Deephyper on Theta breakout, October - 1, 2019, ALCF SDL Workshop

Prasanna - `pbalapra@anl.gov`

Romain - `romainegele@gmail.com`

Misha - `msalim@anl.gov`

Romit - `rmaulik@anl.gov`

Note: Most of the material in this document can be revisited at

https://deephyper.readthedocs.io/en/latest/

^^ Note that we will be looking at the **latest** version of the docs ^^

Some tips

Downloading data from theta to your local machine:

```
scp -r username@theta.alcf.anl.gov:/path_to_file /path_on_local_machine
```

Uploading data to theta from your local machine:

```
scp -r /path_on_local_machine username@theta.alcf.anl.gov:/path_to_file
```

Download this markdown document

`tinyurl.com/sdl-oct2019-slides`

Installation

From login node

1. Load miniconda: `module load miniconda-3.6/conda-4.5.12`
2. Load balsam: `module load balsam/0.3`
3. Make a new directory for running this breakout: `mkdir dh-handson`
4. Make a virtual environment: `python -m venv --system-site-packages deephyper-dev-env`
5. Activate this environment: `source deephyper-dev-env/bin/activate`
6. **Temporary**: `pip install --upgrade setuptools`
7. Install an ipykernel for postprocessing - `pip install ipykernel`
8. Install an ipython kernel: `python -m ipykernel install --user --name deephyper-dev-env --display-name "Python deephyper-dev-env"` - will be needed for analytics later.

9. Clone deephyper from Github: `git clone https://github.com/deephyper/deephyper.git deephyper_repo/`
10. Go to root of cloned directory: `cd deephyper_repo/`
11. Checkout the develop branch: `git checkout develop`
12. Install the package: `pip install -e .['analytics']`

Deephyper should now be good to go. Run the following command to make sure all is well: `deephyper --help`

Setting up to run Hyperparameter search (HPS) and neural architecture search (NAS)

In order to run a Deephyper project on Theta, one must initialize a balsam database:

1. Go to the root of the hands-on directory: `cd dh-handson/`
2. Use balsam command line tool to initialize a database: `balsam init deephyper_breakout_db`
3. Activate the balsam server with: `source balsamactivate deephyper_breakout_db`

Deephyper has some convenient command line tools to organize your HPS and/or NAS project. We can set up a *project* directory within which all our HPS and NAS runs can be contained. To do this

1. Create a new project directory: `deephyper start-project theta_breakout_runs`. This will create a folder called `theta_breakout_runs` which has been *pip installed* like a package.

Make a HPS folder

1. Go to this project directory: `cd theta_breakout_runs/`
2. **`deephyper new-problem hps hps_run`**

Lets inspect our project directory now -

```
theta_breakout_runs/
      __init__.py
      setup.py
      theta_breakout_runs.egg-info
      hps_run/
          __init__.py
          load_data.py
          model_run.py
          problem.py
```

HPS

Case-directory

Let us define and execute an HPS on Theta. The following steps will lead you guide you through this process

1. Go to the HPS run folder: `cd hps_run/`. There are three files in this folder each having a specific function (we encourage you to retain this structure). The `load_data.py` script defines an interface for reading in your training and validation data. For this problem we are

