Log Files Ariel

Jun Wei Chow & Rachel Cheshire

April 2021

Chapter 1

Explanations

1.1 Data Transformation

Inside the utils file there are 2 data transformation type: the original baseline transformation and a yeo-johnson transformation. Ultimately the yeo-johnson transformation adds an extra transformation layer on top of base transformation by default unless passed in a different dataframe (currently requires state from_baseline = False as well, which is redundant, and will change in the future).

The baseline transformation all follows what is being done by the Baseline model from Ariel website, with one additional feature change: that is passing in modification to the original data before transformation, such as squaring the values, take the log of the values or square root of the values, are the ones currently supported. However based on visualizations this will lead to larger fluctuations (noise) where there is no dip and in comparison to that, the noise in the "dip" (due to transition) will be reduced. However, we could not say whether the training will be better or worse without actual experimentation, and the function is there to allow for experimentation in the future.

Yeo johnson transformation ultimately tries to center a non-gaussian to a gaussian. How that works is not known to me. However, based on what is searched online (very barely touching the surface of what yeo-johnson does) is to center the dataset better and perhaps do something to the standard deviation. That is, if the data mean is near 0 but not exactly 0, Yeo-johnson will move it to zero. Again, whether this will improve training or not is unsure until experimented.

1.2 Different models

Either we could do a "multi" time series consisting of a single model taking in a chunk of 300x55 data and output 55 outputs, or we could do multiple "single" time series models where we have 55 different models for each of the "55" in the data, and then each output "1" single logits.

1.3 Food for thoughts: ML model mistakes made

In the beginning the model was thought to be a "time series", be it a "single" time-series or "multi" time series. **We are not predicting what comes next, but more like a "classification" model** on what are the resulting label given a set of features. Albeit we have 300 different "features" that are part of a time series, and we have a "label" that is a float instead of binary. Perhaps more like taking in 300 different features and predict a single value from it. The underlying meaning of how this works is unknown.

1.4 Information on using GCP

- `https://cloud.google.com/bigquery-ml/docs/making-predictions-with-imported-te`
- `https://cloud.google.com/datalab/docs`
- `https://www.tensorflow.org/io/tutorials/bigquery`

1.5 Data flow speed using gcsfuse

By mounting the google cloud bucket using gcsfuse, it is noticed that performance was very bad, and things done on the data could only be at approximately 1 item per seconds. However with loaded onto local disk, the bottleneck would be CPU instead of data retrieval.

Solution:

- Load whole data onto disk locally.
- Install `pip install google-cloud-storage` to use it's API, which *might or might not be* faster, unexperimented.

1.6 Get Started with XGBoost Model

As XGBoost also can only predict one column (at least I am not aware of multiple output channels, although there's a chance it might exist), hence it will use the BigQuery method of splitting the data into 55 different wavelengths that method.

- For each wavelength file, we will need to split the data into training and validation set. This is required for hparam tuning the model with Weights and Biases (wandb). The splitting can be done via sklearn's train_test_split.
- Perhaps instead of using the Ariel metric, we could use root mean squared error here and try to minimize it. Also, since root mean squared error are usually of the order of micrometer, we could boost it up by multiplying its value by 10^5 or 10^6, which will be decided when the result is out, which one looks better (without an extra 0 at the back).
- We will build 55 XGBoost model. The model type and others requires tuning as currently the dataset used in learning how to use it is quite small (less than 1000 values, which aren't ideal for XGBoost). And there are other things such as boosting method, etc.
- Whether or not to include extra params is another question to ask.

1.7 Two ensemble models to try

First: Stacking. Split into features, we'll use 50 as more, it might not learn well, and less, it might take too long. For subsequent models stacked the training error is passed onwards as a training variable as well.

Second: k-fold training, implemented in Kaggle. Since this already have implementation, albeit originally used for classification problem, might have use in our regression problem here as well.

Chapter 2

Logs

2.1 30 June 2021

- One thing to note: **Make sure you call the correct model.parameters() in your optimizer**. There was a mistake made where the transfer learning model is only used for feature extraction, but the optimizer weights are based on it when it doesn't involved in learning anything. This is one of the most awkward error ever made.

  Thing with if you made the error, you "fail silently". That is, you actually don't do very bad, still having 6500-8500 points (better than truly naïve models which claims around 4000 points), but you don't do any better after all. Not to say that using the correct optimizer might let you do better.

2.2 29 June 2021

- **Important** Tries to change transfer learning to using feature extraction. It goes into an error "RuntimeError: mat1 dim 1 must match mat2 dim 0" **during evaluation/validation only** (that is, it trains fine without any non-matching dimension, but it cannot predict). I am guessing it might be because there is not enough data for prediction, but this does not sounds right because we have 12560 data and it is ensured that 10000 for training and 2000 for validation, batch size 25. Trying to reduce this training to 6000 and see what happens. If it still fails then it might be a ghost error that is unable to understand.

The exact traceback is included below. The exact code is uploaded on branch "extract_learning" on github.

```
Traceback (most recent call last):
  File "transfer_learning.py", line 340, in <module>
    main(args)
  File "transfer_learning.py", line 316, in main
    val_scores, model = train_model_feature(model, criterion,
        ↪ loader_train, loader_val, device, num_epochs=30)
  File "transfer_learning.py", line 118, in train_model_feature
    pred = baseline(features.double())
  File "/mnt/batch/tasks/shared/LS_root/jobs/mlwork/azureml/
      ↪ azureml-train_1624977165_c2965db9/wd/azureml/
      ↪ azureml-train_1624977165_c2965db9/utils.py", line 278,
      ↪  in __call__
    out = self.network(out)
  File "/azureml-envs/pytorch-1.7/lib/python3.7/site-packages
      ↪ /torch/nn/modules/module.py", line 727, in _call_impl
    result = self.forward(*input, **kwargs)
  File "/azureml-envs/pytorch-1.7/lib/python3.7/site-packages
      ↪ /torch/nn/modules/container.py", line 117, in forward
    input = module(input)
  File "/azureml-envs/pytorch-1.7/lib/python3.7/site-packages
      ↪ /torch/nn/modules/module.py", line 727, in _call_impl
    result = self.forward(*input, **kwargs)
  File "/azureml-envs/pytorch-1.7/lib/python3.7/site-packages
      ↪ /torch/nn/modules/linear.py", line 93, in forward
    return F.linear(input, self.weight, self.bias)
  File "/azureml-envs/pytorch-1.7/lib/python3.7/site-packages
      ↪ /torch/nn/functional.py", line 1690, in linear
    ret = torch.addmm(bias, input, weight.t())
RuntimeError: mat1 dim 1 must match mat2 dim 0
```

