ALP

Release 0.3.0

Sep 27, 2017

Contents

ALP helps you experiment with a lot of machine learning models quickly. It provides you with a simple way of scheduling and recording experiments. This library has been developped to work well with Keras and Scikit-learn but can suit a lot of other frameworks.

CHAPTER 1

First steps with Alp

Why ALP?

We noticed that, when dealing with a Machine Learning problem, we sometime spend more time working on building a model, testing different architectures, comparing results than actually work on the ideas that will solve our problem. To help that process, we developed an Asynchronous Learning Platform (ALP) that uses the hardware (CPU+GPU) at a convenient capacity. That platform relies on independant services running on Docker containers. For this plateform to be easy to use, we built a convenient command line interface from wich you can easily launch, stop, remove, update and monitor a configuration.

The whole system runs in the background so that the final user does not directly interact with the databases or the broker and just runs code in an usual Jupyter Notebook or from an application. You can also launch monitoring containers and access different dashboards to supervise all of your experiments. Moreover, it is possible to easily retrieve one of the trained model along with it's parameters at test time.

What kind of models can you use?

So far, the whole Keras neural network library is supported, as well as several models from the scikit-learn library.

What do I need to run ALP? What is inside ALP?

You need to use a machine running Linux to use ALP[1]. ALP relies on Docker, RabbitMQ, Celery, MongoDB and nvidia-docker. It also supports interfacing with Fuel thus depends on Theano. It's implemented in Python. However since all services runs into Docker containers, your OS only needs Docker (and nvidia-docker if you want to use a NVIDIA GPU).

All of this concepts and dependencies are explained later in the Setup and Userguide sections.

[1] unfortunately at the time of the development, running MongoDB in a Windows Docker was not a possibility, but we will check out that soon.

How could ALP help me?

We believe it might be useful for several applications such as:

- **hyperparameters tuning**: for instance if you want to test several architectures on your neural network model, ALP can help you in dealing with the tedious task of logging all the architectures, parameters and results. They are all automatically stored in the databases and you just have to select the best model given the validation(s) you specified.
- **fitting several models on several data streams**: you have data streams coming from a source and you want to fit a lot of online models, it is easy with ALP. With the support of Fuel generators, you could transform your data on the fly. The he learning is then done using the resources of the host and the parameters of the models are stored. You could even code an API that returns prediction to your data service.
- **post analysis**: extract and explore the parameters of models given their score on several data blocks. Sometimes it could be helpful to visualise the successful set of parameters.
- **model deployment in production**: when a model is trained, you can load it and deploy it instantly in production.

CHAPTER 2

How to setup ALP?

Requirements

Because the whole architecture has a lot of components we use Docker to manage the platform and isolates the services.

ALP has been developed to run on Ubuntu and has not been tested on other OS.

You should first install Docker and install nvidia-docker, then play a bit with docker (check if you can access your GPU with nvidia-docker). Then, you should be ready to install ALP.

You can then get ALP via pip:

```
pip install git+git://github.com/tboquet/python-alp
```

That will install ALP on your machine, and you will be able to launch it via the Command Line Interface.

Launching ALP with the CLI

To begin, we can generate a base configuration using ALP CLI. We choose to write configuration files on the host machine in order to be able to customize them easily afterwards.

```
alp --help
```

Will provide you with some help about the command line interface.

Generating a new configuration is as easy as:

```
alp --verbose genconfig --outdir=/path/to/a/directory
```

The command will generate a base configuration with one controler, one scikit learn worker and one keras worker. We specify the output directory where we want to write the three configuration files. The first file `alpdb.json` defines the connection between the database of models and other containers. The second file `alpapp.json` defines

the connections between the broker, its database and the other containers. The third file `containers.json` defines all the containers of the architecture. The linking is automatically done and ALP will use the newly created files to launch a new instance.

In any case, verify that the ports that you want to use are free for the broker to communicate with the monitoring containers and for the jupyter notebooks (if any) to run.

To start all the services you can use `alp service start`:

```
alp --verbose service start /path/to/a/directory
```

You can then take a look at the status of the containers:

```
alp --verbose status /path/to/a/directory
```

You should be able to access the Jupyter notebook on the port `440` of the machine where you launched the services.

CHAPTER 3

Some tutorials and usecases

Tutorial 0 : how to launch a basic experiment with keras or sklearn

Step 1 : launching alp

Follow the instructions in the setup section. We assume at this point that you have a Jupyter notebook running on the controller.

Step 2 : defining your model

You can follow step from *Step 2.1 : Keras* or from *Step 2.2 : Scikit learn* regarding if you want to use Keras or scikit-learn. In both case we will do the right imports, get some classification data, put them in the ALP format and instanciate a model. The important thing at the end of step 2 is to have the `data`, `data_val` and `model` objects and a model ready.

Step 2.1 : Keras

The following code gets some data and declares a simple artificial neural network with Keras:

```
# we import numpy and fix the seed
import numpy as np
np.random.seed(1337)  # for reproducibility

# we import alp and Keras tools that we will use
import alp
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.utils import np_utils
import keras.backend as K
from keras.optimizers import Adam
```

```
from alp.appcom.ensembles import HParamsSearch

# if you use tensorflow you must use this configuration
# so that it doesn't use all of the GPU's memory (default config)
import tensorflow as tf

config = tf.ConfigProto(allow_soft_placement=True)
config.gpu_options.allow_growth = True
session = tf.Session(config=config)
K.set_session(session)

batch_size = 128
nb_classes = 10
nb_epoch = 12

# input image dimensions
img_rows, img_cols = 28, 28
# number of features to use
nb_filters = 32

# the data, shuffled and split between train and test sets
(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
print('X_train shape:', X_train.shape)
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')

if K.image_dim_ordering() == 'th':
    X_train = X_train.reshape(X_train.shape[0], 1, img_rows, img_cols)
    X_test = X_test.reshape(X_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    X_train = X_train.reshape(X_train.shape[0], img_rows, img_cols, 1)
    X_test = X_test.reshape(X_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

# convert class vectors to binary class matrices
Y_train = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)

# put the data in the form ALP expects
data, data_val = dict(), dict()
data["X"] = X_train
data["y"] = Y_train
data_val["X"] = X_test
data_val["y"] = Y_test

# finally define and compile the model

model = Sequential()

model.add(Flatten(input_shape=input_shape))
model.add(Dense(nb_filters))
model.add(Activation('relu'))
```

```
model.add(Dropout(0.25))

model.add(Dense(128))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(nb_classes))
model.add(Activation('softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='adadelta',
              metrics=['accuracy'])
```

Note that we compile the model so that we also have information about the optimizer.

