Constellation

- Overview
- Client API
- Platform

Overview

import { CardGrid, Description, LinkTitleCard } from "~/components"

Run machine learning models with Cloudflare Workers.

Constellation allows you to run fast, low-latency inference tasks on pre-trained machine learning models natively on Cloudflare Workers. It supports some of the most popular machine learning (ML) and AI runtimes and multiple classes of models.

Cloudflare provides a curated list of verified models, or you can train and upload your own.

Functionality you can deploy to your application with Constellation:

- Content generation, summarization, or similarity analysis
- Question answering
- Audio transcription
- Image or audio classification
- Object detection
- Anomaly detection
- Sentiment analysis

More resources

Connect with the Workers community on Discord to ask questions, show what you are building, and discuss the platform with other developers. Follow @CloudflareDev on Twitter to learn about product announcements, and what is new in Cloudflare Workers.

Client API

The Constellation client API allows developers to interact with the inference engine using the models configured for each project. Inference is the process of running data inputs on a machine-learning model and generating an output, or otherwise known as a prediction.

Before you use the Constellation client API, you need to:

- Sign up for a Cloudflare account.
- Enable Constellation by logging into the Cloudflare dashboard > **Workers & Pages** > **Constellation**.
- Create a Constellation project and configure the binding.
- Import the `@cloudflare/constellation` library in your code:

```
import { Tensor, run } from "@cloudflare/constellation";
```

Tensor class

Tensors are essentially multidimensional numerical arrays used to represent any kind of data, like a piece of text, an image, or a time series. TensorFlow popularized the use of Tensors in machine learning (hence the name). Other frameworks and runtimes have since followed the same concept.

Constellation also uses Tensors for model input.

Tensors have a data type, a shape, the data, and a name.

```
enum TensorType {
    Bool = "bool",
    Float16 = "float16",
    Float32 = "float32",
    Int8 = "int8",
    Int16 = "int16",
    Int32 = "int32",
    Int64 = "int64",
}

type TensorOpts = {
```

```
  shape?: number[],
  name?: string
}

declare class Tensor<TensorType> {
  constructor(
    type: T,
    value: any | any[],
    opts: TensorOpts = {}
  )
}
```

Create new Tensor

```
new Tensor(
  type:TensorType,
  value:any | any[],
  options?:TensorOpts
  )
```

type

Defines the type of data represented in the Tensor. Options are:

- TensorType.Bool
- TensorType.Float16
- TensorType.Float32
- TensorType.Int8
- TensorType.Int16
- TensorType.Int32
- TensorType.Int64

value

This is the tensor's data. Example tensor values can include:

- scalar: 4
- vector: [1, 2, 3]
- two-axes 3x2 matrix: [[1,2], [2,4], [5,6]]
- three-axes 3x2 matrix [[[1, 2], [3, 4]], [[5, 6], [7, 8]], [[9, 10], [11, 12]]]

options

You can pass options to your tensor:

shape

Tensors store multidimensional data. The shape of the data can be a scalar, a vector, a 2D matrix or multiple-axes matrixes. Some examples:

- [] - scalar data
- [3] - vector with 3 elements
- [3, 2] - two-axes 3x2 matrix
- [3, 2, 2] - three-axis 2x2 matrix

Refer to the TensorFlow documentation for more information about shapes.

If you don't pass the shape, then we try to infer it from the value object. If we can't, we thrown an error.

name

Naming a tensor is optional, it can be a useful key for mapping operations when building the tensor inputs.

Tensor examples

A scalar

```
new Tensor(TensorType.Int16, 123);
```

Arrays

```
new Tensor(TensorType.Int32, [1, 23]);
new Tensor(TensorType.Int32, [ [1, 2], [3, 4], ], { shape: [2, 2] });
new Tensor(TensorType.Int32, [1, 23], { shape: [1] });
```

Named

```
new Tensor(TensorType.Int32, 1, { name: "foo" });
```

Tensor properties

You can read the tensor's properties after it has been created:

```
const tensor = new Tensor(TensorType.Int32, [ [1, 2], [3, 4], ], { shape: [2, 2], name: "test"
});

console.log ( tensor.type );
// TensorType.Int32

console.log ( tensor.shape );
// [2, 2]

console.log ( tensor.name );
// test

console.log ( tensor.value );
//  [ [1, 2], [3, 4], ]
```

Tensor methods

async tensor.toJSON()

Serializes the tensor to a JSON object:

```
const tensor = new Tensor(TensorType.Int32, [ [1, 2], [3, 4], ], { shape: [2, 2], name: "test"
});

tensor.toJSON();

{
  type: TensorType.Int32,
  name: "test",
  shape:  [2, 2],
  value: [ [1, 2], [3, 4], ]
}
```

async tensor.fromJSON()

Serializes a JSON object to a tensor:

```
const tensor = Tensor.fromJSON(
  {
    type: TensorType.Int32,
    name: "test",
    shape:  [2, 2],
    value: [ [1, 2], [3, 4], ]
  }
);
```

InferenceSession class

Constellation requires an inference session before you can run a task. A session is locked to a specific project, defined in your binding, and the project model.

You can, and should, if possible, run multiple tasks under the same inference session. Reusing the same session, means that we instantiate the runtime and load the model to memory once.

```
export class InferenceSession {
    constructor(binding: any, modelId: string, options: SessionOptions = {})
}

export type InferenceSession = {
  binding: any;
  model: string;
  options: SessionOptions;
};
```

InferenceSession methods

new InferenceSession()

To create a new session:

```
import { InferenceSession } from "@cloudflare/constellation";

const session = new InferenceSession(
  env.PROJECT,
  "0ae7bd14-a0df-4610-aa85-1928656d6e9e"
```

```
);
```

- **env.PROJECT** is the project binding defined in your Wrangler configuration.
- **0ae7bd14...** is the model ID inside the project. Use Wrangler to list the models and their IDs in a project.

async session.run()

Runs a task in the created inference session. Takes a list of tensors as the input.

```
import { Tensor, InferenceSession, TensorType } from "@cloudflare/constellation";

const session = new InferenceSession(
  env.PROJECT,
  "0ae7bd14-a0df-4610-aa85-1998656d6e9e"
);

const tensorInputArray = [ new Tensor(TensorType.Int32, 1), new Tensor(TensorType.Int32, 2),
new Tensor(TensorType.Int32, 3) ];

const out = await session.run(tensorInputArray);
```

You can also use an object and name your tensors.

```
const tensorInputNamed = {
  "tensor1": new Tensor(TensorType.Int32, 1),
  "tensor2": new Tensor(TensorType.Int32, 2),
  "tensor3": new Tensor(TensorType.Int32, 3)
};

out = await session.run(tensorInputNamed);
```

This is the same as using the name option when you create a tensor.

```
{ "tensor1": new Tensor(TensorType.Int32, 1) } == [ new Tensor(TensorType.Int32, 1, { name:
"tensor1" } ];
```

Platform

import { DirectoryListing } from "~/components"