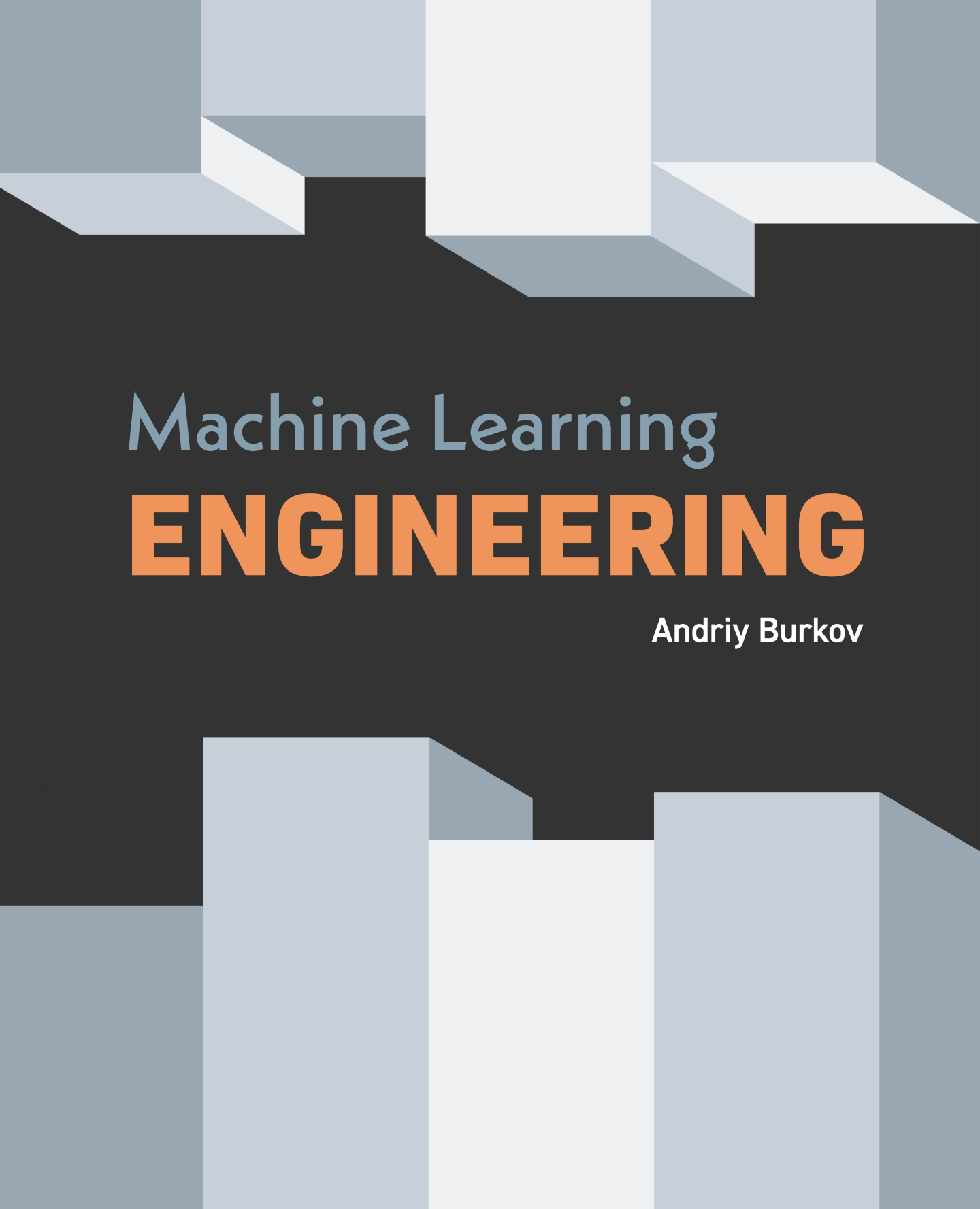
Machine Learning
ENGINEERING
Andriy Burkov

“In theory, there is no difference between theory and practice. But in practice, there is.”

— Benjamin Brewster

“The perfect project plan is possible if one first documents a list of all the unknowns.”

— Bill Langley

“When you’re fundraising, it’s AI. When you’re hiring, it’s ML. When you’re implementing, it’s linear regression. When you’re debugging, it’s printf().”

— Baron Schwartz

The book is distributed on the “read first, buy later” principle.

8 Model Deployment

Once the model has been built and thoroughly tested, it can be deployed. Deploying a model means to make it available for accepting queries generated by the users of the production system. Once the production system accepts the query, the latter is transformed into a feature vector. The feature vector is then sent to the model as input for scoring. The result of the scoring then is returned to the user.

Model deployment is the sixth stage in the machine learning project life cycle:

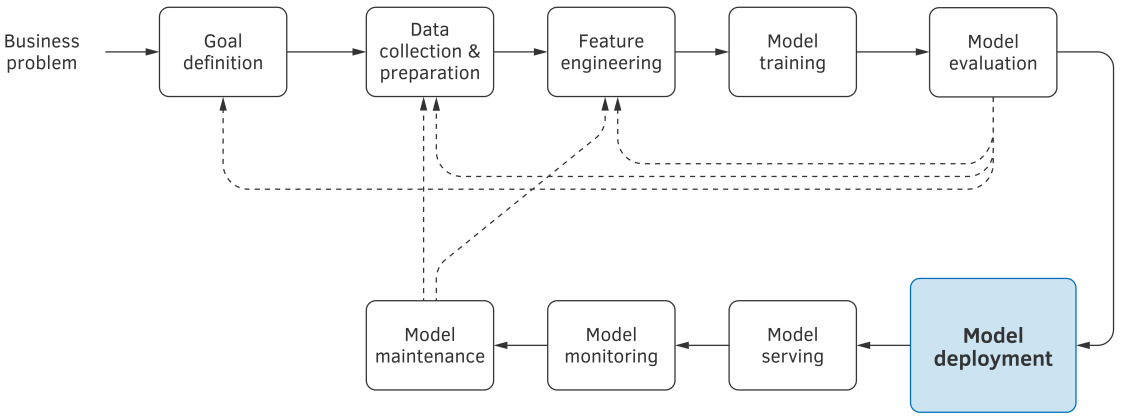

Figure 1: Machine learning project life cycle.

