x))))
        output = F.log_softmax(x, dim=1)
        return output
```

```
@model_wrapper
class ModelSpace(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        # LayerChoice is used to select a layer between Conv2d and DwConv.
        self.conv2 = nn.LayerChoice([
            nn.Conv2d(32, 64, 3, 1),
            DepthwiseSeparableConv(32, 64)
        ])
        # ValueChoice is used to select a dropout rate.
        # ValueChoice can be used as parameter of modules wrapped in "nni.retiarii.nn.pytorch"
        # or customized modules wrapped with "@basic_unit".
        # choose dropout rate from 0.25, 0.5 and 0.75
        self.dropout1 = nn.Dropout(nn.ValueChoice([0.25, 0.5, 0.75]))
        self.dropout2 = nn.Dropout(0.5)
        feature = nn.ValueChoice([64, 128, 256])
        self.fc1 = nn.Linear(9216, feature)
        self.fc2 = nn.Linear(feature, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(self.conv2(x), 2)
        x = torch.flatten(self.dropout1(x), 1)
        x = self.fc2(self.dropout2(F.relu(self.fc1(x))))
        output = F.log_softmax(x, dim=1)
        return output
```

```
model_space = ModelSpace()
model_space
```

Exploration Strategy

- Mutli-trial strategy: trains each sampled model in the model space independently.
  - NASNet is a typical multi-trial NAS.
  - Need a model evaluator to evaluate the performance of each sampled model.
  - Need an exploration strategy to sample models from defined model space.
    - To use an exploration strategy, users simply instantiate an exploration strategy and pass the instantiated object to RetiariiExperiment.

```
import nni.retiarii.strategy as strategy
exploration_strategy = strategy.Random(dedup=True) # dedup=False if deduplication is not wanted
```

- One-shot strategy samples the model from a supermodel.
  - One-shot NAS algorithms leverage weight sharing among models in NAS space to train a supernet, and use this supernet to guide the selection of better models.
  - Greatly reduces computational resource

Built-in Exploration Strategies in NNI

Name	Category	Brief Description
Random	Multi-trial	Randomly sample an architecture each time
GridSearch	Multi-trial	Traverse the search space and try all possibilities
RegularizedEvolut	Multi-trial	Evolution algorithm for NAS
TPE	Multi-trial	Tree-structured Parzen Estimator (TPE)
PolicyBasedRL	Multi-trial	Policy-based reinforcement learning, based on implementation of tianshou
DARTS	One-shot	Continuous relaxation of the architecture representation, allowing efficient search of the architecture using gradient descent
ENAS	One-shot	One-shot RL controller learns to generate the best network on a super-net
GumbelDARTS	One-shot	Choose the best block by using Gumbel Softmax random sampling and differentiable training
RandomOneShot	One-shot	Train a super-net with uniform path sampling
Proxyless	One-shot	A low-memory-consuming optimized version of differentiable architecture search

Performance Estimation

- In NNI, the standard performance estimation process is implemented with *evaluator*
  - *evaluator* is responsible for estimating a model's performance.

- NNI has quite a few built-in supports of evaluators, ranging
  - from the simplest option, e.g., to perform a standard training and validation of the architecture on data,
  - to complex configurations and implementations.

Customize Evaluator with Any Function

- When training code is available, customizing a new evaluator with FunctionalEvaluator only needs to write a fit function that wraps training, validating, and testing of a single model.
- This function takes one positional arguments (model_cls) and possible keyword arguments.
- The keyword arguments (other than model_cls) are fed to FunctionalEvaluator as its initialization parameters

```
from nni.retiarii.evaluator import FunctionalEvaluator
from nni.retiarii.experiment.pytorch import RetiariiExperiment

def fit(model_cls, dataloader):
    model = model_cls()
    train(model, dataloader)
    acc = test(model, dataloader)
    nni.report_final_result(acc)    # report the final score of this model

# The dataloader will be serialized, thus “nni.trace” is needed here.
# See serialization tutorial for more details.
evaluator = FunctionalEvaluator(fit, dataloader=nni.trace(DataLoader)(foo, bar))
experiment = RetiariiExperiment(base_model, evaluator, mutators, strategy)
```

Use Evaluators to Train and Evaluate Models

- Use evaluators to train/evaluate a single, concrete architecture when
  - Debugging your evaluator against a baseline model.
  - Fully train, validate, and test your model after the search process is complete.

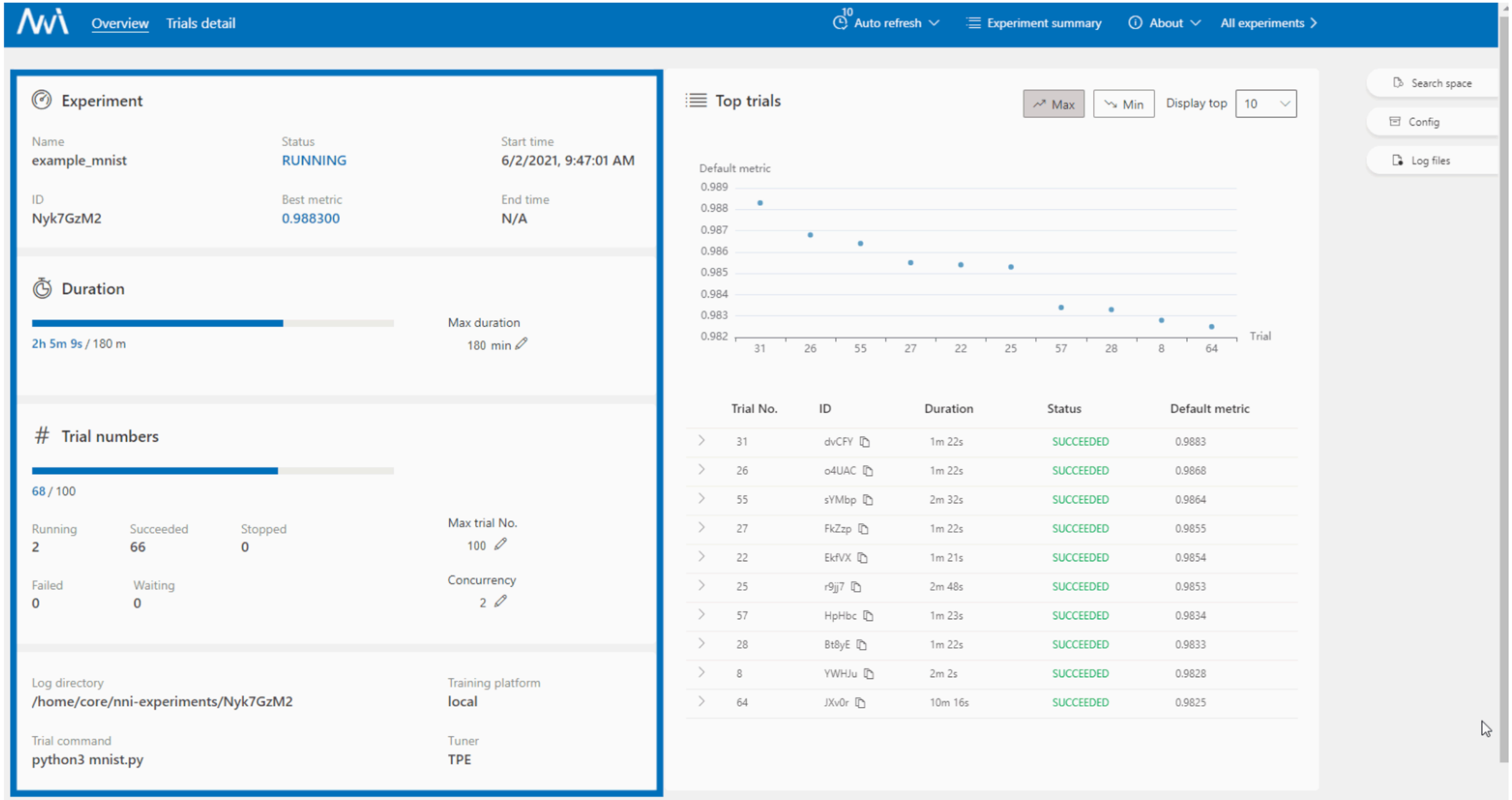
```
# Class definition of single model, for example, ResNet.
class SingleModel(nn.Module):
    def __init__():  # Can't have init parameters here.
        ...

