Scaling Machine Learning

Rahul Dave, for cs109b

github

https://github.com/rahuldave/dasktut

Running Experiments

How do we ensure

(a) repeatability

(b) performance

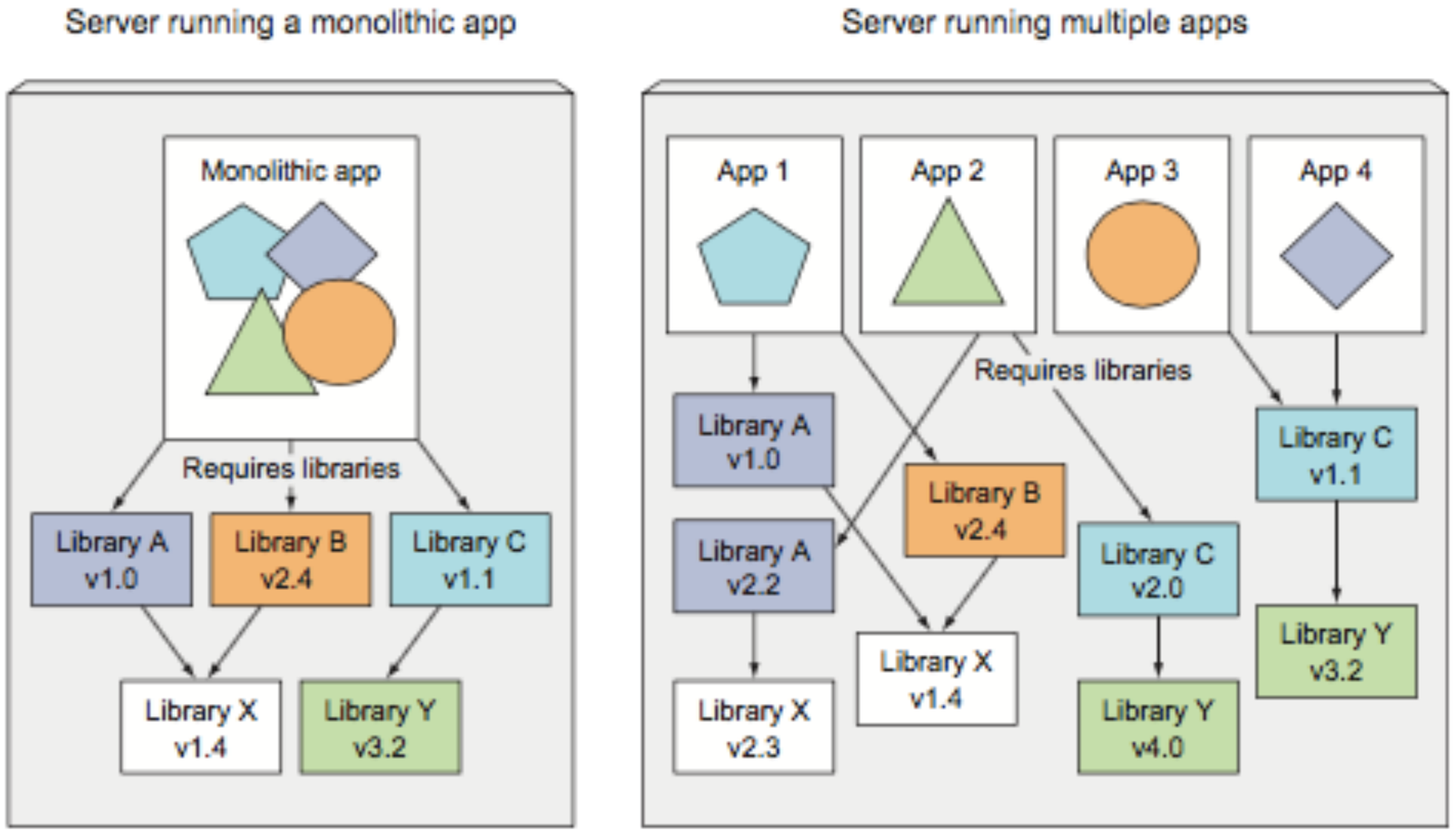
(c) descriptiveness

(d) dont lose our head?

What is scaling?

- Running experiments reproducibly, and keeping track
- Running in parallel, for speed and resilience
- Dealing with large data sets
- Grid or other Hyper-parameter optimization
- optimizing Gradient Descent

The multiple libraries problem



Conda

- create a conda environment for each new project
- put an `environment.yml` in each project folder
- at least have one for each new class, or class of projects
- envoronment for class of projects may grow organically, but capture its requirements from time-to-time.

see here

```yaml
# file name: environment.yml

# Give your project an informative name
name: project-name

# Specify the conda channels that you wish to grab packages from, in order of priority.
channels:
- defaults
- conda-forge

# Specify the packages that you would like to install inside your environment.
#Version numbers are allowed, and conda will automatically use its dependency
#solver to ensure that all packages work with one another.
dependencies:
- python=3.7
- conda
- scipy
- numpy
- pandas
- scikit-learn

# There are some packages which are not conda-installable. You can put the pip dependencies here instead.
- pip:
    - tqdm  # for example only, tqdm is actually available by conda.
```

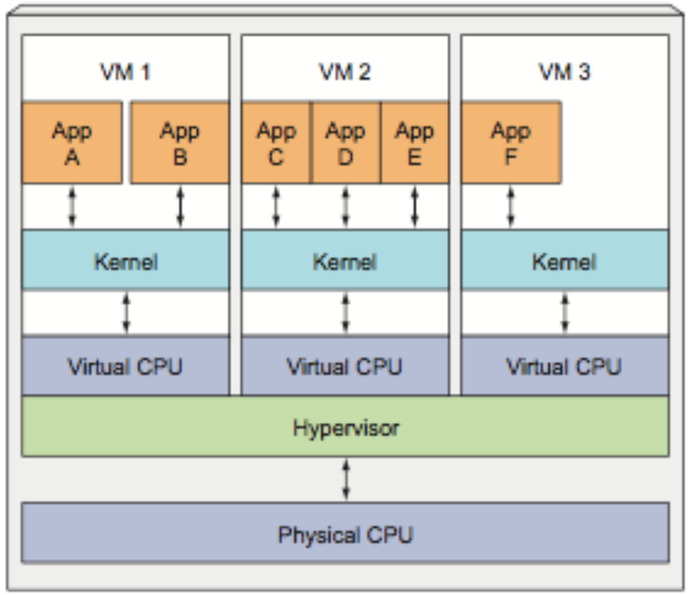
(from http://ericmjl.com/blog/2018/12/25/conda-hacks-for-data-science-efficiency/)

- `conda create --name environment-name [python=3.6]`
- `source[conda] activate environment-name` or project-name in the 1 environment per project paradigm
- `conda env create` in project folder
- `conda install <packagename>`
- or add the package to spec file, type `conda env update environment.yml` in appropriate folder
- `conda env export > environment.yml`

