Clipper Documentation

Release 0.2.0rc1

Dan Crankshaw

Jun 06, 2018

Python API Documentation

This is the API Documentation for Clipper.

CHAPTER 1

Clipper Connection

`ClipperConnection` is the primary way of starting and managing a Clipper cluster.

class `clipper_admin.`**`ClipperConnection`**(*container_manager*)

`__init__`(*container_manager*)

Create a new ClipperConnection object.

After creating a `ClipperConnection` instance, you still need to connect to a Clipper cluster. You can connect to an existing cluster by calling *`clipper_admin.ClipperConnection.connect()`* or create a new Clipper cluster with *`clipper_admin.ClipperConnection.start_clipper()`*, which will automatically connect to the cluster once it Clipper has successfully started.

Parameters **`container_manager`** (`clipper_admin.container_manager.ContainerManager`) – An instance of a concrete subclass of `ContainerManager`.

`start_clipper`(*query_frontend_image='clipper/query_frontend:develop'*, *mgmt_frontend_image='clipper/management_frontend:develop'*, *cache_size=33554432*, *num_frontend_replicas=1*)

Start a new Clipper cluster and connect to it.

This command will start a new Clipper instance using the container manager provided when the `ClipperConnection` instance was constructed.

Parameters

- **`query_frontend_image`** (*`str (optional)`*) – The query frontend docker image to use. You can set this argument to specify a custom build of the query frontend, but any customization should maintain API compability and preserve the expected behavior of the system.
- **`mgmt_frontend_image`** (*`str (optional)`*) – The management frontend docker image to use. You can set this argument to specify a custom build of the management frontend, but any customization should maintain API compability and preserve the expected behavior of the system.

- **cache_size** (*int, optional*) – The size of Clipper's prediction cache in bytes. Default cache size is 32 MiB.

Raises `clipper.ClipperException`

connect()

Connect to a running Clipper cluster.

register_application(*name*, *input_type*, *default_output*, *slo_micros*)

Register a new application with Clipper.

An application in Clipper corresponds to a named REST endpoint that can be used to request predictions. This command will attempt to create a new endpoint with the provided name. Application names must be unique. This command will fail if an application with the provided name already exists.

Parameters

- **name** (*str*) – The unique name of the application.
- **input_type** (*str*) – The type of the request data this endpoint can process. Input type can be one of "integers", "floats", "doubles", "bytes", or "strings".
- **default_output** (*str*) – The default output for the application. The default output will be returned whenever an application is unable to receive a response from a model within the specified query latency SLO (service level objective). The reason the default output was returned is always provided as part of the prediction response object.
- **slo_micros** (*int*) – The query latency objective for the application in microseconds. This is the processing latency between Clipper receiving a request and sending a response. It does not account for network latencies before a request is received or after a response is sent. If Clipper cannot process a query within the latency objective, the default output is returned. Therefore, it is recommended that the SLO not be set aggressively low unless absolutely necessary. 100000 (100ms) is a good starting value, but the optimal latency objective will vary depending on the application.

Raises

- `clipper.UnconnectedException`
- `clipper.ClipperException`

delete_application(*name*)

link_model_to_app(*app_name*, *model_name*)

Routes requests from the specified app to be evaluted by the specified model.

Parameters

- **app_name** (*str*) – The name of the application
- **model_name** (*str*) – The name of the model to link to the application

Raises

- `clipper.UnconnectedException`
- `clipper.ClipperException`

Note: Both the specified model and application must be registered with Clipper, and they must have the same input type. If the application has previously been linked to a different model, this command will fail.

build_and_deploy_model(*name*, *version*, *input_type*, *model_data_path*, *base_image*, *labels=None*, *container_registry=None*, *num_replicas=1*, *batch_size=-1*, *pkgs_to_install=None*)

Build a new model container Docker image with the provided data and deploy it as a model to Clipper.

This method does two things.

1. Builds a new Docker image from the provided base image with the local directory specified by `model_data_path` copied into the image by calling *`clipper_admin.ClipperConnection.build_model()`*.

2. Registers and deploys a model with the specified metadata using the newly built image by calling *`clipper_admin.ClipperConnection.deploy_model()`*.

Parameters

- **name** (*str*) – The name of the deployed model
- **version** (*str*) – The version to assign this model. Versions must be unique on a per-model basis, but may be re-used across different models.
- **input_type** (*str*) – The type of the request data this endpoint can process. Input type can be one of "integers", "floats", "doubles", "bytes", or "strings". See the User Guide for more details on picking the right input type for your application.
- **model_data_path** (*str*) – A path to a local directory. The contents of this directory will be recursively copied into the Docker container.
- **base_image** (*str*) – The base Docker image to build the new model image from. This image should contain all code necessary to run a Clipper model container RPC client.
- **labels** (*list(str), optional*) – A list of strings annotating the model. These are ignored by Clipper and used purely for user annotations.
- **container_registry** (*str, optional*) – The Docker container registry to push the freshly built model to. Note that if you are running Clipper on Kubernetes, this registry must be accesible to the Kubernetes cluster in order to fetch the container from the registry.
- **num_replicas** (*int, optional*) – The number of replicas of the model to create. The number of replicas for a model can be changed at any time with `clipper.ClipperConnection.set_num_replicas()`.
- **batch_size** (*int, optional*) – The user-defined query batch size for the model. Replicas of the model will attempt to process at most *batch_size* queries simultaneously. They may process smaller batches if *batch_size* queries are not immediately available. If the default value of -1 is used, Clipper will adaptively calculate the batch size for individual replicas of this model.
- **pkgs_to_install** (*list (of strings), optional*) – A list of the names of packages to install, using pip, in the container. The names must be strings.

Raises

- `clipper.UnconnectedException`
- `clipper.ClipperException`

build_model(*name*, *version*, *model_data_path*, *base_image*, *container_registry=None*, *pkgs_to_install=None*)

Build a new model container Docker image with the provided data"

This method builds a new Docker image from the provided base image with the local directory specified by `model_data_path` copied into the image. The Dockerfile that gets generated to build the image is equivalent to the following:

```
FROM <base_image>
COPY <model_data_path> /model/
```

The newly built image is then pushed to the specified container registry. If no container registry is specified, the image will be pushed to the default DockerHub registry. Clipper will tag the newly built image with the tag [<registry>]/<name>:<version>.

This method can be called without being connected to a Clipper cluster.

Parameters

- **name** (*str*) – The name of the deployed model.
- **version** (*str*) – The version to assign this model. Versions must be unique on a per-model basis, but may be re-used across different models.
- **model_data_path** (*str*) – A path to a local directory. The contents of this directory will be recursively copied into the Docker container.
- **base_image** (*str*) – The base Docker image to build the new model image from. This image should contain all code necessary to run a Clipper model container RPC client.
- **container_registry** (*str, optional*) – The Docker container registry to push the freshly built model to. Note that if you are running Clipper on Kubernetes, this registry must be accesible to the Kubernetes cluster in order to fetch the container from the registry.
- **pkgs_to_install** (*list (of strings), optional*) – A list of the names of packages to install, using pip, in the container. The names must be strings.

Returns The fully specified tag of the newly built image. This will include the container registry if specified.

Return type str

Raises `clipper.ClipperException`

Note: Both the model name and version must be valid DNS-1123 subdomains. Each must consist of lower case alphanumeric characters, '-' or '.', and must start and end with an alphanumeric character (e.g. 'example.com', regex used for validation is '[a-z0-9]([-a-z0-9]*[a-z0-9])?Z'.

deploy_model(*name*, *version*, *input_type*, *image*, *labels=None*, *num_replicas=1*, *batch_size=-1*)

Deploys the model in the provided Docker image to Clipper.

Deploying a model to Clipper does a few things.

1. It starts a set of Docker model containers running the model packaged in the `image` Docker image. The number of containers it will start is dictated by the `num_replicas` argument, but the way that these containers get started depends on your choice of `ContainerManager` implementation.

2. It registers the model and version with Clipper and sets the current version of the model to this version by internally calling *`clipper_admin.ClipperConnection.register_model()`*.

Notes

If you want to deploy a model in some other way (e.g. a model that cannot run in a Docker container for some reason), you can start the model manually or with an external tool and call `register_model` directly.