# Use a callable returning a model
evaluator.evaluate(SingleModel)
# Or initialize the model beforehand
evaluator.evaluate(SingleModel())
```

Launch an Experiment

```
from nni.retiarii.experiment.pytorch import RetiariiExperiment, RetiariiExeConfig
exp = RetiariiExperiment(model_space, evaluator, [], search_strategy)
exp_config = RetiariiExeConfig('local')
exp_config.experiment_name = ‘mnist_search’

exp_config.max_trial_number = 4    # spawn 4 trials at most
exp_config.trial_concurrency = 2    # will run two trials concurrently


exp_config.trial_gpu_number = 1    # 1 for using GPU and 0 for not using GPU
exp_config.training_service.use_active_gpu = True
# set true if you wish to use an occupied GPU (possibly running a GUI).

exp.run(exp_config, 8081)
```

NNI Web Portal

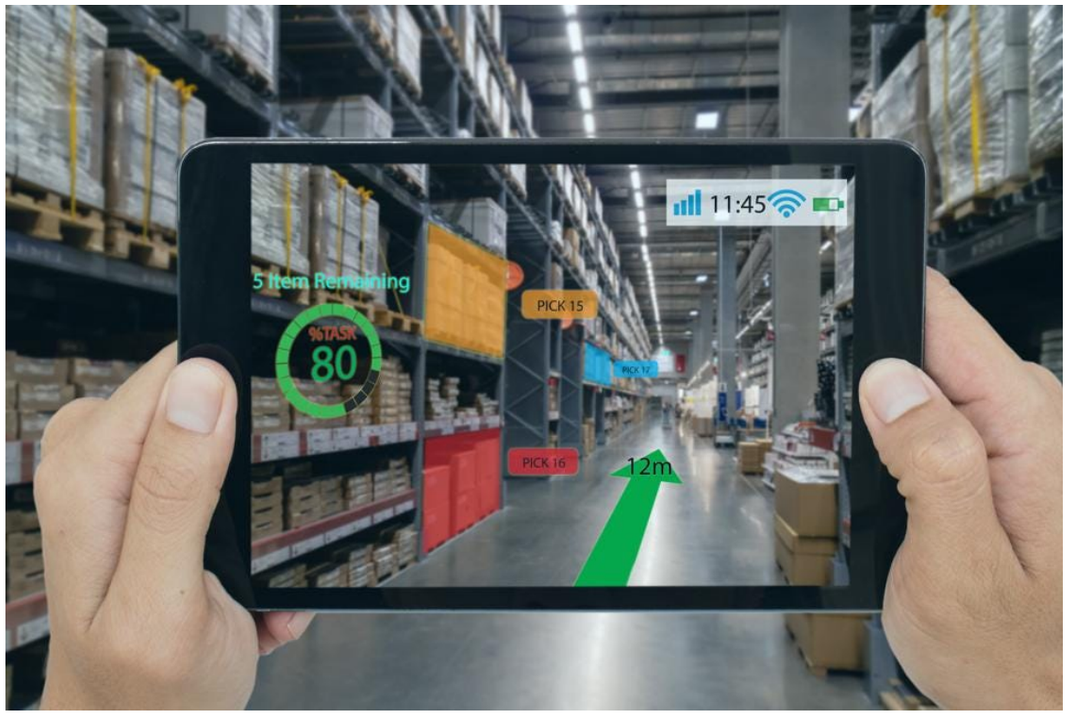
- https://nni.readthedocs.io/en/stable/experiment/web_portal/web_portal.html



Trial No.	ID	Duration	Status	Default metric
31	dvCFY	1m 22s	SUCCEEDED	0.9883
26	o4UAC	1m 22s	SUCCEEDED	0.9868
55	sYMbp	2m 32s	SUCCEEDED	0.9864
27	FkZzp	1m 22s	SUCCEEDED	0.9855
22	EkfVX	1m 21s	SUCCEEDED	0.9854
25	r9jj7	2m 48s	SUCCEEDED	0.9853
57	HpHbc	1m 23s	SUCCEEDED	0.9834
28	Bt8yE	1m 22s	SUCCEEDED	0.9833
8	YWHJu	2m 2s	SUCCEEDED	0.9828
64	JXv0r	10m 16s	SUCCEEDED	0.9825

Resource-Aware NAS