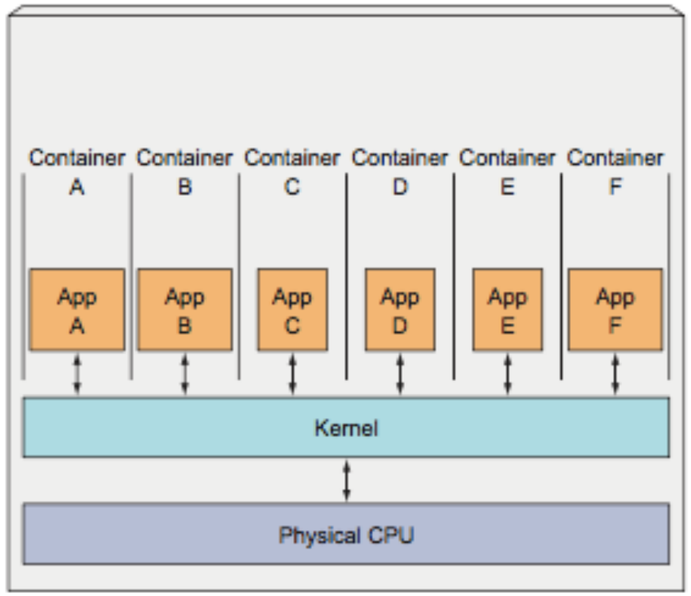
Docker

More than python libs

Apps running in multiple VMs



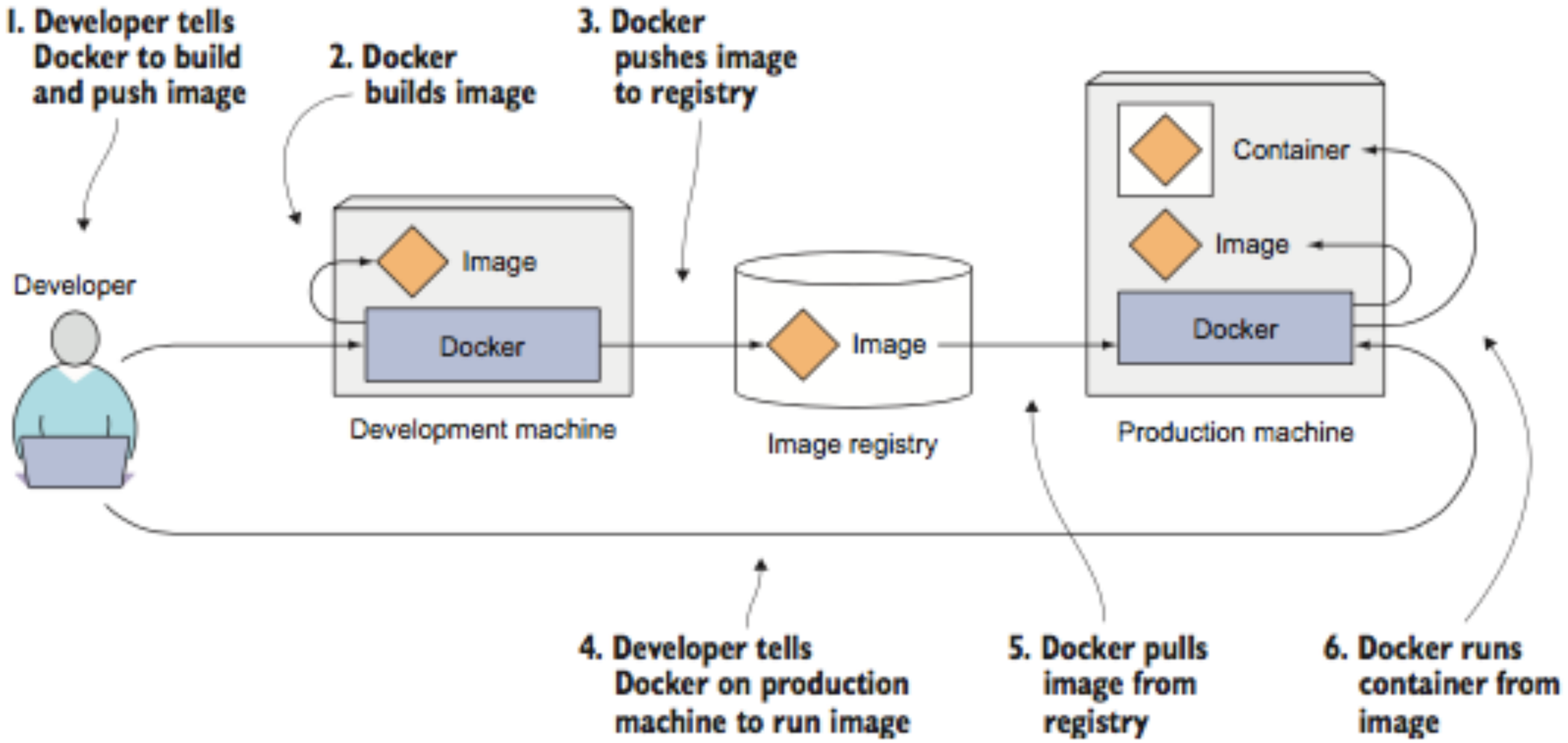
Apps running in isolated containers



Containers vs Virtual Machines

- VMs meed an OS level "hypervisor"
- are more general, but more resource hungry
- containers provide process isolation, process throttling
- but work at library and kernel level, and can access hardware more easily
- hardware access important for gpu access
- containers can run on VMS, this is how docker runs on mac

Docker Architecture



Docker images

- docker is linux only, but other OS's now have support
- allow for environment setting across languages and runtimes
- can be chained together to create outcomes
- base image is a linux (full) image, others are just layers on top

