Summary of Steps for TRecNet

Before using TRecNet code:
One will need to ensure they are in an environment that contains all necessary libraries and GPU. Necessary files for setting up an appropriate environment are located in the "image/" directory. One can run the command "`source build.sh TRecNetContainer_tf_2_16_1.sif`" to create the container. One can subsequently run the container by using the command: "`singularity run --nv --bind <directory_with_data> <directory_container_is_in>/TRecNetContainer_tf_2_16_1.sif`". *(We might need an environment setup for slurm submissions too ...).*

Prep Your Data:

Before testing or training a TRecNet model, the data must be formatted properly to be fed into the model. The steps for properly preparing data are listed below. They pretty much only require the use of "MLPrep.py" and its functions, which are described in more detail below. The procedure is relatively similar for testing and training datasets, with differences noted in the steps. Examples of running the steps (in the form of bash scripts written so as to be submitted to *slurm*) can be found in "scripts/".

1. **Append jet match tags to your root files (this should only needed is using/training the TRecNet+ttbar+JetPretrain model).**
    - This is done using code "MLPrep.py" and its function "appendJetMatches", on your ntuples (root files).
    - "appendJetMatches_slurm.sh" in the "scripts/" directory is an example of a bash script that can be submitted to *slurm* to run this on all of the data files in a particular directory.
2. **Create H5 files with truth and reco data from the root files.**
    - This is done using code "MLPrep.py" and its function "makeH5File", on your ntuples (root files) from step one.
    - "makeH5Files_slurm.sh" in the "scripts/" directory is an example of a bash script that can be submitted to *slurm* to run this on all of the data files in a particular directory, and it makes the nominal, up systematics, and down systematics H5 files.
3. **Make a text file that lists the H5 files you'd ultimately like to use.**
    - Make separate text files for nominal, up systematics, and down systematics.
    - In each text file, make sure the file names include the full path, and each file name is on a separate line.
    - "ML_6j_file_list_08_01_22.txt" in the "scripts/" directory is an example of a text file listing all nominal H5 files I would want to use.
4. **Split combine the H5 files and split into training and test data.**

- This is done using code "MLPrep.py" and its function "makeTrainTestH5Files", on the text list of files you just created.
- If you are simply running the data through an already-trained model and don't need any training data, you can either use 0 for the "--split" argument, or you can instead use the function "combineH5Files" instead of "makeTrainTestH5Files" in order to simply combine all files into a test dataset.
- "makeTrainTestFiles_slurm.sh" in the "scripts/" directory is an example of a bash script that can be submitted to *slurm* to create a train and test dataset from some set of nominal H5 files, as well as test datasets for the up and down systematic H5 files.

5. **ONLY IF YOU ARE TRAINING A NEW MODEL, you need to save the max and mean of each observable in your training dataset.**
    - This is done "MLPrep.py" and its function "makeTrainTestH5Files", on the train dataset you just created.
    - "saveMaxMean_slurm.sh" in the "scripts/" directory is an example of a bash script that can be submitted to *slurm* to save the max and mean of the observables in a training dataset.

MLPrep.py:

A python file that has multiple functions, each with their own set of arguments, that are meant to prepare root data files for TRecNet (both training and testing).

Functions:

- "appendJetMatches": Creates a new root file (output) that includes the nominal tree, up systematics tree, and down systematics tree from the input root file. Before saving the nominal tree to the new file, events with "bad" pseudo rapidity values (eta<-100) or non-semileptonic events (if any) are removed and a new binary value for each of the jets, called '`jet_isTruth`', is saved to the reco tree. This value is '`1`' if the jet is likely to have come from a ttbar decay product, and '`0`' otherwise (see description of the algorithm in this thesis). Information about the matches that were made are also saved in a sub-directory (called "matching_info/") of the save directory.
    - "--input": Input file (including path)
    - "--save_dir": Path for directory where file will be saved.
    - "--dR_cut": Maximum dR for the cut on dR (default: 1.0).
    - "--allow_double_matching": Use this flag to allow double matching (i.e. allow two or more decay products to be matched to the same jet).

- "makeH5File": Creates and saves h5 file with the reco and truth variables. One must decide on the number of jets to include per event (default: 6) and the cut to make on met (default: 20). The output file name will automatically include the number of jets

included, the met cut that was made, and ‘_sysUP’ or ‘_sysDOWN’ if the tree was up or down systematics, respectively.
    - “--input”: Input file (including path).
    - “--output”: Output file (including path). Note: Tags will be appended to the file name (e.g. _6jets if you include 6 jets).
    - “--tree_name”: Name of the tree from which to extract the data.
    - “--jn”: Number of jets to include per event (using padding if necessary) (default: 6). *Note that you can change the number of jets you wish to train/test on later, so it’s better in some sense to overestimate the number you think you’ll want.
    - “--met_cut”: Minimum cut value for the ‘met_met’ (default: 20).