generating synthetic data using the `polynome_2()` function. You can play around with the expression of this function.

```
import os
import numpy as np

np.random.seed(2018)

def load_data(dim=10, a=-50, b=50, prop=0.80, size=10000):
    """Generate a random distribution of data for polynome_2 function: SUM(X**2)
where "**" is an element wise operator in the continuous range [a, b].

    Args:
        dim (int): size of input vector for the polynome_2 function.
        a (int): minimum bound for all X dimensions.
        b (int): maximum bound for all X dimensions.
        prop (float): a value between [0., 1.] indicating how to split data
between training set and testing set. `prop` corresponds to the ratio of data in
training set. `1.-prop` corresponds to the amount of data in testing set.
        size (int): amount of data to generate. It is equal to
`len(training_data)+len(testing_data).

    Returns:
        tuple(tuple(ndarray, ndarray), tuple(ndarray, ndarray)): of Numpy
arrays: `(train_X, train_y), (valid_X, valid_y)`.
    """

    def polynome_2(x):
        return -sum([x_i**2 for x_i in x])

    d = b - a
    x = np.array([a + np.random.random(dim) * d for i in range(size)])
    y = np.array([[polynome_2(v)] for v in x])

    sep_index = int(prop * size)
    train_X = x[:sep_index]
    train_y = y[:sep_index]

    test_X = x[sep_index:]
    test_y = y[sep_index:]

    print(f'train_X shape: {np.shape(train_X)}')
    print(f'train_y shape: {np.shape(train_y)}')
    print(f'test_X shape: {np.shape(test_X)}')
    print(f'test_y shape: {np.shape(test_y)}')
    return (train_X, train_y), (test_X, test_y)

if __name__ == '__main__':
    load_data()
```

The entire data generation (or loading from disk) operation is wrapped in a `load_data()` function which returns a dataset split into training and validation.

2. Lets look at the `model_run.py` script contains the actual function `run()` for training your framework (i.e., the model for which you want to find the best hyperparameters). Note how this script imports `load_data` to interface with your training and validation data. You will

note that the `run` function requires a dictionary to specify the hyperparameters for one training. This is how deephyper will interface with your model. The `run()` functions returns the value of an objective function which deephyper will try to **MAXIMIZE**.

```
import numpy as np
import keras.backend as K
import keras
from keras.callbacks import EarlyStopping
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import RMSprop

import os
import sys
here = os.path.dirname(os.path.abspath(__file__))
sys.path.insert(0, here)
from load_data import load_data


def r2(y_pred, y_true):
    SS_res = K.sum(K.square(y_true-y_pred))
    SS_tot = K.sum(K.square(y_true - K.mean(y_true)))
    return (1 - SS_res/(SS_tot + K.epsilon()))


HISTORY = None


def run(point):
    global HISTORY
    (x_train, y_train), (x_test, y_test) = load_data()

    model = Sequential()
    model.add(Dense(
        point['units'],
        activation=point['activation'],
        input_shape=tuple(np.shape(x_train)[1:])))
    model.add(Dense(1))

    model.summary()

    model.compile(loss='mse', optimizer=RMSprop(lr=point['lr']), metrics=[r2])

    history = model.fit(x_train, y_train,
                        batch_size=64,
                        epochs=1000,
                        verbose=1,
                        callbacks=[EarlyStopping(
                            monitor='val_r2',
                            mode='max',
                            verbose=1,
                            patience=10
                        )],
                        validation_data=(x_test, y_test))

    HISTORY = history.history
```

```
    return history.history['val_r2'][-1]


if __name__ == '__main__':
    point = {
        'units': 10,
        'activation': 'relu',
        'lr': 0.01
    }
    objective = run(point)
    print('objective: ', objective)
    import matplotlib.pyplot as plt
    plt.plot(HISTORY['val_r2'])
    plt.xlabel('Epochs')
    plt.ylabel('Objective: $R^2$')
    plt.grid()
    plt.show()
```

3. The search space for the hyperparameters are specified in `problem.py`. One can use the `add_dim` option to add hyperparameter keys to the dictionary along with bounds. Hyperparameters can be real, integer or ordinal in nature. Starting points where you think the search algorithm can get a headstart can also be specified here.

```
from deephyper.benchmark import HpProblem

Problem = HpProblem()

Problem.add_dim('units', (1, 100))
Problem.add_dim('activation', [None, 'relu', 'sigmoid', 'tanh'])
Problem.add_dim('lr', (0.0001, 1.))

Problem.add_starting_point(
    units=10,
    activation=None,
    lr=0.01)

if __name__ == '__main__':
    print(Problem)
```

4. Ensure each script works appropriately by using running it from the command line (`python model_run.py`, `python load_data.py`, `python problem.py`) with your virtual environment activated.

