using post hoc methods. In addition, we introduced such techniques as feature importance, Shapley values, and Testing CAVs. Although this is a constantly evolving field (like nearly all of ML is), this chapter should have provided you with a solid foundation in model interpretation.

- The grayscale version of this plot in printed versions of this book won't show this, but when you're using the SHAP library, it will be displayed in color with red indicating a high feature value and blue a low feature value.

- You can find a full-color version of this plot online.

- You can find a full-color version of this plot online.

- You can find a full-color version of this plot online.

Chapter 10. Neural Architecture Search

Neural architecture search (NAS) is a technique for automating the design of neural networks. By running through a number of architecture permutations, NAS allows us to determine the most optimal architecture for a given problem. Models found by NAS are often on par with, or outperform, hand-designed architectures for many types of problems. It has recently been a very active area of both research and practical application.

The goal of NAS is to find an optimal model architecture. Keep in mind that modern neural networks cover a huge parameter space, so automating the search with tools like automated machine learning (AutoML) makes a lot of sense, but it can be very demanding of compute resources.

In this chapter, we will introduce techniques to optimize your ML models, starting with hyperparameter tuning, NAS, and AutoML. At the end of this chapter, we will introduce cloud services for AutoML.

Hyperparameter Tuning

Before taking a deep dive into NAS, let’s understand the problem it solves by analyzing one of the most tedious processes in ML modeling (if done naively): hyperparameter tuning. As we think you’ll see, there are similarities between hyperparameter tuning and NAS. We’re going to

assume that you’re already familiar with hyperparameter tuning, so we’re not going to go into great detail in this section. Rather, we will help you understand the similarities between hyperparameter tuning and NAS.

In ML models, there are two types of parameters:

Model parameters

These are the parameters in the model that must be determined using the training dataset. These are the fitted or trained parameters of our models, usually the weights and biases.

Hyperparameters

These are adjustable parameters that must be tuned to create a model with optimal performance. The tunable parameters can be things such as learning rate and layer types. 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.

Hyperparameter tuning is an iterative process in which you try one set of hyperparameters, train the model, check the model results on the test set, and then decide what to do next. You could make an adjustment to the hyperparameter settings and retrain the model to see if the results improve, or you could decide to stop the process and move forward with one of the

sets of hyperparameter settings that you tried. When using manual hyperparameter tuning, you do all of this by hand.

Hyperparameter tuning can also be automated, using one of several approaches to determine the next set of hyperparameters to be tried, and when to stop. The essential process is still the same—training the model repeatedly and checking the results—but since the process is automated, it is much less tedious for the developer to use. Often the choice of how to adjust the hyperparameters is based on an optimization approach, which can frequently make better choices than a random approach.

Hyperparameter tuning can have a big impact on a model's performance. Unfortunately, the number of hyperparameters can be substantial, even for small models. In a simple deep neural network (DNN), you can adjust various hyperparameters like architecture, activation functions, weight initialization, and optimization methods. Manual tuning can be overwhelming because you need to track numerous combinations and their results. An exhaustive search is often impractical, so hyperparameter tuning tends to rely on a developer's intuition. Nonetheless, when done properly, hyperparameter tuning can help boost model performance significantly.

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 that lets you easily perform hyperparameter tuning with TensorFlow 2.0. It provides various hyperparameter tuning techniques,

such as random search, Hyperband, and Bayesian optimization. Similar to hyperparameter selection, model architecture design can also be performed either manually or automatically.

Designing a model architecture is also an iterative process, requiring you to train the model and check the results. For a single model the design choices are many, including the number of layers, the width of each layer, the types of neurons, the activation functions, and the interconnect between layers. Just as automating hyperparameter tuning can make life easier for a developer, automating model design can also make life easier.

Introduction to AutoML

AutoML is a set of very versatile tools for automating the ML development process end to end, primarily focusing on the model architecture and parameters.

AutoML is aimed at enabling developers with very little experience in ML to make use of ML models and techniques. It tries to automate the process of ML development to produce simple solutions, create those solutions more quickly, and train models that sometimes outperform even hand-tuned models.

AutoML applies ML and search techniques to the process of creating ML models and pipelines. It covers the complete pipeline, from the raw dataset

to the deployable ML model. In traditional ML, we write code for all 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 our task, train our model, and perform hyperparameter tuning. Then we validate our model's performance. ML requires a lot of manual programming and a highly specialized skill set.