- “combineH5Files”: Combines several h5 files into one h5 file.
    - “--file_list”: Text file containing list of input files (including path). Each input file must be listed on its own line.
    - “--output”: Output file (including path).

- “makeTrainTestH5Files”: Combines several h5 files. The first portion of events in each file go into a ‘train’ file, while the remaining events go into a ‘test’ file, where the divide is determined by the split argument. For example, if the split argument is 0.8, then the first 80% of events in each h5 file will go into the combined train file
    - “--file_list”: Text file containing list of input files (including path). Each input file must be listed on its own line.
    - “--output”: Output file (including path).
    - “--split”: Percentage of events (expressed as a decimal number) to include in training file (default: 0.75).

- “saveMaxMean”: Saves two dictionaries of [max, mean] values, one for X (reco) and one for Y (truth) variables
    - “--input”: Input file (including path).
    - “--save_dir”: Path for directory where file will be saved.

Training a TRecNet Model:

Training a TRecNet model is simple, and the steps are listed below. They only require the use of the python code "training.py", where you must specify a (pre-defined) model name and a configuration file. There is also options for hypertuning, but these are perhaps less well flushed out (since hypertuning didn't seem to improve results anyways). The arguments are defined in slightly more detail below, as are the elements of the training configuration file.

Each trained model is assigned a unique id (based on date and time of training), and the information for each model is saved in a directory called "<model_name>/<model_id>/". This information includes the model (keras file) itself, a numpy file full of the training history, and a text file full of information about the model, the data it was trained on, and a few other specifications.

1. **Create a training configuration (json) file.**
    - "training_config.json" in the "config/" directory is an example of a configuration file for training. Its elements are also described in more detail below.
2. **Train your network!**
    - Use "training.py", providing the model name and configuration file as arguments (e.g. type in the shell "`python training.py --model_name TRecNet+ttbar --config_file training_config.json`").

training.py:

A python file that defines classes and functions relevant for training and hypertuning neural networks.

Arguments:

- "--model_name": Name of the model to be trained. Must be set to '`TRecNet`', '`TRecNet+ttbar`', '`JetPretrainer`', '`TRecNet+ttbar+JetPretrain`', or '`TRecNet+ttbar+JetPretrainUnfrozen`'.
- "--config_file": File (including path) with training (or hypertuning) specifications.
- "--hypertune": Use this flag to hypertune your model.
- "--tuner": Choice of tuner ('`Hyperband`' or '`BayesianOptimization`'). ONLY include this if you've used the hypertune flag.

training_config.json

A json file that contains all the necessary configuration information for the training you'd like to do.

Items:

- "data": Name of the h5 training file (including path) that you'd like to use.

- "xmaxmean": Name of the X_maxmean file that was produced from this training data.
- "ymaxmean": Name of the Y_maxmean file that was produced from this training data.
- "split": An array defined as follows [% of total data for training, % of total data for validation, % of total data for testing]. Note: be careful that the % for training and % for validating should add up to the % of the total data you had in the training h5 file, while the % for testing should be equal to the % of total data you had in the testing h5 file.
- "jet_pretrain": Name (including path) of the jet pretrain model (keras) file. This may be set to "null" if not needed.
- "frozen_model": Name (including path) of the frozen model (keras) file. This may be set to "null" if not needed.
- "n_jets": Number of jets to train on.
- "max_epochs": Maximum number of epochs to train on.
- "patience": Patience (maximum number of epochs with no improvements) to train on.
- "training": Hyperparameters meant for when you're training a model.
    - "initial_lr": Initial learning rate.
    - "final_lr_div": Number to divide the initial learning rate by to get the final learning rate.
    - "lr_power": Exponent for the polynomial decay learning rate.
    - "lr_decay_step": Learning rate decay step.
    - "batch_size": Batch size.
- "hypertuning": Hyperparameters meant for when you're hypertuning a model. These are the same as what we have for training, but they have sub-attributes that determine how we're going to iterate over different options for the hyperparameters in the hypertuning process.

IMPORTANT NOTES:

- IF you're training '`TRecNet+ttbar+JetPretrain`', you MUST include a '`jet_pretrain`' model in your configuration file.
- IF you're training '`TRecNet+ttbar+JetPretrainUnfrozen`', you MUST include a '`frozen_model`' in your configuration file.
- For both of these, the '`n_jets`' specified in your configuration model must be the same as the number of jets the '`jet_pretrain`' or '`frozen_model`' was trained on.

