Runway Model SDK

Release v0.1.0

Runway AI, Inc.

Jul 07, 2020
## CONTENTS

1 Installing 3

2 Runway Models 5
   2.1 Example runway_model.py ................................. 5
   2.2 Example runway.yml ................................. 6
These documents serve as a reference for the Runway Model SDK. With a few lines of code, you can port existing ML models to the Runway platform so they can be used and shared by others.
This SDK supports both Python 3.6+. You can install the module using either `pip` or `pip3` like so:

```
pip3 install runway-python
```

Published versions of the SDK are hosted on the PyPI project website.
A Runway model consists of two special files:

- **runway_model.py**: A python script that imports the `runway` module (SDK) and exposes its interface via one or more `runway.command()` functions. This file is used as the **entrypoint** to your model.
- **runway.yml**: A configuration file that describes dependencies and build steps needed to build and run the model.

### 2.1 Example runway_model.py

Runway models expose a standard interface that allows the Runway app to interact with them over HTTP. This is accomplished using three functions: `@runway.setup()`, `@runway.command()`, and `runway.run()`.

Any Python-based model, independent of the ML framework or toolkit, can be converted into a Runway model using this simple interface. For more information about the `runway` module, see the module reference page.

**Note:** This is example code for demonstration purposes only. It will not run, as the `your_image_generation_model` import is not a real python module.

```python
import runway
from runway.data_types import category, vector, image
from your_image_generation_model import big_model, little_model

# The setup() function runs once when the model is initialized, and will run
# again for each well formed HTTP POST request to http://localhost:9000/setup.
@runway.setup(options={'model_size': category(choices=['big', 'little'])})
def setup(opts):
    if opts['model_size'] == 'big':
        return big_model()
    else:
        return little_model()

inputs = { 'noise_vector': vector(length=128, description='A random seed.') }
outputs = { 'image': image(width=512, height=512) }

# The @runway.command() decorator is used to create interfaces to call functions
# remotely via an HTTP endpoint. This lets you send data to, or get data from,
# your model. Each command creates an HTTP route that the Runway app will use
# to communicate with your model (e.g. POST /generate). Multiple commands
# can be defined for the same model.
@runway.command('generate', inputs=inputs, outputs=outputs, description='Generate an image.')
```

(continues on next page)
def generate(model, input_args):
    # Functions wrapped by @runway.command() receive two arguments:
    # 1. Whatever is returned by a function wrapped by @runway.setup(),
    #    usually a model.
    # 2. The input arguments sent by the remote caller via HTTP. These values
    #    match the schema defined by inputs.
    img = input_args['image']
    return model.generate(img)

# The runway.run() function triggers a call to the function wrapped by
# @runway.setup() passing model_options as its single argument. It also
# creates an HTTP server that listens for and fulfills remote requests that
# trigger commands.
if __name__ == '__main__':
    runway.run(host='0.0.0.0', port=9000, model_options={ 'model_size': 'big' })

If you are looking to port your own model, we recommend starting from our Model Template repository hosted on GitHub. This repository contains a basic model that you can use as boilerplate instead of having to start from scratch.

### 2.2 Example runway.yml

Each Runway model must have a runway.yml configuration file in its root directory. This file defines the steps needed to build and run your model for use with the Runway app. This file is written in YAML, a human-readable superset of JSON. Below is an example runway.yml file. This example file illustrates how you can provision your model’s environment.

```yaml
version: 0.1
python: 3.6
entrypoint: python runway_model.py
cuda: 9.2
framework: tensorflow
files:
    ignore:
        - image_dataset/*
build_steps:
    - pip install runway-python
    - pip install -r requirements.txt
```

Continue on to the Runway YAML reference page to learn more about the possible configuration values supported by the runway.yml file, or hop over to the Example Models page to check out the source code for some of the models that have already been ported to Runway.

