Automated Machine Learning (AutoML)

Amir H. Payberah
payberah@kth.se
2020-11-23
The Course Web Page

https://fid3024.github.io
Building an ML model is an iterative, complex, and time-consuming process. It can take a lot of trial and error.

[Elshawi et al., Automated Machine Learning: State-of-The-Art and Open Challenges, 2019]
Automated vs. Manual Machine Learning

Task → Human expert → Models

manual trial and error (and intuition)

performance
Automated vs. Manual Machine Learning

- **AutoML**: build models in a **data-driven**, **intelligent**, and **purposeful** way.

[Joaquin Vanschoren, Automatic Machine Learning - A Tutorial]
AutoML Subproblems - Neural Architecture Search

- Represent and search all pipelines or neural nets, e.g., neural layers, interconnections, etc.

[Search space design] → [Architecture search strategy] → [Performance estimation strategy]

[arch] → [score]

[Joaquin Vanschoren, Automatic Machine Learning - A Tutorial]
AutoML Subproblems - Hyperparameter Optimization

- Which hyperparameters are important? How to optimize them?

[Joaquin Vanschoren, Automatic Machine Learning - A Tutorial]
AutoML Subproblems - Meta-learning

- How can we transfer experience from previous tasks?
- Don’t start from scratch (search space is too large).

[Joakin Vanschoren, Automatic Machine Learning - A Tutorial]
Meta-learning  Neural Architecture Search  Hyperparameter Optimization
Hyper-Parameter Optimization (HPO)
AutoML Definition

- $A$ denotes an ML algorithms with $m$ hyperparameters.

- $\{A_1, A_2, \ldots, A_n\}$ is a set of ML algorithms.

- $\Lambda_j$ is the domain of the $j$th hyperparameter.

- $\Lambda = \Lambda_1 \times \Lambda_2 \times \cdots \times \Lambda_m$ is the overall hyperparameter configuration space.

- $\theta \in \Lambda$ is a vector of hyperparameters.

- $J(\theta, X_{\text{train}}, X_{\text{valid}})$ is the loss of the ML model created by $\theta$, trained on $X_{\text{train}}$, and validated on $X_{\text{valid}}$.

- Find the configuration that minimizes the expected loss on a dataset $X_{\text{train}}$:
  $$\hat{\theta} = \arg\min_{\theta \in \Lambda} E(X_{\text{train}}, X_{\text{valid}}) \sim X_{\text{train}} J(\theta, X_{\text{train}}, X_{\text{valid}}).$$
AutoML Definition

- $A$ denotes an ML algorithms with $m$ hyperparameters.
- $\{A_1, A_2, \cdots, A_n\}$ is a set of ML algorithms.
AutoML Definition

- $A$ denotes an ML algorithms with $m$ hyperparameters.
- $\{A_1, A_2, \cdots, A_n\}$ is a set of ML algorithms.
- $\Lambda_j$ is the domain of $j$th hyperparameter.
AutoML Definition

- $A$ denotes an ML algorithms with $m$ hyperparameters.
- $\{A_1, A_2, \cdots, A_n\}$ is a set of ML algorithms.
- $\Lambda_j$ is the domain of $j$th hyperparameter.
- $\Lambda = \Lambda_1 \times \Lambda_2 \times \cdots \times \Lambda_m$ is the overall hyperparameter configuration space.
AutoML Definition

- $\mathcal{A}$ denotes an ML algorithms with $m$ hyperparameters.
- $\{\mathcal{A}_1, \mathcal{A}_2, \cdots, \mathcal{A}_n\}$ is a set of ML algorithms.
- $\Lambda_j$ is the domain of $j$th hyperparameter.
- $\Lambda = \Lambda_1 \times \Lambda_2 \times \cdots \times \Lambda_m$ is the overall hyperparameter configuration space.
- $\theta \in \Lambda$ is a vector of hyperparameters.
AutoML Definition

- $A$ denotes an ML algorithms with $m$ hyperparameters.
- \( \{A_1, A_2, \cdots, A_n\} \) is a set of ML algorithms.
- $\Lambda_j$ is the domain of $j$th hyperparameter.
- $\Lambda = \Lambda_1 \times \Lambda_2 \times \cdots \times \Lambda_m$ is the overall hyperparameter configuration space.
- $\theta \in \Lambda$ is a vector of hyperparameters.
- $J(\theta, X_{\text{train}}, X_{\text{valid}})$ is the loss of the ML model created by $\theta$, trained on $X_{\text{train}}$, and validated on $X_{\text{valid}}$. 
AutoML Definition

- $A$ denotes an ML algorithms with $m$ hyperparameters.
- $\{A_1, A_2, \ldots, A_n\}$ is a set of ML algorithms.
- $\Lambda_j$ is the domain of $j$th hyperparameter.
- $\Lambda = \Lambda_1 \times \Lambda_2 \times \cdots \times \Lambda_m$ is the overall hyperparameter configuration space.
- $\theta \in \Lambda$ is a vector of hyperparameters.
- $J(\theta, X_{\text{train}}, X_{\text{valid}})$ is the loss of the ML model created by $\theta$, trained on $X_{\text{train}}$, and validated on $X_{\text{valid}}$.
- Find the configuration that minimizes the expected loss on a dataset $X_{\text{train}}$:
  \[\theta^* = \arg \min_{\theta \in \Lambda} \mathbb{E}_{(X_{\text{train}}, X_{\text{valid}}) \sim X} J(\theta, X_{\text{train}}, X_{\text{valid}})\]
Types of Hyperparameters

- **Continuous**
  - E.g., learning rate
Types of Hyperparameters

- **Continuous**
  - E.g., learning rate

- **Integer**
  - E.g., number of hidden units
Types of Hyperparameters

- **Continuous**
  - E.g., learning rate

- **Integer**
  - E.g., number of hidden units

- **Categorical**
  - E.g., choose of operator (Convolution, MaxPooling, DropOut, etc.)
  - E.g., activation function (ReLU, Leaky ReLU, tanh, etc.)
Types of Hyperparameters

- **Continuous**
  - E.g., learning rate

- **Integer**
  - E.g., number of hidden units

- **Categorical**
  - E.g., choose of operator (Convolution, MaxPooling, DropOut, etc.)
  - E.g., activation function (ReLU, Leaky ReLU, tanh, etc.)

