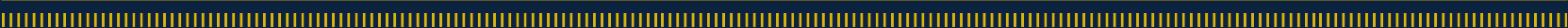




---

# Distributed Hyperparameter Optimization and Model Search with Examples using SHADHO

---

A horizontal line composed of many small gold dots, spanning the width of the slide.

Jeffery Kinnison, Sadaf Ghaffari, Nathaniel Blanchard,  
Walter Scheirer



# Overview

1. Problem Definition
2. Manual, Grid, and Random Search
3. Bayesian Optimization
4. Population-Based Methods
5. Additional Methods
6. Neural Architecture Search
7. Hyperparameter Importance

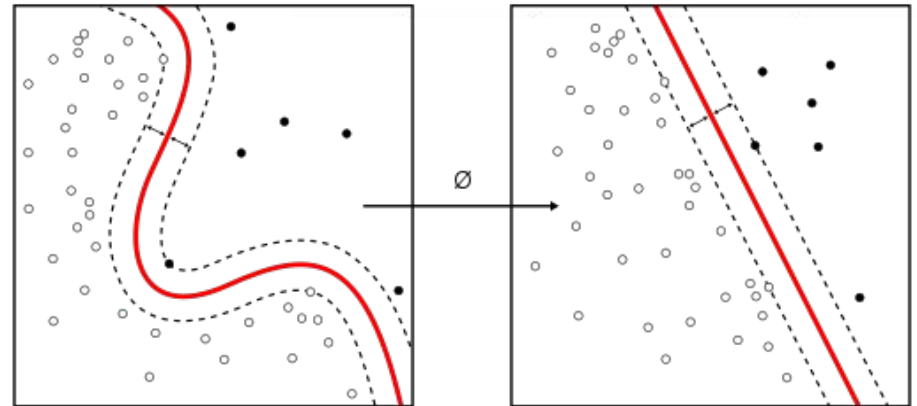
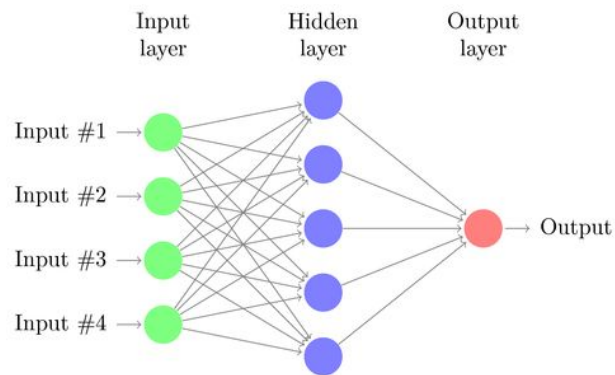
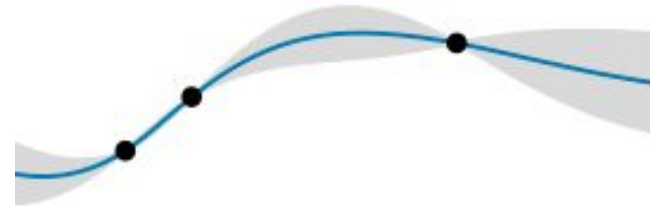
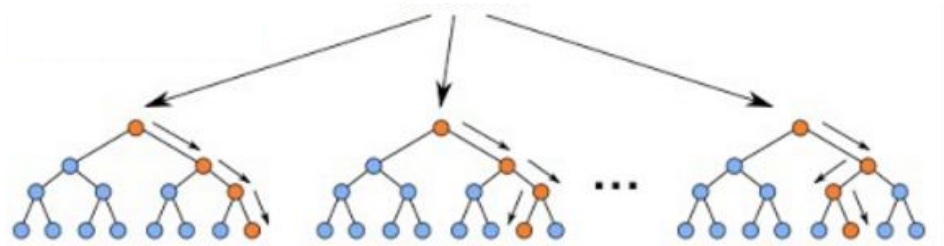
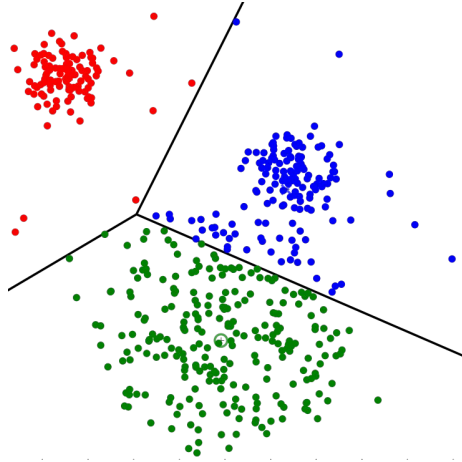
<https://jeffkinnison.github.io/shadho-tutorial/>



# 1. Problem Definition

<https://jeffkinnison.github.io/shadho-tutorial/>

# Learning Algorithms

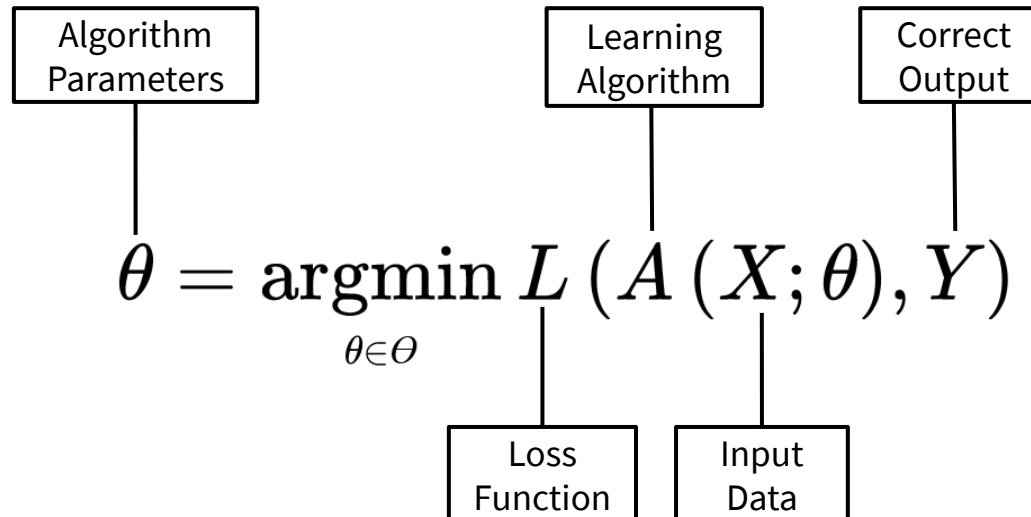


# Some Properties of Learning Algorithms

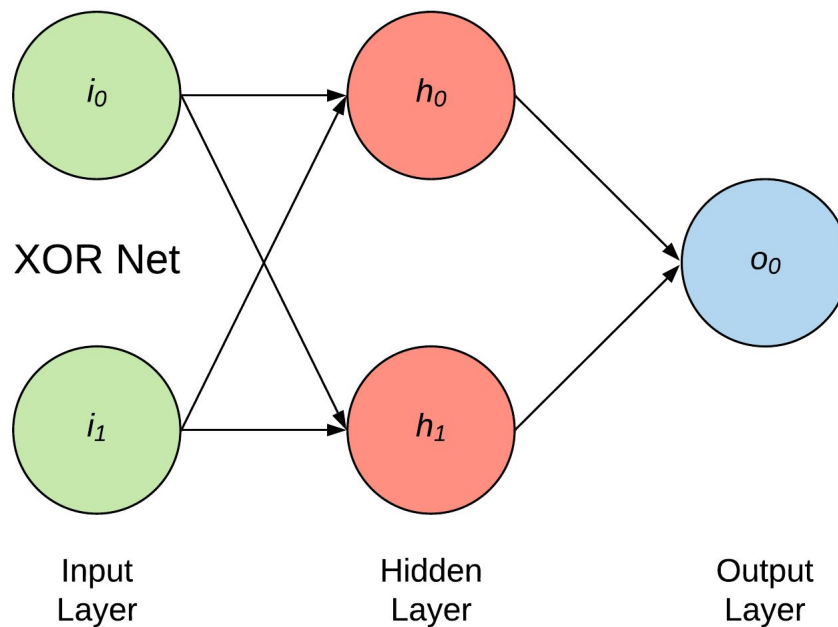
- Adjustable parameters (internal state)
- Updates to parameters in response to data
- Hyperparameters that control updates

# Training Learning Algorithms

Optimize an instance of an algorithm (model) to predict values from data.

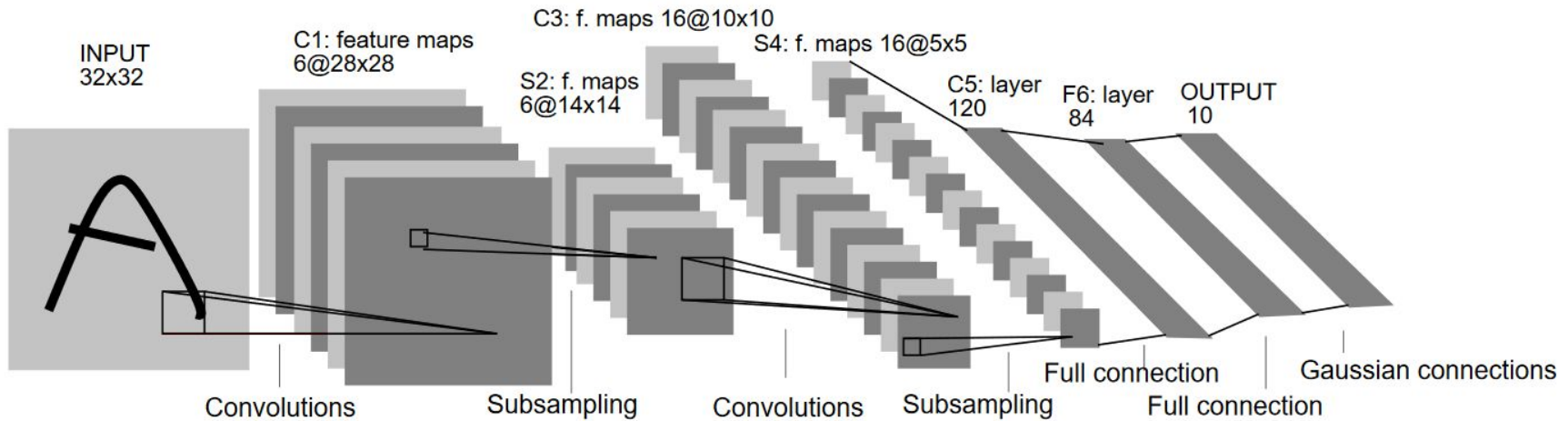


# Training Learning Algorithms



$$|\theta| = 9$$

# Training Learning Algorithms

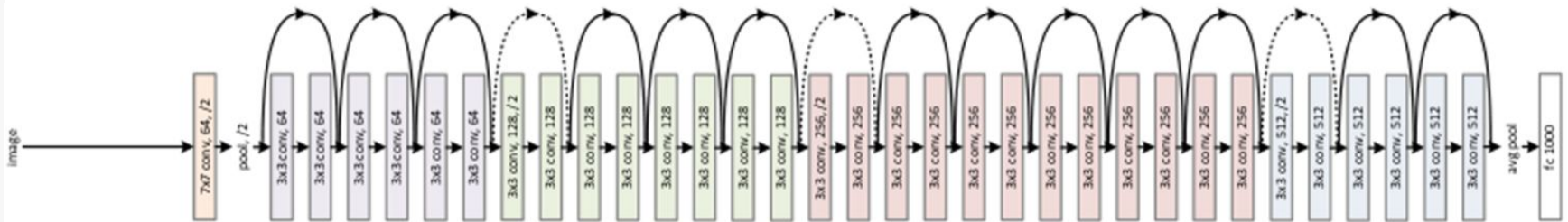


$$|\theta| = 340,908$$



# Training Learning Algorithms

## ResNet-34



$$|\theta| = 3.6 \times 10^9$$

# Some Properties of Learning Algorithms

- Adjustable parameters (internal state)
- Updates to parameters in response to data
- Hyperparameters that control updates

# Hyperparameters

Tunable parameters that control the training process and model quality.

## K-Means Clustering

- Number of Clusters

## Naive Bayes

- Prior Distribution
- Smoothing Factor

## SVM

- Kernel Function
- Regularization parameter (C)
- Kernel Parameters

## Random Forests

- Ensemble Size
- Split Criterion
- Maximum Depth
- Splitting Threshold

## Neural Nets

- Number of layers
- Size of layers
- Activations
- Regularization
- Dropout/Batch Norm
- Optimizer
- Learning Rate
- Momentum
- Learning Rate Decay
- ...

# Hyperparameter Domains

## **Continuous Domains**

Values are drawn from a continuous probability distribution.

## **Categorical Domains**

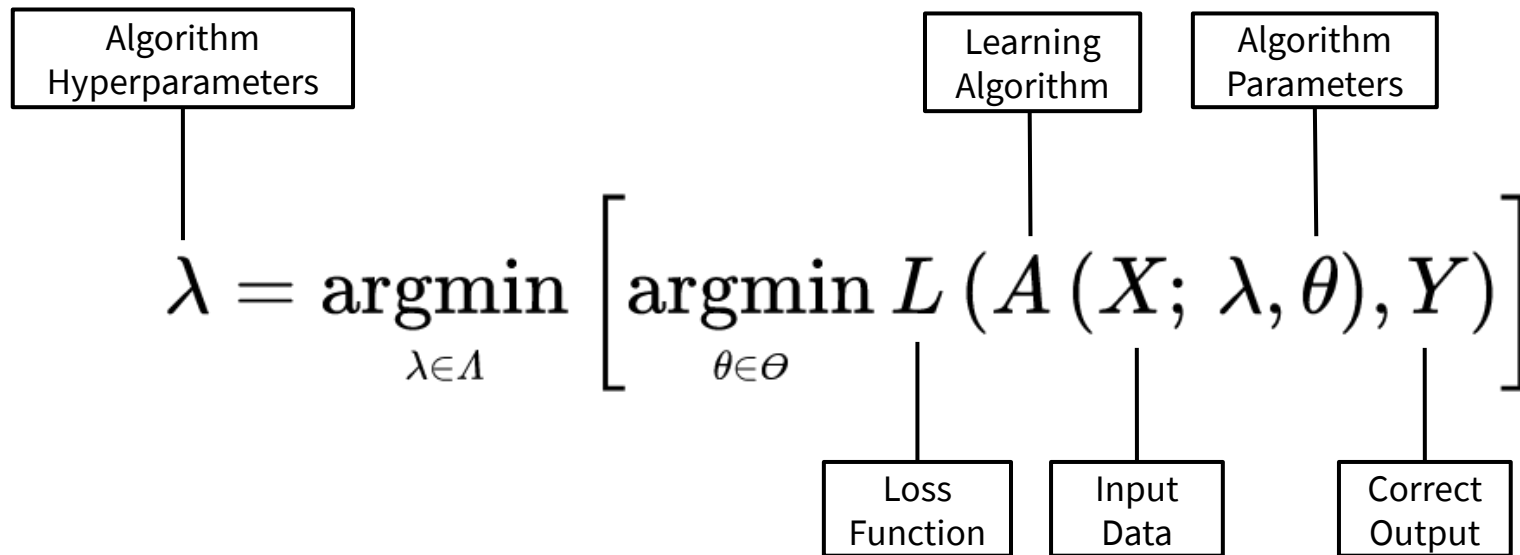
Values selected from a set with some probability.

## **Dependent Domains**

Values selected relative to another hyperparameter.

# Hyperparameter Optimization

Select hyperparameters that result in the best performance for a given learning algorithm and data.



# Hyperparameter Optimization Objective

## **Training Performance**

e.g., loss, accuracy, precision/recall on the training set

## **Generalization Performance**

e.g., loss, accuracy, precision/recall on the validation set

## **Learning Speed/Adaptability**

e.g., rate of convergence, time to learn a new class

## **Model Matching**

e.g., student/teacher models, biological model similarity



## 2. Manual, Grid, and Random Search

<https://jeffkinnison.github.io/shadho-tutorial/>

# Manual Search

A human selects hyperparameters, evaluates trained model(s), and decides how to adjust.