Interactive Deep Learning (DL) Applications



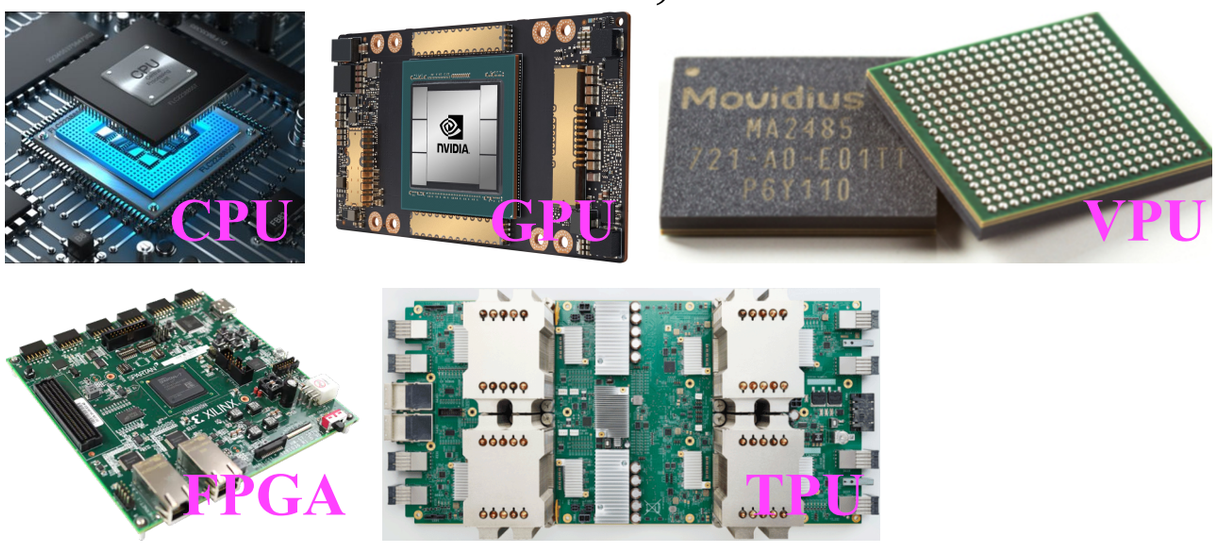
Mobile Augmented Reality (HAQ [CVPR'19])

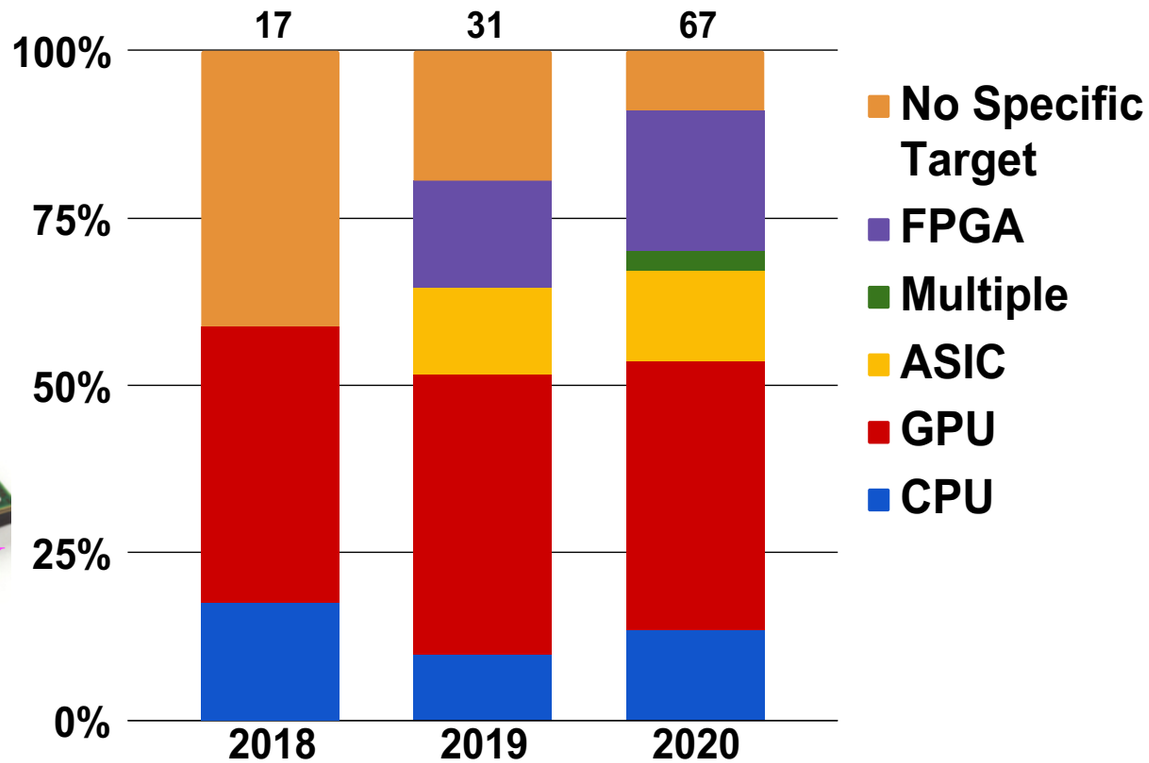
Autonomous Driving (GAMMA [ICCAD'20])

Traffic Monitoring (Mistify [NSDI'21])

Diverse Platforms and Devices

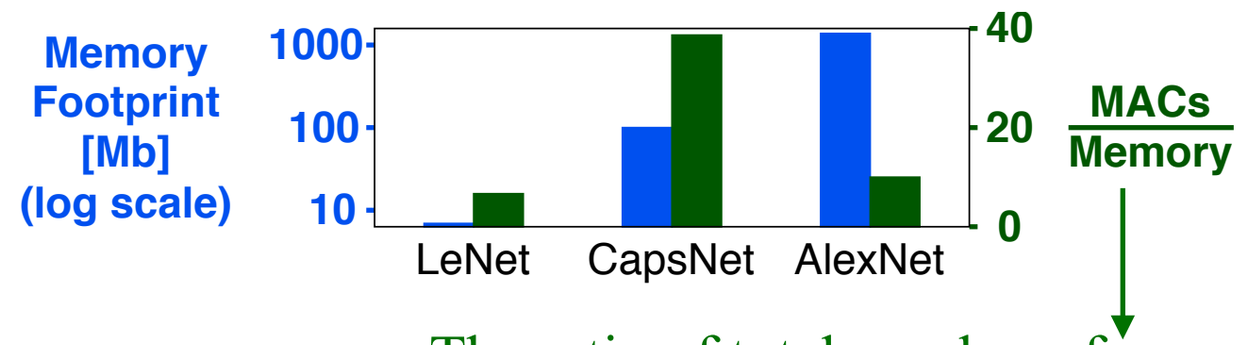
- Diverse platforms
  - Mobile devices (Fbnet [CVPR'19])
  - IoT-edge devices (MCUNet [NeurIPS'20])
  - Edge-cloud hierarchical systems (Neurosurgeon [ASPLOS'17])
  - Server clusters (Clockwork [OSDI'20])
- Diverse devices [IJCAI'21]
  - General-purpose processors: CPU, GPU, FPGA
  - AI accelerator: VPU, ASIC





Various Customized User Demands

- Inference latency
  - DL models have distinct compute-intensive nature and numbers of operators.

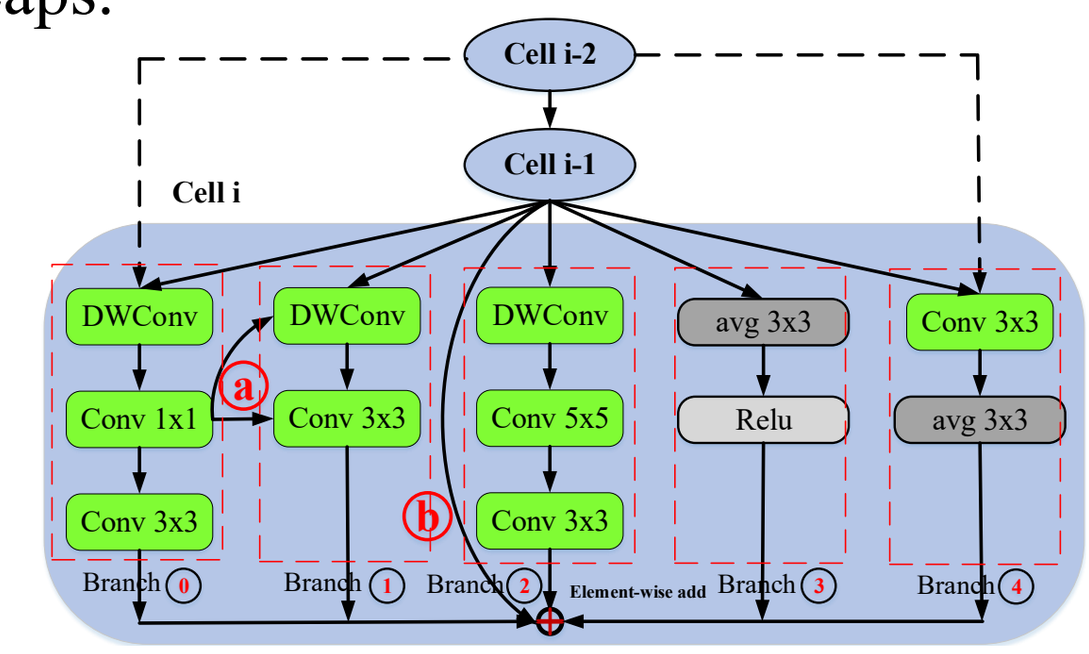


- Memory footprint

The ratio of total number of multiply-and-accumulate (MAC) operations to memory footprints

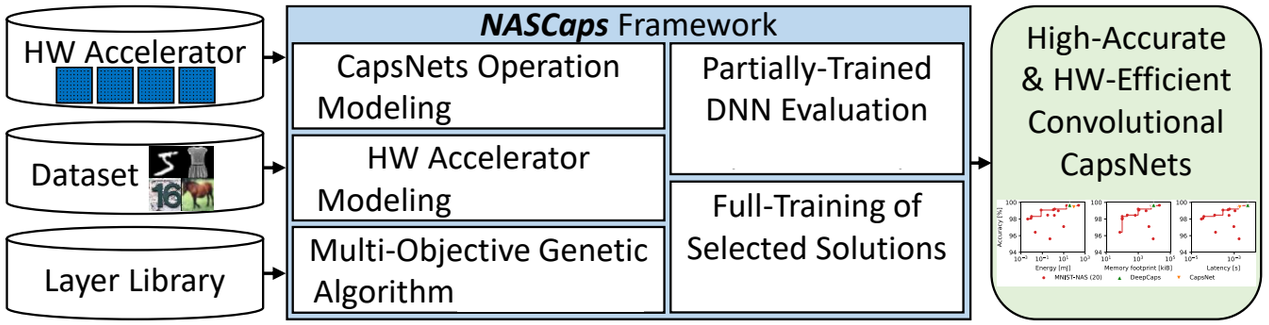
- Power/energy efficiency
  - Diverse platforms have different power caps.

- Prediction robustness/reliability

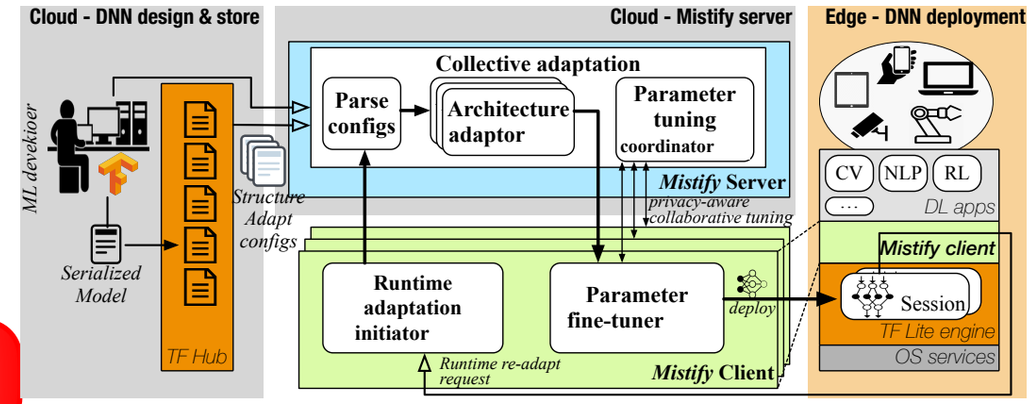


Multi-branch cell structure for robustness

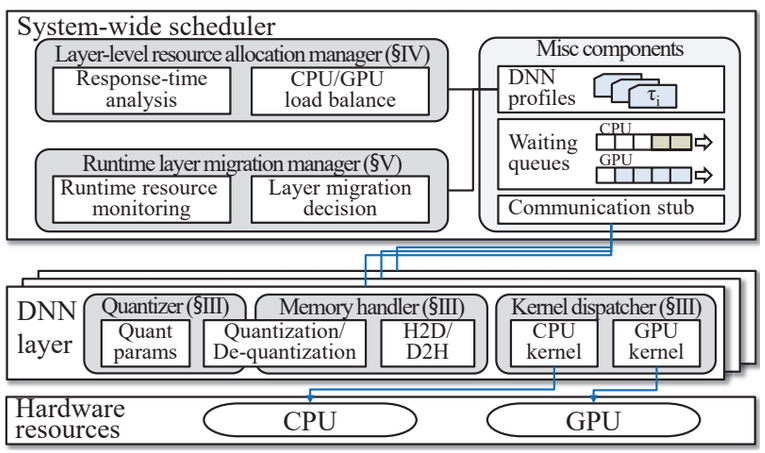
To meet customized demands, intelligent effective optimization techniques have been designed in ubiquitous DL systems on diverse platforms.