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

Key Components of NAS

NAS is at the heart of AutoML. There are three main parts to NAS: a search space, a search strategy, and a performance estimation strategy.

The *search space* defines the range of architectures that can be represented. To reduce the size of the search problem, we need to limit the search space to the architectures that are best suited to the problem we're trying to model. This helps reduce the search space, but it also means a human bias will be introduced, which might prevent NAS 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 space quickly, but this might lead to premature convergence to a suboptimal region in the search space.

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 estimate of the model's performance to the search strategy.

The search space, search strategy, and performance estimation strategy are the key components of NAS, and we'll discuss each of them in turn.

Search Spaces

There are two main types of search spaces, macro and micro, and actually their names are kind of backward, but that's what they're called. Let's look at both.

First, let's define what we mean by a node. A *node* is a layer in a neural network, like a convolution or pooling layer. In Figure 10-1, an arrow from layer L_0 to layer L_1 indicates that L_1 receives the output of L_0 as input.

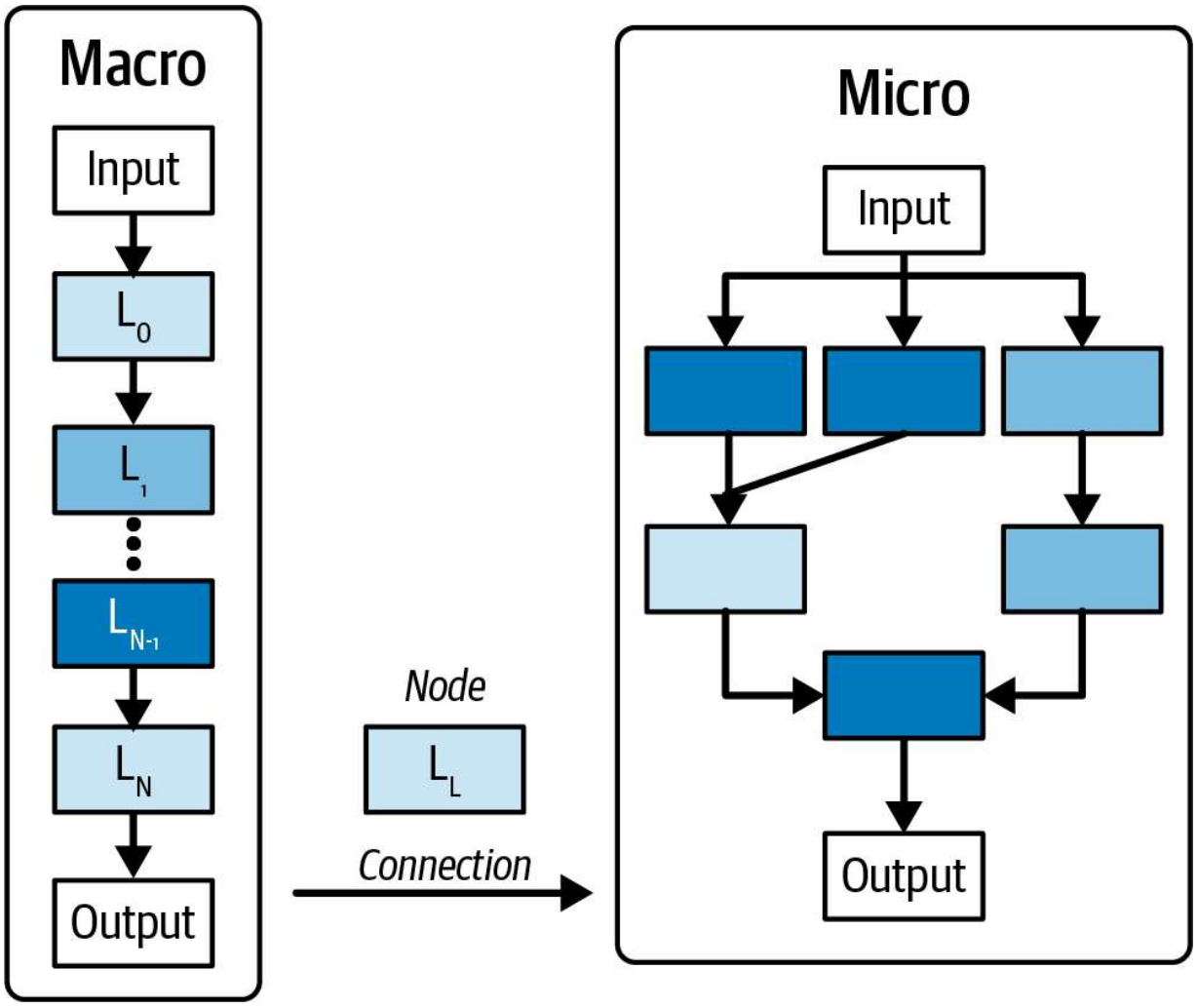

Figure 10-1. Search space types (reproduced from Elsken et al., 2019 with permission)

Macro search space

A macro search space contains the individual layers and connection types of a neural network, and NAS searches within that space for the best model, building the model layer by layer.

