[COMSE6998-015] Fall 2024
Introduction to Deep Learning and LLM based Generative AI Systems

Parijat Dube and Chen Wang

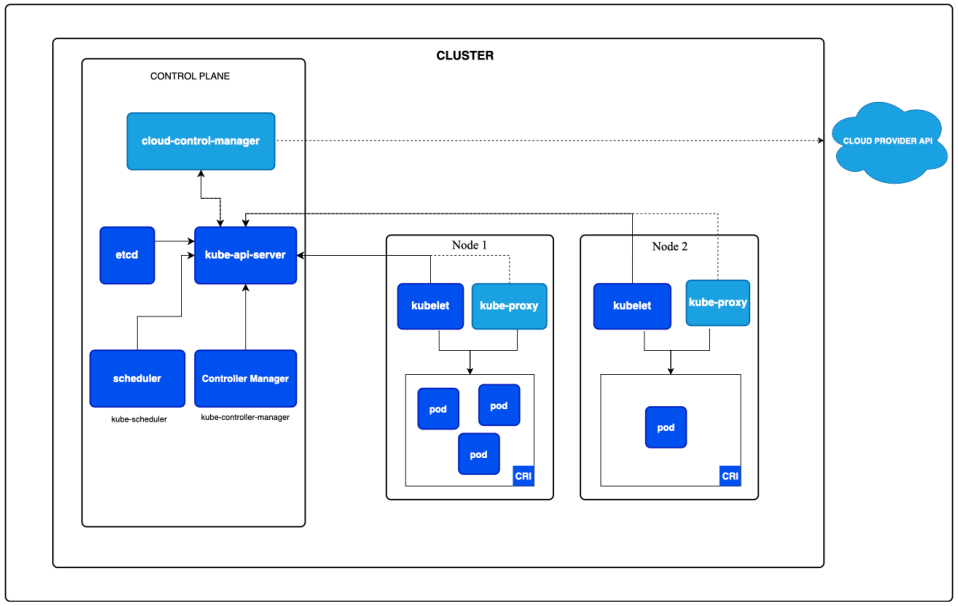
Lecture 4 09/24/24

Today's Agenda

- Job Scheduling on DL Clusters
- Automated ML
  - Neural Architecture Search
  - Hyperparameter Optimization
- Open Neural Network Exchange (ONNX)
- Introduction to Transfer learning

Scheduling Pods in K8s

- Pods are matched to Nodes so that kubelet can run them
- Steps in Scheduling
  1. Scheduler watches for newly created Pods that have no Node assigned
  2. Scheduler finds the best Node for that Pod to run on
- kube-scheduler – default scheduler in K8s



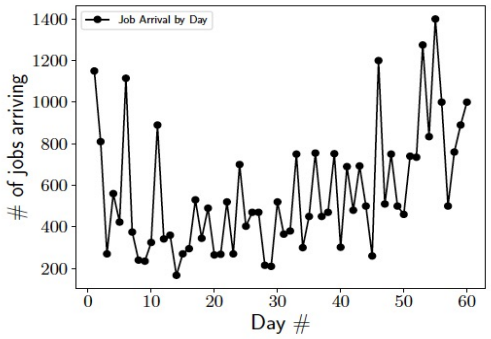
Node selection in kube-scheduler

- kube-scheduler selects a node for the pod in a 2-step operation:
  1. Filtering
  2. Scoring
- Filtering: find the set of feasible Nodes to schedule the Pod
  - Check whether a candidate Node has enough available resources to meet a Pod's specific resource requests
- Scoring: scheduler scores the feasible Nodes for Pod placement
  - scheduler assigns a score to each Node that survived filtering, basing this score on the active scoring rules
- kube-scheduler assigns the Pod to the Node with the highest ranking

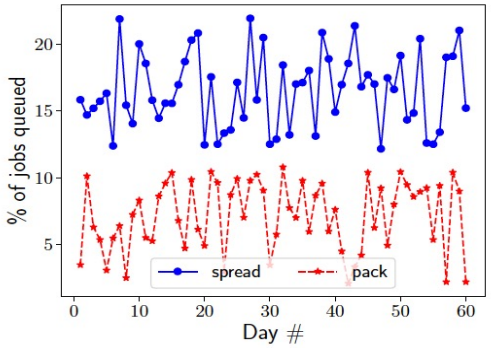
Job Scheduling: Spread vs Pack

- Spread
  - Default placement policy of K8s pods
  - Spread distributes pods over the cluster nodes, and avoids placing two pods belonging to the same job on the same physical machine
  - Problems:
    - Increased communication costs
    - Increased fragmentation
- Pack
  - Pods from a DL job are packed into as few physical machines (pod consolidation) as possible
  - Implemented as an extension to K8S scheduler

Pack results in significantly fewer jobs queued for more than 15 minutes – over 3× fewer queued jobs



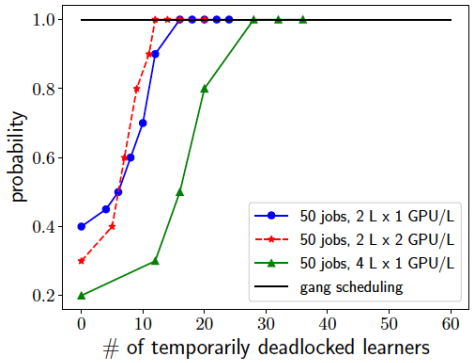
(a) Arrival of jobs by day over a 60 day period at a production cluster



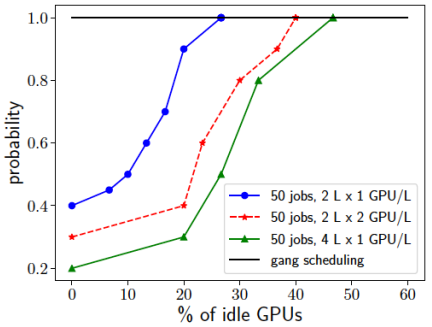
(b) Percentage of arriving jobs that would be queued for over 15 mins, in SPREAD vs. PACK

Gang Scheduling

- Ideally a job should either be fully scheduled or fully queued
- DL job can have multiple pods
- K8S scheduler doesn't consider one whole job while scheduling, it considers each of the learner pods individually
- Scheduling pods of a job individually can cause temporary deadlocks; learners are waiting and holding hardware resources while waiting for other pods (learners) of the job to get deployed
  - GPU held by the learner is idle because training has not started
- Gang scheduler schedules all pods that belong to a DL job holistically, as a group/gang



(a) CDF of the probability of temporarily deadlocked learners with and without gang scheduling



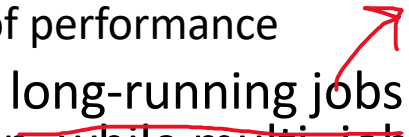
(b) CDF of the probability of idle GPUs with and without gang scheduling

Problem with current scheduling of Deep Learning Training (DLT) jobs on clusters

- Cloud operators and large companies that manage clusters of tens of thousands of GPUs rely on cluster schedulers to ensure efficient utilization of the GPUs.
- A typical cluster scheduler, such as Kubernetes or YARN (for Hadoop clusters) treats a DL job simply as a yet another big-data job
- DLT jobs are assigned a fixed set of GPUs at startup
  - Job holds exclusive access to its GPUs until completion
  - Head-of-line blocking, preventing early feedback and resulting in high queuing times for incoming jobs
  - Low GPU utilization
- Typical cluster schedulers treat DLT job as a black box

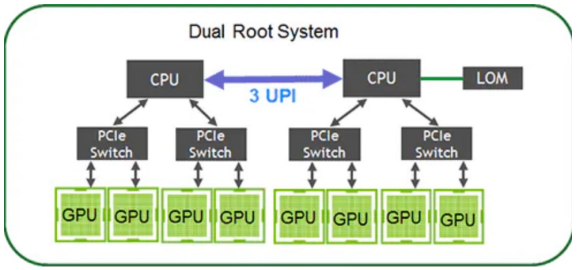
DLT Job Features

- Short jobs vs long jobs on a DLT cluster
- *Feedback-driven exploration*
  - Hyperparameter search; manual or automated
  - Do not need to run jobs to completion
  - Require early indication of performance
- *Head-of-line-blocking*, as long-running jobs hold exclusive access to the GPUs until completion, while multi- jobs depending on early feedback wait in queue
- DLT jobs are heterogeneous targeting diverse domains
  - Jobs widely differ in terms of memory usage, GPU core utilization, sensitivity to interconnect bandwidth, and/or interference from other jobs



DL Jobs: Sensitivity to Locality

Performance of a multi-GPU DLT job depends on the affinity of the allocated GPUs



Single node setting

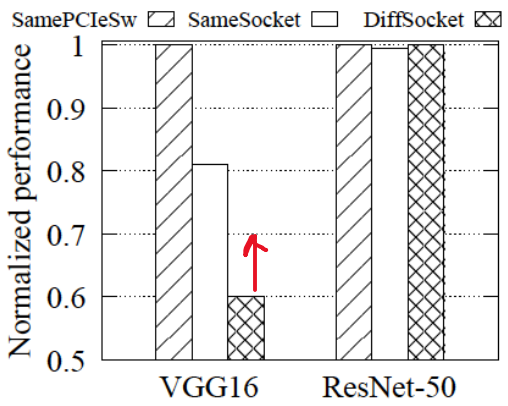


Figure 1: Intra-server locality.

Distributed setting

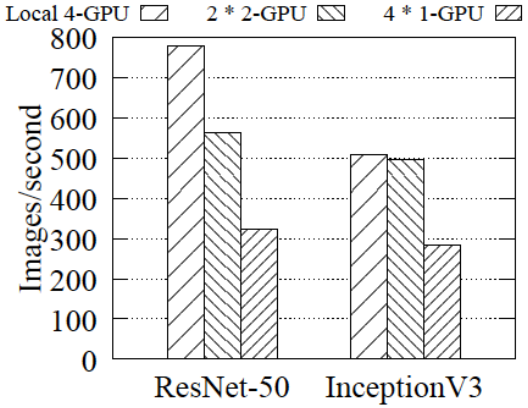


Figure 2: Inter-server locality.

- DiffSocket: GPUs on different CPU sockets
- SameSocket: GPUs in the same CPU socket, but on different PCIe switches
- SamePCIeSw: GPUs on the same PCIe switch interconnected with a 40G InfiniBand network

- Different DLT jobs exhibit different levels of sensitivity to inter-GPU affinity. Even for GPUs on the same machine, we observe different levels of inter-GPU affinity due to asymmetric architecture
- Sensitivity is model dependent; VGG16 is a larger neural model than ResNet-50
  - Model synchronization in each mini-batch incurs a higher communication load on the underlying PCIe bus.
- DLT scheduler needs to take into account a job's sensitivity to locality when allocating GPUs

Xiao et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. OSDI 2018

DL Jobs: Sensitivity to Interference

Intra-server interference

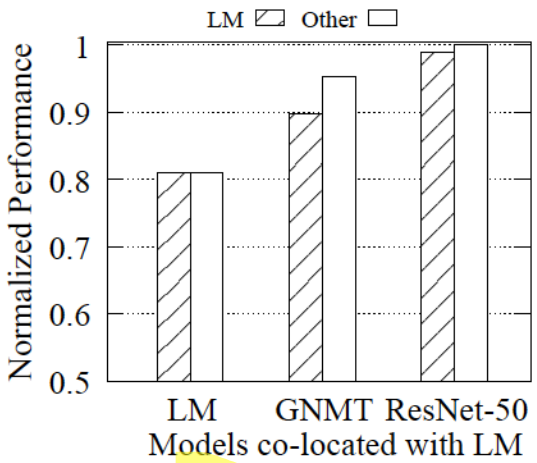


Figure 3: 1-GPU interference.

Inter-server interference

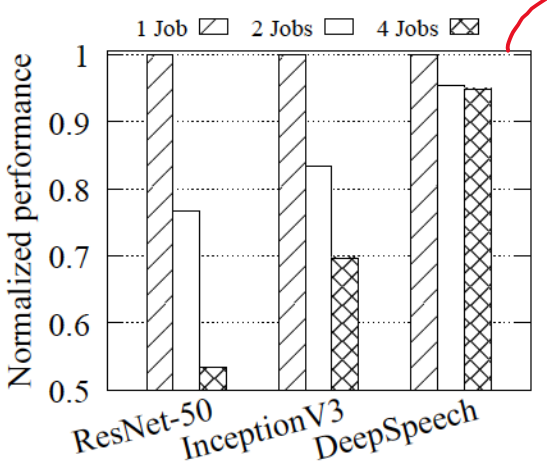
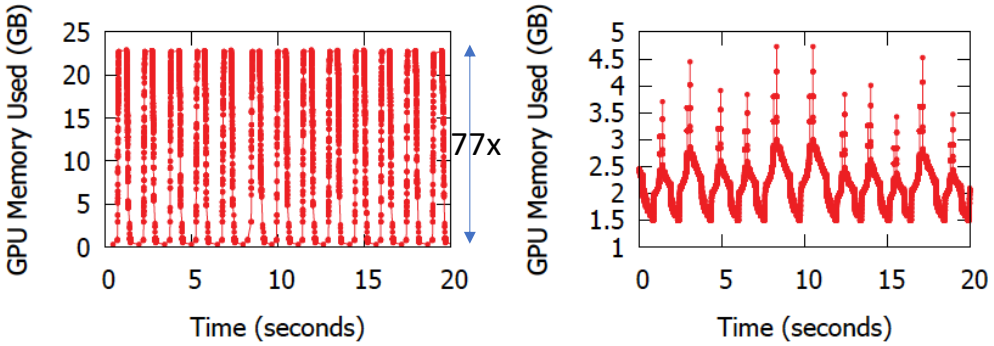


Figure 4: NIC interference.