#### 2.2.1 Runway YAML File

A runway.yml file is required to be in the root file directory of each Runway model. This file provides instructions for defining an environment, installing dependencies, and running your model in a standard and reproducible manner. These instructions are used by the Runway build pipeline to create a Docker image that can be run in a platform independent way on a local machine or a remote GPU cloud instance (although this process is abstracted away from the model builder). The runway.yml config file is written in YAML and has a simple structure. You are free to copy and paste the example config file below, changing or removing the values that you need.

**Note:** The Runway configuration file must be named runway.yml and exist in the root (top-level) directory of your model.
model. Also, make sure you use the `.yml` file extension as the alternative `.yaml` extension is not supported.

### Example

```yaml
# Specify the version of the runway.yml spec.
version: 0.1
# Supported python versions are: 3.6
python: 3.6
# The command to run your model. This value is used as the CMD value in
# the generated Docker image.
entrypoint: python runway_model.py
# Which NVIDIA CUDA version to use. Supported versions include 10.2, 10, 9.2, and 9.
cuda: 9.2
# Which ML framework would you like to pre-install? The appropriate GPU/CPU
# versions of these libraries are selected automatically. Accepts values
# "tensorflow" and "pytorch", installing Tensorflow v1.12 and Pytorch v1.0
# respectively.
framework: tensorflow
# Builds are created for CPU and GPU environments by default. You can use the
# spec object to limit your builds to one environment if you'd like, for
# instance if your model doesn't use CUDA or run on a GPU you can set
# gpu: False.
spec:
  cpu: True
  gpu: True
files:
  # All files in the root project directory will be copied to the Docker image
  # automatically. Builds that require excessive storage can fail or take a
  # very long time to install on another user's machine. You can use the
  # files.ignore array to exclude files from your build.
  ignore:
    - my_dataset/*
    - secrets.txt
# The build_steps array allows you to run shell commands at build time. Each
# Each build step is executed in the order it appears in the array.
build_steps:
  # We recommend pinning to a specific version of the Runway Model SDK until
  # the first major release, as breaking changes may be introduced to the SDK
  - pip install runway-python==0.0.74
  - pip install -r requirements.txt
# The if_gpu and if_cpu directives can be used to run build steps
# conditionally depending on the build environment.
  - if_gpu: echo "Building in a GPU environment..."
  - if_cpu: echo "Building in a CPU only environment..."
```

**Note:** If you require an ML framework other than Tensorflow or Pytorch, or a version of these libraries that is
different than the versions provided by the `frameworks` object, you can install these dependencies manually in the
build steps.

```bash
build_steps:
  - pip install tensorflow==1.0
  - if_gpu: pip install tensorflow-gpu==1.0
```

### 2.2. Example `runway.yml`
Schema Reference

- **version** (int, optional, default = 0.1): This version specifies the schema of the configuration file not the version of the Runway Model SDK itself.

- **python** (float, **required**): The Python version to use when running the model installing python dependencies. Only 3.6 is supported at this time.

- **entrypoint** (string, **required**): The command to run your model. This value is used as the CMD value in the generated Docker image. A standard value for this field might be `entrypoint: python runway_model.py` where `runway_model.py` implements the `@runway.setup()`, `@runway.command()`, and most importantly the `runway.run()` functions.

- **cuda** (float, **required if building for GPU**): The NVIDIA CUDA version to use in the production GPU runtime environment. The currently supported CUDA versions are 10.2, 10.1, 9.2, and 9.

- **framework** (string, optional, default = None): The machine learning framework to pre-install during the build. Currently we support "tensorflow" and "pytorch" which will install the appropriate CPU or GPU packages of Tensorflow v1.12.0 and Pytorch v1.0 respectively depending on the build environment. If you require an ML framework other than Tensorflow or Pytorch, or a version of these libraries that is different than the versions provided by the `frameworks` object, you can omit this object and install these dependencies manually in the build steps.