Parameters

- **name** (*str*) – The name of the deployed model
- **version** (*str*) – The version to assign this model. Versions must be unique on a per-model basis, but may be re-used across different models.
- **input_type** (*str*) – The type of the request data this endpoint can process. Input type can be one of "integers", "floats", "doubles", "bytes", or "strings". See the User Guide for more details on picking the right input type for your application.
- **image** (*str*) – The fully specified Docker image to deploy. If using a custom registry, the registry name must be prepended to the image. For example, if your Docker image is stored in the quay.io registry, you should specify the image argument as "quay.io/my_namespace/image_name:tag". The image name and tag are independent of the `name` and `version` arguments, and can be set to whatever you want.
- **labels** (*list(str), optional*) – A list of strings annotating the model. These are ignored by Clipper and used purely for user annotations.
- **num_replicas** (*int, optional*) – The number of replicas of the model to create. The number of replicas for a model can be changed at any time with `clipper.ClipperConnection.set_num_replicas()`.
- **batch_size** (*int, optional*) – The user-defined query batch size for the model. Replicas of the model will attempt to process at most *batch_size* queries simultaneously. They may process smaller batches if *batch_size* queries are not immediately available. If the default value of -1 is used, Clipper will adaptively calculate the batch size for individual replicas of this model.

Raises

- `clipper.UnconnectedException`
- `clipper.ClipperException`

Note: Both the model name and version must be valid DNS-1123 subdomains. Each must consist of lower case alphanumeric characters, '-' or '.', and must start and end with an alphanumeric character (e.g. 'example.com', regex used for validation is '[a-z0-9]([-a-z0-9]*[a-z0-9])?Z'.

register_model (*name*, *version*, *input_type*, *image=None*, *labels=None*, *batch_size=-1*)

Registers a new model version with Clipper.

This method does not launch any model containers, it only registers the model description (metadata such as name, version, and input type) with Clipper. A model must be registered with Clipper before it can be linked to an application.

You should rarely have to use this method directly. Using one the Clipper deployer methods in `clipper_admin.deployers` or calling `build_and_deploy_model` or `deploy_model` will automatically register your model with Clipper.

Parameters

- **name** (*str*) – The name of the deployed model
- **version** (*str*) – The version to assign this model. Versions must be unique on a per-model basis, but may be re-used across different models.
- **input_type** (*str*) – The type of the request data this endpoint can process. Input type can be one of "integers", "floats", "doubles", "bytes", or "strings". See the User Guide for more details on picking the right input type for your application.

- **image** (*str, optional*) – A docker image name. If provided, the image will be recorded as part of the model descrtipin in Clipper when registering the model but this method will make no attempt to launch any containers with this image.
- **labels** (*list(str), optional*) – A list of strings annotating the model. These are ignored by Clipper and used purely for user annotations.
- **batch_size** (*int, optional*) – The user-defined query batch size for the model. Replicas of the model will attempt to process at most *batch_size* queries simultaneously. They may process smaller batches if *batch_size* queries are not immediately available. If the default value of -1 is used, Clipper will adaptively calculate the batch size for individual replicas of this model.

Raises

- `clipper.UnconnectedException`
- `clipper.ClipperException`

get_current_model_version(*name*)

Get the current model version for the specified model.

Parameters **name** (*str*) – The name of the model

Returns The current model version

Return type str

Raises

- `clipper.UnconnectedException`
- `clipper.ClipperException`

get_num_replicas(*name*, *version=None*)

Gets the current number of model container replicas for a model.

Parameters

- **name** (*str*) – The name of the model
- **version** (*str, optional*) – The version of the model. If no version is provided, the currently deployed version will be used.

Returns The number of active replicas

Return type int

Raises

- `clipper.UnconnectedException`
- `clipper.ClipperException`

set_num_replicas(*name*, *num_replicas*, *version=None*)

Sets the total number of active replicas for a model.

If there are more than the current number of replicas currently allocated, this will remove replicas. If there are less, this will add replicas.

Parameters

- **name** (*str*) – The name of the model
- **version** (*str, optional*) – The version of the model. If no version is provided, the currently deployed version will be used.

- **num_replicas** (*int, optional*) – The desired number of replicas.

Raises

- `clipper.UnconnectedException`
- `clipper.ClipperException`

get_all_apps(*verbose=False*)
Gets information about all applications registered with Clipper.

Parameters **verbose** (*bool*) – If set to False, the returned list contains the apps' names. If set to True, the list contains application info dictionaries. These dictionaries have the same attribute name-value pairs that were provided to *clipper_admin.ClipperConnection.register_application()*.

Returns Returns a list of information about all apps registered to Clipper. If no apps are registered with Clipper, an empty list is returned.

Return type list

Raises

- `clipper.UnconnectedException`
- `clipper.ClipperException`

get_app_info(*name*)
Gets detailed information about a registered application.

Parameters **name** (*str*) – The name of the application to look up

Returns Returns a dictionary with the specified application's info. This will contain the attribute name-value pairs that were provided to *clipper_admin.ClipperConnection.register_application()*. If no application with name `name` is registered with Clipper, None is returned.

Return type dict

Raises `clipper.UnconnectedException`

get_linked_models(*app_name*)
Retrieves the models linked to the specified application.

Parameters **app_name** (*str*) – The name of the application

Returns Returns a list of the names of models linked to the app. If no models are linked to the specified app, None is returned.

Return type list

Raises

- `clipper.UnconnectedException`
- `clipper.ClipperException`

get_all_models(*verbose=False*)
Gets information about all models registered with Clipper.

Parameters **verbose** (*bool*) – If set to False, the returned list contains the models' names. If set to True, the list contains model info dictionaries.

Returns Returns a list of information about all apps registered to Clipper. If no models are registered with Clipper, an empty list is returned.

Return type list

Raises

- `clipper.UnconnectedException`
- `clipper.ClipperException`

`get_model_info`(*name*, *version*)

Gets detailed information about a registered model.

Parameters

- **`model_name`** (*`str`*) – The name of the model to look up
- **`model_version`** (*`int`*) – The version of the model to look up

Returns Returns a dictionary with the specified model's info. If no model with name *model_name@model_version* is registered with Clipper, None is returned.

Return type dict

Raises

- `clipper.UnconnectedException`
- `clipper.ClipperException`

`get_all_model_replicas`(*verbose=False*)

Gets information about all model containers registered with Clipper.

Parameters **`verbose`** (*`bool`*) – If set to False, the returned list contains the apps' names. If set to True, the list contains container info dictionaries.

Returns Returns a list of information about all model containers known to Clipper. If no containers are registered with Clipper, an empty list is returned.

Return type list

Raises

- `clipper.UnconnectedException`
- `clipper.ClipperException`

`get_model_replica_info`(*name*, *version*, *replica_id*)

Gets detailed information about a registered container.

Parameters

- **`name`** (*`str`*) – The name of the container to look up
- **`version`** (*`int`*) – The version of the container to look up
- **`replica_id`** (*`int`*) – The container replica to look up

Returns A dictionary with the specified container's info. If no corresponding container is registered with Clipper, None is returned.

Return type dict

Raises

- `clipper.UnconnectedException`
- `clipper.ClipperException`

`get_clipper_logs`(*logging_dir='clipper_logs/'*)

Download the logs from all Clipper docker containers.

Parameters **`logging_dir`** (*`str, optional`*) – The directory to save the downloaded logs. If the directory does not exist, it will be created.

Raises `clipper.UnconnectedException`

`inspect_instance`()

Fetches performance metrics from the running Clipper cluster.

Returns The JSON string containing the current set of metrics for this instance. On error, the string will be an error message (not JSON formatted).

Return type str

Raises

- `clipper.UnconnectedException`
- `clipper.ClipperException`

`set_model_version`(*name*, *version*, *num_replicas=None*)

Changes the current model version to "model_version".

This method can be used to perform model roll-back and roll-forward. The version can be set to any previously deployed version of the model.

Parameters

- **`name`** (*`str`*) – The name of the model
- **`version`** (*`str | obj with __str__ representation`*) – The version of the model. Note that *version* must be a model version that has already been deployed.
- **`num_replicas`** (*`int`*) – The number of new containers to start with the newly selected model version.

Raises

- `clipper.UnconnectedException`
- `clipper.ClipperException`

Note: Model versions automatically get updated when py:meth:*clipper_admin.ClipperConnection.deploy_model()* is called. There is no need to manually update the version after deploying a new model.

`get_query_addr`()

Get the IP address at which the query frontend can be reached request predictions.

Returns The address as an IP address or hostname.

Return type str

Raises `clipper.UnconnectedException` – versions. All replicas for each version of each model will be stopped.

`stop_models`(*model_names*)

Stops all versions of the specified models.

