Cortex Documentation

Devon Hjelm

Sep 17, 2018

User Documentation

CHAPTER 1

Installation

1.1 Prerequisites

1.1.1 Visdom

```
$pip install visdom
```

1.2 From Source

```
$git clone https://github.com/rdevon/cortex2.0.git
$cd cortex2.0
$pip install .
```

CHAPTER 2

Getting Started

2.1 Configuration

The first thing to do is to set up the config.yaml. This file is user-specific (it got tracked at some point, so I need to fix this), and will tell cortex everything user-specific regarding data locations, visualation, and outputs.

```
$ rm -rf ~/.cortex.yml
$ cortex setup
```

2.1.1 Configuration File Example

Located at `~/.cortex.yml`

```
torchvision_data_path: /data/milatmp1/hjelmdev/data/
data_paths: {
 Imagenet-12: /data/lisa/data/ImageNet2012_jpeg, CelebA: /tmp/hjelmdev/CelebA}viz: {
 font: /usr/share/fonts/truetype/liberation/LiberationSerif-Regular.ttf, server:
↪'http://132.204.26.180'}
out_path: /data/milatmp1/hjelmdev/outs/
```

These are as follows:

- torchvision_data_path: the path to all torchvision-specific datasets (details can be found in torchvision.datasets)
- data_paths: user-specified custom datasets. Currently, only support is for image folders (a la imagenet), but other dataset types (e.g., text) are planned in the near-future.
- vis: visdom specific arguments.
- out_path: Out path for experiment outputs

2.1.2 Usage

cortex –help

2.1.3 Built-ins

setup Setup cortex configuration.

GAN Generative adversarial network.

VAE Variational autoencoder.

AdversarialAutoencoder Adversarial Autoencoder.

ALI Adversarially learned inference.

ImageClassification Basic image classifier.

GAN_MINE GAN + MINE.

2.1.4 Options

-h, --help show this help message and exit

-o OUT_PATH, --out_path OUT_PATH Output path directory. All model results will go here. If a new directory, a new one will be created, as long as parent exists.

-n NAME, --name NAME Name of the experiment. If given, base name of output directory will be *–name*. If not given, name will be the base name of the *–out_path*

-r RELOAD, --reload RELOAD Path to model to reload.

-M LOAD_MODELS, --load_models LOAD_MODELS Path to model to reload. Does not load args, info, etc

-m META, --meta META TODO

-c CONFIG_FILE, --config_file CONFIG_FILE Configuration yaml file. See *exps/* for examples

-k, --clean Cleans the output directory. This cannot be undone!

-v VERBOSITY, --verbosity VERBOSITY Verbosity of the logging. (0, 1, 2)

-d DEVICE, --device DEVICE TODO

2.1.5 Usage Example

To run an experiment.

```
cortex GAN --d.source CIFAR10 --d.copy_to_local
```

2.1.6 Custom models

It is possible to run experiments with custom models made with Pytorch under the Cortex framework. For doing so, the model has to be added to the demos folder under the root of the project. You can have a look to the given demo autoencoder and classifier already implemented. The main difference is that, rather than registering the plugins, the run function of main.py has to be called. For example,

```
if __name__ == '__main__':
classifier = MyClassifier()
run(model=classifier)
```

To run an experiment with a custom model.

```
python my_model.py --d.source <Dataset> --d.copy_to_local
```

CHAPTER 3

cortex

3.1 cortex package

3.1.1 Subpackages

3.1.2 Submodules

3.1.3 cortex.main module

3.1.4 cortex.plugins module

3.1.5 Module contents

CHAPTER 4

Develop

4.1 Documentation

Make sure that the cortex package is installed and configured. For development purpose, if you are making changes to documentation, for example modifications inside docstrings or changes in some .rst files

4.1.1 Building Documentation

To build the documentation, the docs.py script under the root of the project is facilitating the process. Before making a Pull Request to the remote repository, you should run the script.

```
$ python docs.py
```

4.1.2 Serving Documentation Locally

If you want to have a look at your changes before making a Pull Request on GitHub, it is possible to serve locally the generated html files.

```
$ cd docs/build/html
$ python -m http.server 8000 --bind 127.0.0.1
```

CHAPTER 5

Custom demos

While cortex has built-in functionality, but it is meant to meant to be used with your own modules. An example of making a model that works with cortex can be found at: https://github.com/rdevon/cortex/blob/master/demos/demo_classifier.py and https://github.com/rdevon/cortex/blob/master/demos/demo_custom_ae.py

Documentation on the API can be found here: https://github.com/rdevon/cortex/blob/master/cortex/plugins.py

For instance, the demo autoencoder can be used as:

```
python cortex/demos/demo_custom_ae.py --help
```

CHAPTER 6

A walkthrough a custom classifier:

Let's look a little more closely at the autoencoder demo above to see what's going on. cortex relies on using and overriding methods of plugins classes.