If you don't satisfy these requirements, the code should tell you so and cease training.

For VSCode users:

One can start a *Tensorboard* session to watch the progress of the training:

1. Install the *Tensorboard* extension on your machine.
2. Open the command palette (Ctrl/Cmd+Shift+P).
3. Search for the command "`Python: Launch Tensorboard`" and press enter.
4. Select the folder where the desired *Tensorboard* logs are located. The current code should place these in a directory named by the unique model ID within "tensorboard_logs/fit/".

For long training sessions:

Most of the time, your training will take longer than you're willing to sit waiting with the terminal open. In these cases, it is best to run the code within a *tmux* or *screen* session, such that it will continue to run even when you log off the server.

For *tmux*:

1. Create a *tmux* session (called "session_name" here), using the command "`tmux new -s session_name`" OR attach to an existing session using "`tmux attach-session -t session_name`" (one can use "`tmux ls`" to get a list of currently running sessions).
2. Run training as per usual.
3. To detach from a session, hit "`ctrl+b`", release, and then hit "`d`".

For *screen*:

1. Create a screen session (called "session_name" here), using the command "`screen -S session_name`" OR attach to an existing session using "`screen -r session_name`" (one can use "`screen -ls`" to get a list of currently running sessions).
2. Run training as per usual.
3. To detach from a session, hit "`ctrl+A+D`".

Validating a TRecNet Model:

Sometimes you may also want to quickly check your model by looking at the predicted vs. true observables on the validation dataset. This can easily be done using the python code "validation.py" with a few arguments (much like in training).

1. **Validate your network!**
    - Use "validation.py", providing the necessary arguments (see below).

validation.py:

A python file that defines classes and functions relevant for validating neural networks. Running this code will save a set of plots to a directory called "model_name/model_id/val_plots/". The plots in the "scaled/" subdirectory are the observables the network sees, while the plots in the "original/" subdirectory are the observables in their original form.

Arguments:

- "--model_name": Name of the model to be trained. Must be set to '`TRecNet`', '`TRecNet+ttbar`', '`JetPretrainer`', '`TRecNet+ttbar+JetPretrain`', or '`TRecNet+ttbar+JetPretrainUnfrozen`'.
- "--data": Path and file name that was used for training the data.
- "--xmaxmean": Path and file name for the X_maxmean file to be used.
- "--ymaxmean": Path and file name for the Y_maxmean file to be used.
- "--model_id": Unique save name for the trained model.

***IMPORTANT NOTE:** Currently the split is hard coded as 70% to training, 15% to validation. *(I think I might change this to take the training config file as well.)*

***IMPORTANT NOTE:** Be careful to use the SAME maxmean files that the model was trained on. This should be listed in the information text file for the model. *(Admittedly, this should probably be coded in somewhere to prevent mistakes, but it has not been yet).*

Testing/Using a TRecNet Model:

Testing a TRecNet model is also fairly simple! There is one step, which uses the python code "testing.py" with a few arguments (much like in training).

At this time, if your "--data_type" argument is set to 'nominal', your results will include both truth data and prediction data. If your "--data_type" argument is set to anything else, your results will just include the prediction data. *(This could probably be slightly and easily edited such that there is a flag if your testing vs just inferring, and then "--data_type" will only be used for file-naming purposes).*

If you'd like to use an already trained model, there are a number available in the TRecNet repository. They are organized as described in the previous section, and each model has a text file full of any information you may want. *(I can't currently remember which versions ended up getting used in the thesis linked above, but I have it written down somewhere...)*

1. **Test your network!**
    - Use "testing.py", providing the necessary arguments (see below).

testing.py:

A python file that defines classes and functions relevant for testing neural networks. Results are saved to a root file in the desired save directory.

Arguments:

- "--model_name": Name of the model to be trained. Must be set to '`TRecNet`', '`TRecNet+ttbar`', '`JetPretrainer`', '`TRecNet+ttbar+JetPretrain`', or '`TRecNet+ttbar+JetPretrainUnfrozen`'.
- "--data": Path and file name for the testing data to be used.
- "--xmaxmean": Path and file name for the X_maxmean file to be used.
- "--ymaxmean": Path and file name for the Y_maxmean file to be used.
- "--model_id": Unique save name for the trained model.
- "--data_type": Type of data. Must be set to '`nominal`', '`sysUP`', '`sysDOWN`', or '`ttbb`'.
- "--save_loc": Directory in which to save the results.

***IMPORTANT NOTE:** Be careful to use the SAME maxmean files that the model was trained on. This should be listed in the information text file for the model. *(Admittedly, this should probably be coded in somewhere to prevent mistakes, but it has not been yet).*