## Pros

- Extremely easy to implement
- Domain knowledge is easily baked in
- Decision-making is transparent

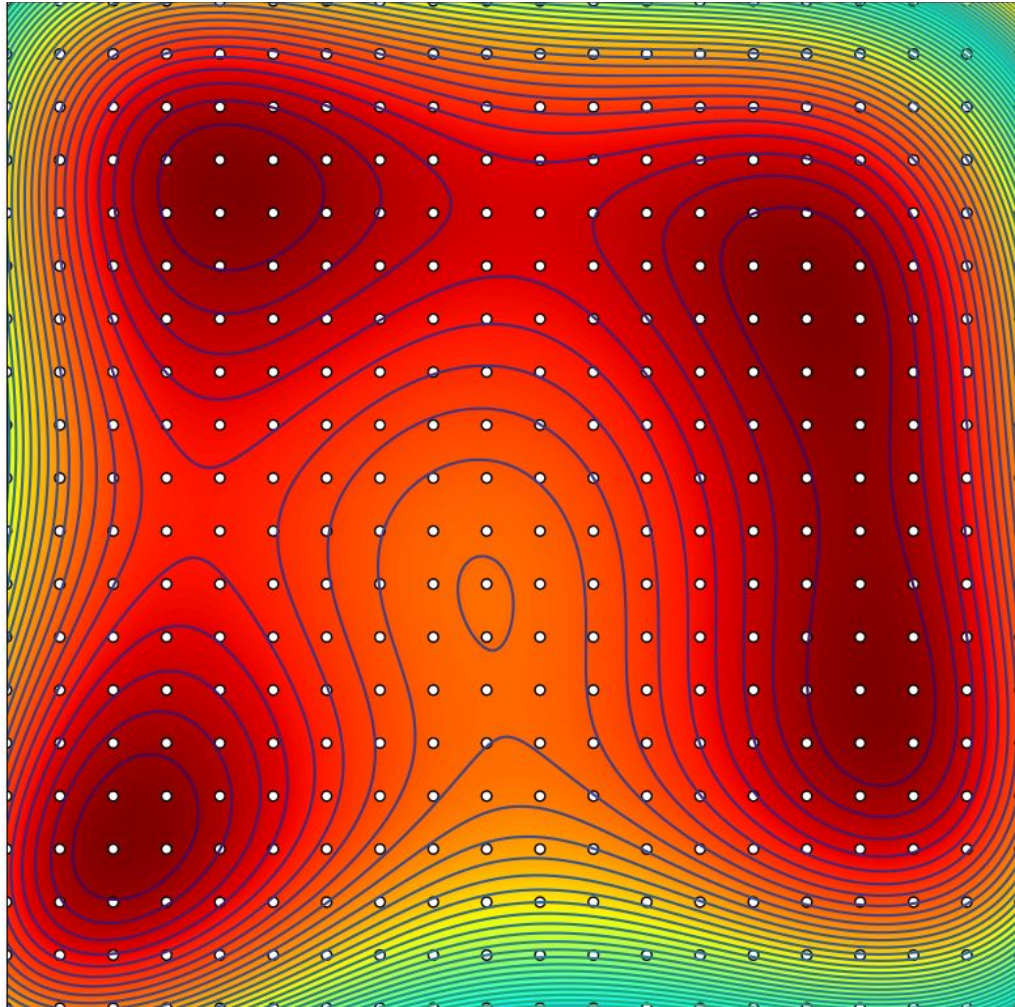
## Cons

- Human in the loop
- Limited in scope/scalability
- Tuning becomes difficult with more complex learning methods



# Grid Search

Hyperparameter sets are selected from a regular grid and evaluated.



# Grid Search

## Pros

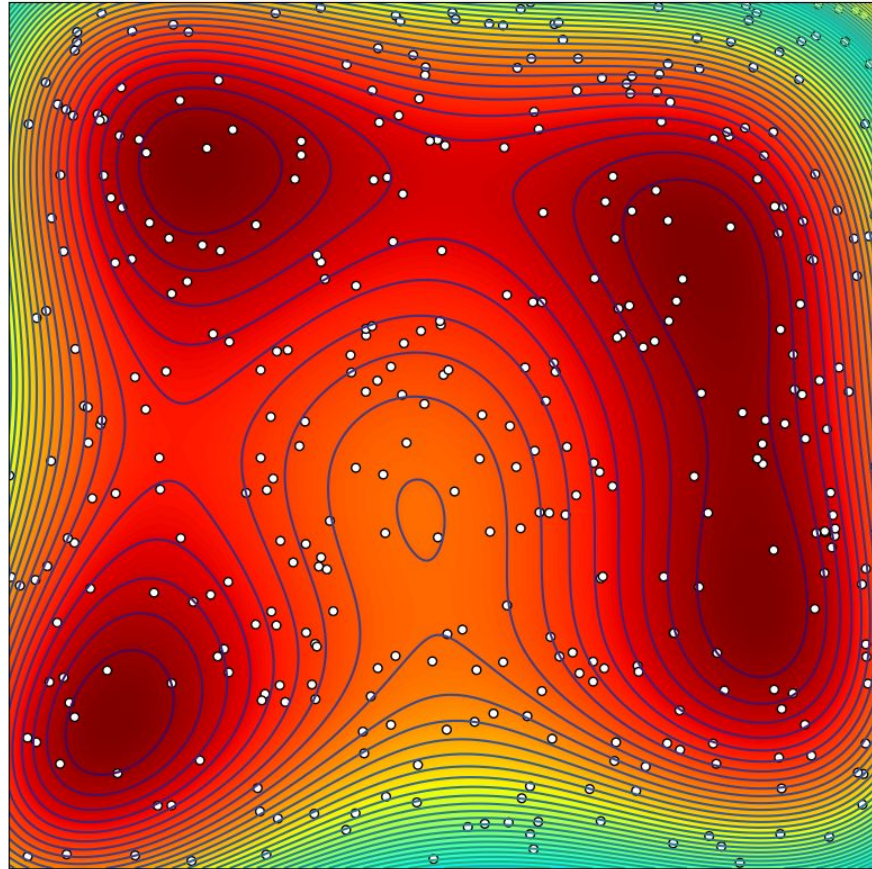
- Easy to implement and automate
- Domain knowledge is easily baked in
- Parallelizable over grid points
- Will not get stuck in local minima
- Can be completely searched

## Cons

- Limited in scope
- Inbuilt holes in the search space
- Not sensitive to the shape of the objective

# Random Search

Hyperparameters are defined over continuous/categorical domains and selected randomly.



# Random Search

## Continuous Domains

Typically uniform or normal distributions.

$$\text{ex. } \lambda \sim \mathcal{U}(-7, 43), \lambda \sim \mathcal{N}(0, 1)$$

## Categorical Domains

Typically uniform (e.g., randint).

$$\text{ex. } \lambda \sim \{32, 64, 128, 256, 512\}$$

# Random Search

## Pros

- Easy to implement and automate
- Domain knowledge can be baked in
- Parallelizable over sampled hyperparameters
- No holes in the search domain
- Optimal in the limit

## Cons

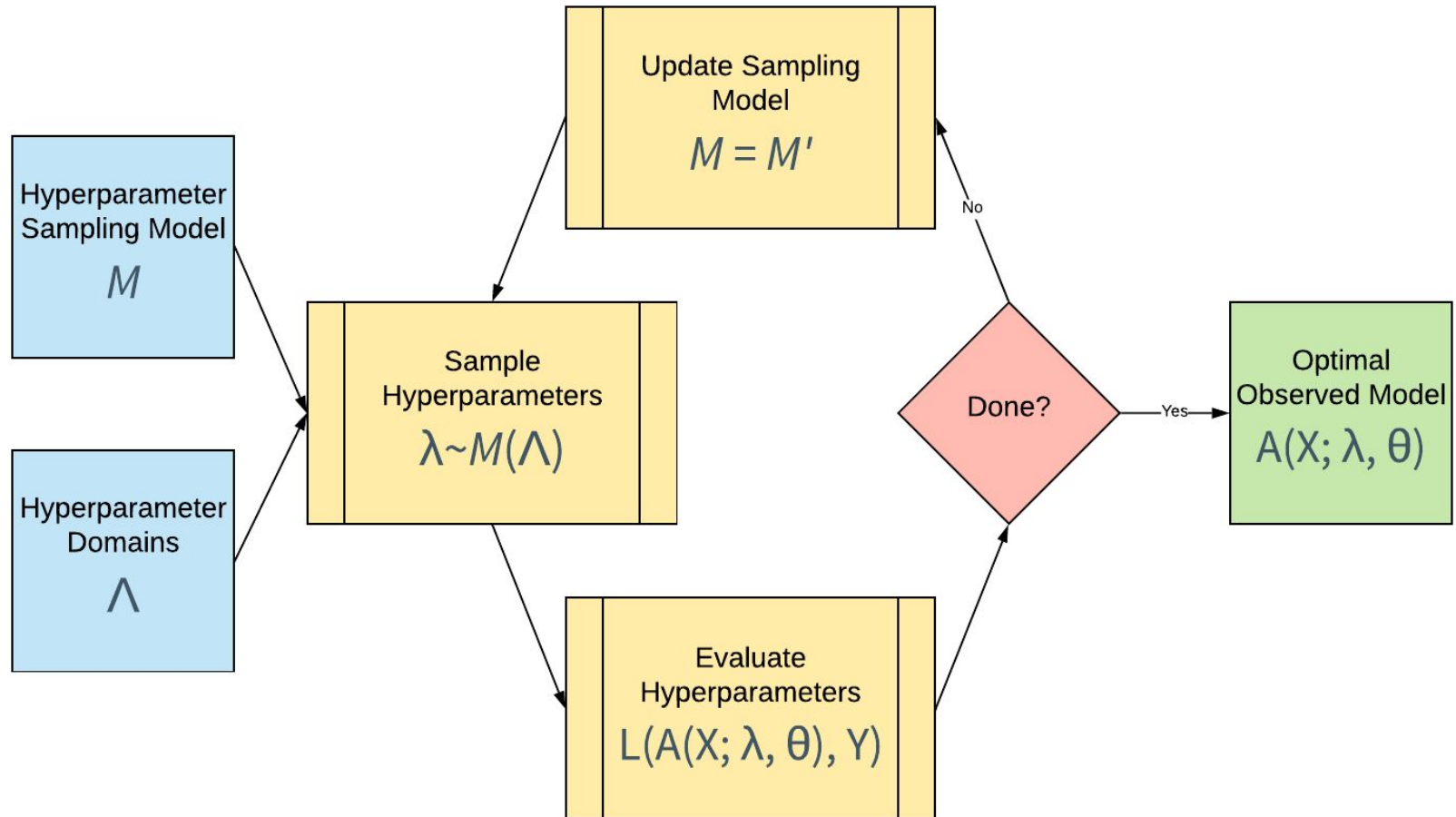
- No defined “end”
- No guidance to a minimum
- “The limit” is the heat death of the universe



# 3. Bayesian Optimization

<https://jeffkinnison.github.io/shadho-tutorial/>

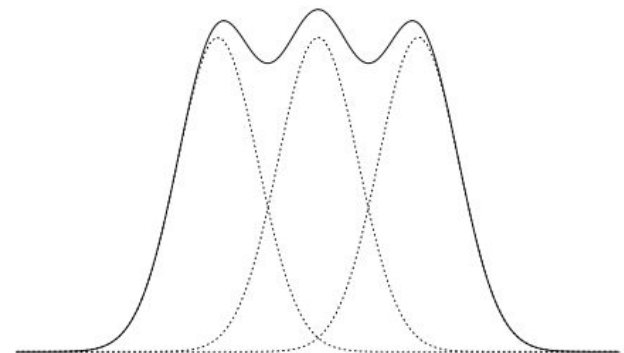
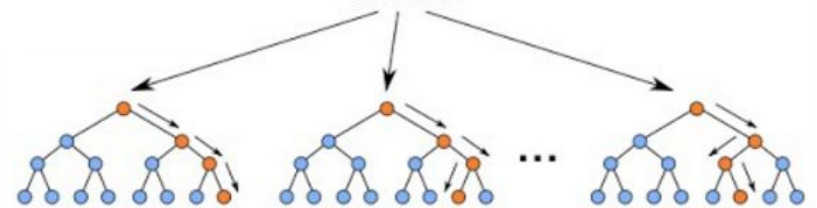
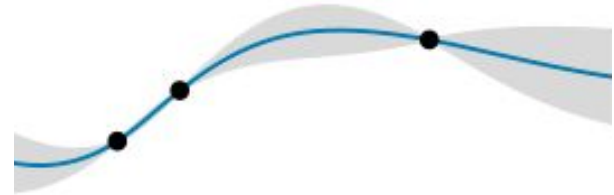
# Sequential Model-Based Optimization





# Sequential Model-Based Optimization

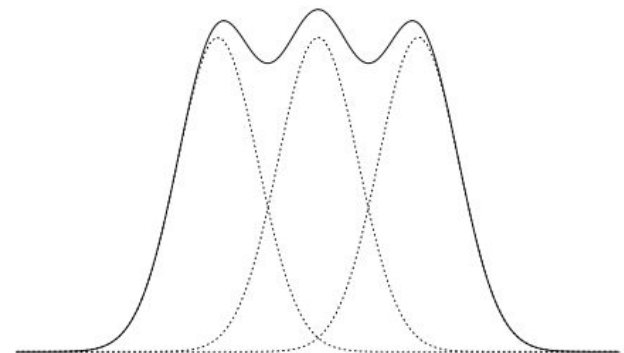
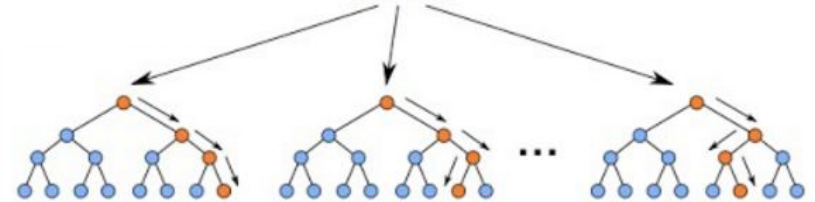
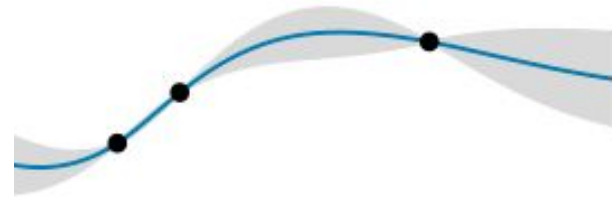
1. Initialize a model mapping parameters to the objective





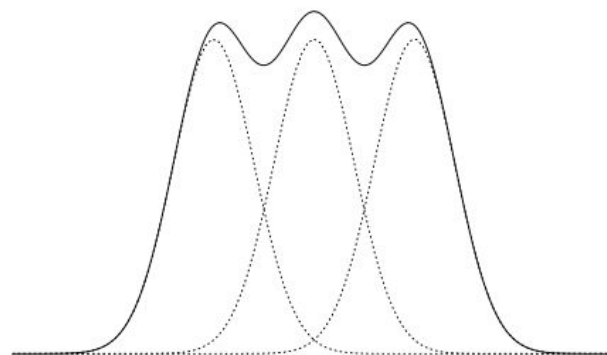
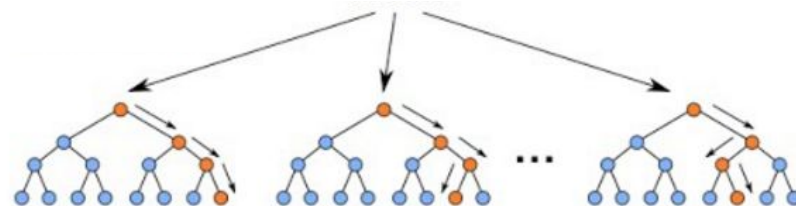
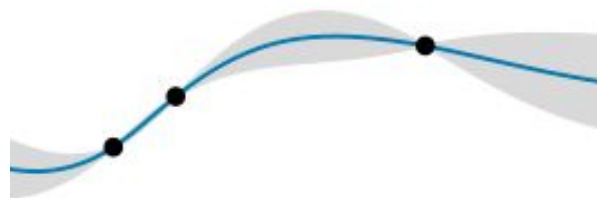
# Sequential Model-Based Optimization

1. Initialize a model mapping hyperparameters to the objective
2. Sample hyperparameters from the model



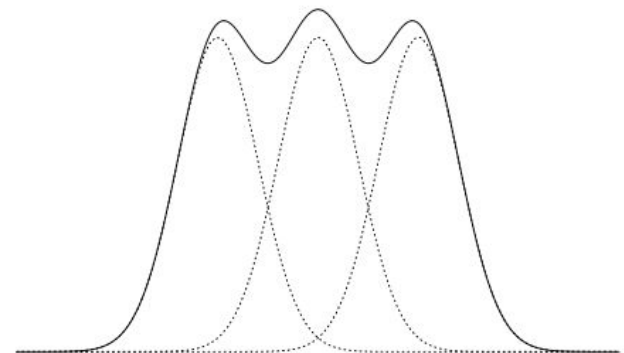
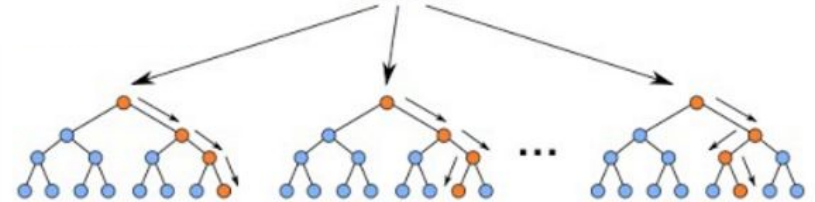
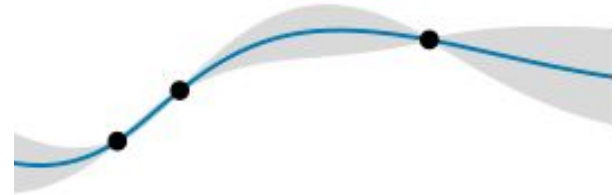
# Sequential Model-Based Optimization

1. Initialize a model mapping hyperparameters to the objective
2. Sample hyperparameters from the model
3. Evaluate the objective using the hyperparameters