A trained model can be deployed in various ways. It can be deployed on a server, or on a user's device. It can be deployed for all users at once, or to a small fraction of users. Below, we consider all the options.

A model can be deployed following several **patterns**:

- statically, as a part of an installable software package,
- dynamically on the user's device,
- dynamically on a server, or
- via model streaming.

8.1 Static Deployment

The static deployment of a machine learning model is very similar to traditional software deployment: you prepare an installable binary of the entire software. The model is packaged

as a resource available at the runtime. Depending on the operating system and the runtime environment, the objects of both the model and the feature extractor can be packaged as a part of a dynamic-link library (DLL on Windows), Shared Objects (*.so files on Linux), or be serialized and saved in the standard resource location for virtual machine-based systems, such as Java and .Net.

Static deployment has many advantages:

- the software has direct access to the model, so the execution time is fast for the user,
- the user data doesn't have to be uploaded to the server at the time of prediction; this saves time and preserves privacy,
- the model can be called when the user is offline, and
- the software vendor doesn't have to care about keeping the model operational; it becomes the user's responsibility.

However, a static deployment also has several drawbacks. First and foremost, the separation of concerns between the machine learning code and the application code isn't always obvious. This makes it harder to upgrade the model without also having to upgrade the entire application. Second, if the model has certain computational requirements for scoring (such as access to an accelerator or a GPU), it may add complexity and confusion as to where the static deployment can or cannot be used. Finally, static deployments are typically difficult to scale, because the model's compute environment is shared with the application.

8.2 Dynamic Deployment on User's Device

A dynamic deployment on devices is similar to a static deployment, in the sense the user runs a part of the system as a software application on their device. The difference is that in dynamic deployment, the model is not part of the binary code of the application. Thus it achieves better separation of concerns. Pushing model updates is done without updating the whole application running on the user's device. Moreover, a dynamic deployment may allow the same piece of code to select the right model, based on the available compute resources.

Dynamic deployment can be achieved in several ways:

- by deploying model parameters,
- by deploying a serialized object, and
- by deploying to the browser.

8.2.1 Deployment of Model Parameters

In this deployment scenario, the model file only contains the learned parameters, while the user's device has installed a runtime environment for the model. Some machine learning packages, like **TensorFlow**, have a lightweight version that can run on mobile devices.

Alternatively, frameworks such as Apple's **Core ML** allow running models created using popular packages, including **scikit-learn**, **Keras**, and **XGBoost**, on Apple devices.

8.2.2 Deployment of a Serialized Object

Here, the model file is a serialized object that the application would deserialize. The advantage of this approach is that you don't need to have a runtime environment for your model on the user's device. All needed dependencies will be deserialized with the object of the model.

An evident drawback is that an update might be quite "heavy," which is a problem if your software system has millions of users.

8.2.3 Deploying to Browser

Most modern devices have access to a browser, either desktop or mobile. Some machine learning frameworks, such as **TensorFlow.js**, have versions that allow to train and run a model in a browser, by using JavaScript as a runtime.

It's even possible to train a TensorFlow model in Python, and then deploy it to, and run it in the browser's JavaScript runtime environment. Additionally, if a GPU (graphics processing unit) is available on the client's device, Tensorflow.js can leverage it.

8.2.4 Advantages and Drawbacks

The main advantage of dynamic deployment to users' devices is that the calls to the model will be fast for the user. It also reduces the impact on the organization's servers, as most computations are performed on the user's device. Additionally, if the model is deployed to the browser, the organization's infrastructure only needs to serve a web page that includes the model's parameters. A downside of the browser-based deployment is that the bandwidth cost and application startup time might increase. The users must download the model's parameters each time they start the web application, as opposed to doing it only once when they install an application.

Another drawback occurs during model updates. Recall, a serialized object can be quite voluminous. Some users may be offline during the update, or even turn off all future updates. In that case, you may end up with different users using very different model versions. Now it becomes difficult to upgrade the server-side part of the application.

Deploying models on the user's device means that the model easily becomes available for third-party analyses. They may try to reverse-engineer the model to reproduce its behavior. They may search for weaknesses by providing various inputs and observing the output. Or, they may adapt their data so the model predicts what they want.

Suppose the mobile application allows the user to read news related to their interests. A content provider might try to reverse engineer the model, so that it now recommends more often the news from that content provider.

As with static deployment, deploying to a user's device makes it difficult to monitor the model performance. The difficulty to scale with the number of scoring requests is also a shared disadvantage of both static and dynamic deployments on the user's device.

8.3 Dynamic Deployment on a Server

Because of the above complications, and problems with performance monitoring, the most frequent deployment pattern is to place the model on a server (or servers), and make it available as a Representational State Transfer application programming interface (**REST API**) in the form of a web service, or Google's Remote Procedure Call (**gRPC**) service.

8.3.1 Deployment on a Virtual Machine

In a typical web service architecture deployed in a cloud environment, the predictions are served in response to canonically-formatted HTTP requests. A web service running on a virtual machine receives a user request containing the input data, calls the machine learning system on that input data, and then transforms the output of the machine learning system into the output JavaScript Object Notation (JSON) or Extensible Markup Language (XML) string. To cope with high load, several identical virtual machines are running in parallel.

A **load balancer** dispatches the incoming requests to a specific virtual machine, depending on its availability. The virtual machines can be added and closed manually, or be a part of an **autoscaling group** that launches or terminates virtual machines based on their usage. Figure 2 illustrates that deployment pattern. Each instance, denoted as an orange square, contains all the code needed to run the feature extractor and the model. The instance also contains a web service that has access to that code.