This is a convenience method to avoid the need to explicitly list all versions of a model when calling *`clipper_admin.ClipperConnection.stop_versioned_models()`*.

Parameters **`model_names`** (*`list(str)`*) – A list of model names. All replicas of all versions of each model specified in the list will be stopped.

Raises `clipper.UnconnectedException` – versions. All replicas for each version of each model will be stopped.

`stop_versioned_models`(*model_versions_dict*)

Stops the specified versions of the specified models.

Parameters **`model_versions_dict`** (*`dict(str, list(str))`*) – For each entry in the dict, the key is a model name and the value is a list of model

Raises `clipper.UnconnectedException` – versions. All replicas for each version of each model will be stopped.

Note: This method will stop the currently deployed versions of models if you specify them. You almost certainly want to use one of the other stop_* methods. Use with caution.

`stop_inactive_model_versions`(*model_names*)

Stops all model containers serving stale versions of the specified models.

For example, if you have deployed versions 1, 2, and 3 of model "music_recommender" and version 3 is the current version:

```
clipper_conn.stop_inactive_model_versions(["music_recommender"])
```

will stop any containers serving versions 1 and 2 but will leave containers serving version 3 untouched.

Parameters **`model_names`** (*`list(str)`*) – The names of the models whose old containers you want to stop.

Raises `clipper.UnconnectedException`

`stop_all_model_containers`()

Stops all model containers started via Clipper admin commands.

This method can be used to clean up leftover Clipper model containers even if the Clipper management frontend or Redis has crashed. It can also be called without calling `connect` first.

`stop_all`()

Stops all processes that were started via Clipper admin commands.

This includes the query and management frontend Docker containers and all model containers. If you started Redis independently, this will not affect Redis. It can also be called without calling `connect` first.

`test_predict_function`(*query*, *func*, *input_type*)

Tests that the user's function has the correct signature and can be properly saved and loaded.

The function should take a dict request object like the query frontend expects JSON, the predict function, and the input type for the model.

For example, the function can be called like: clipper_conn.test_predict_function({"input": [1.0, 2.0, 3.0]}, predict_func, "doubles")

Parameters

- **`query`** (*`JSON or list of dicts`*) – Inputs to test the prediction function on.
- **`func`** (*`function`*) – Predict function to test.
- **`input_type`** (*`str`*) – The input_type to be associated with the registered app and deployed model. One of "integers", "floats", "doubles", "bytes", or "strings".

CHAPTER 2

Container Managers

Container managers abstract away the low-level tasks of creating and managing Clipper's Docker containers in order to allow Clipper to be deployed with different container orchestration frameworks. Clipper currently provides two container manager implementations, one for running Clipper locally directly on Docker, and one for running Clipper on Kubernetes.

class clipper_admin.**DockerContainerManager**(*docker_ip_address='localhost'*, *clipper_query_port=1337*, *clipper_management_port=1338*, *clipper_rpc_port=7000*, *redis_ip=None*, *redis_port=6379*, *prometheus_port=9090*, *docker_network='clipper_network'*, *extra_container_kwargs={}*)

Bases: `clipper_admin.container_manager.ContainerManager`

__init__(*docker_ip_address='localhost'*, *clipper_query_port=1337*, *clipper_management_port=1338*, *clipper_rpc_port=7000*, *redis_ip=None*, *redis_port=6379*, *prometheus_port=9090*, *docker_network='clipper_network'*, *extra_container_kwargs={}*)

Parameters

- **docker_ip_address** (*str, optional*) – The public hostname or IP address at which the Clipper Docker containers can be accessed via their exposed ports. This should almost always be "localhost". Only change if you know what you're doing!
- **clipper_query_port** (*int, optional*) – The port on which the query frontend should listen for incoming prediction requests.
- **clipper_management_port** (*int, optional*) – The port on which the management frontend should expose the management REST API.
- **clipper_rpc_port** (*int, optional*) – The port to start the Clipper RPC service on.
- **redis_ip** (*str, optional*) – The address of a running Redis cluster. If set to None, Clipper will start a Redis container for you.

- **`redis_port`** (*`int, optional`*) – The Redis port. If `redis_ip` is set to None, Clipper will start Redis on this port. If `redis_ip` is provided, Clipper will connect to Redis on this port.
- **`docker_network`** (*`str, optional`*) – The docker network to attach the containers to. You can read more about Docker networking in the Docker User Guide.
- **`extra_container_kwargs`** (*`dict`*) – Any additional keyword arguments to pass to the call to `docker.client.containers.run()`.

`get_logs`(*logging_dir*)

Get the container logs for all Docker containers launched by Clipper.

This will get the logs for both Clipper core containers and any model containers deployed by Clipper admin. Any previous log files from existing containers will be overwritten.

Parameters **`logging_dir`** (*`str`*) – The directory to write the log files to. If the directory does not exist, it will be created.

Returns The list of all log files created.

Return type list(str)

`stop_models`(*models*)

Stops all replicas of the specified models.

Parameters **`models`** (*`dict(str, list(str))`*) – For each entry in the dict, the key is a model name and the value is a list of model versions. All replicas for each version of each model will be stopped.

`stop_all`()

Stop all resources associated with Clipper.

class `clipper_admin.`**`KubernetesContainerManager`**(*kubernetes_proxy_addr=None, redis_ip=None, redis_port=6379, useInternalIP=False, namespace='default', create_namespace_if_not_exists=False*)

Bases: `clipper_admin.container_manager.ContainerManager`

`__init__`(*kubernetes_proxy_addr=None, redis_ip=None, redis_port=6379, useInternalIP=False, namespace='default', create_namespace_if_not_exists=False*)

Parameters

- **`kubernetes_proxy_addr`** (*`str, optional`*) – The proxy address if you are proxying connections locally using `kubectl proxy`. If this argument is provided, Clipper will construct the appropriate proxy URLs for accessing Clipper's Kubernetes services, rather than using the API server addres provided in your kube config.
- **`redis_ip`** (*`str, optional`*) – The address of a running Redis cluster. If set to None, Clipper will start a Redis deployment for you.
- **`redis_port`** (*`int, optional`*) – The Redis port. If `redis_ip` is set to None, Clipper will start Redis on this port. If `redis_ip` is provided, Clipper will connect to Redis on this port.
- **`useInternalIP`** (*`bool, optional`*) – Use Internal IP of the K8S nodes . If `useInternalIP` is set to False, Clipper will throw an exception if none of the nodes have ExternalDNS. If `useInternalIP` is set to true, Clipper will use the Internal IP of the K8S node if no ExternalDNS exists for any of the nodes.

- **`namespace`** (*`str, optional`*) – The Kubernetes namespace to use . If this argument is provided, all Clipper artifacts and resources will be created in this k8s namespace. If not "default" namespace is used.
- **`create_namespace_if_not_exists`** (*`bool, False`*) – Create a k8s namespace if the namespace doesnt already exist. If this argument is provided and the k8s namespace does not exist a new k8s namespace will be created.

Note: Clipper stores all persistent configuration state (such as registered application and model information) in Redis. If you want Clipper to be durable and able to recover from failures, we recommend configuring your own persistent and replicated Redis cluster rather than letting Clipper launch one for you.

`get_logs`(*logging_dir*)

Get the container logs for all Docker containers launched by Clipper.

This will get the logs for both Clipper core containers and any model containers deployed by Clipper admin. Any previous log files from existing containers will be overwritten.

Parameters **`logging_dir`** (*`str`*) – The directory to write the log files to. If the directory does not exist, it will be created.

Returns The list of all log files created.

Return type list(str)

`stop_models`(*models*)

Stops all replicas of the specified models.

Parameters **`models`** (*`dict(str, list(str))`*) – For each entry in the dict, the key is a model name and the value is a list of model versions. All replicas for each version of each model will be stopped.

`stop_all`()

Stop all resources associated with Clipper.