- 2 4-GPU servers connected by 40GB InfiniBand network
- 2-GPU jobs with each GPU on a different server

- Jobs interfere caused by resource contention due the presence of multiple jobs
- Different DLT jobs exhibit different degrees of interference
- Interference exists both for single-GPU and multi-GPU jobs
- When running multiple 2-GPU jobs, where each GPU is placed on different server, ResNet-50 shows up to 47% slowdown, InceptionV3 shows 30% slow-down

DL Jobs: Intra Job Predictability



(a) ResNet50/Imagenet (b) GNMT/WMT'14 En-De

Figure 5: GPU memory usage during training.

- Gradient descent algorithm performing many mini-batch iterations
- Cyclic pattern in GPU memory used
- Each cycle corresponds to one mini-batch processing
- Leveraging predictability
  - Define micro-tasks (collection of cycles) as scheduling units
  - Very efficient suspend/resume and migration by performing them at end of cycle when GPU memory usage is minimum
- Mini-batch progress rate can be profiled and used as proxy to evaluate the effectiveness of applying performance optimization mechanisms

Xiao et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. OSDI 2018

Gandiva Scheduler

- **Goals**
  - Early feedback
    - Gandiva supports over-subscription by allocating GPUs to a new job immediately and using the suspend-resume mechanism to provide early results
  - Cluster efficiency
    - Through a continuous optimization process that uses profiling and a greedy heuristic that takes advantage of mechanisms such as packing, migration, and grow-shrink
- Modes of operations
  - Reactive
  - Introspective
- Scheduling framework to exploit the unique characteristics of the deep learning workload

Gandiva Mechanisms

- **3 ways to remove exclusivity and fixed assignment of GPUs** to DLT jobs
  - **Time-sharing GPUs:** During overload, instead of waiting for current jobs to depart, *Gandiva* allows incoming jobs to **time-share GPUs** with existing jobs. This is enabled using a custom **suspend-resume mechanism** tailored for DLT jobs along with selective packing.
  - **Job migration:** *Gandiva* supports **efficient migration of DLT jobs** from one set of GPUs to another. Migration allows time-sliced jobs to migrate to other (recently vacated) GPUs or for defragmentation of the cluster so that incoming jobs are assigned GPUs with good locality.
  - **GPU grow-shrink:** *Gandiva* supports a **GPU grow-shrink mechanism** so that idle GPUs can be used opportunistically. In order to support these mechanisms efficiently and enable effective resource management, *Gandiva* introspects DLT jobs by continuously profiling their resource usage and estimating their performance.

Gandiva Mechanisms

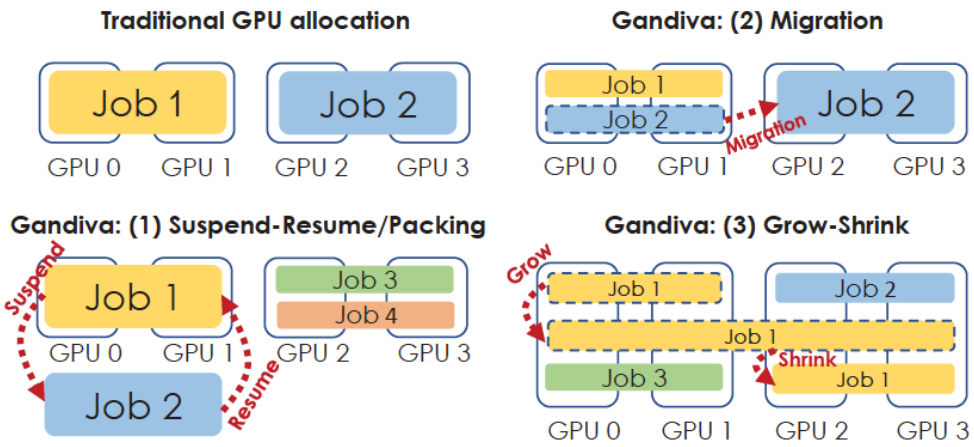


Figure 6: GPU usage options in Gandiva.

adds custom support for GPU time-slicing.

Xiao et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. OSDI 2018

Job Migration

Co-locate jobs after a departure

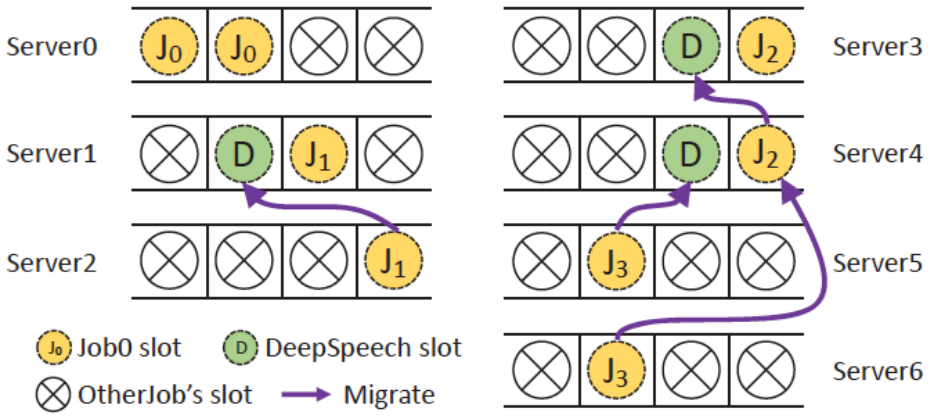


Figure 8: Job migration in a shared cluster.

Migration helpful in
i) Moving time-sliced jobs to vacated GPUs anywhere in the cluster;
ii) Migrating interfering jobs away from each other;
iii) Defragmentation of the cluster so that incoming jobs get GPUs with good locality.

Xiao et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. OSDI 2018

Gandiva Performance: Time Slicing

Each of the long job
share of GPU time drop to 4/6

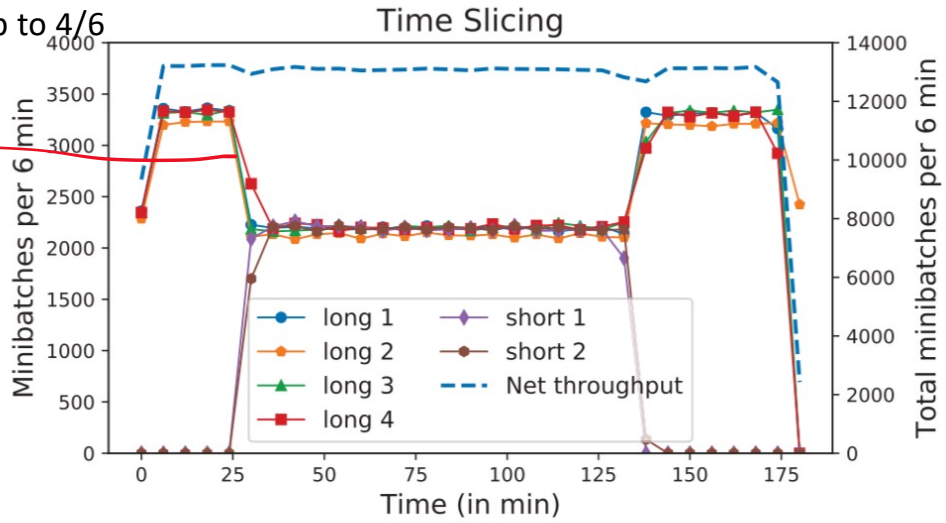
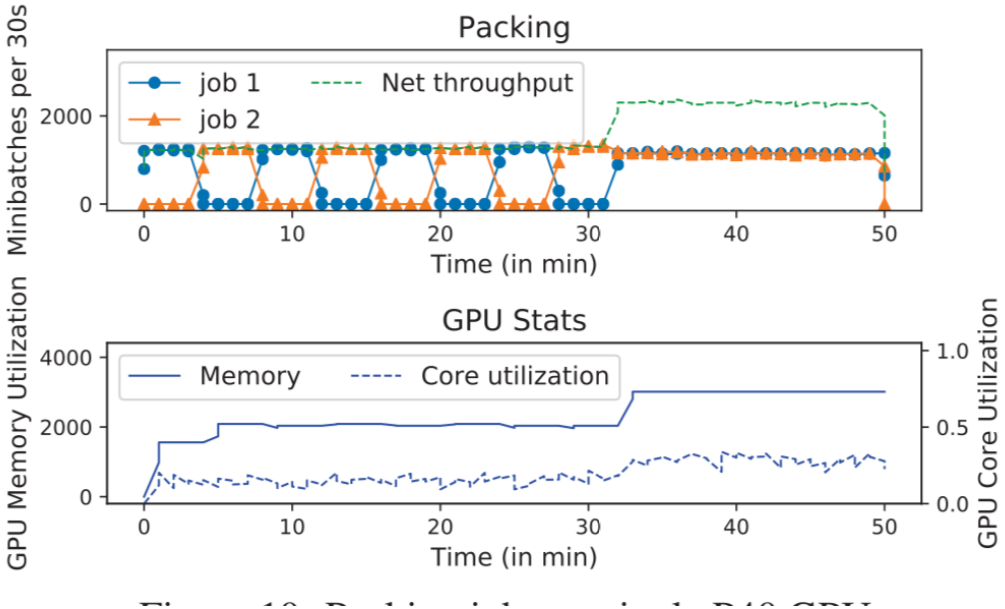


Figure 9: Time slicing six 1-GPU jobs on 4 GPUs.

Xiao et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. OSDI 2018

Gandiva Performance: Packing



Jobs are initially being time- sliced on the same GPU. After some time, the scheduler concludes that their memory and GPU core utilization is small enough that packing them is feasible and schedules them together on the GPU. The scheduler continues to profile their performance. Because their aggregate performance improves, packing is retained; otherwise (not shown), packing is undone and the jobs continue to use time-slicing.

Xiao et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. OSDI 2018

Gandiva Performance: Grow-Shrink

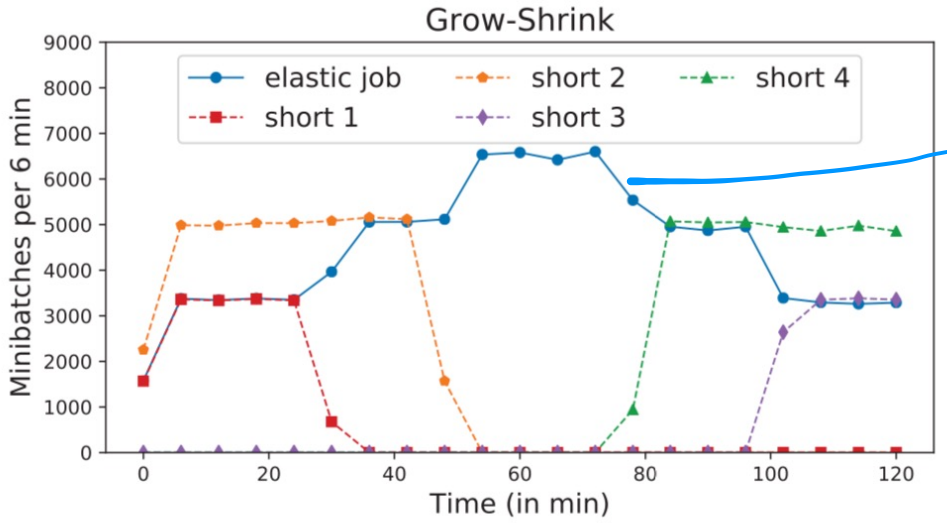


Figure 11: Grow from 1 to 4 GPUs, Shrink to 1-GPU.

