CHAPTER 12

The Intelligence Runtime

The intelligence runtime puts intelligence into action. It is responsible for interfacing with the rest of the Intelligent System, gathering the information needed to execute the system's intelligence, loading and interpreting the intelligence, and connecting the intelligence's predictions back to the rest of the system.

Conceptually, an intelligence runtime might look something like this:

```
Intelligence theIntelligence =
               InitializeIntelligence(<intelligence data file>,
                              <server address>,
                              <etc...>);

Context theContext =
               GatherContext(<sensors>,
                              <content>,
                              <user history>,
                              <other system properties>,
                              <etc...>);

Prediction thePrediction =
        ExecuteIntelligence(theIntelligence, theContext);

UpdateTheExperience(thePrediction);
```

It's simple in principal, but, like most good things, it can require a great deal of complexity in practice.

This section covers the principal elements that an intelligence runtime will handle:

- Gathering context.
- Constructing features.
- Loading models.

G. Hulten, *Building Intelligent Systems*, https://doi.org/10.1007/978-1-4842-3432-7_12

- Executing models on the context/features.
- Connecting the resulting predictions with the experience.

Context

The context in an intelligence runtime consists of all the data the intelligence might use to make its decision. For example:

- If the Intelligent System is trying to determine if a sprinkler system should run, the context might include.
  - The amount of time since the sprinkler last ran.
  - The forecast for the next several days (will it rain or not).
  - The amount of rain that has fallen in the past several days.
  - The type of landscaping the user has.
  - The time of day.
  - The temperature.
  - The humidity.
  - And so on.
- If the Intelligent System is looking for cats in an image, the context might include.
  - The raw image data (RGB intensity for the pixels).
  - Metadata about how the image was captured (the exposure and frame rate, the time, the zoom).
  - The location the image came from.
  - And so on.
- If the Intelligent System is trying to tell if a user should take a break from their computer, the context might include.
  - The time since the last break.
  - The activities the user has done since the break (number of keystrokes and mouse movements).

- An indication of how active the user has been.
- The amount of time until the user's next meeting (according to their calendar).
- The name of the application that the user is currently interacting with.
- And so on.

The context can be just about any computer-processable information, like a URL, a sensor reading from some embedded device, a digital image, a catalog of movies, a log of a user's past behavior, a paragraph of text, the output of a medical device, sales receipts from a store, the behavior of a program, and so on.

These are the things that might be useful in making the decision about what the Intelligent System should do. In fact, the context is usually a superset of the things the intelligence will actually use to make the decision.

A good context will:

- Contain enough information to allow the intelligence be effective.
- Contain enough information to allow the intelligence to grow and adapt.
- Contain enough information to allow the Intelligent System's orchestrators to track down mistakes.
- Be efficient enough to gather at runtime for the intelligence call.
- Be compact enough to include in telemetry.

Clearly there are trade-offs. More complete context gives the intelligence more potential, but it may also encounter engineering and operational constraints, requiring additional CPU and memory usage and network bandwidth, and introducing latency and increasing data size.

Feature Extraction

Gathering these context variables can require significant effort. But it isn't the end of the process. There is also another layer of code that converts these variables into the representation the intelligence needs. This extra layer is often called "feature extraction" or "featurization" and is a common part of machine-learning systems.

There are a number of reasons this matters for an intelligence implementation:

- The code that produces features from context needs to run in the runtime. That is, it needs to be efficient and reliable.
- This code is often technical, computationally expensive, mathy, and not intuitive to people who don't have extensive experience in working with data.
- The intelligence creators are going to want to constantly change what this code does; in fact, they may need to change it periodically to continue to grow the intelligence.
- This code is tightly coupled to the intelligence, and the feature extraction code and intelligence (models) absolutely must be kept in sync—the exact version of the feature extraction code used to build the intelligence must also be used in the runtime when the intelligence is executed.

