Machine Learning Modeling Pipelines in Production

In the third course of Machine Learning Engineering for Production Specialization, you will build models for different serving environments; implement tools and techniques to effectively manage your modeling resources and best serve offline and online inference requests; and use analytics tools and performance metrics to address model fairness, explainability issues, and mitigate bottlenecks.

Understanding machine learning and deep learning concepts is essential, but if you're looking to build an effective AI career, you need production engineering capabilities as well. Machine learning engineering for production combines the foundational concepts of machine learning with the functional expertise of modern software development and engineering roles to help you develop production-ready skills.

Week 1: Neural Architecture Search

Contents

Neural Architecture Search

Neural Architecture Search

- Neural architecture search (NAS) is a technique for automating the design of artificial neural networks
- It helps finding the optimal architecture
- This is a search over a huge space
- AutoML is an algorithm to automate this search

- We'll get started with neural architecture search.
- We'll start with a discussion of something that you might be more familiar with, hyperparameter tuning. As I think you'll see, there are similarities between hyperparameter tuning and neural architecture search.
- Neural architecture search, or NAS, takes a prominent role in recent ML developments.
- In essence, it's a technique for **automating the design of neural networks**.
- Models found by NAS are often on par with or outperform hand designed architectures for many types of problem. The goal of NAS is to find the optimal architecture.
- Keep in mind that modern neural networks cover a huge parameter space.
- Automating the search with tools like AutoML makes a lot of sense.

Types of parameters in ML Models

- Trainable parameters:
  - Learned by the algorithm during training
  - e.g. weights of a neural network
- Hyperparameters:
  - set before launching the learning process
  - not updated in each training step
  - e.g: learning rate or the number of units in a dense layer

- Before taking a deep dive into AutoML, let's understand the problem it solves by analyzing one of the most tedious processes in ML modeling you've done naively, which is hyperparameter tuning.
- In machine learning models, there are two types of parameters. There are model parameters. Those are parameters that the model must learn using the training set.
- They're fitted or trained parameters of our models. Usually that means weights and biases.
- Then we have hyperparameters. These are adjustable parameters that must be tuned in order to create a model with optimal performance, but unlike model parameters, hyperparameters are not automatically optimized during the training process.
- They need to be set before model training begins, and they affect how the model trains.

Manual hyperparameter tuning is not scalable

- Hyperparameters can be numerous even for small models
- e.g shallow DNN:
  - Architecture choices
  - activation functions
  - Weight initialization strategy
  - Optimization hyperparameters such as learning rate, stop condition
- Tuning them manually can be a real brain teaser
- Tuning helps with model performance

- Hyperparameter tuning can have a high impact on model performance, and unfortunately, the number of hyperparameters can be substantial even for small models.
- For instance, in a shallow DNN, you can make hyperparameter choices for architecture options, for activation functions, for weight initialization strategy, and optimization hyperparameters, and others.
- Performing manual hyperparameter tuning can be a real brain teaser, which is a nice way to put it, since you're going to need to keep track of what you've tried and launch different experiments, so a manual process like that is tedious.
- Nonetheless, when done properly, hyperparameter tuning helps boost model performance significantly.

Automating hyperparameter tuning with Keras Tuner

- Automation is key: open source resources to the rescue
- Keras Tuner:
  - Hyperparameter tuning with Tensorflow 2.0.
  - Many methods available

- Several open source libraries have been created using various approaches to hyperparameter tuning. The Keras team has released one of the best, Keras Tuner, which is a library to easily perform hyperparameter tuning with TensorFlow 2.0.
- It offers a variety of different tuning methods, including random search, Hyperband, and Bayesian optimization. Let's take a look at a concrete example in the next video.

Setting up libraries and dataset

```
import tensorflow as tf
from tensorflow import keras
mnist = tf.keras.datasets.mnist


(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
```

Deep learning “Hello world!”

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(512, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

Model performance

```
Epoch 1/5
1875/1875 - 10s 5ms/step - loss: 0.3603 - accuracy: 0.8939
Epoch 2/5
1875/1875 - 10s 5ms/step - loss: 0.1001 - accuracy: 0.9695
Epoch 3/5
1875/1875 - 10s 5ms/step - loss: 0.0717 - accuracy: 0.9781
Epoch 4/5
1875/1875 - 10s 5ms/step - loss: 0.0515 - accuracy: 0.9841
Epoch 5/5
1875/1875 - 10s 5ms/step - loss: 0.0432 - accuracy: 0.9866
```

Parameters rational: if any

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(512, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```

Is this architecture optimal?

- Do the model need more or less hidden units to perform well?
- How does model size affect the convergence speed?
- Is there any trade off between convergence speed, model size and accuracy?
- Search automation is the natural path to take
- Keras tuner built in search functionality.

- You can train a model in just a few seconds, which is about 97% accurate on this dataset.
- As you can see when we run this, we end up with 0.9866 accuracy and each epoch takes a little over 10 seconds. But can we do better?
- The first question I usually get after showing this is where do these numbers come from? Why does this architecture have 512 neurons? Why not another number? And why does the dropout use 0.2, why not 0.5 or 0.1?
- Often, the answer is that I used a lot of trial and error to come up with these numbers. Or as is often the case, I just copy the code from a working example and didn't really think about it.
- Let's think about whether this is an attribute choice or not. The natural questions are will the model do better with more hidden units or less? Will it learn faster with less units? Can a smaller architecture do that without losing accuracy?
- You can experiment by changing it, rerunning it, changing it again and rerunning it again, et cetera. But that's a lot of work. What if you could automate this? So Keras tuner to the rescue.

Automated search with Keras tuner

```
# First, install Keras Tuner
!pip install -q -U keras-tuner

# Import Keras Tuner after it has been installed
import kerastuner as kt
```

Building model with iterative search

```
def model_builder(hp):
  model = keras.Sequential()
  model.add(keras.layers.Flatten(input_shape=(28, 28)))

  hp_units = hp.Int('units', min_value=16, max_value=512, step=16)
  model.add(keras.layers.Dense(units=hp_units, activation='relu'))
  model.add(tf.keras.layers.Dropout(0.2))
  model.add(keras.layers.Dense(10))

  model.compile(optimizer='adam',loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])
  return model
```

- Instead of hard coding 512 neurons into the hidden layer, we'll use this code instead. Notice that the number of units in the first dense layer is set to **hp.Int**.
- This will be tuned by the Keras tuner. So this sets up an integer value starting at 16 and going to 512 in steps of 16. Later you'll see how Keras tuner will run the model multiple times, gathering metrics each time and optimizing for the best one.
- In this case, the value is what drives the number of times Keras tuner does its thing.

Search strategy

```
tuner = kt.Hyperband(model_builder,
                     objective='val_accuracy',
                     max_epochs=10,
                     factor=3,
                     directory='my_dir',
                     project_name='intro_to_kt')
```

- Next, you'll define your search strategy. Keras tuner supports multiple strategies and you define which one you want to use like this.
- In this case, I'm using the **Hyperband** strategy.
- It also supports random search and Bayesian optimization and Sklearn strategies.
- The parameters you choose will vary based on your strategy. But the important one to note is the objective. In this case, our objective is val_accuracy, so we want to maximize on the validation accuracy.

Callback configuration

```
stop_early =
    tf.keras.callbacks.EarlyStopping(monitor='val_loss',
                                     patience=5)

tuner.search(x_train,
             y_train,
             epochs=50,
             validation_split=0.2,
             callbacks=[stop_early])
```

- Search can take a while to complete and use a lot of compute resources. But you can configure a an early stopping callback that stops the search when the conditions are met
- So for example, here I'm monitoring the validation loss and the **patience** is set to five, which means that it doesn't change significantly, or if it doesn't change significantly in five epochs, then stop searching on this iteration.
- You set the call back as a search parameter. The rest of the parameter specify how to search, such as the data and label, the number of epochs to train for, and the validation split.