Step 2.2 : Scikit learn

The following code gets some data and declares a simple logistic regression with `scikit-learn`:

```
# some imports
from sklearn import cross_validation
from sklearn import datasets
from sklearn.linear_model import LogisticRegression

# get some data
iris = datasets.load_iris()
X_train, X_test, y_train, y_test = cross_validation.train_test_split(
                iris.data, iris.target, test_size=0.2, random_state=0)

# put the data in the form ALP expects
data, data_val = dict(), dict()
data["X"] = X_train
data["y"] = y_train
data_val["X"] = X_test
data_val["y"] = y_test

# define the model
model = LogisticRegression()
```

Please note that by default for the `LogisticRegression`, the `multi-class` parameter is set to OvR, that is to say one classifier per class. On the iris dataset, it means 3 classifiers. Unlike in Keras, the model is not compiled. So far, the measure of performance (validation metric) can only be the mean absolute error, but we will soon have several metrics working.

Step 3 : fitting the model with ALP

Step 3.1 : defining the Experiment

In ALP, the base object is the Experiment. An Experiment trains, predicts, saves and logs a model. So the first step is to import and define the Experiment object.

```
from alp.appcom.core import Experiment

expe = Experiment(model)
```

Step 3.2 : fit the model

You have access to two types of methods to fit the model.

- The `fit` and `fit_gen` methods allows you to fit the model in the same process.

  For the `scikit-learn` backend, you can launch the computation with the following command without extra arguments:

  ```
  expe.fit([data], [data_val])
  ```

  Note that the `data` and the `data_val` are put in lists.

  With Keras you might want to specify the number of epochs and the batch_size, as you would have done to fit directly a Keras `model` object. These arguments will flow trough to the final call. Note that they are not necessary for the fit, see the default arguments in the Keras model doc.

  ```
  expe.fit([data], [data_val], nb_epoch=2, batch_size=batch_size)
  ```

  In both cases, the model is trained and automatically saved in the databases.

- **The `fit_async` method sends the model to the broker container that will manage the training using the workers you def**
  For the `scikit-learn` backend:

  ```
  expe.fit_async([data], [data_val])
  ```

  For the Keras backend you still need to provide extra arguments to override the defaults.

  ```
  expe.fit_async([data], [data_val], nb_epoch=2, batch_size=batch_size)
  ```

  In both cases, the model is also trained and automatically saved in the databases.

Step 4 : Identifying and reusing the fitted model

Once the experiment has been fitted, you can access the id of the model in the db and load it to make prediction or access the parameters in the current process.

```
print(expe.mod_id)
print(expe.data_id)

expe.load_model(expe.mod_id, expe.data_id)
```

It's then possible to make predictions using the loaded model.

```
expe.predict(data['X'])
```

You could of course provide new data to the model. You can also load the model in another experiment.

Tutorial 1 : Simple Hyperparameter Tuning with ALP - sklearn models

In this tutorial, we will get some data, build an Experiment with a simple model and tune the parameters of the model to get the best performance on validation data (by launching several experiments). We will then reuse this best model on unseen test data an check that it's better than the untuned model. The whole thing will be using the asynchronous fit to highlight the capacity of ALP.

1 - Get some data

Let us start with the usual Iris dataset. Note that we will split the test set in 2 samples of size 25: the "validation" set to select the best model, and the "new" set to assess that the selected model was the best.

```
from sklearn import datasets
from sklearn.model_selection import train_test_split

# get some data
iris = datasets.load_iris()
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, test_size=50, random_state=0)
X_test_val, X_test_new, y_test_val, y_test_new = train_test_split(
    X_test, y_test, test_size=25, random_state=1)

# put it in ALP expected format
data, data_val, data_new = dict(), dict(), dict()
data["X"], data["y"] = X_train,  y_train
data_val["X"], data_val["y"] = X_test_val, y_test_val
data_new["X"], data_new["y"] = X_test_new, y_test_new
```

2 - Define an easy model and an ALP Experiment in a loop

We will define a simple LogisticRegression to demostrate how to use ensembles of experiments in ALP.

Let us first define an helper function.

```
import random
import sklearn.linear_model
from alp.appcom.core import Experiment
from operator import mul

def grid_search(grid_dict, tries, model_type='LogisticRegression'):
    ''' This function randomly build Experiments with different hyperparameters and␣
↪return the list of experiments.

    Args:
        grid_dict(dict) : hyperparameter grid from which to draw samples from
        tries(int) : number of model to be generated and tested
        async(bool) : should the fit be asynchronous
        model_type(string) : type of model to be tested (must be in sklearn.linear_
↪model)

    Returns:
        expes(list): a list of Experiments.

    '''

    expes = dict()

    # 1 - infos
    size_grid = reduce(mul, [len(v) for v in grid_dict.values()])
    print("grid size: {}".format(size_grid))
    print("tries: {}".format(tries))


    # 2 - models loop
```

```
    for i in range(tries):
        select_params =  {}
        key = [str(i)]
        for k, v in grid_dict.items():
            value = random.choice(v)
            select_params[k] = value
            key += ['{}:{}'.format(k, value)]
        model = getattr(sklearn.linear_model, model_type)(**select_params)
        expe = Experiment(model)
        expes['_'.join(key)] = expe
    return expes
```

Details of what this function does is: 1. display some infos about the size of the grid. 2. models loop: as many times as `tries`, it selects randomly a point in the hyperparameter grid, creates an Experiment object with the model parametrized with this point.