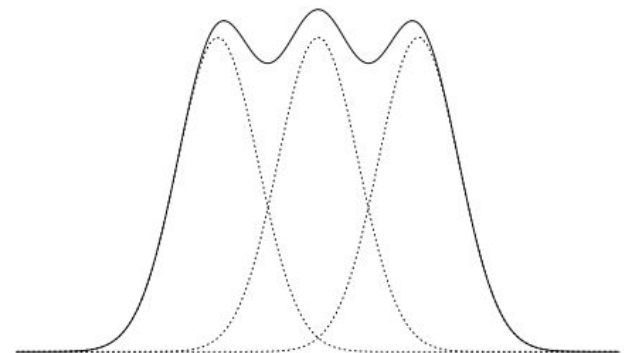
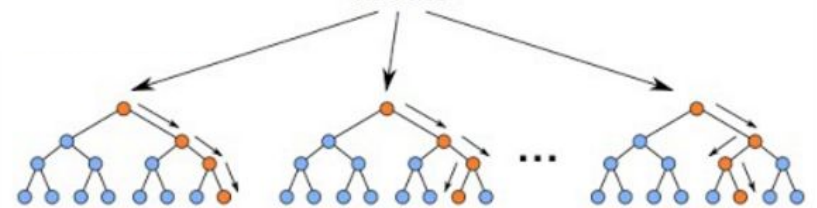
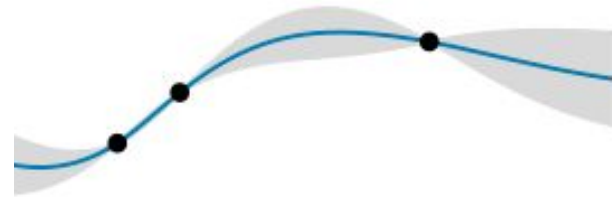
# Sequential Model-Based Optimization

1. Initialize a model mapping hyperparameters to the objective
2. Sample hyperparameters from the prior distribution
3. Evaluate the objective using the hyperparameters
4. Update the model



# Sequential Model-Based Optimization

1. Initialize a model mapping hyperparameters to the objective
2. Sample hyperparameters from the prior distribution
3. Evaluate the objective using the hyperparameters
4. Update the model
5. Repeat



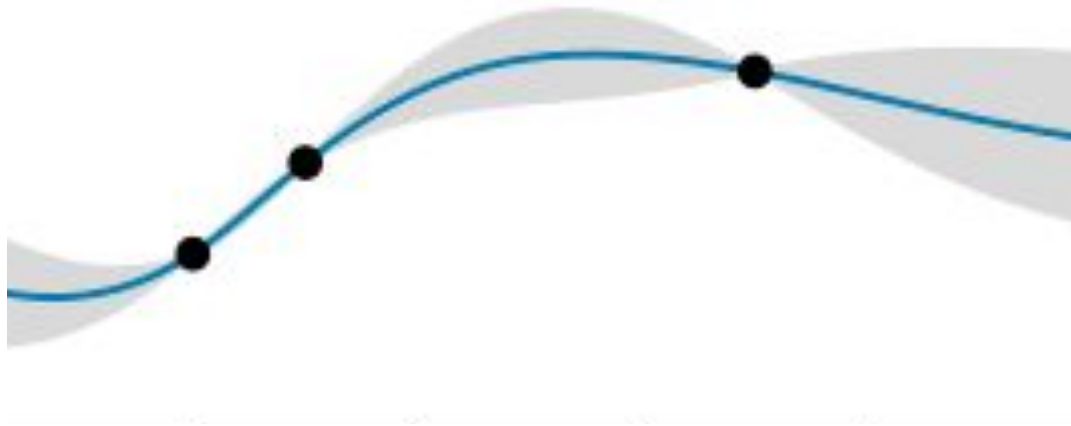
# Guiding the Search

Bayesian Optimization is guided by evaluating hyperparameters that are predicted to improve model performance.

$$EI = \int_{-\infty}^{y_{best}} (y_{best} - y) \underbrace{p(y|\lambda, \Lambda_{observed})}_{\text{Probability of Generating Loss}} dy$$

# Gaussian Process Regression

Model the objective with a Gaussian Process and score potential hyperparameters by Expected Improvement.



# Gaussian Process Regression

Model the objective with a Gaussian Process and score potential hyperparameters by Expected Improvement.

1. Collect baseline hyperparameter evaluations
2. Fit a Gaussian Process to approximate the objective
3. Generate hyperparameters and compute EI
4. Evaluate hyperparameters with best expected improvement
5. Repeat 2-4

# Gaussian Process Regression

Model the objective with a Gaussian Process and score potential hyperparameters by Expected Improvement.

$\mu$  = GPR predictive mean

$\sigma$  = GPR predictive variance

$$\gamma = \frac{y_{best} - \mu}{\sigma}$$

$$EI = \sigma \gamma \underbrace{\Phi(\gamma)} + \underbrace{\mathcal{N}(\gamma; 0, 1)}$$

Standard Normal  
CDF

Standard Normal  
PDF



# Gaussian Process Regression

## Pros

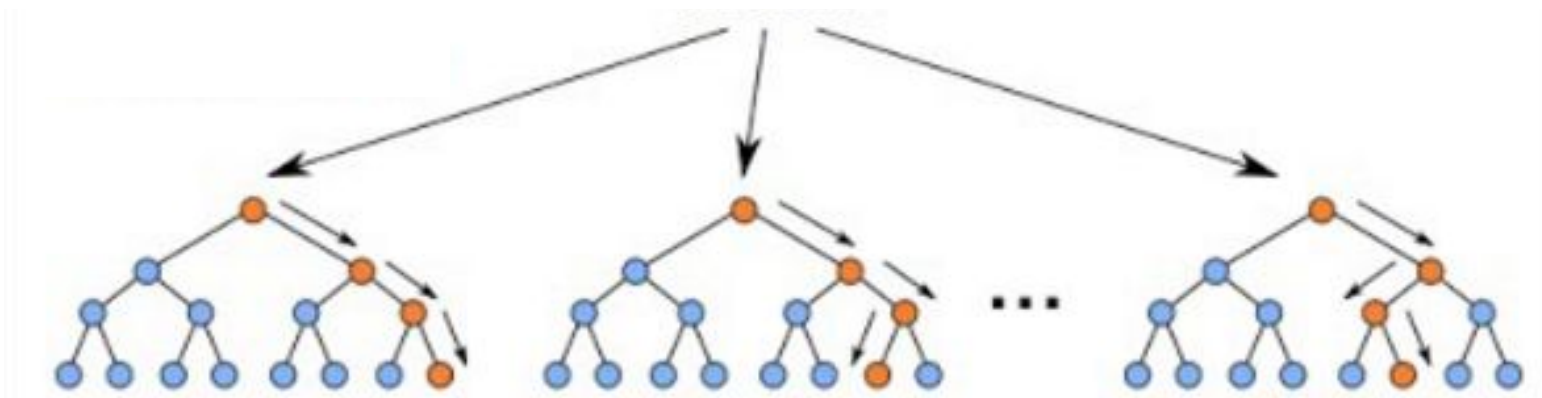
- Sequentially guides the search towards optimal hyperparameters
- Search spaces can be continuous
- Incorporates all prior knowledge into GP

## Cons

- Difficult to account for hyperparameter value dependencies
- Can become stuck in local minima
- Heavily dependent on prior
- Gaussian Process hyperparameters must be tuned
- Care must be taken to parallelize searches

# Random Forest Regression

Model the objective with a Random Forest and score potential hyperparameters by Expected Improvement.



# Random Forest Regression

Model the objective with a Random Forest and score potential hyperparameters by Expected Improvement.

1. Collect baseline hyperparameter evaluations
2. Fit/update a Random Forest to approximate the objective
3. Generate hyperparameters and compute EI
4. Evaluate hyperparameters with best expected improvement
5. Repeat 2-4

# Random Forest Regression

Model the objective with a Random Forest and score potential hyperparameters by Expected Improvement.

$\mu$  = mean predicted score over ensemble

$\sigma$  = predicted score variance over ensemble

$$\gamma = \frac{y_{best} - \mu}{\sigma}$$

$$EI = \sigma \gamma \underbrace{\Phi(\gamma)} + \underbrace{\mathcal{N}(\gamma; 0, 1)}$$

Standard Normal  
CDF

Standard Normal  
PDF

# Random Forests Regression

## Pros

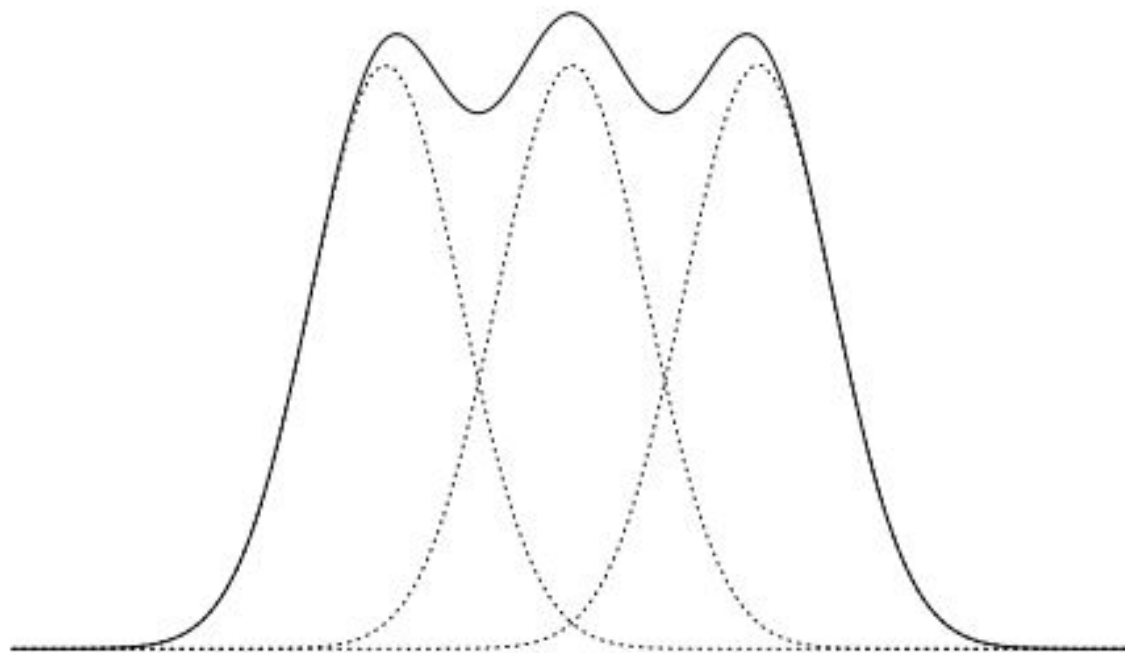
- Sequentially guides the search towards optimal hyperparameters
- Handles continuous domains
- Handles conditional domains
- Inbuilt ensemble learning

## Cons

- Can get stuck in local minima
- Heavily dependent on prior
- Random forest hyperparameters must be tuned
- Care must be taken to parallelize searches

# Tree-Structured Parzen Estimators

Model the top  $k$  and bottom  $N - k$  hyperparameters in separate Gaussian mixture models and update to separate the two models.



# Tree-Structured Parzen Estimators

Model the top  $k$  and bottom  $N - k$  hyperparameters in separate Gaussian mixture models and update to separate the two models.

1. Collect baseline hyperparameter evaluations
2. Fit a Gaussian Process  $l$  to the top  $k$  performing hyperparameters
3. Fit a Gaussian Process  $g$  to the remaining  $N - k$  hyperparameters
4. Generate hyperparameters and compute EI
5. Evaluate hyperparameters with best expected improvement
6. Repeat 2-4

# Tree-Structured Parzen Estimators

Model the top  $k$  and bottom  $N - k$  hyperparameters in separate Gaussian mixture models and update to separate the two models.

$l(x)$  =  $l$  GMM predicted score

$g(x)$  =  $g$  GMM predicted score

$\gamma = p(y < y^*)$

$EI = (\gamma + \frac{g(x)}{l(x)} (1 - \gamma))^{-1}$



# Tree-Structured Parzen Estimators

Model the top  $k$  and bottom  $N - k$  hyperparameters in separate Gaussian mixture models and update to separate the two models.

	<b>convex</b>	<b>MRBI</b>
TPE	<b>14.13</b> $\pm 0.30$ %	<b>44.55</b> $\pm 0.44$ %
GP	16.70 $\pm 0.32$ %	47.08 $\pm 0.44$ %
Manual	18.63 $\pm 0.34$ %	47.39 $\pm 0.44$ %
Random	18.97 $\pm 0.34$ %	50.52 $\pm 0.44$ %

Automatic methods were allotted 200 trials; manual search used 82 trials for convex and 27 trials for MRBI.

# Tree-Structured Parzen Estimators

Model the top  $k$  and bottom  $N - k$  hyperparameters in separate Gaussian mixture models and update to separate the two models.

Dataset	10-Fold C.V. Performance (%)					Test Performance (%)				
	EX-DEF	GRID SEARCH	RAND. SEARCH	AUTO-WEKA		EX-DEF	GRID SEARCH	RAND. SEARCH	AUTO-WEKA	
				TPE	SMAC				TPE	SMAC
DEXTER	10.20	<b>5.07</b>	10.60	9.83	5.66	8.89	<b>5.00</b>	9.18	8.89	7.49
GERMANCREDIT	22.45	20.20	20.15	21.26	<b>17.87</b>	27.33	<b>26.67</b>	29.03	27.54	28.24
DOROTHEA	6.03	6.73	8.11	6.81	<b>5.62</b>	6.96	5.80	<b>5.22</b>	6.15	6.21
YEAST	39.43	39.71	38.74	<b>35.01</b>	35.51	40.45	42.47	43.15	<b>40.10</b>	40.67
AMAZON	43.94	<b>36.88</b>	59.85	50.26	47.34	28.44	<b>20.00</b>	41.11	36.59	33.99
SECOM	6.25	6.12	5.24	6.21	<b>5.24</b>	8.09	8.09	8.03	8.10	<b>8.01</b>
SEMEION	6.52	4.86	6.06	6.76	<b>4.78</b>	8.18	6.29	6.10	8.26	<b>5.08</b>
CAR	2.71	0.83	<b>0.53</b>	0.91	0.61	0.77	0.97	<b>0.01</b>	0.18	0.40
MADELON	25.98	26.46	27.95	24.25	<b>20.70</b>	21.38	21.15	24.29	21.56	<b>21.12</b>
KR-VS-KP	0.89	0.64	0.63	0.43	<b>0.30</b>	0.31	1.15	0.58	0.54	<b>0.31</b>
ABALONE	73.33	72.15	72.03	72.14	<b>71.71</b>	73.18	73.42	74.88	<b>72.94</b>	73.51
WINE QUALITY	38.94	35.23	35.36	35.98	<b>34.65</b>	37.51	34.06	34.41	<b>33.56</b>	33.95
WAVEFORM	12.73	12.45	12.43	12.55	<b>11.92</b>	14.40	14.66	14.27	<b>14.23</b>	14.42
GISSETTE	3.62	2.59	4.84	3.55	<b>2.43</b>	2.81	2.40	4.62	3.94	<b>2.24</b>
CONVEX	28.68	<b>22.36</b>	33.31	28.56	25.93	25.96	23.45	31.20	25.59	<b>23.17</b>
CIFAR-10-SMALL	66.59	<b>53.64</b>	67.33	58.41	58.84	65.91	56.94	66.12	57.01	<b>56.87</b>
MNIST BASIC	5.12	<b>2.51</b>	5.05	10.02	3.75	5.19	<b>2.64</b>	5.05	12.28	3.64
ROT. MNIST + BI	66.15	<b>56.01</b>	68.62	73.09	57.86	63.14	57.59	66.40	70.20	<b>57.04</b>
SHUTTLE	0.0328	0.0361	0.0345	0.0251	<b>0.0224</b>	0.0138	0.0414	0.0157	0.0145	<b>0.0130</b>
KDD09-APPENTENCY	1.8776	1.8735	1.7510	1.8776	<b>1.7038</b>	1.7405	1.7400	1.7400	1.7381	<b>1.7358</b>
CIFAR-10	65.54	<b>54.04</b>	69.46	67.73	62.36	64.27	63.13	69.72	66.01	<b>61.15</b>

# Tree-Structured Parzen Estimators

## Pros

- Sequentially guides the search towards optimal hyperparameters
- Handles continuous hyperparameters
- Handles conditional hyperparameters
- Maximizes difference between majority and best (tail modeling)
- Simpler EI computation

## Cons

- Can get stuck in local minima
- Heavily dependent on prior
- Care must be taken to parallelize searches

# Additional Related Methods

- Bayesian Optimization using Deep Learning

- Snoek, Jasper, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Mostofa Patwary, Mr Prabhat, and Ryan Adams. "Scalable bayesian optimization using deep neural networks." In *International conference on machine learning*, pp. 2171-2180. 2015.

- Bayesian Optimization using Gaussian Process Bandits

- Kandasamy, Kirthevasan, Jeff Schneider, and Barnabás Póczos. "High dimensional Bayesian optimisation and bandits via additive models." In *International Conference on Machine Learning*, pp. 295-304. 2015.

- Bayesian Optimization using Bayesian Neural Networks

- Springenberg, Jost Tobias, Aaron Klein, Stefan Falkner, and Frank Hutter. "Bayesian optimization with robust Bayesian neural networks." In *Advances in neural information processing systems*, pp. 4134-4142. 2016.

- Bayesian Optimization using Boosted Decision Trees

- Xia, Yufei, Chuanzhe Liu, YuYing Li, and Nana Liu. "A boosted decision tree approach using Bayesian hyper-parameter optimization for credit scoring." *Expert Systems with Applications* 78 (2017): 225-241.