Search output

```
Trial 24 Complete [00h 00m 22s]
val_accuracy: 0.3265833258628845
Best val_accuracy So Far: 0.5167499780654907
Total elapsed time: 00h 05m 05s
Search: Running Trial #25
Hyperparameter    |Value           |Best Value So Far
units             |192             |48
tuner/epochs      |10              |2
tuner/initial_e...|4               |0
tuner/bracket     |1               |2
tuner/round       |1               |0
tuner/trial_id    |a2edc917bda476c...|None
```

- As it searches, you'll see the results of each trial. We're only searching on one parameter, the units in the dense hidden layer.
- As you can see as it updates, you can keep track of the best value so far. Which at this point we have in the slide is 48, and in this case actually it ended up that way too when it completed.

Back to your model

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(48, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])
```

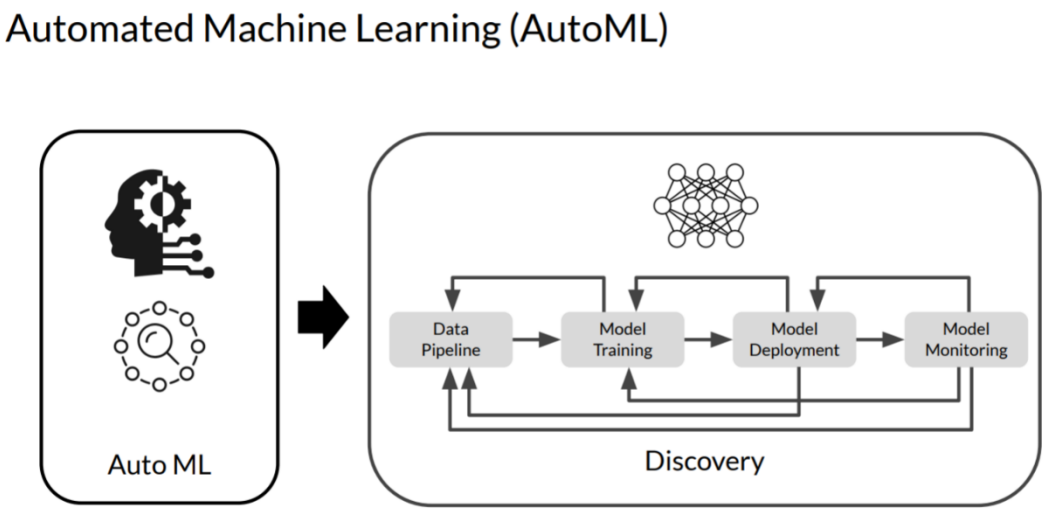
- So I can go back and try my architecture again and use the results of the Keras tuner to set the number of units manually, this time with 48 neurons in the layer, which is the number that we got from Keras tuner.

Training output

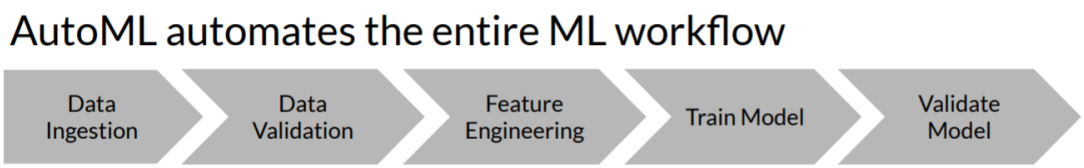
```
Epoch 1/5
1875/1875 - 3s 1ms/step - loss: 0.6427 - accuracy: 0.8090
Epoch 2/5
1875/1875 - 3s 1ms/step - loss: 0.2330 - accuracy: 0.9324
Epoch 3/5
1875/1875 - 3s 1ms/step - loss: 0.1835 - accuracy: 0.9448
Epoch 4/5
1875/1875 - 3s 1ms/step - loss: 0.1565 - accuracy: 0.9515
Epoch 5/5
1875/1875 - 3s 1ms/step - loss: 0.1393 - accuracy: 0.9564
```

- And when it's retrained, you'll see the results.
- It's only been five epochs and the value isn't quite as good as it was before hand, but our epochs are **three times quicker** than they were (3 seconds compared to the earlier 10 seconds per epoch)
- So I can maybe train for longer to get better results knowing that at the very least I've optimized part of my architecture.

Intro to AutoML

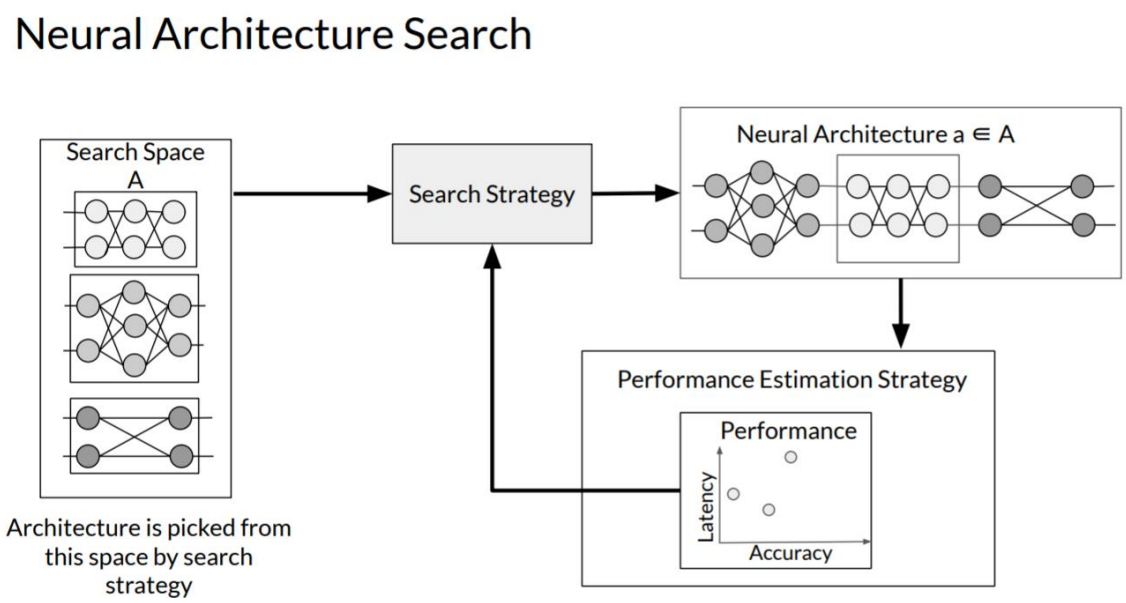


- So now that we've looked at hyper parameter tuning, let's take a look at actual AutoML.
- In this lesson, we'll be discussing AutoML, a set of very versatile tools to automate the machine learning process end to end.
- Finding the right model is an important piece of the puzzle.
- Neural Architecture Search is a process to help find relevant models. You'll be learning about search spaces and strategies which are key for both hyperparameter tuning and AutoML.
- We'll also discuss tools to quantify performance estimation on the explored models in your search.
- Finally, you learn about different AutoML offerings available in the Cloud by major service providers.
- Automated Machine Learning or AutoML is aimed at enabling developers with very little experience in machine learning, to make use of machine learning models and techniques.
- It tries to automate the process of machine learning end to end, in order to produce simple solutions, faster creation of those solutions and models that sometimes outperform even hand tuned models.
- AutoML applies machine learning and search techniques to the process of creating machine learning models and pipelines.
- It covers the complete pipeline from the raw data set to the deployable machine learning model.



- In traditional machine learning, we write code for all of the phases of the process.
- We start off with ingesting and cleansing the raw data and then perform feature selection and feature engineering.
- We select a model architecture for a task, train or model and perform hyper parameter tuning, maybe manually or using a tuner like Keras tuner and then we validate our models performance.
- ML requires a lot of manual programming and a highly specialized skill set.

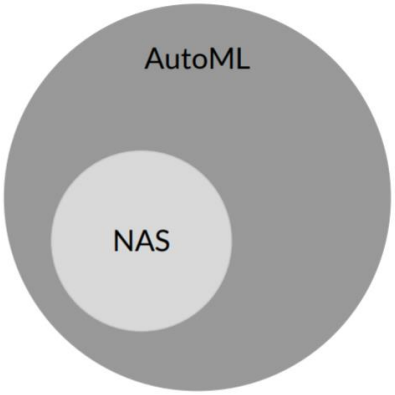