The number of possible ways to stack individual layers in a linear fashion defines a *chain-structured* search space, and the number of ways to stack

individual layers with multiple branches and skip connections defines a much larger *complex* search space.

As shown in Figure 10-2, a network can be built very simply by stacking individual layers in a chain-structured space, or with multiple branches and skip connections in a complex space.

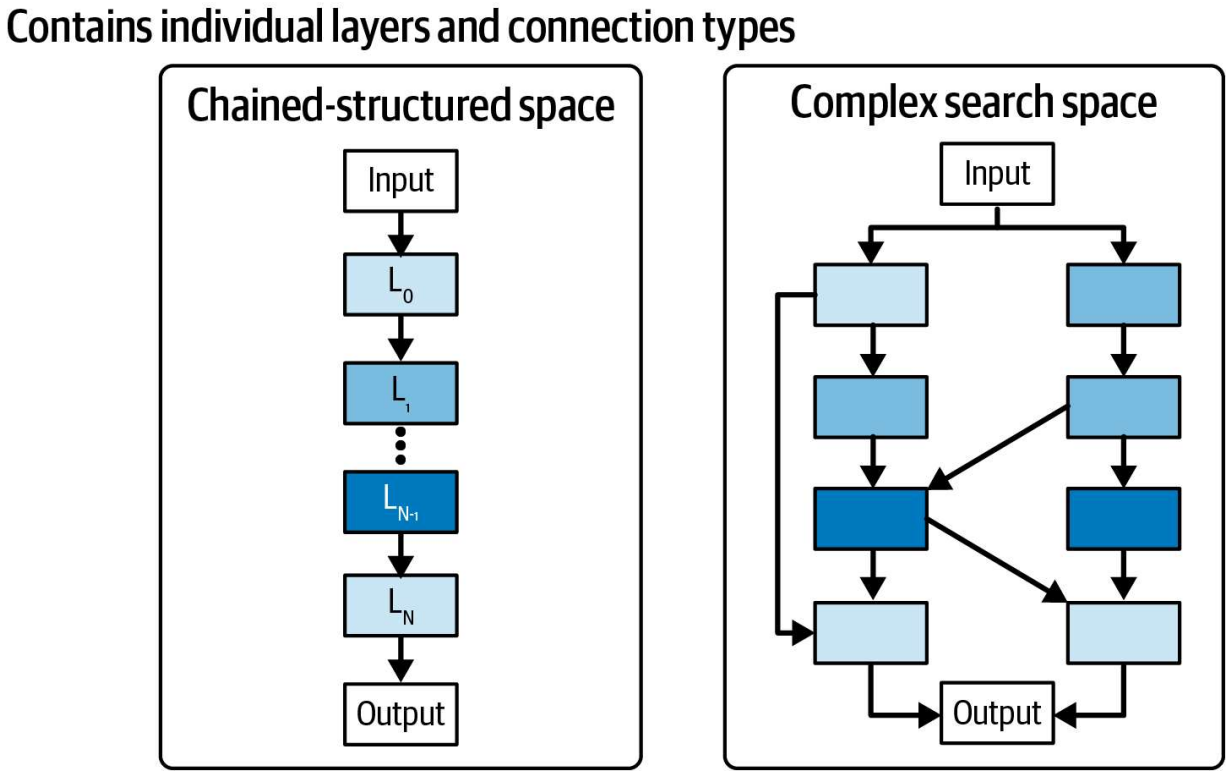

Figure 10-2. Macro search spaces (reproduced from Elsken et al., 2019 with permission)

Micro search space

By contrast, in a micro search space, NAS builds a neural network from cells, where each cell is a smaller network.

Figure 10-3 shows two different cell types, a normal cell (top) and a reduction cell (bottom). Cells are stacked to produce the final network. This approach has been shown to have significant performance advantages compared to a macro approach. The architecture shown on the right side of Figure 10-3 was built by stacking the cells sequentially. Note that cells can also be combined in a more complex manner, such as in multibranch spaces, by simply replacing layers with cells.