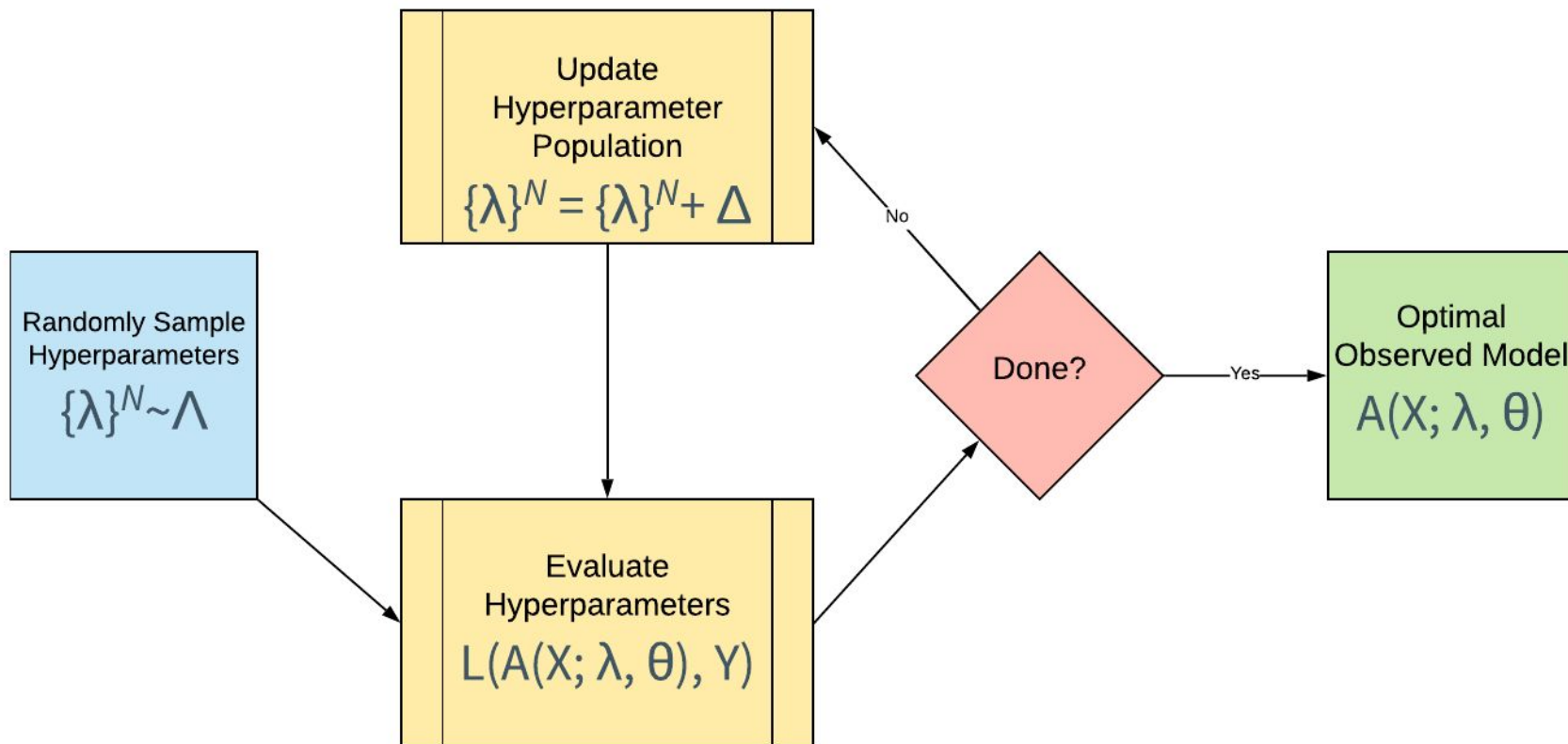
# Additional Related Methods

- Initializing Bayesian Optimization with Meta Learning
  - Feurer, Matthias, Jost Tobias Springenberg, and Frank Hutter. "Initializing bayesian hyperparameter optimization via meta-learning." In *Twenty-Ninth AAAI Conference on Artificial Intelligence*. 2015.
- Batched Bayesian Optimization
  - González, Javier, Zhenwen Dai, Philipp Hennig, and Neil Lawrence. "Batch bayesian optimization via local penalization." In *Artificial intelligence and statistics*, pp. 648-657. 2016.
- Bayesian Optimal Stopping
  - Dai, Zhongxiang, Haibin Yu, Bryan Kian Hsiang Low, and Patrick Jaillet. "Bayesian optimization meets Bayesian optimal stopping." In *International Conference on Machine Learning*, pp. 1496-1506. 2019.
- Bayesian Optimization with Adaptive Regressor Hyperparameters
  - Berkenkamp, Felix, Angela P. Schoellig, and Andreas Krause. "No-Regret Bayesian optimization with unknown hyperparameters." *arXiv preprint arXiv:1901.03357* (2019).

# 4. Population-Based Methods

<https://jeffkinnison.github.io/shadho-tutorial/>

# Population-Based Optimization



# Population-Based Optimization

1. Randomly initialize  $n$  hyperparameter sets
2. Evaluate all  $n$  hyperparameter sets
3. Rank the population by their “fitness”
4. Update each hyperparameter set with some criterion
5. Repeat 2-3 until hyperparameter sets and/or performance converge



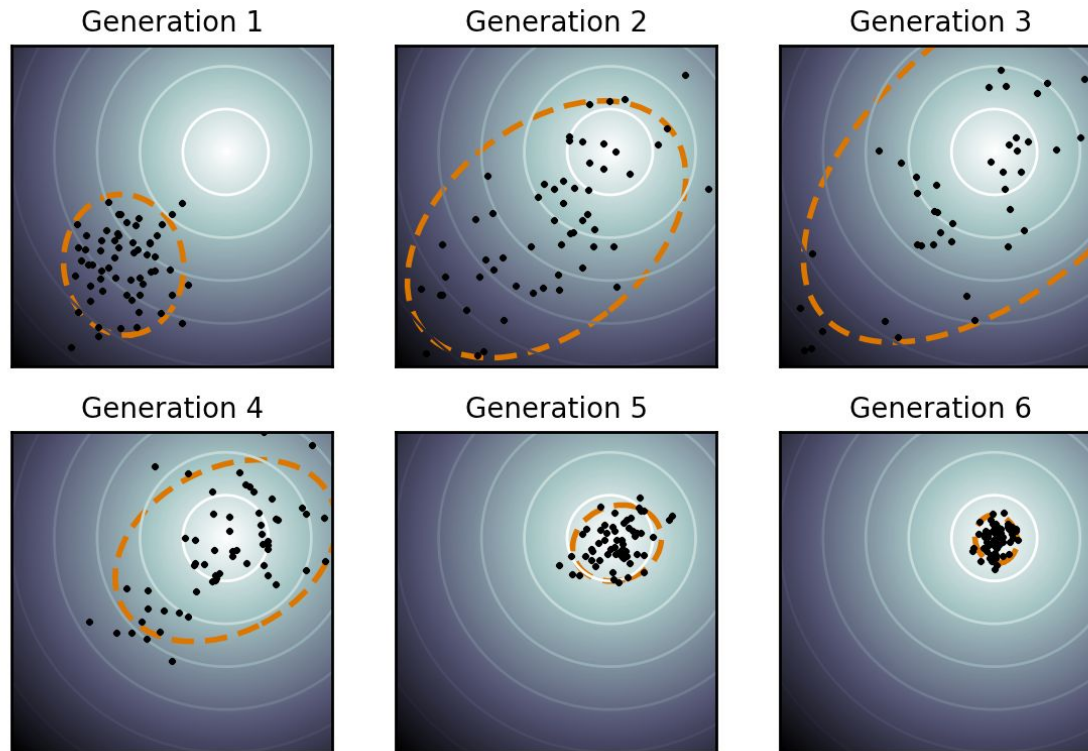
# Covariance Matrix Adaptation Evolution

Iteratively update generations of hyperparameters based on a multivariate Gaussian.

1. Initialize the  $n \times n$  covariance matrix  $C_0 = I$
2. Sample  $n$  hyperparameter sets from a multivariate normal using  $C_k$
3. Evaluate all  $n$  hyperparameter sets
4. Compute a maximum likelihood update  $C_{k+1}$  over the top  $m$  hyperparameter sets
5. Repeat 2-4

# Covariance Matrix Adaptation Evolution

Iteratively update generations of hyperparameters based on a multivariate Gaussian.



# Covariance Matrix Adaptation Evolution

## Pros

- Handles continuous hyperparameters
- Convergence guarantees
- Self-balances exploration/exploitation

## Cons

- Can get stuck in local minima
- Several parameters must be tuned
- Discrete hyperparameters require extra handling

# Particle-Swarm Optimization

Iteratively update generations of hyperparameters with a stochastic velocity term.

1. Uniformly sample a population of  $n$  hyperparameter sets
2. Evaluate all  $n$  members of the population
3. Determine the best-performing hyperparameter set from this step and overall
4. Compute the velocity vector for each hyperparameter set
5. Integrate to update hyperparameter values
6. Repeat 2-4

# Particle Swarm Optimization

Iteratively update generations of hyperparameters with a stochastic velocity term.

The diagram illustrates the velocity update equation for Particle Swarm Optimization. It features a central equation with four boxes above it and two boxes below it, connected by lines to their respective parts of the equation.

Boxes above the equation:

- Velocity Vector
- Inertia Weight
- Acceleration Coefficients
- Difference from Step Best

Equation:

$$v_{i_t} = \omega v_{i_{t-1}} + \phi_p r_p (\lambda_i^* - \lambda_{i_{t-1}}) + \phi_g r_g (\lambda_i^S - \lambda_{i_{t-1}})$$

Boxes below the equation:

- Stochastic Terms (connected to  $\phi_p r_p$ )
- Difference from Swarm Best (connected to  $\lambda_i^S$ )

# Particle Swarm Optimization

## Pros

- Handles continuous hyperparameters
- Convergence guarantees
- Self-balances exploration/exploitation

## Cons

- Can get stuck in local minima
- Discrete hyperparameters require extra handling



# 5. Other Methods

<https://jeffkinnison.github.io/shadho-tutorial/>

# Hyperband

Search over hyperparameters given a budget, a finite number of resources, and successive halving of the search size.

**Algorithm 1:** HYPERBAND algorithm for hyperparameter optimization.

```

input          :  $R, \eta$  (default  $\eta = 3$ )
initialization:  $s_{\max} = \lfloor \log_{\eta}(R) \rfloor, B = (s_{\max} + 1)R$ 
1 for  $s \in \{s_{\max}, s_{\max} - 1, \dots, 0\}$  do
2    $n = \lceil \frac{B}{R} \frac{\eta^s}{(s+1)} \rceil, \quad r = R\eta^{-s}$ 
   // begin SUCCESSIVEHALVING with  $(n, r)$  inner loop
3    $T = \text{get\_hyperparameter\_configuration}(n)$ 
4   for  $i \in \{0, \dots, s\}$  do
5      $n_i = \lfloor n\eta^{-i} \rfloor$ 
6      $r_i = r\eta^i$ 
7      $L = \{\text{run\_then\_return\_val\_loss}(t, r_i) : t \in T\}$ 
8      $T = \text{top\_k}(T, L, \lfloor n_i/\eta \rfloor)$ 
9   end
10 end
11 return Configuration with the smallest intermediate loss seen so far.
```



# Hyperband

Search over hyperparameters given a budget, a finite number of resources, and successive halving of the search size.

## Central Idea



Outer loop  
decreases population.



Inner loop  
increases resources.

# Hyperband Resources

## Training Time

Return performance after an allotted amount of training time.

## Dataset Subsampling

Only provide some subset of training data for hyperparameter evaluation.

## Feature Subsampling

Only allow a subset of features to be used in each evaluation.

# Hyperband

## Pros

- Wraps around other methods
- Enables joint optimization over the resource and performance

## Cons

- Additional constraint of selecting a resource
- Hyperband hyperparameters must be tuned

# Radial Basis Function Surrogates

Update a population of hyperparameters by cubic spline RBF modeling.

1. Collect baseline hyperparameter evaluations using Latin hypercube sampling
2. Fit a surrogate cubic spline RBF model to all evaluated data
3. Compute the probability of perturbing a hyperparameter
4. Sample  $n$  candidate points from the hypercube and perturb them with Gaussian noise
5. Evaluate hyperparameters using the surrogate RBF model
6. Add the evaluated hyperparameters to the session sample
7. Repeat 2-5 until a maximum number of samples are collected

# RBF Surrogates

Update a population of hyperparameters by cubic spline RBF modeling.

# RBF Surrogates

## Pros

- Handles continuous hyperparameters
- Does not require explicit training and evaluation after initial sample

## Cons

- Requires possibly extensive pre-sampling
- Requires solving a linear system over all evaluated hyperparameters
- Performance is tied to the initial sample

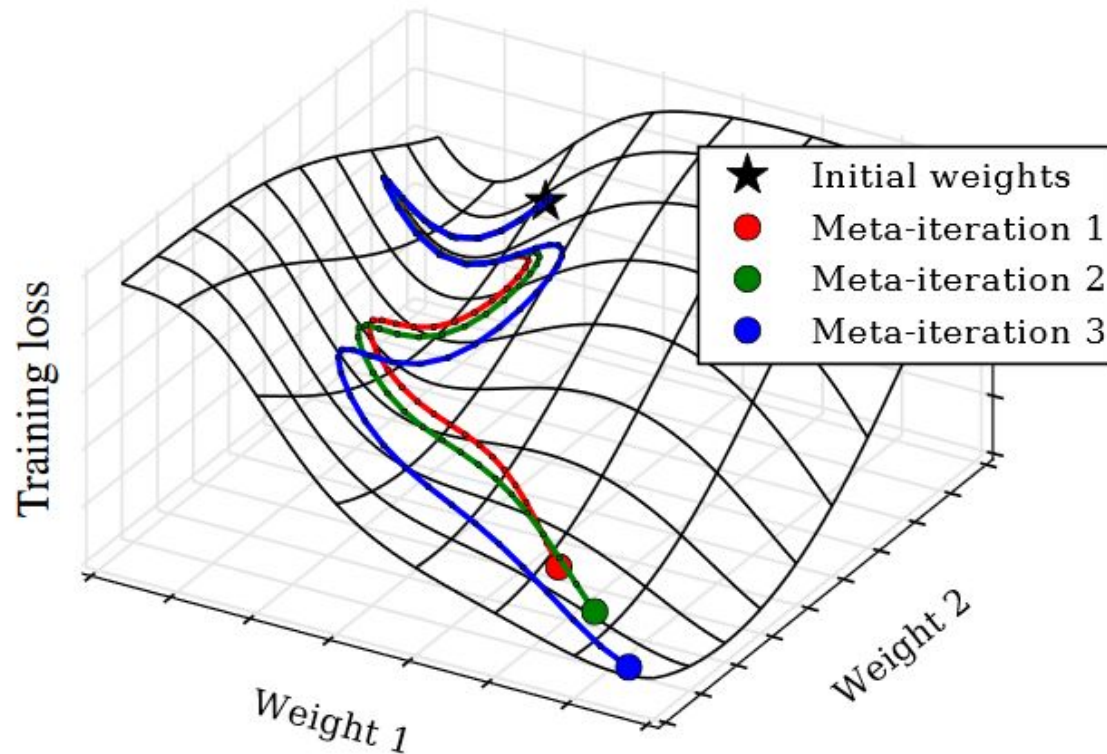
# Hypergradients

Compute the gradient of validation loss with respect to hyperparameters.

1. Initialize hyperparameter weight gradient and velocity gradient vectors
2. Train a model with hyperparameters  $\lambda$
3. Exactly reverse backprop to compute validation loss gradient w.r.t. hyperparameters
4. Update hyperparameters in the direction of this gradient
5. Repeat 2-4

# Hypergradients

Compute the gradient of validation loss with respect to hyperparameters.





# Hypergradients

Compute the gradient of validation loss with respect to hyperparameters.