- It is figured out thereafter. It is due to the size being passed in. Because for training we used RandomResizedCrop with an integer (here 224) pass in, it will make a $224x224$ image. However, with Resize, it does not, it will retain the ratio of the image, hence the output is different. The walkover to make it work is to use (224, 224) instead of just passing

in integer to force it square, or we could use RandomResizedCrop in validation as well (why not? People aren't using it perhaps it might be bad, but we could try and see, it might give surprises).

- We are indeed overfitting the validation set. This gives us a good chance to revisit our hypothesis of not overfitting stuffs. Realizes the **reason is because this one does not have shuffle after choosing, so train set leaked into test set**. This is really annoying.
- Transfer learning is successful. Perhaps because I lower the learning rate too much, it is underfitting. Trying to increase learning rate. And trying other types of optimizer such as SGD.
- The first successful train was using SGD with learning rate of 1e-4. Also, this transfer learning was fine tuning all the way (just because we forgot to freeze the layers).
- Second tuning will change SGD with lr = 1e-3, add gradient clipping. This will be learning the last layer for half of the total epochs, and fine tuning thereafter. We will also try Adam with same learning rate, and set `amsgrad` to True.
- Efficient Net isn't suitable for transfer learning in this field.

2.3 28 June 2021

- This will be a list of stuffs discovered during sending job to Azure machine learning compute clusters. These are the failed lessons learnt after 23 Runs, (and still failing).
  - Remember to check what packages to install. Especially if you are using your own conda specifications, you are responsible for all the package dependencies (including pip and python), and check whether it is supported by the GPU running underlying the compute clusters or not. This can be frustrating. The **easiest workaround** would be to use one of the predefined environment, and then use `os.system("pip install ...")` at the **very top of the script before any importing except import os**. This way, we can skip building condas and its docker environment, if using docker.

- Multiprocessing using clusters means by default, even if you are to only use one GPU, but if you set num_workers for dataloader to be larger than 0 (meaning ≥ 1), then it will start failing with "cannot fork process". Online they say that to use `torch.multiprocessing.set_start_method("spawn")`. This have been tested (and still testing) to whether run or not. Perhaps might continue tomorrow or some time later. However, setting to 0 is also working.
- Check that whatever data passing in is in the correct order. Sometimes, it can say "KeyError" and this is very clear that it is either python dict or pandas dataframe error. However, the rest of the explanation following it wasn't clearly explaining especially in the heat of time. Even the final line where it supposes to make clear wasn't that clear (because the "pandas error" is shown at the penultimate line/paragraph so couldn't find it at the final line/-paragraph of error code).
- Because transfer learning, some models are using Float() to train instead of the default Double() that is used by newer pytorch, we would need to convert our data accordingly.
- `pin_memory=True` in Dataloaders does not work with cuda FloatTensor. Only dense CPU tensors can be pinned. Hence, this is especially the case if we are pushing our data to GPU storage vRAM **before** passing it through dataloader, hence, either you put to GPU after passing through dataloader, or don't pin memory.
- It so happens that with torch.multiprocessing and num_workers > 0 we get

  > Producer process has been terminated before all shared CUDA tensors released. See Note [Sharing CUDA tensors]

  error.
- What wasn't frustrating is before the above failed, it managed to run in 25 seconds with a P4 GPU (for 200 mini batches. In CPU, we barely managed to finish 2 mini batches in the same amount of time). With no num_workers set, we can train in barely above 1 minute per epoch with P4 GPU.

  - It is also calculated, training on a GPU (which cost (much) more per hour than pure cpu instances) than training on a CPU is (much) more **cost effective**, because GPU training is so fast that you shut it down before the price exceeds that of using CPU. This is only true for fast GPU, not slow GPUs. Example, on a V100 training something for 2.5 minutes per epoch is equivalent as training for 7.5 minutes per epoch on single T4, and more than 15 minutes (or 30 can't remember) on **half** a K80 GPU (half as in half of its memory, but its compute power is **still using whole**).
  - Wondering how much cost have been used since running for 24 runs now. Each run, the GPU is reserved for more than 5 minutes, and they aggregates. Although, certainly not exceeding 1 dollar.
  - The compute clusters on Azure ML Studio are very cost effective. The importance is to set min_nodes = 0 which will deallocate (and deprovision) all your resources. This not only frees up paying for the resources after finishing (which it's important to set the idle time before deallocation as the default is 30 minutes which is too long unless during testing phase where you don't want to get allocated and deallocated too frequently), but also frees up your quota as well. However, whether you have to pay for the price of "reserving" the compute cluster or not is not researched on, and is worth researching although theoretically and through my memory this perhaps does not need payment when not in use. They are also useful as choosing "low priority" could save a lot of cost. Of course, try not to choose the V100 GPU as it is (almost always) in use and you might never get to start training. Choose some second tier P100 or even third tier T4 GPU.

- We go "deep" by building a model based on multiple linear layers, and manage to overfit the training set (ie, loss = 0.0). The thing is, while overfitting the training set, we are also overfitting the validation set. Despite all in all, if our validation set is correct, then we are achieving 9930+ points. That is, with a little intense fluctuations which have major drop in a single epoch but move back to getting better again thereafter, validation set still keeps improving. It's unsure whether the model is actually overfitting the validation set. All else being equal requires sending it on the holdout test set to see how well it works.