- Auto ML aims to automate the entire ML workflow. If we can provide the AutoML system with raw data and our model validation requirements, it goes through all of the phases in the ML workflow and performs the iterative process of ML development in a systematic way until the model is trained.



- Neural Architecture Search or NAS is at the heart of AutoML.
- There are **three main parts to Neural Architecture Search**: a search space, a search strategy and a performance estimation strategy.
- The search space defines the range of architectures which can be represented.
- To reduce the size of the search problem, we need to limit the search space to the architectures which are best suited to the problem that we're trying to model. This helps reduce the search space, but it also means that a **human bias** will be introduced, which might prevent Neural Architecture Search from finding architectural blocks that go beyond current human knowledge.
- The search strategy defines how we explore the search space. We want to explore the search based quickly, but this might lead to premature convergence to a sub optimal region in the search space.
- The objective of Neural Architecture Search is to find architecture is that perform well on our data.
- The performance estimation strategy helps in measuring and comparing the performance of various architectures.
- A search strategy selects an architecture from a predefined search space of architectures. The selected architecture is passed to a performance estimation strategy, which returns its estimated performance to the search strategy.
- The search space, search strategy and performance estimation strategy are the key components of Neural Architecture Search and will be discussing each of them in turn.

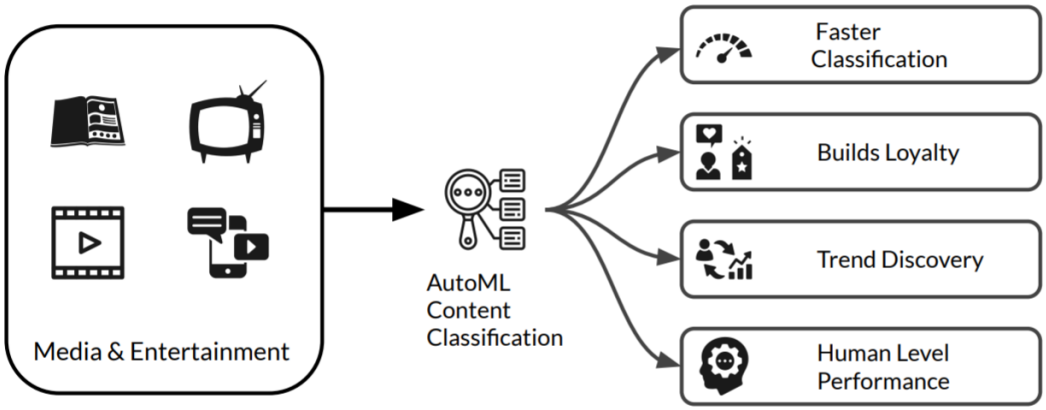
Neural Architecture Search

- **AutoML** automates the development of ML models
- **AutoML** is not specific to a particular type of model.
- Neural Architecture Search (**NAS**) is a subfield of AutoML
- NAS is a technique for automating the design of artificial neural networks (ANN).



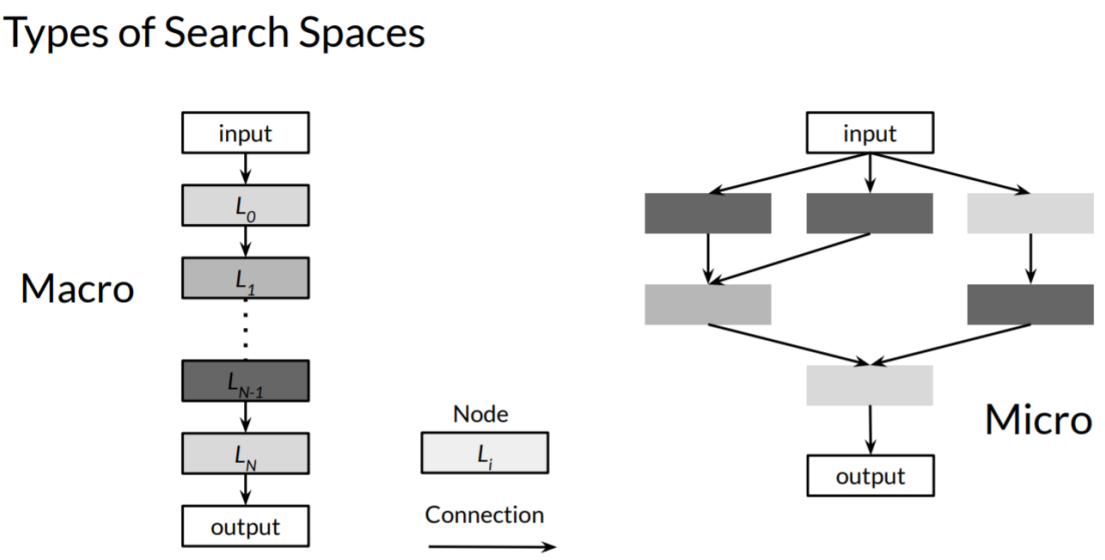
- AutoML facilitates building models in an automated fashion. Furthermore, AutoML is not restricted to a specific family or subset of models.
- **Neural Architecture Search is a sub field of AutoML.** It specifically focuses on the model selection part of the AutoML workflow and the design of neural networks. It has been used to design architectures that are on par with or outperform hand designed models.
- **It is important to note that NAS does NOT equate to AutoML.** NAS is the process of automating architecture engineering

Real-World example: Meredith Digital

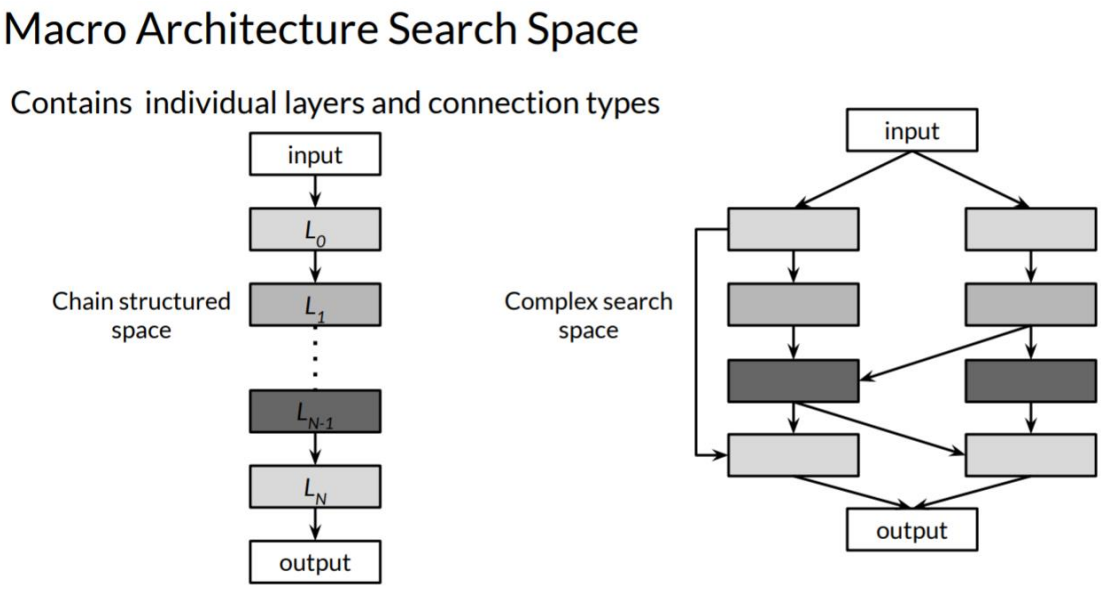


- Let's look at a real-world use of AutoML. Meredith Digital is a publishing company specializing in multiple formats of media and entertainment. Meredith Digital uses AutoML to train models, mostly **natural language based to automate content classification.**
- AutoML speeds up the classification process by reducing the process for months to just a few days.
- It also helps by providing insightful, actionable recommendations to help build customer loyalty.
- It helps by identifying new user trends and customer interests, to adapt content to better serve customers.
- To test its effectiveness, they conducted a test where they compared AutoML with their manually generated models and the results were pretty striking.
- The **Google Cloud AutoML natural language tools** provided content classification that was comparable to human level performance.

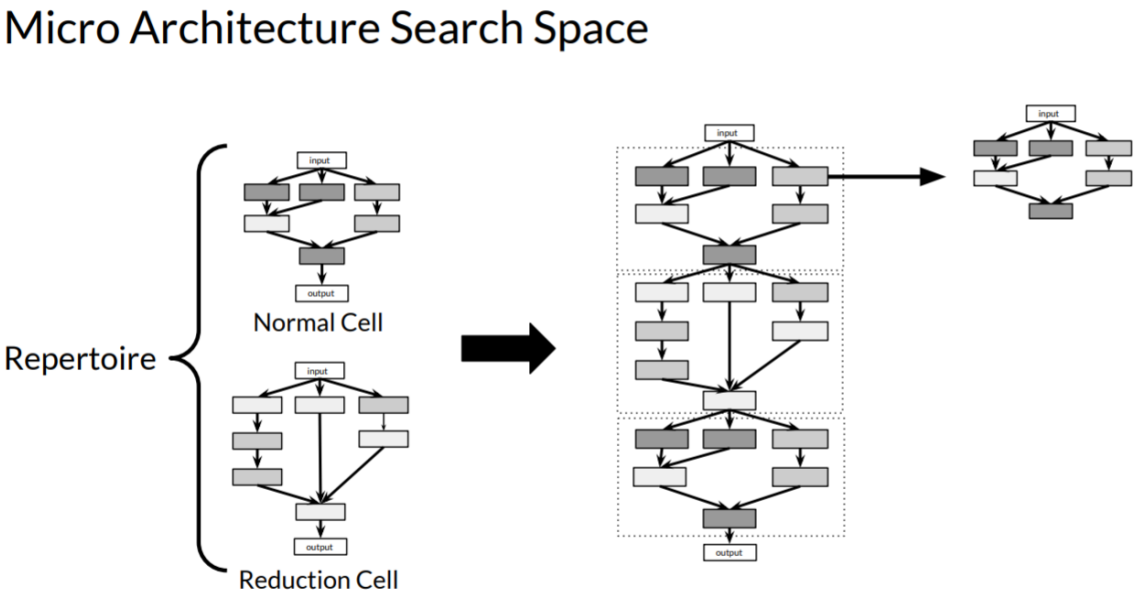
Understanding Search Spaces



- There are two main types of search spaces, macro and micro. And actually their names are kind of backwards, but that's what they're called.
- First, let's define what we mean by node.
- **A node is a layer in a neural network like a convolution or pooling layer.**