First, let's look at the methods, `build`, `routine`, and `visualize`. These are special methods for the plugin that can be overridden to change the behavior of your model for your needs.

The signature of these functions look like:

```
def build(self, dim_z=64, dim_encoder_out=64):
    ...

def routine(self, inputs, targets, ae_criterion=F.mse_loss):
    ...

def visualize(self, inputs, targets):
    ...
```

Each of these functions have arguments and keyword arguments. Note that the keyword arguments showed up in the help in the above example. This is part of the functionality of cortex: it manages your hyperparameters to these functions, organizes them, and provides command line control automatically. Even the docstrings are used in the command line, so other users can get the usage docs directly from there.

The arguments are *data*, which are to be manipulated as needed in those methods. These are for the most part handled automatically, but all of these methods can be used as normal functions as well.

6.1 Building models

The `build` function takes the hyperparameters and sets networks.

```
class Autoencoder(nn.Module):
    def __init__(self, encoder, decoder):
        super(Autoencoder, self).__init__()
        self.encoder = encoder
```

(continues on next page)

(continued from previous page)

```
        self.decoder = decoder

    def forward(self, x, nonlinearity=None):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded

...

    def build(self, dim_z=64, dim_encoder_out=64):
        encoder = nn.Sequential(
            nn.Linear(28, 256),
            nn.ReLU(True),
            nn.Linear(256, 28),
            nn.ReLU(True))
        decoder = nn.Sequential(
            nn.Linear(28, 256),
            nn.ReLU(True),
            nn.Linear(256, 28),
            nn.Sigmoid())
        self.nets.ae = Autoencoder(encoder, decoder)
```

All that's being done here is the hyperparameters are being used to create an instance of an `nn.Module` subclass, which is being added to the set of "nets". Note that they keyword `ae` is very important, as this is going to be how you retrieve your nets and define their losses farther down.

Also note that cortex *only* currently supports `nn.Module` subclasses from Pytorch.

CHAPTER 7

Defining losses and results

Adding losses and results from your model is easy, just compute your graph given you models and data, then add the losses and results by setting those members:

```
def routine(self, inputs, targets, ae_criterion=F.mse_loss):
    encoded = self.nets.ae.encoder(inputs)
    outputs = self.nets.ae.decoder(encoded)
    r_loss = ae_criterion(
        outputs, inputs, size_average=False) / inputs.size(0)
    self.losses.ae = r_loss
```

Additional results can be added similarly. For instance, in the demo classifier:

```
def routine(self, inputs, targets, criterion=nn.CrossEntropyLoss()):
    ...
    classifier = self.nets.classifier

    outputs = classifier(inputs)
    predicted = torch.max(F.log_softmax(outputs, dim=1).data, 1)[1]

    loss = criterion(outputs, targets)
    correct = 100. * predicted.eq(
        targets.data).cpu().sum() / targets.size(0)

    self.losses.classifier = loss
    self.results.accuracy = correct
```

CHAPTER 8

Visualization

Cortex allows for visualization using visdom, and this can be defined in a similar way as above:

```
def visualize(self, images, inputs, targets):
    predicted = self.predict(inputs)
    self.add_image(images.data, labels=(targets.data, predicted.data),
                   name='gt_pred')
```

See the ModelPlugin API for more more details.

CHAPTER 9

Putting it together

Finally, we can specify default arguments:

```
defaults = dict(
    data=dict(
        batch_size=dict(train=64, test=64), inputs=dict(inputs='images')),
    optimizer=dict(optimizer='Adam', learning_rate=1e-4),
    train=dict(save_on_lowest='losses.ae'))
```

and then add `cortex.main.run` to `__main__`:

```
if __name__ == '__main__':
    autoencoder = AE()
    run(model=autoencoder)
```

And that's it. cortex also allows for lower-level functions to be overridden (e.g., train_step, eval_step, train_loop, etc) with more customizability coming soon. For more examples of usage, see the built-in models: https://github.com/rdevon/cortex/tree/master/cortex/built_ins/models