Job shrinks and grows

4-P100 server

Xiao et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. OSDI 2018

Automated Machine Learning

Neural network identification
Hyperparameter optimization

- Data Preprocessing: normalization, data-augmentation
- Neural architecture search (NAS)
  - Standard, off the shelf
  - Synthesize a new
    - Types of layers (conv, maxpool), number of different layers, how to stack the layers
    - Convolution layer parameters (filter dim, stride dim)
- Hyperparameter optimization
  - Batch size, learning rate, momentum
- NAS and hyperparameter optimization can be done jointly or sequentially

Neural Architecture Search

Neural Architecture Search

Defines the neural architectures a NAS approach may discover

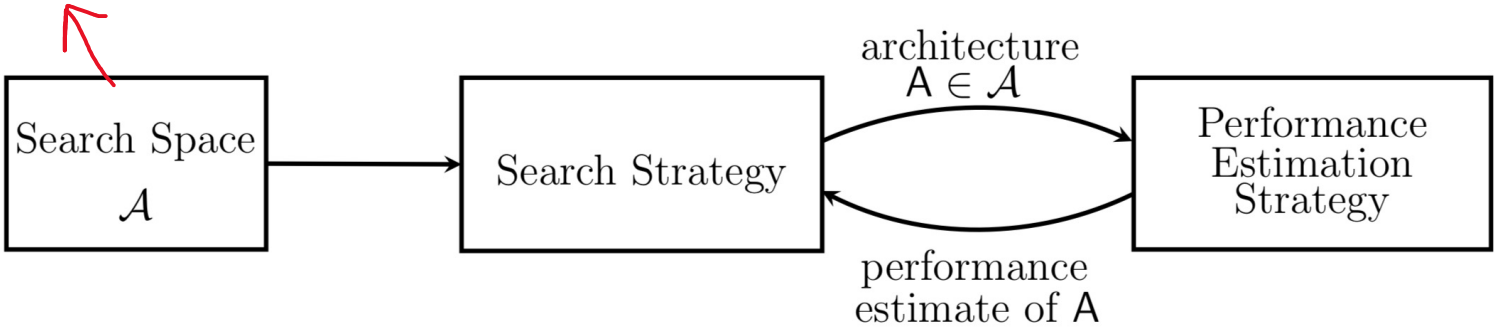
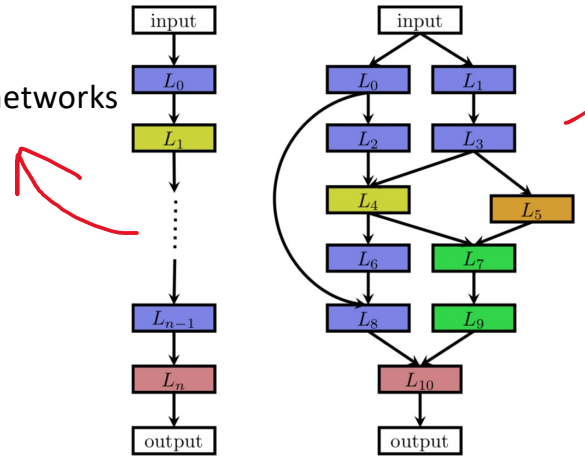


Figure 1: Abstract illustration of Neural Architecture Search methods. A search strategy selects an architecture A from a predefined search space $\mathcal{A}$. The architecture is passed to a performance estimation strategy, which returns the estimated performance of A to the search strategy.

Elsken et al. Neural Architecture Search: A Survey. JMLR 2019

NAS Search Space: Layer Based

Chain structured networks



Mutli branch networks with skip connections

Problem with layer based NAS search space

- Search space is of the order of layer types and associated parameters
  - Conv layer: filter size, number of filters, stride length, padding

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

Elsken et al. Neural Architecture Search: A Survey. JMLR 2019

NAS Search Space: Cell Based

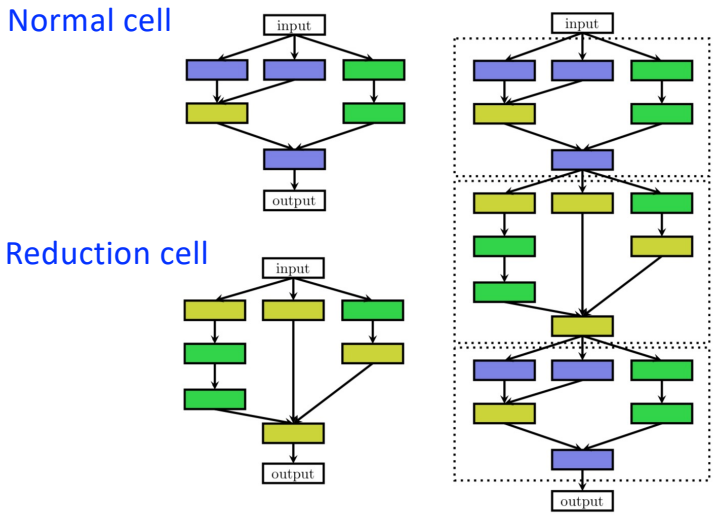


Figure 3: Illustration of the cell search space. Left: Two different cells, e.g., a normal cell (top) and a reduction cell (bottom) (Zoph et al., 2018). Right: an architecture built by stacking the cells sequentially. Note that cells can also be combined in a more complex manner, such as in multi-branch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

Motivated by hand-crafted architectures consisting of repeated motifs (cells/blocks)

- Inception blocks: normal cell, reduction cell
- Residual blocks: normal cell reduction cell

Elsken et al. Neural Architecture Search: A Survey. JMLR 2019

NAS Search Space: Cell Based

Advantages of cell-based NAS search space

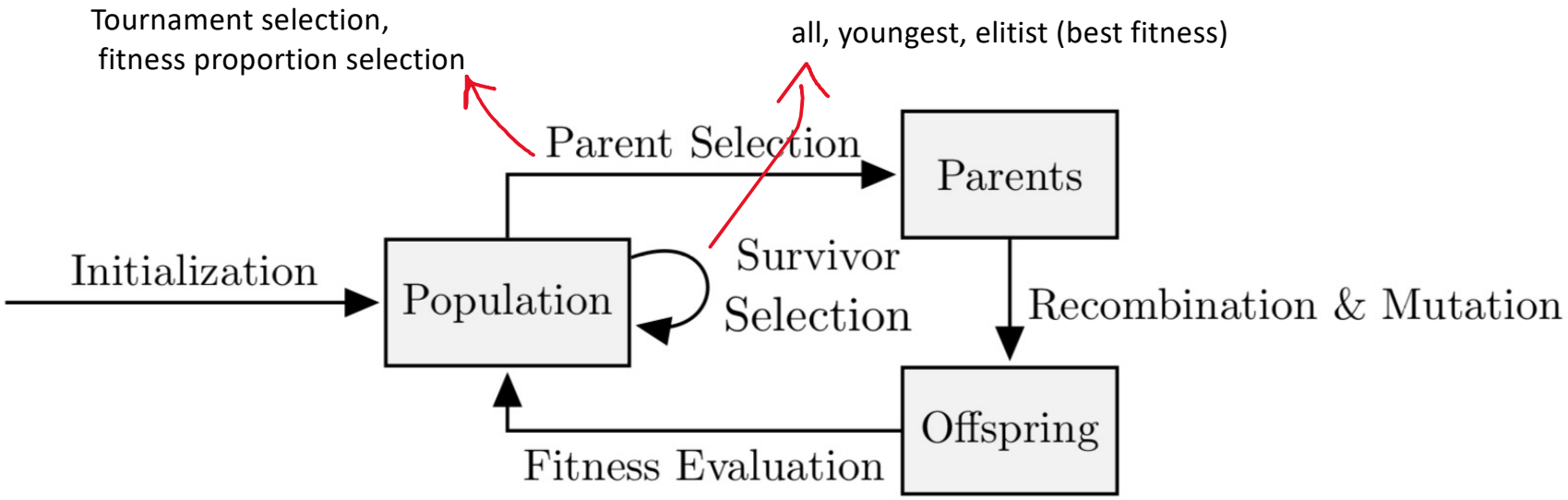
- Drastically reduced search space compared to layer based
- Easy adaptability of a network for across datasets of varying size and complexity a given domain
- Applicability across different domains, LSTM blocks in RNN, inception and residual blocks in CNNs

- Macro-architecture search problem: how many cells shall be used and how should they be connected to build the actual model?

NAS Search Strategy

- Random search
- Bayesian optimization
- **Evolutionary algorithms (EA)**
- **Reinforcement learning (RL)**
- **Gradient based methods**

EA based NAS search



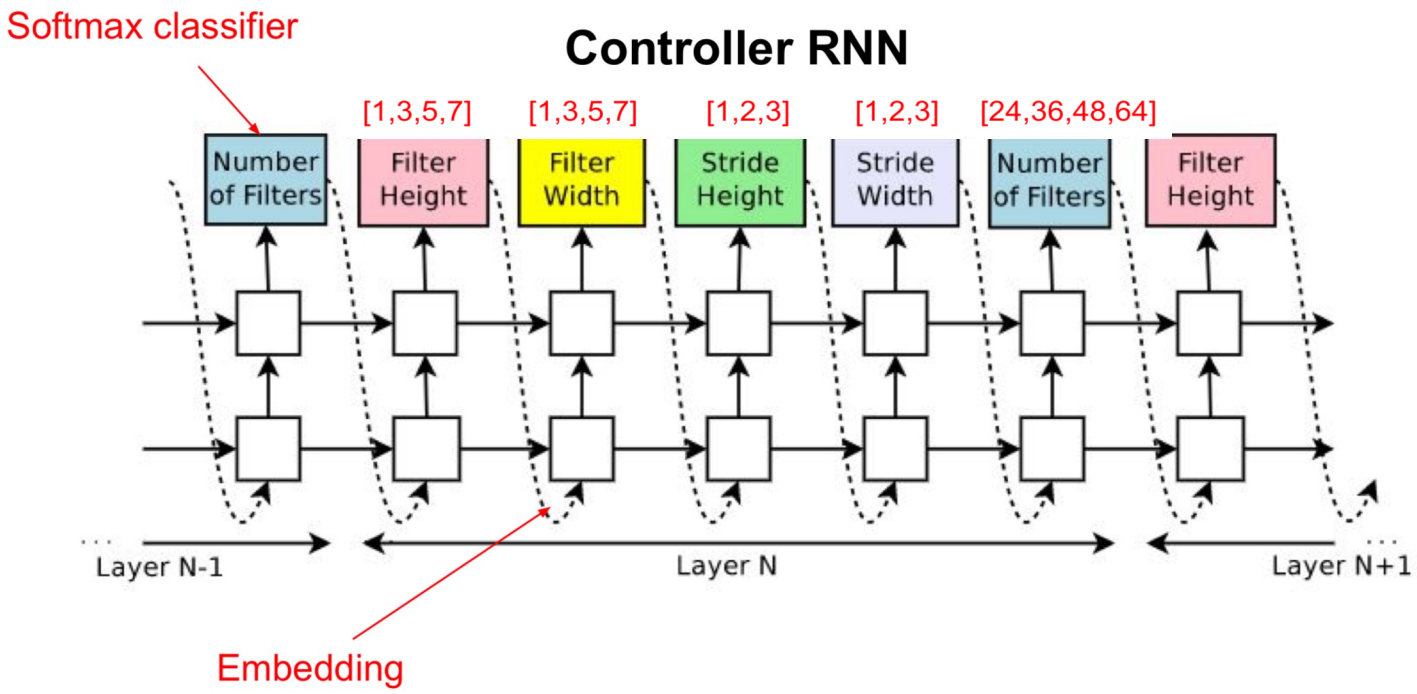
A general framework for evolutionary algorithms

Witsuba et al. A Survey on Neural Architecture Search. 2019

RL based NAS search

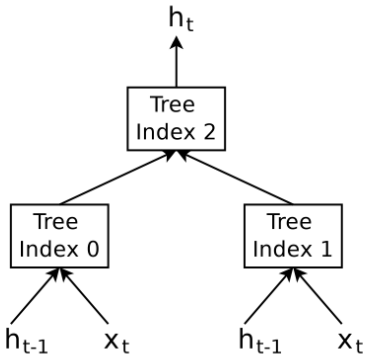
- Specify the structure and connectivity of a neural network by using a configuration string (e.g., *[“Filter Width: 5”, “Filter Height: 3”, “Num Filters: 24”]*)
- Zoph and Le (2017): Use a RNN (“Controller”) to generate this string that specifies a neural network architecture
- Train this architecture (“Child Network”) to see how well it performs on a validation set
- Use reinforcement learning to update the parameters of the Controller model based on the accuracy of the child model

Neural Architecture Search for Convolutional Networks

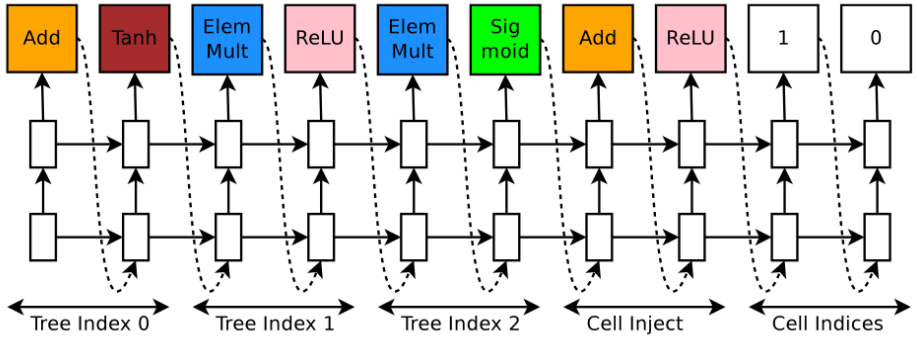


Zoph et al. NEURAL ARCHITECTURE SEARCH WITH REINFORCEMENT LEARNING. ICLR 2017

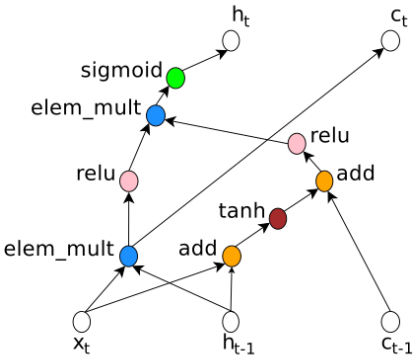
Recurrent Cell Prediction from NAS



tree that defines the computation steps to be predicted by controller



example set of predictions made by the controller for each computation step in the tree