- In this illustration, each color represents a different layer type. An arrow from layer LI to LJ indicates that layer LJ receives the output of LI as input.

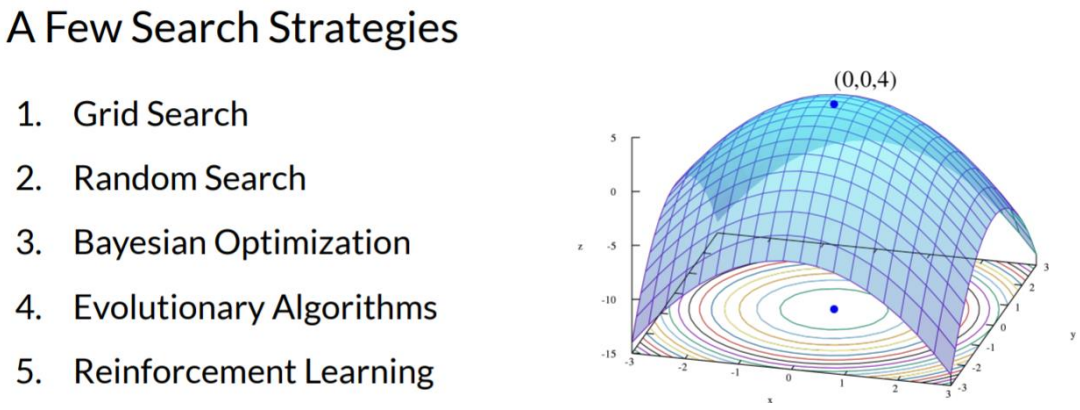


- A macro search space contains the individual layers and connection types of a neural network.
- Neural architecture search searches within that space for the best model, building the model layer by layer. As shown here, a network can be built very simply by **stacking** individual layers referred to as a chain structured space or with multiple branches and skip connections in a complex space.



- By contrast, in a micro search space, neural architecture search builds a neural network from cells where **each cell is a smaller network**.
- Here are two different types of cells, a normal cell on the top and a reduction cell on the bottom. Cells are stacked to produce the final network.
- **This approach has been shown to have significant performance advantages compared to a macro approach**.
- Here is an architecture built by stacking the cells sequentially. Note that cells can also be combined in a more complex manner, such as in multi branch spaces by simply replacing layers with cells.

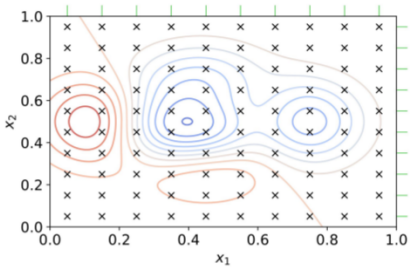
Search Strategies



- How does neural architecture search decide which options in the search space to try next? It needs to have a **search strategy**
- Neural architecture search searches through the search base for the architecture that produces the best performance. A variety of different approaches can be used for that search.
- These include grid search, random search, Bayesian optimization, evolutionary algorithms and reinforcement learning.

Grid Search and Random Search

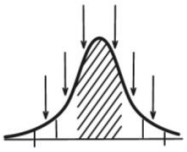
- Grid Search
  - Exhaustive search approach on fixed grid values
- Random Search
- Both suited for smaller search spaces.
- Both quickly fail with growing size of search space.

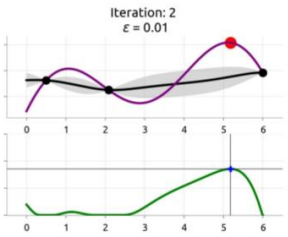


- In **grid search** you just search everything, which means you cover every option we have in the search base. In **random search**, you select the next option randomly within the search space.
- **Both of these work reasonably well in smaller search bases, but both also fail fairly quickly** when the search space grows beyond a certain size, which is all too common.

Bayesian Optimization

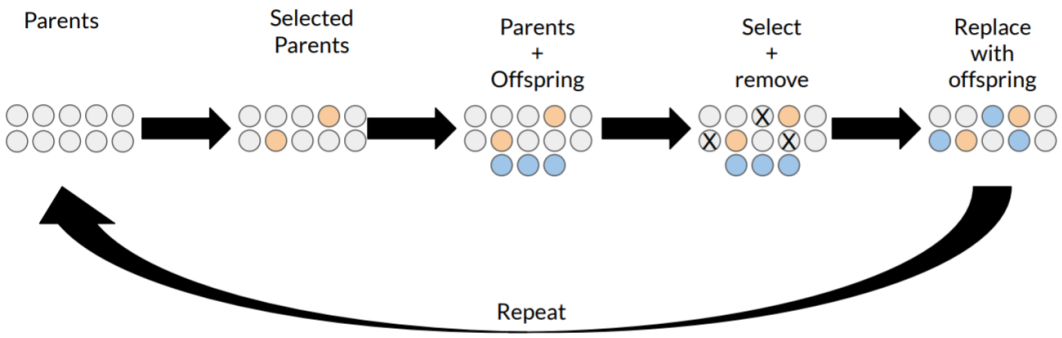
- Assumes that a *specific probability distribution*, is underlying the performance.
- Tested architectures constrain the probability distribution and guide the selection of the next option.
- In this way, promising architectures can be stochastically determined and tested.





- Bayesian optimization is a little more sophisticated. It assumes that a specific probability distribution, which is typically a **Gaussian distribution** is underlying the performance of model architectures.
- So you use observations from tested architectures to constrain the probability distribution and guide the selection of the next option.
- This allows us to **build up an architecture stochastically** based on the test results and the constrained distribution.

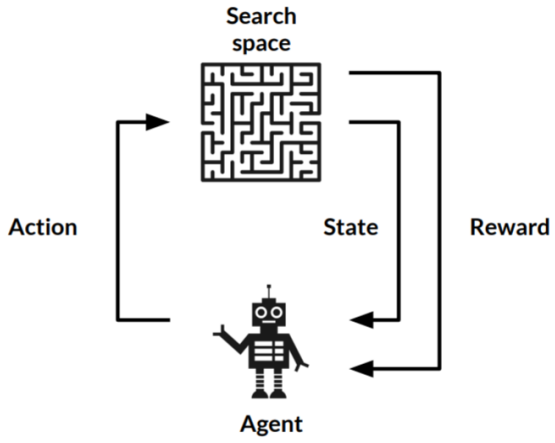
Evolutionary Methods



- Neural architecture search can also use an **evolutionary method** to search, and here's how it works.
- First, an initial population of n different model architecture is randomly generated. The performance of each individual, in other words, each architecture, is evaluated as defined by the performance estimation strategy
- Then the X highest performers are **selected as parents for a new generation**.
- This new generation of architectures might be copies of the respective parents with induced random alterations or mutations. Or they might arise from combinations of the parents, the performance of the offspring is assessed. Again using the performance estimation strategy.