- **Conditional**
  - E.g., convolution kernel size, if convolution layer is selected
Hyper-Parameter Optimization

- Black-box optimization

- Multi-fidelity optimization
Hyper-Parameter Optimization

- Black-box optimization
  - Grid search
  - Random search
  - Population-based search
  - Bayesian optimization

- Multi-fidelity optimization
Hyper-Parameter Optimization

- **Black-box optimization**
  - Grid search
  - Random search
  - Population-based search
  - Bayesian optimization

- **Multi-fidelity optimization**
  - Modeling learning curve
  - Bandit based
Hyper-Parameter Optimization

▶ Black-box optimization
• Grid search
• Random search
• Population-based search
• Bayesian optimization

▶ Multi-fidelity optimization
• Modeling learning curve
• Bandit based
Black-box Optimization - Grid and Random Search

[Grid Search vs Random Search diagram]

[Hutter et al., Automated Machine Learning, 2019]
They maintain a **population**, i.e., a **set of configurations**.
They maintain a population, i.e., a set of configurations.

Improve this population to obtain a new generation of better configurations.
They maintain a population, i.e., a set of configurations.

Improve this population to obtain a new generation of better configurations.

Achieve this by applying:
- Local perturbations (so-called mutations)
- Combinations of different members (so-called crossover)
They maintain a population, i.e., a set of configurations.

Improve this population to obtain a new generation of better configurations.

Achieve this by applying:
- Local perturbations (so-called mutations)
- Combinations of different members (so-called crossover)

E.g., genetic algorithms, evolutionary algorithms, particle swarm optimization
Start with a few (random) hyperparameter configurations.
Black-box Optimization - Bayesian Optimization (1/3)

- Start with a few (random) hyperparameter configurations.
- Fit a surrogate model to predict other configurations.

[Hutter et al., Automated Machine Learning, 2019]
Start with a few (random) hyperparameter configurations.

Fit a surrogate model to predict other configurations.

An acquisition function drives the proposition of new points to test, in an exploration and exploitation trade-off.

[Hutter et al., Automated Machine Learning, 2019]
- Start with a few (random) hyperparameter configurations.
- Fit a surrogate model to predict other configurations.
- An acquisition function drives the proposition of new points to test, in an exploration and exploitation trade-off.
- Sample for the best configuration under that function.

[Hutter et al., Automated Machine Learning, 2019]
Black-box Optimization - Bayesian Optimization (2/3)
Black-box Optimization - Bayesian Optimization (2/3)
Black-box Optimization - Bayesian Optimization (2/3)

[Hutter et al., Automated Machine Learning, 2019]
Black-box Optimization - Bayesian Optimization (3/3)

[Hutter et al., Automated Machine Learning, 2019]
Hyper-Parameter Optimization

- **Black-box optimization**
  - Grid search
  - Random search
  - Population-based search
  - Bayesian optimization

- **Multi-fidelity optimization**
  - Modeling learning curve
  - Bandit based
Massive dataset sizes and complex models make blackbox performance evaluation expensive.
Multi-fidelity Optimization

- Massive **dataset sizes** and **complex models** make **blackbox** performance evaluation expensive.

- **Probe a hyperparameter configuration** on a **small subset**.
Multi-fidelity Optimization

- Massive **dataset sizes** and **complex models** make **blackbox** performance evaluation expensive.

- **Probe a hyperparameter configuration** on a **small subset**.

- **Multi-fidelity** methods use **low fidelity approximations** of the **actual loss function** to minimize.
Multi-fidelity Optimization

- Massive **dataset sizes** and **complex models** make **blackbox** performance evaluation expensive.

- **Probe a hyperparameter configuration** on a **small subset**.

- **Multi-fidelity** methods use **low fidelity approximations** of the **actual loss function** to minimize.

- These approximations introduce a **tradeoff** between **optimization performance** and runtime.
Learning curve extrapolation is used in predicting early termination for a particular configuration.
Learning curve extrapolation is used in predicting early termination for a particular configuration.

Models learning curves during hyper-parameter optimization.
Learning curve extrapolation is used in predicting early termination for a particular configuration.

Models learning curves during hyper-parameter optimization.

Decides whether to allocate more resources or to stop the training procedure for a particular configuration.
Learning curve extrapolation is used in predicting early termination for a particular configuration.

Models learning curves during hyper-parameter optimization.

Decides whether to allocate more resources or to stop the training procedure for a particular configuration.

The learning process is terminated if the performance of the predicted configuration is less than the performance of the best model trained so far in the optimization process.
Multi-fidelity Optimization - Bandit-Based

- Successive halving algorithm (SHA)
- HyperBand
Train on small subsets, infer which regions may be interesting to evaluate in more depth.

[Hutter et al., Automated Machine Learning, 2019]
Multi-fidelity Optimization - SHA (1/4)

- **Train** on small subsets, infer which regions may be interesting to evaluate in more depth.
- Randomly sample candidates and evaluate on a small data sample.

[Hutter et al., Automated Machine Learning, 2019]
Multi-fidelity Optimization - SHA (1/4)

- **Train** on small subsets, infer which regions may be interesting to evaluate in more depth.
- Randomly sample candidates and evaluate on a small data sample.
- E.g., retrain the 50% best candidates on twice the data.

[Hutter et al., Automated Machine Learning, 2019]
Successive halving for eight algorithms/configurations.

After evaluating all algorithms on 1/8 of the total budget, half of them are dropped and the budget given to the remaining algorithms is doubled.