---

## Algorithm 2 Reverse-mode differentiation of SGD

---

- 1: **input:**  $\mathbf{w}_T, \mathbf{v}_T, \gamma, \alpha$ , train loss  $L(\mathbf{w}, \boldsymbol{\theta}, t)$ , loss  $f(\mathbf{w})$
  - 2: initialize  $d\mathbf{v} = \mathbf{0}, d\boldsymbol{\theta} = \mathbf{0}, d\alpha_t = \mathbf{0}, d\gamma = \mathbf{0}$
  - 3: initialize  $d\mathbf{w} = \nabla_{\mathbf{w}} f(\mathbf{w}_T)$
  - 4: **for**  $t = T$  **counting down to** 1 **do**
  - 5:      $d\alpha_t = d\mathbf{w}^\top \mathbf{v}_t$
  - 6:      $\mathbf{w}_{t-1} = \mathbf{w}_t - \alpha_t \mathbf{v}_t$
  - 7:      $\mathbf{g}_t = \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$
  - 8:      $\mathbf{v}_{t-1} = [\mathbf{v}_t + (1 - \gamma_t)\mathbf{g}_t]/\gamma_t$
  - 9:      $d\mathbf{v} = d\mathbf{v} + \alpha_t d\mathbf{w}$
  - 10:     $d\gamma_t = d\mathbf{v}^\top (\mathbf{v}_t + \mathbf{g}_t)$
  - 11:     $d\mathbf{w} = d\mathbf{w} - (1 - \gamma_t) d\mathbf{v} \nabla_{\mathbf{w}} \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$
  - 12:     $d\boldsymbol{\theta} = d\boldsymbol{\theta} - (1 - \gamma_t) d\mathbf{v} \nabla_{\boldsymbol{\theta}} \nabla_{\mathbf{w}} L(\mathbf{w}_t, \boldsymbol{\theta}, t)$
  - 13:     $d\mathbf{v} = \gamma_t d\mathbf{v}$
  - 14: **end for**
  - 15: **output** gradient of  $f(\mathbf{w}_T)$  w.r.t  $\mathbf{w}_1, \mathbf{v}_1, \gamma, \alpha$  and  $\boldsymbol{\theta}$
-

# Hypergradients

## Pros

- Works with all standard neural network optimizers
- Makes use of more information to guide search
- Can be incorporated into other hyperparameter optimization methods

## Cons

- Requires specialized handling of parameter vector updates
- Has many of the same issues as standard backpropagation
- Number of hyperparameters bounded by validation set size
- Discrete hyperparameters require special handling
-



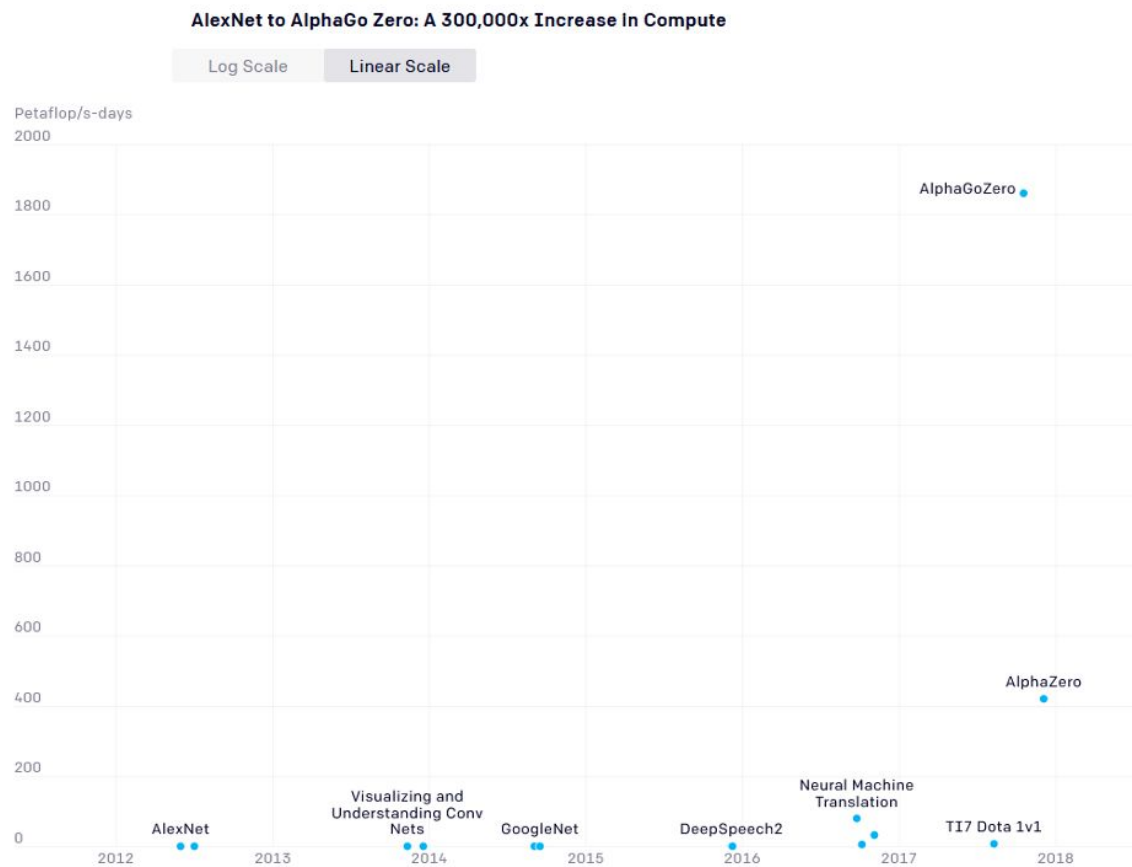
# 6. Neural Architecture Search

<https://jeffkinnison.github.io/shadho-tutorial/>

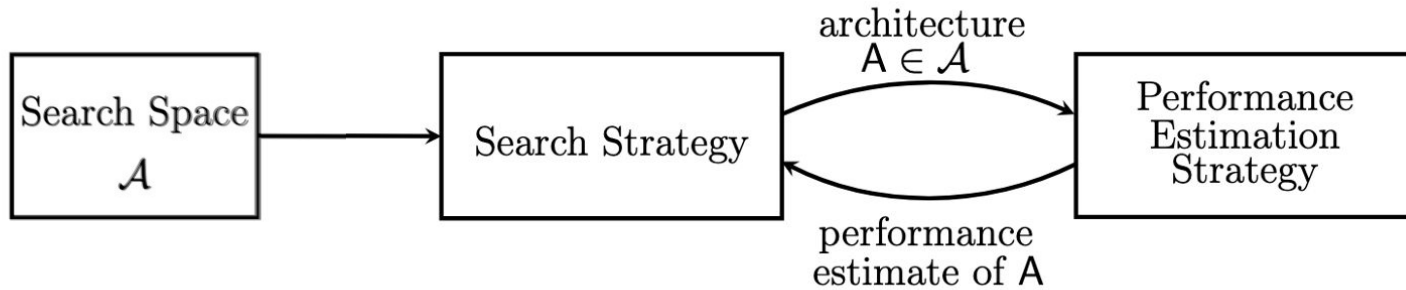
# Compute Costs



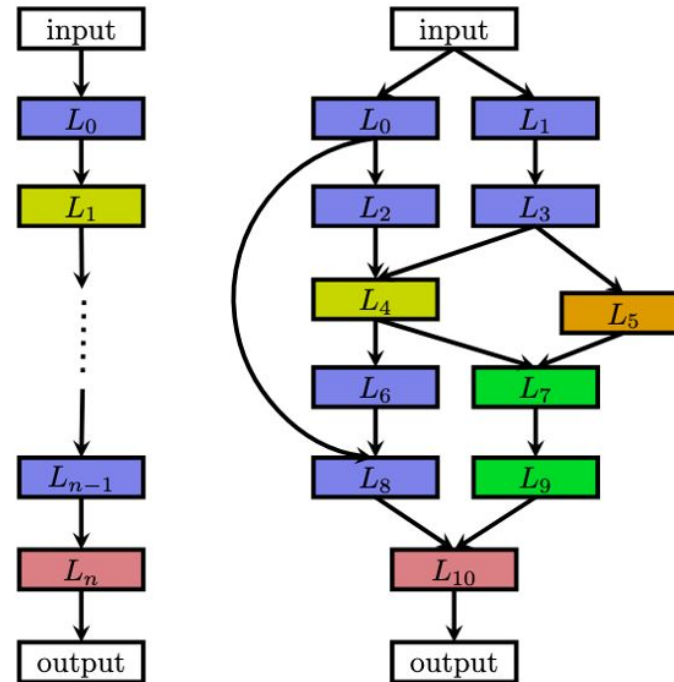
# Compute Costs



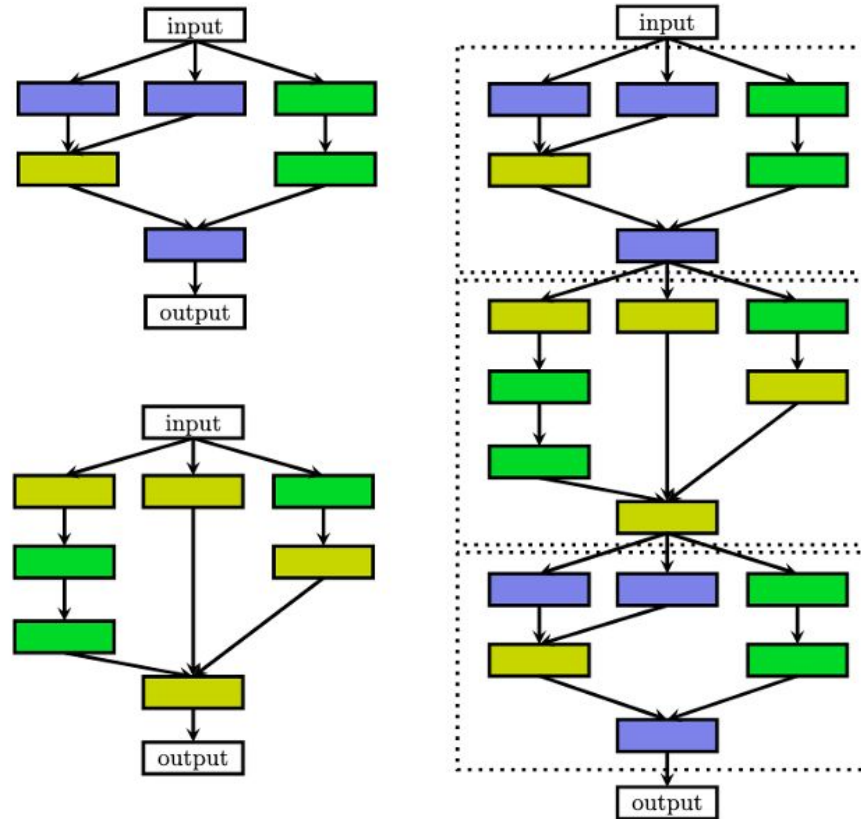
# Optimization Opportunities



# Search Space



# Search Space Reduction





# Search Space Reduction

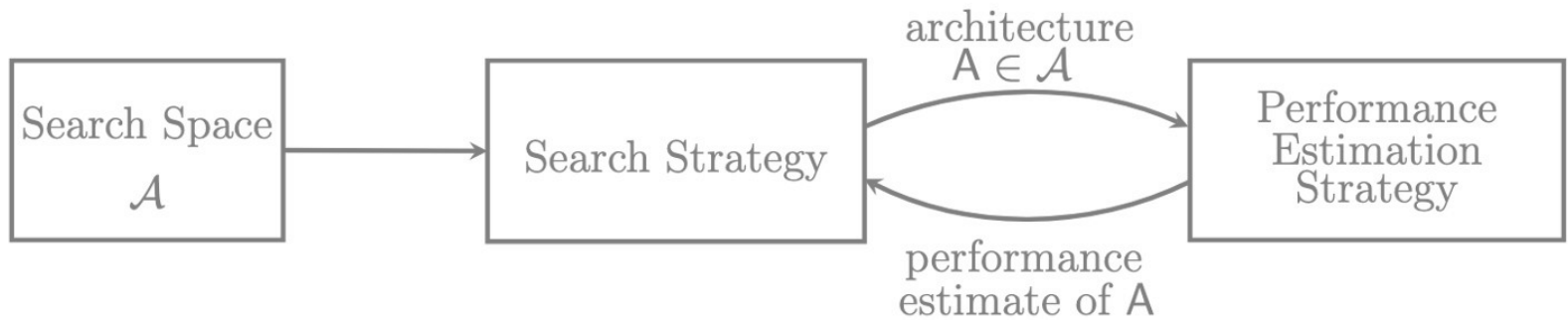
## Pros:

1. Smaller Search Space = speed up in search
2. Cells can be transferred/adapted to other datasets
3. Historical intuition for model building aligns with cells e.g. repeating an LSTM block

## Cons:

1. Cells allow us to ignore microstructures ... but what cells should we consider? How should they be connected?
2. No room for architectural innovation
3. The search space is still huge

# Performance Estimation



# Low Fidelity estimates

Reduce training time

Train on a subset of data

Downscale models

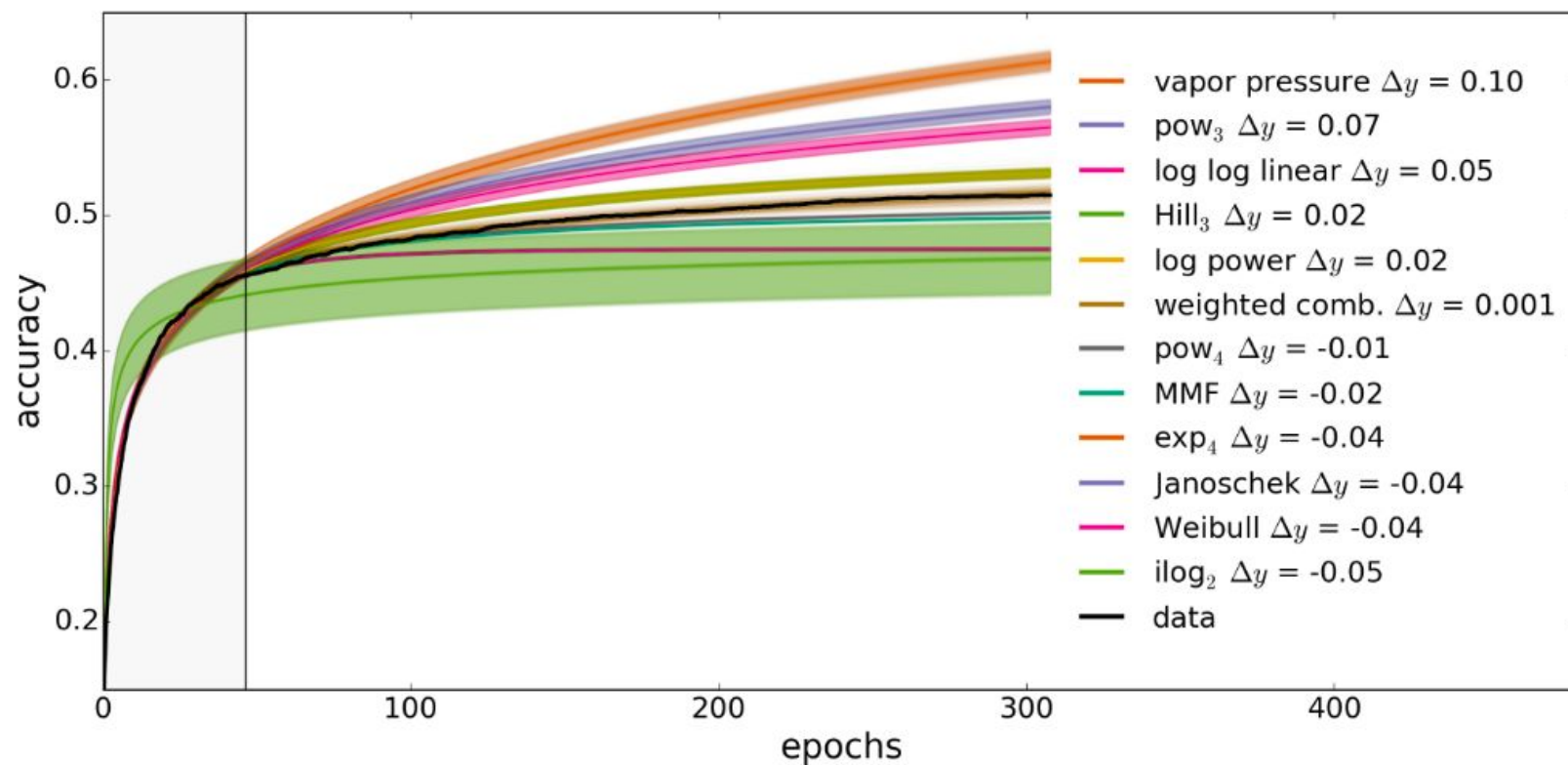
Downscale data

# Low Fidelity Estimates

Training time is reduced

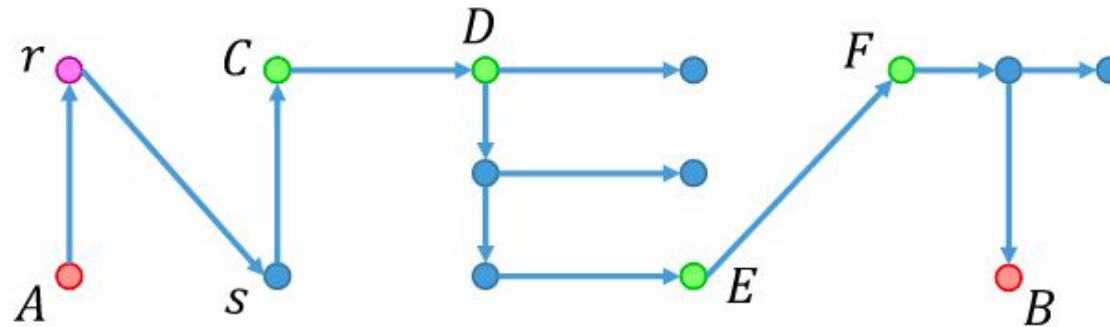
Fails when approximations aren't good approximates!