- The list of possible mutations can include operations like adding or removing a layer, adding or removing a connection, changing the size of a layer or changing another hyper parameter.
- Y architectures are selected to be removed from the population. This might be the Y worst performers, the Y oldest individuals in the population, or a selection of individuals based on a combination of these parameters.
- The offspring replaces the removed architectures and the process is restarted with this new population.

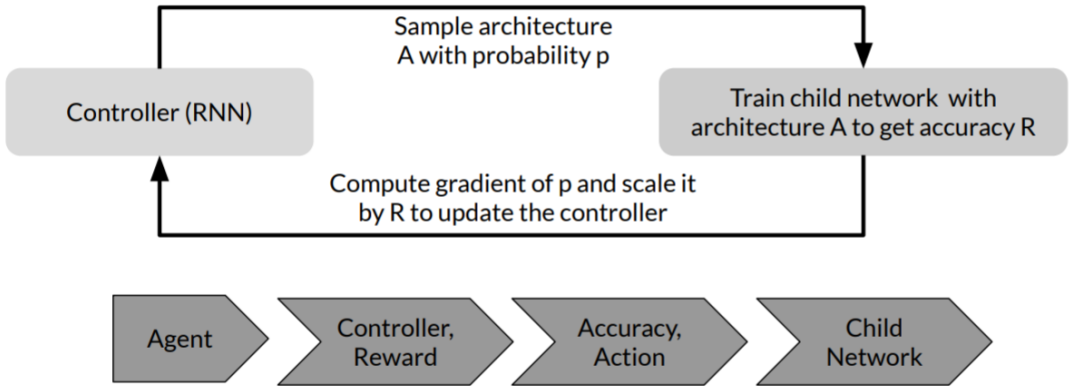
Reinforcement Learning

- Agents goal is to maximize a reward
- The available options are selected from the search space
- The performance estimation strategy determines the reward



- In **reinforcement learning,** agents take actions in an environment, trying to maximize a reward.
- After each action, the state of the agent and the environment is updated and a reward is issued based on a performance metric, then the range of possible next actions is evaluated
- The environment in this case is our search space, and the reward function is our performance estimation strategy.

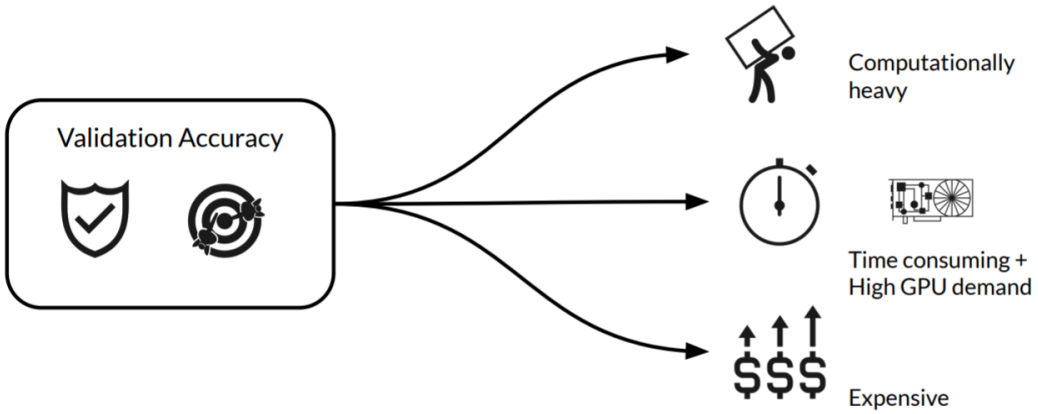
Reinforcement Learning for NAS



- A neural network can also be specified by a variable length string where the elements of the string specify individual network layers.
- That enables us to use a recurrent neural network or RNN to generate that string, as we might do for an NLP model.
- The RNN that generates the string is referred to as the **controller**
- After training the network referred to as the child network on real data, we can measure the accuracy on the validation set
- The accuracy determines the reinforcement learning reward in this case.
- Based on the accuracy, we can compute the **policy gradient** to update the controller RNN.
- In the next iteration, the controller will have learned to give higher probabilities to architectures that result in higher accuracies during training.
- This is how the controller will learn to improve its search over time,
- For example, on the CIFAR-10 data set, this method starting from scratch can design a new network architecture that rivals the best human designed architecture in terms of test set accuracy.

Measuring AutoML Efficacy

Performance Estimation Strategy



- Neural architecture search relies on being able to measure the accuracy or effectiveness of the different architectures that it tries. This requires a performance estimation strategy.

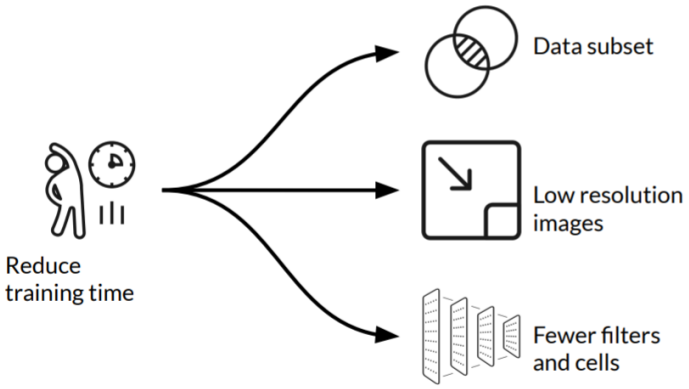
- The search strategies in neural architecture search need to estimate the performance of generated architectures so that they can generate architectures that perform better.
- Let's discuss some of the performance estimation strategies which are used in neural architecture search.
- The simplest approach is to measure the **validation accuracy** of each architecture that is generated, like we saw with the reinforcement learning approach.
- This becomes **computational heavy**, especially for large search spaces and complex networks. And as a result, it can take several GPU days to find the best architectures using this approach, which makes it expensive and slow. It also makes neural architecture search **impractical** for many use cases.

Strategies to Reduce the Cost

1. Lower fidelity estimates
2. Learning Curve Extrapolation
3. Weight Inheritance/ Network Morphisms

- Is there a way to reduce performance cost estimation? Several strategies have been proposed, including lower fidelity estimates, learning curve extrapolation, and weight inheritance or network morphism. Let's discuss each of these.

Lower Fidelity Estimates

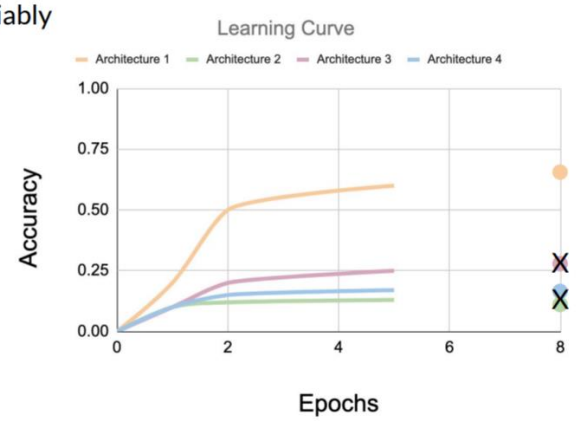


- Reduce cost but underestimates performance
- Works if **relative ranking** of architectures does not change due to lower fidelity estimates