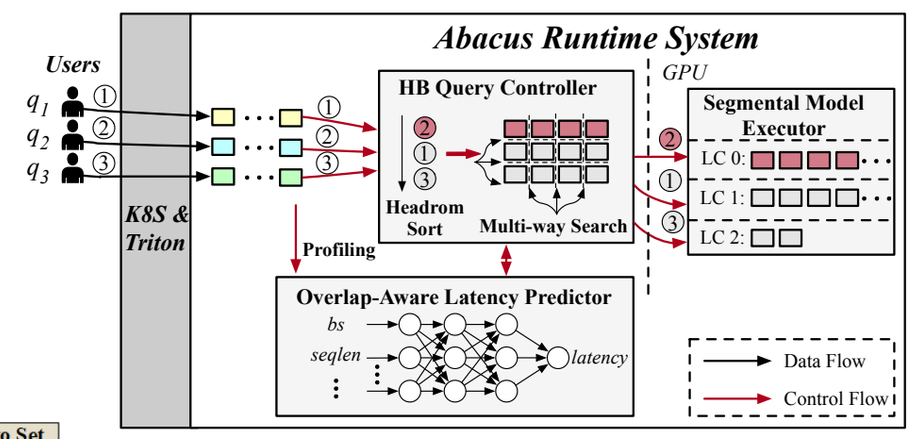
Hardware-aware neural architecture search (HW-NAS)



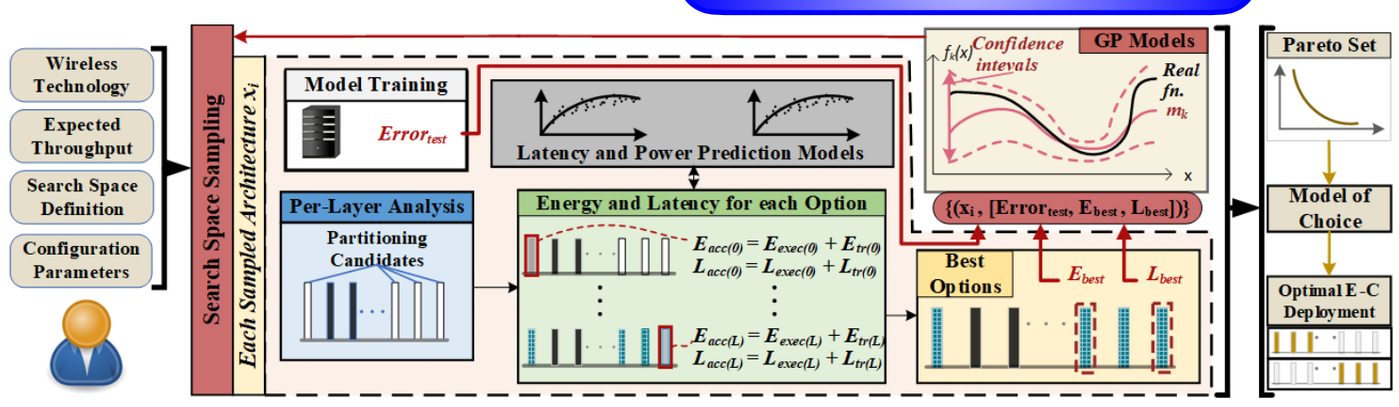
Neural architecture adaptation for deployment at the edge



Resource allocation for inference in embedded systems



Neural network (NN) scheduling in model serving

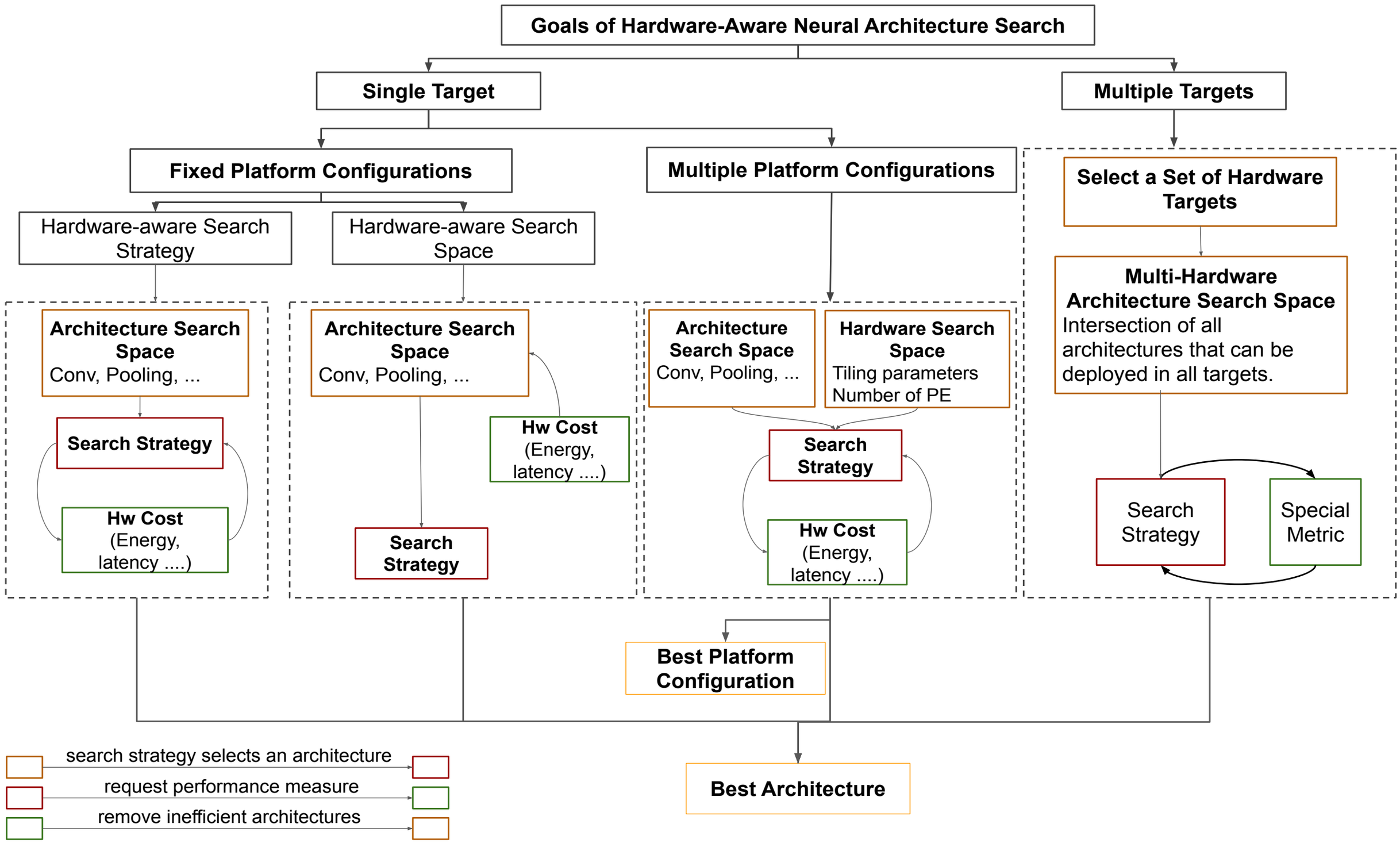


Computation offloading for neural network inference at the edge

Hardware-Aware NAS

- NAAS: Neural Accelerator Architecture Search [DAC'21]
  - https://www.youtube.com/watch?v=NYEwiWnqWaA
- Fast Hardware-Aware Neural Architecture Search [CVPR'20]
  - https://www.youtube.com/watch?v=Ij7oqAwLPOk
- NASCaps: A Framework for Neural Architecture Search to Optimize the Accuracy and Hardware Efficiency of Convolutional Capsule Networks [ICCAD'20]
- Co-Exploration of Neural Architectures and Heterogeneous ASIC Accelerator Designs Targeting Multiple Tasks [DAC'20]

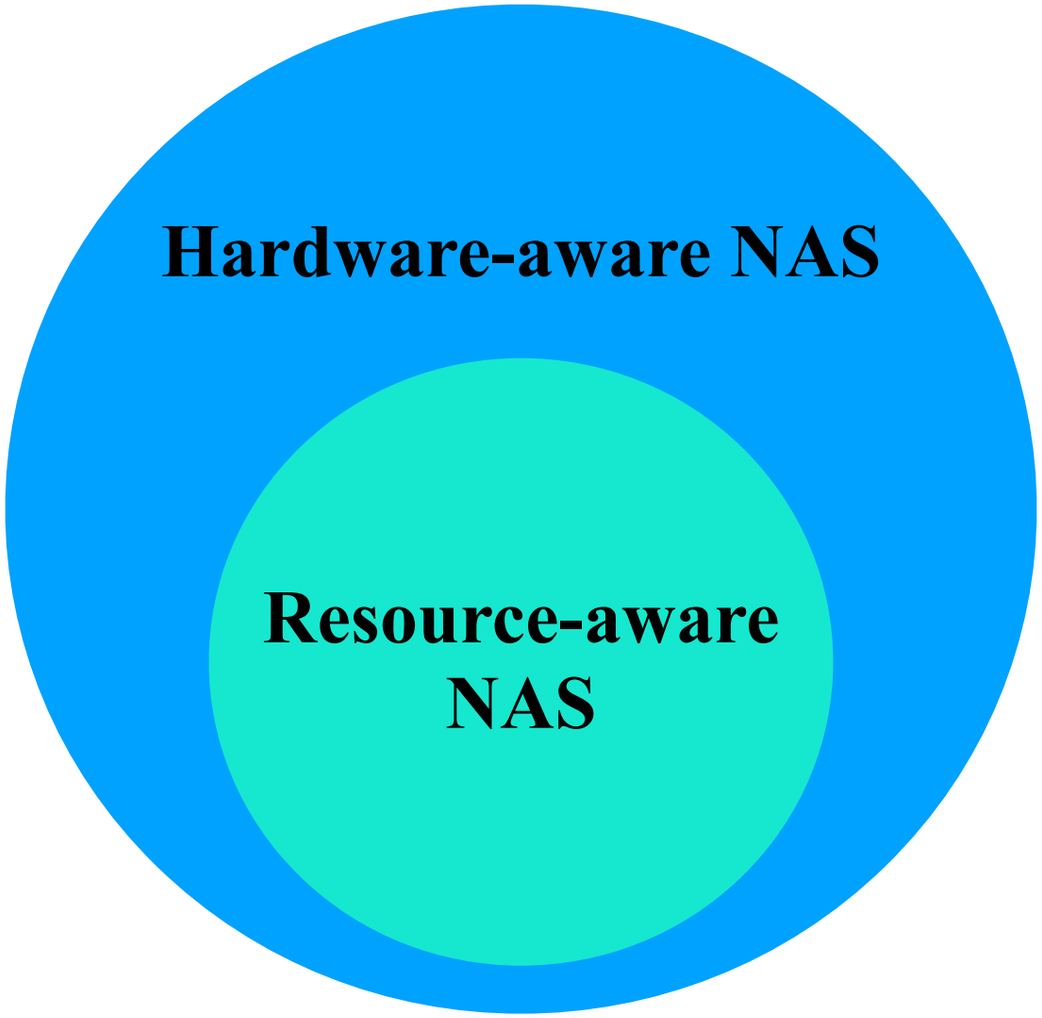