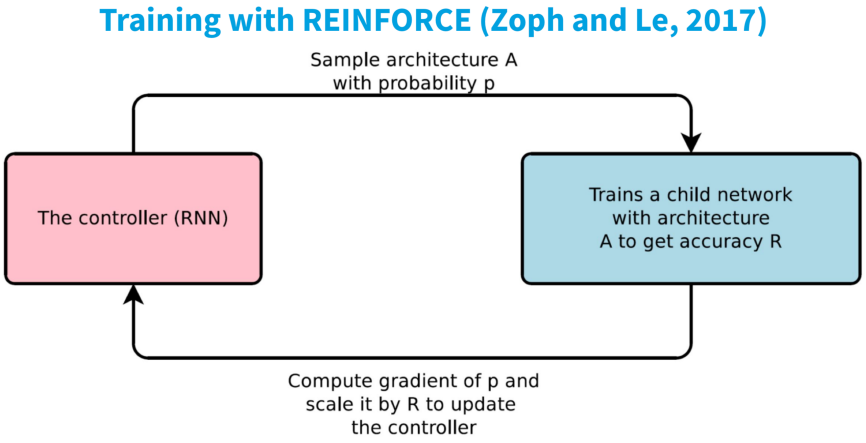


computation graph of the recurrent cell constructed from example predictions of the controller

Computation Steps in Predicted RNN Cell

- The controller predicts Add and $Tanh$ for tree index 0, this means we need to compute $a_0 = \tanh(W_1 * x_t + W_2 * h_{t-1})$.
- The controller predicts $ElemMult$ and $ReLU$ for tree index 1, this means we need to compute $a_1 = \mathrm{ReLU}\big((W_3 * x_t) \odot (W_4 * h_{t-1})\big)$.
- The controller predicts 0 for the second element of the "Cell Index", Add and $ReLU$ for elements in "Cell Inject", which means we need to compute $a_0^{new} = \mathrm{ReLU}(a_0 + c_{t-1})$. Notice that we don't have any learnable parameters for the internal nodes of the tree.
- The controller predicts $ElemMult$ and $Sigmoid$ for tree index 2, this means we need to compute $a_2 = \mathrm{sigmoid}(a_0^{new} \odot a_1)$. Since the maximum index in the tree is 2, h_t is set to a_2.
- The controller RNN predicts 1 for the first element of the "Cell Index", this means that we should set c_t to the output of the tree at index 1 before the activation, i.e., $c_t = (W_3 * x_t) \odot (W_4 * h_{t-1})$.

RL based NAS Search



Expected reward of controller

$$J(\theta_c) = E_{P(a_{1:T};\theta_c)}[R]$$

Gradient of expected reward

$$\nabla_{\theta_c} J(\theta_c) = \sum_{t=1}^{T} E_{P(a_{1:T};\theta_c)}\big[\nabla_{\theta_c} \log P(a_t | a_{(t-1):1}; \theta_c) R\big]$$

Empiricial approximation of gradient of expected reward

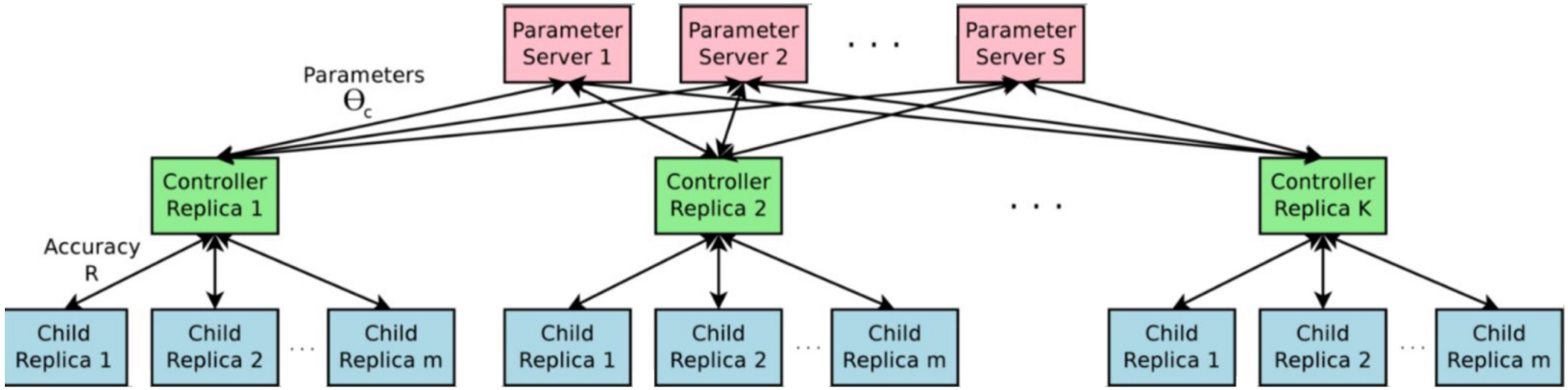
$$\frac{1}{m} \sum_{k=1}^{m} \sum_{t=1}^{T} \nabla_{\theta_c} \log P(a_t | a_{(t-1):1}; \theta_c) R_k$$

Scaling NAS

- Parameter server of S shards
- Each controller samples m child networks and train them in parallel
- Controller then collects gradients according to the results of that minibatch of m architectures at convergence and sends them to the parameter server in order to update the weights across all controller replicas

Distributed Training

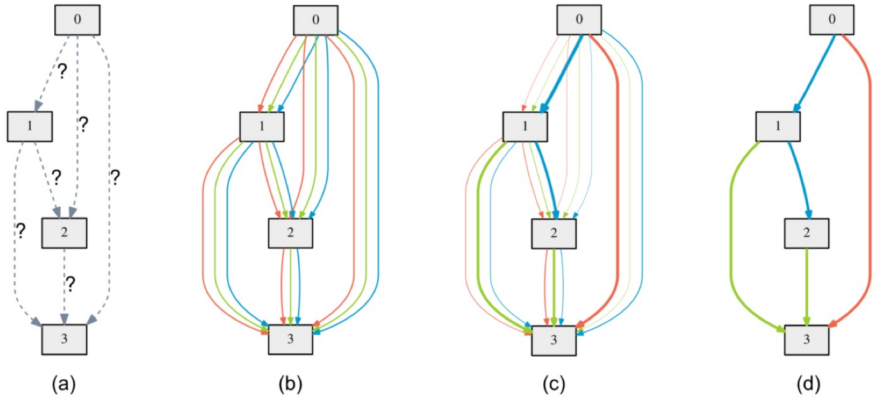
with asynchronous parameter updates



Zoph et al. NEURAL ARCHITECTURE SEARCH WITH REINFORCEMENT LEARNING. ICLR 2017

Differentiable Architecture Search

Continuous relaxation of the architecture representation, allowing efficient search of the architecture using gradient descent



$$x^{(j)} = \sum_{i<j} o^{(i,j)}(x^{(i)})$$

Figure 1: An overview of DARTS: (a) Operations on the edges are initially unknown. (b) Continuous relaxation of the search space by placing a mixture of candidate operations on each edge. (c) Joint optimization of the mixing probabilities and the network weights by solving a bilevel optimization problem. (d) Inducing the final architecture from the learned mixing probabilities.

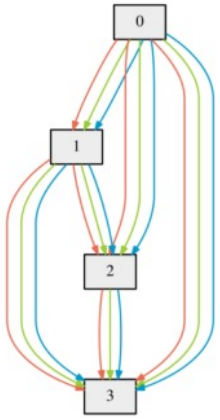
Liu et al. DARTS: DIFFERENTIABLE ARCHITECTURE SEARCH. ICLR 2019

Details of DARTS

- Computation procedure for an architecture (or a cell in it) is represented as a directed acyclic graph
- Cell-based search
- https://arxiv.org/pdf/1806.09055.pdf

relax the categorical choice of a particular operation to a softmax over all possible operations:

$$\bar{o}^{(i,j)}(x) = \sum_{o \in \mathcal{O}} \frac{\exp(\alpha_o^{(i,j)})}{\sum_{o' \in \mathcal{O}} \exp(\alpha_{o'}^{(i,}}$$



where the operation mixing weights for a pair of nodes (i, j) are parameterized by a vector α (i,j) of dimension |O|.

architecture search then reduces to learning a set of continuous variables α = α (i,j) . a discrete architecture can be obtained by replacing each mixed operation $\bar{o}^{(i,j)}$ with the most likely operation,

$$o^{(i,j)} = \text{argmax}_{o \in \mathcal{O}} \ \alpha_o^{(i,j)}$$

DARTS

- Goal is to jointly learn the architecture α and the weights w within all the mixed operations (e.g. weights of the convolution filters)
- The continuous variable α determines the operation mixing weights for different pair of nodes in the network

$\mathcal{L}_{train}$ and $\mathcal{L}_{val}$ the training and the validation loss,

- Both losses are determined not only by the architecture α, but also the weights w in the network.

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

Bi-level optimization problem

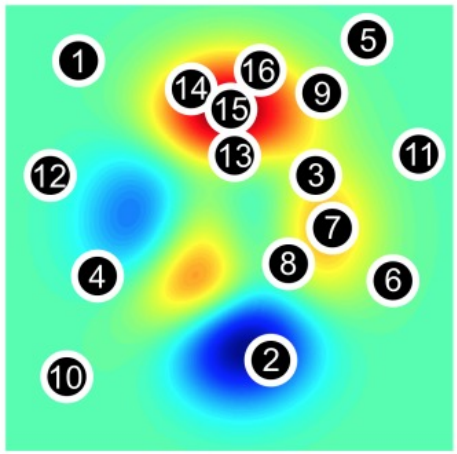
α as the upper-level variable

w as the lower-level variable

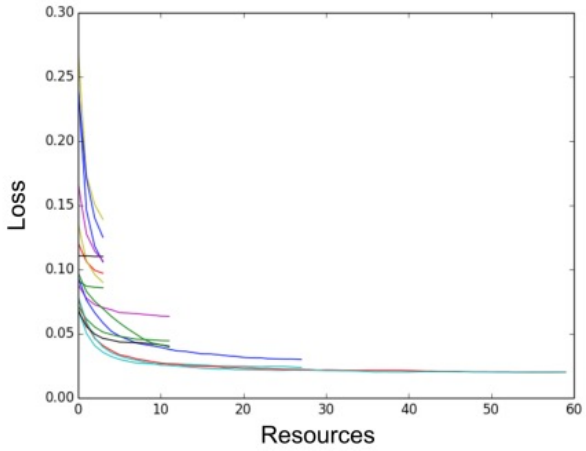
Hyperparameter Optimization

Hyperparameter optimization

- Problem of identifying good hyperparameter configuration(s) from the set of possible configurations
- Two sub-problems
  - Configuration selection
    - Efficient selection of good configuration
  - Configuration evaluation
    - Adaptive computation, allocating more resources to promising hyperparameter configurations while eliminating poor ones.



(a) Configuration Selection



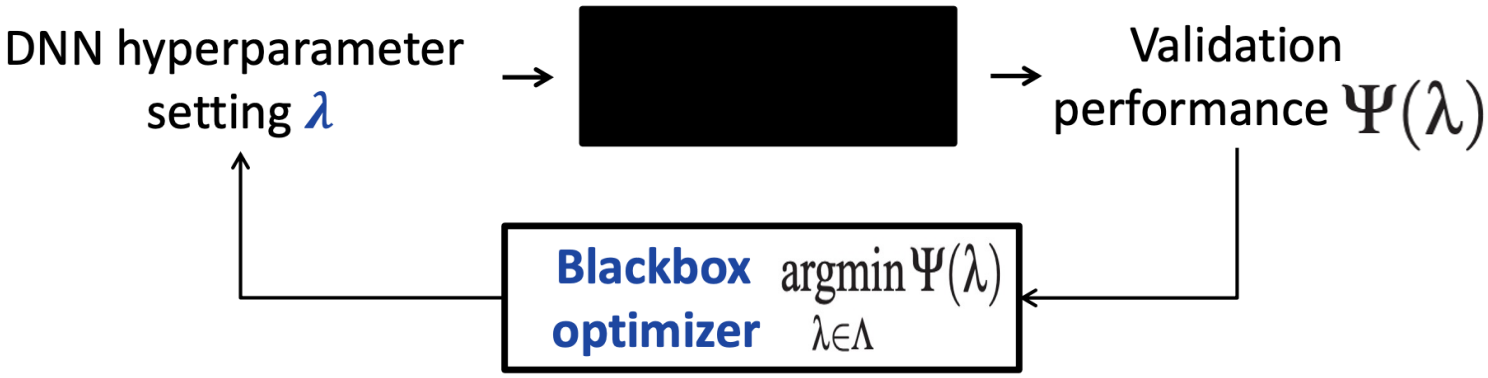
(b) Configuration Evaluation

(a) The heatmap shows the validation error over a two-dimensional search space with red corresponding to areas with lower validation error. Configuration selection methods adaptively choose new configurations to train, proceeding in a sequential manner as indicated by the numbers. (b) The plot shows the validation error as a function of the resources allocated to each configuration (i.e. each line in the plot). Configuration evaluation methods allocate more resources to promising configurations.