In Python, a REST API web service is usually implemented using a web application framework such as **Flask** or **FastAPI**. An R equivalent is **Plumber**.

TensorFlow, a popular framework used to train deep models, comes with TensorFlow Serving, a built-in gRPC service.

The advantage of deploying on a virtual machine is that the architecture of the software system is conceptually simple: it's a typical web or gRPC service.

Among the downsides, there is a need to maintain servers (physical or virtual). If virtualization is used, then there is an additional computational overhead due to virtualization and running multiple operating systems. Another is network latency, which can be a serious issue, depending on how fast you need to process scoring results. Finally, deploying on a virtual

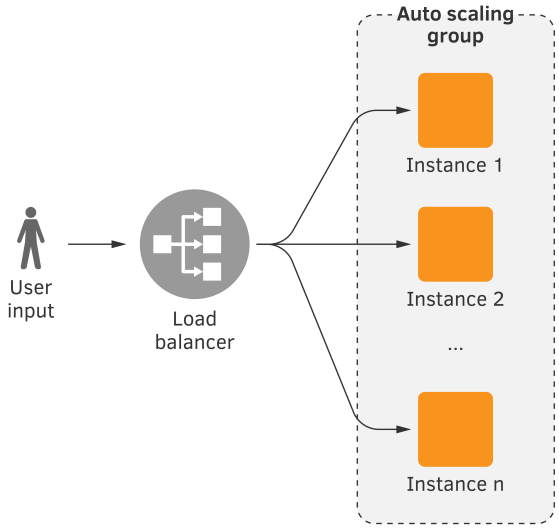

Figure 2: Deploying a machine learning model as a web service on a virtual machine.

machine has a relatively higher cost, compared to deployment in a container, or a serverless deployment that we consider below.

8.3.2 Deployment in a Container

A more modern alternative to a virtual-machine-based deployment is a container-based deployment. Working with containers is typically considered more resource-efficient and flexible than with virtual machines. A container is similar to a virtual machine, in the sense that it is also an isolated runtime environment with its own filesystem, CPU, memory, and process space. The main difference, however, is that all containers are running on the same virtual or physical machine and share the operating system, while each virtual machine runs its own instance of the operating system.

The deployment process looks as follows. The machine learning system and the web service are installed inside a container. Usually, a container is a **Docker** container, but there are alternatives. Then a container-orchestration system is used to run the containers on a cluster of physical or virtual servers. A typical choice of a container-orchestration system for running on-premises or in a cloud platform, is **Kubernetes**. Some cloud platforms provide both their own container-orchestration engine, such as **AWS Fargate** and **Google Kubernetes Engine**, and support Kubernetes natively.

Figure 3 illustrates that deployment pattern. Here, the virtual or physical machines are

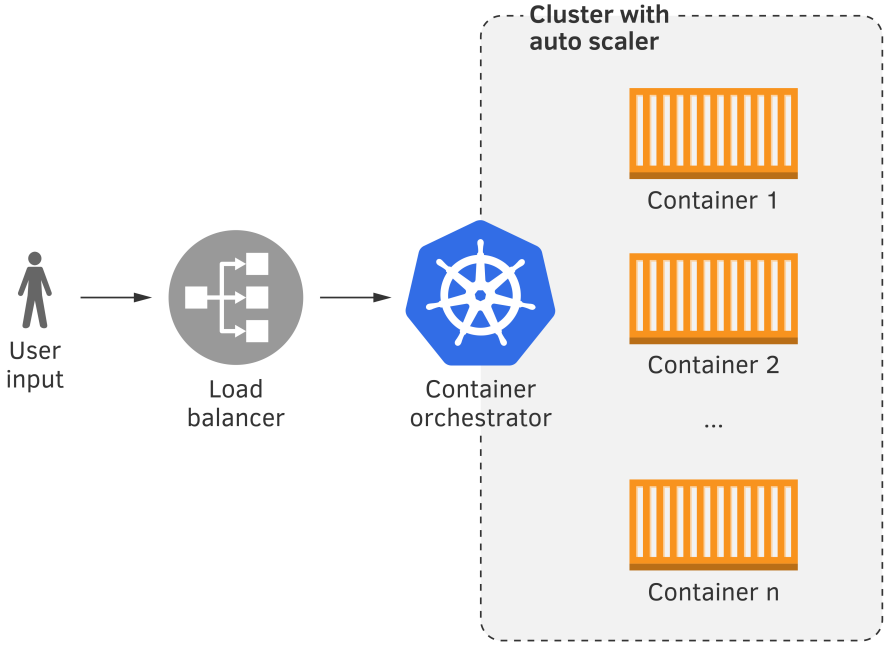

Figure 3: Deploying a model as a web service in a container running on a cluster.

organized into a cluster, whose resources are managed by the container orchestrator. New virtual or physical machines can be manually added to the cluster, or closed. If your software is deployed in a cloud environment, a cluster autoscaler can launch (and add to the cluster) or terminate virtual machines, based on the usage of the cluster.

Deployment in a container has the advantage of being more resource-efficient as compared to the deployment on a virtual machine. It allows the possibility to automatically scale with scoring requests. It also allows us to **scale-to-zero**. The idea of the scale-to-zero is that a container can be reduced down to zero replicas when idle and brought back up if there is a request to serve. As a result, the resource consumption is low compared to always running services. This leads to less power consumption and saves cost of cloud resources.

One drawback is that the containerized deployment is generally seen as more complicated, and requires expertise.

8.3.3 Serverless Deployment

Several cloud services providers, including Amazon, Google, and Microsoft, offer so-called **serverless computing**. It's known under the name of Lambda-functions on Amazon Web Services, and Functions on Microsoft Azure and Google Cloud Platform.