- **spec** (object, optional): A dictionary of boolean values specifying which CPU/GPU environments to build for. Both the `cpu` and `gpu` environments are enabled (`True`) by default.
  - **cpu** (boolean, optional, default = `True`): Create a CPU build.
  - **gpu** (boolean, optional, default = `True`): Create a GPU build.

- **files** (object, optional): A dictionary that defines special behaviors for certain files. All values in this dictionary are specified as paths, with support for the glob character (e.g. `data/*.jpg`).
  - **ignore** (array of strings, optional): A list of file paths to exclude from the build.

- **build_steps** (array of strings or dictionary values containing the `if_cpu` and `if_gpu` keys, optional): A list of shell commands to run at build time. Use this list to define custom build steps. Build steps are run in the order they appear in the array. The `if_gpu` and `if_cpu` directives can be used to run build steps conditionally depending on the build environment.

### 2.2.2 Runway Module

The Runway module exposes three simple functions that can be combined to expose your models to the Runway app using a simple interface.

- **@runway.setup():** A Python decorator used to initialize and configure your model.

- **@runway.command():** A Python decorator used to define the interface to your model. Each command creates an HTTP route which can process user input and return outputs from the model.

- **runway.run():** The entrypoint function that starts the SDK’s HTTP interface. It fires the function decorated by `@runway.setup()` and listens for commands on the network, forwarding them along to the appropriate functions decorated with `@runway.command()`.
2.2.3 Data Types

The Runway Model SDK provides several data types that can be used to pass values to and from runway models and the applications that control them. The data types currently supported by the SDK are number, text, image, array, vector, category, file, and any, an extensible data type. These data types are primarily used in two places:

- The options parameter in the @runway.setup() decorator
- The input and output parameters in @runway.command() decorator

Note: This is example code for demonstration purposes only. It will not run, as the your_code import is not a real python module.

```python
import runway
from runway.data_types import category, vector, image
from your_code import model

options = {"network_size": category(choices=[64, 128, 256, 512], default=256)}
@runway.setup(options=options)
def setup(opts):
    return model(network_size=opts["network_size"])  

sample_inputs = {
    "z": vector(length=512),
    "category": category(choices=["day", "night"])
}

sample_outputs = {
    "image": image(width=1024, height=1024)
}

@runway.command("sample", inputs=sample_inputs, outputs=sample_outputs)
def sample(model, inputs):
    img = model.sample(z=inputs["z"], category=inputs["category"])
    # `img` can be a PIL or numpy image. It will be encoded as a base64 URI
    # string automatically by @runway.command().
    return { "image": img }

if __name__ == "__main__":
    runway.run()
```

Reference

2.2.4 Exceptions

The Runway Model SDK defines several custom exception types. All of these exceptions derive from the base RunwayError class, which itself derives from the standard Python Exception class. Each RunwayError contains error message and code properties, and a to_response() method that converts the exception to a Python dictionary, which can be returned as a JSON HTTP response.
try:

do_something_with_runway()
except RunwayError as e:
    print(e.code, e.message)
    # 500 An unknown error has occurred
    print(e.to_response())
    # { "error": "An unknown error has occurred", "traceback": "..." }

Reference

2.2.5 User Interface Components

One of the main advantages to porting a machine learning model to Runway is that you don’t have to spend time designing and building a user interface for users to interact with your model. The Model SDK, in tandem with Runway itself, works to abstract away the details of how users get data into and out of a model. As a model porter, you design your model commands, inputs, and outputs using simple Data Types which get automatically transformed into UI components inside the app. You don’t have to worry about where the data comes from or where it goes once you process it; Runway takes care of that for you.

This abstraction process also unifies the user experience. Once a user learns how to use their camera to process a live video feed, select an input file from disk, or export the output of a model to CSV, they now know how to do these things with all models; They don’t have to re-learn the process for each model.