CHAPTER 3

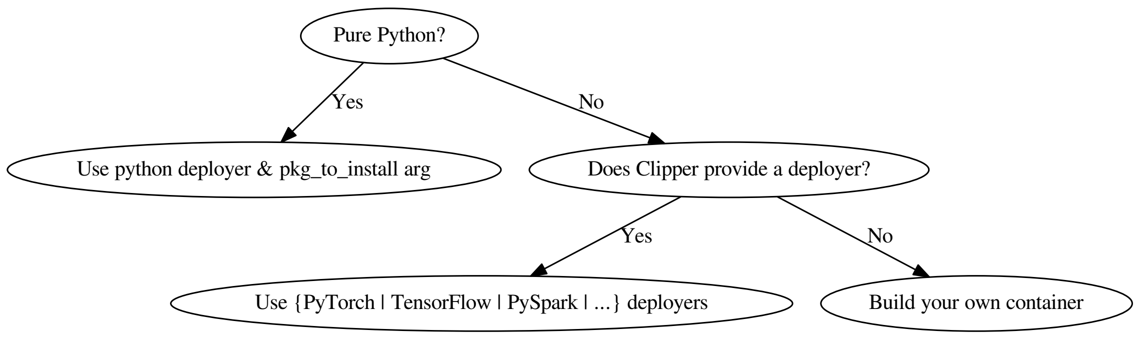
Model Deployers

Clipper provides a collection of model deployer modules to simplify the process of deploying a trained model to Clipper and avoid the need to figure out how to save models and build custom Docker containers capable of serving the saved models for some common use cases. With these modules, you can deploy models directly from Python to Clipper.

Currently, Clipper provides the following deployer modules:

1. Arbitrary Python functions
2. PySpark Models
3. PyTorch Models
4. Tensorflow Models
5. MXNet Models
6. PyTorch Models exported as ONNX file with Caffe2 Serving Backend (Experimental)

These deployers support function that can only be pickled using Cloudpickle and/or pure python libraries that can be installed via *pip*. For reference, please use the following flowchart to make decision about which deployer to use.


Pure Python?
Yes
No
Use python deployer & pkg_to_install arg
Does Clipper provide a deployer?
Yes
No
Use {PyTorch | TensorFlow | PySpark | ...} deployers
Build your own container

Note: You can find additional examples of using model deployers in Clipper's integration tests.

3.1 Pure Python functions

This module supports deploying pure Python function closures to Clipper. A function deployed with this module must take a list of inputs as the sole argument, and return a list of strings of exactly the same length. The reason the prediction function takes a list of inputs rather than a single input is to provide models the possibility of computing multiple predictions in parallel to improve model performance. For example, many models that run on a GPU can significantly improve throughput by batching predictions to better utilize the many parallel cores of the GPU.

In addition, the function must only use pure Python code. More specifically, all of the state captured by the function will be pickled using Cloudpickle, so any state captured by the function must be able to be pickled. Most Python libraries that use C extensions create objects that cannot be pickled. This includes many common machine-learning frameworks such as PySpark, TensorFlow, PyTorch, and Caffe. You will have to use Clipper provided containers or create your own Docker containers and call the native serialization libraries of these frameworks in order to deploy them.

While this deployer will serialize your function, any Python libraries that the function depends on must be installed in the container to be able to load the function inside the model container. You can specify these libraries using the `pkgs_to_install` argument. All the packages specified by that argument will be installed in the container with pip prior to running it.

If your function has dependencies that cannot be installed directly with pip, you will need to build your own container.

`clipper_admin.deployers.python.`**`deploy_python_closure`**(*clipper_conn*, *name*, *version*, *input_type*, *func*, *base_image='default'*, *labels=None*, *registry=None*, *num_replicas=1*, *batch_size=-1*, *pkgs_to_install=None*)

Deploy an arbitrary Python function to Clipper.

The function should take a list of inputs of the type specified by *input_type* and return a Python list or numpy array of predictions as strings.

Parameters

- **clipper_conn** (*clipper_admin.ClipperConnection()*) – A `ClipperConnection` object connected to a running Clipper cluster.
- **name** (*str*) – The name to be assigned to both the registered application and deployed model.
- **version** (*str*) – The version to assign this model. Versions must be unique on a per-model basis, but may be re-used across different models.
- **input_type** (*str*) – The input_type to be associated with the registered app and deployed model. One of "integers", "floats", "doubles", "bytes", or "strings".
- **func** (*function*) – The prediction function. Any state associated with the function will be captured via closure capture and pickled with Cloudpickle.
- **base_image** (*str, optional*) – The base Docker image to build the new model image from. This image should contain all code necessary to run a Clipper model container RPC client.

- **labels** (*list(str), optional*) – A list of strings annotating the model. These are ignored by Clipper and used purely for user annotations.
- **registry** (*str, optional*) – The Docker container registry to push the freshly built model to. Note that if you are running Clipper on Kubernetes, this registry must be accesible to the Kubernetes cluster in order to fetch the container from the registry.
- **num_replicas** (*int, optional*) – The number of replicas of the model to create. The number of replicas for a model can be changed at any time with `clipper.ClipperConnection.set_num_replicas()`.
- **batch_size** (*int, optional*) – The user-defined query batch size for the model. Replicas of the model will attempt to process at most *batch_size* queries simultaneously. They may process smaller batches if *batch_size* queries are not immediately available. If the default value of -1 is used, Clipper will adaptively calculate the batch size for individual replicas of this model.
- **pkgs_to_install** (*list (of strings), optional*) – A list of the names of packages to install, using pip, in the container. The names must be strings.

Example

Define a pre-processing function `center()` and train a model on the pre-processed input:

```python
from clipper_admin import ClipperConnection, DockerContainerManager
from clipper_admin.deployers.python import deploy_python_closure
import numpy as np
import sklearn

clipper_conn = ClipperConnection(DockerContainerManager())

# Connect to an already-running Clipper cluster
clipper_conn.connect()

def center(xs):
    means = np.mean(xs, axis=0)
    return xs - means

centered_xs = center(xs)
model = sklearn.linear_model.LogisticRegression()
model.fit(centered_xs, ys)

# Note that this function accesses the trained model via closure capture,
# rather than having the model passed in as an explicit argument.
def centered_predict(inputs):
    centered_inputs = center(inputs)
    # model.predict returns a list of predictions
    preds = model.predict(centered_inputs)
    return [str(p) for p in preds]

deploy_python_closure(
    clipper_conn,
    name="example",
    input_type="doubles",
    func=centered_predict)
```

`clipper_admin.deployers.python.`**`create_endpoint`**(*clipper_conn*, *name*, *input_type*, *func*, *default_output='None'*, *version=1*, *slo_micros=3000000*, *labels=None*, *registry=None*, *base_image='default'*, *num_replicas=1*, *batch_size=-1*, *pkgs_to_install=None*)

Registers an application and deploys the provided predict function as a model.

Parameters

- **`clipper_conn`** (*`clipper_admin.ClipperConnection()`*) – A `ClipperConnection` object connected to a running Clipper cluster.
- **`name`** (*`str`*) – The name to be assigned to both the registered application and deployed model.
- **`input_type`** (*`str`*) – The input_type to be associated with the registered app and deployed model. One of "integers", "floats", "doubles", "bytes", or "strings".
- **`func`** (*`function`*) – The prediction function. Any state associated with the function will be captured via closure capture and pickled with Cloudpickle.
- **`default_output`** (*`str, optional`*) – The default output for the application. The default output will be returned whenever an application is unable to receive a response from a model within the specified query latency SLO (service level objective). The reason the default output was returned is always provided as part of the prediction response object. Defaults to "None".
- **`version`** (*`str, optional`*) – The version to assign this model. Versions must be unique on a per-model basis, but may be re-used across different models.
- **`slo_micros`** (*`int, optional`*) – The query latency objective for the application in microseconds. This is the processing latency between Clipper receiving a request and sending a response. It does not account for network latencies before a request is received or after a response is sent. If Clipper cannot process a query within the latency objective, the default output is returned. Therefore, it is recommended that the SLO not be set aggressively low unless absolutely necessary. 100000 (100ms) is a good starting value, but the optimal latency objective will vary depending on the application.
- **`labels`** (*`list(str), optional`*) – A list of strings annotating the model. These are ignored by Clipper and used purely for user annotations.
- **`registry`** (*`str, optional`*) – The Docker container registry to push the freshly built model to. Note that if you are running Clipper on Kubernetes, this registry must be accessible to the Kubernetes cluster in order to fetch the container from the registry.
- **`base_image`** (*`str, optional`*) – The base Docker image to build the new model image from. This image should contain all code necessary to run a Clipper model container RPC client.
- **`num_replicas`** (*`int, optional`*) – The number of replicas of the model to create. The number of replicas for a model can be changed at any time with `clipper.ClipperConnection.set_num_replicas()`.
- **`batch_size`** (*`int, optional`*) – The user-defined query batch size for the model. Replicas of the model will attempt to process at most *batch_size* queries simultaneously. They may process smaller batches if *batch_size* queries are not immediately available. If the default value of -1 is used, Clipper will adaptively calculate the batch size for individual replicas of this model.