Example: base notebook -> minimal notebook -> scipy notebook -> tensorflow notebook

```
ARG
BASE_CONTAINER=ubuntu:bionic-20180526@sha256:c8c275751219dadad8fa56b3ac41ca6c
b22219ff117ca98fe82b42f24e1ba64e
FROM $BASE_CONTAINER
ARG NB_USER="jovyan"
...
USER root
RUN apt-get update && apt-get -yq dist-upgrade \
 && apt-get install -yq --no-install-recommends \
    wget \
    ...
RUN echo "en_US.UTF-8 UTF-8" > /etc/locale.gen && \
    locale-gen
ENV CONDA_DIR=/opt/conda \
    NB_USER=$NB_USER \
    ...
ADD fix-permissions /usr/local/bin/fix-permissions
RUN groupadd wheel -g 11 && \
    useradd -m -s /bin/bash -N -u $NB_UID $NB_USER && \
    ...
USER $NB_UID
...
ENV MINICONDA_VERSION 4.5.11
RUN cd /tmp && \
    wget --quiet https://repo.continuum.io/miniconda/Miniconda3-$
    {MINICONDA_VERSION}-Linux-x86_64.sh && \
    echo "e1045ee415162f944b6aebfe560b8fee *Miniconda3-${MINICONDA_VERSION}
    -Linux-x86_64.sh" | md5sum -c - && \
    /bin/bash Miniconda3-${MINICONDA_VERSION}-Linux-x86_64.sh -f -b -p
    $CONDA_DIR && \
    ...
RUN conda install --quiet --yes 'tini=0.18.0' && \
    ...
RUN conda install --quiet --yes \
'notebook=5.7.2' \
'jupyterhub=0.9.4' \
'jupyterlab=0.35.4' && ...
USER root
EXPOSE 8888
ENTRYPOINT ["tini", "-g", "--"]
CMD ["start-notebook.sh"]
COPY start.sh /usr/local/bin/
...
USER $NB_UID
```

```
ARG BASE_CONTAINER=jupyter/base-notebook
FROM $BASE_CONTAINER

LABEL maintainer="Jupyter Project <jupyter@googlegroups.com>"

USER root

# Install all OS dependencies for fully functional notebook server
RUN apt-get update && apt-get install -yq --no-install-recommends \
    build-essential \
    emacs \
    git \
    inkscape \
    jed \
    libsm6 \
    libxext-dev \
    libxrender1 \
    lmodern \
    netcat \
    pandoc \
    python-dev \
    texlive-fonts-extra \
    texlive-fonts-recommended \
    texlive-generic-recommended \
    texlive-latex-base \
    texlive-latex-extra \
    texlive-xetex \
    unzip \
    nano \
    && rm -rf /var/lib/apt/lists/*

# Switch back to jovyan to avoid accidental container runs as root
USER $NB_UID
```

```
ARG BASE_CONTAINER=jupyter/minimal-notebook
FROM $BASE_CONTAINER
...
# ffmpeg for matplotlib anim
RUN apt-get update && \
    apt-get install -y --no-install-recommends ffmpeg && \
    rm -rf /var/lib/apt/lists/*
RUN conda install --quiet --yes \
    'conda-forge::blas=*=openblas' \
    'ipywidgets=7.4*' \
    'pandas=0.23*' \
    'numexpr=2.6*' \
    'matplotlib=2.2*' \
    'scipy=1.1*' \
    'seaborn=0.9*' \
    'scikit-learn=0.20*' \
    'scikit-image=0.14*' \
    'sympy=1.1*' \
    'cython=0.28*' \
    'patsy=0.5*' \
    'statsmodels=0.9*' \
    'cloudpickle=0.5*' \
    'dill=0.2*' \
    'numba=0.38*' \
    'bokeh=0.13*' \
    'sqlalchemy=1.2*' \
    'hdf5=1.10*' \
    'h5py=2.7*' \
    'vincent=0.4.*' \
    'beautifulsoup4=4.6.*' \
    'protobuf=3.*' \
    'xlrd'  && \
    conda remove --quiet --yes --force qt pyqt && \
    ...

# Install facets which does not have a pip or conda package at the moment
RUN cd /tmp && \
    git clone https://github.com/PAIR-code/facets.git && \
    cd facets && \
    jupyter nbextension install facets-dist/ --sys-prefix && \
    cd && ...
```

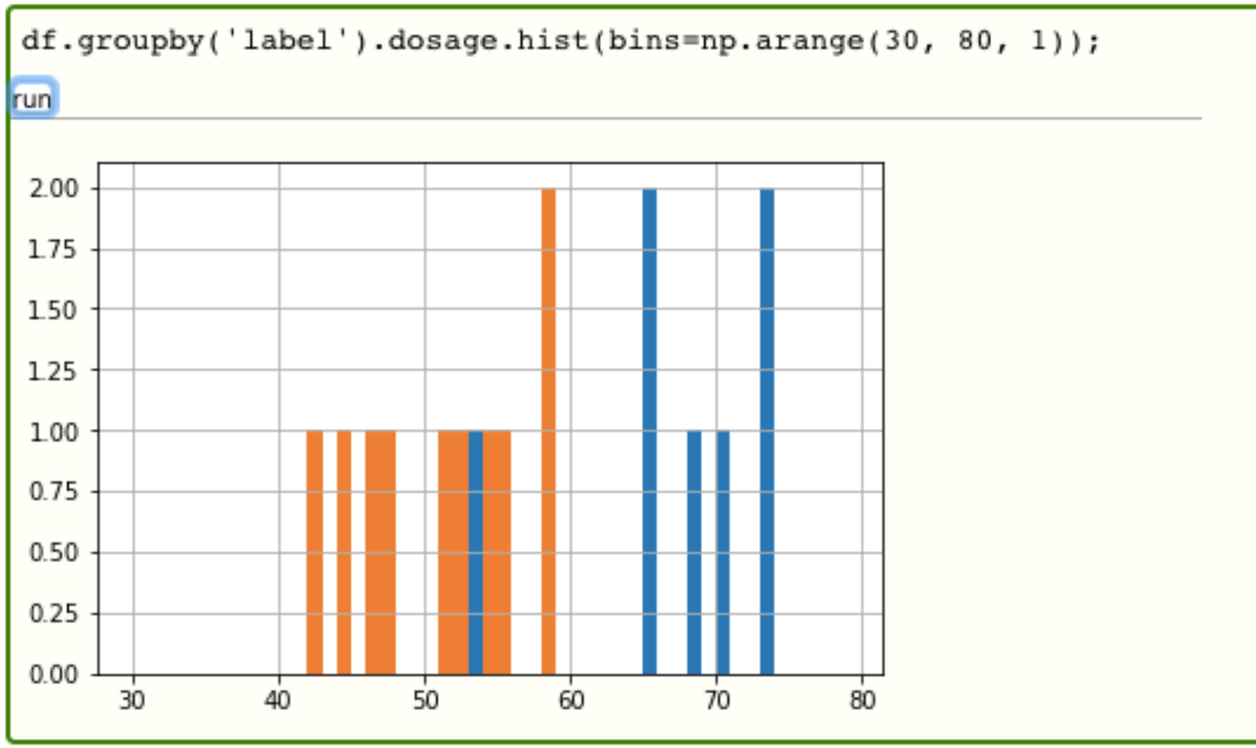
```
# Copyright (c) Jupyter Development Team.
# Distributed under the terms of the Modified BSD License.
ARG BASE_CONTAINER=jupyter/scipy-notebook
FROM $BASE_CONTAINER

LABEL maintainer="Jupyter Project <jupyter@googlegroups.com>"

# Install Tensorflow
RUN conda install --quiet --yes \
    'tensorflow=1.12*' \
    'keras=2.2*' && \
    conda clean -tipsy && \
    fix-permissions $CONDA_DIR && \
    fix-permissions /home/$NB_USER
```

repo2docker and binder

- building docker images is not dead simple
- the Jupyter folks created repo2docker for this.
- provide a github repo, and repo2docker makes a docker image and uploads it to the docker image repository for you
- binder builds on this to provide a service where you provide a github repo, and it gives you a working jupyterhub where you can "publish" your project/demo/etc

usage example: AM207 and thebe-lab

- see https://github.com/am207/shadowbinder , a repository with an environment file only
- this repo is used to build a jupyterlab with some requirements where you can work.
- see here for example
- uses thebelab