Hyperparameter Types

- Continuous
  - Example: learning rate
- Integer
  - Example: #units
- Categorical
  - Finite domain, unordered
    - Example 1: algo ∈ {SVM, RF, NN}
    - Example 2: activation function ∈ {ReLU, Leaky ReLU, tanh}
    - Example 3: operator ∈ {conv3x3, separable conv3x3, max pool, …}
  - Special case: binary

Blackbox Hyperparameter Optimization



- The blackbox function is expensive to evaluate
  $\rightarrow$ sample efficiency is important

Hyperparameter Optimization Formulation

$$\lambda^{(*)} = \underset{\lambda \in \Lambda}{\operatorname{argmin}} \; \mathbb{E}_{x \sim \mathcal{G}_x}[\mathcal{L}\left(x; \mathcal{A}_\lambda(\mathcal{X}^{(\text{train})})\right)]$$

$$\lambda^{(*)} \approx \underset{\lambda \in \Lambda}{\operatorname{argmin}} \; \underset{x \in \mathcal{X}^{(\text{valid})}}{\operatorname{mean}} \; \mathcal{L}\left(x; \mathcal{A}_\lambda(\mathcal{X}^{(\text{train})})\right)$$

$$\equiv \underset{\lambda \in \Lambda}{\operatorname{argmin}} \, \Psi(\lambda)$$

$$\approx \underset{\lambda \in \{\lambda^{(1)} \ldots \lambda^{(S)}\}}{\operatorname{argmin}} \; \Psi(\lambda) \equiv \hat{\lambda}$$

critical step is to choose the set of trials $\{\lambda^{(1)} \ldots \lambda^{(S)}\}$

Techniques for Hyperparameter Optimization

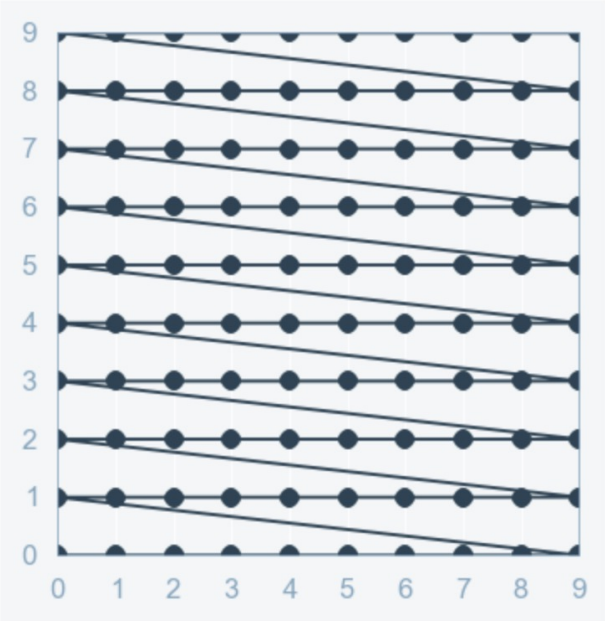
- Grid search
- Random search
- Hyperband : random configuration search with adaptive resource allocation
- Bayesian optimization methods
  - Focus on configuration selection
  - Identify good configurations more quickly than standard baselines like random search by selecting configurations in an adaptive manner
- Bayesian optimization with adaptive resource allocation

Grid Search

- Every combination of a preset list of values of the hyper-parameters and evaluate the model for each combination
- Let K be the number of hyperparameters
- grid search requires that we choose a set of values for each variable $(L^{(1)} \ldots L^{(K)})$
- Number of configurations to consider

$$S = \prod_{k=1}^{K} |L^{(k)}|$$

- Suffers from curse of dimensionality

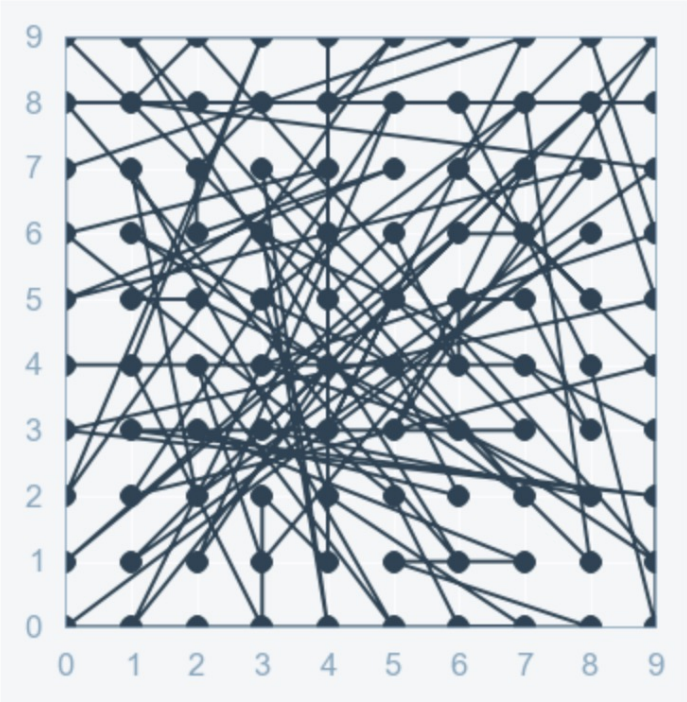


Visual Representation of grid search

Random Search

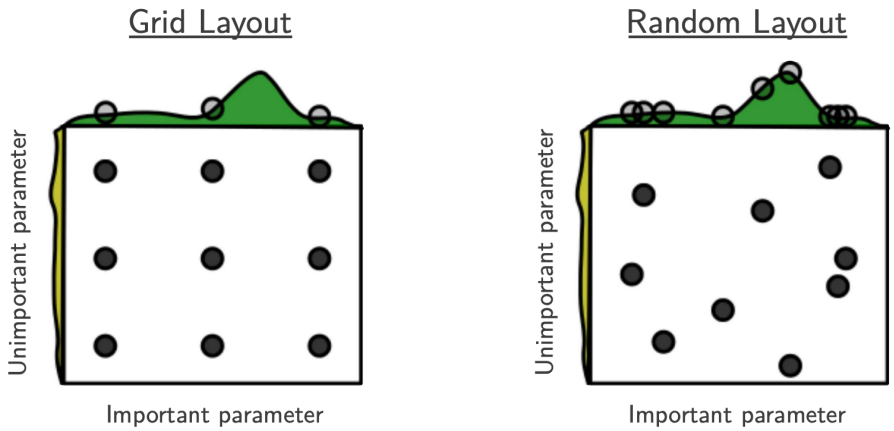
- Technique where random combinations of the hyperparameters are used to find the best solution for the built model

- Empirically and theoretically shown that random search is more efficient for parameter optimization than grid search.



Visual Representation of Random search

Grid Search Vs Random Search



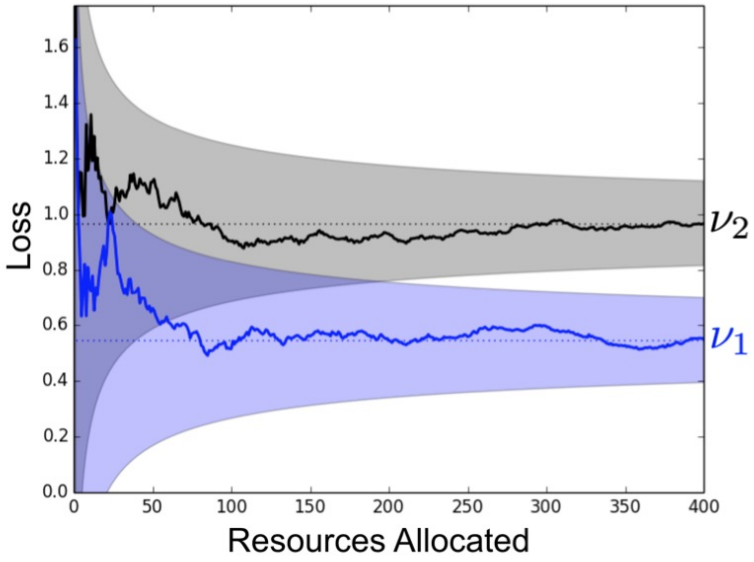
Grid and random search of nine trials for optimizing a function $f(x,y) = g(x) + h(y) \approx g(x)$ with low effective dimensionality. Above each square $g(x)$ is shown in green, and left of each square $h(y)$ is shown in yellow. With grid search, nine trials only test $g(x)$ in three distinct places. With random search, all nine trials explore distinct values of g. This failure of grid search is the rule rather than the exception in high dimensional hyper-parameter optimization.

Training resources

- Size of training set
- Number of features
- Number of iterations for iterative algorithms
- Hours of training time

Validation loss vs Resource allocated

Validation loss as a function of total resources allocated for two configurations with terminal validation losses ν_1 and ν_2



The shaded areas bound the maximum distance of the intermediate losses from the terminal validation loss and monotonically decrease with the resource.

Possible to distinguish between the two configurations when the envelopes no longer overlap

More resources are needed to differentiate between the two configurations when either
(1) the envelope functions are wider or
(2) the terminal losses are closer together.

Successive Halving

- Underlying principle: Even if performance after a small number of iterations is very unrepresentative of the *absolute* performance of any configuration, its *relative* performance compared with many alternatives trained with the same number of iterations is roughly maintained.
  - Relative ordering among configurations converges much faster than their true values
- Uniformly allocate a budget to a set of hyperparameter configurations, evaluate the performance of all configurations, throw out the worst half, and repeat until one configuration remains
- Allocates exponentially more resources to more promising configurations
- Given some finite budget B and n configurations, it is not clear a priori whether we should
  - Consider many configurations (large n) with a small average training resources; or
  - Consider a small number of configurations (small n) with longer average training resources.
- Successive Halving suffers from the "*n vs B/n*" trade-off

n Vs. B/n

- Consider a simple strategy
  - If hyper-parameter configurations can be discriminated quickly
    - *n* should be chosen large
  - If hyper-parameter configurations are slow to differentiate
    - *B/n* should be large
- Drawbacks of the simple strategy
  - If *n* is large, then some good configurations which can be slow to converge at the beginning will be killed off early.
  - If *B/n* is large, then bad configurations will be given a lot of resources, even though they could have been stopped before.

Hyperband

- Formulating hyperparameter optimization as a pure-exploration adaptive resource allocation problem addressing *how to allocate resources among randomly sampled hyperparameter configurations*.
- *Considers several possible values of n for a fixed B, in essence performing a grid search over feasible value of n*
- Hedges and loops over varying degrees of the aggressiveness balancing breadth versus depth-based search.
- Resource constrained hyperparameter optimization

Li et al. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. 2018

Algorithm 1: HYPERBAND algorithm for hyperparameter optimization.

input : R, η (default $\eta = 3$)
initialization : $s_{\max} = \lfloor \log_\eta(R) \rfloor$, $B = (s_{\max} + 1)R$

for $s \in \{s_{\max}, s_{\max} - 1, \ldots, 0\}$ **do**
  $n = \left\lceil \frac{B}{R(s+1)} \right\rceil \eta^s$ $r = R\eta^{-s}$
  // begin SUCCESSIVEHALVING with (n, r) inner loop
  $T =$ `get_hyperparameter_configuration`(n)
  for $i \in \{0, \ldots, s\}$ **do**
    $n_i = \lfloor n\eta^{-i} \rfloor$
    $r_i = r\eta^i$
    $L = \{$`run_then_return_val_loss`$(t, r_i) : t \in T\}$
    $T =$ `top_k`$(T, L, \lfloor n_i/\eta \rfloor)$
  end
end
return *Configuration with the smallest intermediate loss seen so far.*

R: max amount of resource allocated to a single configuration

η : proportion of configurations discarded in each round of SUCCESSIVEHALVING

two inputs dictate how many different brackets are considered

Bracket (outer loop)

Successive Halving (inner loop)

Hyperband

- **`get_hyperparameter_configuration(n):`** returns a set of *n* i.i.d samples from some distribution defined over the hyperparameter configuration space. Uniformly sample the hyperparameters from a predefined space (hypercube with min and max bounds for each hyperparameter).

- **`run_then_return_val_loss(t, r):`** a function that takes a hyperparameter configuration t and resource allocation *r* as input and returns the validation loss after training the configuration for the allocated resources.