[Hutter et al., Automated Machine Learning, 2019]
SUCCESSIVE_HALVING (Finite horizon)

**Input:** Budget $B$, and $n$ arms where $\ell_{i,k}$ denotes the $k$th loss from the $i$th arm, maximum size $R$, $\eta \geq 2$ ($\eta = 3$ by default).

**Initialize:** $S_0 = [n]$, $s = \min\{t \in \mathbb{N} : nR(t + 1)\eta^{-t} \leq B, t \leq \log_\eta (\min\{R, n\})\}$.

For $k = 0, 1, \ldots, s$

- Set $n_k = \lfloor n\eta^{-k} \rfloor$, $r_k = \lfloor R\eta^{k-s} \rfloor$
- Pull each arm in $S_k$ for $r_k$ times.
- Keep the best $\lfloor n\eta^{-(k+1)} \rfloor$ arms in terms of the $r_k$th observed loss as $S_{k+1}$.

**Output:** $\hat{i}, \ell_{i,R}$ where $\hat{i} = \arg\min_{i \in S_{s+1}} \ell_{i,R}$
Successive halving suffers from the budget-vs-number of configurations trade off.
Successive halving suffers from the budget-vs-number of configurations trade off.

Given a total budget, the user has to decide beforehand whether:
Successive halving suffers from the budget-vs-number of configurations trade off.

Given a total budget, the user has to decide beforehand whether:

- to try many configurations and only assign a small budget to each, or
Successive halving suffers from the budget-vs-number of configurations trade off.

Given a total budget, the user has to decide beforehand whether:
  • to try many configurations and only assign a small budget to each, or
  • to try only a few and assign them a larger budget.
Successive halving suffers from the budget-vs-number of configurations trade off.

Given a total budget, the user has to decide beforehand whether:
- to try many configurations and only assign a small budget to each, or
- to try only a few and assign them a larger budget.

Assigning too small a budget can result in prematurely terminating good configurations.
Successive halving suffers from the budget-vs-number of configurations trade-off. Given a total budget, the user has to decide beforehand whether:
- to try many configurations and only assign a small budget to each, or
- to try only a few and assign them a larger budget.

Assigning too small a budget can result in prematurely terminating good configurations.

Assigning too large a budget can result in running poor configurations too long and thereby wasting resources.
HyperBand combats SHA problem when selecting from randomly sampled configurations.
HyperBand combats SHA problem when selecting from randomly sampled configurations.

It divides the total budget into several combinations of number of configurations vs. budget for each.
HyperBand combats SHA problem when selecting from randomly sampled configurations.

It divides the total budget into several combinations of number of configurations vs. budget for each.

Then it calls SHA on each set of random configurations.
The inner loop invokes SHA for fixed values of \( n \) and \( r \).

The outer loop iterates over different values of \( n \) and \( r \).
Neural Architecture Search (NAS)
Neural Architecture Search

- The process of automating architecture engineering.
Neural Architecture Search

- The process of automating architecture engineering.
- Search space: which architectures can be represented in principle.

[Hutter et al., Automated Machine Learning, 2019]
Neural Architecture Search

- The process of **automating architecture engineering**.
- **Search space**: which architectures can be represented in principle.
- **Search strategy**: how to explore the search space.

[Hutter et al., Automated Machine Learning, 2019]
Neural Architecture Search

- The process of automating architecture engineering.
- Search space: which architectures can be represented in principle.
- Search strategy: how to explore the search space.
- Performance estimation: to perform a standard training and validation of the architecture on data.

[Hutter et al., Automated Machine Learning, 2019]
Search Space
Search Space

- Which neural architectures a NAS approach might discover.
Search Space

- Which neural architectures a NAS approach might **discover**.
- **Chain-structured** neural network
Search Space

- Which neural architectures a NAS approach might discover.
- Chain-structured neural network
- Multi-branch networks
Search Space

- Which neural architectures a NAS approach might discover.
- Chain-structured neural network
- Multi-branch networks
- Repeated motifs
Chain-Structured Neural Network

- A sequence of $n$ layers.
Chain-Structured Neural Network

- A sequence of $n$ layers.
- The $i$’th layer $L_i$ receives its input from layer $i - 1$ and its output serves as the input for layer $i + 1$. 

Parameters of the search space:
- The (maximum) number of layers $n$.
- The type of operation every layer can execute, e.g., pooling, conv.
- Hyperparameters associated with the operation, e.g., number of filters, kernel size and strides for a convolutional layer.
Chain-Structured Neural Network

- A sequence of $n$ layers.
- The $i$’th layer $L_i$ receives its input from layer $i - 1$ and its output serves as the input for layer $i + 1$.
- Parameters of the search space:
A sequence of \( n \) layers.

The \( i \)’th layer \( L_i \) receives its input from layer \( i - 1 \) and its output serves as the input for layer \( i + 1 \).

Parameters of the search space:

- The (maximum) number of layers \( n \).
A sequence of \( n \) layers.

The \( i \)'th layer \( L_i \) receives its input from layer \( i - 1 \) and its output serves as the input for layer \( i + 1 \).

**Parameters** of the search space:
- The (maximum) number of layers \( n \).
- The type of operation every layer can execute, e.g., pooling, conv.
Chain-Structured Neural Network

- A sequence of $n$ layers.
- The $i$’th layer $L_i$ receives its input from layer $i-1$ and its output serves as the input for layer $i+1$.

Parameters of the search space:
- The (maximum) number of layers $n$.
- The type of operation every layer can execute, e.g., pooling, conv.
- Hyperparameters associated with the operation, e.g., number of filters, kernel size and strides for a convolutional layer.
Multi-Branch Networks

- The input of layer $i$: a function $g_i(L_{i-1}^{\text{out}}, \cdots, L_0^{\text{out}})$ of previous layer outputs.
Multi-Branch Networks

- The input of layer $i$: a function $g_i(L_{i-1}^{\text{out}}, \ldots, L_0^{\text{out}})$ of previous layer outputs.