What we actually done for the validation set is not just randomly picking a sample but to reflect what is in the test set, to hold out the AAAA's that are not included in training. There are some more thoughts to be considered, however:

  - The AAAA's holdout on the test set are actually stars chosen at random. That is, they range through from $AAAA = 5$ up to $AAAA = 1000+$. However, for ours, we are just **choosing the last few AAAA's as the validation, and the first few AAAA's as training**, so this does not perfectly reflects the test set.

- What should have been done next time is to hold out a test set for ourselves apart from the validation set, where we could test it for ourselves as well rather than sending it to say Kaggle to be displayed in the leaderboard, if requires to keep private. If we just want to casually play around, we can just use specifically-selected validation set and would work fine.

2.4 27 June 2021

- Make changes based on `https://www.kaggle.com/ronaldokun/multilabel-stratificatio` and `https://www.kaggle.com/veb101/transfer-learning-using-efficientnet-models` and `https://pytorch.org/tutorials/beginner/finetuning_torchvision_models_tutorial.html` and `https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html`

2.5 26 June 2021

- Recently learned about not randomly choosing validation set but selectively choosing. For example, our test data is using data that is not available in train, so our val dataset should hold out some data that is not available for train to see as well. Hence, we will choose the dataset first before shuffling.

- One problem that is noticed with before is dataset loader for train and validation are called separately. And the data is shuffled before

assigning index. This means there is a chance (in fact, very high chance) that training data leaks into validation data. We fixed this by **selecting data first before shuffling within selected data**.

- To use Min Max Scaler, we would want our data to be in NumPy, so the translate to Tensor will only occur after the changes. Furthermore, this is not tested if it will fail with Tensors.

- Since our transfer learning layer requires the input to be $224x224$, we will have to combine some of our data or else $55x300$ cannot be reshape. This can be done by combining all same group's CC together (i.e. with fixed AAAA and BB, all CCs are stacked on top of one another, so the final size becomes $550x300$). We can then perform `torchvision.transforms.RandomResizedCrop(size)` to the data. Image augmentation is using random flip such as `transforms.RandomHorizontalFlip(` We would not be using `transforms.Normalize()` due to difficulty understanding how the number of the array feed into it works (unless we study about it further on, which was limited by time), and normalize beforehand using MinMaxScaler, as stated in the previous bullet point.

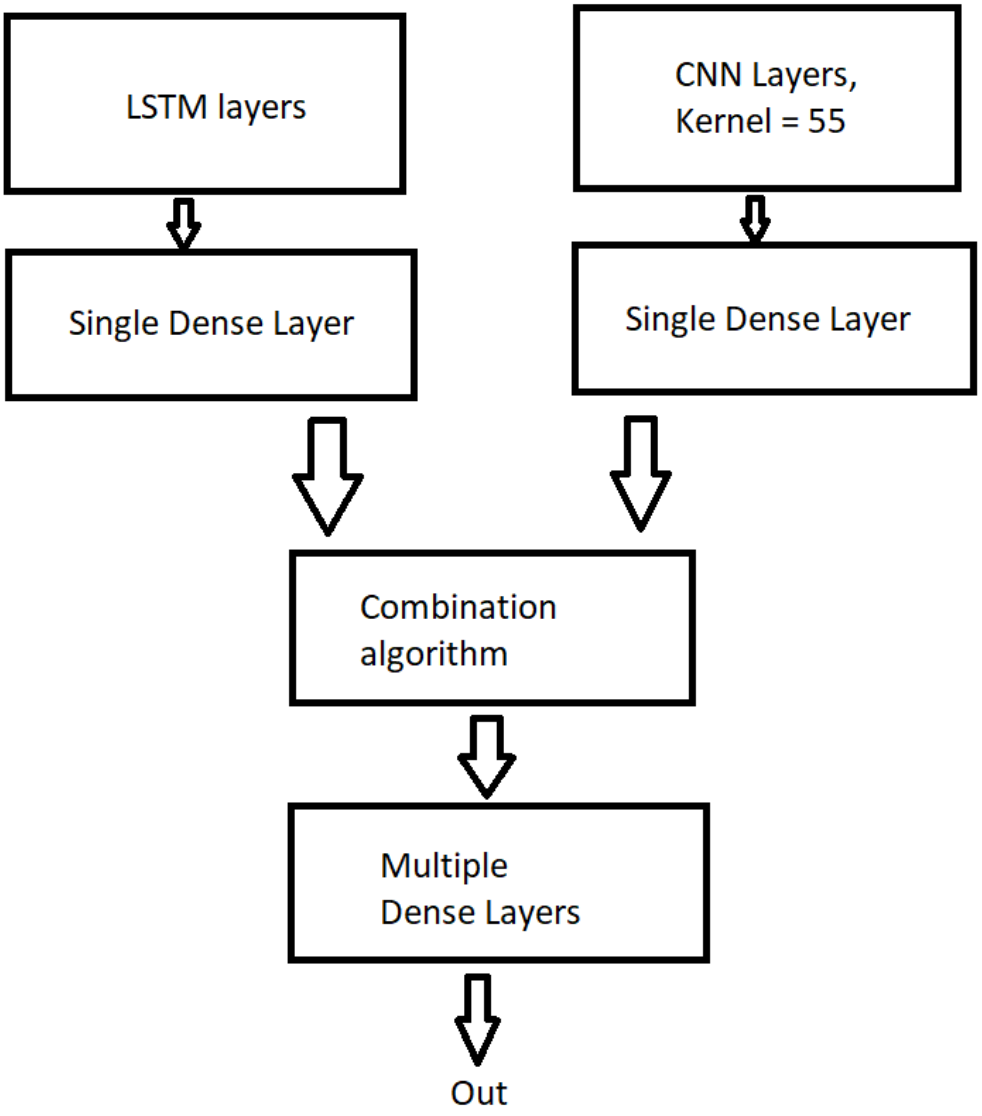
- **Uploading today is from folder 27062021.** This is the LSTM-CNN multi-headed stacking model with plus instead of concat at the three way point (middle of model where outputs from each head joins).

- Perhaps I have make a mistake. Perhaps setting num_workers instead of CPU for training might speed up training the LSTM. Perhaps it is the data that is the bottleneck. Trying to set both but raises Segmentation Error.