The serverless deployment consists of preparing a zip archive with all the code needed to run the machine learning system (model, feature extractor, and scoring code). The zip archive must contain a file with a specific name that contains a specific function, or class-method definition with a specific signature (an entry point function). The zip archive is uploaded to the cloud platform and registered under a unique name.

The cloud platform provides an API to submit inputs to the serverless function. This specifies its name, provides the payload, and yields the outputs. The cloud platform takes care of deploying the code and the model on an adequate computational resource, executing the code, and routing the output back to the client.

Usually, the function's execution time, zip file size, and amount of RAM available on the runtime are limited by the cloud service provider.

The zip file size limit can be a challenge. A typical machine learning model requires multiple heavyweight dependencies. Python's libraries, to include Numpy, SciPy, and scikit-learn, are often needed for the model to be properly executed. Depending on the cloud platform, other supported programming languages can include Java, Go, PowerShell, Node.js, C#, and Ruby.

There are many advantages to relying on serverless deployment. The obvious advantage is that you don't have to provision resources such as servers or virtual machines. You don't have to install dependencies, maintain, or upgrade the system. Serverless systems are highly scalable and can easily and effortlessly support thousands of requests per second. Serverless functions support both synchronous and asynchronous modes of operation.

Serverless deployment is also cost-efficient: you only pay for compute-time. This may also be achieved with the previous two deployment patterns using autoscaling, but autoscaling has significant latency. While the demand may drop, excessive virtual machines could still keep running before they are terminated.

Serverless deployment also simplifies **canary deployment**, or **canarying**. In software engineering, canarying is a strategy when the updated code is pushed to just a small group of end-users, usually unaware. Because the new version is only distributed to a small number of users, its impact is relatively low, and changes can be reversed quickly, should the new code contain bugs. It is easy to set up two versions of serverless functions in production, and start sending low volume traffic to just one, and test it without affecting many users. We will talk more about canarying in Section 8.4.

Rollbacks are also very simple in the serverless deployment because it is easy to switch back to the previous version of the function by replacing one zip archive.

We've discussed the zip archive size limit, and the RAM available on the runtime. These are

important drawbacks to serverless deployment. Likewise, the unavailability of GPU access[1] can be a significant limitation for deploying deep models.

Of course, complex software systems may combine deployment patterns. A deployment pattern appropriate for one model may be less optimal for another one. A combination of several deployment patterns is called a **hybrid deployment pattern**. Personal assistants like Google Home or Amazon Echo might have a model that recognizes the activation phrase (such as "OK, Google" or "Alexa") deployed on the client's device, and more complex models handle requests like "put song X on device Y" will instead run on the server. Alternatively, the deployment on the user's mobile device might augment the video and add simple intelligent effects in realtime. Server deployment would be used to apply more complex effects, such as stabilization and super-resolution.

8.3.4 Model Streaming

Model streaming is a deployment pattern that can be seen as an inverse to the REST API. In REST API, the client sends a request to the server, and then waits for a response (a prediction).

In complex systems, there can be many models applied to the same input. Or, a model can input a prediction from another model. For example, the input may be a news article. One model can predict the topic of the article, another model can extract named entities, the third model can generate a summarization of the article, and so on.

According to the REST API deployment pattern, we need one REST API per model. The client would call one API by sending a news article as a part of the request, and get the topic as response. Then the client calls another API by sending a news article, and gets the named entities as response; etc.

Streaming works differently. Instead of having one REST API per model, all models, as well as the code needed to run them, are registered within a **stream-processing engine** (SPE). Examples are **Apache Storm**, **Apache Spark**, and **Apache Flink**. Or, they are packaged as an application based on a **stream-processing library** (SPL), such as **Apache Samza**, **Apache Kafka Streams**, and **Akka Streams**.

The descriptions of these SPEs and SPLs are beyond the scope of this book, but they all share the same property making them different from the REST-API-based applications. In each stream-processing application, there is an implicit or explicit notion of **data processing topology**. The input data flows in as an infinite stream of data elements sent by the client. Following a predefined topology, each data element in the stream undergoes a transformation in the nodes of the topology. Transformed, the flow continues to other nodes.

In a stream-processing application, nodes transform their input in some way, and then either,

- send the output to other nodes, or
- send the output to the client, or

[1] As of July 2020.

