

Working with conda

I would suggest conda as package management system when working with python. For compiling software the system-installation of python should be used. This prevents mismatches between package versions, since programmers know what versions e.g. come with a Linux distro. Both installations can exist side-to-side with each other. Click [here](#) for installation instructions. Miniconda is a minimal installation, use it when you don't have much space available or know which packages you want to install. Anaconda includes a large collection of packages. During installation I would recommend choosing not to activate conda by default in every terminal session. You can activate your conda environment after installation as follows:

```
$ conda activate
(base) $
```

This will set all necessary paths. While inside of a conda environment it can be closed with

```
(base) $ conda deactivate
$
```

Furthermore, virtual environments are a useful tool to keep only the dependencies for a project that you need, since different projects might require different versions of the same package. This makes managing several projects easier. To create a new virtual environment use the following command:

```
$ conda create --name myenv
```

myenv can be replaced by the desired name. You can enter a specific env with

```
$ conda activate myenv
(myenv) $
```

Packages can be installed with

```
(myenv) $ conda install ...
```

By default conda will install the package in the active environment. With the `-n` flag a user can install packages in an environment without activating it. Same goes for commands like `conda update`, e.g.

```
$ conda update -n myenv --all
```

Will update all packages in myenv. To show all available environments one can use

```
$ conda env list
```

To see all installed packages use

```
(myenv) $ conda list
```

Packages in conda come in channels from which they can be installed. The `-c` flag during installation can be used to mark a specific channel. A popular channel for example is `conda-forge`. There one can find many useful packages like `pyroot` and `cuda-related` packages. There can be a list of different channels, these are usually given in decreasing priority. conda will try to install a package and all dependencies from the channel with highest priority first. To list all active channels use

```
(myenv) $ conda config --show channels
```

To add the channel `conda-forge` in the virtual environment use

```
(myenv) $ conda config --add channels conda-forge
```

I would recommend adding the following channel as highest priority to get the official pytorch releases

```
(myenv) $ conda config --add channels pytorch
```

If you want to delete an environment use the following command

```
$ conda env remove --name myenv
```

More detailed guides for working with virtual environments and channels can be found on the [conda webpage](#).

Jupyter Notebook

Jupyter notebooks are a great resource for testing. Data has to be loaded only once and afterwards one can at leisure adjust plots or test the functionality of a function. When working with virtual environments one could install the notebook everywhere and use it from that environment, but it is also possible to install and setup the notebook just once and afterwards use kernels from every virtual environment with that single installation.

Install conda and activate your base environment or create a environment for the notebook. Then use the following command to install Jupyter Notebook

```
(base) $ conda install notebook
(base) $ conda install nb_conda_kernels
```

The package `nb_conda_kernels` makes it possible to choose kernels from other environments. Additionally one can install Notebook extensions. These can be useful, but are not necessary

```
(base) $ conda install jupyter_contrib_nbextensions
```

To be able to use kernels from different virtual environments in the Jupyter Notebook one needs to install one additional package. Activate the virtual environment and use

```
(myenv) $ conda install ipykernel
```

Now one should be able to choose a kernel from the virtual environment for the notebook in your Jupyter environment. Also, when using new to create a notebook one should be able to choose between kernels from the different environments. The package `ipykernel` has to be installed in every environment that the Jupyter Notebook should have access to.

Workin with the example scripts

There are four example scripts provided with some training, test and normalization data. These scripts are self-contained and are able to run. You will need the following packages:

```
$ conda install numpy ray-tune hyperopt gpy -c conda-forge
```

Pytorch is used in the scripts for constructing and training neural networks. For installtion insctructions it is best to check the official website. The first question is, has your system a CUDA capable gpu? If not, then you can use the following command

```
$ conda install pytorch torchvision torchaudio cpuonly -c pytorch
```

If you have a CUDA-capable gpu the next question is which CUDA version is running on your PC. You can check this with the terminal command

```
$ nvidia-smi
```

At the time of writing e.g. I'm using a GTX 1080 with CUDA 11.2. The newest pytorch version is 1.10 with possible CUDA-versions 10.2 and 11.3. Since pytorch comes with the `cuda-toolkit`-package we don't need to match the version exactly, our CUDA needs to be equal or newer than `cuda-toolkit` and the rest should just work. So the following command would be used in that case to install pytorch:

```
$ conda install pytorch torchvision torchaudio cudatoolkit=10.2 -c pytorch
```

You should now be able to run the scripts. There are some more packages that can be very useful

```
$ conda install pandas scikit-learn matplotlib seaborn -c conda-forge
```

`pandas` is a tool for data analysis. It groups data into rows and coloums, which feel similar to tables in statistical programs like Excel, but in python and more efficient. Check out their starting guides, if this sounds usefull. `Scikit-learn` contains many methods for linear regression or tree-based models, it is one of the standard

libraries for machine learning in python. Matplotlib is the standard library for plotting, check their huge collection of examples. Additionally one can use seaborn, which builds upon matplotlib. It is a high-level interface for drawing attractive plots.

Now a few words regarding the four scripts. The scripts use Ray Tune for parallel model training and hyperparameter optimization. So many different trial runs can be started and all the book keeping and resource management will be done by ray. Also the framework makes scaling from cpu to gpu to multiple gpus very easy. There are two different schedulers implemented in the scripts. Two use the ASHA-scheduler (Asynchronous Successive Halving Algorithm). This scheduler aggressively stops bad performing trials early. It can be used to test different hyperparameter combinations as well as architecture configurations, e.g. the number of neurons in dense layers or the number of channels in feature extraction layers. In addition ASHA works in conjunction with a search-algorithm for HPO (hyperparameter optimisation) called Hyperopt in the script. Standard ray would use random points on the grid to probe the loss-function, but this is very inefficient for large search spaces. Bayesian optimisation algorithms like Hyperopt in contrast take the information of prior runs into consideration when choosing new configurations. Though they will also have to start with some random configs to probe the loss function. Why use these algorithms? Well, the loss function is not analytically known to us and so we have to use some form of black-box optimization. Bayesian methods work by constructing a prior or surrogate function and choosing configurations according to this function. After some trials have been run, this function will be updated and new points will be chosen according to the new function and from this point onward repeat this process. There are many different algorithms that do this, they differ in how they construct their surrogate and update them. One can check ray for all algorithms that work with the framework. Since BO works by taking into account previous trials this limits the number of trials that should be run concurrently. A value of 4-6 concurrent trials is fine (This of course also depends on how many trials one wishes to run).

The other scripts implement the population based bandits algorithm. These population based algorithm can be regarded as applying evolutionary principles, but it can only be used to tune hyperparameters. A number of trials with identical architectures will be started and after a number of iterations the performance of all members in the population will be compared. The worst performing members will update their hyperparameters. In 'naive' implementations of population based algorithm this will be a random permutation of the best performers, but here there is a more sophisticated model at work for choosing new configuration, which generally outperforms the 'naive' model with fewer trials run being necessary. I would recommend setting the number of trials to the maximum number that can run concurrently.

There are two variants of each script. One shows how to load the data for each trial individually. This is useful for small datasets or cases, where you don't load the whole data into memory, e.g. when working with large pictures and only the pictures of the current batch will be loaded. These are denoted by `RayTune*_trial.py`. The scripts denoted with `RayTune*_shared.py` demonstrate how a large dataset can be loaded into the shared memory of ray. Every trial then has access to the data in this shared memory. For many more details on search algorithm, trial schedulers, and parameters of functions like `ray.run` check their documentation. They also link to many research papers in case of the search algorithms and schedulers.