- Special cases:
Multi-Branch Networks

- The input of layer $i$: a function $g_i(L_{i-1}^{\text{out}}, \cdots, L_0^{\text{out}})$ of previous layer outputs.

- Special cases:
  - The chain-structured networks: $g_i(L_{i-1}^{\text{out}}, \cdots, L_0^{\text{out}}) = L_{i-1}^{\text{out}}$
Multi-Branch Networks

- The input of layer $i$: a function $g_i(L_{\text{out}, i-1}, \cdots, L_{\text{out}, 0})$ of previous layer outputs.

- Special cases:
  - The chain-structured networks: $g_i(L_{\text{out}, i-1}, \cdots, L_{\text{out}, 0}) = L_{\text{out}, i-1}$
  - Residual networks, where previous layer outputs are summed: $g_i(L_{\text{out}, i-1}, \cdots, L_{\text{out}, 0}) = L_{\text{out}, i-1} + L_{\text{out}, i}$, $j < i$
Multi-Branch Networks

- The input of layer $i$: a function $g_i(L_{i-1}^{\text{out}}, \cdots, L_0^{\text{out}})$ of previous layer outputs.

- Special cases:
  - The chain-structured networks: $g_i(L_{i-1}^{\text{out}}, \cdots, L_0^{\text{out}}) = L_i^{\text{out}}$
  - Residual networks, where previous layer outputs are summed: $g_i(L_{i-1}^{\text{out}}, \cdots, L_0^{\text{out}}) = L_{i-1}^{\text{out}} + L_i^{\text{out}}, j < i$
  - DenseNets, where previous layer outputs are out concatenated: $g_i(L_{i-1}^{\text{out}}, \cdots, L_0^{\text{out}}) = \text{concat}(L_{i-1}^{\text{out}}, \cdots, L_0^{\text{out}})$
Repeated Motifs

- **Normal cell**: preserves the dimensionality of the input.
- **Reduction cell**: reduces the spatial dimension.
Search Strategy

Search Space $\mathcal{A}$ → Search Strategy → Performance Estimation Strategy

architecture $A \in \mathcal{A}$

performance estimate of $A$
Search Strategy

- Random search
- Reinforcement learning
- Gradient-based optimization
- Bayesian optimization
- Evolutionary methods
Random Search

- For each node in the DAG, determine what decisions must be made.

[Li et al., Random Search and Reproducibility for Neural Architecture Search, 2020]
Random Search

- For each node in the DAG, determine what decisions must be made.
  - Choose a node as input and a corresponding operation to apply to generate the output of the node.

[Li et al., Random Search and Reproducibility for Neural Architecture Search, 2020]
Random Search

- For each node in the DAG, determine what decisions must be made.
  - Choose a node as input and a corresponding operation to apply to generate the output of the node.
  - E.g., node $i$ can take the outputs of nodes 0 to node $i - 1$ as input.

[Li et al., Random Search and Reproducibility for Neural Architecture Search, 2020]
For each node in the DAG, determine what decisions must be made.

- Choose a node as input and a corresponding operation to apply to generate the output of the node.
- E.g., node $i$ can take the outputs of nodes 0 to node $i - 1$ as input.
- E.g., choose an operation, e.g., tanh, relu, sigmoid to apply to the output of node $i$.

[Li et al., Random Search and Reproducibility for Neural Architecture Search, 2020]
Random Search

- For each node in the DAG, determine what decisions must be made.
  - Choose a node as input and a corresponding operation to apply to generate the output of the node.
  - E.g., node \( i \) can take the outputs of nodes 0 to node \( i - 1 \) as input.
  - E.g., choose an operation, e.g., tanh, relu, sigmoid to apply to the output of node \( i \).

- Sample uniformly from the set of possible choices for each decision that needs to be made.

[Li et al., Random Search and Reproducibility for Neural Architecture Search, 2020]
Random Search

- For each node in the DAG, determine what decisions must be made.
  - Choose a node as input and a corresponding operation to apply to generate the output of the node.
  - E.g., node $i$ can take the outputs of nodes 0 to node $i - 1$ as input.
  - E.g., choose an operation, e.g., tanh, relu, sigmoid to apply to the output of node $i$.

- Sample uniformly from the set of possible choices for each decision that needs to be made.

- Moving from node to node.

[Li et al., Random Search and Reproducibility for Neural Architecture Search, 2020]
Evolutionary Methods

- Evolves a population of models, i.e., a set of (possibly trained) networks.
Evolutionary Methods

- Evolves a population of models, i.e., a set of (possibly trained) networks.
- In every evolution step, at least one model from the population is sampled and serves as a parent to generate offsprings by applying mutations to it.
Evolutionary Methods

- Evolves a population of models, i.e., a set of (possibly trained) networks.

- In every evolution step, at least one model from the population is sampled and serves as a parent to generate offsprings by applying mutations to it.
  - E.g., adding or removing a layer, altering the hyperparameters of a layer, adding skip connections, etc.
Evolutionary Methods

- Evolves a population of models, i.e., a set of (possibly trained) networks.

- In every evolution step, at least one model from the population is sampled and serves as a parent to generate offsprings by applying mutations to it.
  - E.g., adding or removing a layer, altering the hyperparameters of a layer, adding skip connections, etc.

- After training the offsprings, their fitness (e.g., performance on a validation set) is evaluated and they are added to the population.
Evolutionary Methods

- Evolves a population of models, i.e., a set of (possibly trained) networks.

- In every evolution step, at least one model from the population is sampled and serves as a parent to generate offsprings by applying mutations to it.
  - E.g., adding or removing a layer, altering the hyperparameters of a layer, adding skip connections, etc.

- After training the offsprings, their fitness (e.g., performance on a validation set) is evaluated and they are added to the population.