- **`top_k(configs, losses, k):`** a function that takes a set of configurations as well as their associated losses and returns the top *k* performing configurations.

Hyperband: Sweeping over different s

- Each inner loop indexed by s is designed to take B total iterations
- Each value of s takes about the same amount of time on average.
- For large values of s the algorithm considers many configurations (max_iter at the most) but discards hyperparameters based on just a very small number of iterations which may be undesirable for hyperparameters like the learning_rate.
- For small values of s the algorithm will not throw out hyperparameters until after many iterations have been performed but fewer configurations are considered (logeta(max_iter)+1 at the least). The outerloop hedges over all possibilities.

Hyperband in action

Each bracket uses B total resources and corresponds to a different tradeoff of n and B/n

```
max_iter = 81
eta = 3
B = 5*max_iter
```

	$s=4$		$s=3$		$s=2$		$s=1$		$s=0$	
i	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i
0	81	1	27	3	9	9	6	27	5	81
1	27	3	9	9	3	27	2	81		
2	9	9	3	27	1	81				
3	3	27	1	81						
4	1	81								

random search

A larger value of n corresponds to a smaller r and hence more aggressive early-stopping

A single execution of Hyperband takes a finite budget of $(s_{max}+1)B$

AutoML as Hyperparameter Optimization

Definition: Combined Algorithm Selection and Hyperparameter Optimization (CASH)

Let

- $\mathcal{A} = \{A^{(1)}, \ldots, A^{(n)}\}$ be a set of algorithms
- $\mathbf{\Lambda}^{(i)}$ denote the hyperparameter space of $A^{(i)}$, for $i = 1, \ldots, n$
- $\mathcal{L}(A^{(i)}_{\boldsymbol{\lambda}}, D_{train}, D_{valid})$ denote the loss of $A^{(i)}$, using $\lambda \in \mathbf{\Lambda}^{(i)}$ trained on D_{train} and evaluated on D_{valid}.

The Combined Algorithm Selection and Hyperparameter Optimization (CASH) problem is to find a combination of algorithm $A^* = A^{(i)}$ and hyperparameter configuration $\boldsymbol{\lambda}^* \in \mathbf{\Lambda}^{(i)}$ that minimizes this loss:

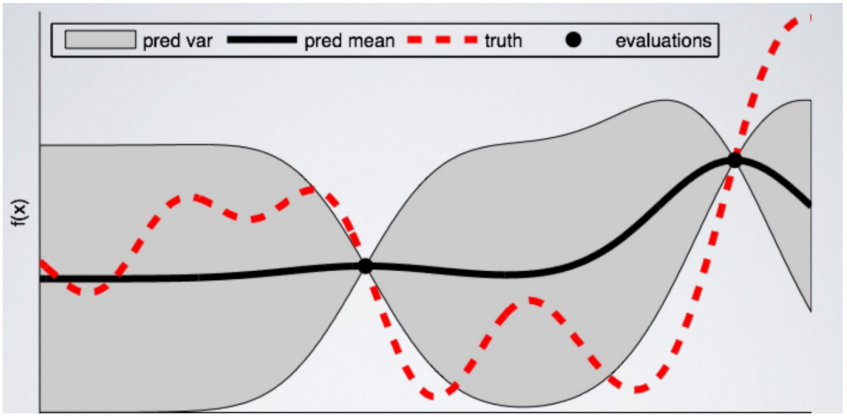
$$A^*_{\boldsymbol{\lambda}^*} \in \underset{A^{(i)} \in \mathcal{A}, \boldsymbol{\lambda} \in \mathbf{\Lambda}^{(i)}}{\arg\min} \mathcal{L}(A^{(i)}_{\boldsymbol{\lambda}}, D_{train}, D_{valid})$$

Bayesian Hyperparameter Optimization

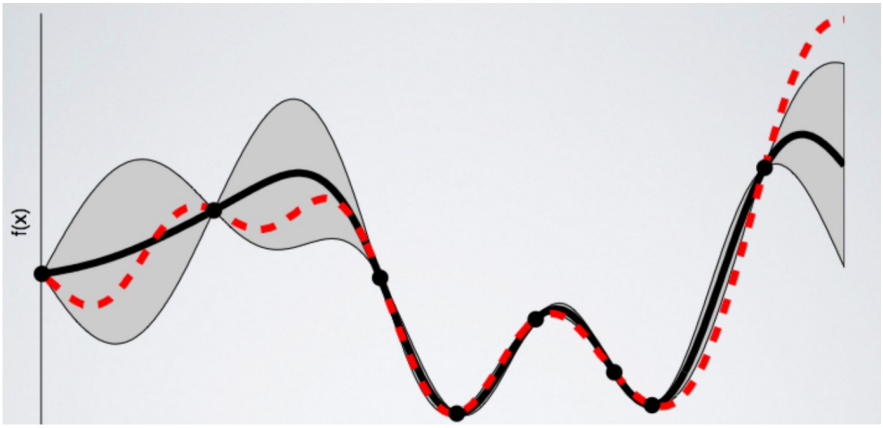
- Builds a probability model of the objective function and use it to select the most promising hyperparameters to evaluate in the true objective function.
- Approach
  1. Fit a probabilistic model to the function evaluations $\langle \lambda, f(\lambda) \rangle$
  2. Use that model to trade off exploration vs. exploitation
- Surrogate probability model for the objective function: p(score|hyperparameters)
- Steps:
  1. Build a surrogate probability model of the objective function
  2. Find the hyperparameters that perform best on the surrogate
  3. Apply these hyperparameters to the true objective function
  4. Update the surrogate model incorporating the new results
  5. Repeat steps 2–4 until max iterations or time is reached
- The surrogate probability model is updated after each evaluation of the objective function

Surrogate model performance

After 2 evaluations



After 8 evaluations

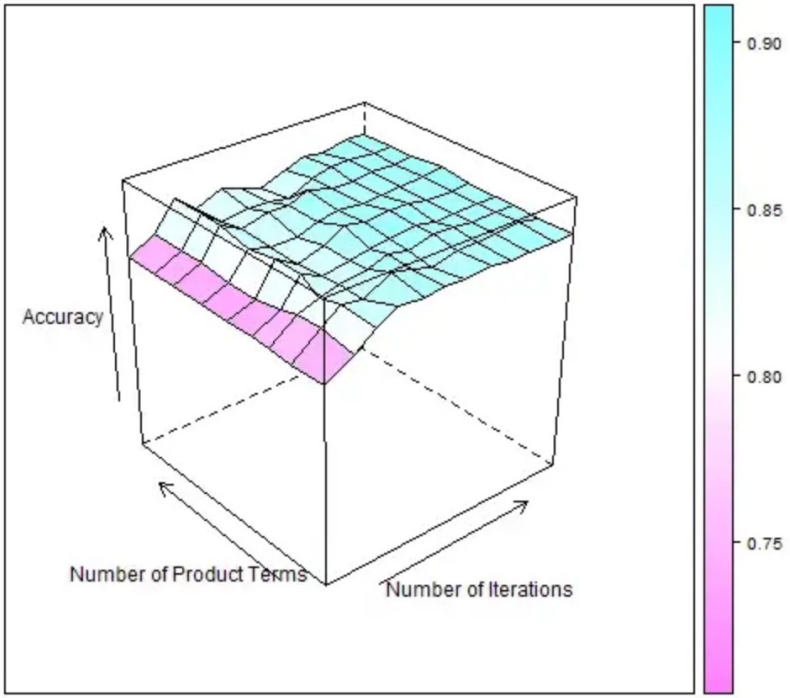


Sequential Model-Based Optimization (SMBO)

- Running trials one after another, each time trying better hyperparameters by applying Bayesian reasoning and updating a probability model (surrogate)
- Main components
  1. A domain of hyperparameters over which to search
  2. An objective function which takes in hyperparameters and outputs a score that we want to minimize (or maximize)
  3. The surrogate model of the objective function
  4. A criteria, called a selection function, for evaluating which hyperparameters to choose next from the surrogate model
  5. A history consisting of (score, hyperparameter) pairs used by the algorithm to update the surrogate model

Surrogate model

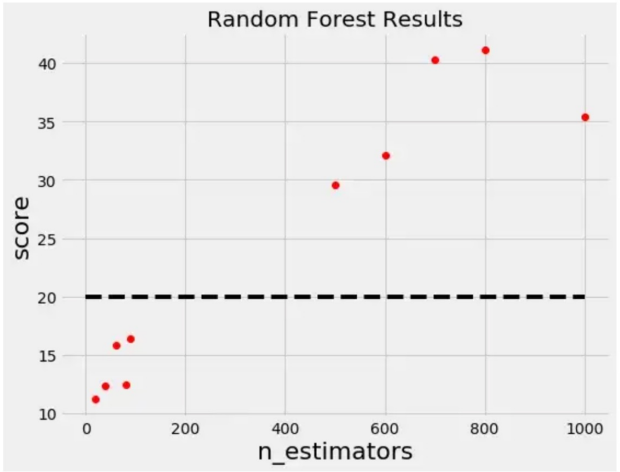
- Gaussian Processes
- Random Forest Regressions
- Tree Parzen Estimators (TPE)



Selection Function

- Expected Improvement

$$\mathrm{EI}_{y^*}(x) = \int_{-\infty}^{y^*} (y^* - y) p(y|x) dy$$

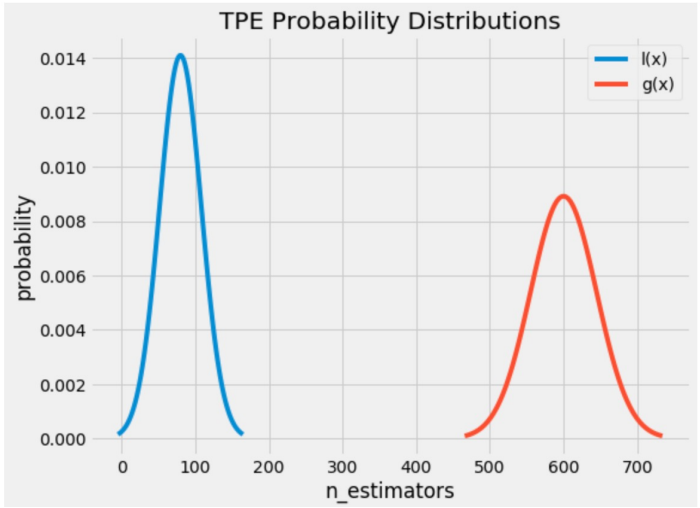


Here y* is a threshold value of the objective function, x is the proposed set of hyperparameters, y is the actual value of the objective function using hyperparameters x, and p(y | x) is the surrogate probability model expressing the probability of y given x.

Tree structured Parzen estimator

$$p(y|x) = \frac{p(x\,|y) * p(y)}{p(x)}$$

$$p(x|y) = \begin{cases} \ell(x) & \text{if } y < y^* \\ g(x) & \text{if } y \geq y^* \end{cases}$$



$$EI_{y^*}(x) = \frac{\gamma y^* \ell(x) - \ell(x) \int_{-\infty}^{y^*} p(y)dy}{\gamma\ell(x) + (1-\gamma)g(x)} \propto \left(\gamma + \frac{g(x)}{\ell(x)}(1-\gamma)\right)^{-1}$$

ONNX

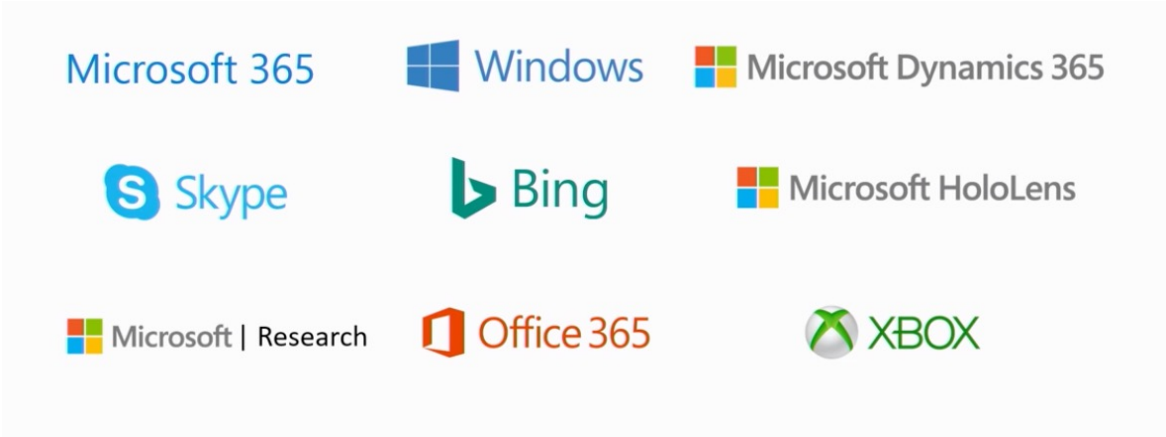
- Open Neural Network Exchange
- An open format to represent traditional machine learning and deep learning models
- ONNX Features
  - Framework interoperability
    - Models trained in one DL framework to be transferred to another for inference
  - Hardware optimizations
    - ONNX-compatible runtimes and libraries designed to maximize performance on specific DL hardware
- Visit http://onnx.ai