- **pkgs_to_install** (*list (of strings), optional*) – A list of the names of packages to install, using pip, in the container. The names must be strings.

3.2 PySpark Models

The PySpark model deployer module provides a small extension to the Python closure deployer to allow you to deploy Python functions that include PySpark models as part of the state. PySpark models cannot be pickled and so they break the Python closure deployer. Instead, they must be saved using the native PySpark save and load APIs. To get around this limitation, the PySpark model deployer introduces two changes to the Python closure deployer discussed above.

First, a function deployed with this module *takes two additional arguments*: a PySpark `SparkSession` object and a PySpark model object, along with a list of inputs as provided to the Python closures in the `deployers.python` module. It must still return a list of strings of the same length as the list of inputs.

Second, the `pyspark.deploy_pyspark_model` and `pyspark.create_endpoint` deployment methods introduce two additional arguments:

- `pyspark_model`: A PySpark model object. This model will be serialized using the native PySpark serialization API and loaded into the deployed model container. The model container creates a long-lived SparkSession when it is first initialized and uses that to load this model once at initialization time. The long-lived SparkSession and loaded model are provided by the container as arguments to the prediction function each time the model container receives a new prediction request.
- `sc`: The current SparkContext. The PySpark model serialization API requires the SparkContext as an argument

The effect of these two changes is to allow the deployed prediction function to capture all pure Python state through closure capture but explicitly declare the additional PySpark state which must be saved and loaded through a separate process.

`clipper_admin.deployers.pyspark.`**`deploy_pyspark_model`**(*clipper_conn, name, version, input_type, func, pyspark_model, sc, base_image='default', labels=None, registry=None, num_replicas=1, batch_size=-1, pkgs_to_install=None*)

Deploy a Python function with a PySpark model.

The function must take 3 arguments (in order): a SparkSession, the PySpark model, and a list of inputs. It must return a list of strings of the same length as the list of inputs.

Parameters

- **clipper_conn** (*clipper_admin.ClipperConnection()*) – A `ClipperConnection` object connected to a running Clipper cluster.
- **name** (*str*) – The name to be assigned to both the registered application and deployed model.
- **version** (*str*) – The version to assign this model. Versions must be unique on a per-model basis, but may be re-used across different models.
- **input_type** (*str*) – The input_type to be associated with the registered app and deployed model. One of "integers", "floats", "doubles", "bytes", or "strings".
- **func** (*function*) – The prediction function. Any state associated with the function will be captured via closure capture and pickled with Cloudpickle.

- **pyspark_model** (*pyspark.mllib.* or pyspark.ml.pipeline.PipelineModel object*) – The PySpark model to save.
- **sc** (*SparkContext,*) – The current SparkContext. This is needed to save the PySpark model.
- **base_image** (*str, optional*) – The base Docker image to build the new model image from. This image should contain all code necessary to run a Clipper model container RPC client.
- **labels** (*list (str), optional*) – A list of strings annotating the model. These are ignored by Clipper and used purely for user annotations.
- **registry** (*str, optional*) – The Docker container registry to push the freshly built model to. Note that if you are running Clipper on Kubernetes, this registry must be accesible to the Kubernetes cluster in order to fetch the container from the registry.
- **num_replicas** (*int, optional*) – The number of replicas of the model to create. The number of replicas for a model can be changed at any time with `clipper.ClipperConnection.set_num_replicas()`.
- **batch_size** (*int, optional*) – The user-defined query batch size for the model. Replicas of the model will attempt to process at most *batch_size* queries simultaneously. They may process smaller batches if *batch_size* queries are not immediately available. If the default value of -1 is used, Clipper will adaptively calculate the batch size for individual replicas of this model.
- **pkgs_to_install** (*list (of strings), optional*) – A list of the names of packages to install, using pip, in the container. The names must be strings.

Example

Define a pre-processing function `shift()` to normalize prediction inputs:

```python
from clipper_admin import ClipperConnection, DockerContainerManager
from clipper_admin.deployers.pyspark import deploy_pyspark_model
from pyspark.mllib.classification import LogisticRegressionWithSGD
from pyspark.sql import SparkSession
import numpy as np

spark = SparkSession.builder.appName("example").getOrCreate()

sc = spark.sparkContext

clipper_conn = ClipperConnection(DockerContainerManager())

# Connect to an already-running Clipper cluster
clipper_conn.connect()

# Loading a training dataset omitted...
model = LogisticRegressionWithSGD.train(trainRDD, iterations=10)

def shift(x):
    return x - np.mean(x)

# Note that this function accesses the trained PySpark model via an explicit
# argument, but other state can be captured via closure capture if necessary.
def predict(spark, model, inputs):
```

(continues on next page)

(continued from previous page)

```
    return [str(model.predict(shift(x))) for x in inputs]

deploy_pyspark_model(
    clipper_conn,
    name="example",
    input_type="doubles",
    func=predict,
    pyspark_model=model,
    sc=sc)
```

`clipper_admin.deployers.pyspark.`**`create_endpoint`**(*clipper_conn*, *name*, *input_type*, *func*, *pyspark_model*, *sc*, *default_output='None'*, *version=1*, *slo_micros=3000000*, *labels=None*, *registry=None*, *base_image='default'*, *num_replicas=1*, *batch_size=-1*, *pkgs_to_install=None*)

Registers an app and deploys the provided predict function with PySpark model as a Clipper model.

Parameters

- **clipper_conn** (*clipper_admin.ClipperConnection()*) – A `ClipperConnection` object connected to a running Clipper cluster.
- **name** (*str*) – The name to be assigned to both the registered application and deployed model.
- **input_type** (*str*) – The input_type to be associated with the registered app and deployed model. One of "integers", "floats", "doubles", "bytes", or "strings".
- **func** (*function*) – The prediction function. Any state associated with the function will be captured via closure capture and pickled with Cloudpickle.
- **pyspark_model** (*pyspark.mllib.* or pyspark.ml.pipeline.PipelineModel object*) – The PySpark model to save.
- **sc** (*SparkContext,*) – The current SparkContext. This is needed to save the PySpark model.
- **default_output** (*str, optional*) – The default output for the application. The default output will be returned whenever an application is unable to receive a response from a model within the specified query latency SLO (service level objective). The reason the default output was returned is always provided as part of the prediction response object. Defaults to "None".
- **version** (*str, optional*) – The version to assign this model. Versions must be unique on a per-model basis, but may be re-used across different models.
- **slo_micros** (*int, optional*) – The query latency objective for the application in microseconds. This is the processing latency between Clipper receiving a request and sending a response. It does not account for network latencies before a request is received or after a response is sent. If Clipper cannot process a query within the latency objective, the default output is returned. Therefore, it is recommended that the SLO not be set aggressively low unless absolutely necessary. 100000 (100ms) is a good starting value, but the optimal latency objective will vary depending on the application.
- **labels** (*list(str), optional*) – A list of strings annotating the model. These are ignored by Clipper and used purely for user annotations.

- **registry** (*str, optional*) – The Docker container registry to push the freshly built model to. Note that if you are running Clipper on Kubernetes, this registry must be accesible to the Kubernetes cluster in order to fetch the container from the registry.
- **base_image** (*str, optional*) – The base Docker image to build the new model image from. This image should contain all code necessary to run a Clipper model container RPC client.
- **num_replicas** (*int, optional*) – The number of replicas of the model to create. The number of replicas for a model can be changed at any time with `clipper.ClipperConnection.set_num_replicas()`.
- **batch_size** (*int, optional*) – The user-defined query batch size for the model. Replicas of the model will attempt to process at most *batch_size* queries simultaneously. They may process smaller batches if *batch_size* queries are not immediately available. If the default value of -1 is used, Clipper will adaptively calculate the batch size for individual replicas of this model.
- **pkgs_to_install** (*list (of strings), optional*) – A list of the names of packages to install, using pip, in the container. The names must be strings.

3.3 PyTorch Models

Similar to the PySpark deployer, the PyTorch deployer provides a small extension to the Python closure deployer to allow you to deploy Python functions that include PyTorch models.

For PyTorch, Clipper will serialize the model using `torch.save` and it will be loaded using `torch.load`. It is expected the model has a forward method and can be called using `model(input)` to predict output.

`clipper_admin.deployers.pytorch.`**`deploy_pytorch_model`**(*clipper_conn*, *name*, *version*, *input_type*, *func*, *pytorch_model*, *base_image='default'*, *labels=None*, *registry=None*, *num_replicas=1*, *batch_size=-1*, *pkgs_to_install=None*)

Deploy a Python function with a PyTorch model.