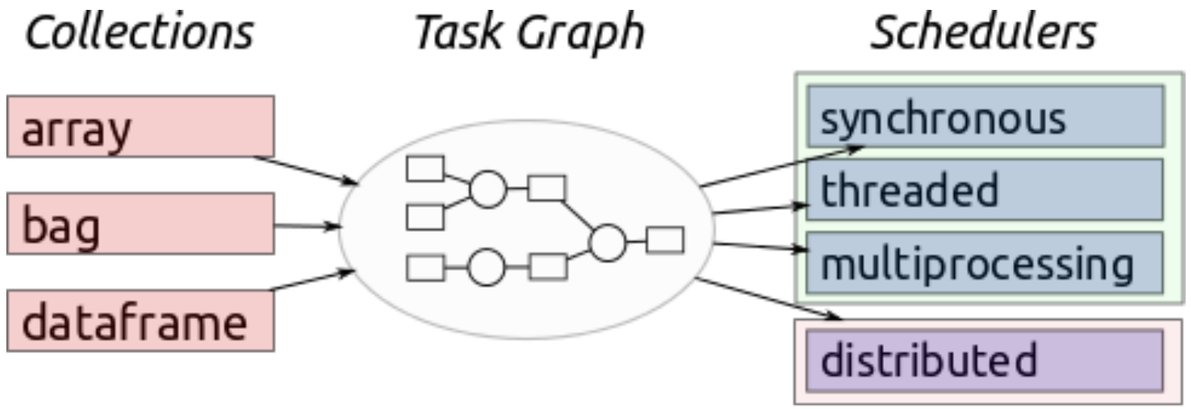


```
<script type="text/x-thebe-config">
      thebeConfig = {
        binderOptions: {
          repo: "AM207/shadowbinder",
        },
        kernelOptions: {
          name: "python3",
        },
        requestKernel: true
      }
    </script>
    <script src="/css/thebe_status_field.js" type="text/javascript"></script>
    <link rel="stylesheet" type="text/css" href="/css/thebe_status_field.css"/>
    <script>
      $(function() {
          var cellSelector = "pre.highlight code";

          if ($(cellSelector).length > 0) {
             $(' <span>|</span><span class="thebe_status_field"></span>')
                .appendTo('article p:first');
             thebe_place_activate_button();
          }
      });
    </script>
    <script>window.onload = function() { $("div.language-python pre.highlight code").attr("data-executable", "true")};</script>
```

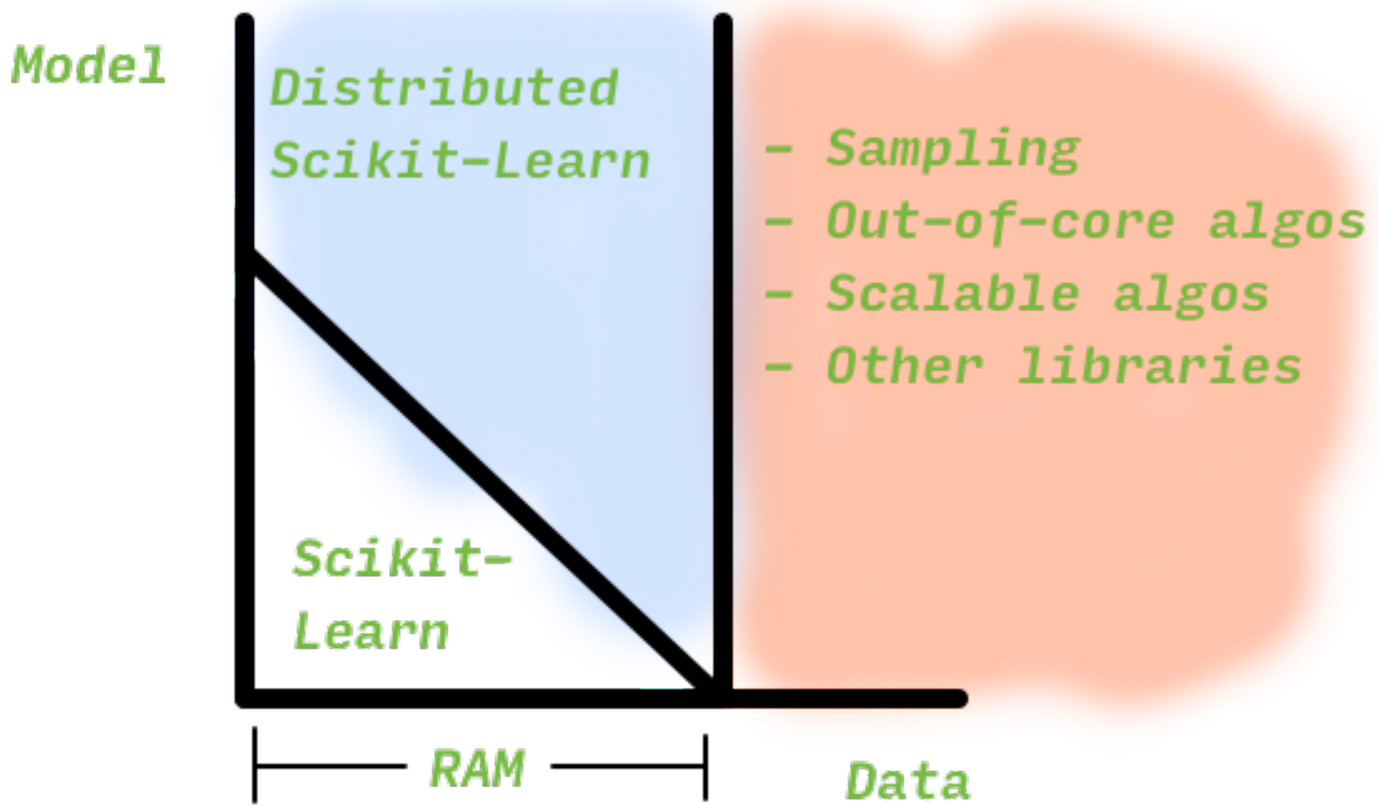
Dask

Running in parallel

Dask



- library for parallel computing in Python.
- 2 parts. Dynamic task scheduling optimized for computation like `Airflow`. “Big Data” collections like parallel (numpy) arrays, (pandas) dataframes, and lists
- scales up (1000 core cluster) and doqn (laptop)
- designed with interactive computing in mind, with web based diagnostics



(from https://github.com/TomAugspurger/dask-tutorial-pycon-2018)