- Recent research shows this is not the case

- Lower fidelity or lower precision estimates try to reduce the training time by reframing the problem to make it easier to solve by training on a **subset of data or using lower resolution images**, for example, or using **fewer filters per layer and fewer cells**.
- It greatly reduces the computational cost, but ends up underestimating performance. That's okay if you can make sure that the relative ranking of the architectures does not change due to their lower fidelity estimates.
- But unfortunately, recent research has shown that this is not the case, bummer. What else can we try?

Learning Curve Extrapolation

- Requires predicting the learning curve reliably
- Extrapolates based on initial learning.
- Removes poor performers



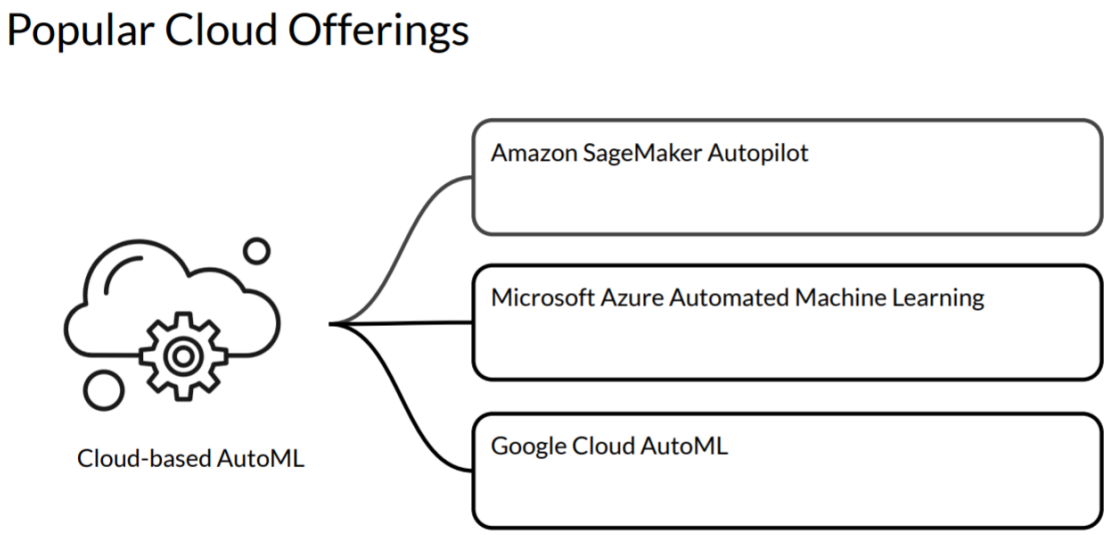
- Learning Curve Extrapolation is based on the assumption that you have mechanisms to **predict the learning curve reliably**, and so extrapolation is a sensitive and valid choice.
- On the right there are learning curves for different architectures. Based on a few iterations and available knowledge, the method extrapolates the initial learning curves and terminates all architectures that perform poorly.
- The progressive neural architecture search algorithm, which is one of the approaches for neural architecture search, uses a similar method by training a **surrogate model** and using it to predict the performance using architectural properties.

Weight Inheritance/Network Morphisms

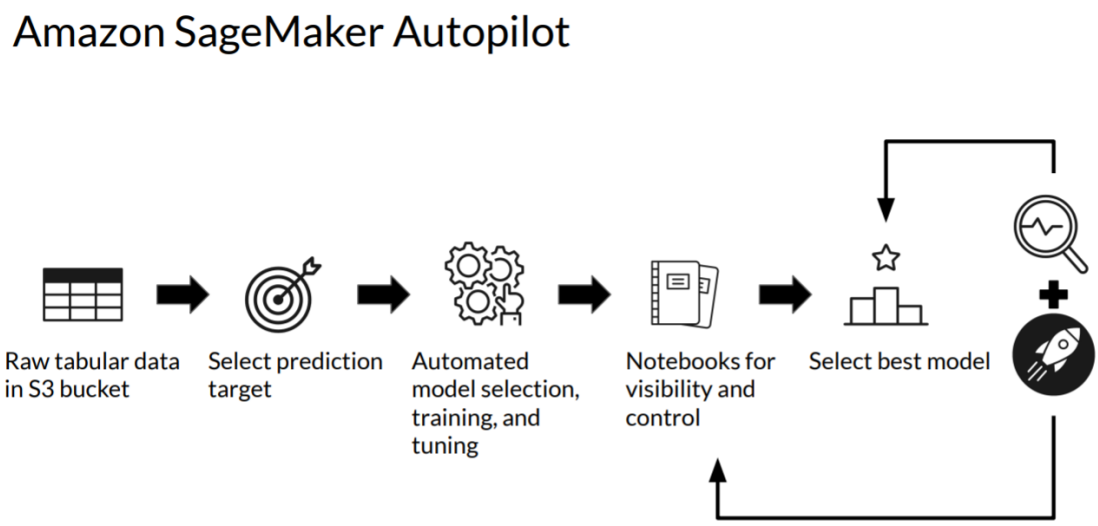
- Initialize weights of new architectures based on previously trained architectures
  - Similar to transfer learning
- Uses **Network Morphism**
- Underlying function unchanged
  - New network inherits knowledge from parent network.
  - Computational speed up: only a few days of GPU usage
  - Network size not inherently bounded

- Another method for speeding up architecture search initializes the weights of novel architectures based on the weights of other architectures that have been trained before, **similar to the way that transfer learning works**.
- One way of achieving this is referred to as network morphism.
- Network morphism modifies the architecture without changing the underlying function.
- This is advantageous because the network inherits knowledge from the parent network, which results in methods that require only a few GPU days to design and evaluate.
- This allows for increasing the capacity of network successively and retaining high performance **without requiring training from scratch.**
- One advantage of this approach is that it allows for search bases that don't have an inherent upper bound on the architecture's size.

AutoML on the Cloud

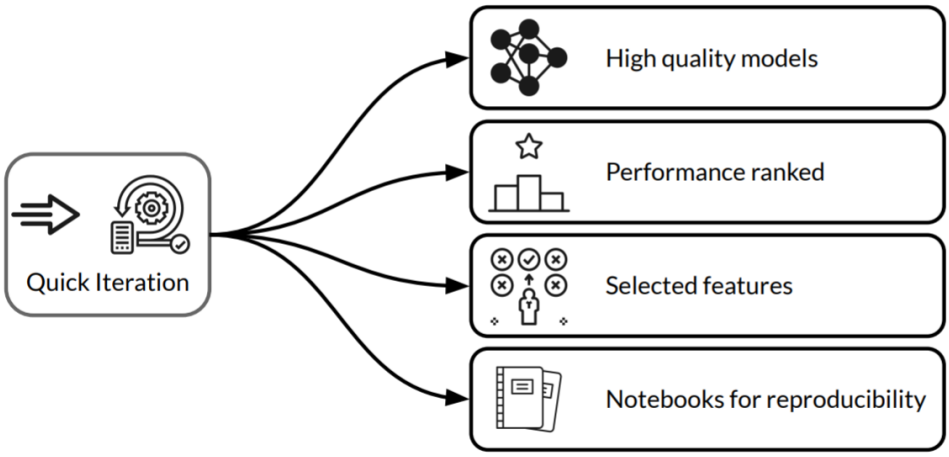


- One of the easiest ways to use AutoML is by using a Cloud service.
- We'll review three popular offerings; Amazon SageMaker Autopilot, Microsoft Azure Automated Machine Learning, and Google Cloud AutoML.
- In the exercises, you'll train a model using Google Cloud AutoML.



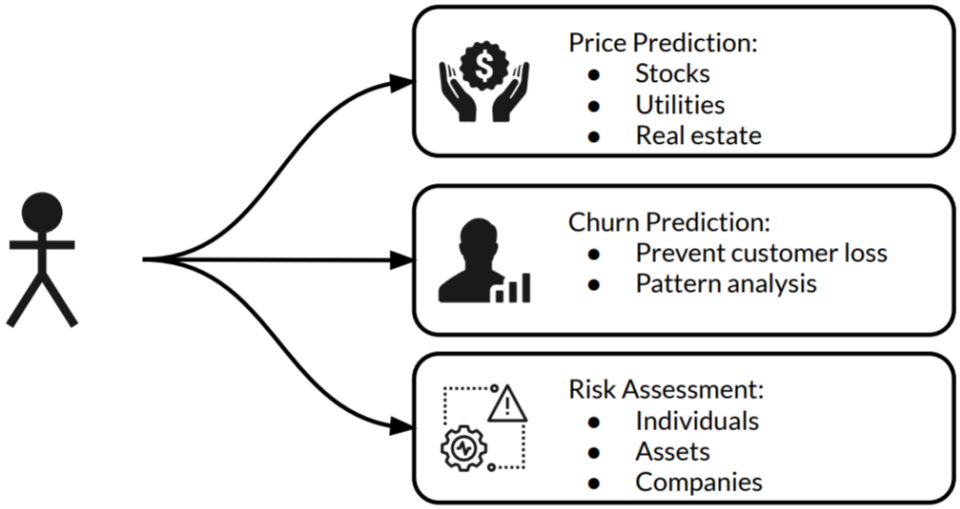
- Let's begin with Amazon SageMaker Autopilot.
- Amazon SageMaker Autopilot automatically trains and tunes the best machine learning models for classification or regression based on your data, while allowing you to maintain full control and visibility.
- Starting with your raw data, you select a label or target. Autopilot then searches for candidate models for you to review and choose from.
- All of these steps are **documented** on executable notebooks that give you full control and reproducibility of the process.
- This includes a leader board of model candidates to help you select the best model for your needs.
- You can then deploy the model to production or iterate on the recommended solutions to further improve model quality.

Key features