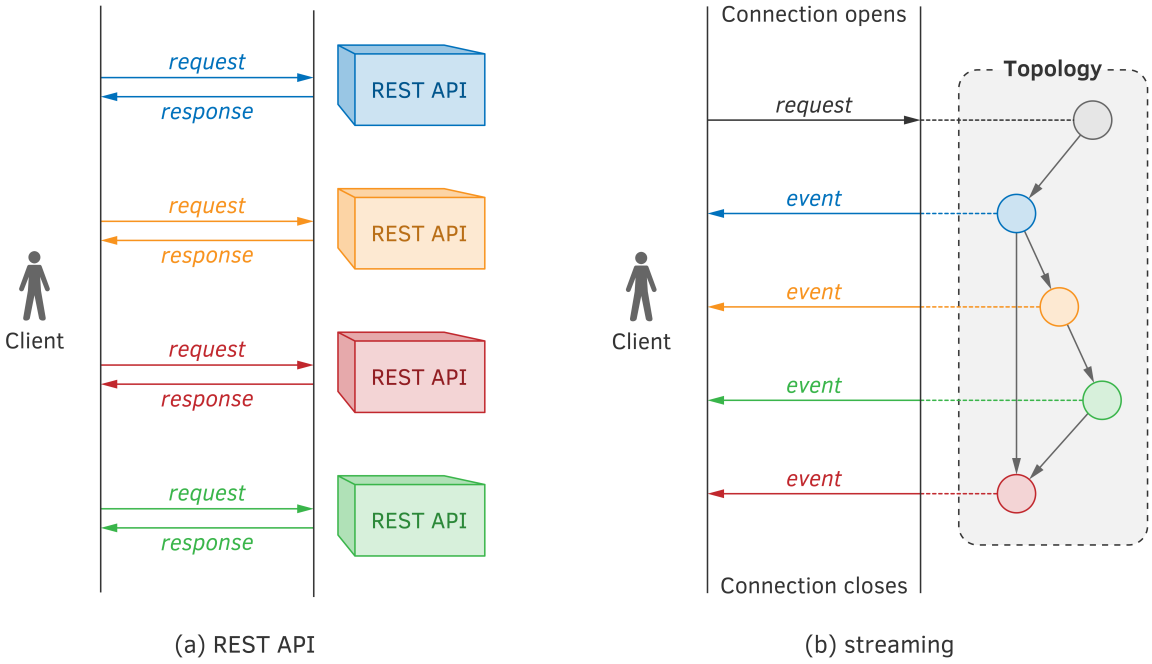

Figure 4: The difference between a REST API and streaming: (a) to process one data element, a client using a REST API sends a series of requests, one by one, and receives responses synchronously; (b) to process one data element, a client using streaming opens a connection, sends a request, and receives update events as they happen.

- persist the output to the database or a filesystem.

One node could take a news article and predict its topic; another node could take both the news article and the predicted topic and generate a summary; and so on.

The difference between a REST-API-based application and a streaming-based one is shown in Figure 4. Figure 4a shows a client using a REST API, processing one data element, such as a news article, by sending a series of requests. One by one, various REST APIs receive requests and produce responses synchronously. On the other hand, a client using streaming (Figure 4b) opens a connection to the streaming application, sends a request, and receives update events as they happen.

On the right-hand side of the streaming-based application in Figure 4b, there's a topology that defines the data flow in the application. Each input element sent by the client passes through all the nodes of the **topology graph**. Nodes can send updated events to the client, and/or persist data to the database or a filesystem.

An SPE-based streaming application runs on its own cluster of virtual or physical machines, and takes care of distributing the data processing load among the available resources. An SPL-based streaming application doesn't need a dedicated cluster for data processing. It can be integrated with available resources, such as virtual or physical machines, or a container orchestrator (such as Kubernetes).

REST APIs are usually employed to let clients send ad-hoc requests that don't follow a certain frequently-repeated pattern. It's the best choice when the client wants the liberty of deciding what to do with the API response. On the other hand, if each request of the client is:

- typical,
- undergoes a certain pattern of transformations, especially multiple intermediate transformations, and
- always results in the same actions, such as persistence of specific data elements to the filesystem or database, then streaming-based applications provide better resource-efficiency, lower latency, security, and fault-tolerance.

8.4 Deployment Strategies

Typical deployment strategies are:

- single deployment,
- silent deployment,
- canary deployment, and
- multi-armed bandit.

Let's consider each of them.

8.4.1 Single Deployment

Single deployment is the simplest one. Conceptually, once you have a new model, you serialize it into a file, and then replace the old file with the new one. You also replace the feature extractor, if needed.

To deploy on a server in a cloud environment, you prepare a new virtual machine, or a container running the new version of the model. Then you replace the virtual machine image or that of the container. Finally, you gradually close the old machines or containers, and let the autoscaler start the new ones.

To deploy on a physical server, you will upload a new model file (and the feature extraction object, if needed) on the server. Then you replace old files and old code with the new versions, and restart the web service.

To deploy on the user's device, you push the new model file to the user's device, along with any needed feature extraction object, and restart the software.

If you use interpretable code, the feature extractor object can be deployed by replacing one source code file with another one. To avoid redeploying the entire software application, on either the server or the user's device, the feature extractor's object can be serialized into a file. Then, on each startup, the software running the model would deserialize the feature extractor object.

Single deployment has the advantage of being simple; however, it's also the riskiest strategy. If the new model or the feature extractor contains a bug, all users will be affected.

8.4.2 Silent Deployment

A counterpart of the single deployment is **silent deployment**. It deploys the new model version and new feature extractor, and keeps the old ones. Both versions run in parallel. However, the user will not be exposed to the new version until the switch is done. The predictions made by the new version are only logged. After some time, they are analyzed to detect possible bugs.

Silent deployment has the benefit of providing enough time to ensure the new model works as expected, without adversely affecting any users. The drawback is the need to run twice as many models, which consumes more resources. Furthermore, for many applications, it's impossible to evaluate the new model without exposing its predictions to the user.

8.4.3 Canary Deployment

Recall, **canary deployment**, or **canarying**, pushes the new model version and code to a small fraction of users, while keeping the old version running for most users. Contrary to the silent deployment, canary deployment allows validating the new model's performance, and its predictions' effects. Contrary to the single deployment, canary deployment doesn't affect lots of users in case of possible bugs.

By opting for the canary deployment, you accept the additional complexity of having and maintaining several versions of the model deployed simultaneously.

An obvious drawback of the canary deployment is that it's impossible for engineers to spot rare errors. If you deploy the new version to 5% of users, and a bug affects 2% of users, then you have only 0.1% chance that the bug will be discovered.

8.4.4 Multi-Armed Bandits

As seen in Section **??** of Chapter 7, **multi-armed bandits** (MAB) are a way to compare one or more versions of the model in the production environment, and select the best performing one. MABs have an interesting property: after an initial exploration period, during which the MAB algorithm gathers enough evidence to evaluate the performance

of each model (arm), the best arm is eventually played all the time. It means that after the convergence of the MAB algorithm, most of the time, all users are routed to the software version running the best model.

The MAB algorithm, thus, solves two problems — online model evaluation and model deployment — simultaneously.

8.5 Automated Deployment, Versioning, and Metadata

The model is an important asset, but it's never delivered alone. There are additional assets for production model testing that ensure the model is not broken.