ONNX Capabilities

- Common set of operators related to ML
- Common file format for representing ML models
- Supported Tools
  - Visit http://onnx.ai/supported-tools
- ONNX Tutorials
  - Visit https://github.com/onnx/tutorials
- Model Zoo
  - A collection of pre-trained, state-of-the-art models in the ONNX format contributed by community members
  - Visit https://github.com/onnx/models

ONNX in Enterprise: Microsoft

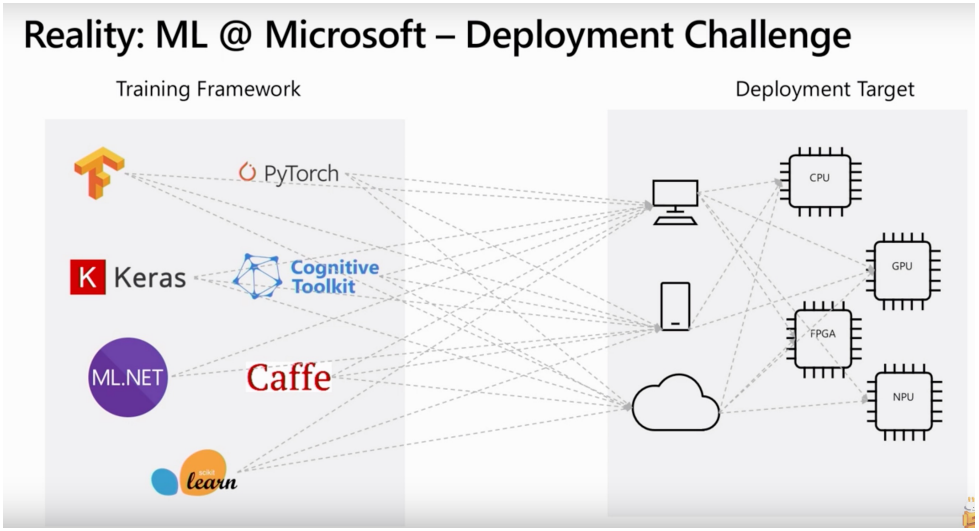
ML used in many products to improve performance and productivity



- AI developers work with different frameworks
- Deployment of trained models to production
  - Different deployment targets: cloud, IoT devices, edge devices
  - Different hardware: CPUs, GPUs, TPUs, FPGAs

Youtube Video: Open Neural Network Exchange (ONNX) in the enterprise: how Microsoft scales ML

Multiple ML frameworks + multiple deployment targets



- Maintaining multiple frameworks in deployment
  - Not scalable
  - Hard to maintain
  - Degrades application performance
    - Frameworks compete for system resources
- Decouple training and deployment frameworks

Youtube Video: Open Neural Network Exchange (ONNX) in the enterprise: how Microsoft scales ML

Bridge Model Training and Deployment

Open and Interoperable industry wide standard

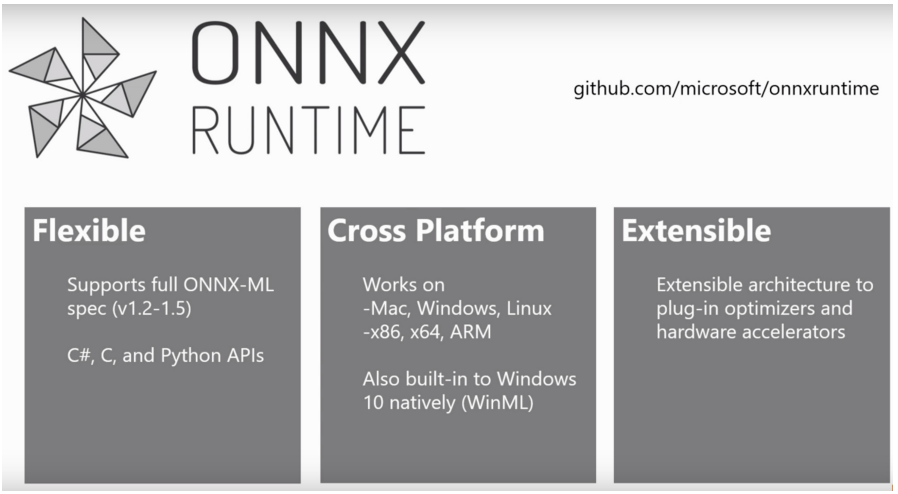
Build and train model in framework A

Convert trained model in framework A to ONNX model

Convert ONNX model to framework B and deploy

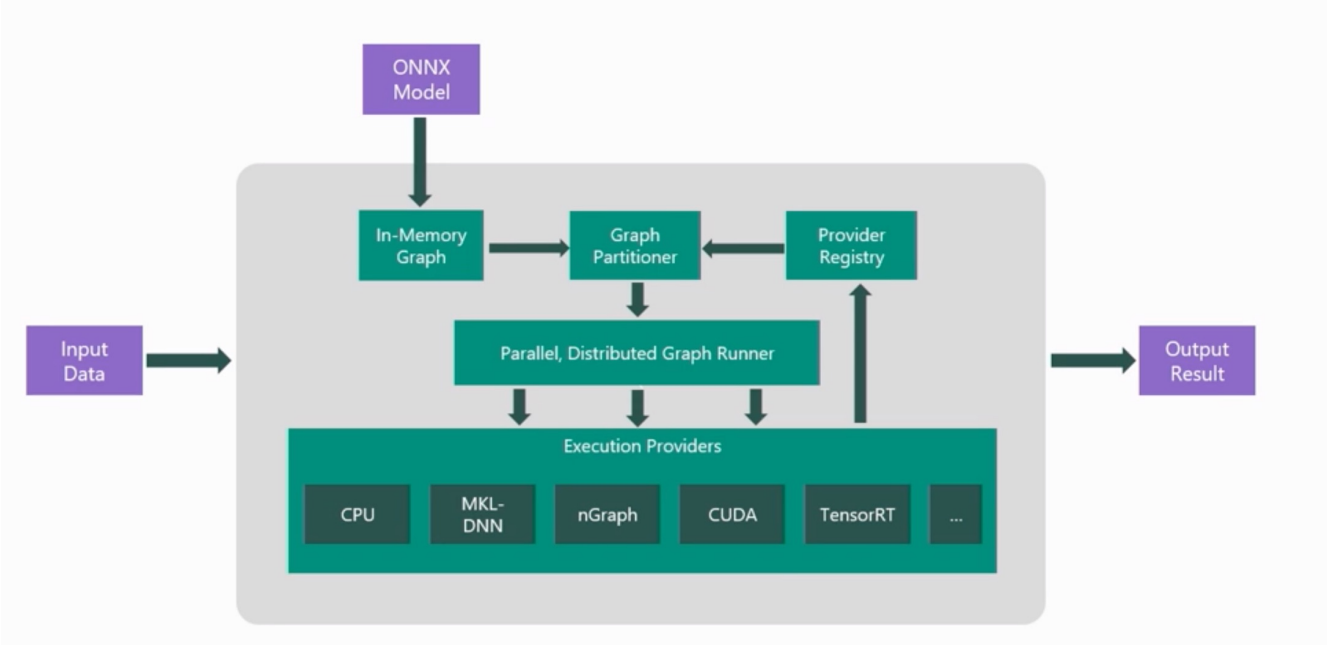
Deploy ONNX model using ONNX Runtime

ONNX Runtime

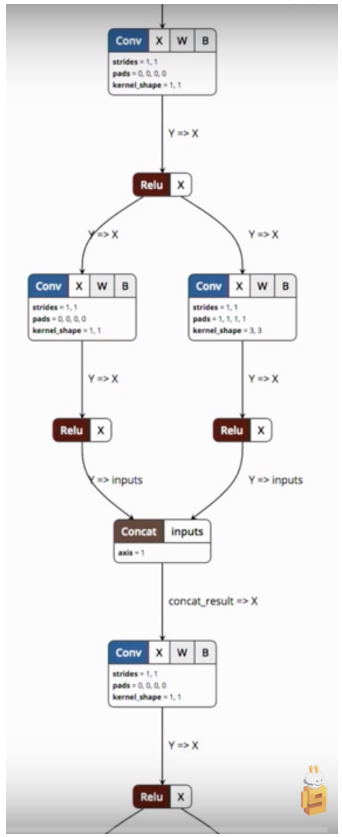


- ONNX Runtime is a cross-platform inference and training machine-learning accelerator.
- Available on Mac, Windows, Linux platforms
- Supports full ONNX-ML spec
- Open-sourced https://github.com/microsoft/onnxruntime

ONNX Runtime Architecture



Computational dataflow graph



ONNX Runtime has a graph parser which takes ONNX model, parses the graph, applies runtime optimizations like fusion ops, executes portions of the graph on specific hardware, provides the inference output result

Youtube Video: Open Neural Network Exchange (ONNX) in the enterprise: how Microsoft scales ML

Frameworks provide implementations of ONNX operators

Example

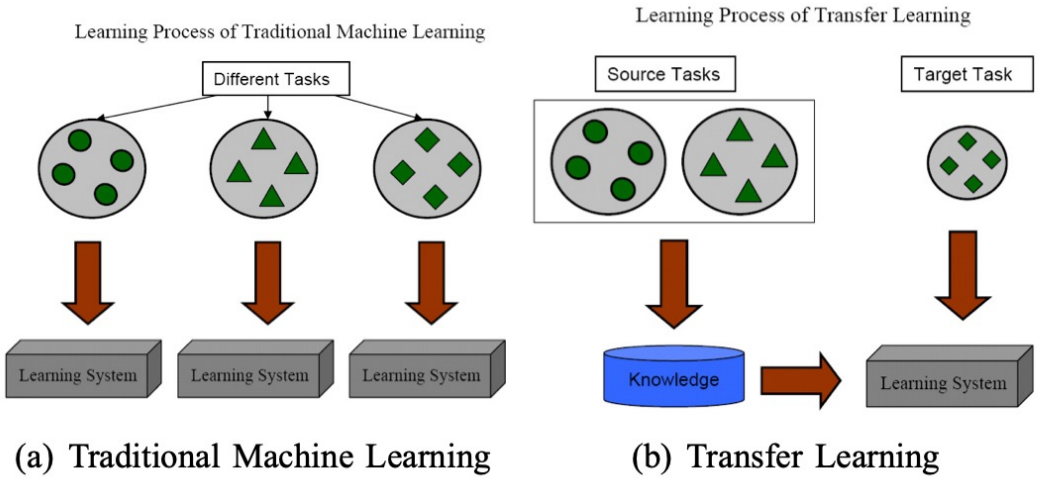
https://github.com/huggingface/notebooks/blob/main/examples/onnx-export.ipynb

- Export 🤗 Transformers Models to ONNX
- Leverage ONNX runtime for inference over an ONNX graph
- Benchmark Pytorch and ONNX model on CPU

Learning with limited data

- Building good machine learning models require lot of training data
  - To capture robust representation of unknown input distribution
- Small training jobs are common and labeled data is scarce in many domains
  - In commercial VR service (Bhatta et al. 2019), average number of images submitted is 250 and average number of classes are 5; ~50 images per class
- Can we leverage knowledge learnt from related tasks for target task ?

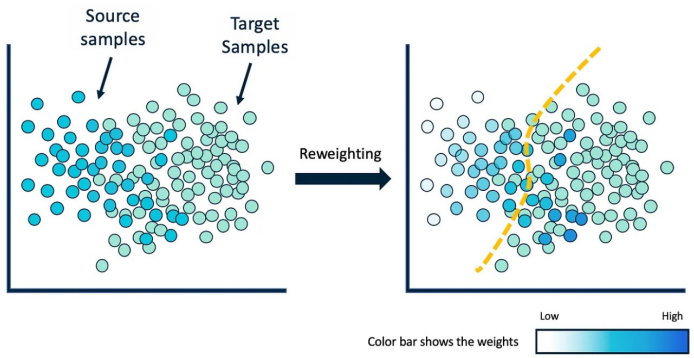
Transfer Learning



- Transfer learning is a class of techniques to reuse knowledge gathered from “source” tasks (with sufficiently rich set of labeled data) for a “target task (with few labeled data)

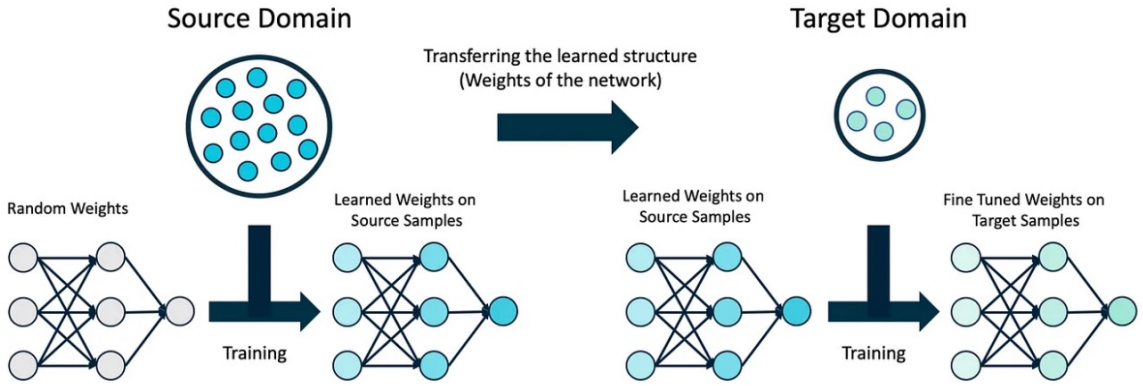
Inductive Transfer Learning Approaches

- *Common intuition*: Networks which have learned compact representations of a "source" task, can reuse these representations to achieve higher performance on a related "target" task.
- *Instance based* approaches attempt to identify appropriate data used in the source task to supplement target task training
  - Source and target domain data use exactly the same set of features and labels, but the distributions of the data in the two domains are different



Inductive Transfer Learning Approaches

- *Feature representation based* approaches attempt to leverage source task weight matrices
  - Trained weights in source network have captured a representation of the input that can be transferred by fine-tuning the weights or retraining the final dense layer of the network on the new task.