Parameters

- **clipper_conn** (*clipper_admin.ClipperConnection()*) – A `ClipperConnection` object connected to a running Clipper cluster.
- **name** (*str*) – The name to be assigned to both the registered application and deployed model.
- **version** (*str*) – The version to assign this model. Versions must be unique on a per-model basis, but may be re-used across different models.
- **input_type** (*str*) – The input_type to be associated with the registered app and deployed model. One of “integers”, “floats”, “doubles”, “bytes”, or “strings”.
- **func** (*function*) – The prediction function. Any state associated with the function will be captured via closure capture and pickled with Cloudpickle.
- **pytorch_model** (*pytorch model object*) – The Pytorch model to save.

- **base_image** (*str, optional*) – The base Docker image to build the new model image from. This image should contain all code necessary to run a Clipper model container RPC client.
- **labels** (*list (str), optional*) – A list of strings annotating the model. These are ignored by Clipper and used purely for user annotations.
- **registry** (*str, optional*) – The Docker container registry to push the freshly built model to. Note that if you are running Clipper on Kubernetes, this registry must be accesible to the Kubernetes cluster in order to fetch the container from the registry.
- **num_replicas** (*int, optional*) – The number of replicas of the model to create. The number of replicas for a model can be changed at any time with `clipper.ClipperConnection.set_num_replicas()`.
- **batch_size** (*int, optional*) – The user-defined query batch size for the model. Replicas of the model will attempt to process at most *batch_size* queries simultaneously. They may process smaller batches if *batch_size* queries are not immediately available. If the default value of -1 is used, Clipper will adaptively calculate the batch size for individual replicas of this model.
- **pkgs_to_install** (*list (of strings), optional*) – A list of the names of packages to install, using pip, in the container. The names must be strings.

Example

Define a pytorch nn module and save the model:

```
from clipper_admin import ClipperConnection, DockerContainerManager
from clipper_admin.deployers.pytorch import deploy_pytorch_model
from torch import nn

clipper_conn = ClipperConnection(DockerContainerManager())

# Connect to an already-running Clipper cluster
clipper_conn.connect()
model = nn.Linear(1, 1)

# Define a shift function to normalize prediction inputs
def predict(model, inputs):
    pred = model(shift(inputs))
    pred = pred.data.numpy()
    return [str(x) for x in pred]

deploy_pytorch_model(
    clipper_conn,
    name="example",
    version=1,
    input_type="doubles",
    func=predict,
    pytorch_model=model)
```

`clipper_admin.deployers.pytorch.`**`create_endpoint`**(*clipper_conn*, *name*, *input_type*, *func*, *pytorch_model*, *default_output='None'*, *version=1*, *slo_micros=3000000*, *labels=None*, *registry=None*, *base_image='default'*, *num_replicas=1*, *batch_size=-1*, *pkgs_to_install=None*)

Registers an app and deploys the provided predict function with PyTorch model as a Clipper model.

Parameters

- **`clipper_conn`** (*`clipper_admin.ClipperConnection()`*) – A `ClipperConnection` object connected to a running Clipper cluster.
- **`name`** (*`str`*) – The name to be assigned to both the registered application and deployed model.
- **`input_type`** (*`str`*) – The input_type to be associated with the registered app and deployed model. One of "integers", "floats", "doubles", "bytes", or "strings".
- **`func`** (*`function`*) – The prediction function. Any state associated with the function will be captured via closure capture and pickled with Cloudpickle.
- **`pytorch_model`** (*`pytorch model object`*) – The PyTorch model to save.
- **`default_output`** (*`str, optional`*) – The default output for the application. The default output will be returned whenever an application is unable to receive a response from a model within the specified query latency SLO (service level objective). The reason the default output was returned is always provided as part of the prediction response object. Defaults to "None".
- **`version`** (*`str, optional`*) – The version to assign this model. Versions must be unique on a per-model basis, but may be re-used across different models.
- **`slo_micros`** (*`int, optional`*) – The query latency objective for the application in microseconds. This is the processing latency between Clipper receiving a request and sending a response. It does not account for network latencies before a request is received or after a response is sent. If Clipper cannot process a query within the latency objective, the default output is returned. Therefore, it is recommended that the SLO not be set aggressively low unless absolutely necessary. 100000 (100ms) is a good starting value, but the optimal latency objective will vary depending on the application.
- **`labels`** (*`list(str), optional`*) – A list of strings annotating the model. These are ignored by Clipper and used purely for user annotations.
- **`registry`** (*`str, optional`*) – The Docker container registry to push the freshly built model to. Note that if you are running Clipper on Kubernetes, this registry must be accesible to the Kubernetes cluster in order to fetch the container from the registry.
- **`base_image`** (*`str, optional`*) – The base Docker image to build the new model image from. This image should contain all code necessary to run a Clipper model container RPC client.
- **`num_replicas`** (*`int, optional`*) – The number of replicas of the model to create. The number of replicas for a model can be changed at any time with `clipper.ClipperConnection.set_num_replicas()`.
- **`batch_size`** (*`int, optional`*) – The user-defined query batch size for the model. Replicas of the model will attempt to process at most *batch_size* queries simultaneously. They may process smaller batches if *batch_size* queries are not immediately available. If

the default value of -1 is used, Clipper will adaptively calculate the batch size for individual replicas of this model.

- **pkgs_to_install** (*list (of strings), optional*) – A list of the names of packages to install, using pip, in the container. The names must be strings.

3.4 Tensorflow Models

Similar to the PySpark deployer, the TensorFlow deployer provides a small extension to the Python closure deployer to allow you to deploy Python functions that include TensorFlow models.

For Tensorflow, Clipper will save the Tensorflow Session.

`clipper_admin.deployers.tensorflow.`**`deploy_tensorflow_model`**(*clipper_conn, name, version, input_type, func, tf_sess_or_saved_model_path, base_image='default', labels=None, registry=None, num_replicas=1, batch_size=-1, pkgs_to_install=None*)

Deploy a Python prediction function with a Tensorflow session or saved Tensorflow model.

Parameters

- **clipper_conn** (*clipper_admin.ClipperConnection()*) – A `ClipperConnection` object connected to a running Clipper cluster.
- **name** (*str*) – The name to be assigned to both the registered application and deployed model.
- **version** (*str*) – The version to assign this model. Versions must be unique on a per-model basis, but may be re-used across different models.
- **input_type** (*str*) – The input_type to be associated with the registered app and deployed model. One of “integers”, “floats”, “doubles”, “bytes”, or “strings”.
- **func** (*function*) – The prediction function. Any state associated with the function will be captured via closure capture and pickled with Cloudpickle.
- **tf_sess** (*tensorflow.python.client.session.Session*) – The tensor flow session to save or path to an existing saved model.
- **base_image** (*str, optional*) – The base Docker image to build the new model image from. This image should contain all code necessary to run a Clipper model container RPC client.
- **labels** (*list(str), optional*) – A list of strings annotating the model. These are ignored by Clipper and used purely for user annotations.
- **registry** (*str, optional*) – The Docker container registry to push the freshly built model to. Note that if you are running Clipper on Kubernetes, this registry must be accesible to the Kubernetes cluster in order to fetch the container from the registry.
- **num_replicas** (*int, optional*) – The number of replicas of the model to create. The number of replicas for a model can be changed at any time with `clipper.ClipperConnection.set_num_replicas()`.

- **batch_size** (*int, optional*) – The user-defined query batch size for the model. Replicas of the model will attempt to process at most *batch_size* queries simultaneously. They may process smaller batches if *batch_size* queries are not immediately available. If the default value of -1 is used, Clipper will adaptively calculate the batch size for individual replicas of this model.
- **pkgs_to_install** (*list (of strings), optional*) – A list of the names of packages to install, using pip, in the container. The names must be strings.

Example

Save and deploy a tensorflow session:

```
from clipper_admin import ClipperConnection, DockerContainerManager
from clipper_admin.deployers.tensorflow import deploy_tensorflow_model

clipper_conn = ClipperConnection(DockerContainerManager())

# Connect to an already-running Clipper cluster
clipper_conn.connect()

def predict(sess, inputs):
    preds = sess.run('predict_class:0', feed_dict={'pixels:0': inputs})
    return [str(p) for p in preds]

deploy_tensorflow_model(
    clipper_conn,
    model_name,
    version,
    input_type,
    predict_fn,
    sess)
```

`clipper_admin.deployers.tensorflow.`**`create_endpoint`**(*clipper_conn, name, input_type, func, tf_sess_or_saved_model_path, default_output='None', version=1, slo_micros=3000000, labels=None, registry=None, base_image='default', num_replicas=1, batch_size=-1, pkgs_to_install=None*)

Registers an app and deploys the provided predict function with TensorFlow model as a Clipper model.