2.6 23 June 2021

- Upload a half-trained model of LSTM (still continuing training but predicting from best checkpoint) just to not waste "quota" hahahha!

- Created a new model BaselineLSTMAlt which is a semi-multi-head model, which combines into one model at the end. This will go into 27062021 folder in gcp bucket.



Currently for the combination algorithm we're using + (mathematical symbol plus). However, we are going to try how this work. If this does not work well, we might switch to using `torch.cat()` and experiment. Batch size is 50 here. Also $H1 = 256, H2 = 512, \text{hidden\_layer} = 128$ where hidden_layer is used both in LSTM and CNN while H1 and H2 are used in Dense layers. Training on Intel Xeon CPU 2.30GHz CPU family 6 Model 63 (Haswell) 12vCPU with 32GB RAM will take 1.9

hours per epoch. This is about the same time if batch size is increase to 100 as first, we are not prefetching data (because I couldn't get it to work, requires more investigation into the error), and second CPU is one such bottleneck when passing through LSTM layer (and not when passing through CNN layer) so this is a very spiky training, meaning it is not fully utilizing CPU.

- The concat method have been implemented in instance-2 of our shared project. This is to parallelize training speed. An extra thing that is implemented in this concat method that is not implemented in the + method is omitting the first 50 timesteps. Garbage in, garbage out, so unneeded garbage can be thrown away to achieve higher score.

- Note to do prediction for LSTM requires the more CPU the better. With 16 vCPU around 1 hour can complete while with 10 vCPU requires more than 1 hour. DataLoader is also set with num_worker = 2 however the bottleneck here is number of CPU as CPU is running always above 80% (on average, with extreme spikes sometimes).

2.7 22 June 2021

- Usage of num_workers in dataloader during train means that we are training faster than ever on Dense Layers. This means prefetching data and training significantly improves. However, this is only True if CPU is not the bottleneck (as in LSTM model) but IO is (which is the case for Linear layer only model).

2.8 21 June 2021

- The upload today is **on the folder 22nd June**. Due to unexpected circumstances we upload those later than today earlier. And the ones due on 20th haven't finish training until now and will be uploaded next. Do **note that due to time constraints, this is an underfitting model**. This means that scores could still improves if train for longer.

- It is noted that training on the whole dataset, with or without dropout, for LSTM model, the first 2 epochs did not do well, after which it escalates quickly to good marks.

- Realized prediction originally did not use multi workers to prefetch data. Changed this to half the number of CPUs (in this case, 4 num_workers), reduce the batch size to a quarter of original (1000 to 250). This speed up predictions from 33 mins to 3.5 mins (for one of the case).

2.9 20 June 2021

- Using LSTM to train with 2 layers and 512 nodes for each, plus a dropout layer of 0.1, is good. However, it easily escalates memory and soon OOM error arises and state dict is not saved so retraining is required. What is noted to say is whether it is worth the time training an LSTM layer when a "good enough" accuracy can be achieved in a Dense layer, in real life. The balance between how much time one have and how much improvement does it gives is a difficult choice, though 80:20 rule seems more satisfying. Back to the main story, LSTM, depending on how many CPU you have, will take a decent amount of time to train per epoch. While with 6-8 CPUs (threaded core) on the previous Dense layers we have take 30 minutes per epoch (up to 40 minutes on a bad day) (and on the E2 compute instance if you got AMD processor it trains slower as well so shutting it down and restart to get an Intel newer family, might help increase training speed as well), LSTM takes 16-20 CPUs (threaded core) a decent 4-5 hours to train, per epoch (and 5.5 hours on 24 threaded AMD cores, perhaps due to it being a lower strength CPUs or older family). Real reason have to investigate again to determine.

- Yet to try on a K80 GPU and see if it speeds up.

- The original function written for prediction by Baseline was problematic especially with non Dense layers models. It seems to scale up indefinitely (LSTM, it exceeds 100GB of RAM) just after about 25 batches (batch size 100). Rewriting function.

2.10 17 June 2021

- Nope. XGBoost failed. It did very bad. Far worse than any of the model I had seen before. Perhaps this is because we did not have a more stronger tree, but overall this is a problem as well.

- Gonna try one more time by increasing depth of tree and let it train longer. Increase max_depth from 15 to 50 (we bloat up ourselves a lot lol), and n_estimators from either 25 or 50 previously to 50.

- New discovery! Before thought to be very bad now is still very bad if compared to one of the other files (by calculating ariel score). However, when checking the graph it isn't that bad at all, so perhaps there might be problem with the scoring method? Or perhaps the graph seems like it shifted up a bit? Anyway, this could still be used in prediction and upload and we will see how the score tells and confirm that XGBoost hypothesis is bad is valid (or disprove it to be invalid).

- It is worth noting that after we increase the max depth to 50, we smoothen the graph by **a lot** (from spiky). We are going to try model 3 with even higher (and hence more difficult to train) parameters and see if it fits better or worse with our current highest score prediction. Parameters changed includes max_depth = 75 and n_estimators = 150.

- After some data describing I noticed the difference for making its score so low. The minimum it can go is not as low as what is gone to by baseline. In fact, the min values for each rows are about 5-10x larger than the ones from our current best prediction. This may be the cause of why the points are low in the 4000s era. Of course, perhaps real prediction might be slightly different but it does not seems likely this XGBoost model will do well if it cannot go low. In technical term, values below the 25th percentile (towards the left) are capped at the 25th percentile.

- Noticed that colsample_bytree is really low, perhaps this is the problem. Am going to increase both colsample_bytree and subsample to their default value, which is using the whole datasets and all features.

2.11 16 June 2021

- Started XGBoost optimization task yesterday. Only doing 20 steps (I specified 15 steps but it goes to 20 which was weird, but anyways) for optimization (which wasn't really getting to the best of what was wanted for some wavelengths as the mean squared error is still a little too high) is already very little, and yet it takes about 1 whole day to finish the optimization on 8 hyperthreading cores (with 2 min bottleneck in between to retrieve data from bigquery, which is also not efficient). Anyway, it's finally done and we have the hyperparameters.