# Learning Curve Extrapolation

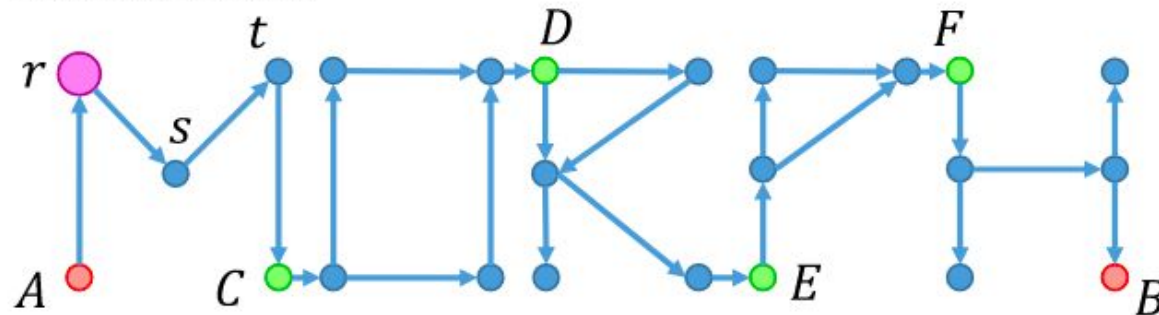


# Weight Inheritance

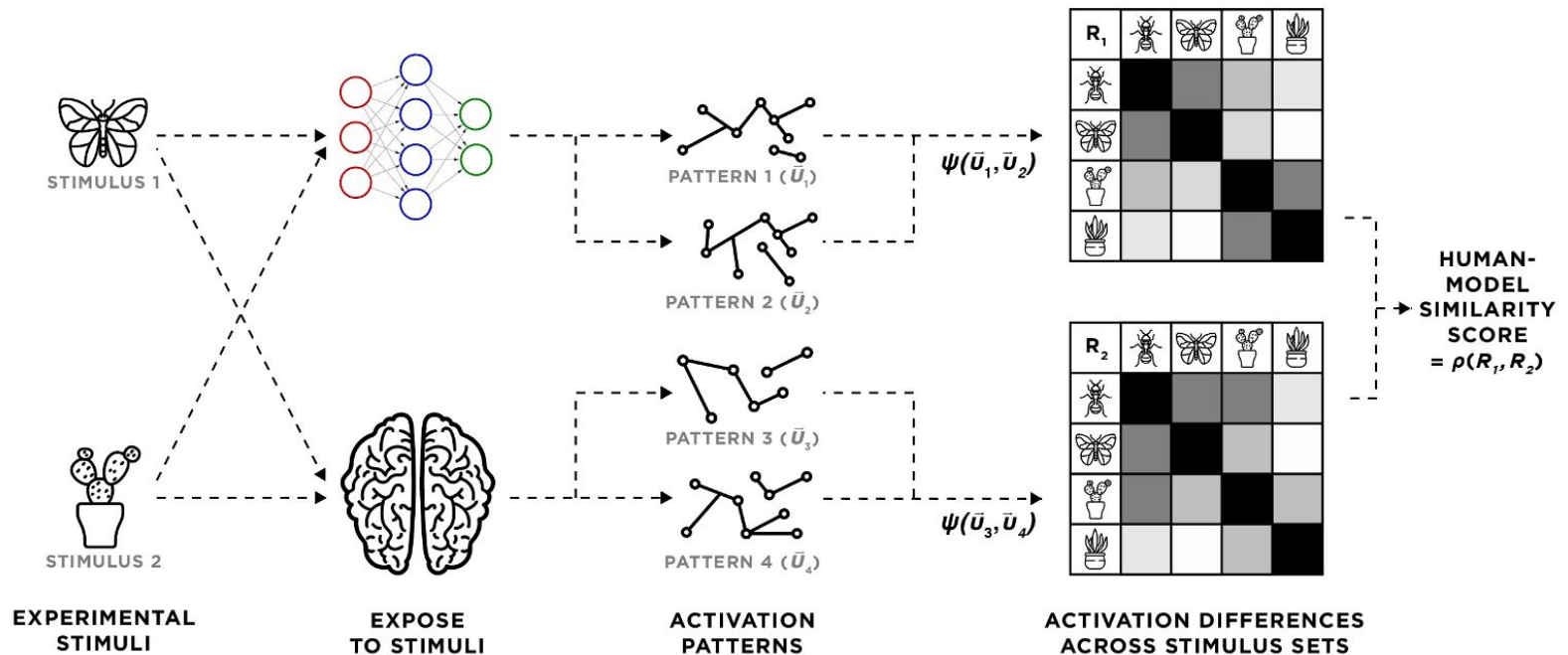
Parent Network



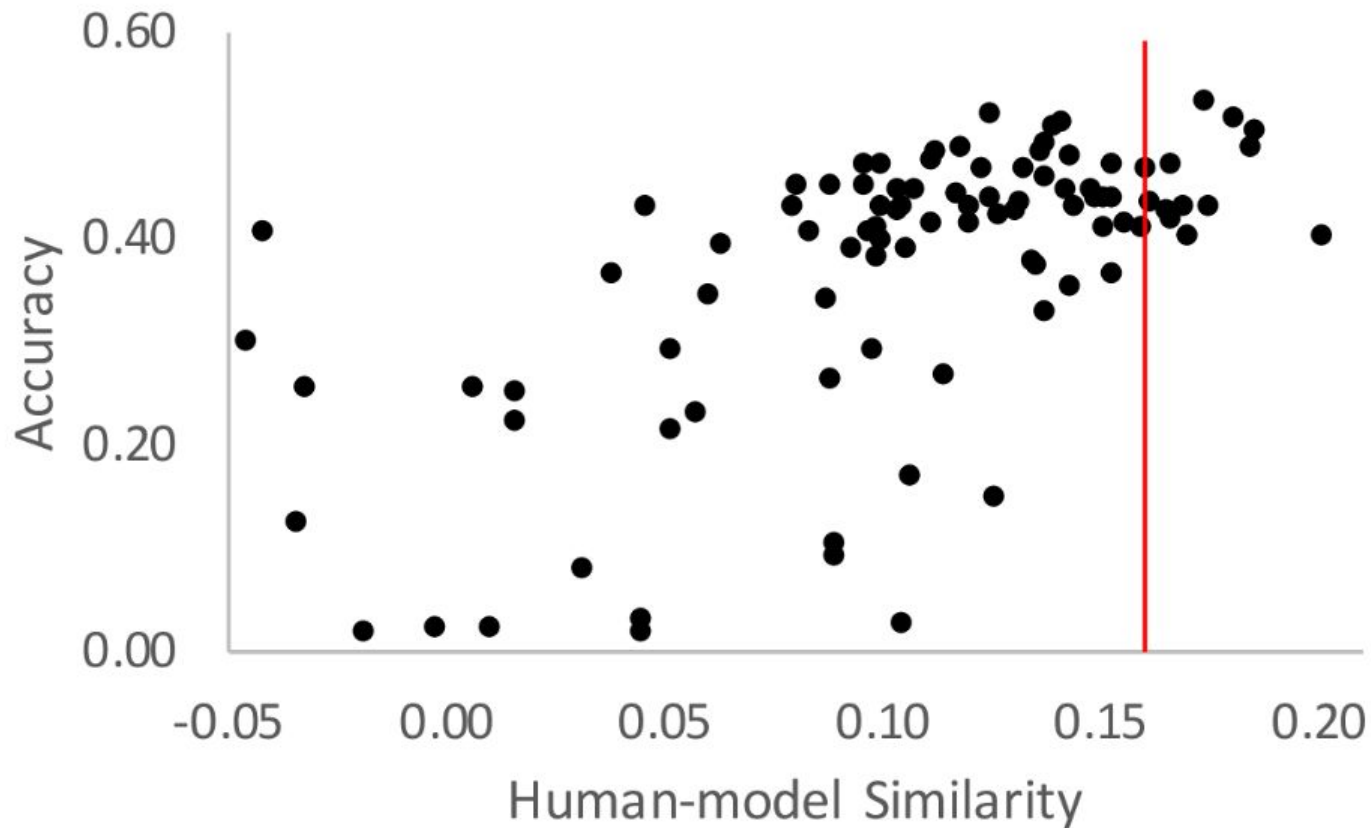
Child Network



# Brain-like Internal Behavior (Human-model Similarity)

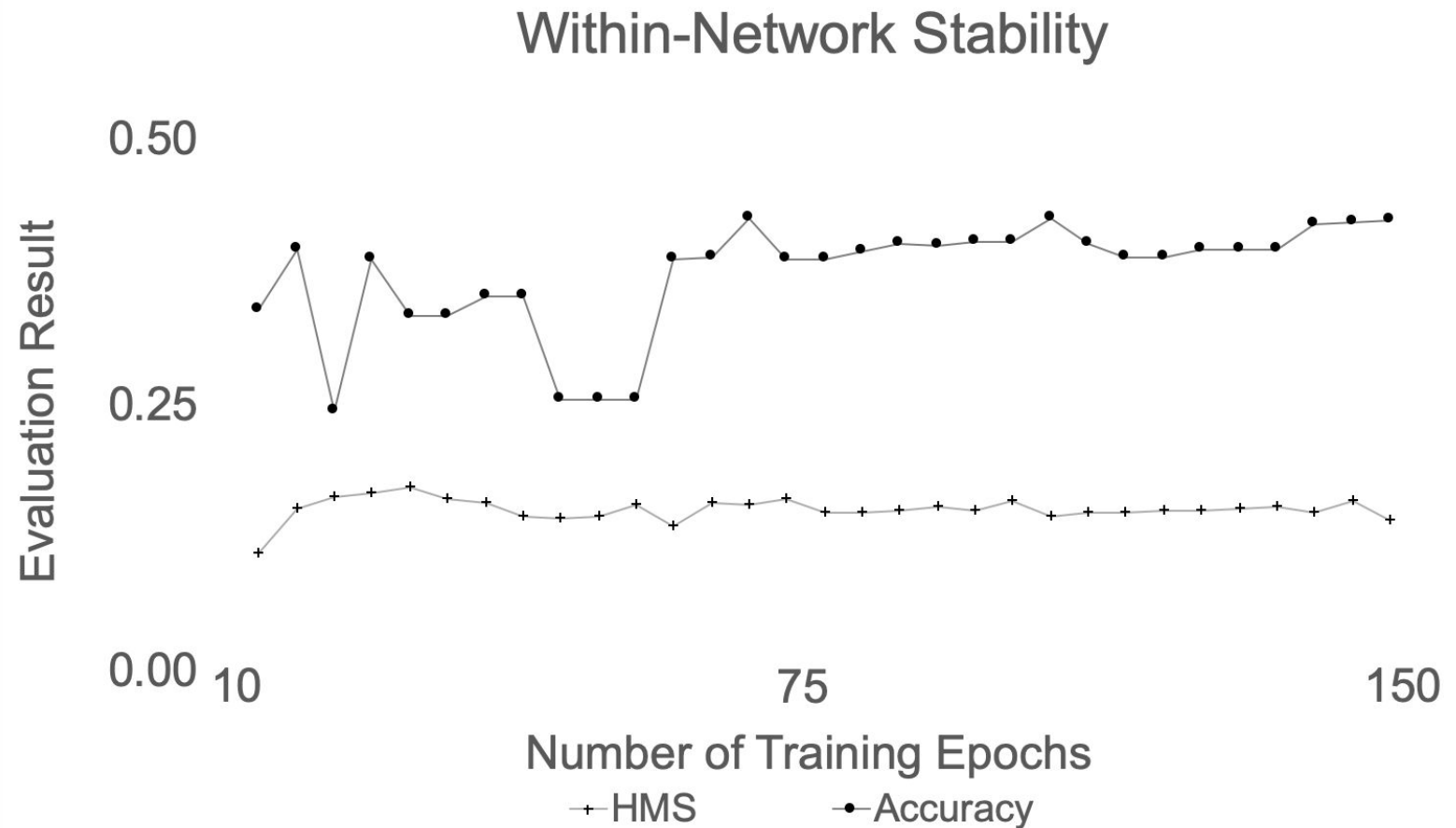


# Brain-like Internal Behavior (Human-model Similarity)





# Brain-like Internal Behavior (Human-model Similarity)



# Brain-like Internal Behavior (Human-model Similarity)

Identifying Stability:

Measure Human-model Similarity at every epoch

Assume stability if, for the last 25 epochs, HMS standard deviation over those epochs  $< 0.01$

Experiment:

Trained 117 CNNs for variable epochs

96% converged

Average convergence 31 epochs, 19% of training

Converged HMS predicts final similarity

Correlation of 0.656

# Search Strategy



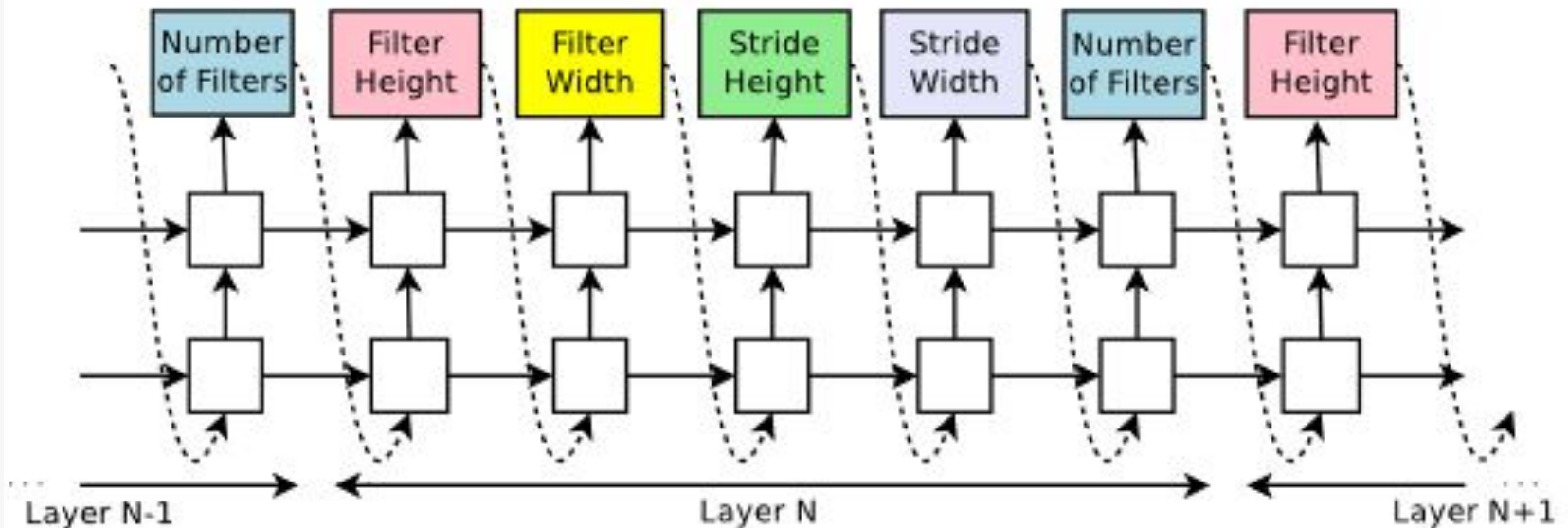
# Neural Architecture Search

Generate architectures using a recurrent neural network controller trained by reinforcement learning.

1. Encode architectural hyperparameters as token sequences
2. Initialize the recurrent controller
3. Sample an architecture
4. Train the sampled architecture
5. Update the controller to minimize validation loss using REINFORCE
6. Repeat 3-5

# Neural Architecture Search

Generate architectures using a recurrent neural network controller trained by reinforcement learning.



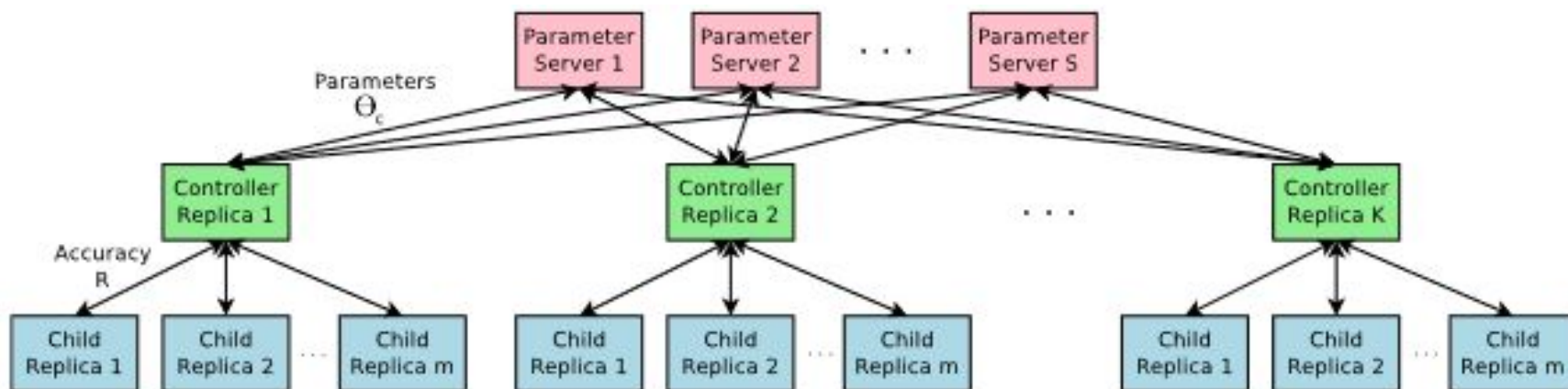
# Neural Architecture Search

Generate architectures using a recurrent neural network controller trained by reinforcement learning.