- Evolutionary methods differ in how they sample parents, update populations, and generate offsprings.
Action: the generation of a neural architecture.
Reinforcement Learning

- **Action**: the generation of a neural architecture.
- **Action space**: the search space.
Reinforcement Learning

- **Action**: the generation of a neural architecture.
- **Action space**: the search space.
- **Reward**: based on an estimate of the performance of the trained architecture on unseen data.
Reinforcement Learning

- **Action**: the generation of a neural architecture.
- **Action space**: the search space.
- **Reward**: based on an estimate of the performance of the trained architecture on unseen data.
- **Policy**: different approaches.
The previous methods search over a discrete set of candidate architectures.

[Liu et al., DARTS: Differentiable Architecture Search, 2019]
Gradient-based Optimization

- The previous methods search over a discrete set of candidate architectures.
- Here, it relaxes the search space to be continuous, so that the architecture can be optimized with respect to its validation set performance by gradient descent.

[Liu et al., DARTS: Differentiable Architecture Search, 2019]
Gradient-based Optimization

- The previous methods search over a discrete set of candidate architectures.
- Here, it relaxes the search space to be continuous, so that the architecture can be optimized with respect to its validation set performance by gradient descent.
- We relax the categorical choice of a particular operation to a softmax over all possible operations.

[Liu et al., DARTS: Differentiable Architecture Search, 2019]
Find the architecture $a \in A$ that maximizes $f(a)$. 
Bayesian Optimization (1/3)

- Find the architecture $a \in A$ that maximizes $f(a)$.
- Choose several architectures from $A$ at random and evaluating $f(a)$ for each of them.
Find the architecture $a \in A$ that maximizes $f(a)$.

Choose several architectures from $A$ at random and evaluating $f(a)$ for each of them.

Based on these results, iteratively choose new architectures to evaluate.
Find the architecture \( a \in A \) that maximizes \( f(a) \).

Choose several architectures from \( A \) at random and evaluating \( f(a) \) for each of them.

Based on these results, iteratively choose new architectures to evaluate.

The full algorithm: \( T \) rounds of choosing an architecture \( a_1 \) and computing \( f(a_1) \).
Find the architecture \( a \in A \) that maximizes \( f(a) \).

Choose several architectures from \( A \) at random and evaluating \( f(a) \) for each of them.

Based on these results, iteratively choose new architectures to evaluate.

The full algorithm: \( T \) rounds of choosing an architecture \( a_i \) and computing \( f(a_i) \).

The output is the architecture \( a^* \) with the largest value of \( f(a^*) \) among all those that were tried in the previous rounds.
Choose the next architecture in round $i + 1$, given $f(a_1), \ldots, f(a_i)$. 

Assume $f : A \rightarrow [0, 1]$ follows a Gaussian Process (GP).

The assumptions about the distribution $f(A)$ are constantly being updated as the algorithm gathers more data in the form of $f(a_1), \ldots, f(a_i)$.

Chooses the architecture with the greatest chance of giving a large improvement.

$f^*\text{ is the best accuracy observed so far.}$
Bayesian Optimization (2/3)

- Choose the next architecture in round $i + 1$, given $f(a_1), \ldots, f(a_i)$.

- Assume $f : A \to [0, 1]$ follows a Gaussian Process (GP).
Bayesian Optimization (2/3)

- Choose the next architecture in round $i + 1$, given $f(a_1), \cdots, f(a_i)$.

- Assume $f : A \to [0, 1]$ follows a Gaussian Process (GP).

- Makes an assumption about the distribution $f(A)$. 
Choose the next architecture in round $i + 1$, given $f(a_1), \cdots, f(a_i)$.

Assume $f : A \to [0, 1]$ follows a Gaussian Process (GP).

Makes an assumption about the distribution $f(A)$.

The assumptions about the mean and variance of $f(A)$ are constantly being updated as the algorithm gathers more data in the form of $f(a_1), \cdots, f(a_i)$. 

$f^*$ is the best accuracy observed so far.
Choose the next architecture in round $i + 1$, given $f(a_1), \cdots, f(a_i)$.

Assume $f : A \rightarrow [0, 1]$ follows a Gaussian Process (GP).

Makes an assumption about the distribution $f(A)$.

The assumptions about the mean and variance of $f(A)$ are constantly being updated as the algorithm gathers more data in the form of $f(a_1), \cdots, f(a_i)$.

Chooses the architecture with the greatest chance of giving a large improvement.
Choose the next architecture in round $i + 1$, given $f(a_1), \ldots, f(a_i)$.

Assume $f : A \rightarrow [0, 1]$ follows a Gaussian Process (GP).

Makes an assumption about the distribution $f(A)$.

The assumptions about the mean and variance of $f(A)$ are constantly being updated as the algorithm gathers more data in the form of $f(a_1), \ldots, f(a_i)$.

Chooses the architecture with the greatest chance of giving a large improvement.

The algorithm chooses $a_{i+1} = \arg \max_{a \in A} \max(0, E[f(a)] - f^*) = \arg \max_{a \in A} E[f(a)]$. 

$f^*$ is the best accuracy observed so far.
Choose the **next architecture** in round $i + 1$, given $f(a_1), \cdots, f(a_i)$.

Assume $f : A \to [0, 1]$ follows a **Gaussian Process (GP)**.

Makes an assumption about the **distribution** $f(A)$.

The assumptions about the **mean and variance** of $f(A)$ are **constantly being updated** as the algorithm gathers more data in the form of $f(a_1), \cdots, f(a_i)$.

Chooses the **architecture** with the **greatest chance** of giving a large improvement.

The algorithm chooses $a_{i+1} = \arg \max_{a \in A} \max(0, E[f(a) - f^*]) = \arg \max_{a \in A} E[f(a)]$.

$f^*$ is the **best accuracy** observed so far.
Bayesian Optimization (3/3)

- The top graph: three evaluations of $f$ (blue circles), an estimate of $f$ (solid red line), and confidence intervals (dotted red lines).