Execution

Now that the case directory has been set up properly, we must execute a search using Deephyper and Balsam. We follow these steps in order:

1. Create an *app* in balsam for the executable: `balsam app --name AMBS --exec "$(which python) -m deephyper.search.hps.ambs"`. Check `balsam ls apps` for newly added app.
2. Create a *job* in balsam for running the search: `balsam job --name hps_run --workflow hps_run --app AMBS --args '--evaluator balsam --run hps_run.model_run.run --problem hps_run.problem.Problem --max-evals 10000 --acq-func LCB --learner RF'`. Search for the job created using `balsam ls jobs`.

3. Finally submit your job for execution using: `balsam submit-launch -q training -t 30 -n 4 -A SDL_Workshop --job-mode serial --wf-filter hps_run`

Postprocessing and Analytics

After the job starts running, one can find results being populated in the balsam database we created earlier. You can access these results in: `dh_handson/theta_breakout_db/data/hps_run/hps_run_**`. For HPS these constitute a `results.csv` file which shows how AMBS has sampled in hyperparameter space and corresponding values of the objective function. If your model was spitting out any metrics/data/plots while training these will be saved in `dh_handson/theta_breakout_db/hps_run/task**`.

A jupyter-notebook with some statistical assessments of the search can be created in the following manner:

1. Go to `dh-handson/theta_breakout_db/data/hps_run/hps_run_**/`
2. With your virutal environment and balsam database active run: `deephyper-analytics hps -p results.csv -n visualization` at this location. This will create a `visualization.ipnyb` file which can be run on ALCF jupyter.
3. Through `jupyter.alcf.anl.gov`, navigate to the location of this file and visualize the results of your search and find the best combination of hyperparameters. (Make sure you are using the right Python kernel from the kernel dropdown menu) - Note there may be issues here which we are looking into right now.

Cleanup

Lets say something went wrong in your job specification/execution and you would like to *resubmit* the problem in `hps_run`. Cleaning up presubmitted jobs and apps can be carried out with -

1. Removing the jobs: `balsam rm jobs --name hps_run`
2. Removing apps: `balsam rm apps --name AMBS`

NAS

Make a NAS folder

1. Go to this project directory: `cd theta_breakout_runs/`
2. `deephyper new-problem nas nas_run`

Lets inspect our project directory now -

```
theta_breakout_runs/
      __init__.py
      setup.py
      theta_breakout_runs.egg-info
      hps_run/
          __init__.py
          load_data.py
          model_run.py
          problem.py
      nas_run/
          __init__.py
          load_data.py
          problem.py
```

```
        search_space.py
```

Deephyper-NAS differs from HPS in that the model specification is no longer *black-box*. By now you must know that HPS requires the specification of a scalar metric to maximize and does not care for the model architecture beyond how it interacts with the hyperparameters. In NAS, a reinforcement learning agent (an LSTM) explores a model search space which can be interpreted as a directed acylical graph. The sequential nature of the graph can thus be exploited for more efficient neural architecture exploration.

Case-directory

Let us define and execute NAS on Theta. The following steps will lead you guide you through this process