Model	Depth	Parameters	Error rate (%)
Wide ResNet (Zagoruyko & Komodakis, 2016)	16	11.0M	4.81
	28	36.5M	4.17
ResNet (pre-activation) (He et al., 2016b)	164	1.7M	5.46
	1001	10.2M	4.62
DenseNet ( $L = 40, k = 12$ ) (Huang et al., 2016a)	40	1.0M	5.24
DenseNet ( $L = 100, k = 12$ ) (Huang et al., 2016a)	100	7.0M	4.10
DenseNet ( $L = 100, k = 24$ ) (Huang et al., 2016a)	100	27.2M	3.74
DenseNet-BC ( $L = 100, k = 40$ ) (Huang et al., 2016b)	190	25.6M	3.46
Neural Architecture Search v1 no stride or pooling	15	4.2M	5.50
Neural Architecture Search v2 predicting strides	20	2.5M	6.01
Neural Architecture Search v3 max pooling	39	7.1M	4.47
Neural Architecture Search v3 max pooling + more filters	39	37.4M	3.65

# Neural Architecture Search

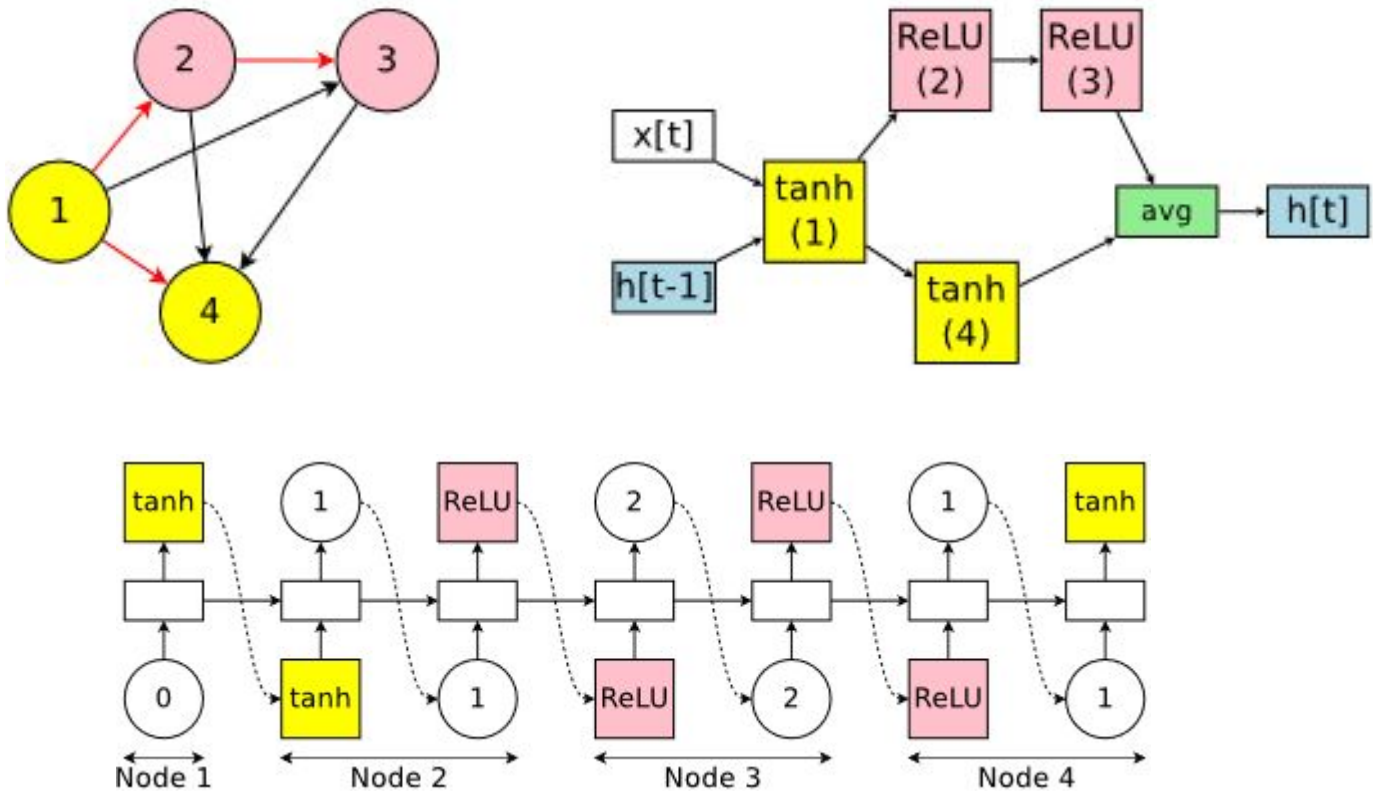
Generate architectures using a recurrent neural network controller trained by reinforcement learning.



Reportedly required 800 GPUs  
3 weeks of training

# Efficient Neural Architecture Search

Update NAS to operate on arbitrary digraphs of architectural hyperparameters.





# Efficient Neural Architecture Search

Update NAS to share weights across many generated architectures.

$$\nabla_{\omega} \mathbb{E}_{\mathbf{m} \sim \pi(\mathbf{m}; \theta)} [\mathcal{L}(\mathbf{m}; \omega)] \approx \frac{1}{M} \sum_{i=1}^M \nabla_{\omega} \mathcal{L}(\mathbf{m}_i, \omega)$$

Diagram illustrating the equation for weight sharing in NAS:

- Shared Weights**: Points to the  $\omega$  parameter in the loss function  $\mathcal{L}(\mathbf{m}; \omega)$ .
- Sampled Architecture**: Points to the sampled architecture  $\mathbf{m}_i$  in the loss function  $\mathcal{L}(\mathbf{m}_i, \omega)$ .

# Efficient Neural Architecture Search

Update NAS to share weights across many generated architectures.

Method	GPUs	Times (days)	Params (million)	Error (%)
SMASH (Brock et al., 2018)	1	1.5	16.0	4.03
NAS (Zoph & Le, 2017)	800	21-28	7.1	4.47
NAS + more filters (Zoph & Le, 2017)	800	21-28	37.4	<b>3.65</b>
ENAS + macro search space	1	0.32	21.3	4.23
ENAS + macro search space + more channels	1	0.32	38.0	<b>3.87</b>
Hierarchical NAS (Liu et al., 2018)	200	1.5	61.3	3.63
Micro NAS + Q-Learning (Zhong et al., 2018)	32	3	—	3.60
Progressive NAS (Liu et al., 2017)	100	1.5	3.2	3.63
NASNet-A (Zoph et al., 2018)	450	3-4	3.3	3.41
NASNet-A + CutOut (Zoph et al., 2018)	450	3-4	3.3	<b>2.65</b>
ENAS + micro search space	1	0.45	4.6	3.54
ENAS + micro search space + CutOut	1	0.45	4.6	<b>2.89</b>

# Additional Related Methods

- Progressive Neural Architecture Search

- Snoek, Jasper, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Mostofa Patwary, Mr Prabhat, Liu, Chenxi, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. "Progressive neural architecture search." In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 19-34. 2018.

- Autoaugment: Learning Augmentation Strategies from Data

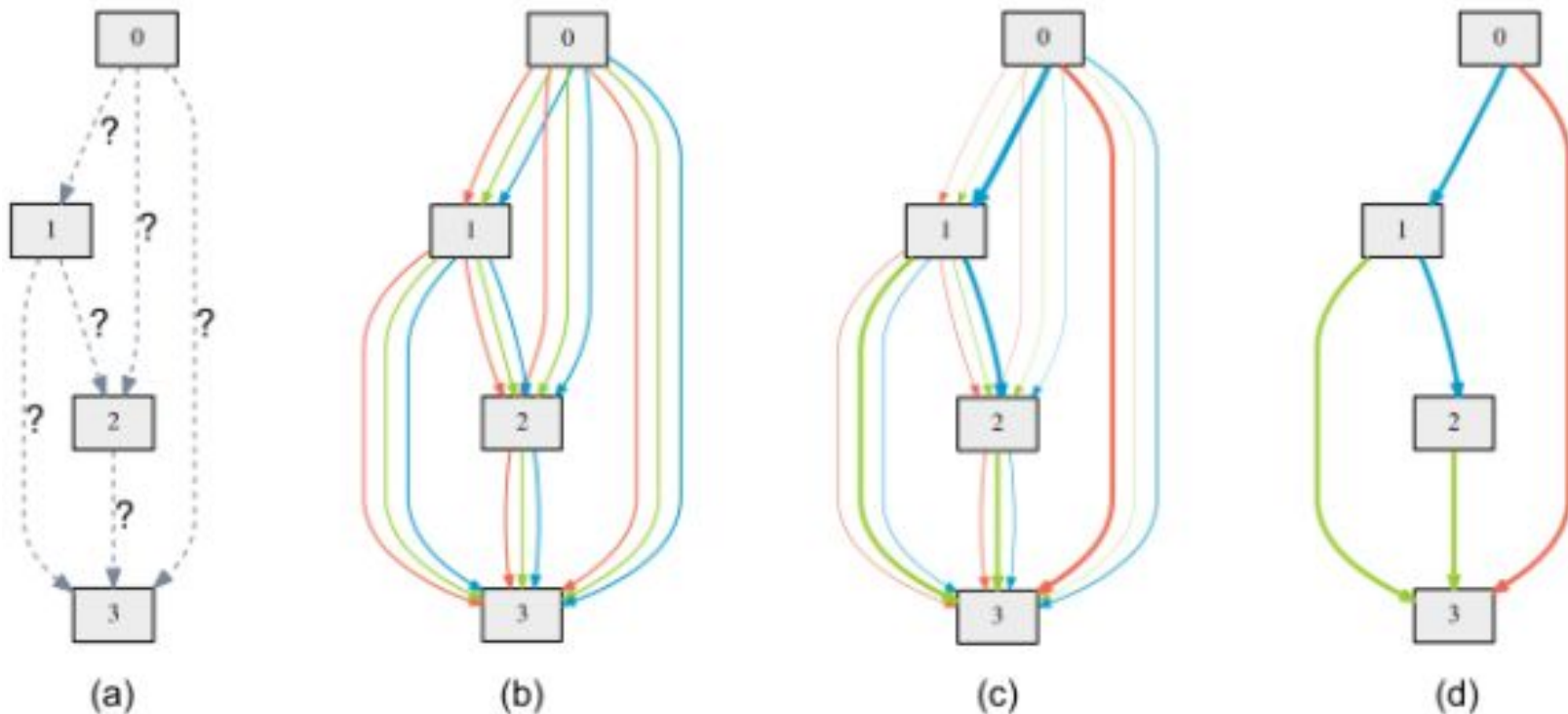
- Cubuk, Ekin D., Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V. Le. "Autoaugment: Learning augmentation strategies from data." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 113-123. 2019.

- MNASNet: Platform-Aware Neural Architecture Search for Mobile

- Tan, Mingxing, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. "Mnasnet: Platform-aware neural architecture search for mobile." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2820-2828. 2019.

# Differentiable Architecture Search

Determine the optimal architecture through continuous relaxation to approximate the architecture gradient.



# Differentiable Architecture Search

Determine the optimal architecture through continuous relaxation to approximate the architecture gradient.

Architecture	Test Error (%)	Params (M)	Search Cost (GPU days)	#ops	Search Method
DenseNet-BC (Huang et al., 2017)	3.46	25.6	–	–	manual
NASNet-A + cutout (Zoph et al., 2018)	2.65	3.3	2000	13	RL
NASNet-A + cutout (Zoph et al., 2018) <sup>†</sup>	2.83	3.1	2000	13	RL
BlockQNN (Zhong et al., 2018)	3.54	39.8	96	8	RL
AmoebaNet-A (Real et al., 2018)	$3.34 \pm 0.06$	3.2	3150	19	evolution
AmoebaNet-A + cutout (Real et al., 2018) <sup>†</sup>	3.12	3.1	3150	19	evolution
AmoebaNet-B + cutout (Real et al., 2018)	$2.55 \pm 0.05$	2.8	3150	19	evolution
Hierarchical evolution (Liu et al., 2018b)	$3.75 \pm 0.12$	15.7	300	6	evolution
PNAS (Liu et al., 2018a)	$3.41 \pm 0.09$	3.2	225	8	SMBO
ENAS + cutout (Pham et al., 2018b)	2.89	4.6	0.5	6	RL
ENAS + cutout (Pham et al., 2018b) <sup>*</sup>	2.91	4.2	4	6	RL
Random search baseline <sup>†</sup> + cutout	$3.29 \pm 0.15$	3.2	4	7	random
DARTS (first order) + cutout	$3.00 \pm 0.14$	3.3	1.5	7	gradient-based
DARTS (second order) + cutout	$2.76 \pm 0.09$	3.3	4	7	gradient-based



# 7. Hyperparameter Importance

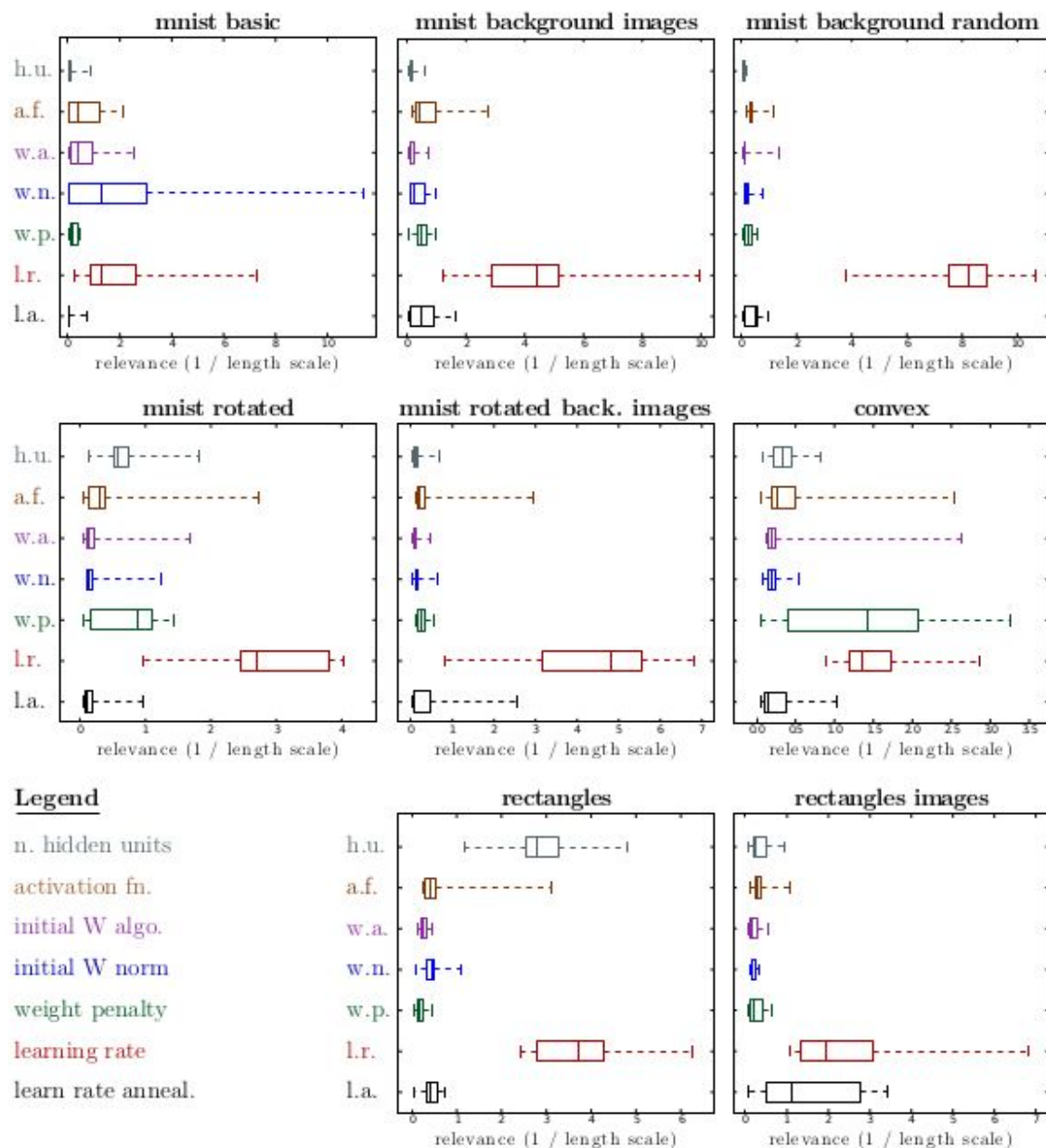
<https://jeffkinnison.github.io/shadho-tutorial/>

# Assessments for Random Search

How does a hyperparameter influence performance when taking on different values?

1. Evaluate a number of different hyperparameters
2. Model the objective with respect to each hyperparameter individually with a Gaussian Process with RBF kernel
3. Estimate length scale of the kernel of each model







# QIM

Evaluate hyperparameter importance with Plackett-Burman design.