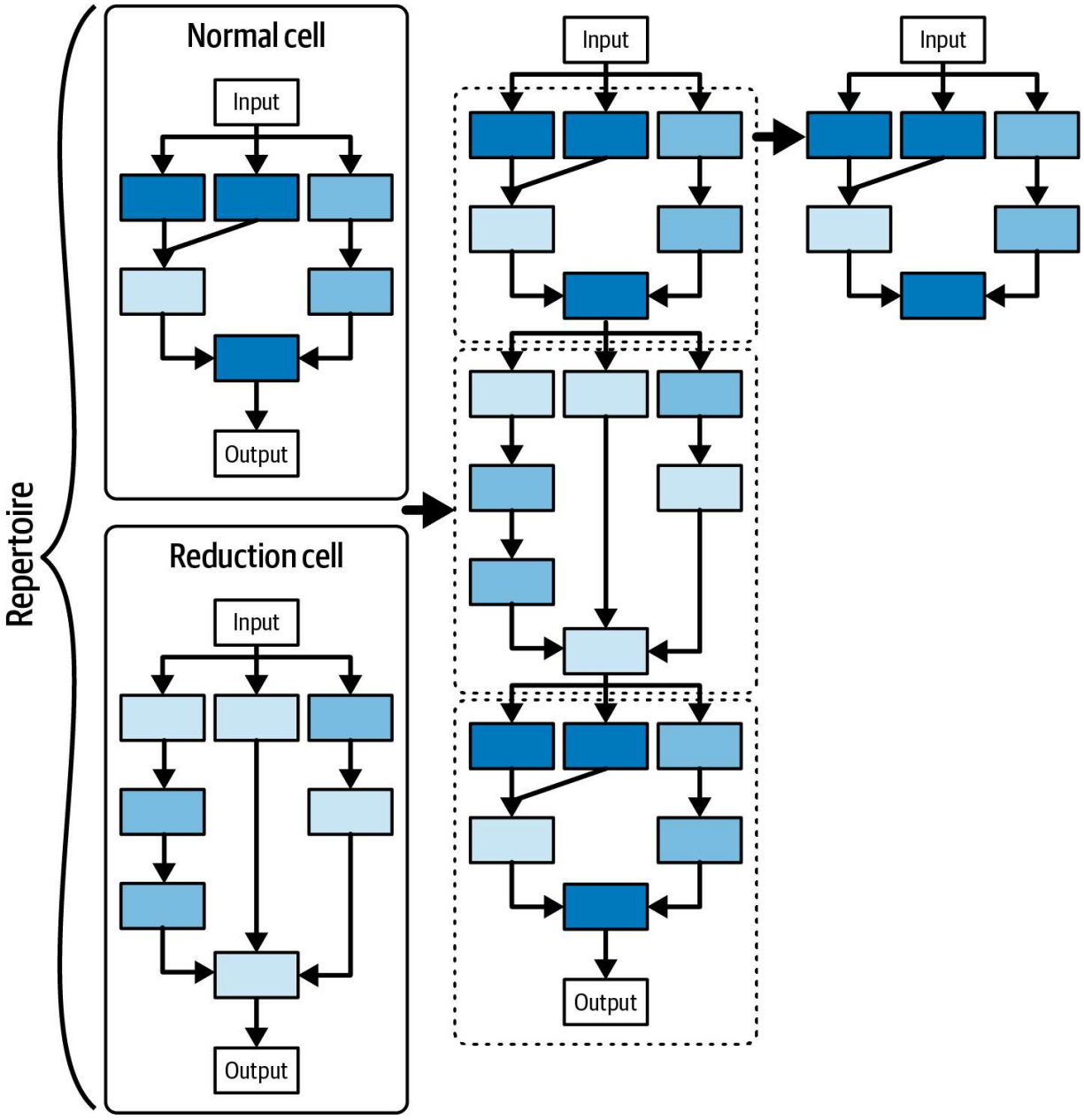

Figure 10-3. Micro search spaces (reproduced from Elsken et al., 2019 with permission)

Search Strategies

But how does NAS decide which options in the search space to try next? It needs to have a search strategy.

NAS searches through the search space for the architecture that produces the best performance. A variety of different approaches can be used to perform that search, including grid search, random search, Bayesian optimization, evolutionary algorithms, and reinforcement learning.