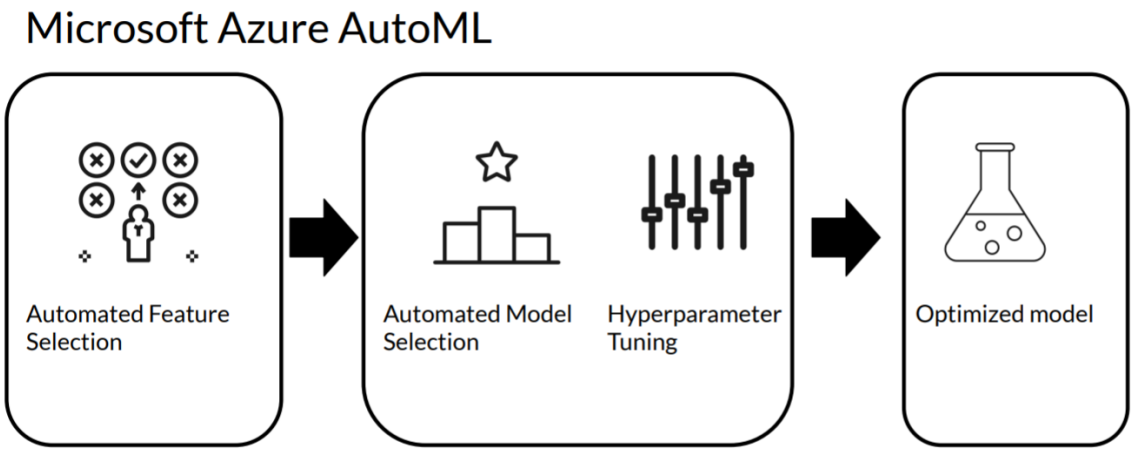
- Autopilot is optimized for quick iteration. It generates high-quality models quickly.
- After the initial set of iterations, autopilot creates a leaderboard of models ranked by performance. You can see which features in your dataset were selected by each model. You can then deploy a model to production.
- The process of generation of the models is completely transparent. Autopilot allows you to create a SageMaker notebook from any model it created.
- You can then check the Notebook to dive into details of the models implementations.
- If need be, you can refine the model and recreate it from the Notebook at any point in time.

Typical use cases

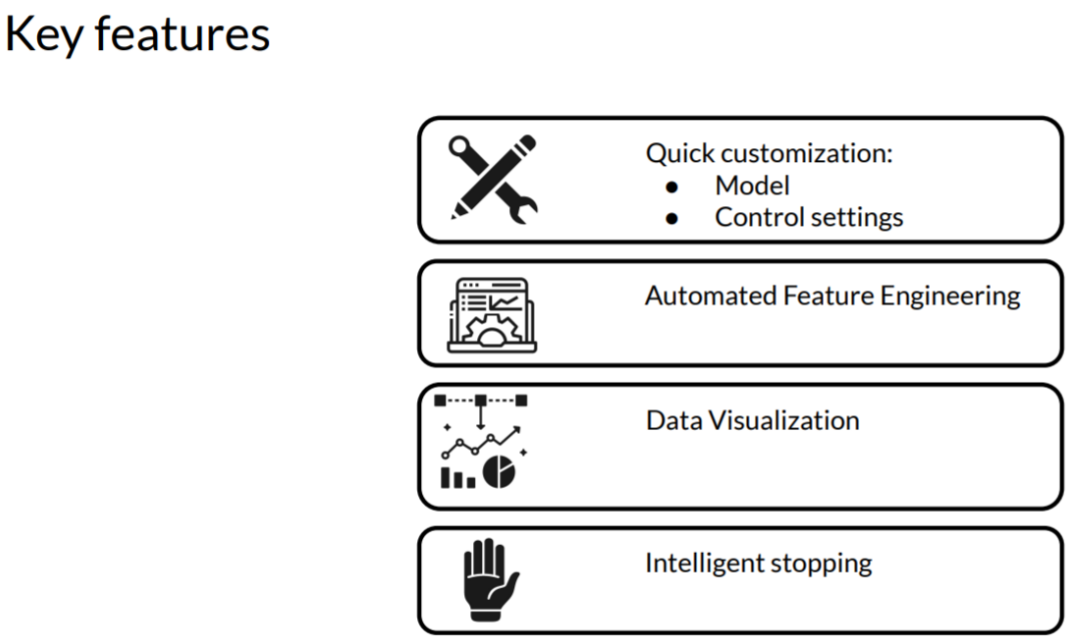


- Amazon suggests that autopilot can be applied for a number of different use cases.
- For example, it can be used to predict future prices to help you make sound investment decisions based on your historical data, such as demand, seasonal trends, and the price of other commodities.
- Price predictions are particularly useful in the financial services sector to predict the price of stocks or real estate, to predict real estate prices, or energy and utilities to predict the prices of natural resources.
- Churn prediction can be useful in predicting the loss of customers or churn. Companies are always looking for ways to eliminate churn.
- Churn prediction works by learning patterns in your existing data and identifying patterns in the new datasets so that the model can predict which customers are most likely to churn.
- Another application is risk assessment. Risk assessment requires identifying and analyzing potential events that may negatively impact individuals, assets, and your company.

- Risk assessment models are trained using your existing datasets so that you can optimize the models predictions for your business.

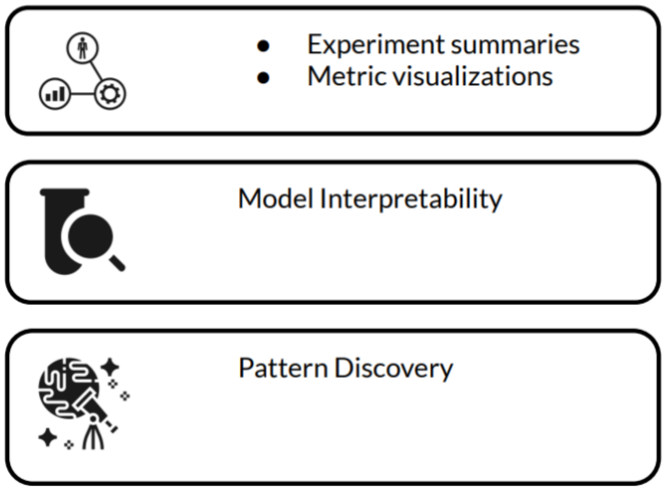


- Microsoft Azure Automated Machine Learning aims to empower professional and non-professional data scientists to build machine learning models rapidly.
- It automates time-consuming and iterative task of model development, so basically it does AutoML.
- It starts with automatic feature selection, followed by model selection and hyperparameter tuning on the selected model to generate the most optimized model for the task at hand.



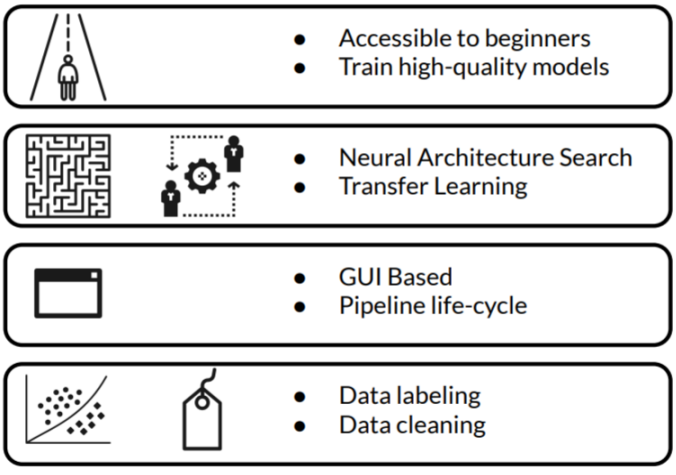
- You can either create your models using a no code UI or using code first notebooks.
- You can quickly customize your models, apply control settings to iterations, thresholds, validations, blocked algorithms, and other experimental criteria.
- It also provides tools to fully automate the feature engineering process.
- You can also easily visualize and profile your data to spot trends, as well as discover common errors and inconsistencies in your data.
- This helps you better understand the recommended actions and apply them automatically.
- It also provides intelligent stopping to save time on computing and prioritize the primary metric and subsampling to streamline experiment runs and speed results.

Key features



- It also has built-in support for experiment runs summaries, and detailed metrics visualizations to help you understand your models and compare model performance.
- Model interpretability helps evaluate model fit for raw and engineered features and provides insights into feature importance.
- You can discover patterns, perform what-if analysis, and develop deeper understanding of models to support transparency and trust in your business.

Google Cloud AutoML



- Google Cloud AutoML is a suite of machine learning products that enables developers with limited machine-learning expertise to train high-quality models specific to their business needs.
- It relies on Google, state of the art transfer learning and neural architecture search technologies.