[https://medium.com/abacus-ai/an-introduction-to-bayesian-optimization-for-neural-architecture-search-d324830ec781]
The top graph: three evaluations of $f$ (blue circles), an estimate of $f$ (solid red line), and confidence intervals (dotted red lines).

The bottom graph: the expected improvement value for each architecture. The architecture with the largest expected improvement is chosen (blue $x$).
Performance Estimation
The search strategies need to estimate the performance of a given architecture \( A \) they consider.
The search strategies need to estimate the performance of a given architecture \( A \) they consider.

The simplest way of doing this is to train \( A \) on training data and evaluate its performance on validation data.
Performance Estimation

- The search strategies need to estimate the performance of a given architecture $A$ they consider.

- The simplest way of doing this is to train $A$ on training data and evaluate its performance on validation data.

- However, training each architecture to be evaluated from scratch frequently yields computational demands in the order of thousands of GPU days for NAS.
Reduce the Computational Burden

- Low-fidelity approximation
- Learning curve extrapolation
- One-shot architecture
Meta-Learning
Meta-Learning

- Meta-learning or learning to learn
Meta-Learning

- Meta-learning or learning to learn

- Systematically observe how different ML approaches perform on a wide range of learning tasks.
Meta-Learning

- **Meta-learning** or *learning to learn*

- Systematically **observe** how different ML approaches perform on a **wide range of learning tasks**.

- Then, **learning from this experience (meta-data)**, to learn **new tasks** much **faster** than otherwise possible.
Meta-Learning

- Learning from task properties
- Learning from model evaluation
- Learning from prior models
Meta-Learning

- Learning from task properties
  - Using meta-features
  - Building meta-models

- Learning from model evaluation

- Learning from prior models
Meta-Learning

- Learning from task properties
  - Using meta-features
  - Building meta-models

- Learning from model evaluation
  - Relative landmarks
  - Surrogate model
  - Warm-started multi-task learning

- Learning from prior models
Meta-Learning

- Learning from **task properties**
  - Using meta-features
  - Building meta-models

- Learning from **model evaluation**
  - Relative landmarks
  - Surrogate model
  - Warm-started multi-task learning

- Learning from **prior models**
  - Transfer learning
  - Few-shot learning
Learning from Task Properties
Learning from Task Properties (1/3)

- Uses meta-features:
  - Number of instances
  - Number of features
  - Statistical features (e.g., skewness, correlation, average, etc.)
  - Information theoretic features (e.g., the entropy of class labels)

The selection of meta-features is highly dependent on the application.
Learning from Task Properties (1/3)

- Uses **meta-features**: 
  - **Number of instances**
Learning from Task Properties (1/3)

- Uses **meta-features**:  
  - Number of instances  
  - Number of features
Learning from Task Properties (1/3)

- **Uses meta-features:**
  - Number of instances
  - Number of features
  - Statistical features (e.g., skewness, correlation, average, etc.)
Learning from Task Properties (1/3)

- Uses **meta-features**:
  - Number of instances
  - Number of features
  - Statistical features (e.g., skewness, correlation, average, etc.)
  - Information theoretic features (e.g., the entropy of class labels)
Uses meta-features:

- Number of instances
- Number of features
- Statistical features (e.g., skewness, correlation, average, etc.)
- Information theoretic features (e.g., the entropy of class labels)

The selection of meta-features is highly dependent on the application.
Each prior task $t_j$ is characterized by a meta-feature vector $m(t_j)$. 

Information from a prior task $t_j$ can be transferred to a new task $t_{new}$ based on their similarity. The similarity between two tasks is the distance between the feature vectors.
Learning from Task Properties (2/3)

- Each prior task $t_j$ is characterized by a meta-feature vector $m(t_j)$.

- Information from a prior task $t_j$ can be transferred to a new task $t_{\text{new}}$ based on their similarity.
Learning from Task Properties (2/3)

- Each prior task \( t_j \) is characterized by a meta-feature vector \( m(t_j) \).

- Information from a prior task \( t_j \) can be transferred to a new task \( t_{\text{new}} \) based on their similarity.

- The similarity between two tasks is the distance between the feature vectors.
Learning from Task Properties (3/3)

- Building meta-model.
Learning from Task Properties (3/3)

- Building meta-model.

- Building a meta-model $L$ to learn the relationships between meta-features of prior tasks $t_j$. 
Learning from Task Properties (3/3)

- Building meta-model.

- Building a meta-model $L$ to learn the relationships between meta-features of prior tasks $t_j$.

- For a new task $t_{\text{new}}$, the meta-model $L$ recommends the best configurations.
Learning from Prior Model Evaluation
Learning from Prior Model Evaluation (1/3)

- $t_j \in T$: $t_j$ is a ML task and $T$ is the set of all prior ML tasks.
Learning from Prior Model Evaluation (1/3)

- $t_j \in T$: $t_j$ is a ML task and $T$ is the set of all prior ML tasks.
- $\Theta$: the configuration space (hyper-parameter setting, pipeline components, etc.).
Learning from Prior Model Evaluation (1/3)

- $t_j \in T$: $t_j$ is a ML task and $T$ is the set of all prior ML tasks.
- $\Theta$: the configuration space (hyper-parameter setting, pipeline components, etc.).
- $P$: the set of all prior evaluations $P_{i,j}$ of configuration $\theta_i$ on a prior task $t_j$. 

Learn a meta-learner $L$ that is trained on meta-data $P \cup P_{\text{new}}$ to predict recommended configuration $\Theta_{\text{new}}$ for a new task $t_{\text{new}}$.