- Starting second round hyperparameter tuning. This time, we are searching much less parameters and I expect it to be simpler. For each we are only doing 4 instead of 20 steps. I expect it to be faster but still overnight. This second level is required to ensure that the parameters tuned are best and not doing worse than default and the best parameters found throughout the whole lifecycle of tuning (from other wavelengths as well). In this sense, we are not using a single set of hyperparameters for all models but tuned to localize to each wavelengths.

- After this round is tuned, we will start XGBoost our way through.

- After all, XGBoost model takes so fast to train compared to deep neural net, with the same amount of data compared to one single epoch, although with smaller dataset size deep neural network can also train quite fast.

- Also for ensembling Dense models as previously thought, perhaps I should have record down the points from the Ariel website which could be used as weights, but I did not, so now we cannot use weighting to do the averaging method. I could only take the highest (those approximate to be above 9600 points) as ensemble and discard the others weaker ones. Ultimately with weights we could have do with all with heavy penalization for weak models in favor of strong models but yeah...

- However the ensembling method seen from graph seems to be fine. They are located on the branch on github called **average**. **Please do not delete this branch "average"**.

2.12 14 June 2021

- Model 10.5 is trained for 16 epoch, saved every 5 epoch, and took the best score model. Training seems to plateau after about 11 epoch.
- Perhaps could start trying XGBoost model.
- Getting idea of XGBoost model, perhaps we could do something similar to Neural network, by getting one more pipeline after those model predictions. That is, to merge those model prediction as one by getting their averages of their outputs. We could easily do this by reloading the values from older training session as numpy and perform element wise sum together and perform broadcasted division by the number of models used.

2.13 13 June 2021

- Model 12 is logical sweep 26. Model 13 is honest sweep 67. Their respective hyperparameters code is being stored in google cloud storage.

2.14 12 June 2021

- Used Weights and Biases (wandb) to tune hyperparameters. What was tune: (check gs://arielml_data/trained_model/14062021 for the yaml file).
  - All layers including dropouts. Their specific parameters passed to tuning will be defined later.
  - Batch size.
  - Learning Rate.
  - Standard Deviation.
- What wasn't tune (yet) (due to too many parameters and might tune later on after obtaining few peaks on current tuning to restrict parameters):
  - Mean (using 1.0 for now, but could restrict between 0.0 and 1.0).

  - Train size: (using 2000 as training size. This could be tune say, between 500 and 10000, but not going further up as it will take too long).

- What is difficult if not impossible to tune:
  - Number of layers: let the tuner immediately creating layers is something dreamt up but not possible by the hparam tuner itself. It requires writing our own program to create the model on par.
  - Tuning for different types of layers such as LSTM, Convolutional, etc. This is also same problem as above.
  - Trying out different models including classical ML (non Deep Learning NN) models: requires using if else statement as well to switch between.
  - What is to be passed into the model: i.e. the input. Tune how many parameters/features are passed into the model, randomly select columns and how many columns to be passed into as training, including but not necessarily including extra parameters. This requires tuner flexibly change the input dim, and also requires an if else statement on how to use the simple transform, how to call for data since they need to perform transformation as well, what to include what not to include is also another question, how to write it is also another question.

- **Model 10.5** is based on fresh-sweep-13. **Model 11** is based on light-sweep-20. Model 10.5 is because the prefix forgotten to change and refuse to restart both model so only one get restarted and using non-integer labelling.

- Turns out tuning train size wasn't a good idea. It shall be fixed at 2000, and continue to tune for mean.

2.15 11 June 2021

- Looks like CNN does deprove training instead of improvements. Hence, refrain from using CNNs unless found out a proper tilting (currently they gave around 7000 points for the 2 models trained).

- Looks like the extra params does not improve training, sometimes even deprove training. Hence, refrain from using them if necessary.
- Perhaps try to do feature engineering: taking every 3rd column instead of using every column might help.
- Perhaps try XGBoost model some other time.
- **Job sent for hparams tuning in private project**.
- Hparam tuning failed as gcp have some internal error encountered. Follow this `https://stackoverflow.com/questions/67932401/google-vertex-ai-hyperpar` for more info. Solved by changing the location endpoint from us-central1 to us-east1.
- Trying LSTM as a layer.

2.16 8 June 2021

- Model 7 suddenly jump from around 9000 points to 5000 points in a single epoch after training for about 25 epochs. It is also seems decreasing in model accuracy in model 8 as well.
- We will try not to restrict the max value of extra parameters by not cutting off the fat-tailed distribution in model 9. Model 10 will try only use star temp and not others totally. Both model will use the layer defined for model 8. Note that input dim changes according to how many data we use, in accordance.

2.17 7 June 2021

- Model "no. 7" (this is called the "prefix" variable used in train_baseline.py) uses $H1 = 1024, H2 = 512, H3 = 256$ and model "no. 8" uses $H1 = 256, H2 = 1024, H3 = 256$. However, this time the only change is the data taken includes the extra parameters, where the transformation are being described before.
- Still wondering whether do we need the transformation or not. Training of model no. 7 for 5 epochs seems to stay around 9200 without getting

higher. This are not sure if due to transformation or due to using the extra parameters causing points to decrease.

- Training on CNN is longer than expected. The "deadline" for catching up with submission of model will be missed today as epoch 20 has not yet reached (where the first model save will occur). Perhaps decrease model save to 15 or even 10 might be a better idea especially for super long training (CNN takes 1h 5min per epoch).

2.18 6 June 2021

- We developed a new CNN model consisting of 1 layer of CNN, then to make sure output shape is correct, flatten it and add an extra Linear (Dense) layer at the end of the output.

- After some frustration, there are 2 types. One is with kernel_size = 1, which we will just "unsqueeze" the model in the axis=2 (dimension), then train as usual will work. Another is we have kernel_size = 5 but this means we need to have input_dim = original_input_dim // kernel_size and make sure that input_dim * kernel_size = original_value of our reshaping, or else it wouldn't work, so, very restrictive. Luckily our number are divisible by 5. Also this reshaping means during reshaping we have to hard code the batch size as well as the variable stays in another .py file unless we have ways to import the batch size from the other file, so this is another restriction against using this method (however I haven't try to use -1 to force the dimension not sure if that works, perhaps we could try).