Because of these factors, feature extraction must be carefully planned. Some best practices include:

- Have extensive test sets for feature extraction code. This should include:
  - Unit tests on the extractors.
  - Performance tests to make sure the code meets CPU and RAM requirements.
  - A test that periodically executes feature extraction code against a set of known contexts (maybe daily, maybe at each check-in) and compares the output to a known benchmark. Any change in output should be examined by engineers and intelligence creators to make sure it is what was intended.
  - A test that ensures the feature extraction code at runtime (when users are interacting with it) is doing exactly the same thing it is doing in the intelligence creation environment (when it is being used to create models).
- Have version information encoded in intelligence and in the feature extraction code and have the runtime verify that they are in sync.

- Have the ability to change feature-extraction code out-of-band of the full application; for example, by sending a new version of the feature extraction code with each model update.
- Develop a life-cycle for moving feature extraction changes from intelligence creators, to engineers, to deployment. The life-cycle should allow intelligence creators to move quickly in evaluating changes, and the quality of the code deployed to users to be high.

Models

The model is the representation of the intelligence (and we will discuss representation options in coming chapters). For now, imagine the model as a data file. In some Intelligent Systems, the models are changed every few days. In some systems they are changed every few minutes.

The rate of change will depend on these factors:

1. How quickly the system gets enough data to improve the models.
2. How long it takes to construct new models.
3. How much it costs to distribute the new models.
4. How necessary change is relative to the success criteria.

In general, models will change (much) more quickly than the larger program. An effective runtime will:

- Allow the model to be updated easily (that is, without having to restart the whole system).
- Make sure the models it uses are valid, and all the components (the feature extractors, the models, the context) are in sync.
- Make it easy to recover from mistakes in models (by rolling back to previous versions).

One key consideration with models is the amount of space they take up on disk and in RAM. Left unchecked, machine learning can produce models that are quite large. But it is always possible to make tradeoffs, for example between model size and accuracy, if needed.

Execution

Executing a model is the process of asking the model to make its prediction based on the context (and associated features).

In the simplest case, when there is a single model and it lives on the client, executing a model can be as easy as loading the features into an array and making a function call into an off-the-shelf model execution engine. There are libraries to execute pretty much every type of model that machine learning can build, and in most cases these libraries are perfectly acceptable—so most of the time you won't even need to write much code to load and execute a model.

Of course, in some cases these libraries aren't acceptable. Examples include when RAM or CPU is a constraint; or when the execution is happening on a device with special hardware (like a GPU or FPGA) that the libraries don't take advantage of.

Another form of execution involves a model running on a server. In these cases, the client needs to bundle some (compact) representation of the context (or the features), ship them to the server, and wait for an answer, dealing with all the coordination required to properly account for the latency of the call and keep it from resulting in bad user experiences.

When the system has more than one model (which is often the case), the execution needs to execute each of the models, gather their results, and combine them into a final answer. There are many ways to do this, including:

- Averaging their results.
- Taking the highest (most certain) result.
- Having some rules about the context that select which one to trust (for example, one model nails house prices in 90210 but is terrible everywhere else, another model works with condos but not single-family homes).
- Having a model that combines the outputs of the other models.

The chapters in Part IV, "Creating Intelligence," will discuss model combination in more detail along with the pros and cons of these (and other) approaches.

Results

Executing the intelligence results in an answer. Maybe the intelligence outputs a 0.91, which represents a 91% probability in whatever the intelligence is talking about. The implementation must use this to affect what the intelligent experience will do. For example, it might compare the value to a threshold and if the value is high (or low) enough, initiate a prompt, or automate something, or annotate some information—whatever the intelligent experience wants to do.

Instability in Intelligence

Using the raw output of intelligence can be risky, because intelligence is intrinsically unstable. For example:

Take a digital image of a cow. Use a cow-detector model to predict the probability the image contains a cow, and the output might be high, maybe 97.2%.

Now wait a day. The back-end system has new training data, builds a new cow-detector model, distributes it to the runtime. Apply the updated model to the same cow picture and the probability estimate will almost certainly be different. The difference might be small, like 97.21% (compared to 97.2% from the original) or it might be relatively large, like 99.2%. The new estimate might be more accurate, or it might be less accurate. But it is very, very unlikely the estimate will be the same between two different versions of intelligence.