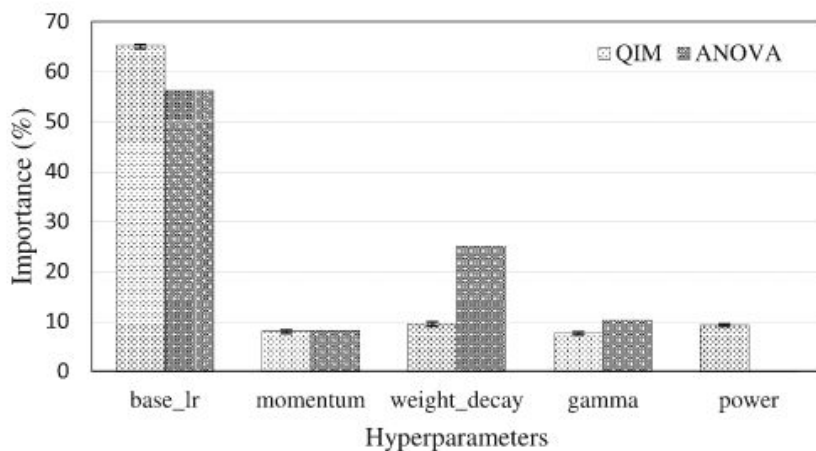
1. Evaluate a number of different hyperparameters to determine a viable range for each hyperparameter
2. Apply Plackett-Burman matrix to combine performance measures for each trial
3. The resulting vector gives an importance score for each hyperparameter

**Table 1.** The PB design matrix with 8 experiments.

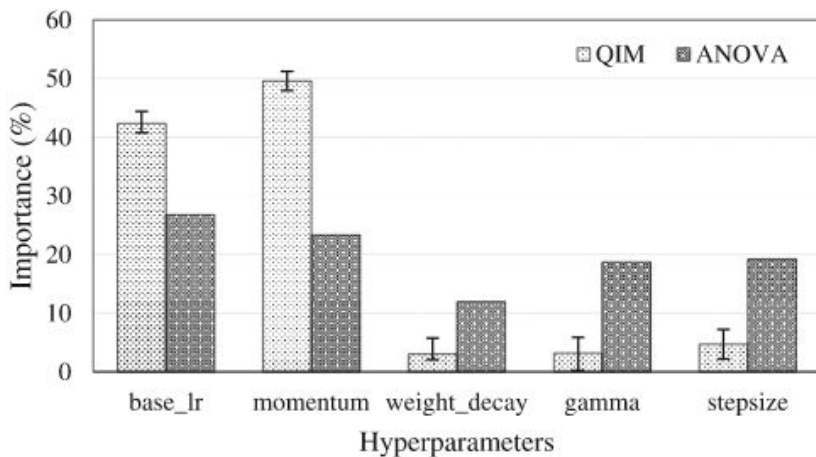
Assembly	Parameters or factors						
	x1	x2	x3	x4	x5	x6	x7
1	+1	+1	+1	-1	+1	-1	-1
2	-1	+1	+1	+1	-1	+1	-1
3	-1	-1	+1	+1	+1	-1	+1
4	+1	-1	-1	+1	+1	+1	-1
5	-1	+1	-1	-1	+1	+1	+1
6	+1	-1	+1	-1	-1	+1	+1
7	+1	+1	-1	+1	-1	-1	+1
8	-1	-1	-1	-1	-1	-1	-1

# QIM

Evaluate hyperparameter importance with Plackett-Burman design.



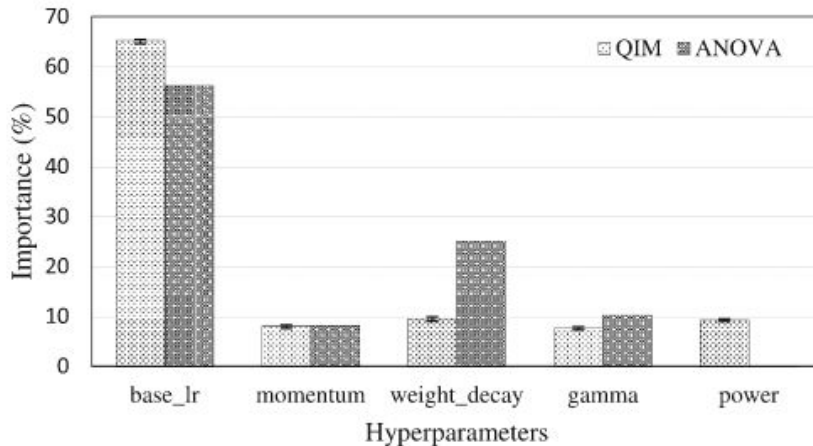
LeNet on MNIST



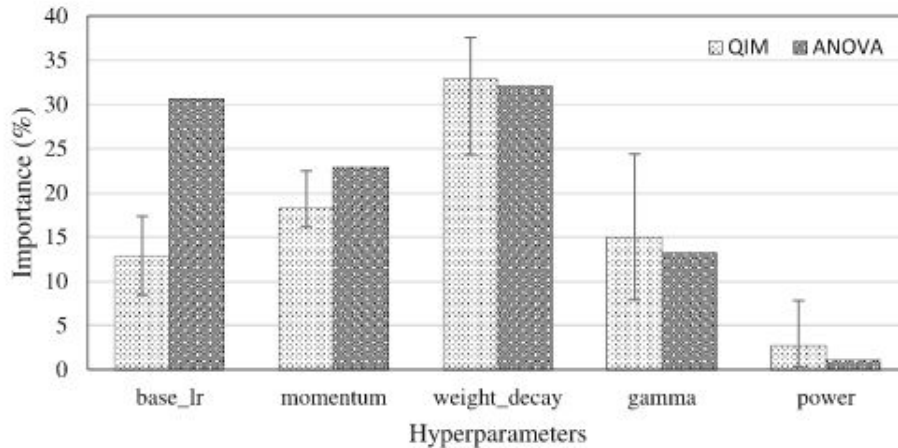
Autoencoder on MNIST

# QIM

Evaluate hyperparameter importance with Plackett-Burman design.



LeNet on MNIST



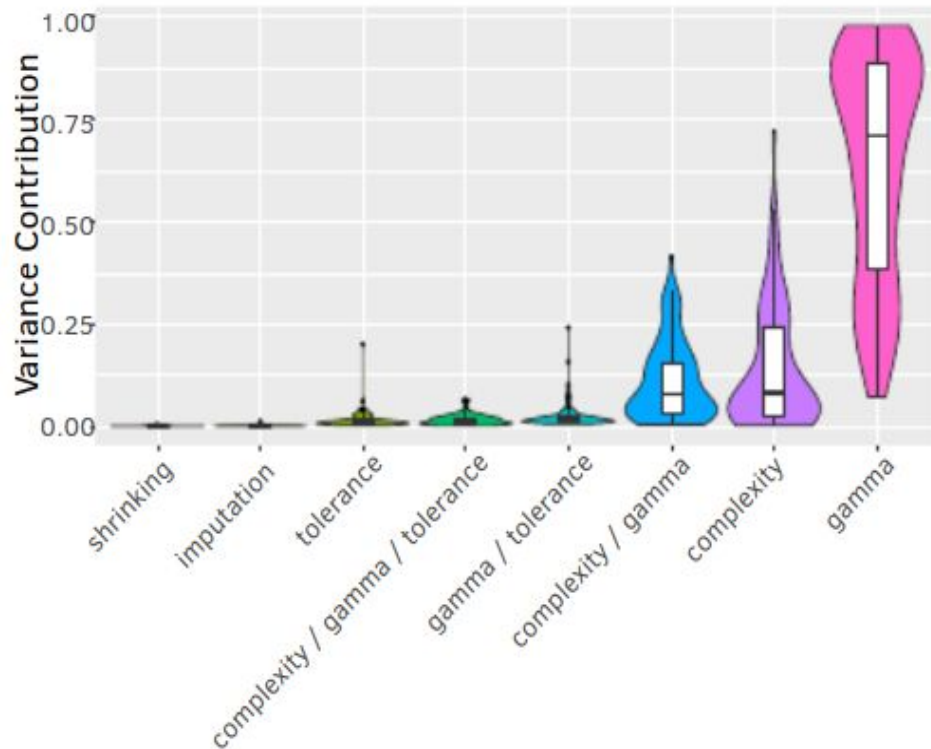
LeNet on CIFAR-10

# Functional ANOVA

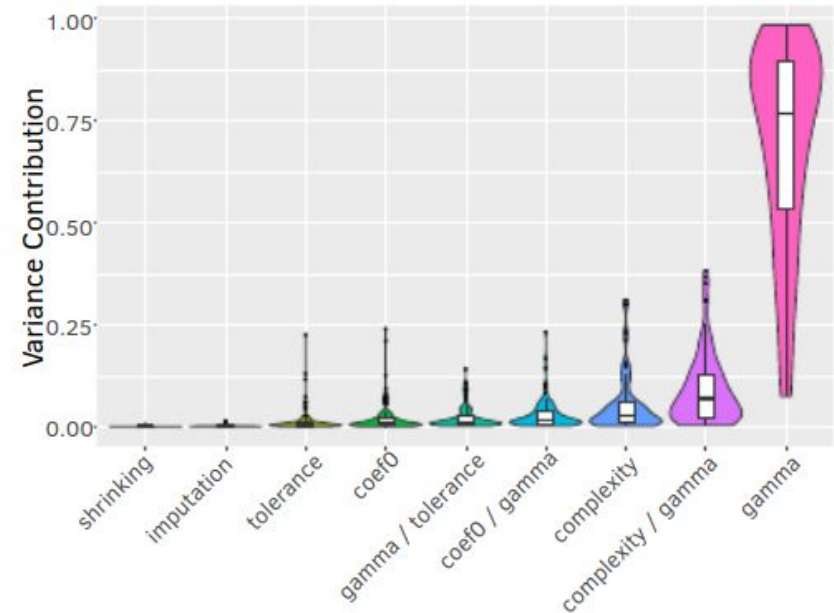
Evaluate hyperparameter importance with Plackett-Burman design.

1. Evaluate a number of different hyperparameters
2. Model the objective with a functional ANOVA random forest
3. Retrieve the variance contribution for each hyperparameter from the forest

# Functional ANOVA

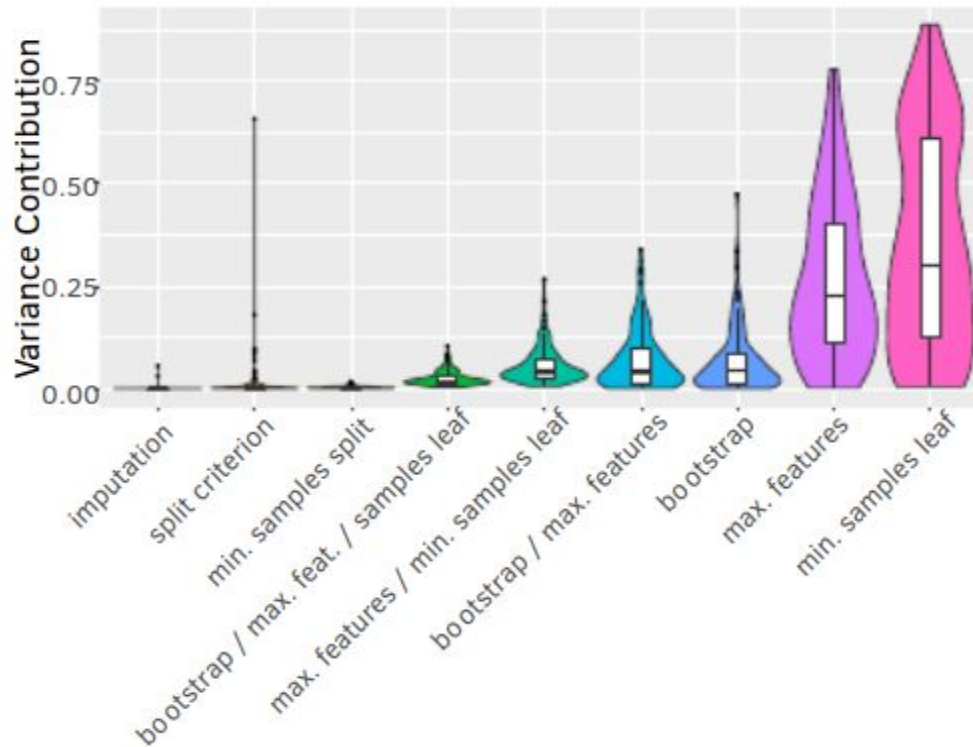


SVM w/RBF kernel

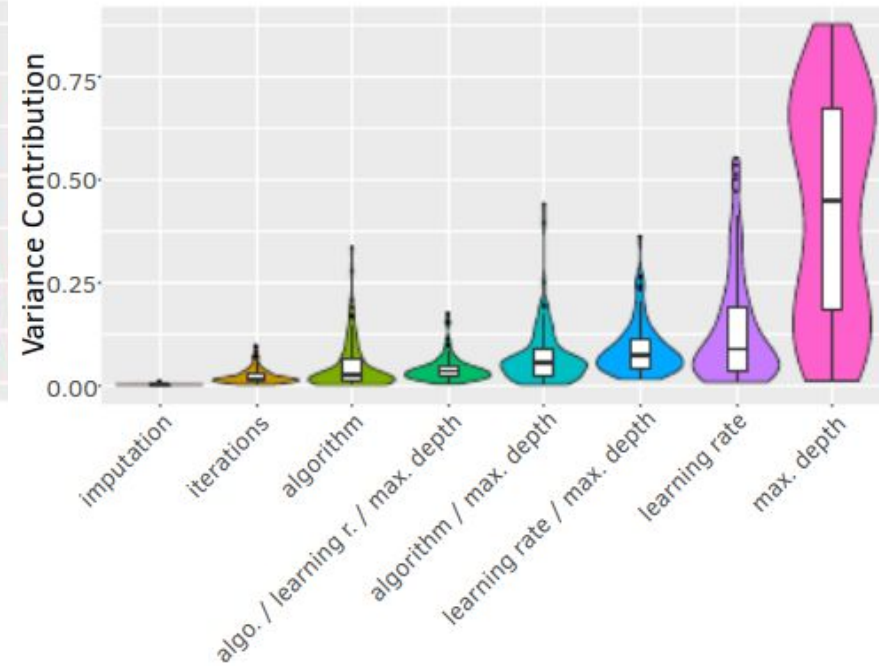


SVM w/sigmoid kernel

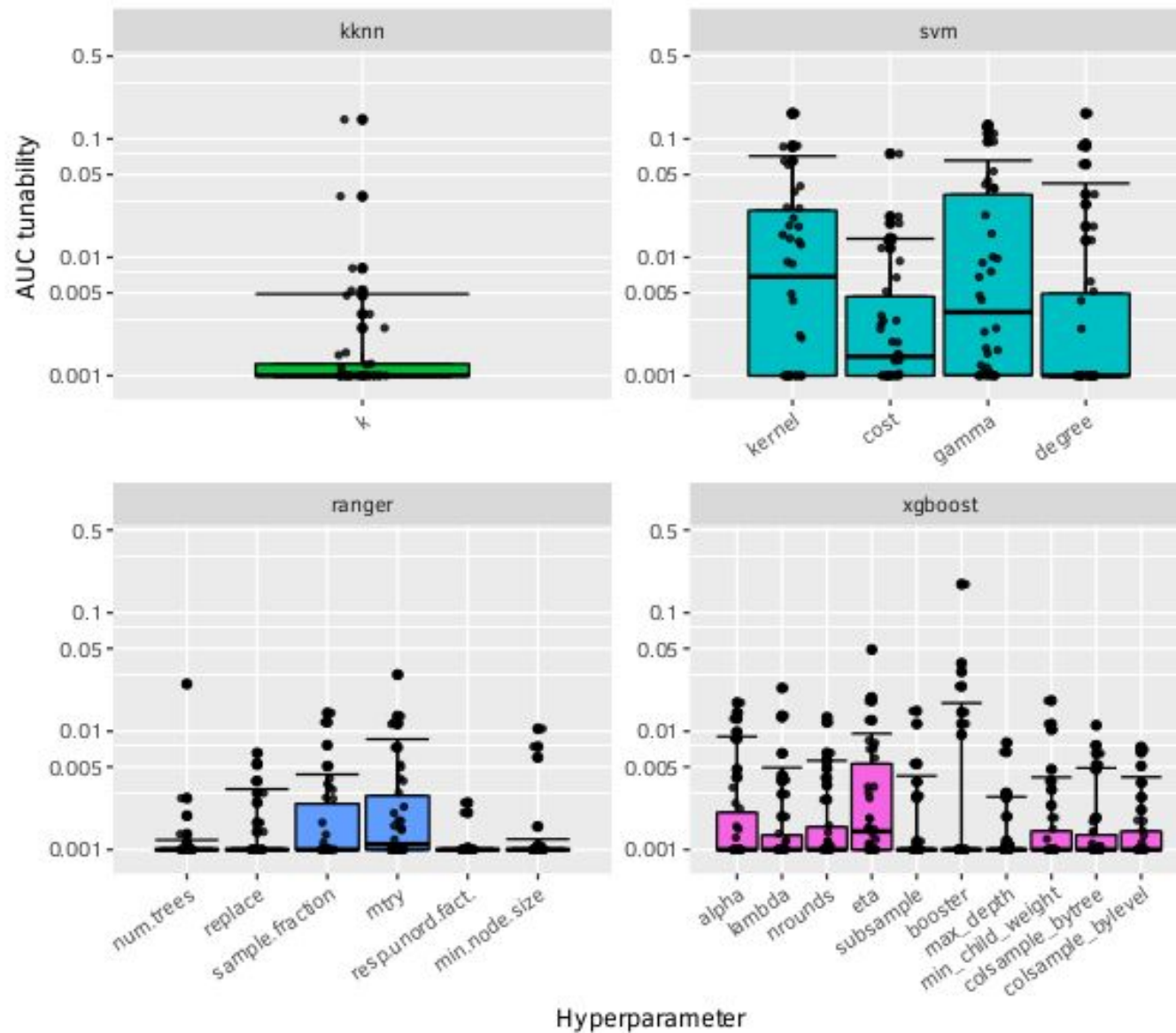
# Functional ANOVA



Random Forest



Adaboost



# Takeaway

Hyperparameter importance methods consistently show that their impact on performance depends on the learning algorithm (e.g. varying ANN architectures) and dataset.





# THANKS!

---

**Any questions?**