Three different ways:

1. Relative landmarks
2. Surrogate models
3. Warm-started multitask learning
Learning from Prior Model Evaluation (1/3)

- $t_j \in T$: $t_j$ is a ML task and $T$ is the set of all prior ML tasks.
- $\Theta$: the configuration space (hyper-parameter setting, pipeline components, etc.).
- $P$: the set of all prior evaluations $P_{1,j}$ of configuration $\theta_1$ on a prior task $t_j$.
- Learn a meta-learner $L$ that is trained on meta-data $P \cup P_{new}$ to predict recommended configuration $\theta_{new}$ for a new task $t_{new}$.
Learning from Prior Model Evaluation (1/3)

- $t_j \in T$: $t_j$ is a ML task and $T$ is the set of all prior ML tasks.
- $\Theta$: the configuration space (hyper-parameter setting, pipeline components, etc.).
- $P$: the set of all prior evaluations $P_{i,j}$ of configuration $\theta_i$ on a prior task $t_j$.
- Learn a meta-learner $L$ that is trained on meta-data $P \cup P_{\text{new}}$ to predict recommended configuration $\theta_{\text{new}}$ for a new task $t_{\text{new}}$.
- Three different ways:
  1. Relative landmarks
  2. Surrogate models
  3. Warm-started multitask learning
Relative landmarks measure the performance difference between two model configurations on the same task.
Relative landmarks measure the performance difference between two model configurations on the same task.

Two tasks $t_{\text{new}}$ and $t_j$ are considered similar, if their relative landmarks performance of the considered configurations are also similar.
Relative landmarks measure the performance difference between two model configurations on the same task.

Two tasks $t_{\text{new}}$ and $t_j$ are considered similar, if their relative landmarks performance of the considered configurations are also similar.

Once similar tasks have been identified, a meta-learner can be trained on the evaluations $P_{i,j}$ and $P_{i,\text{new}}$ to recommend new configurations for task $t_{\text{new}}$. 
Surrogate models get trained on all prior evaluations $P$ of all prior tasks $t_j$. 
Surrogate models get trained on all prior evaluations \( P \) of all prior tasks \( t_j \).

For a particular task \( t_j \), if the surrogate model can predict accurate configuration for a new task \( t_{\text{new}} \), then tasks \( t_{\text{new}} \) and \( t_j \) are considered similar.
Learning from Prior Models
Using transfer learning that utilizes pretrained models on prior tasks $t_j$ to be adapted on a new task $t_{new}$, where tasks $t_j$ and $t_{new}$ are similar.
Learning from Prior Models

- Using **transfer learning** that utilizes **pretrained models** on prior tasks $t_j$ to be adapted on a **new task** $t_{\text{new}}$, where tasks $t_j$ and $t_{\text{new}}$ are similar.

- E.g., **NN architecture and parameters** are trained on prior task $t_j$ that can be used as an **initialization** for model adaptation on a **new task** $t_{\text{new}}$. 
Learning from Prior Models

- Using **transfer learning** that utilizes **pretrained models** on **prior tasks** $t_j$ to be adapted on a **new task** $t_{\text{new}}$, where tasks $t_j$ and $t_{\text{new}}$ are similar.

- E.g., **NN architecture and parameters** are trained on **prior task** $t_j$ that can be used as an **initialization** for model adaptation on a **new task** $t_{\text{new}}$.

- Then, the **new model** can be **fine-tuned**.
Learning from Prior Models

- Using transfer learning that utilizes pretrained models on prior tasks $t_j$ to be adapted on a new task $t_{\text{new}}$, where tasks $t_j$ and $t_{\text{new}}$ are similar.

- E.g., NN architecture and parameters are trained on prior task $t_j$ that can be used as an initialization for model adaptation on a new task $t_{\text{new}}$.

- Then, the new model can be fine-tuned.

- Transfer learning usually works well when the new task to be learned is similar to the prior tasks.
BOHB: Robust and Efficient Hyperparameter Optimization at Scale
BOHB: Bayesian Optimization and Hyperband

- Bayesian optimization (BO): for choosing the configuration to evaluate
- Hyperband (HB): for deciding how to allocate budgets
Bayesian Optimization vs. Random Search

- BO advantage: much improved final performance
Hyperband vs. Random Search

- HB advantage: much improved anytime performance
Combining Bayesian Optimization and Hyperband

- **Best of both worlds**: strong anytime and final performance
HBOB Algorithm

- Relies on HB to determine how many configurations to evaluate with which budget.
HBOB Algorithm

- Relies on HB to determine how many configurations to evaluate with which budget.

- Replaces the random selection of configurations at the beginning of each HB iteration by a BO model-based search.
HBOB Algorithm

- Relies on HB to determine how many configurations to evaluate with which budget.

- Replaces the random selection of configurations at the beginning of each HB iteration by a BO model-based search.

- Once the desired number of configurations for the iteration is reached, the SHA procedure is carried out using these configurations.
A System for Massively Parallel Hyperparameter Tuning
SHA

▶ SHA allocates a small budget to each configuration, evaluate all configurations and keep the top $\frac{1}{\rho}$. 

It then increases the budget per configuration by a factor of $\rho$. Repeats until the maximum per-configuration budget of $R$ is reached.

SHA requires the number of configurations, a min and max resource, a reduction factor, and a minimum early-stopping rate.
SHA

- SHA allocates a small budget to each configuration, evaluate all configurations and keep the top $\frac{1}{\rho}$.

- It then increases the budget per configuration by a factor of $\rho$. 
SHA allocates a small budget to each configuration, evaluate all configurations and keep the top $\frac{1}{\rho}$.

It then increases the budget per configuration by a factor of $\rho$.

Repeats until the maximum per-configuration budget of $R$ is reached.
SHA allocates a small budget to each configuration, evaluate all configurations and keep the top $\frac{1}{\rho}$.

It then increases the budget per configuration by a factor of $\rho$.

Repeats until the maximum per-configuration budget of $R$ is reached.

SHA requires the number of configurations, a min and max resource, a reduction factor, and a minimum early-stopping rate.
Asynchronous SHA (ASHA)

- ASHA is a technique to parallelize SHA, leveraging *asynchrony* to mitigate stragglers and maximize parallelism.
Asynchronous SHA (ASHA)

- ASHA is a technique to parallelize SHA, leveraging asynchrony to mitigate stragglers and maximize parallelism.