Parameters

- **clipper_conn** (*clipper_admin.ClipperConnection()*) – A `ClipperConnection` object connected to a running Clipper cluster.
- **name** (*str*) – The name to be assigned to both the registered application and deployed model.
- **input_type** (*str*) – The input_type to be associated with the registered app and deployed model. One of “integers”, “floats”, “doubles”, “bytes”, or “strings”.
- **func** (*function*) – The prediction function. Any state associated with the function will be captured via closure capture and pickled with Cloudpickle.

- **tf_sess** (`tensorflow.python.client.session.Session`) – The Tensorflow Session to save or path to an existing saved model.
- **default_output** (`str, optional`) – The default output for the application. The default output will be returned whenever an application is unable to receive a response from a model within the specified query latency SLO (service level objective). The reason the default output was returned is always provided as part of the prediction response object. Defaults to "None".
- **version** (`str, optional`) – The version to assign this model. Versions must be unique on a per-model basis, but may be re-used across different models.
- **slo_micros** (`int, optional`) – The query latency objective for the application in microseconds. This is the processing latency between Clipper receiving a request and sending a response. It does not account for network latencies before a request is received or after a response is sent. If Clipper cannot process a query within the latency objective, the default output is returned. Therefore, it is recommended that the SLO not be set aggressively low unless absolutely necessary. 100000 (100ms) is a good starting value, but the optimal latency objective will vary depending on the application.
- **labels** (`list(str), optional`) – A list of strings annotating the model. These are ignored by Clipper and used purely for user annotations.
- **registry** (`str, optional`) – The Docker container registry to push the freshly built model to. Note that if you are running Clipper on Kubernetes, this registry must be accesible to the Kubernetes cluster in order to fetch the container from the registry.
- **base_image** (`str, optional`) – The base Docker image to build the new model image from. This image should contain all code necessary to run a Clipper model container RPC client.
- **num_replicas** (`int, optional`) – The number of replicas of the model to create. The number of replicas for a model can be changed at any time with `clipper.ClipperConnection.set_num_replicas()`.
- **batch_size** (`int, optional`) – The user-defined query batch size for the model. Replicas of the model will attempt to process at most *batch_size* queries simultaneously. They may process smaller batches if *batch_size* queries are not immediately available. If the default value of -1 is used, Clipper will adaptively calculate the batch size for individual replicas of this model.
- **pkgs_to_install** (`list (of strings), optional`) – A list of the names of packages to install, using pip, in the container. The names must be strings.

3.5 MXNet Models

Similar to PySpark deployer, the MXNet deployer provides a small extension to the Python closure deployer to allow you to deploy Python functions that include MXNet models.

For MXNet, Clipper will serialize the model using `mxnet_model.save_checkpoint(..., epoch=0)`.

`clipper_admin.deployers.mxnet.`**`deploy_mxnet_model`**(*clipper_conn*, *name*, *version*, *input_type*, *func*, *mxnet_model*, *mxnet_data_shapes*, *base_image='default'*, *labels=None*, *registry=None*, *num_replicas=1*, *batch_size=-1*, *pkgs_to_install=None*)

Deploy a Python function with a MXNet model.

Parameters

- **clipper_conn** (*clipper_admin.ClipperConnection()*) – A `ClipperConnection` object connected to a running Clipper cluster.
- **name** (*str*) – The name to be assigned to both the registered application and deployed model.
- **version** (*str*) – The version to assign this model. Versions must be unique on a per-model basis, but may be re-used across different models.
- **input_type** (*str*) – The input_type to be associated with the registered app and deployed model. One of "integers", "floats", "doubles", "bytes", or "strings".
- **func** (*function*) – The prediction function. Any state associated with the function will be captured via closure capture and pickled with Cloudpickle.
- **mxnet_model** (*mxnet model object*) – The MXNet model to save.
- **mxnet_data_shapes** (*list of DataDesc objects*) – List of DataDesc objects representing the name, shape, type and layout information of data used for model prediction. Required because loading serialized MXNet models involves binding, which requires the shape of the data used to train the model. https://mxnet.incubator.apache.org/api/python/module.html#mxnet.module.BaseModule.bind
- **base_image** (*str, optional*) – The base Docker image to build the new model image from. This image should contain all code necessary to run a Clipper model container RPC client.
- **labels** (*list(str), optional*) – A list of strings annotating the model. These are ignored by Clipper and used purely for user annotations.
- **registry** (*str, optional*) – The Docker container registry to push the freshly built model to. Note that if you are running Clipper on Kubernetes, this registry must be accesible to the Kubernetes cluster in order to fetch the container from the registry.
- **num_replicas** (*int, optional*) – The number of replicas of the model to create. The number of replicas for a model can be changed at any time with `clipper.ClipperConnection.set_num_replicas()`.
- **batch_size** (*int, optional*) – The user-defined query batch size for the model. Replicas of the model will attempt to process at most *batch_size* queries simultaneously. They may process smaller batches if *batch_size* queries are not immediately available. If the default value of -1 is used, Clipper will adaptively calculate the batch size for individual replicas of this model.
- **pkgs_to_install** (*list (of strings), optional*) – A list of the names of packages to install, using pip, in the container. The names must be strings.

Note: Regarding *mxnet_data_shapes* parameter: Clipper may provide the model with variable size input batches. Because MXNet can't handle variable size input batches, we recommend setting batch size for input data to 1, or dynamically reshaping the model with every prediction based on the current input batch size. More information regarding a DataDesc object can be found here: https://mxnet.incubator.apache.org/versions/0.11.0/api/python/io.html#mxnet.io.DataDesc

Example

Create a MXNet model and then deploy it:

```
from clipper_admin import ClipperConnection, DockerContainerManager
from clipper_admin.deployers.mxnet import deploy_mxnet_model
import mxnet as mx

clipper_conn = ClipperConnection(DockerContainerManager())

# Connect to an already-running Clipper cluster
clipper_conn.connect()

# Create a MXNet model
# Configure a two layer neuralnetwork
data = mx.symbol.Variable('data')
fc1 = mx.symbol.FullyConnected(data, name='fc1', num_hidden=128)
act1 = mx.symbol.Activation(fc1, name='relu1', act_type='relu')
fc2 = mx.symbol.FullyConnected(act1, name='fc2', num_hidden=10)
softmax = mx.symbol.SoftmaxOutput(fc2, name='softmax')

# Load some training data
data_iter = mx.io.CSVIter(
    data_csv="/path/to/train_data.csv", data_shape=(785, ), batch_size=1)

# Initialize the module and fit it
mxnet_model = mx.mod.Module(softmax)
mxnet_model.fit(data_iter, num_epoch=1)

data_shape = data_iter.provide_data

deploy_mxnet_model(
    clipper_conn,
    name="example",
    version = 1,
    input_type="doubles",
    func=predict,
    mxnet_model=model,
    mxnet_data_shapes=data_shape)
```

`clipper_admin.deployers.mxnet.`**`create_endpoint`**(*clipper_conn*, *name*, *input_type*, *func*, *mxnet_model*, *mxnet_data_shapes*, *default_output='None'*, *version=1*, *slo_micros=3000000*, *labels=None*, *registry=None*, *base_image='default'*, *num_replicas=1*, *batch_size=-1*, *pkgs_to_install=None*)

Registers an app and deploys the provided predict function with MXNet model as a Clipper model.

Parameters

- **`clipper_conn`** (*`clipper_admin.ClipperConnection()`*) – A `ClipperConnection` object connected to a running Clipper cluster.
- **`name`** (*`str`*) – The name to be assigned to both the registered application and deployed model.
- **`input_type`** (*`str`*) – The input_type to be associated with the registered app and deployed model. One of “integers”, “floats”, “doubles”, “bytes”, or “strings”.