- Cloud AutoML leverages more than 10 years of Google research to help your machine-learning models achieve faster performance and more accurate predictions.
- You can use Cloud AutoML, simple graphical user interface to train, evaluate, improve, and deploy models based on your data.
- You're really only a few minutes away from your own custom machine-learning model.
- Google's human labeling service can also put a team of people to work annotating or cleaning your labels to make sure that your models are being trained on high-quality data.

Cloud AutoML Products

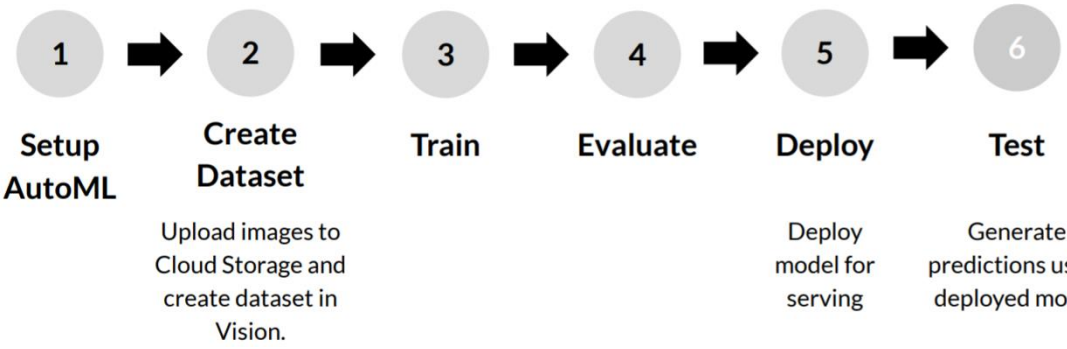
Sight	Auto ML Vision Derive insights from images in the cloud or at the edge.	Auto ML Video Intelligence Enable powerful content discovery and engaging video experiences.
Language	AutoML Natural Language Reveal the structure and meaning of text through machine learning.	Auto ML Translation Dynamically detect and translate between languages.
Structured Data	AutoML Tables Automatically build and deploy state-of-the-art machine learning models on structured data.	

- Because different problems and different data need to be treated differently, Cloud AutoML isn't just one thing.
- It's a suite of different products, each focused towards particular use cases and data types.
- For example, for image data, there's AutoML Vision and for video data, there's AutoML Video Intelligence, for natural language, there's AutoML natural language and for translation, there's AutoML Translation.
- Finally, for general structured data, there's AutoML Tables.

AutoML Vision Products

Auto ML Vision Classification	AutoML Vision Edge Image Classification
AutoML Vision Object Detection	AutoML Vision Edge Object Detection

Steps to Classify Images using AutoML Vision



- Some of these are broken down even further. For image data, for example, there's both vision and classification and Vision Object Detection.

- Then there are also Edge versions of both of these focused on optimizing for running inference at the edge in mobile applications or IoT devices.

AutoML Video Intelligence Products

AutoML Video Intelligence Classification

Enables you to train machine learning models, to classify shots and segments on your videos according to your own defined labels.

AutoML Video Object detection

Enables you to train machine learning models to detect and track multiple objects, in shots and segments.

- For video, there's both Video Intelligence classification and video object detection. Again, focused on these specific use cases.

So what's in the secret sauce?

How do these Cloud offerings perform AutoML?

- We don't know (or can't say) and they're not about to tell us
- The underlying algorithms will be similar to what we've learned
- The algorithms will evolve with the state of the art

- How do all three of these different Cloud services operate under the hood?
- Well, no one knows exactly or can't say if they work for a given provider.
- However, it is safe to assume that the algorithms at play will be similar to the ones that we've discussed in previous lessons.
- Also, keep in mind that ML Engineering for Production is a rapidly changing field and new technologies and developments emerge every few months.
- These new advancements are typically incorporated and exploited to their greatest extent by the AutoML providers.
- Also notice that these different AutoML offerings each perform the same operations that you perform or should perform as you train your model, including feature selection and feature engineering, and cleaning your labels. AutoML designers understand the importance of these activities.

References

- Neural Architecture Search
- Bayesian Optimization
- Neural Architecture Search with Reinforcement Learning
- Progressive Neural Architecture Search
- Network Morphism
- Amazon SageMaker Autopilot
- Microsoft Azure Automated Machine Learning
- Google Cloud AutoML