Hardware-Aware NAS



Hardware-Aware v.s. Resource-Aware NAS

- Hardware-aware NAS
  - Includes software and hardware co-design

- Resource-aware NAS
  - How to design or adapt a DL model, which is dedicated to the target hardware



Resource-Aware NAS v.s. Model Adaption

- Resource-aware NAS
  - How to design or adapt a DL model, which is dedicated to the target hardware

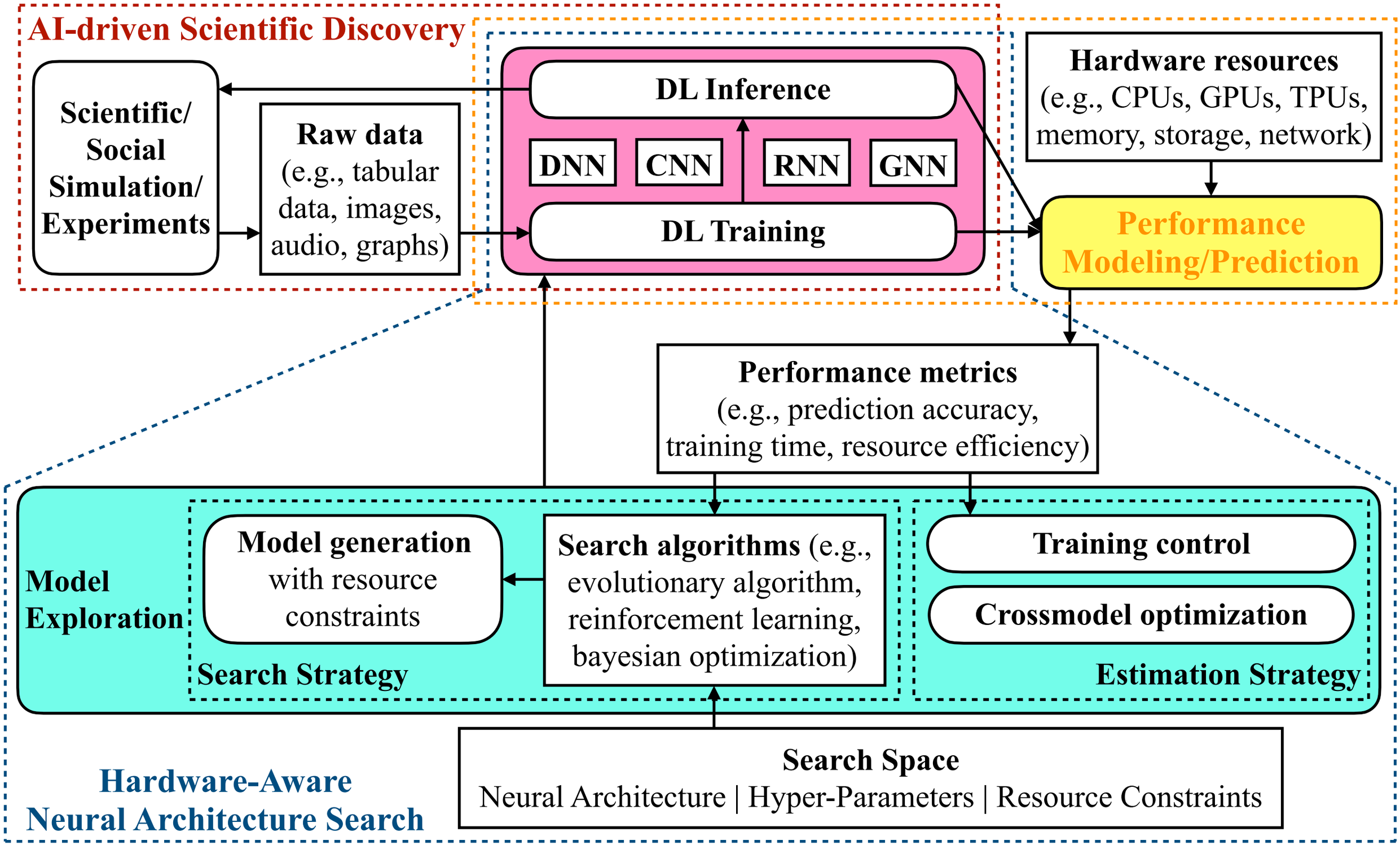
- Model adaption
  - How to change an existing DL model to fit the target hardware
    - Mistify: Automating DNN Model Porting for On-Device Inference at the Edge [NSDI'21]
      - https://www.youtube.com/watch?v=sbZBj84YKms

Scalable Resource-Aware Neural Architecture Search

- Goal
  - Jointly optimize neural architecture, hyper-parameters, resource allocation, and execution configurations of data-specific DL models on target devices to meet user requirements on prediction accuracy, inference latency, and resource consumption