- Yea after tried, it works with -1.

- Also, some data transformation is being done. This is using train_mean and train_std for the simple transformation, and eliminating fat-tail distribution for the extra params with fat-tail, and eliminate star-logg completely.

- And we will use the unsqueezed method (kernel_size = 1) prediction next, and then after the reshaped model (kernel_size = 5). See which one is better. Training last from last time, batch_size = 100, epochs = 40. That is, nothing else changed.

2.19 5 June 2021

- Starting to train the model said yesterday. After seeing that 30 epochs seems not the end of gradient descent, changed to 40, which still don't seems like end of gradient descent. However, in afraid of overfitting, we will stop here to see how well it does first before deciding whether or not to train for more epochs.

- We also got that model that are planned to predict for today is only 7691.0 so not very good model.

- So am I expecting the upcoming model to be worse than the best score we had but better than any other score we had trained on TensorFlow.

2.20 4 June 2021

- Created a model in tensorflow that trains with 30000 columns in a row, excluding the extra parameters, for a single value of the label, since noticing all AAAA for the same wavelength are the same label. Hence, perhaps we could train it this was as well. Still separately as 55 models but now the data is compressed. Unsure how well will the model be doing until tested. This will be on the queue for 6th June.

- For this model, we will minus everything by train_mean before dividing by abs(df).max(). We will use a batch_size of 157 (because 1256 is divisible by 157). We will use the original model developed by Rachel but the second layer is changed from 1064 nodes to 1024 nodes. We will still be training on the entire dataset, no validation and test set split from there. This means optimizer = adam (default), loss = mean squared error (default). We will also have no callbacks. Training will last for 30 epochs for each model. The model.predict part will have half of the defined batch size (using // operator).

- Note that after getting idea from Baseline model, we are not going to apply any transformation to our label/target. They are only applied to features.

2.21 3 June 2021

- Note that the model_state_3.pt is trained for only 20 epochs and same for model_state_2.pt. No improvement have been suggested for further training. Note that the model for 50 epochs originally for model_state_3.pt in the vm had been wiped out by the 20 epochs stored in gcp bucket. However this does not apply for model_state_2.pt.

- Planning to calculate the mean and standard deviation values for training to be used in normalization. Will store this into 2 files (txt or csv to be determined later), one for mean and one for standard deviation. This ease numpy loadtxt.

- Second thing is we don't know exactly what are the top 100 until we collect data about it and count. Hence, this is going to be done as well.

- As tested today, the difference between our model and their model aren't really big difference, merely about 60 points differences for 20 epochs training.

- After making final count of the values and taking statistics of 100 most correlated columns (including both positively- and negatively-correlated since using absolute values) (check main_2 branch of github, create_check.py file), it is realized that all the columns are included in, so all the columns are ranking as the highest 100 correlations in some files.

  Moreover, the least ranking is column 268 with occurrences 38058 and the top being column 167 with occurrences 47122. This is not a big difference, so we can confirm to take all 300 columns for training can be done instead of nit-picking columns.

  Although the training is already in progress with column 99 and 199 inclusive in between. The result, training for 40 epochs, will be uploaded tomorrow.

To do:

- Encode star_logg as categorical values before training. Also ensure data transformation does not affect star_logg (perhaps by setting the values = 1 in the csv file correspondingly after we check it out).

2.22 2 June 2021

- Haven't feel like giving up that the most important characteristics is from 99 to 199 of timestep inclusive, although previously trained model give worse result. Is going to try on this again with the same model uploaded today (model number 3); and this will be called model number 4. Batch size remains as 100. And model epoch changed from 50 to 40. Nothing else changed except for slicing of data.

2.23 1 June 2021

- Training for 3 layers, but uses nodes number as a factor of 16506 (the length of inputs). Factors are found from here.
- Introduction of Dropout in between $H1$ and $H2$, of value 0.1. The values are $H1 = 131 * 9, H2 = 131 * 3$ and $H3 = 110$ which is not a factor of 16506 but is 2×55.
- **Emergency:** Just realized that there are lots of wrongdoings in the simple_trainsform. First, we use abs(out.mean()) but then how ddo we revert back to the original after making prediction? Remember that out.mean() is different for each and every one. Perhaps we might need to calculate a global out.mean() through the entire dataset before setting it inside the simple_transform.
- Second: The normalization for the 6 extra papameters is not correct as they are not already normalized like the matrix values. Hence they requires different normalization techniques. How do we do this? Provided in the original pytorch model we can only normalize one type. So do we abandon it?
- Third: Some value cannot be calculated outside, like the train mean: because they are not normalized to be equal already, so requires to be inside simple_transform. This also means the values rae calculated per-batch. How do we create a more broadcasting method of divide? Perhaps pass in a division as pandas DataFrame? Same for standard deviation division?

- The above have been solved. It could be subtracted and divided as we wanted it to. However we just need to use np.divide for element-wise true division rather than / = which doesn't work element-wise. And make sure the divisor is a numpy array rather than python list or other form as Tensor is currently proven to works with numpy (and not yet experimented for other forms). This means we could compile the statistics, column-wise, and save it into a csv file to do the transformation. This also allows reverse-transform.

- To just test the model, changed back to original value of division by 0.04. Remove the 6 extra features. Train. This is for model comparison only. Also, batch size set to 100.

Problem Solving

We will discuss how to solve this problem using a stupid method that I tried very hard to avoid and perhaps now it's not time to avoid.

Important: Make sure to ignore star_logg since that only have 4 distinct values.

- First, flatten the array into $(55 \times 300 + 6,)$, so we will have around 16000 columns (which is stupid). Then, also include AAAA, BB and CC at the beginning.

- Second, make another table with AAAA, BB, CC and the 55 labels, so 58 columns in total.

- **Caution:** If the previous doesn't work with how BigQuery flow from BigQuery works, then we will combine everything into one single table.