1. The `load_data.py` script is similar in function to its counterpart in `hps_run/`.

```
import os
import numpy as np

np.random.seed(2018)

def load_data(dim=10, a=-50, b=50, prop=0.80, size=10000):
    """Generate a random distribution of data for polynome_2 function: SUM(X**2)
where "**" is an element wise operator in the continuous range [a, b].

    Args:
        dim (int): size of input vector for the polynome_2 function.
        a (int): minimum bound for all X dimensions.
        b (int): maximum bound for all X dimensions.
        prop (float): a value between [0., 1.] indicating how to split data
between training set and testing set. `prop` corresponds to the ratio of data in
training set. `1.-prop` corresponds to the amount of data in testing set.
        size (int): amount of data to generate. It is equal to
`len(training_data)+len(testing_data).

    Returns:
        tuple(tuple(ndarray, ndarray), tuple(ndarray, ndarray)): of Numpy
arrays: `(train_X, train_y), (valid_X, valid_y)`.
    """

    def polynome_2(x):
        return -sum([x_i**2 for x_i in x])

    d = b - a
    x = np.array([a + np.random.random(dim) * d for i in range(size)])
    y = np.array([[polynome_2(v)] for v in x])

    sep_index = int(prop * size)
    train_X = x[:sep_index]
    train_y = y[:sep_index]

    test_X = x[sep_index:]
    test_y = y[sep_index:]

    print(f'train_X shape: {np.shape(train_X)}')
```

```
    print(f'train_y shape: {np.shape(train_y)}')
    print(f'test_X shape: {np.shape(test_X)}')
    print(f'test_y shape: {np.shape(test_y)}')
    return (train_X, train_y), (test_X, test_y)

if __name__ == '__main__':
    load_data()
```

2. The `search_space.py` represents the primary effort of the user in NAS. This script must define a graph with a number of nodes from which multiple architectures can be sampled. Within this script, the function `create_search_space` contains instructions for the addition of nodes and (possible) skip connections to a search space. The choice of operations at each node is selected by the NAS agent in the form of a real-valued number (`index`) between 0-1 which maps to an operation through `int(index * len(ops))`. Skip connections can be incorporated using `AddByProjecting` which projects the skipco input to the right shape through a linear operation. Other options include `Concatenate` and `AddByPadding`.

```
import collections

import tensorflow as tf

from deephyper.search.nas.model.space import AutoKSearchSpace
from deephyper.search.nas.model.space.node import ConstantNode, VariableNode
from deephyper.search.nas.model.space.op.basic import Tensor
from deephyper.search.nas.model.space.op.connect import Connect
from deephyper.search.nas.model.space.op.merge import AddByProjecting
from deephyper.search.nas.model.space.op.op1d import Dense, Identity


def add_dense_to_(node):
    node.add_op(Identity()) # we do not want to create a layer in this case

    activations = [None, tf.nn.relu, tf.nn.tanh, tf.nn.sigmoid]
    for units in range(16, 97, 16):
        for activation in activations:
            node.add_op(Dense(units=units, activation=activation))


def create_search_space(input_shape=(10,),
                        output_shape=(7,),
                        num_layers=10,
                        *args, **kwargs):

    arch = AutoKSearchSpace(input_shape, output_shape, regression=True)
    source = prev_input = arch.input_nodes[0]

    # look over skip connections within a range of the 3 previous nodes
    anchor_points = collections.deque([source], maxlen=3)

    for _ in range(num_layers):
        vnode = VariableNode()
        add_dense_to_(vnode)

        arch.connect(prev_input, vnode)

        # * Cell output
```

```
        cell_output = vnode

        cmerge = ConstantNode()
        cmerge.set_op(AddByProjecting(arch, [cell_output], activation='relu'))

        for anchor in anchor_points:
            skipco = VariableNode()
            skipco.add_op(Tensor([]))
            skipco.add_op(Connect(arch, anchor))
            arch.connect(skipco, cmerge)

        # ! for next iter
        prev_input = cmerge
        anchor_points.append(prev_input)


    return arch


def test_create_search_space():
    """Generate a random neural network from the search_space definition.
    """
    from random import random
    from tensorflow.keras.utils import plot_model
    import tensorflow as tf

    search_space = create_search_space(num_layers=10)
    ops = [random() for _ in range(search_space.num_nodes)]

    print(f'This search_space needs {len(ops)} choices to generate a neural
network.')

    search_space.set_ops(ops)

    model = search_space.create_model()
    model.summary()

    plot_model(model, to_file='sampled_neural_network.png', show_shapes=True)
    print("The sampled_neural_network.png file has been generated.")


if __name__ == '__main__':
    test_create_search_space()
```