3 - Run the grid search

We use the *HParamsSearch* class to wrap several *Experiment*. For now, because the grid is defined outside of the class, you have to pass a dictionnary mapping experiments name to *Experiment*.

```
from alp.appcom.ensemble import HParamsSearch

# setting the seed for reproducibility: feel free to change it
random.seed(12345)

# defining the grid that will be explored
grid_tol = [i*10**-j for i in (1,2,5) for j in (1, 2, 3, 4, 5, 6)]
grid_C = [i*10**-j for i in (1,2,5) for j in (-2, -1, 1, 2, 3, 4, 5, 6)]
grid = {'tol':grid_tol, 'C':grid_C}

tries = 100

expes = grid_search(grid, tries)

# we define the ensemble with our experiments and a metric
ensemble = HParamsSearch(experiments=expes, metric='score', op=np.max)

results = ensemble.fit([data], [data_val])

label, predictions = ensemble.predict(data['X'])
print('Best model: {}'.format(label)
```

Note: You can also use the *fit_async()* method.

```
grid size : 432
tries : 100

Best model: 52_C:100_tol:1e-06
```

A word on the interpretation of the params:

- the parameter C is the regularisation parameter of the Logistic Regression. A small value of C means a higher L2 constraint on w (the L2 constraint is not applied on C, the intercept parameter). A larger C

can lead to overfitting, while a smaller value can lead to too much regularization. As such, it is the ideal candidate for automatic tuning.

- the tol parameter is the tolerance for stopping criteria. Our experiments did not show a strong impact of this parameter unless it was set to high values.

4 - Validation that the best model is better than the untuned one

ALP makes prediction with the loaded best model on the unseen data easy. The accuracy of the best model is decent (one mistake over 25 points).

```
label, predictions = ensemble.predict(data_new['X'])
print('Best model: {}'.format(label))
```

```
0.96
```

We can now create an untuned model (C=1 by default) and assess its precision on unseen data is lower that the tuned one.

```
model = sklearn.linear_model.LogisticRegression()
expe = Experiment(model)
expe.fit([data], [data_val])
pred_worst_new = expe.predict(X_test_new)
print(sklearn.metrics.accuracy_score(pred_worst_new, data_new["y"]))
```

```
0.88
```

Tutorial 2 : Feed simple data to your ALP Experiment

In this tutorial, we will build an Experiment with a simple model and fit it on various number of pieces of data. The aim of this tutorial is to explain the expected behaviour of ALP.

1 - Get some data

Let us start with the usual Iris dataset.

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# get some data
iris = datasets.load_iris()
X_train, X_val, y_train, y_val = train_test_split(
                    iris.data, iris.target, test_size=100, random_state=0)
```

The data is then put in the form ALP expects: a dictionary with a field 'X' for the input and a field 'y' for the output. Note that the same is done for the validation data.

```
data, data_val = dict(), dict()
data["X"], data["y"] = X_train,  y_train
data_val["X"], data_val["y"] = X_val, y_val
```

Let us shuffle the data some more. After these lines, 2 more datasets are created.

```
more_data, some_more_data = dict(), dict()
more_data["X"], some_more_data["X"], more_data["y"], some_more_data["y"] = train_test_
↪split(
                    iris.data, iris.target, test_size=75, random_state=1)
```

2 - Expected behaviour with sklearn

2.1 - Defining the experiment and model

We then define a first simple sklearn logistic regression.

```
from alp.appcom.core import Experiment
from  sklearn.linear_model import LogisticRegression

lr = LogisticRegression()
Expe = Experiment(lr)
```

2.2 - Fitting with one data set and one validation

Fitting one data set with one validation set is done this way:

```
Expe.fit([data],[data_val])
```

```
({'data_id': '1c59c0c562a5abdb84ad4f4a2c1868bf',
  'metrics': {'iter': nan,
   'score': [0.97999999999999998],
   'val_score': [0.93999999999999995]},
  'model_id': '5cabd17bbac6934fb487fa7f69bbda6e',
  'params_dump': u'/parameters_h5/
↪5cabd17bbac6934fb487fa7f69bbda6e1c59c0c562a5abdb84ad4f4a2c1868bf.h5'},
 None)
```

Now let's take a look at the results:

- there is a data_id field: that is where the data is stored in the appropriate collection.
- there is a model_id field: this is where the model architecture is stored.
- the param_dump field is path of a file where the *attributes* of the fitted model are stored.
- **the metrics field is itself a dictionary with several attributes:**
  - the iter field is here for compatibility with the keras backend.
  - the score field is model specific, you will have to look into sklearn's documentation to see what kind of metric is used. For the logistic regression, it is the accuracy. This field is then the accuracy of the fitted model on the training data.
  - the val_score is the score on the validation data (it is still the accuracy in this case).

You can access the full result of the experiment in the full_res attribut of the object.

```
Expe.full_res
```

```
{'data_id': '1c59c0c562a5abdb84ad4f4a2c1868bf',
 'metrics': {'iter': nan,
  'score': [0.97999999999999998],
  'val_score': [0.93999999999999995]},
 'model_id': '5cabd17bbac6934fb487fa7f69bbda6e',
 'params_dump': u'/parameters_h5/
↪5cabd17bbac6934fb487fa7f69bbda6e1c59c0c562a5abdb84ad4f4a2c1868bf.h5'}
```

Predicting the "more_data" on the model fitted on "data" is done this way.

```
pred_on_more_data = Expe.predict(more_data["X"])
```

At this point, pred_on_more_data is a vector of prediction. It's accuracy is obtained as follows:

```
accuracy_score(pred_on_more_data,more_data["y"])
```

```
0.95999999999999996
```

Now you can check that the full_res field of the Expe object was not modified during the predict call.