8.5.1 Model Accompanying Assets

Only deploy a model in production when it's accompanied with the following assets:

- an **end-to-end set** that defines model inputs and outputs that must always work,
- a **confidence test set** that correctly defines model inputs and outputs, and is used to compute the value of the metric,
- a **performance metric** whose value will be calculated on the confidence test set by applying the model to it, and
- the **range of acceptable values** of the performance metric.

Once the system using the model is initially evoked on an instance of a server or client's device, an external process must call the model on the end-to-end test data and validate that all predictions are correct. Furthermore, the same external process must validate that the value of the performance metric computed by applying the model to the confidence test set is within the range of acceptable values. If either of two evaluations fails, the model should not be served to the client.

8.5.2 Version Sync

The versions of the following three elements must always be in sync:

1) training data,
2) feature extractor, and
3) model.

Each update to the data must produce a new version in the data repository. The model trained using a specific version of the data must be put into the model repository with the same version number as that of the data used to train the model.

If the feature extractor was not changed, its version still must be updated to be in sync with the data and the model. If the feature extractor was updated, then a new model must be

built using an updated feature extractor, and the versions are incremented for the feature extractor, the model, and the training data (even if the latter wasn't changed).

The deployment of a new model version must be automated by a script in a transactional way. Given a version of the model to deploy, the deployment script will fetch the model and the feature extraction object from the respective repositories and copy them to the production environment. The model must be applied to the end-to-end and confidence test data by simulating a regular call from the outside. If there's a prediction error for the end-to-end test data, or the value of the metric is not within the range of acceptable values, the entire deployment has to be rolled back.

8.5.3 Model Version Metadata

Each model version must be accompanied with the following code and metadata:

- the name and the version of the library or package used to train the model,
- if Python was used to build the model, then requirements.txt of the virtual environment used to build the model (or, alternatively, a Docker image name pointing to a specific path on Docker Hub or in your Docker registry),
- the name of the learning algorithm, and names and values of the hyperparameters,
- the list of features required by the model,
- the list of outputs, their types, and how the outputs should be consumed,
- the version and location of the data used to train the model,
- the version and location of the validation data used to tune model's hyperparameters,
- the model scoring code that runs the model on new data and outputs the prediction.

The metadata and the scoring code may be saved to a database or to a JSON/XML text file.

For audit purposes, the following information must also accompany each deployment:

- who built the model and when,
- who and when made the decision of deploying that model, and based on what grounds,
- who reviewed the model for privacy and security compliance purposes.

8.6 Model Deployment Best Practices

In this section, we discuss practical aspects of deploying machine learning systems in production. We also outline several useful and practical tips for model deployment.

8.6.1 Algorithmic Efficiency

Most data analysts work in Python or R. While there are web frameworks that allow building web services in those two languages, they are not considered the most efficient languages.

Indeed, when you use scientific packages in Python, much of their code was written in efficient C or C++, and then compiled for your specific operating system. However, your own data preprocessing, feature extraction, and scoring code may not be as efficient.

Furthermore, not all algorithms are practical. While some algorithms can quickly solve the problem, others are too slow. For some problems, no fast algorithms can exist.

The subfield of computer science called **analysis of algorithms** is concerned with determining and comparing the complexity of algorithms. The **big O notation** is used to classify algorithms, according to how their running time or space requirements grow, as the input size grows.

For example, let's say we have the problem of finding the two most distant one-dimensional examples in the set of examples $\mathcal{S}$ of size N. One Python algorithm we could craft would look like this:

```
def find_max_distance(S):
    result = None
    max_distance = 0
    for x1 in S:
        for x2 in S:
            if abs(x1 - x2) >= max_distance:
                max_distance = abs(x1 - x2)
                result = (x1, x2)
    return result
```

or, like this in R:

```
find_max_distance <- function(S) {
    result <- NULL
    max_distance <- 0
    for (x1 in S) {
        for (x2 in S) {
            if (abs(x1 - x2) >= max_distance) {
                max_distance <- abs(x1 - x2)
                result <- c(x1, x2)
            }
        }
    }
    result
}
```

In the above algorithms, we loop over all values in $\mathcal{S}$, and, at every iteration of the first loop, we loop over all values in $\mathcal{S}$ once again. Therefore, the above algorithm makes N^2 comparisons of numbers. If we take the time the comparison, `abs`, and assignment operations take as a unit time, then the time complexity (or, simply, complexity) of this algorithm is at most $5N^2$. At each iteration, we have one comparison, two `abs`, and two assignment

operations ($1 + 2 + 2 = 5$). When the complexity of an algorithm is measured in the worst case, the big O notation is used. For the above algorithm, using big O notation, we say that the algorithm's complexity is $\mathrm{O}(N^2)$; the constants, like 5, are ignored.

For the same problem, we can craft another Python algorithm like this:

```
def find_max_distance(S):
    result = None
    min_x = float("inf")
    max_x = float("-inf")
    for x in S:
        if x < min_x:
            min_x = x
        if x > max_x:
            max_x = x
    result = (max_x, min_x)
    return result
```

or in R, like this:

```
find_max_distance <- function(S):
    result <- NULL
    min_x <- Inf
    max_x <- -Inf
    for (x in S) {
        if (x < min_x) {
            min_x <- x
        }
        if (x > max_x) {
            max_x = x
        }
    result <- c(max_x, min_x)
    result
```

In the above algorithms, we loop over all values in $\mathcal{S}$ only once, so the algorithm's complexity is $\mathrm{O}(N)$. In this case, we say that the latter algorithm is more efficient than the former.

An algorithm is called **efficient** when its complexity is polynomial in the input size. Therefore both $\mathrm{O}(N)$ and $\mathrm{O}(N^2)$ are efficient because N is a polynomial of degree 1, while N^2 is a polynomial of degree 2. However, for very large inputs, an $\mathrm{O}(N^2)$ algorithm can still be slow. In the big data era, scientists and engineers often look for $\mathrm{O}(\log N)$ algorithms.