Subtle changes in context can have similar effects. For example, point your camera at a cow and snap two digital images right in a row, one after the other, as fast as you can. To a human, the cow looks the same in both images, the background looks the same, and the lighting looks the same—the "cow detector" model should output the same probability estimation on the first image as on the second. According to a human.

But to a model, the images might look very different: digital sensors introduce noise, and the noise profile will be different between the images (resulting in little speckles that human eyes have a hard time seeing); the cow might have blinked in one and not the other; the wind might have blown, changing the location of shadows from leaves; the position of the camera may have changed ever-so-slightly; the camera's auto-focus or exposure correction might have done slightly different things to the images.

Because of all of this, the intelligence might make different estimations on the two images. Maybe 94.1% in one and 92.7% in the other.

Maybe that change isn't a big deal to the intelligent experience that results, but maybe it is.

Trying to pretend that there isn't instability in machine-learning systems can result in less effective (or downright bad) intelligent experiences.

Intelligence APIs

To address instability in intelligence outputs, it's often helpful to encapsulate intelligence behind an interface that exposes the minimum amount of information needed to achieve the system's goals:

- When dealing with a probability, consider throwing away a bunch of resolution: round 92.333534% to 90%.
- Consider turning probabilities into classifications: instead of "45% there is a cow," say, "There is not a cow."
- Consider quantizing predictions: instead of "There is a cow at coordinates 14,92 in the image," say, "There is a cow on the left side of the image."

The thresholds and policies needed to implement these types of transformations are tightly coupled to the intelligence and should be included in the model with each update.

And while these practices certainly help, making the model's output more stable, and providing some level of encapsulation of the model's inner workings, they aren't perfect. For example, imagine the cow detector is 81% sure there is a cow in one frame. The model's threshold is at 80%. So the system creates a "there is a cow here" classification and lights up a UX element.

Then on the next frame, the cow detector is 79.5% sure there is a cow present, and the user experience flickers off. This can look really bad.

A well designed API, combined with an experience designed to minimize flaws, are both required to make an effective intelligent experience.

Summary

The intelligence runtime is responsible for: gathering the context of the system; converting this context into a form that works with the intelligence; executing the various components that make up the intelligence; combining the intelligence results and

creating a good interface between the intelligence and the experience; and using the output of the intelligence to affect the user experience.

Some key components that an intelligence runtime will deal with include these:

- **The context,** including all the information relevant to making a good decision in the Intelligent System.
- **The features** (and the feature extraction code), which convert the context into a form that is compatible with the specific models that contain the system's intelligence.
- **The models,** which represent the intelligence and are typically contained in data files that will change relatively frequently during the lifetime of an Intelligent System.
- **The execution engine,** which executes the models on the features and returns the predictions. There are many great libraries to support executing models, but these will often need to be wrapped to combine intelligence; they will sometimes need to be replaced for unique execution environments.
- **The results,** which are the predictions of the intelligence. It is good practice to keep the raw results private and create an intelligence API that exposes the minimum information to power the intelligent experience, while being robust to changes over time.

An effective intelligence runtime will make it easy to track down mistakes; will execute in the runtime the same way it executes in the intelligence creation environment; will support easy changes to both models and feature extraction; and will make it hard for parts of the system to get out of sync.

For Thought…

After reading this chapter, you should:

- Be able to design a runtime that executes intelligence and uses it to power user experience.
- Understand how to structure an intelligence runtime to allow innovation in intelligence and support the other components of the intelligence implementation.

You should be able to answer questions like these:

- What is the difference between the context of an intelligence call and the features used by the machine-learned model?

Consider the Intelligent System you used most recently.

- What type of information might be in the context of the intelligence calls it makes?
- At a high level, walk through the steps to go from that context to a user-impacting experience (invent any details you need to about how the system might work).
- What are some ways this system's intelligence might be encapsulated to mitigate the effects that small intelligence changes have on the user?