```
Expe.full_res
```

```
{'data_id': '1c59c0c562a5abdb84ad4f4a2c1868bf',
 'metrics': {'iter': nan,
  'score': [0.97999999999999998],
  'val_score': [0.93999999999999995]},
 'model_id': '5cabd17bbac6934fb487fa7f69bbda6e',
 'params_dump': u'/parameters_h5/
↪5cabd17bbac6934fb487fa7f69bbda6e1c59c0c562a5abdb84ad4f4a2c1868bf.h5'}
```

2.3 - Fitting with one data set and no validation:

If you want to fit an experiment and don't have a validation set, you need to specify a None in the validation field. Note that all the fields have changed. Since the data has changed, the data_id is different. The model created is a new one, so are the parameters. Finally, the metrics are different.

```
Expe.fit([some_more_data],[None])
```

```
({'data_id': '3554c1421fd9056e69c3cdf1b0ec8c3f',
  'metrics': {'iter': nan, 'score': [0.95999999999999996], 'val_score': [nan]},
  'model_id': 'ceb5d5632334515c4ebbd72a256bd421',
  'params_dump': u'/parameters_h5/
↪ceb5d5632334515c4ebbd72a256bd4213554c1421fd9056e69c3cdf1b0ec8c3f.h5'},
 None)
```

As a result, the model actually stored in the Experiment at that time of the code execution is not the same as in 2.2. You can check that by predicting on the more_data set and check that the score is not the same.

```
pred_on_more_data = Expe.predict(more_data["X"])
accuracy_score(pred_on_more_data,more_data["y"])
```

```
0.94666666666666666
```

2.4 - Fitting several dataset

Now it's an important point since the behavior of sklearn differs from the keras one: if you feed different datasets to an Experiment with an sklearn model, ALP proceeds as such:

- the first model is fitted, then the score and validation score are computed (on the first validation data, if provided).
- the second model is fitted, then the score and validation score are computed (on the second validation data, if provided).
- and so on

As a result, the parameters data_id, model_id and param_dumps in the full_res field of the Experiment of the following line are the one of the second model. The metrics (score and val_score) fields have a length of 2, one for each model.

Note that you can specify a None as validation set if you don't want to validate a certain model.

```
Expe.fit([data,more_data],[None,some_more_data])
```

```
({'data_id': '2767007837282c3da5a86cfe41b57cce',
  'metrics': {'iter': nan,
   'score': [0.97999999999999998, 0.9466666666666666],
   'val_score': [nan, 0.92000000000000004]},
  'model_id': 'c6f885968087dc779ce47f3f1af86a9b',
  'params_dump': u'/parameters_h5/
↪c6f885968087dc779ce47f3f1af86a9b2767007837282c3da5a86cfe41b57cce.h5'},
 None)
```

Tutorial 3 : Feed more data with Fuel or generators

Because we aim at supporting online learning on streamed data, we think that generators support was a good start. We support Fuel, a library that helps you to pre-process and yield chunks of data while being serializable.

1 - Create some data

You can easily use Fuel iterators in an Experiment. We will first create some fake data.

```
import fuel
import numpy as np
input_dim = 2
nb_hidden = 4
nb_class = 2
batch_size = 5
train_samples = 512
test_samples = 128
(X_tr, y_tr), (X_te, y_te) = get_test_data(nb_train=train_samples,
                                           nb_test=test_samples,
                                           input_shape=(input_dim,),
                                           classification=True,
                                           nb_class=nb_class)

y_tr = np_utils.to_categorical(y_tr)
y_te = np_utils.to_categorical(y_te)

data, data_val = dict(), dict()
```

```
X = np.concat([X_tr, X_te])
y = np.concat([y_tr, y_te])

inputs = [X, X]
outputs = [y]
```

2 - Transform the data

We then import an helper function that will convert our list of inputs to an HDF5 dataset. This dataset has a simple structure and we can divide it into multiple sets.

```
# we save the mean and the scale (inverse of the standard deviation)
# for each channel
scale = 1.0 / inputs[0].std(axis=0)
shift = - scale * inputs[0].mean(axis=0)

# for 3 sets, we need 3 slices
slices = [0, 256, 512]

# and 3 names
names = ['train', 'test', 'valid']

file_name = 'test_data_'
file_path_f = to_fuel_h5(inputs, outputs, slices, names, file_name, '/data_generator')
```

3 - Build your generator

The next step is to construct our Fuel generator using our dataset, a scheme and to transform the data so it's prepared for our model.

```
train_set = H5PYDataset(file_path_f,
                        which_sets=('train','test', 'valid'))

scheme = SequentialScheme(examples=128, batch_size=32)

data_stream_train = DataStream(dataset=train_set, iteration_scheme=scheme)

stand_stream_train = ScaleAndShift(data_stream=data_stream_train,
                                   scale=scale, shift=shift,
                                   which_sources=('input_X',))
```

4 - Build and wrap your model

We finally build our model and wrap it in an experiment.

```
inputs = Input(shape=(input_dim,), name='X')

x = Dense(nb_hidden, activation='relu')(inputs)
x = Dense(nb_hidden, activation='relu')(x)
predictions = Dense(nb_class, activation='softmax')(x)

model = Model(input=inputs, output=predictions)
```

```
model.compile(loss='categorical_crossentropy',
                optimizer='rmsprop',
                metrics=['accuracy'])

expe = Experiment(model)
```

5 - Train your model

We can finally use the *alp.appcom.core.Experiment.fit_gen()* method with our model and dataset.

```
expe.fit_gen([gen], [val], nb_epoch=2,
              model=model,
              metrics=metrics,
              custom_objects=cust_objects,
              samples_per_epoch=128,
              nb_val_samples=128)
```

You can also use *alp.appcom.core.Experiment.fit_gen_async()* with the same function parameters if you have a worker running.