From a practical standpoint, when implementing an algorithm, you should avoid using loops whenever possible, and implement vectorization using NumPy or similar tools. For example, you should use operations on matrices and vectors, instead of loops. In Python, to compute $\mathbf{w} \cdot \mathbf{x}$ (a dot product of two vectors), you should type,

```
import numpy
wx = numpy.dot(w,x)
```

and not,

```
wx = 0
for i in range(N):
    wx += w[i]*x[i]
```

Similarly, in R, you should type,

```
wx = w %*% x
```

and not,

```
wx <- 0
for (i in seq(N)):
    wx <- wx + w[i]*x[i]
```

Use appropriate **data structures**. If the order of elements in a collection doesn't matter, use `set` instead of `list`. In Python, the operation of verifying whether a specific example belongs to $\mathcal{S}$ is fast when $\mathcal{S}$ is a `set`, and is slow when $\mathcal{S}$ is a `list`.

Another important data structure to make your Python code more efficient is `dict`. It is called a **dictionary** or a **hash table** in other languages. It allows you to define a collection of key-value pairs with very fast lookups for keys.

Using libraries is generally more reliable — you should only write your own code when you are a researcher, or when it's truly needed. Scientific Python packages like NumPy, SciPy, and scikit-learn were built by experienced scientists and engineers with efficiency in mind. They have many methods implemented compiled C and C++ for maximum performance.

If you need to iterate over a vast collection of elements, use Python **generators** (or their R alternative in the **iterators** package) that create a function returning one element at a time, rather than all elements at once.

Use the **cProfile** package in Python (or its R counterpart, **lineprof**) to find code inefficiencies.

Finally, when nothing can be improved in your code from the algorithmic perspective, you can further boost the speed by using:

- the **multiprocessing** package in Python, or its R counterpart **parallel**, to run computations in parallel; or use a distributed processing framework such as **Apache Spark**, and
- **PyPy**, **Numba** or similar tools to compile your Python code (or the **compiler** package for R) into fast, optimized machine code.

8.6.2 Deployment of Deep Models

Sometimes, to achieve the required speed, it might be necessary to do the **scoring** on a graphics processing unit (GPU). The cost of a GPU instance in a cloud environment is typically much higher than the cost of a "normal" instance. So only the model could be deployed in an environment with one or several GPUs optimized for fast scoring. The remainder of the application could be deployed separately in a CPU environment. This approach allows reducing the cost, but, at the same time, it might add a communication overhead between two parts of the application.

8.6.3 Caching

Caching is a standard practice in software engineering. Memory cache is used to store the result of a function call, so the next time that function is called with the same values of parameters, the result is read from the cache.

Caching helps speed up the application when it contains resource-consuming functions that take time to process, or are frequently called with the same parameter values. In machine learning, such resource-consuming functions are models, especially when they run on GPUs.

The simplest cache may be implemented in the application itself. For example, in Python, the `lru_cache` decorator can wrap a function with a **memoizing** callable that saves up to the `maxsize` most recent calls:

```
from functools import lru_cache

# Read the model from file
model = pickle.load(open("model_file.pkl", "rb"))

@lru_cache(maxsize=500)
def run_model(input_example):
    return model.predict(input_example)

# Now you can call run_model
# on new data
```

The first time the function `run_model` is called for some input, `model.predict` will be called. For the subsequent calls of `run_model` with the same value of the input, the output will be read from cache that memorizes the result of `maxsize` most recent calls of `model.predict`.

In R, a similar result can be obtained using the `memo` function:

```
library(memo)

model <- readRDS("./model_file.rds")

```

```
run_model <- function(input_example) {
    result <- predict(model, input_example)
    result
}

# Create a memoized version of run_model
run_model_memo <- memo(run_model, cache = lru_cache(500))

# Now you can use run_model_memo
# instead of run_model on new data
```

Although using `lru_cache` and similar approaches is very convenient for an analyst, typically, in large scale production systems, engineers employ general purpose scalable and configurable cache solutions such as **Redis** or **Memcached**.

8.6.4 Delivery Format for Model and Code

Recall, serialization is the most straightforward way to deliver the model and the feature extractor code to the production environment.

Every modern programming language has serialization tools. In Python, it's **pickle**:

```
import pickle
from sklearn import svm, datasets

classifier = svm.SVC()
X, y = datasets.load_iris(return_X_y=True)
classifier.fit(X, y)

# Save model to file
with open("model.pickle","wb") as outfile:
    pickle.dump(classifier, outfile)

# Read model from file
classifier2 = None
with open("model.pickle","rb") as infile:
    classifier2 = pickle.load(infile)
if classifier2:
    prediction = classifier2.predict(X[0:1])
```

while in R, it's RDS:

```
library("e1071")

classifier <- svm(Species ~ ., data = iris, kernel = 'linear')
```

```
4
5  # Save model to file
6  saveRDS(classifier, "./model.rds")
7
8  # Read model from file
9  classifier2 <- readRDS("./model.rds")
10
11 prediction <- predict(classifier2, iris[1,])
```

In scikit-learn, it may be better to use **joblib**'s replacement of pickle, which is more efficient on objects that carry large **NumPy** arrays:

```
1  from joblib import dump, load
2
3  # Save model to file
4  dump(classifier, "model.joblib")
5
6  # Read model from file
7  classifier2 = load("model.joblib")
```

The same approach can be applied to save the serialized object of the feature extractor to a file, copy it to the production environment, and then read it from the file.

For some applications, the prediction speed is critical. In such cases, the production code is written in a compiled language, such as Java or C/C++. If a data analyst has built a model using Python or R, there are three options to deploy for production:

- rewrite the code in a compiled, production-environment programming language,
- use a model representation standard such as PMML or PFA, or
- use a specialized execution engine such as MLeap.