Parallel Hyperparameter Optimization

Why is this bad?

```
from sklearn.model_selection import GridSearchCV

vectorizer = TfidfVectorizer()
vectorizer.fit(text_train)

X_train = vectorizer.transform(text_train)
X_test = vectorizer.transform(text_test)

clf = LogisticRegression()
grid = GridSearchCV(clf, param_grid={'C': [.1, 1, 10, 100]}, cv=5)
grid.fit(X_train, y_train)
```

Grid search on pipelines

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
from sklearn.linear_model import SGDClassifier
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
from sklearn.datasets import fetch_20newsgroups

categories = [
    'alt.atheism',
    'talk.religion.misc',
]
data = fetch_20newsgroups(subset='train', categories=categories)
pipeline = Pipeline([('vect', CountVectorizer()),
                     ('tfidf', TfidfTransformer()),
                     ('clf', SGDClassifier())])
grid = {'vect__ngram_range': [(1, 1)],
        'tfidf__norm': ['l1', 'l2'],
        'clf__alpha': [1e-3, 1e-4, 1e-5]}

if __name__=='__main__':
    grid_search = GridSearchCV(pipeline, grid, cv=5, n_jobs=-1)
    grid_search.fit(data.data, data.target)
    print("Best score: %0.3f" % grid_search.best_score_)
    print("Best parameters set:", grid_search.best_estimator_.get_params())
```

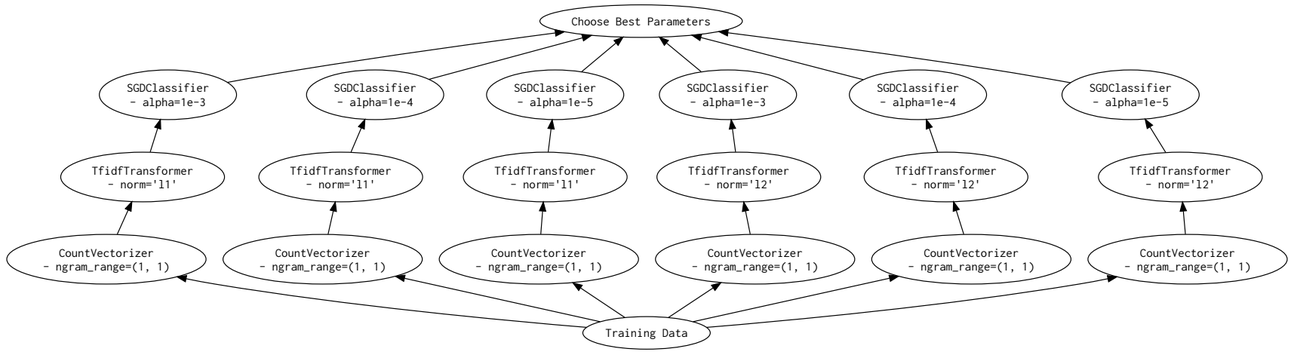
From sklearn.pipeline.Pipeline.html :

Sequentially apply a list of transforms and a final estimator. Intermediate steps of the pipeline must be 'transforms', that is, they must implement `fit` *and* `transform` *methods. The final estimator only needs to implement* `fit`*. The transformers in the pipeline can be cached using memory argument.*

The purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters.

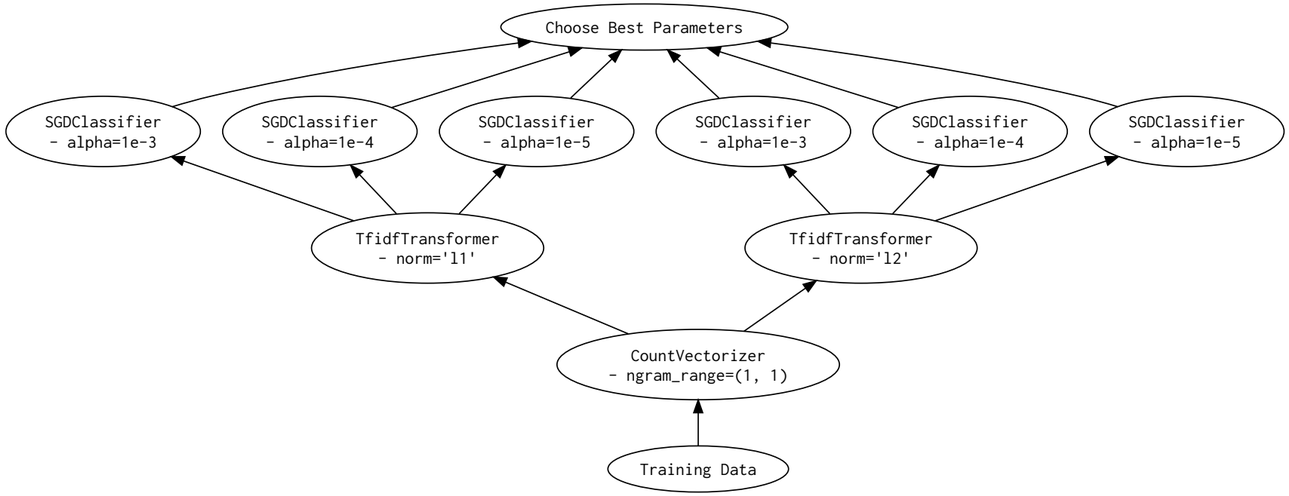
sklearn pipelines: the bad

```
scores = []
for ngram_range in parameters['vect__ngram_range']:
        for norm in parameters['tfidf__norm']:
                for alpha in parameters['clf__alpha']:
                        vect = CountVectorizer(ngram_range=ngram_range)
                        X2 = vect.fit_transform(X, y)
                        tfidf = TfidfTransformer(norm=norm)
                        X3 = tfidf.fit_transform(X2, y)
                        clf = SGDClassifier(alpha=alpha)
                        clf.fit(X3, y)
                        scores.append(clf.score(X3, y))
best = choose_best_parameters(scores, parameters)
```



dask pipelines: the good

```
scores = []
for ngram_range in parameters['vect__ngram_range']:
        vect = CountVectorizer(ngram_range=ngram_range)
        X2 = vect.fit_transform(X, y)
        for norm in parameters['tfidf__norm']:
                tfidf = TfidfTransformer(norm=norm)
                X3 = tfidf.fit_transform(X2, y)
                for alpha in parameters['clf__alpha']:
                        clf = SGDClassifier(alpha=alpha)
                        clf.fit(X3, y)
                        scores.append(clf.score(X3, y))
best = choose_best_parameters(scores, parameters)
```



Now, lets parallelize

- for data that fits into memory, we simply copy the memory to each node and run the algorithm there
- if you have created a re-sizable cluster of parallel machines, dask can even dynamically send parameter combinations to more and more machines
- see PANGEO and Grisel for this

Hyperopt