```
expe.fit_gen([gen], [val], nb_epoch=2,
              model=model,
              metrics=metrics,
              custom_objects=cust_objects,
              samples_per_epoch=128,
              nb_val_samples=128)
```

Tutorial 4 : how to use custom layers for Keras with ALP

Because serialization of complex Python objects is still a challenge we will present a way of sending a custom layer to a Keras model with ALP.

1 - Get a dataset

We will work with the CIFAR10 dataset available via Keras.

```
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D
from keras.optimizers import SGD
from keras.utils import np_utils

from fuel.datasets.hdf5 import H5PYDataset
from fuel.schemes import SequentialScheme
from fuel.streams import DataStream
from fuel.transformers import ScaleAndShift

from alp.appcom.core import Experiment

from alp.appcom.utils import to_fuel_h5
```

```
import numpy as np

nb_classes = 10
nb_epoch = 25

# input image dimensions
img_rows, img_cols = 32, 32
# the CIFAR10 images are RGB
img_channels = 3

# the data, shuffled and split between train and test sets
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train = X_train/255
X_test = X_test/255

batch_size = 128
print('X_train shape:', X_train.shape)
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')

# convert class vectors to binary class matrices
Y_train = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)
```

2 - Build the generators

We build two generators, one for training and one for validation.

```
def dump_data():
    inputs = [np.concatenate([X_train, X_test])]
    outputs = [np.concatenate([Y_train, Y_test])]

    file_name = 'test_data_dropout'
    scale = 1.0 / inputs[0].std(axis=0)
    shift = - scale * inputs[0].mean(axis=0)

    file_path, i_names, o_names = to_fuel_h5(inputs, outputs, [0, 50000],
                                             ['train', 'test'],
                                             file_name,
                                             '/data_generator')
    return file_path, scale, shift, i_names, o_names

file_path, scale, shift, i_names, o_names = dump_data()


def make_gen(set_to_gen, nb_examples):
    file_path_f = file_path
    names_select = i_names
    train_set = H5PYDataset(file_path_f,
                            which_sets=set_to_gen)

    scheme = SequentialScheme(examples=nb_examples, batch_size=64)
```

```
    data_stream_train = DataStream(dataset=train_set, iteration_scheme=scheme)

    stand_stream_train = ScaleAndShift(data_stream=data_stream_train,
                                       scale=scale, shift=shift,
                                       which_sources=(names_select[-1],))
    return stand_stream_train, train_set, data_stream_train

train, data_tr, data_stream_tr = make_gen(('train',), 50000)
test, data_te, data_stream_te = make_gen(('test',), 10000)
```

3 - Build your custom layer

Imagine you want to reimplement a dropout layer. We could wrap it in a function that returns the object:

```
def return_custom():
    import keras.backend as K
    import numpy as np
    from keras.engine import Layer
    class Dropout_cust(Layer):
        '''Applies Dropout to the input.
        '''
        def __init__(self, p, **kwargs):
            self.p = p
            if 0. < self.p < 1.:
                self.uses_learning_phase = True
            self.supports_masking = True
            super(Dropout_cust, self).__init__(**kwargs)

        def call(self, x, mask=None):
            if 0. < self.p < 1.:
                x = K.in_train_phase(K.dropout(x, level=self.p), x)
            return x

        def get_config(self):
            config = {'p': self.p}
            base_config = super(Dropout_cust, self).get_config()
            return dict(list(base_config.items()) + list(config.items()))
    return Dropout_cust
```

4 - Build you model

We then define our model and call our function to instanciate this custom layer.

```
model = Sequential()

model.add(Convolution2D(64, 3, 3, border_mode='same',
                        input_shape=(img_channels, img_rows, img_cols)))
model.add(Activation('relu'))
model.add(Convolution2D(64, 3, 3))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Convolution2D(128, 3, 3, border_mode='same'))
```

```
model.add(Activation('relu'))
model.add(Convolution2D(128, 3, 3))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(1024))
model.add(Activation('relu'))
model.add(return_custom()(0.5))
model.add(Dense(nb_classes))
model.add(Activation('softmax'))

sgd = SGD(lr=0.02, decay=1e-7, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])
```

5 - Fit your model

We then map the name of the custom object to our function that returns the custom object in a dictionnary. After wrapping the model in an *alp.appcom.core.Experiment()*, we call the *alp.appcom.core.Experiment.fit_gen()* method and send the custom_objects.

```
custom_objects = {'Dropout_p': return_custom}

expe = Experiment(model)