Workspace View

Users interact with your model by adding it to a workspace. From there they are able to chose input and output sources based on the Data Types the model porter specifies in each @runway.command() decorator (more on the Runway
The screenshot above depicts an example of OpenAI’s Glow model which manipulates facial features of images containing faces. The Glow model that’s been ported to Runway expects an image as input accompanied by a facial feature category like “Young”, “Gray_Hair”, or “Smiling” and an amount that controls the intensity of the transformation. You will see all three of these inputs listed as “Inference Options” on the right UI control panel. Inference options can be controlled by the user while the model is running (see Setup Options vs Inference Options below).

Each model’s Inference Options UI panel is automatically generated based on the code in its runway_model.py file. The model porter controls which types of UI components are used for each model using only the inputs and outputs keyword arguments to each @runway.command(). Below is a toy example of what the runway_model.py code that generates the UI components looks like for Glow. This code is used for illustrative purposes. Click here to see the actual code for the Glow model in Runway.

```python
import runway
from runway import image, category, number
from fake_glow_code import manipulate_image_with_glow

# Facial feature manipulations that can be applied to the input image. Truncated for brevity.
features = '5_o_Clock_Shadow Arched_Eyebrows Attractive Bags_Under_Eyes Bald Bangs'
features = features.split()

manipulate_command_inputs = {
    'image': image(description='The input image containing a face to transform.'),
    'feature': category(choices=features, default=features[2], description='Facial feature to apply.'),
    'amount': number(default=0.5, min=0, max=1, step=0.1, description='The transformation intensity.'),
}

manipulate_command_outputs = {
    'output': image(description='The output image containing the transformed face.'),
}

manipulate_command_desc = 'Manipulate an image of a face by applying facial feature transformations.'

@runway.command('manipulate',
                inputs=manipulate_command_inputs,
                outputs=manipulate_command_outputs,
                description=manipulate_command_desc)
def manipulate(model, args):
    return {
        'output': manipulate_image_with_glow(args)
    }
```

**Input Sources and Output Destinations**

Model porters define the types of input a model receives and the types of output a model produces, but they do not define the sources of those inputs or the destinations for those outputs. When a user runs a model that accepts an image as input and produces another image as output the user choses where that image comes from before it’s sent to the model and where it goes once the model produces a new image as output. This concept applies to all data types, not just images.
Input Sources

- **Camera**: A webcam or USB camera can be used as an input source for all models that accept an image as input.
- **Text**: A simple text area box for typing or pasting text based input.
- **Vector**: A fixed-width z-vector of floats used as the seed for some generative models like StyleGAN
- **Segmentation**: A drawing tool for creating images composed of objects that represent semantic classes. See SPADE-COCO for an example of this input source.
- **File**: File is unique because it’s used to load files from disk in a uniform way across all Data Types. This input source is used to feed a model input from your filesystem.
- **From Network**: This input source is also unique in that it appears whenever you interact with a model via the network. This input source is available for all models but only appears once model a model has been triggered by a network request.

Note: Both the File and From Network input sources are unique in that they aren’t made available to users as a result something the model porter defines in a @runway.command() decorator, but rather they are **always** available to users to load their own data of arbitrary type. From Network will only appear once a model has been triggered by a network request.

Output Destinations

- **Preview**: A basic preview window for users to view the output of the model inside of Runway. Preview renders output differently depending on the the output type: For instance, it will render an image if a model outputs an image and text if a model outputs text.
- **Export**: Save model output to disk as an exported video, image sequence, or structured text file like CSV or JSON depending on the output data type.