- Load these two tables into two different dataframes. **Only load a subset as memory insufficient to load everything**. This is the caveat, unless flow from bigquery. Also, make sure they are ordered by AAAA, BB and CC so they don't get mixed up.

- Pass them into how Keras to with from_tensor_slices() function. This should work.

- Otherwise, flow from bigquery directly during training. This should also work.

- Another way is to create a csv file containing all the metadata required to make the transform that could be called in the simple transform, element-wise. Then, we can back-transform as well. However, not sure if this can be done with torch.Tensor() although this could be done with pandas DataFrame (which cannot be trained on unless converted to torch.Tensor()).

Advantage:

- Can do preprocessing BEFORE inserting data into BigQuery, making it hard-coded. This saves effort of preprocessing after loading data. Of course, we could also preprocess data AFTER as well. Dunno.

2.24 31 May 2021

- AutoML gives worse result than other methods. This is unsure whether training is too little or other reason. Most importantly, we are also mixing all 55 wavelengths together although create a column to distinguish it but it is not very predictive as detected by the model.

  Most importantly, it is not even known whether the prediction gives correct result as it returns 3 rows: value, start and end. So this is a value with uncertainty? And AutoML tables (Google) had not explained what these rows are exactly, although it seems to me like a lower and upper bound for the start and end values.

- We created new way of prediction based on the Baseline model pytorch created. One model we swap such that $H1 = 256, H2 = 1024$, while another we give $H1 = 256, H2 = 1024, H3 = 256$.

- The former starts off having great value for validation, however as training proceeds the validation loss increases rapidly and stable around 5000+.

- We also changed the code to take in extra 6 parameters (star_temp, etc).

2.25 30 May 2021

- AutoML tuning:

  - 5 node-hours of training.
  - Add extra column named wavelength_num to differentiate between different wavelengths giving different value.
  - Does not pass in AAAA, BB and CC. This means when we make prediction, we choose 3 columns from the 300 timestep values PLUS an extra wavelength_num column in combination to make comparison to reverse engineer back which AAAA, BB, and CC it belongs to as the prediction are not arranged properly. This proves to give unique results, successfully reverse engineered.
  - Caveat: unknown whether it gives good result or not. Checking on the own testing based on other already predicted value, it doesn't seems to give good result. However, it is know that this prediction is inaccurate from previous experience. Will upload on Monday and check what results is predicted (how good/bad is done with AutoML training).

2.26 29 May 2021

- Starting to think of using model which takes $300x55$ then output 55. We are thinking of improving based on the Baseline model designed by PyTorch instead of using our own model as TensorFlow is difficult to build this model (due to inexperiences).
- Also thinking about doing AutoML Tables (Google Cloud Platform). Currently on the stage of preparing data in the form that fits AutoML training.
- Yesterday's conclusion is proven. RMSProp is also tried and it seems to give better results. Unsure about this and will upload the result tomorrow to see how good it gives. SGD and AdaDelta will also be tried soon.

2.27 28 May 2021

- Table 35, 42 unable to gradient descent. Table 46 gradient descent once then stuck at around 62+ for our metrics. This is for the latest model

we have under 27052021 folder.

- New Model:
  - Batch size set to 5.
  - First neural layer number of nodes set to batch size.
  - Changed optimizer from Adam to Nadam.
  - Does not train on the whole dataset. Instead, (non-randomly) select value every 47 counts in an organized by AAAA, BB, CC table. The reason we use 47 is because if we choose a number ending with other value except for 3 (and perhaps 9) it will be quite repetitive. Say if we choose 50 then it will always pick those BB, CC with 5, 0 and 0, 0.
  - Training epoch changes to 15.
  - Changed second layer from 1064 neurons to 1024 neurons.
- **Expectation:** Expecting to be better than 26052021 but worse than the other two. Let's prove our conclusion tomorrow for a better baseline of checking.

2.28 27 May 2021

- Not good. Yesterday's point decrease to 7384.0. Perhaps because it isn't representative in some cases by choosing such number of points only. Requires more thinking on this later on.
- I also noticed there are more anomalies in the ArielChallengeMetric-Numerator in more tables. Perhaps we could do on smaller dataset to check this out first as indicators and minimize it based on this, by using a small dataset of say, 512 elements, loop through all tables, for 1 epoch (sufficient), and see how it works.

  However, the loading part of bigquery still requires parallelism though as that is slow.
- With the above insights, a query to take subset of the full dataset are saved into another set of bigquery dataset for easy retrieval (as joining and picking row numbers are compute expensive). This might ease

hparams tuning. This is under the new dataset called "partial_set". Note the table is not stored in any normalized form (not even 1NF) as we are loading the whole table the whole time so that is not required.

- Implementation of new way of normalizing data and working on validation split for training.

2.29 26 May 2021

- Attained 8436.0 for the points. Perhaps training for 13 epochs is too little? Or other reason?
- After data preparation once again, instead of using all available columns for training, I use correlation to find those having top 100 correlations and found that all of them lies between the column "_99" and "_199" inclusive. Hence, instead of nit-picking the exact columns, we will choose these columns for training only, ignoring the others.
- Retain the method of normalizing the data by the original normalization method.
- New model in place. We now have a "SQUID" (term extracted from Solid State Physics notes) model that forms like a closed-form oscillation shape (unable to understand by others what I am talking about).

  In simple words, It has a small opening and big middle and small ending, like a standing wave shaped for the number of neurons in our new model. The layers from top to bottom number of neurons are: 30, 125, 250, 125, 30, 1.
- Except for the last layer we used sigmoid, the other layers are changed from `tf.nn.relu` to `tf.nn.leaky_relu`. This is an experimentation and unsure whether they will be correct.
- Also, batch size are increased by five-fold (to 50) for faster training. Unsure if this helps training accuracy and precision or not (as in some model trains better with smaller batch size). However, it does improves training time (from 60-80 seconds per epoch to 5-10 seconds per epoch).