results = expe.fit_gen_async([train], [test], nb_epoch=nb_epoch,
                             model=model,
                             metrics=['accuracy'],
                             samples_per_epoch=50000,
                             nb_val_samples=10000,
                             verbose=2,
                             custom_objects=custom_objects))
```

Note: Why do we wrap this class and all the dependencies?

We use dill to be able to serialize object but unfortunatly, handling class with inheritance is not doable. It's also easier to pass the information about all the dependencies of the object. All the dependencies and your custom objects will be instanciated during the evaluation of the function so that it will be available in the `__main__`. This way the information could be sent to workers without problems.

CHAPTER 4

Userguide

In this user guide we explain with more details how to use the architecture and the main objects available in ALP.

Warning: The userguide is currently under construction. Please visit this section in a few days.

Services

In this section we describe the different services (such as the Jupyter Notebook, RabbitMQ, the Models databases ...) running in separated Docker containers (resp. the Controller, the Broker, Mongos Models ...). As we tried to separate the services as much as possible, sometimes the container is assimilated to the service.

Controller

The Controller is the user endpoint of the library. By default, it serves a Jupyter notebook in which the user sends commands (such as *import alp*). You can also use it to run an application using ALP for either training or prediction.

Mongo Models

Mongo Models is a container that runs a MongoDB service in which the architecture of the models that are trained through ALP are saved.

Mongo Results

Mongo Results is a container that runs a MongoDB service in wich the meta informations about a tasks is saved.

Broker

Also called scheduler in the architecture, it distributes the tasks and gather the results.

Worker(s)

The workers run the tasks and send results to the MongoDB services. Each backend need at least one worker consuming from the right queue.

Job monitor

You can plug several containers to monitor jobs.

Experiment

Experiment section

CHAPTER 5

Reference

alp

class **Experiment** (*model=None*, *metrics=None*, *verbose=0*)

An Experiment trains, predicts, saves and logs a model

Variables

- **model** (*model*) – the model used in the experiment
- **metrics** (*list*) – a list of callables

fit (*data*, *data_val*, *model=None*, **args*, ***kwargs*)

Build and fit a model given data and hyperparameters

Parameters

- **data** (*list(dict)*) – a list of dictionnaries mapping inputs and outputs names to numpy arrays for training.
- **data_val** (*list(dict)*) – a list of dictionnaries mapping inputs and outputs names to numpy arrays for validation.
- **model** (*model, optionnal*) – a model from a supported backend

Returns the id of the model in the db, the id of the data in the db and path to the parameters.

fit_async (*data*, *data_val*, *model=None*, **args*, ***kwargs*)

Build and fit asynchronously a model given data and hyperparameters

Parameters

- **data** (*list(dict)*) – a list of dictionnaries mapping inputs and outputs names to numpy arrays for training.
- **data_val** (*list(dict)*) – a list of dictionnaries mapping inputs and outputs names to numpy arrays for validation.
- **model** (*model, optionnal*) – a model from a supported backend

Returns the id of the model in the db, the id of the data in the db and a path to the parameters.

fit_gen(*gen_train*, *data_val*, *model=None*, **args*, ***kwargs*)
Build and fit asynchronously a model given data and hyperparameters

Parameters

- **gen_train** (*list(dict)*) – a list of generators.
- **data_val** (*list(dict)*) – a list of dictionnaries mapping inputs and outputs names to numpy arrays or generators for validation.
- **model** (*model, optionnal*) – a model from a supported backend

Returns the id of the model in the db, the id of the data in the db and a path to the parameters.

fit_gen_async(*gen_train*, *data_val*, *model=None*, **args*, ***kwargs*)
Build and fit asynchronously a model given generator(s) and hyperparameters.

Parameters

- **gen_train** (*list(dict)*) – a list of generators.
- **data_val** (*list(dict)*) – a list of dictionnaries mapping inputs and outputs names to numpy arrays or generators for validation.
- **model** (*model, optionnal*) – a model from a supported backend

Returns the id of the model in the db, the id of the data in the db and a path to the parameters.

load_model(*mod_id=None*, *data_id=None*)
Load a model from the database form it's mod_id and data_id

Parameters

- **mod_id** (*str*) – the id of the model in the database
- **data_id** (*str*) – the id of the data in the database

predict(*data*, **args*, ***kwargs*)
Make predictions given data

Parameters **data** (*np.array*)

Returns an np.array of predictions

predict_async(*data*, **args*, ***kwargs*)
Make predictions given data

Parameters **data** (*np.array*)

Returns an np.array of predictions

alp.celapp module

Celery config

alp.config module

autodoc: failed to import module u'alp.config'; the following exception was raised: Traceback (most recent call last): File "/home/docs/checkouts/readthedocs.org/user_builds/python-alp/envs/latest/local/lib/python2.7/site-packages/sphinx/ext/autodoc.py", line 551, in import_object __import__(self.modname) ImportError: No module named config

alp.appcom package

A simple module to perform training and prediction of models

Using celery, this module helps to schedule the training of models if the users send enough models in a short period of time.

class **Experiment** (*model=None*, *metrics=None*, *verbose=0*)
Bases: `object`

An Experiment trains, predicts, saves and logs a model

Variables

- **model** (*model*) – the model used in the experiment
- **metrics** (*list*) – a list of callables

data_id

fit (*data*, *data_val*, *model=None*, **args*, ***kwargs*)
Build and fit a model given data and hyperparameters

Parameters

- **data** (*list(dict)*) – a list of dictionnaries mapping inputs and outputs names to numpy arrays for training.
- **data_val** (*list(dict)*) – a list of dictionnaries mapping inputs and outputs names to numpy arrays for validation.
- **model** (*model, optionnal*) – a model from a supported backend

Returns the id of the model in the db, the id of the data in the db and path to the parameters.

fit_async (*data*, *data_val*, *model=None*, **args*, ***kwargs*)
Build and fit asynchronously a model given data and hyperparameters

Parameters

- **data** (*list(dict)*) – a list of dictionnaries mapping inputs and outputs names to numpy arrays for training.
- **data_val** (*list(dict)*) – a list of dictionnaries mapping inputs and outputs names to numpy arrays for validation.
- **model** (*model, optionnal*) – a model from a supported backend