- **func** (*function*) – The prediction function. Any state associated with the function will be captured via closure capture and pickled with Cloudpickle.
- **mxnet_model** (*mxnet model object*) – The MXNet model to save. the shape of the data used to train the model.
- **mxnet_data_shapes** (*list of DataDesc objects*) – List of DataDesc objects representing the name, shape, type and layout information of data used for model prediction. Required because loading serialized MXNet models involves binding, which requires https://mxnet.incubator.apache.org/api/python/module.html#mxnet.module.BaseModule.bind
- **default_output** (*str, optional*) – The default output for the application. The default output will be returned whenever an application is unable to receive a response from a model within the specified query latency SLO (service level objective). The reason the default output was returned is always provided as part of the prediction response object. Defaults to "None".
- **version** (*str, optional*) – The version to assign this model. Versions must be unique on a per-model basis, but may be re-used across different models.
- **slo_micros** (*int, optional*) – The query latency objective for the application in microseconds. This is the processing latency between Clipper receiving a request and sending a response. It does not account for network latencies before a request is received or after a response is sent. If Clipper cannot process a query within the latency objective, the default output is returned. Therefore, it is recommended that the SLO not be set aggressively low unless absolutely necessary. 100000 (100ms) is a good starting value, but the optimal latency objective will vary depending on the application.
- **labels** (*list(str), optional*) – A list of strings annotating the model. These are ignored by Clipper and used purely for user annotations.
- **registry** (*str, optional*) – The Docker container registry to push the freshly built model to. Note that if you are running Clipper on Kubernetes, this registry must be accesible to the Kubernetes cluster in order to fetch the container from the registry.
- **base_image** (*str, optional*) – The base Docker image to build the new model image from. This image should contain all code necessary to run a Clipper model container RPC client.
- **num_replicas** (*int, optional*) – The number of replicas of the model to create. The number of replicas for a model can be changed at any time with `clipper.ClipperConnection.set_num_replicas()`.
- **batch_size** (*int, optional*) – The user-defined query batch size for the model. Replicas of the model will attempt to process at most *batch_size* queries simultaneously. They may process smaller batches if *batch_size* queries are not immediately available. If the default value of -1 is used, Clipper will adaptively calculate the batch size for individual replicas of this model.
- **pkgs_to_install** (*list (of strings), optional*) – A list of the names of packages to install, using pip, in the container. The names must be strings.

Note:

Regarding *mxnet_data_shapes* parameter: Clipper may provide the model with variable size input batches. Because MXNet can't handle variable size input batches, we recommend setting batch size for input data to 1, or dynamically reshaping the model with every prediction based on the current input batch size. More information regarding a DataDesc object can be found here: https://mxnet.incubator.apache.org/versions/0.11.0/api/python/io.html#mxnet.io.DataDesc

3.6 Create Your Own Container

If none of the provided model deployers will meet your needs, you will need to create your own model container.

Tutorial on building your own model container

CHAPTER 4

Exceptions

exception clipper_admin.**ClipperException**(*msg*, **args*)

A generic exception indicating that Clipper encountered a problem.

exception clipper_admin.**UnconnectedException**(**args*)

A `ClipperConnection` instance must be connected to a Clipper cluster to issue this command.

CHAPTER 5

RClipper

5.1 Rclipper

Rclipper is a package for building serveable Clipper models from R functions. Given an API-compatible R function, Rclipper's **build_model** function builds a Docker image for a Clipper model. This model can then be deployed to Clipper via the Python clipper_admin package.

5.2 Dependencies

Rclipper depends on the Python clipper_admin package for building and deploying models. In order to use this admin package, Docker for Python must also be installed.

5.3 Importing Rclipper

It is very important that Rclipper be imported before a prediction function or its dependencies are defined. Rclipper makes use of the histry package to statically analyze dependency definitions. In order to locate these definition expressions during function serialization, histry must be imported before the expressions are executed.

5.4 Writing an API-compatible R prediction function

An API-compatible prediction function must accept a type-homogeneous **list** of inputs of one of the following types:

- Raw Vector
- Integer Vector
- Numeric Vector
- String (length-1 character vector)

- Data Frame
- Matrix
- Array
- List

Additionally, **given a list of inputs** of length *N*, a prediction function **must return a list of outputs** of length *N*. All elements of the output list must be of the same type.

Note: If a prediction function returns a list of string (length-1 character vector) objects, each output will be returned as-is, without any additional serialization. Otherwise, all non-string outputs will be string-serialized via the jsonlite package, and their serialized representations will be returned.

5.5 Building a model

Once you've written an API-compatible prediction function, you can build a Clipper model with it via the **build_model** function:

```
#' @param model_name character vector of length 1. The name to assign to the model␣
↪image.
#' @param model_version character vector of length 1. The version tag to assign to␣
↪the model image.
#' @param prediction_function function. This should accept a type-homogeneous list of
#' inputs and return a list of outputs of the same length. If the elements of the␣
↪output list
#' are not character vectors of length 1, they will be converted to a serialized
#' string representation via 'jsonlite'.
#' @param sample_input For a prediction function that accepts a list of inputs of␣
↪type X,
#' this should be a single input of type X. This is used to validate the compatability
#' of the function with Clipper and to determine the Clipper data type (bytes, ints,␣
↪strings, etc)
#' to associate with the model.
#' @param model_registry character vector of length 1. The name of the image registry
#' to which to upload the model image. If NULL, the image will not be uploaded to a␣
↪registry.

Rclipper::build_model(model_name, model_version, prediction_function, sample_input,␣
↪model_registry = NULL)
```

This will build a Docker image with the tag: *model_registry/model_name*:*model_version*. If no registry was specified, the image will have the tag: *model_name*:*model_version*. Additonally, this function will output a command that you can execute within an interactive Python environment to deploy the model with the clipper_admin package.

5.6 Deploying a model

Once you've built a model, use the provided command to deploy it with the clipper_admin package. For information about how to register the model with an application so that it can be queried, please consult the clipper_admin API documentation.

5.7 Querying a model

After you've built a model, deployed the model, and registered the model with an application, you can query it with input data of the correct type. The following table maps the input type of your model's prediction function to the Clipper input type associated with your deployed model:

R input type	Clipper Input Type	JSON Format	Example
Raw Vector	Bytes	Base64-encoded string	Y2xpcHBlciB0ZXh0
Integer Vector	Ints	Integer array	[1,2,3,4]
Numeric Vector	Doubles	Floating point array	[1.0,2.0,3.0.,4.0]
Character Vector	Strings	String	"input text"
Data Frame	Strings	String	jsonlite::toJSON(mtcars)
Matrix	Strings	String	jsonlite::toJSON(diag(3))
Array	Strings	String	jsonlite::toJSON(array(1:4))

List

Strings

String

jsonlite::toJSON(list(1:4))

5.8 Example

5.8.1 Import Rclipper

```
library(Rclipper)

## Loading required package: CodeDepends

## Loading required package: histry
```

5.8.2 Define an API-compatible prediction function

```
#' Given a list of vector inputs,
#' outputs a list containing the
#' length of each input vector as a string
pred_fn = function(inputs) {
return(lapply(inputs, function(input) {
return(as.character(length(input)))
}))
}

print(pred_fn(list(c(1,2), c(3))))

## [[1]]
## [1] "2"
##
## [[2]]
## [1] "1"
```

5.8.3 Build a model

```
# Specify that the prediction function expects integer vectors
# by supplying an integer vector as the sample input
Rclipper::build_model("test-model", "1", pred_fn, sample_input = as.integer(c(1,2,3)))

## [1] "Serialized list of dependent libraries: Rclipper: knitr: histry: CodeDepends:␣
↪stats: graphics: grDevices: utils: datasets: methods: base"
## [1] "Serialized model function!"
## [1] "Done!"
## To deploy this model, execute the following command from a connected␣
↪ClipperConnection object `conn`:
## conn.deploy_model("test-model", "1", "ints", "test-model:1", num_replicas=<num_
↪container_replicas>)
```

5.8.4 Deploy and link the model

This assumes that a Clipper cluster is running on *localhost* with a registered application that has the name *app1*. In a Python interactive environment:

```
from clipper_admin import DockerContainerManager, ClipperConnection
cm = DockerContainerManager()
conn = ClipperConnection(cm)
conn.connect()

# Deploy a single replica of the model
conn.deploy_model(name="test-model", version="1", input_type="ints", image="test-
↪model:1", replicas=1)

conn.link_model_to_app(app_name="app1", model_name="test-model")
```

5.8.5 Query the model

You can now query the model from any HTTP client. For example, directly from the command line with cURL:

```
$ curl -X POST --header "Content-Type:application/json" -d '{"input": [1,2,3,4]}' 127.
↪0.0.1:1337/app1/predict

$ {"query_id":2,"output":4,"default":false}
```

Index