Model Setup Options vs Inference Options

The data types specified in the options keyword argument of @runway.setup() and the inputs keyword arguments of @runway.command() functions are handled somewhat differently by Runway. The former defines options that a user configures **before** the model is started, while the later can be adjusted at any time while the model is running.

```python
import runway
from runway import category, text
from elsewhere import model

setup_options={
    'architecture': category(choices=['Artistic', 'Stable', 'Video'])
}
@runway.setup(options=setup_options)
def setup(opts):
    return model(opts['architecture'])

generate_inputs={
    'render_factor': number(min=1, max=50, default=35)
}
```

(continues on next page)
As a model porter you must decide which of the options that are configurable by the user should be setup options and which should be inference options. A good rule of thumb is that if an option is needed to load or initialize a model it should be a setup option. Otherwise it should be an inference option.

**Note:** The same data types and UI components are used in both Model Setup Options and Inference Options. The only difference is that Setup Model options can only be selected before you run the model.

### UI Components

Only a limited set of all Data Types generate UI components at this time. We are working quickly to render all data types in the app, but we ask for your patience until this task is complete. If you have any questions about data types, UI components, or the Model SDK in general feel free to bring them up in the #model-sdk channel in the public Runway slack.

### Support/Compatibility Matrix

* Denotes this feature is coming soon!

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Setup Option</th>
<th>Command Input</th>
<th>Command Output</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>image</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>The image type is also used to process video sequences.</td>
</tr>
<tr>
<td>text</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>Yes</td>
<td>Yes</td>
<td>No*</td>
<td>Boolean data types used as input appear on the model options UI on the right side of the workspace view.</td>
</tr>
<tr>
<td>category</td>
<td>Yes</td>
<td>Yes</td>
<td>No*</td>
<td>Category data types used as input appear on the model options UI on the right side of the workspace view. If you are looking to output a category, instead output the category value as a string using the <code>text</code> type for now.</td>
</tr>
<tr>
<td>number</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Number data types used as input appear on the model options UI on the right side of the workspace view. If you are looking to output a number, instead output a number as a string using the <code>text</code> type for now.</td>
</tr>
<tr>
<td>vector</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>The vector input uses a grid of images. More data types will be supported soon. See StyleGAN for an example.</td>
</tr>
<tr>
<td>segmentation</td>
<td>No*</td>
<td>Yes</td>
<td>Yes</td>
<td>Use a regular image as output for now.</td>
</tr>
<tr>
<td>file*</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>* The file input should not be used as an input or output type for <code>@runway.command()</code>'s. Instead the file UI component will be rendered as an optional input source for all types.</td>
</tr>
<tr>
<td>array</td>
<td>No</td>
<td>No*</td>
<td>No*</td>
<td>If you are looking to output an array of strings, join them using a <code>,</code> and return a <code>text</code> type for now.</td>
</tr>
<tr>
<td>any</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>The any data type is meant to be extensible and used primarily via the Network.</td>
</tr>
</tbody>
</table>
Examples

Each data type is rendered differently in the Runway workspace UI. We’ll have a look at what each input data type in detail below.

- **Image**
- **Text**
- **Vector**
- **Segmentation**
- **Number**
- **Boolean**

### Image

```python
import runway
from runway import image

# this is a fake import for demonstration purposes only
from elsewhere import process_input_image

# The "image" and "output" keys in the input and output dictionaries are arbitrarily named
@runway.command('example', inputs={ 'image': image }, outputs={ 'output': image })
def example(model, args):
    return process_input_image(args['image'])
```

The above code will generate an image UI component. The user will then be able to choose either a camera or an image file as the source of the input.

Each input image type also creates an Image Options UI component on the right panel of the workspace view. This panel gives the users the option to resize and manipulate the input image before its sent to the model. The model porter does not have control over these parameters as they are up to the user to chose and will have already been applied to the image by the time it reaches the Model SDK code.
Text

```python
import runway
from runway import text

# The "caption" and "informalized" keys in the input and output dictionaries are arbitrarily named
@runway.command('informalize', inputs={ 'caption': text }, outputs={ 'informalized': text })
def informalize(model, args):
    return args['caption'].lower()
```

The above code will generate a text UI component. The user will be able to type a string of text into the text box, and that text will be used as the caption input to the model.