  Reason not increased to ten-fold is due to noticed difficulty in gradient descent (loss keeps oscillating despite overall trend going down, a signal

of instability in training). It is also a signal to tune the batch size to reduce noise in oscillation of loss (and other metrics if applicable and important).

- Training is not for 13 epochs anymore but 50.

Downsides noticed:

- Still, the downside is loading data into memory and preparation, and writing data to memory/disk. This has a lot of time spent here.
- With a larger batch size, it uses more memory. Previously it uses about 6-9 GB of RAM, but not it uses 9-13 GB of RAM.

2.30 25 May 2021

- **Warning:** Noticed table 45 training have a very large numerator value: much much larger (2x) than from other tables. Perhaps worth checking for problems in that table.
- However, seeing table 46 also have quite high a numerator: although only about 1.5x the value.
- This trend continue to decrease into table 47. Since training is still on table 47 hence unsure what would happen afterwards. Will check again tomorrow.

2.31 24 May 2021

- Created a model that is exactly like Baseline, except of course we are training 55 models.
- We are using 3 Dense layers with first layer 1064, second 256 and final output 1. We are also training for 13 epochs per model.
- We create a metric trying to copy from the Baseline but realized it exploded (well, almost exploded at about a few trillions which is not feasible). We realized that perhaps we are implementing the wrong metrics, and forced it to fit by removing the 10^6 from the equation.

- Finally, perhaps the sum on the numerator is the one that is important rather than the denominator, so only the numerator is displayed is also implemented. This comes from the realization that the `sample_weight` passed in is always none, hence always evaluate to 1. How to implement this weight is another question.

2.32 23 May 2021

- Inefficient way of storing data in bigquery found. There is a row where AAAA = label for each wavelength. This could be stored separately into a table where we have AAAA and 55 different values for the labels. Now we are repeating the same number 100 times instead of just once, and it takes up some storage space and inefficiencies.

2.33 22 May 2021

- Transferred test csv files into bigquery.
- Start making of model for training. We will start by using baseline just with a few dense layers.

2.34 21 May 2021

- Fixed hard programming on comparing list; rather, change them to frozenset (set) and compare its length and its content. Length comparison will ensure all items inside are unique, or else one element would be overwritten and length of one would not be same as length of expected "hardcoded". If not unique then another method will be required. But assumed unique because we would not want same file name under the same directory.
- Done all except for last bullet point from 19 May 2021.

2.35 20 May 2021

- Initialize multiprocessing to use **Pool** so we could process file faster.

- Added 3 columns "AAAA", "BB" and "CC" into the original form, so that "AAAA" could be used as the foreign key in a DB when trying to retrieve data from the extra params file (containing the 6 params such as sun temperature, etc).

Some drawbacks:

- Unfortunately we did not fix the table for extra params. Originally we use AAAA as the index_cols labelling, however now we need to extract it, and this is done manually by hand instead of code, which is part of the major drawbacks of no automation in this, just because we only have 3 files and it's much easier to just edit them without any thinking requirement rather than think hard on it. And also we don't need a pipeline for production. This could be add in later if requires pipelining, after transpose, using Pandas.
- Files are **left untransformed** in the SQL table and requires transformation after loading into memory (Pandas). This way we could hyperparameter tune the data transformation to see which works best rather than a rigid hardcoded transformation. Ultimately, if this is pipelined, perhaps transformation before entering into sql table would be the best but not during experimentation, which is our stage.

2.36 19 May 2021

- Original extra parameters file are being transposed.
- Make txt_to_csv_noisy.py to change the original text file data into csv files, stored separately as 55 different csv files corresponding to 55 different wavelengths for training on 55 models.

To do next time:

- First row needs renaming for extra params to become a primary key field.
- Add primary key field to the 55 csv files.
- Add tqdm to see progress when looping over large files.

- Do the same thing (without the labels) for the noisy test files.
- Refactoring of the functions into utils.py to make things simpler.

2.37 17 May 2021

- New information from Baseline model: They remove the "changed initial 30 points = 1 to remove ramp" and the model does slightly better. Do we want to implement the removal of ramp or not? If so, do we want to remove less of the ramp or do we not? What are the consequences of removing and what are the consequences of not removing? What is the cause of the ramp?

2.38 30 April 2021

- Changed how extra params function read files by looping over smaller dataset and save instead of looping over everything. This way could save memory more. However, still does not have EC method in place to resume after dropout.

  And this is only for temporary used. Implementation of SQL server will come soon.

- Updated function read_Ariel_dataset._choose_train_or_test() does not modify the original list (now original list treated as immutable). Also remove batch size support classified as "Not Implemented".

- Own notice: Notice after calling "git stash push" all the changes are missing. This is because git revert the current change back to original "pull" file. Either call "git stash pop" to pop the saved changes out from the stash, or use "git stash apply" to apply the latest stashed file change to current. This is unnoticed in the past when I used git stash hence unsure whether some changes are being made after I started using git stash.

- Feature store for noisy time series is in process of creation (for SQL server). Up next!

- Feature store for extra params created with csv file in gcp bucket under "feature store" folder in parent directory.

2.39 29 April 2021

- Copied ArielMLDataset from https://github.com/ucl-exoplanets/ML-challenge-baseline/blob/main/utils.py and remove the particulars for pytorch.

- Added comparison for feature store: the extra parameters outside of time series should be the same for all star (AAAA), with pytest. If all correct, save the combined csv file a new folder named feature store, for all data.

2.40 28 April 2021

- Looked into transformation of data. Basically, transformation of exponential and square of data just before minus by the offset (to make it zero, either by 1 or by e for exponential transformation) will increase the noisy-ness (larger noisy fluctuation) in the non-transforming region, and lesser noisy fluctuation in the transforming region. Note that these are not absolute but **relative values** being discussed here (relative to each other). And not sure if larger noisy fluctuations would affect model learning the "dip" or not since the fluctuation is now as big as the dip after transformation.

  Note this is just an implementation that is being made optional in the current release.

Chapter 3

Discovery

Using setup() and teardown() in pytest would means it setup and teardown with every single test available. However it is repeatedly setup and teardown in every single test which might not be good if targeting only certain test cases.