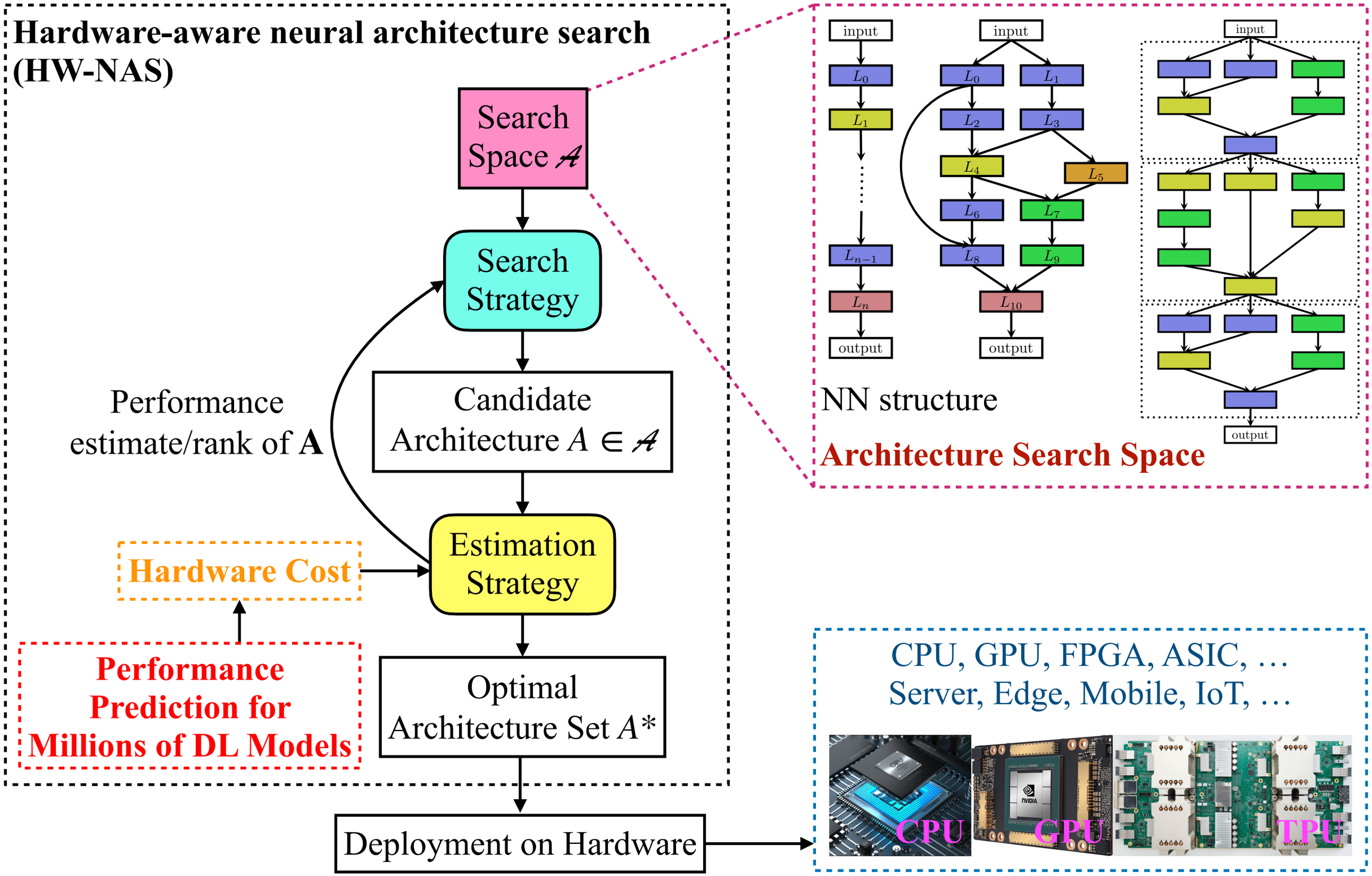
- Challenges
  - There is extremely long exploration-training time caused by multiplicatively increased computational complexity w.r.t. the number of constraints
    - Extremely huge search space in the combination of neural architecture, hyper-parameters, resource allocation, and execution configurations
  - Distinct platforms separately used for training and inference lead to inaccurate and expensive performance metric measurements

Overall Architecture of DL Systems



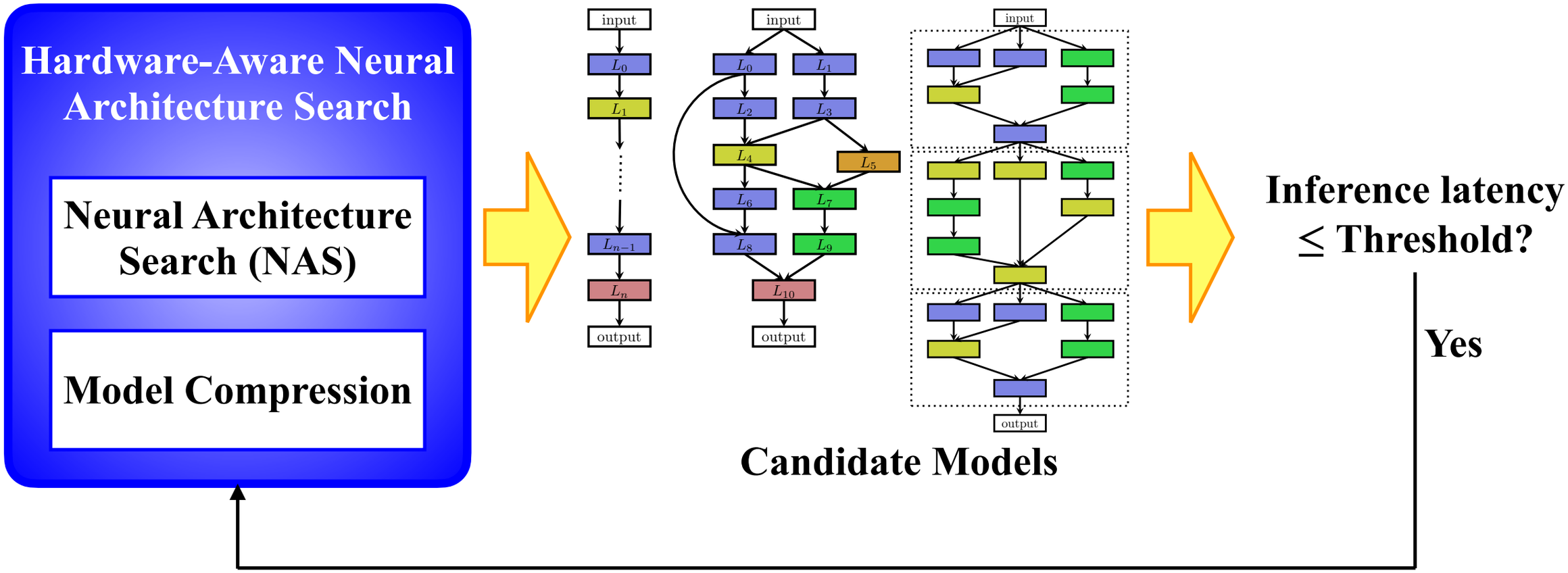
Latency Prediction of DL Inference

Resource-aware NAS heavily relies on accurate & low-cost performance modeling and prediction for DL models.



Important DL Model Design Metric: Latency

- To meet the demands on high prediction accuracy and low inference latency from real-time DL applications, the inference latency must be considered as a critical metric in the DL model design process.