In *grid search*, you just search everything. That means you cover every combination of every option you have in the search space.

In *random search*, you select your next option randomly within the search space. Both grid search and random search work reasonably well in smaller search spaces, but both also fail fairly quickly when the search space grows beyond a certain size.

Bayesian optimization is a bit 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 you to build up an architecture stochastically, based on the test results and the constrained distribution.

NAS can also use an *evolutionary algorithm* to search. First, an initial population of N different model architectures is randomly generated. The performance of each individual (i.e., architecture) is evaluated, as defined by the performance estimation strategy (which we'll talk about in the next section).

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 such as adding or removing a layer, adding or removing a connection, changing the size of a layer, or changing another hyperparameter.

The 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 then replaces the removed architectures, and the process is restarted with this new population.

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.

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 (RNN) to generate that string, as we might

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 accuracy during training. This is how the controller will learn to improve its search over time. For example, on the CIFAR-10 dataset (an image dataset for image classification containing 10 different labels), this method, starting from scratch, can design a new network architecture that rivals the best human-designed architecture as measured by test set accuracy.

Performance Estimation Strategies

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

Simple approach to performance estimation

The simplest approach to performance estimation is to measure the validation accuracy of each architecture that is generated, as we saw with

the reinforcement learning approach. This becomes computationally 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. That makes it expensive and slow. It almost makes NAS impractical for many use cases.

More efficient performance estimation

Is there a way to reduce the cost of performance estimation? Several strategies have been proposed, including lower-fidelity estimates, learning curve extrapolation, weight inheritance, and network morphisms.

Lower-fidelity or *lower-precision estimates* try to reduce the training time by reframing the problem to make it easier to solve. There are various ways to do this, including:

- Training on a subset of the data
- Using lower-resolution images
- Using fewer filters per layer and fewer cells

This strategy reduces the computational cost considerably, but it ends up underestimating performance. That’s OK if you can make sure the relative ranking of the architectures does not change due to lower-fidelity estimates, but unfortunately, recent research has shown that this is not the case. Bummer. What else can we try?

Learning curve extrapolation is based on the assumption that you have mechanisms to predict the learning curve reliably, and so extrapolation is a reasonable choice. Based on a few iterations and available knowledge, the method extrapolates initial learning curves and terminates all architectures that performed poorly. The Progressive Neural Architecture Search (PNAS) algorithm, which is one of the approaches for NAS, uses a similar method by training a surrogate model and using it to predict the performance using architectural properties.

Weight inheritance is another approach for speeding up architecture search. It starts by initializing the weights of new architectures based on the weights of other architectures that have been trained before (similar to the way transfer learning works).

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. Network morphism allows for increasing the capacity of networks successively, and retaining high performance without requiring training from scratch. One advantage of this approach is that it allows for search spaces that don’t have an inherent upper bound on the architecture’s size.

AutoML in the Cloud

Probably the easiest way to use AutoML is by using one of the growing number of cloud services that are available. To illustrate, we'll review a few popular choices. (Note that these services are evolving quickly, so there's a fairly good chance that these descriptions may be out of date by the time you read this. Nevertheless, this should give you some idea of the types of services available.)

Amazon SageMaker Autopilot

Amazon SageMaker Autopilot automatically trains and tunes ML models for classification or regression, based on your data, while allowing you to maintain control and visibility. Starting with your raw data, you identify the label, or target, in your dataset. Autopilot then searches for candidate models for you to review and choose from.

All of these steps are documented with executable notebooks that give you control and reproducibility of the process. This includes a leaderboard of model candidates to help you select the best model for your needs. You then can deploy the model to production, or iterate on the recommended solutions to further improve the model quality.

Autopilot is optimized for quick iteration. After the initial set of iterations, Autopilot creates the leaderboard of models, ranked by performance. You

can see which features in your dataset were selected by each model, and then deploy a model to production. 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 model's implementation, and if need be, you can refine the model and re-create it from the notebook at any point in time.

Autopilot offers a versatile range of applications. It can project future prices, empowering you to make well-informed investment decisions rooted in historical data such as demand, seasonality, and the prices of related commodities. The ability to predict prices proves particularly valuable in:

- Financial services, for anticipating stock prices
- Real estate, for forecasting property values
- Energy and utilities, for predicting the prices of natural resources

Churn prediction aids in forecasting customer turnover by recognizing patterns in past data and using those insights to identify customers at a greater risk of churning in new datasets.