```
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from dask_ml.model_selection import GridSearchCV
from dask.distributed import Client
from sklearn.externals import joblib

def simple_nn(hidden_neurons):
  model = Sequential()
  model.add(Dense(hidden_neurons, activation='relu', input_dim=30))
  model.add(Dense(1, activation='sigmoid'))
  model.compile(loss='binary_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
  return model

param_grid = {'hidden_neurons': [100, 200, 300]}
if __name__=='__main__':
    client = Client()
    cv = GridSearchCV(KerasClassifier(build_fn=simple_nn, epochs=30), param_grid)
    X, y = load_breast_cancer(return_X_y=True)
    X_train, X_test, y_train, y_test = train_test_split(X, y)
    with joblib.parallel_backend("dask", scatter=[X_train, y_train]):
        cv.fit(X_train, y_train)
    print(f'Best Accuracy for {cv.best_score_:.4} using {cv.best_params_}')
```

Large Data Sets

- important for pre-processing,
- also important for prediction on large data (test) sets
- dask provides scalable algoritms which can be run over clusters and are drop-in replacements for the `sklearn` equivalents
- Dask separates computation description (task graphs) from execution (schedulers).
- Write code once, and run it locally or scale it out across a cluster.

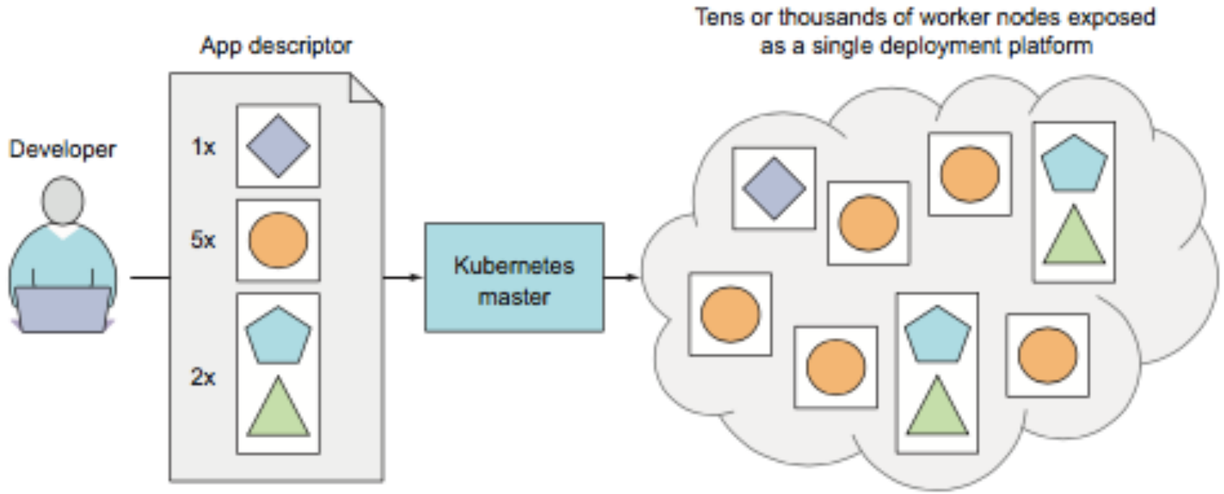
```
# Setup a local cluster.
import dask.array as da
import dask.delayed
from sklearn.datasets import make_blobs
import numpy as np
from dask_ml.cluster import KMeans


n_centers = 12
n_features = 20
X_small, y_small = make_blobs(n_samples=1000, centers=n_centers, n_features=n_features, random_state=0)
centers = np.zeros((n_centers, n_features))
for i in range(n_centers):
    centers[i] = X_small[y_small == i].mean(0)
print(centers)

n_samples_per_block = 20000 # 0
n_blocks = 500
delayeds = [dask.delayed(make_blobs)(n_samples=n_samples_per_block,
                                     centers=centers,
                                     n_features=n_features,
                                     random_state=i)[0] for i in range(n_blocks)]
arrays = [da.from_delayed(obj, shape=(n_samples_per_block, n_features), dtype=X_small.dtype) for obj in delayeds]
X = da.concatenate(arrays)
print(X.nbytes / 1e9)
X = X.persist() #actually run the stuff

clf = KMeans(init_max_iter=3, oversampling_factor=10)
clf.fit(X)
print(clf.labels_[:10].compute()) #actually run the stuff
```

```python
# run using local distributed scheduler
import dask.array as da
import dask.delayed
from sklearn.datasets import make_blobs
import numpy as np
from dask_ml.cluster import KMeans

n_centers = 12
n_features = 20
X_small, y_small = make_blobs(n_samples=1000, centers=n_centers, n_features=n_features, random_state=0)
centers = np.zeros((n_centers, n_features))
for i in range(n_centers):
    centers[i] = X_small[y_small == i].mean(0)
print(centers)

from dask.distributed import Client

# Setup a local cluster.
# By default this sets up 1 worker per core
if __name__=='__main__':
    client = Client()
    print(client.cluster)
    n_samples_per_block = 20000 # 0
    n_blocks = 500
    delayeds = [dask.delayed(make_blobs)(n_samples=n_samples_per_block,
                                         centers=centers,
                                         n_features=n_features,
                                         random_state=i)[0] for i in range(n_blocks)]
    arrays = [da.from_delayed(obj, shape=(n_samples_per_block, n_features), dtype=X_small.dtype) for obj in delayeds]
    X = da.concatenate(arrays)
    print(X.nbytes / 1e9)
    X = X.persist() #actually run the stuff