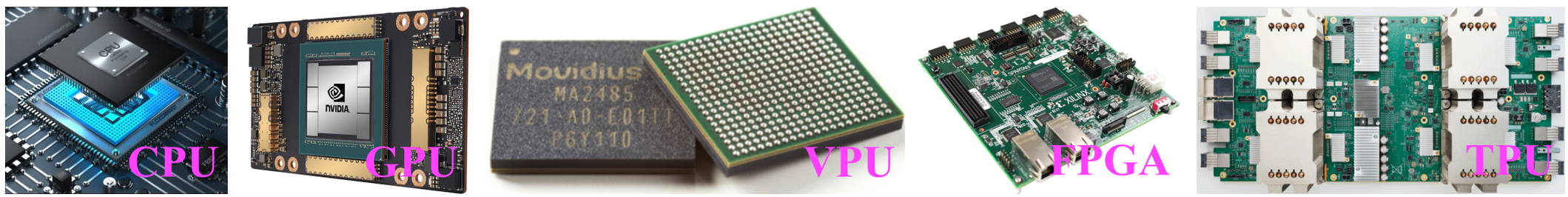
- Key problem
  - How to obtain the inference latency of a NN on various devices?

Expensive Latency Measurements on Real Devices

- Tremendous engineering efforts for model deployment
  - Diverse inference frameworks

PyTorch TensorFlow Keras ONNX Caffe2

  - Many chips



- Time-consuming to measure a huge number of models in NAS
  - E.g., ProxylessNAS explores around 0.3 million models in one search

Latency prediction for DL model inference is highly desirable.

Principles in Performance Modeling and Prediction

Analytical white-box modeling

- Divide
  - Computation on components
  - Communication and data transfer among them

&

- Conquer
  - Simple combination: sum, min, max, ….
  - Complex problem formulation

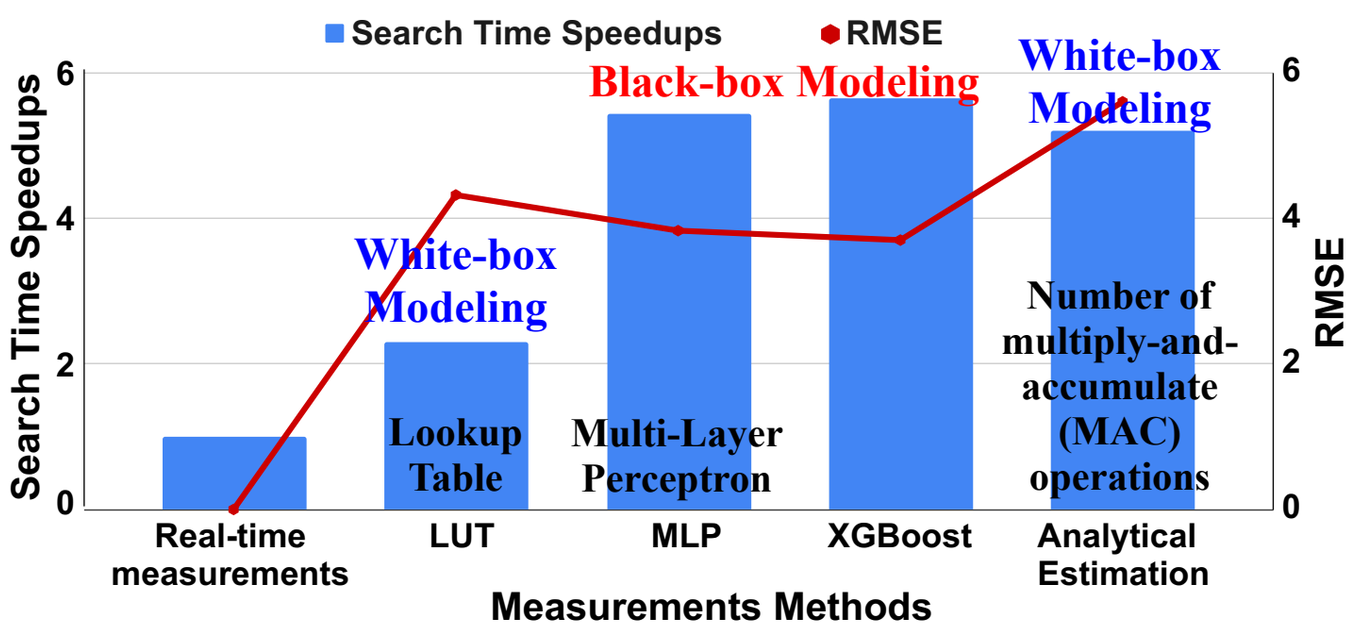
Low cost, but inaccurate prediction …

Learning-based black-box modeling

- Learning models
  - Traditional: random forest, boosted trees, …
  - DL: DNN, CNN, GRU, LSTM, GCN, …
- Learning techniques
  - Active learning, transfer learning, ensemble learning, reinforcement learning, …
  - Bayesian optimization, …

Accurate prediction, but unaffordable cost

- Existing latency prediction for DL models with these two types of methods fails to achieve a satisfactory result.



Root Mean Square Error (RMSE) is a measure of prediction errors.

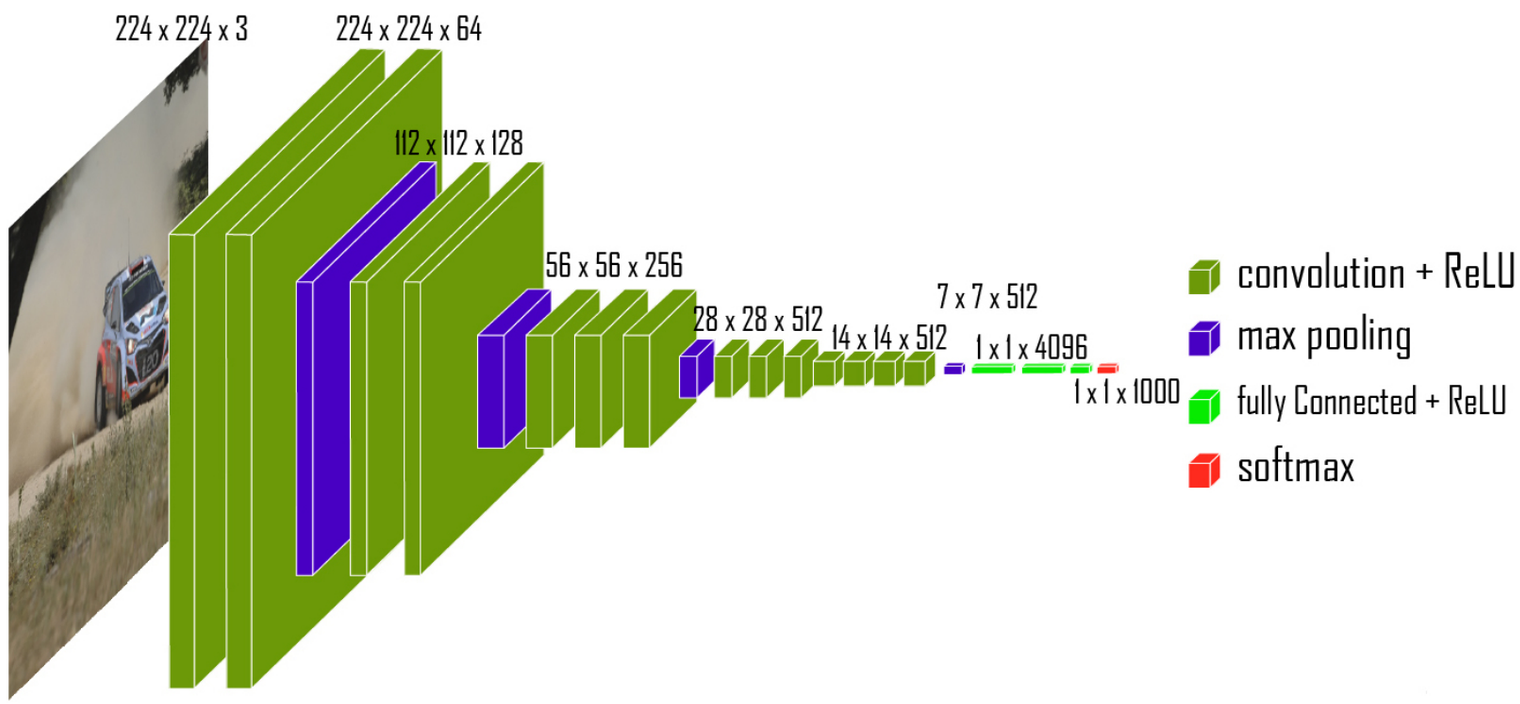
$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} (Predicted_i - Actual_i)^2}{N}}$$

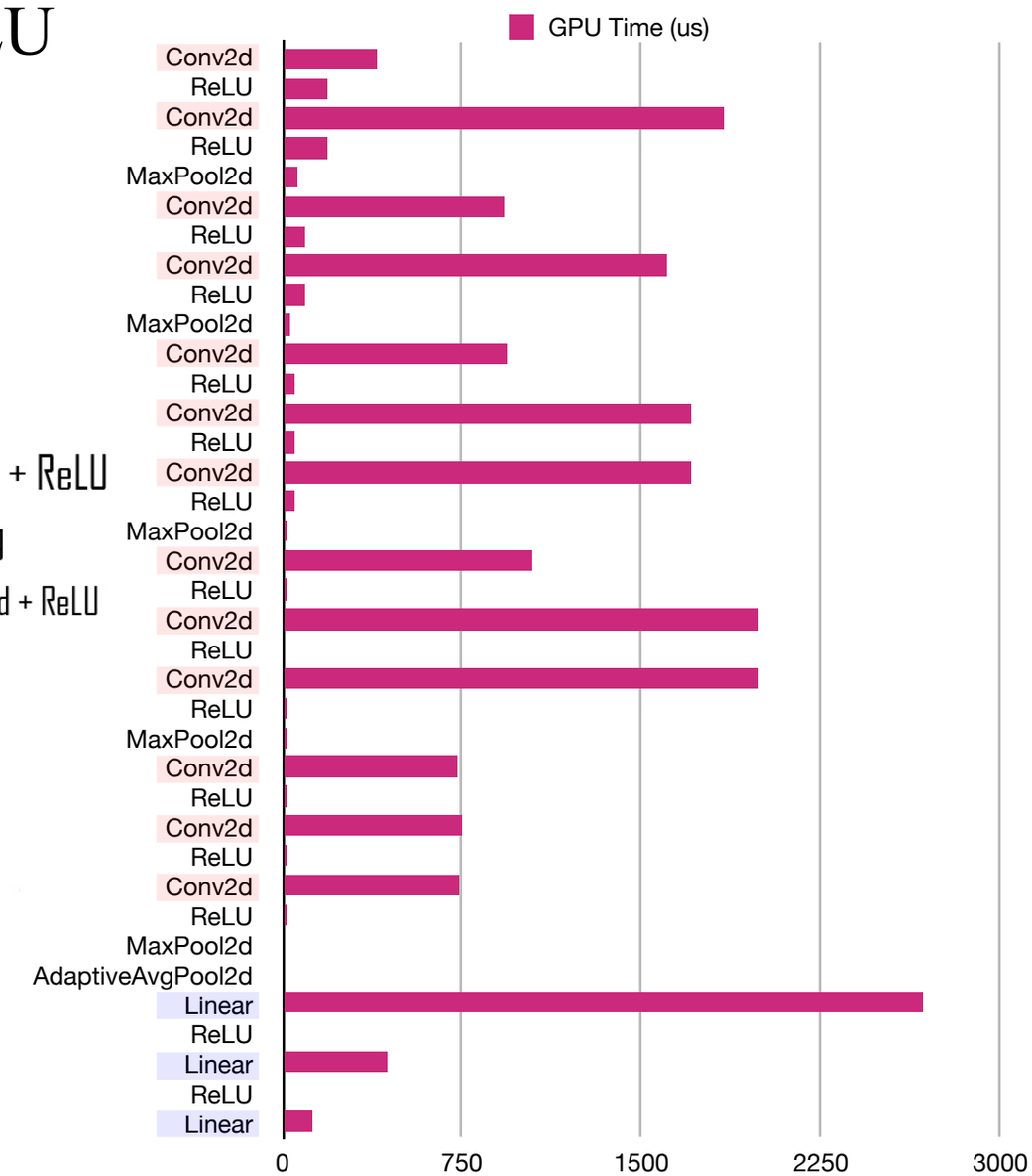
Related Work on Latency Prediction of DL Inference

Type	Method	Description	Disadvantage
White-box	FLOPs-based method [ECCV'18]	Use FLOPs as an latency indicator	FLOPs is not a direct metric of inference latency
	Operator-wise lookup table [CVPR'19]	Sum all the operators' latencies	Unaware of graph optimization at edge devices
	Proxy-device-based method [SigMetrics'21]	Exploit latency monotonicity that the architecture latency rankings on different devices are correlated	Only can predict latency rankings
Black-box	Model graph-level method [NIPS'20]	Use graph convolutional networks to learn the graph optimization	Hard to generalize on unseen graphs due to relying on the quality of training data (NN graphs)
Gray-box	Execution-unit-level method [MobiSys'21]	Use kernel detection and kernel-level predictors to capture hardware optimizations	Limited to sequential execution of DL models