![Text UI component](image)

Vector

The code below illustrates how to use the vector data type to explore a hyperparameter space by incrementally adding noise to fixed-width z-vector that’s used as input to a generative model. This data type is a bit tricky to understand if you don’t have much experience with generative models, but have a look at StyleGAN for an example of its use.

```python
import runway
from runway import vector, image
# this is a fake import for demonstration purposes only
from elsewhere import sample_image_from_z_vector

# The z" and "output" keys in the input and output dictionaries are arbitrarily named
@runway.command('sample', inputs={ 'z': vector }, outputs={ 'output': image })
def sample(model, args):
    return sample_image_from_z_vector(args['z'])
```

2.2. Example runway.yml
Segmentation

The segmentation data type is used to define per-pixel labels to an image. Each pixel is annotated with a label id from 0-255, each corresponding to a different object class. Models like NVIDIA's SPADE use this data type to generate photorealistic output images using input semantic segmentation map images of only a handful of colors.

```python
import runway
from runway import image, segmentation

# this is a fake import for demonstration purposes only
from elsewhere import generate_image_from_semantic_map

label_to_id = {
    'unlabeled': 0,
    'grass': 124,
    'sky': 156,
    'clouds': 105,
    'sea': 154,
    'river': 148,
    'tree': 169,
    'mountain': 134
}

label_to_color = {
    'unlabeled': (0, 0, 0),
    'grass': (29, 195, 49),
    'sky': (95, 219, 255),
    'clouds': (170, 170, 170),
    'sea': (54, 62, 167),
}
```

(continues on next page)
Using the `segmentation` data type as input to a command generates a UI component that allows a user to draw directly on a canvas using primitive painting tools and colors that map to the semantic map labels.
Number

A number used as input to a model command creates a number slider on the right Model Options UI panel. This data type is often used to pass an adjustable scalar value to a model at runtime, as is illustrated below.

```python
import runway
from runway import number, image
from elsewhere import stylize

stylize_inputs = {
    'input_image': image,
    'style_image': image,
    'amount': number(min=0, max=100, default=50)
}

stylize_outputs = {
    'stylized_image': image
}

@runway.command('stylize_image',
    inputs=stylize_inputs,
    outputs=stylize_outputs,
    description='Stylize an image by a certain amount using a style image')
def stylize_image(model, args):
    return stylize(args['input_image'], args['style_image'], args['amount'])
```

Here is an example of the number slider produced by the code above.

Boolean

Using the boolean data type as input to a model command generates a simple switch button on the right Model Options UI panel.

```python
import runway
from runway import boolean, text
from elsewhere import translate, tokenize

translate_inputs = {
    'english_input': image,
    'tokenize': boolean(default=True),
}

translate_outputs = {
    'french_output': text
}

@runway.command('translate',
    inputs=translate_inputs,
    outputs=translate_outputs,
    description='Translate sentences from English to French')
def translate(model, args):
    english = args['english_input']
    if args['tokenize']:
        english = tokenize(english)
    return translate(english, 'french')
```
Here is an example of the number slider produced by the code above.

### 2.2.6 Example Models

In order to better understand the process of porting a model to Runway, we recommend checking out the source code for some of the models that have already been ported. All models published by the runway organization are open source, as well as many of the models contributed by our community.

- **Image-Super-Resolution**: A very simple image upscaling model that gives a good overview of the role of `runway_model.py`.
- **StyleGAN**: A good example of loading a model checkpoint as well as using the `vector data type`.
- **SPADE-COCO**: A good example of the `segmentation data type`.
- **DeepLabV3**: A good example of multiple `@command()` functions and conditional build steps depending on GPU and CPU build environments. Also a very simple model to get started with.
- **Places365**: A good example of a basic image classification task and use of the `text data type` for output.
- **3DDFA**: A good example of dealing with 3D data as images. We plan to add more features for handling true 3D data.