    clf = KMeans(init_max_iter=3, oversampling_factor=10)
    clf.fit(X)
    print(clf.labels_[:10].compute()) #actually run the stuff
```

- we've seen the use of `dask.distributed`
- but we have run it locally. ideally we want to run on a cloud-provisioned cluster
- and we'd like this cluster to be self-repairing
- and then we'd like our code to respond to failures.
- and expand onto more machines if we need them

We need a cluster manager.

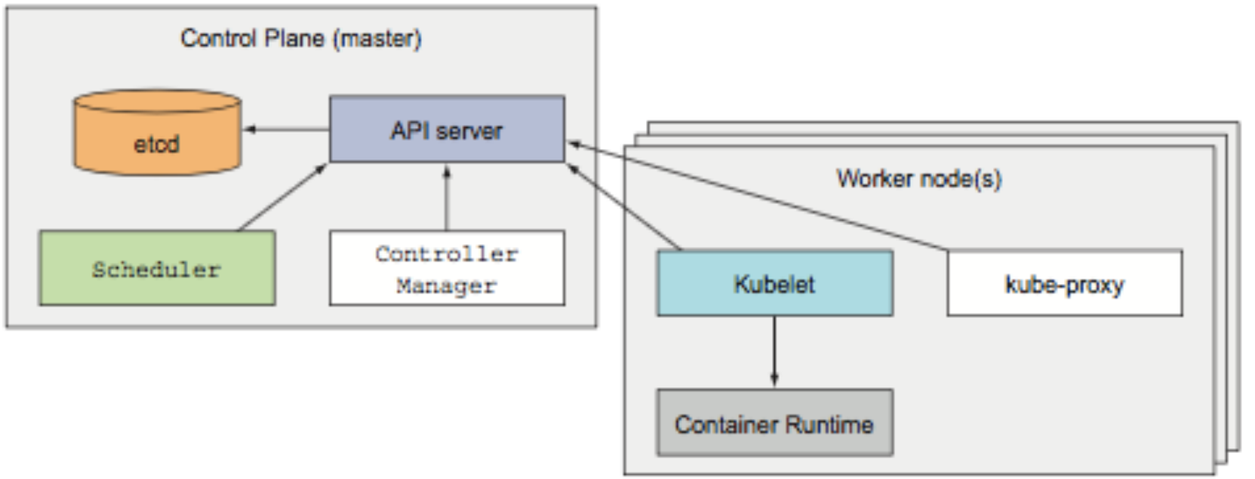
Enter Kubernetes

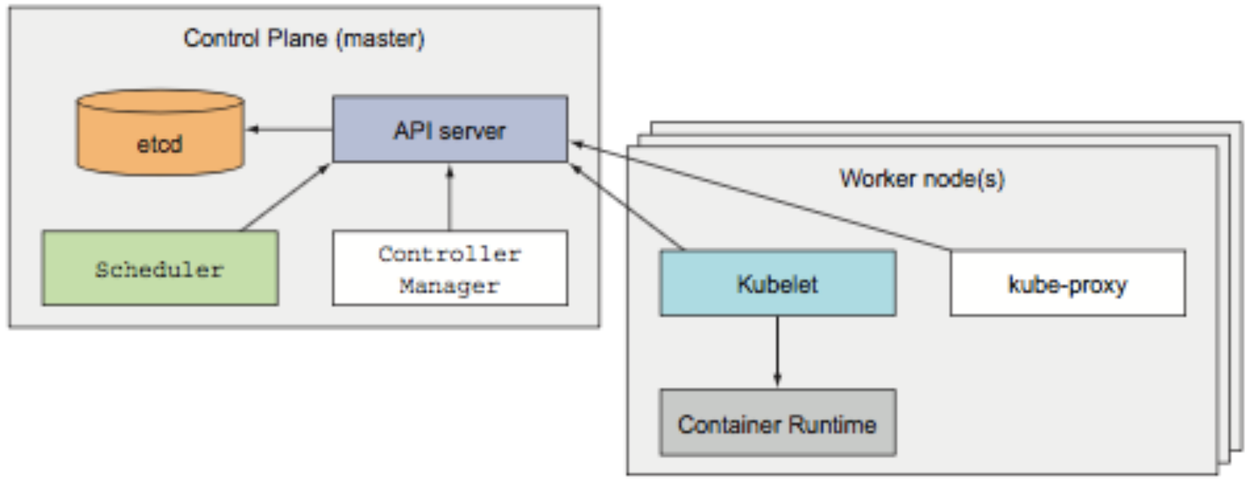


- OS for the cluster
- provides service discovery, scaling, load-balancing, self-healing, leader election
- think of applications as stateless, and movable from one machine to another to enable better resource utilization
- thus does not cover mutable databases which must remain outside the cluster
- there is a controlling master node, and worker nodes

master node:

- API server, communicated with my control-plane components and you (using `kubectl`)
- Scheduler, assigns a worker node to each application
- Controller Manager, performs cluster-level functions, such as replicating components, keeping track of worker nodes, handling node failures
- etcd, a reliable distributed data store that persistently stores the cluster configuration.

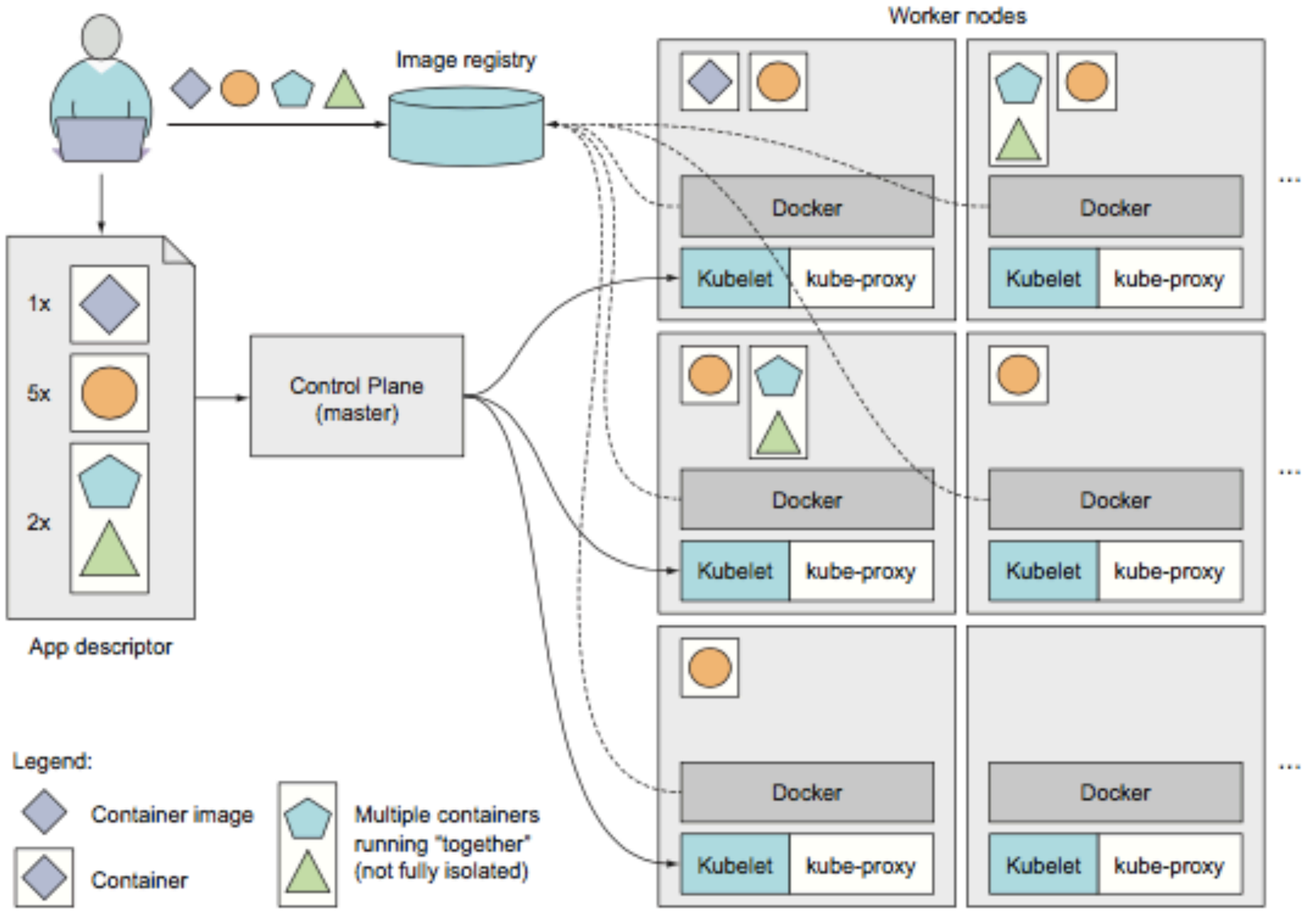




worker node:

- Docker, to run your containers
- you package your apps components into 1 or more docker images, and push them to a registry
- Kubelet, which talks to the API server and manages containers on its node
- kube-proxy, which load-balances network traffic between application components

- To run an application in Kubernetes, you post a description of your app to the Kubernetes API server.
- people have created canned "descriptions" for multi-component software, which you can reuse. These use a "package manager" called `helm`, and its what is used to install dask and jupyterhub on a cluster
- description includes info on component images, their relationship, which ones need co-location, and how many replicas
- internal or external network services are also described. A lookup service is provided, and a given service is exposed at a particular ip address. kube-proxy makes sure connec- tions to the service are load balanced
- master continuously makes sure that the deployed state of the application matches description


Worker nodes
Image registry
Docker
Kubelet
kube-proxy
1x
5x
2x
App descriptor
Control Plane
(master)
Legend:
Container image
Container
Multiple containers
running "together"
(not fully isolated)

Example: website with 3 replicas

Image:

```
FROM nginx:stable-alpine
COPY  site/ /usr/share/nginx/html/
EXPOSE 80
```

Namespace:

```
apiVersion: v1
kind: Namespace
metadata:
  name: website
```

deployment.yaml ->

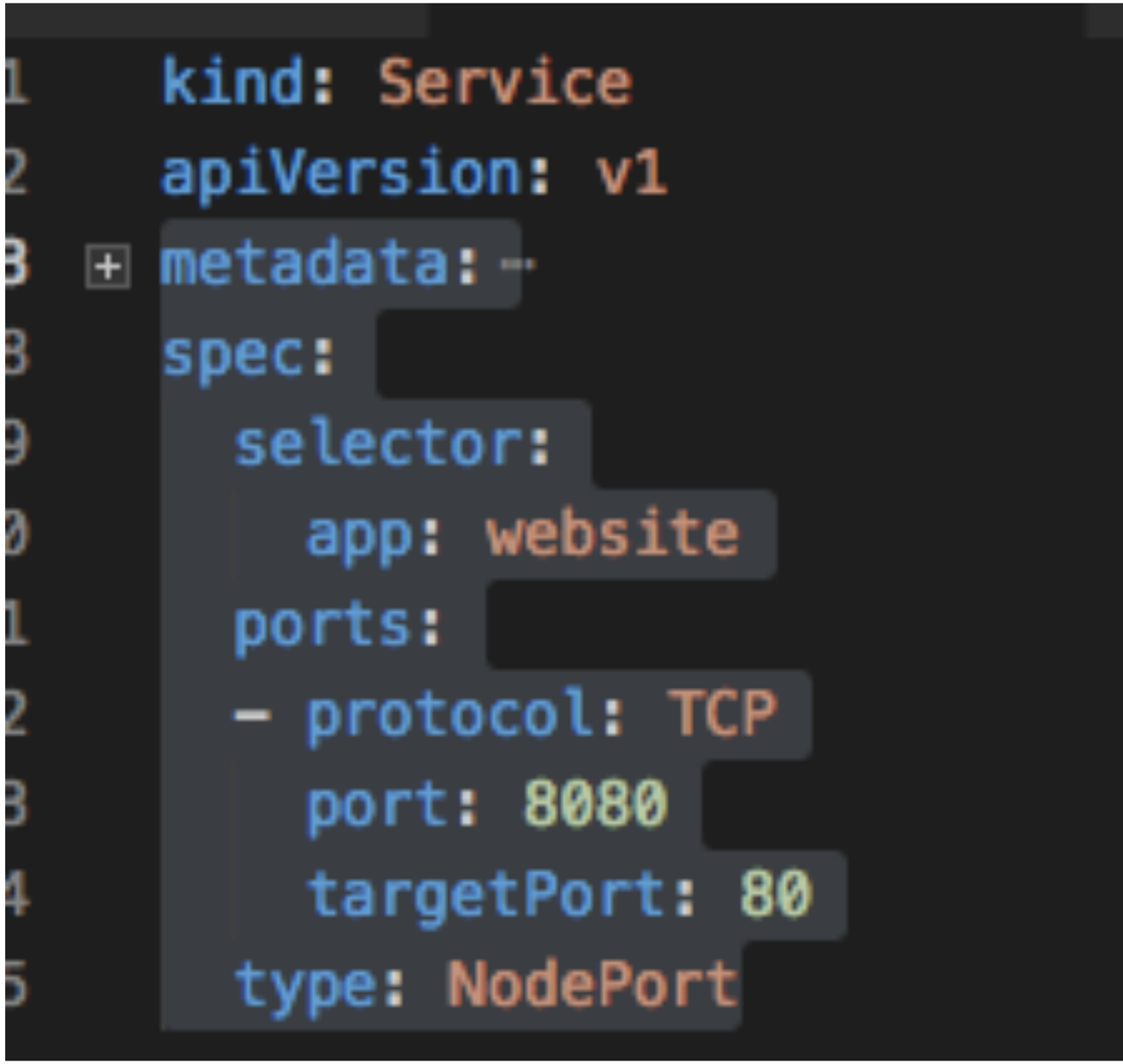
```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: website
  namespace: website
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: website
    spec:
      containers:
      - name: website
        image: gcr.io/univaiweb/website:latest
        imagePullPolicy: Always
        ports:
        - containerPort: 80
```

Networking

right: internal networking

below: external ingress

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: basic-ingress
  namespace: website
  annotations:
    kubernetes.io/ingress.global-static-ip-name: "website-static-ip"
spec:
  backend:
    serviceName: website
    servicePort: 8080
```



Dask cloud deployment

Kubernetes is recommended

This can be done on your local machine using Minikube or on any of the 3 major cloud prociders, Azure, GCP, or AWS.

1. set up a Kubernetes cluster
2. Next you will set up Helm, which is a package maner for Kubernetes which works simply by filling templated yaml files with variables also stored in another yaml file `values.yaml`.
3. Finally you will install dask. First `helm repo update` and then `helm install stable/dask`.

See https://docs.dask.org/en/latest/setup/kubernetes-helm.html for all the details.

Deep Learning on the cloud

- tensorflow can be put on the cloud using `tf.distributed` of `kubeflow`
- parallelism can be trivially used at prediction time--you just need to distribute your weights
- as in our keras example you might have grid optimization
- but it would seem SGD is sequential
- can train it asynchronously using parameter servers. use `tf.distributed`.
- for training as well as serving