Operator-level Latency Measurements

- Observation: countless models, but a limited number of operators
  - E.g., in the PyTorch operator micro-benchmarks, ~60 operators in total and ~10 operators frequently used.
- Case study: convolutional neural network (CNN) VGG-16 in PyTorch
  - Sequential model consisting of only five operators: Convolution, Maximum-Pooling, Average-Pooling, Linear, and ReLU

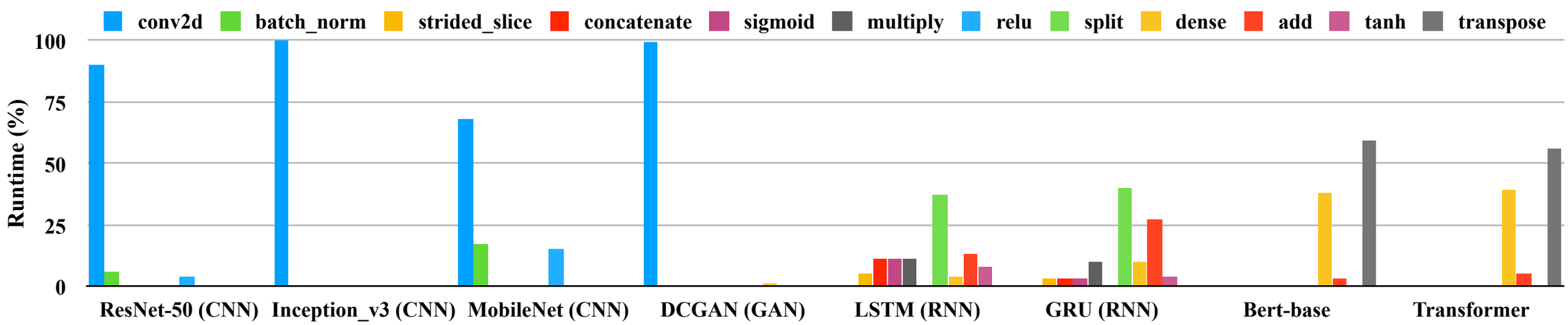




Key Operator Observation

- Key operators in DL models dominate most of model runtime.

- Example
  - The conv2d operator is the only key operator of most CNN and generative adversarial network (GAN) models and dominates over 90% of model runtime, while recurrent neural network (RNN), Bert and Transformer have more than one key operator.



End-to-end GCN-based Latency Predictor

- Graph convolutional network (GCN)
  - GCN learns models for graph-structure data.
  - Given a graph $g = (V, E)$, where V is a set of N nodes with D features and E is a set of edges, a GCN takes as input a feature description $X \in R^{N \times D}$ and a description of the graph structure as an adjacency matrix $A \in R^{N \times N}$.
  - For an L-layer GCN, the layer-wise propagation rule is $H^{l+1} = f(H^l, A) = \sigma(AH^lW^l)$
  - where H^l and W^l are the feature map and weight matrix at the l-th layer respectively, and $\sigma(\bullet)$ is a non-linear activation function like ReLU. $H^0 = X$ and H^L is the output with node-level representations.

- Predictor architecture
  - The GCN predictor has 4 layers of GCNs, with 600 hidden units in each layer, followed by a fully connected layer that generates a scalar prediction of the latency.

State-of-the-Art Latency/Resource Prediction

- BRP-NAS: Prediction-based NAS using GCNs [NIPS'20]
  - https://www.youtube.com/watch?v=USXQPjQ-sHY
- nn-Meter: Towards Accurate Latency Prediction of Deep-Learning Model Inference on Diverse Edge Devices [MobiSys'21]
  - https://www.youtube.com/watch?v=Ufs8wiwgOYU
- One Proxy Device Is Enough for Hardware-Aware Neural Architecture Search [SigMetrics'21]
  - https://www.youtube.com/watch?v=zORNqNKsadE
- Prediction of the Resource Consumption of Distributed Deep Learning Systems [SigMetrics'22]
  - https://www.youtube.com/watch?v=ON7bOCNV3nk

Question & Answer