Returns the id of the model in the db, the id of the data in the db and a path to the parameters.

fit_gen (*gen_train*, *data_val*, *model=None*, **args*, ***kwargs*)
Build and fit asynchronously a model given data and hyperparameters

Parameters

- **gen_train** (*list(dict)*) – a list of generators.
- **data_val** (*list(dict)*) – a list of dictionnaries mapping inputs and outputs names to numpy arrays or generators for validation.
- **model** (*model, optionnal*) – a model from a supported backend

Returns the id of the model in the db, the id of the data in the db and a path to the parameters.

fit_gen_async(*gen_train*, *data_val*, *model=None*, **args*, ***kwargs*)
Build and fit asynchronously a model given generator(s) and hyperparameters.

Parameters

- **gen_train** (*list(dict)*) – a list of generators.
- **data_val** (*list(dict)*) – a list of dictionnaries mapping inputs and outputs names to numpy arrays or generators for validation.
- **model** (*model, optionnal*) – a model from a supported backend

Returns the id of the model in the db, the id of the data in the db and a path to the parameters.

load_model(*mod_id=None*, *data_id=None*)
Load a model from the database form it's mod_id and data_id

Parameters

- **mod_id** (*str*) – the id of the model in the database
- **data_id** (*str*) – the id of the data in the database

mod_id

model_dict

params_dump

predict(*data*, **args*, ***kwargs*)
Make predictions given data

Parameters **data** (*np.array*)

Returns an np.array of predictions

predict_async(*data*, **args*, ***kwargs*)
Make predictions given data

Parameters **data** (*np.array*)

Returns an np.array of predictions

Ensembles module

class Ensemble(*experiments*)
Bases: `object`

Base class to build experiments containers able to execute batch sequences of action. Must implement the *fit*, *fit_gen*, *fit_async fit_gen_async* methods

Parameters **experiments** (*dict or list*) – experiments to be wrapped. If a dictionnary is passed, it should map experiment names to experiments.

fit(*data*, *data_val*, **args*, ***kwargs*)

fit_async(*data*, *data_val*, **args*, ***kwargs*)

fit_gen(*data*, *data_val*, **args*, ***kwargs*)

fit_gen_async(*data*, *data_val*, **args*, ***kwargs*)

plt_summary()

predict(*data*, *data_val*, **args*, ***kwargs*)

summary(*metrics*, *verbose=False*)

class `HParamsSearch`(*experiments*, *hyperparams=None*, *metric=None*, *op=None*)

Bases: `alp.appcom.ensembles.Ensemble`

Hyper parameters search class

Train several experiments with different hyperparameters and save results. Wraps the training process so that it's possible to access results easily.

Parameters

- **experiments** (*dict or list*) – experiments to be wrapped. If a dictionnary is passed, it should map experiment names to experiments
- **hyperparams** (*dict*) – a dict of hyperparameters
- **metric** (*str*) – the name of a metric used in the experiments
- **op** (*str*) – an operator to select a model

`fit`(*data*, *data_val*, **args*, ***kwargs*)

Apply the fit method to all the experiments

Parameters **see 'alp.core.Experiment.fit'**

Returns a list of results

`fit_async`(*data*, *data_val*, **args*, ***kwargs*)

Apply the fit_async method to all the experiments

Parameters **see :meth:'alp.appcom.core.Experiment.fit_async'**

Returns a list of results

`fit_gen`(*data*, *data_val*, **args*, ***kwargs*)

Apply the fit_gen method to all the experiments

Parameters **see :meth:'alp.appcom.core.Experiment.fit_gen'**

Returns a list of results

`fit_gen_async`(*data*, *data_val*, **args*, ***kwargs*)

Apply the fit_gen_async method to all the experiments

Parameters **see :meth:'alp.appcom.core.Experiment.fit_gen_async'**

Returns a list of results

`predict`(*data*, *metric=None*, *op=None*, *partial=False*, **args*, ***kwargs*)

Apply the predict method to all the experiments

Parameters

- **see :meth:'alp.appcom.core.Experiment.predict'**
- **metric** (*str*) – the name of the metric to use
- **op** (*function*) – an operator returning the value to select an experiment

Returns an array of results

`summary`(*metrics*, *verbose=False*)

Build a results table using individual results from models

Parameters

- **verbose** (*bool*) – if True, print a description of the results
- **metrics** (*dict*) – a dictionnary mapping metric's names to ops.

Returns a pandas DataFrame of results

get_best(*experiments*, *metric*, *op*, *partial=False*)
Helper function for manipulation of a list of experiments

In case of equality in the metric, the behaviour of op_arg determines the result.

Parameters

- **experiments** (*list*) – a list of experiments
- **metric** (*str*) – the name of a metric used in the experiments
- **op** (*function*) – operation to perform with the metric (optional)
- **partial** (*bool*) – if True will pass an experiment without result. Raise an error otherwise.

Utility functions for the appcom module

background(*f*)
a threading decorator use @background above the function you want to run in the background

check_gen(*iterable*)
Check if the last object of the iterable is an iterator

Parameters **iterable** (*list*) – a list containing data.

Returns True if the last object is a generator, False otherwise.

get_nb_chunks(*generator*)
Get the number of chunks that yields a generator

Parameters **generator** – a Fuel generator

Returns number of chunks (int)

imports(*packages=None*)
A decorator to import packages only once when a function is serialized

Parameters **packages** (*list or dict*) – a list or dict of packages to import. If the object is a dict, the name of the import is the key and the value is the module. If the object is a list, it's transformed to a dict mapping the name of the module to the imported module.

init_backend(*model*)
Initialization of the backend

Parameters **backend** (*str*) – only 'keras' or 'sklearn' at the moment

Returns the backend, the backend name and the backend version

list_to_dict(*list_to_transform*)
Transform a list of object to a dict

Parameters **list_to_transform** (*list*) – the list to transform

Returns a dictionnary mapping names of the objects to objects

max_v_len(*iterable_to_check*)
Returns the max length of a list of iterable

norm_iterator(*iterable*)
returns a normalized iterable of tuples

pickle_gen(*gen_train*, *data_val*)
Check and serialize the validation data object and serialize the training data generator.

Parameters

- **gen_train** (*generator*) – the training data generator
- **data_val** (*dict or generator*) – the training data object

Returns normalized datasets

switch_backend(*backend_name*)
Switch the backend based on it's name

Parameters **backend_name** (*str*) – the name of the backend to import

Returns the backend asked

to_fuel_h5(*inputs*, *outputs*, *slices*, *names*, *file_name*, *file_path=''*)
Transforms list of numpy arrays to a structured hdf5 file

Parameters

- **inputs** (*list*) – a list of inputs(numpy.arrays)