3. `problem.py` defines the NAS problem allowing for the search agent to interface with the data and the search space. Additionally preprocessing and neural architecture hyperparameters (such as batch size, learning rate etc.) can be provided here. The reward for reinforcement of the search is given by the `objective` option.

```
from deephyper.benchmark import NaProblem
from nas_problems.polynome2.load_data import load_data
from nas_problems.polynome2.search_space import create_search_space
from deephyper.search.nas.model.preprocessing import minmaxstdscaler

Problem = NaProblem(seed=2019)

Problem.load_data(load_data)
```

```
Problem.preprocessing(minmaxstdscaler)

Problem.search_space(create_search_space, num_layers=3)

Problem.hyperparameters(
    batch_size=32,
    learning_rate=0.01,
    optimizer='adam',
    num_epochs=20,
    callbacks=dict(
        EarlyStopping=dict(
            monitor='val_r2', # or 'val_acc' ?
            mode='max',
            verbose=0,
            patience=5
        )
    )
)

Problem.loss('mse') # or 'categorical_crossentropy' ?

Problem.metrics(['r2']) # or 'acc' ?

Problem.objective('val_r2__last') # or 'val_acc__last' ?


# Just to print your problem, to test its definition and imports in the current
python environment.
if __name__ == '__main__':
    print(Problem)
```

Execution

Now that the case directory has been set up properly, we must execute a search using Deephyper and Balsam. We follow these steps in order:

1. Create an *app* in balsam for the executable: `balsam app --name PPO --exec "$(which python) -m deephyper.search.nas.ppo"`. Check `balsam ls apps` for newly added app.
2. Create a *job* in balsam for running the search: `balsam job --name nas_run --workflow nas_run --app PPO --num-nodes 2 --args '--evaluator balsam --problem nas_run.problem.Problem'`. Search for the job created using `balsam ls jobs`.
3. Finally submit your job for execution using: `balsam submit-launch -q training -t 30 -n 12 -A SDL_Workshop --job-mode mpi --wf-filter nas_run`.

Postprocessing and Analytics

NAS requires a slightly different parsing strategy-

1. Execute `deephyper-analytics parse deephyper.log` with the `data/nas_run/nas_run_**/` directory. This will create a json file with the explored networks.
2. Transform from json using: `deephyper-analytics single -p *.json -n visualization`.
3. Use `jupyter.alcf.anl.gov` to access `visualization.ipynb` and run analytics. (Make sure you are using the right Python kernel from the kernel dropdown menu).

NOTE: In case of trouble with analytics on ACLF jupyter

Make a *shadow* virtual environment on your local machine and follow these steps to install and run deephyper-analytics from the command line (balsam will not be required). I am assuming you have some version of python 3 on your machine.

1. Make a new directory somewhere convenient and navigate inside it: `mkdir dh-handson` and `cd dh-handson`
2. Make a virtual environment within this directory: `python -m venv --system-site-packages deephyper-dev-env`
3. Activate this environment: `source deephyper-dev-env/bin/activate`
4. **Temporary**: `pip install --upgrade setuptools`
5. Install an ipykernel for postprocessing - `pip install ipykernel`
6. Install an ipython kernel: `python -m ipykernel install --name deephyper-dev-env --display-name "Python deephyper-dev-env"` - for analytics.
7. Clone deephyper from Github: `git clone https://github.com/deephyper/deephyper.git deephyper_repo/`
8. Go to root of cloned directory: `cd deephyper_repo/`
9. Checkout the develop branch: `git checkout develop`
10. Install the package: `pip install -e .['analytics']`

One can then `scp -r username@theta.alcf.anl.gov:/path_to_results_file /path_to_shadow_directory` and run a jupyter notebook on their local machine.