- ASHA promotes configurations to the next rung whenever possible, instead of waiting for a rung to complete before proceeding to the next rung.
Asynchronous SHA (ASHA)

- ASHA is a technique to **parallelize SHA**, leveraging **asynchrony** to mitigate stragglers and maximize parallelism.

- ASHA **promotes configurations** to the next rung whenever possible, instead of waiting for a rung to complete before proceeding to the next rung.

- If **no promotions** are possible, ASHA simply **adds a configuration** to the base rung, so that more configurations can be promoted to the upper rungs.
Asynchronous SHA (ASHA)

- ASHA is a technique to parallelize SHA, leveraging asynchrony to mitigate stragglers and maximize parallelism.
- ASHA promotes configurations to the next rung whenever possible, instead of waiting for a rung to complete before proceeding to the next rung.
- If no promotions are possible, ASHA simply adds a configuration to the base rung, so that more configurations can be promoted to the upper rungs.
- Given its asynchronous nature it does not require the user to pre-specify the number of configurations to evaluate, but it otherwise requires the same inputs as SHA.
DARTS: Differentiable Architecture Search
Instead of searching over a discrete set of candidate architectures, we relax the search space to be continuous.

The architecture can be optimized with respect to its validation set performance by gradient descent.
Search Space

- It searches for a computation cell as the building block of the final architecture.
Search Space

- It searches for a computation cell as the building block of the final architecture.
- A cell is a DAG consisting of an ordered sequence of $N$ nodes.
Search Space

- It searches for a computation cell as the building block of the final architecture.
- A cell is a DAG consisting of an ordered sequence of $N$ nodes.
- Each node $x^{(i)}$ is a latent representation (e.g. a feature map in CNNs).
Search Space

- It searches for a **computation cell** as the **building block** of the final architecture.
- A **cell** is a **DAG** consisting of an ordered sequence of $N$ nodes.
- Each node $x^{(i)}$ is a **latent representation** (e.g. a feature map in CNNs).
- Each directed edge $(i, j)$ is associated with some operation $o^{(i,j)}$ that transforms $x^{(i)}$.  

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

- It searches for a **computation cell** as the **building block** of the final architecture.
- A **cell** is a **DAG** consisting of an ordered sequence of \( N \) nodes.
- Each node \( x^{(i)} \) is a **latent representation** (e.g. a feature map in CNNs).
- Each **directed edge** \((i, j)\) is associated with some operation \( o^{(i,j)} \) that transforms \( x^{(i)} \).
- Each **intermediate node** is computed based on all of its **predecessors**:
  \[
  x^{(j)} = \sum_{i < j} o^{(i,j)}(x^{i})
  \]
Let $O$ be a set of candidate operations, where each operation represents some function $o$ to be applied to $x^{(i)}$. To make the search space continuous, it relaxes the categorical choice of a particular operation to a softmax over all possible operations:

$$o(i, j)(x) = \frac{\sum_{o \in O} \exp(\alpha(i, j) o)}{\sum_{o' \in O} \exp(\alpha(i, j) o')}$$

The operation mixing weights for a pair of nodes $(i, j)$ are parameterized by a vector $\alpha(i, j)$ of dimension $|O|$. At the end of search, a discrete architecture can be obtained by replacing each mixed operation $o(i, j)$ with the most likely operation, i.e., $o(i, j) = \arg\max_{o \in O} \alpha(i, j) o$. 
Let $\mathcal{O}$ be a set of candidate operations, where each operation represents some function $\circ$ to be applied to $x^{(i)}$.

To make the search space continuous, it relaxes the categorical choice of a particular operation to a softmax over all possible operations:

$$\bar{o}^{(i,j)}(x) = \sum_{o \in \mathcal{O}} \frac{\exp(\alpha_{o}^{(i,j)})}{\sum_{o' \in \mathcal{O}} \exp(\alpha_{o'}^{(i,j)})} o(x)$$
Let $\mathcal{O}$ be a set of candidate operations, where each operation represents some function $o$ to be applied to $x^{(i)}$.

To make the search space continuous, it relaxes the categorical choice of a particular operation to a softmax over all possible operations:

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

The operation mixing weights for a pair of nodes $(i, j)$ are parameterized by a vector $\alpha^{(i,j)}$ of dimension $|\mathcal{O}|$. 

Continuous Relaxation and Optimization

- Let $\mathcal{O}$ be a set of candidate operations, where each operation represents some function $o$ to be applied to $x^{(i)}$.

- To make the search space continuous, it relaxes the categorical choice of a particular operation to a softmax over all possible operations:

  \[
  \bar{o}^{(i,j)}(x) = \sum_{o \in \mathcal{O}} \frac{\exp(\alpha^{(i,j)}_o)}{\sum_{o' \in \mathcal{O}} \exp(\alpha^{(i,j)}_{o'})} o(x)
  \]

- The operation mixing weights for a pair of nodes $(i, j)$ are parameterized by a vector $\alpha^{(i,j)}$ of dimension $|\mathcal{O}|$.

- At the end of search, a discrete architecture can be obtained by replacing each mixed operation $\bar{o}^{(i,j)}$ with the most likely operation, i.e., $o^{(i,j)} = \arg\max_{o \in \mathcal{O}} \alpha^{(i,j)}_o$. 
Summary
Summary

- Hyperparameter optimization
  - Black-box optimization
  - Multi-fidelity optimization

- Neural architecture search
  - Search space
  - Search strategy
  - Performance estimation

- Meta-learning
  - Learning from task properties
  - Learning from prior model evaluation
  - Learning from prior models
Reference

- Elshawi et al., Automated Machine Learning: State-of-The-Art and Open Challenges, 2019
- Falkner et al., BOHB: Robust and Efficient Hyperparameter Optimization at Scale, 2018
- Li et al., A System for Massively Parallel Hyperparameter Tuning, 2020
- Liu et al., DARTS: Differentiable Architecture Search, 2019
Questions?