- **outputs** (*list*) – a list of outputs(numpy.arrays)
- **slices** (*list*) – a list of int representing the end of a slice and the begining of another slice. The last slice is automatically added if missing (maximum length of the inputs).
- **names** (*list*) – a list of names for the datasets
- **file_name** (*str*) – the name of the file to save.
- **file_path** (*str*) – the path where the file is located

Returns The file full path

window(*seq*, *n=2*)
Returns a sliding window (of width n) over data from the iterable

alp.backend package

Adaptor for the Keras backend

Compilation & cache

The models are compiled on the fly after the build. If the model is already compiled and in the *COMPILED_MODEL* dictionnary mapping the models id to the in memory compiled function, this function is used instead.

---

build_predict_func(*mod*)
Build Keras prediction functions based on a Keras model

Using inputs and outputs of the graph a prediction function (forward pass) is compiled for prediction purpose.

Parameters **mod** (*keras.models*) – a Model or Sequential model

Returns a Keras (Theano or Tensorflow) function

check_validation(*dv*)

deserialize(*name_d*, *func_code_d*, *args_d*, *clos_d*, *type_obj*)
A function to deserialize an object serialized with the serialize function.

Parameters

- **name_d** (*unicode*) – the dumped name of the object
- **func_code_d** (*unicode*) – the dumped byte code of the function
- **args_d** (*unicode*) – the dumped information about the arguments
- **clos_d** (*unicode*) – the dumped information about the function closure

Returns a deserialized object

get_backend()

get_function_name(*o*)
Utility function to return the model's name

Parameters **o** (*object*) – an object to check

Returns The name(str) of the object

model_from_dict_w_opt(*model_dict*, *custom_objects=None*)
Builds a model from a serialized model using *to_dict_w_opt*

Parameters

- **model_dict** (*dict*) – a serialized Keras model
- **custom_objects** (*dict, optionnal*) – a dictionnary mapping custom objects names to custom objects (Layers, functions, etc.)

Returns A Keras.Model which is compiled if the information about the optimizer is available.

save_params(*model*, *filepath*)

serialize(*cust_obj*)
A function to serialize custom objects passed to a model

Parameters **cust_obj** (*callable*) – a custom layer or function to serialize

Returns a dict of the serialized components of the object

to_dict_w_opt(*model*, *metrics=None*)
Serialize a model and add the config of the optimizer and the loss.

Parameters

- **model** (*keras.Model*) – the model to serialize
- **metrics** (*list, optionnal*) – a list of metrics to monitor

Returns a dictionnary of the serialized model

train(*model*, *data*, *data_val*, *size_gen*, *generator=False*, **args*, ***kwargs*)
Fit a model given hyperparameters and a serialized model

Parameters

- **model** (*dict*) – a serialized keras.Model
- **data** (*list*) – a list of dict mapping inputs and outputs to lists or dictionnaries mapping the inputs names to np.arrays
- **data_val** (*list*) – same structure than *data* but for validation

Returns the loss (list), the validation loss (list), the number of iterations, and the model

Adaptor for the sklearn backend

get_backend()

getname(*model*, *call=True*)

load_params(*model*, *filepath*)
Load the attributes that have been dumped in a h5 file in a model.

Parameters

- **model** (*sklearn.BaseEstimator*) – a sklearn model (in SUPPORTED).
- **filepath** (*string*) – the file name where the attributes should be read.

Returns the model with updated parameters.

model_from_dict_w_opt(*model_dict*, *custom_objects=None*)
Builds a sklearn model from a serialized model using *to_dict_w_opt*

Parameters

- **model_dict** (*dict*) – a serialized sklearn model
- **custom_objects** (*dict, optionnal*) – a dictionnary mapping custom objects names to custom objects (callables, etc.)

Returns A new sklearn.BaseEstimator (in SUPPORTED) instance. The attributes are not loaded.

save_params(*model*, *filepath*)

Dumps the attributes of the (generally fitted) model in a h5 file.

Parameters

- **model** (*sklearn.BaseEstimator*) – a sklearn model (in SUPPORTED).
- **filepath** (*string*) – the file name where the attributes should be written.

to_dict_w_opt(*model*, *metrics=None*)

Serializes a sklearn model. Saves the parameters, not the attributes.

Parameters

- **model** (*sklearn.BaseEstimator*) – the model to serialize, must be in SUPPORTED
- **metrics** (*list, optionnal*) – a list of metrics to monitor

Returns a dictionnary of the serialized model

train(*model*, *data*, *data_val*, *size_gen*, *generator=False*, **args*, ***kwargs*)
Fit a model given parameters and a serialized model

Parameters

- **model** (*dict*) – a serialized sklearn model
- **data** (*list*) – a list of dict mapping inputs and outputs to lists or dictionnaries mapping the inputs names to np.arrays XOR -a list of fuel generators
- **data_val** (*list*) – same structure than *data* but for validation.
- **it is possible to feed generators for data and plain data for data_val.**
- **it is not possible the other way around.**

Returns the loss (list), the validation loss (list), the number of iterations, and the model

`typeconversion`(*v*)

Utility function to ease serialization of custom types (namely np.types)

Parameters

- **v** (*np.ndarray, list, other*) – the object to return as a jsonable object.
- **If the type of v is not a np.ndarray or a list, the type of the** – returned object is unchanged.

Returns a jsonable object, which type depends on the type of v

CHAPTER 6

Project evolution

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Documentation improvements

ALP could always use more documentation, whether as part of the official ALP docs, in docstrings, or even on the web in blog posts, articles, and such.

Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/tboquet/python-alp/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

Development

To set up *python-alp* for local development:

1. Fork python-alp (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/python-alp.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

Tips

To run a subset of tests:

```
tox -e envname -- py.test -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

Changelog

0.3.0 (2017-01-17)

- Command Line Interface to launch services.
- sklearn backend is stable with 12 models supported and all sklearn metrics
- Keras backend supports custom objects
- asynchronous fit is stable for all backends
- fuel generators are supported as training data and validation data source
- Ensemble class in core (abstraction for many models)
- Basic HyperParameter optimisation
- Better documentation

0.2.0 (2016-04-21)

- Keras backend is stable

0.1.0 (2016-04-12)

- First release

Authors

- Thomas Boquet - https://github.com/tboquet
- Paul Lemaître

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

P

S

T

W