The Predictive Model Markup Language (**PMML**) is an XML-based predictive model interchange format that provides a way for data analysts to save and share models between PMML-compliant applications. PMML allows analysts to develop models within one vendor's application, and then use them within other vendors' applications, so that proprietary issues and incompatibilities are no longer a barrier to the model exchanges between applications.

For example, imagine you use Python to build an SVM model, and then save the model as a PMML file. Let the production runtime environment be a Java Virtual Machine (JVM). As long as PMML is supported by a machine learning library for JVM, and that library has an implementation of SVM, your model can be used in production directly. You don't need to rewrite your code or retrain the model in a JVM language.

The Portable Format for Analytics (**PFA**) is a more recent standard for representing both statistical models and data transformation engines. PFA allows us to easily share models and machine learning pipelines across heterogeneous systems and provides algorithmic flexibility. Models, pre- and post-processing transformations are all functions that can be arbitrarily

composed, chained, or built into complex workflows. PFA has a form of a JavaScript Object Notation (JSON) or a YAML Ain't Markup Language (YAML) configuration file.

There are open source generic "evaluators" for models or pipelines saved as PMML or PFA formatted files. **JPMML** (for Java PMML) and **Hadrian** are two of the most widely adopted. Evaluators read the model or the pipeline from a file, execute it by applying it to the input data, and output the prediction.

Unfortunately, PMML and PFA are not widely supported by the popular machine learning libraries and frameworks.[2] For example, scikit-learn doesn't support those standards, though side-projects such as **SkLearn2PMML** can convert scikit-learn objects to PMML.

Alternatively, such execution engines as **MLeap** can execute machine learning models and pipelines fast in a JVM environment. At the time of the writing of this book, MLeap could execute models and pipelines created in Apache Spark and scikit-learn.

Now, let us briefly outline several useful and practical tips for model deployment.

8.6.5 Start With a Simple Model

Deploying and applying the model in production can be more complex than it might seem. Once the infrastructure to serve a simple model is solid, a more complex model can then be trained and deployed.

A simple interpretable model is easier to debug, especially for feature extractors and entire machine learning pipelines. Complex models and pipelines have many dependencies and large numbers of hyperparameters to tune, and are more prone to implementation and deployment errors.

8.6.6 Test on Outsiders

Before putting your model in production, test your model on outsiders, and not just on the test data. Outsiders could be other team members or company employees. Alternatively, you can use crowdsourcing or a subset of your real customers who agreed to participate in experiments with new product features.

Testing on outsiders will help you avoid personal bias, because you, as the creator of the model, are emotionally involved. It will also give your model an exposure to different users (in cases when, for example, your whole team is male or Caucasian).

[2] As of July 2020.

8.7 Summary

A model can be deployed following several patterns: statically, as a part of installable software, dynamically on the user's device, dynamically on a server, or via model streaming.

The static deployment has many advantages, such as fast execution time, preserved user privacy, and the ability to call the model when the user is offline. There are also several drawbacks: it's harder to upgrade the model without also having to upgrade the entire application, and it's more difficult to scale.

The principal advantage of the dynamic deployment on the users' devices is that the calls to the model will be fast for the user. It also reduces the charge on the organization's servers. Downsides include the difficulty to deliver updates to all users and the availability of the model for third-party analyses.

As with the static deployment, deploying the model on a user's device makes it difficult to monitor the performance of the model. The difficulty to scale with the number of scoring requests is a shared disadvantage of static and dynamic deployments.

Dynamic deployment on a server can have one of the following forms: deployment on a virtual machine, deployment in a container, and serverless deployment.

The most popular deployment pattern is to deploy the model on a server and make it available as a REST API in the form of a web or a gRPC service. Here, the client sends a request to the server, and then waits for a response before sending another request.

Model streaming is different. All models are registered within a stream-processing engine or are packaged as an application based on a stream-processing library. Here, the client sends one request and receives updates as they happen.

Typical deployment strategies are single deployment, silent deployment, canary deployment, and multi-armed bandit.

In single deployment, you serialize the new model into a file, and then replace the old one.

Silent deployment consists of deploying the old and new versions, and running them in parallel. The user will not be exposed to the new version until the switch is done. The predictions made by the new version are only logged and analyzed. Thus, there is enough time to make sure that the new model works as expected without affecting any user. A drawback is the need to run more models, which consumes more resources.

Canary deployment consists of pushing the new version to a small fraction of the users, while keeping the old version running for most users. Canary deployment allows model performance validation and evaluating the users' experience. It won't affect lots of users in case of possible bugs.

Multi-armed bandits allow us to deploy the new model while keeping the old one. The algorithm replaces the old model with the new one only when it is certain that it performs better.

The deployment of a new model version must be automated by a script in a transactional way. Given a version of the model to deploy, the deployment script will fetch the model and the feature extraction object from the respective repositories and copy them to the production environment. The model must be applied to the end-to-end and confidence test data by simulating a regular call from the outside. If there's a prediction error for the end-to-end test data, or the value of the metric is not within the range of acceptable values, the entire deployment has to be rolled back.

The versions of training data, feature extractor, and model must always be in sync.

Algorithmic efficiency is an important consideration in model deployment. Experienced scientists and engineers built Python packages like NumPy, SciPy, and scikit-learn with efficiency in mind. Your own code may not be as reliable or efficient. You should only write your own code when it's absolutely necessary.

If you implement your own algorithmic code, avoid loops. Implement vectorization with NumPy or similar tools. Use appropriate data structures. If the order of elements in a collection doesn't matter, use a `set` instead of a `list`. Using dictionaries (or hash tables) allows you to define a collection of key-value pairs with very fast lookups for keys.

Caching speeds up the application when it contains resource-consuming functions frequently called with the same parameter values. In machine learning, such resource-consuming functions are models, especially when they run on GPUs.