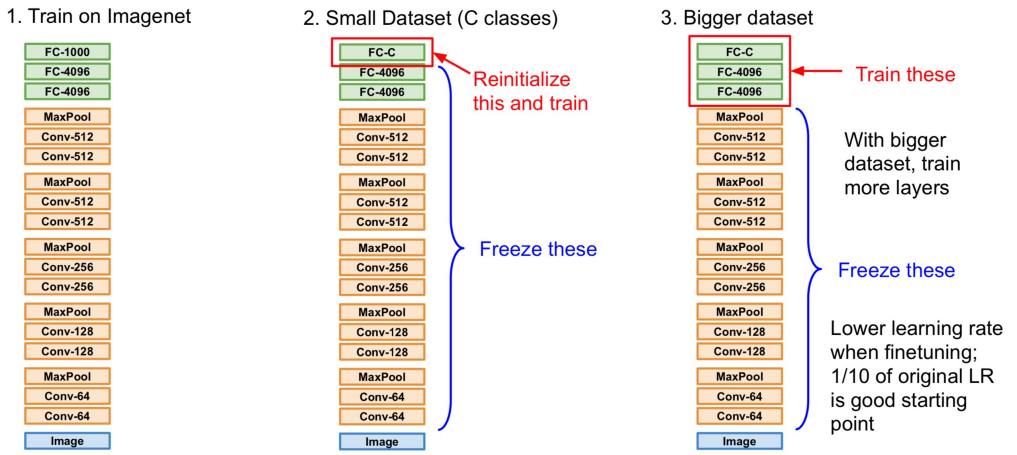
Learn to Transfer

- Which source model to use ?
  - Caffe Model Zoo has several pretrained models for different tasks
  - TensorFlow 2 Detection Model Zoo has several models pretrained on COCO 2017 dataset
- Which method to use ?
  - Shallow learning driving SVMs
  - Fine tuning
  - Single source or ensemble
- Which layers to freeze and finetune ?
  - ResNet101 has 101 layers
  - GoogLeNet has 22 layers
  - VGG16 has 16 layers
- What is the performance requirement?
  - Latency
  - Memory Footprint
  - Target domain accuracy
- What curriculum should I follow?
  - Imagenet1K (1000 classes ,~1.3M images,) -> Food101 (101 classes, 101000 images) -> Greek Food
  - Imagenet22K (22K classes ,~15M images,) -> Greek Food

Transfer Learning: Basic Finetuning

- Take almost any deep network pre-trained on a large dataset of your choice
- Model zoo of pretrained models
  - TensorFlow: https://github.com/tensorflow/models
  - PyTorch: https://github.com/pytorch/vision
- Replace the last (classification) layer with a randomly initialized one
- Train only the new layer's weights, using the "frozen" embedding

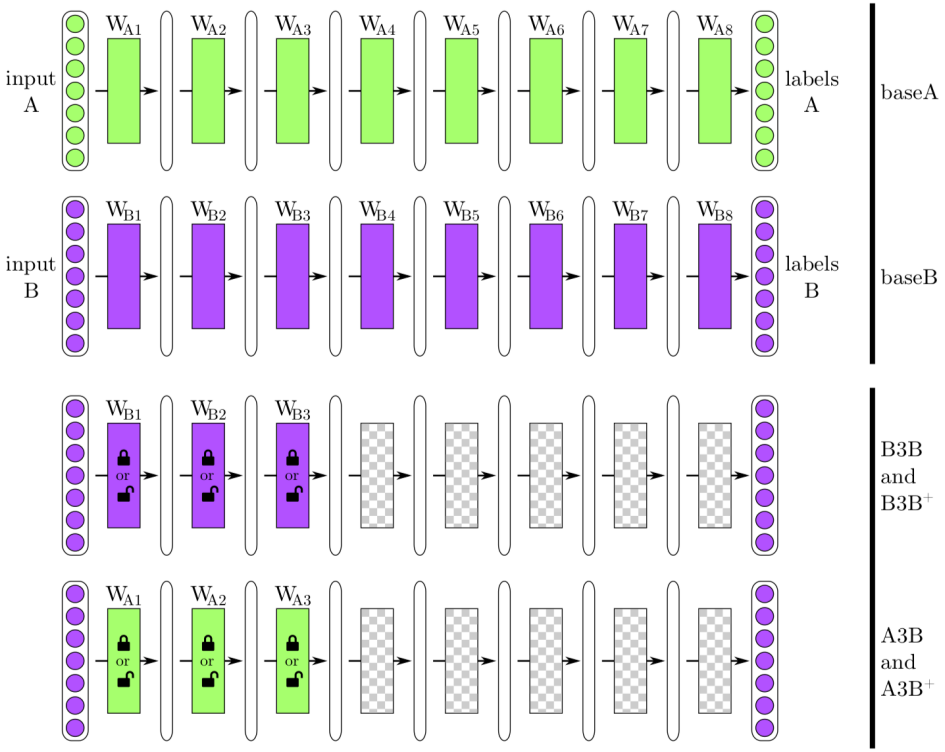
- This baseline method works well in many settings…
- …. Can we improve on it?



Improving Transferability in Transfer Learning

- **Selection of source model:** What is the best ”source” task/model to transfer knowledge for a “target” task ?
  - How to measure “similarity” between source and target task ?
    - Develop similarity measures between source and target datasets in feature space
  - Does size always matter ?
    - Will a source model trained on huge datasets will always outperform source models trained on smaller but more related datasets ?
  - Improper choice of a base dataset/model for a target could result in degraded performance compared to not using transfer learning (negative transfer)
- **Degree of finetuning**
  - Which layers to finetune and which to freeze ?
  - What should be the learning rate of layers to be finetuned ?
    - Higher learning rate ➔ loose information from source task

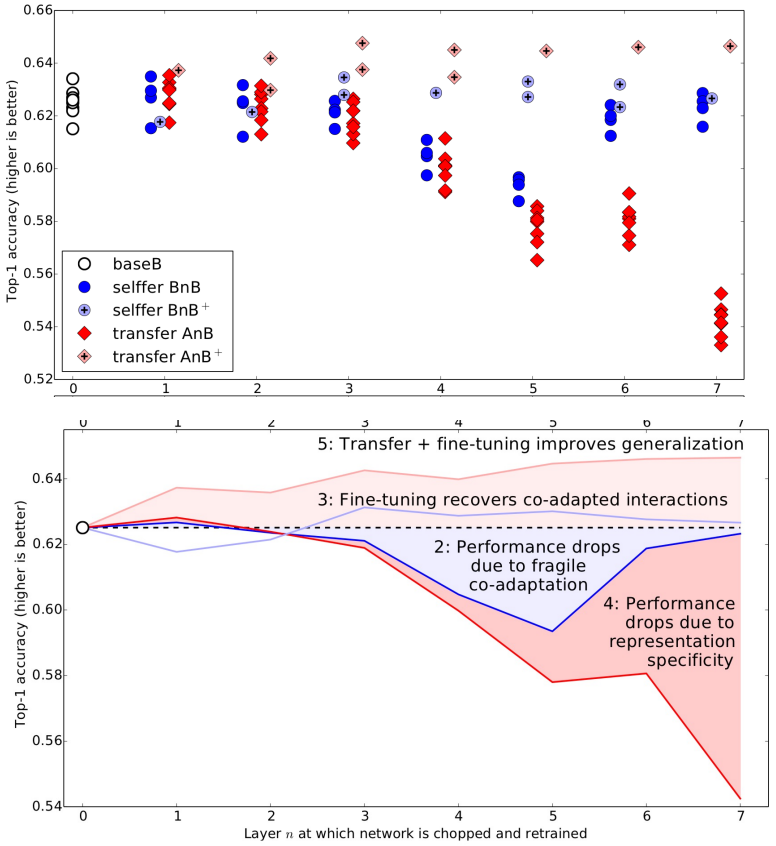
Transferability Experiments



generality versus specificity of neurons

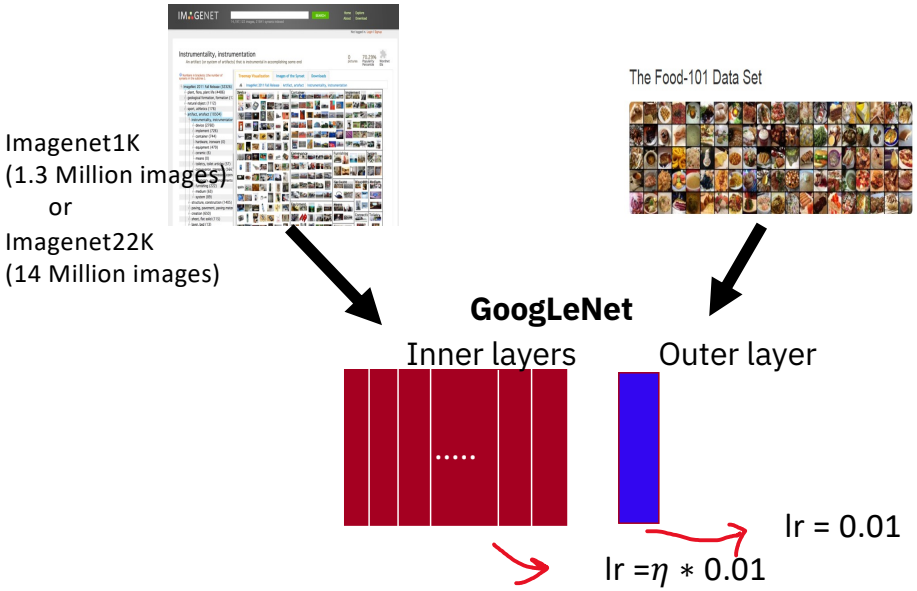
Transferability

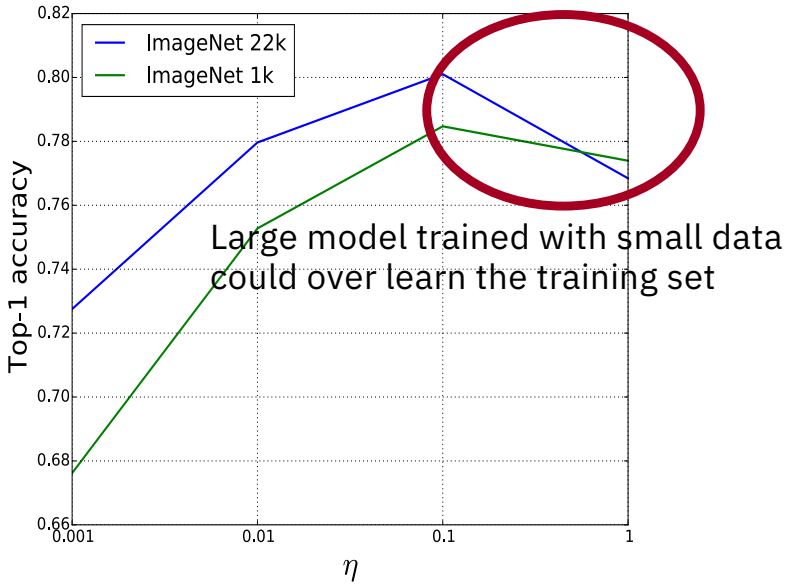
Each marker in the figure represents the average accuracy over the validation set for a trained network. The white circles above n = 0 represent the accuracy of baseB. There are eight points, because we tested on four separate random A/B splits. Each dark blue dot represents a BnB network. Light blue points represent BnB+ networks, or fine-tuned versions of BnB. Dark red diamonds are AnB networks, and light red diamonds are the fine-tuned AnB+ versions.



Lines connecting the means of each treatment.

Impact of base model data size on transfer learning





Fine-tuning results on Food-101 dataset with varying numbers of learning rate multipliers for layers with pre-trained weights

Bhattacharjee et al, "Distributed learning of deep feature embedding for visual recognition tasks", IBM J of R & D, 2017