Another application is risk evaluation, which involves recognizing and analyzing potential events that could adversely affect individuals, assets, and the organization. Risk assessment models are developed using historical data to enhance their predictive accuracy for your specific business context.

Microsoft Azure Automated Machine Learning

Microsoft Azure Automated Machine Learning automates the time-consuming and iterative tasks of model development.

It starts with automatic feature selection, followed by model selection and hyperparameter tuning on the selected model. You can create your models by using a no-code UI or by using code-first notebooks. You can quickly customize your models, applying control settings to iterations, thresholds, validations, blocked algorithms, and other experimental criteria. You also have access to tools to fully automate the feature engineering process.

You can easily visualize and profile your data to spot trends and discover common errors and inconsistencies in your data. This helps you better understand recommended actions and apply them automatically. Microsoft Azure Automated Machine Learning also provides intelligent stopping to save time on computing, and subsampling to reduce the cost of generating results. In addition, it has built-in support for experiment run summaries and detailed visualizations of metrics to help you understand your models and compare model performance.

Model interpretability helps evaluate model fit for raw and engineered features, and it provides insights into feature importance. You can discover patterns, perform what-if analyses, and develop a deeper understanding of your models to support transparency and trust in your business.

Google Cloud AutoML

Google Cloud AutoML is a suite of ML products that enable developers with limited ML expertise to train high-quality models specific to their business needs. It relies on Google's state-of-the-art transfer learning and NAS technologies. Cloud AutoML leverages more than 10 years of Google research to help your ML models achieve faster performance and more accurate predictions.

You can use the simple GUI in Cloud AutoML to train, evaluate, improve, and deploy models based on your data. Google's human labeling service can also put a team of people to work annotating and/or cleaning your labels to make sure your models are being trained on high-quality data.

Because different kinds of problems and different kinds of data need to be treated differently, Cloud AutoML isn't just one thing. It's a suite of different products, each focused on 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.

Some of these are broken down even further. For image data, for example, there's both Vision Classification and Vision Object Detection. And then there are Edge versions of both of these, focused on optimizing for running

inference at the edge, in mobile applications or Internet of Things (IoT) devices. For video there's both Video Intelligence Classification and Video Object Detection, again focused on these specific use cases.

GOOGLE CLOUD AUTOML EXAMPLE: MEREDITH DIGITAL

Let's consider 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 model development process from months to just a few days. It also helps by providing insightful, actionable recommendations to help build customer loyalty, and it identifies new user trends and customer interests to adapt content to better serve customers.

To test its effectiveness, Meredith Digital conducted a test that compared AutoML with its 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.

Using AutoML

How do all three of these different cloud services operate under the hood? Since these are proprietary technologies, the details are not available, but it

is safe to assume that the algorithms being used will be similar to the ones we've discussed. However, in some sense it really doesn't matter what they do under the hood. What matters are the results.

So, how and when should you consider using AutoML? If you are either working on a new model or evaluating an existing model to see if you can improve it, a good first step is to use one or more of the cloud-based AutoML services and examine the results. This will at least give you a baseline. You can then work on adjusting parameters to see how much you can improve those results, and consider whether you think you can do better with a custom model. AutoML may or may not give you an acceptable result, but it's very likely to give you a better result in less time than it would take you to create a baseline model by hand. That gives you the option of using the AutoML model as a temporary solution while you work on a custom model, if you decide you think you can do better with a custom model.

Generative AI and AutoML

The AutoML technologies we've focused on in this chapter predate the explosion of generative AI (GenAI) technologies, including coding-focused large language models (LLMs). At the time of this writing, the use of GenAI to create model architectures is not yet robust or well established, but given the pace of progress in GenAI and the similarities between

creating model architectures and other kinds of code development, we can expect that at some point there will be GenAI approaches that will produce better results than AutoML approaches. However, we should not expect AutoML technology to stand still. Such is the nature of technology. We encourage you to monitor advancements in GenAI, especially in the domain of coding and model architecture design. Of course, we also encourage you to monitor advancements in AutoML as well.

Conclusion

In this chapter, we discussed the field of AutoML, and especially neural architecture search. In many ways, these technologies are fundamentally different from the rest of ML in that the goals are to use search techniques to design new models, rather than creating or using a model to achieve a result. In a production setting, when designing a new model is a goal, these techniques can often achieve that goal more quickly than having an ML engineer or data scientist design a new model. Alternatively, they can provide a baseline or a starting point from which an ML engineer or data scientist can design a better-performing model.