Siim-isic-melanoma-classification

# Abstract

In this competition, the most powerful ur NN is, the higher the score u will get.

For instance, using

EfficientNetB6 with 5 Fold and image size 256x256 and 15 epochs, and external data

Is better than using

3 Folds, image size 128x128, efficientNetB0, and 3 epochs

# TODO

* (done) Setup Colab to be able to run the projects from there using TPU.
* (done)Chris CNN different models to ensemble.
  + (done) Val\_accuracy instead of val\_loss
  + (done) INC2019 to 1
  + (done) 15 epochs
  + (done) Try GlobalMaxPooling2D. Worse Results
  + (done) Try EF7. It runs out of memory even with batch size 1.
* (done) Complete training all resolutions.
* (done) Read about upsampling. Not worth it. Only 500 images new because some of them are already in other datasets. Also may include duplicates.
* (done) Read about CNN embeddings and using them in KNN, or XGBoost with meta features. Not good results as far as i have seen.
* (done) Analyze why the submissions of 384 are different
  + Gcs path correct.
  + It seems that I cannot recreate the same predictions. I have tried even for 1 single fold, without restarting the sessions, and it is always different.
* (done) Gather submissions of all resolutions
* (done) Ensemble different models with different resolutions. Think about how to ensemble.
* (done) When bagging predictions, you can do it in a smart way by finding the weights of each file of predictions: <https://www.kaggle.com/steubk/simple-oof-ensembling-methods-for-classification>

My CV score after applying this is better than the one reported here using top kernels. Really good news.

* (done) Ensemble with meta features: There is a public notebook that only uses meta features [here](https://www.kaggle.com/titericz/simple-baseline). Everyone sees an increase in CV LB if they ensemble with this public notebook (i.e. uses these meta features).
* (done) Read about post processing techniques <https://www.kaggle.com/khoongweihao/post-processing-technique-c-f-1st-place-jigsaw>
* (done) Read about MinMax

# Ideas

* **Fine tuning.** Learn model using a pretrained model and only unfreeze top. After some epochs of training, unfreeze more layers and train more epochs (fine tuning). I don’t think fine tuning is a good idea because the data used for training the base model is very different from the data of our problem. <https://www.tensorflow.org/tutorials/images/transfer_learning>
* **XGBOOST after CNN to use metadata.** From chris. I have extracted image embeddings from the trained CNN models and then inputted the image embeddings into XGBoost with the addition of meta features.For instance, If you modify the CNN to use tabular data, or if you ensemble this notebook with the notebook [here](https://www.kaggle.com/titericz/simple-baseline), then the CV and LB will increase. If ensembling, use something like 0.9\*CNN\_pred + 0.1\*Tabular\_pred.
* Ensembling of CNN models. Notice that the Stratifiedkfold model from Chris doesn’t make an ensemble of models. It just creates folds to train better the model based on similar data. Each tfrecord has similar data. But at the end, we have a set of predictions per each oof, and we have a number of oof equivalent to the number of folds set. Ths is the reason why, all the folds need to have the same specs. If we want to try different image sizes, we need to run another cnn with new folds and new image sizes.

# 

# 

# 

# 

# 

# 

# TFRecords

TFRecords are generally used for TensorFlow models (GPU or TPU). There is code on the internet that uses TFRecords in a PyTorch dataloader. You read in the TFRecord and then convert to a PyTorch tensor. But maybe a folder of jpegs is better for PyTorch.

Some patients have as many as 115 images and some patients have as few as 2 images. When isolating patients into TFRecords, each record has an equal number of patients with 115 images, with 100, with 70, with 50, with 20, with 10, with 5, with 2, etc. This makes validation more reliable.

You can also use the JPEG format instead of TFRecords.

<https://www.kaggle.com/cdeotte/how-to-create-tfrecords>: How to create TFRecords

# Size images

Kaggle's given TFRecords that are 1024x1024.

The size 1024x1024 is NOT best. Sizes 768x768, 512x512, 384x384, and 256x256 do better. Less overfitting.

Additionally, ensembling models using different sizes can outperform a model using a single size. Source: <https://www.kaggle.com/c/siim-isic-melanoma-classification/discussion/160147>

If you ensemble models of different sizes, you can achieve LB 0.960 or higher!

The reason of it is that we are using pretrained models weighted on the imagenet dataset. These weights were trained using 224x224. This affects how the model learned patterns of certain sizes. For instance it learned to detect triangles of 50 pixels of size. Now image that in our dataset these meaningful triangles only appears in 1024x1024 resolution while other meaning patterns lice cercles 20pixels only appear in those of 128x128. If we want to catch everything it is better to provide different sizes.

# Data augmentation

If you're inputting this data into a TensorFlow TPU model then you are restricted to tf.data augmentations. If you are using TensorFlow GPU or PyTorch GPU/TPU then after reading the TFRecords then you can use albumentations.

<https://www.kaggle.com/saife245/cutmix-vs-mixup-vs-gridmask-vs-cutout>: Kernel about how to apply different data augmentation techniques: Cutmix vs Mixup vs Gridmask vs Cutout

These are the most popular data augmentation techniques these days.

**Upsample and coarse dropout**

<https://www.kaggle.com/cdeotte/tfrecord-experiments-upsample-and-coarse-dropout>

# External Data

<https://www.kaggle.com/c/siim-isic-melanoma-classification/discussion/164910>

In all my trainings I am using external data. But I am not using upsampling.

My 512x512 TFRecords [here](https://www.kaggle.com/cdeotte/512x512-melanoma-tfrecords-70k-images) that include external data are created with a Kaggle notebook and they (and only they) use 93% quality. I should change it to 100%. I would have to create the TFRecords by myself and then publish the dataset in order to be able to use it. Not worth it.

### About upsampling

Upsampling is not worth it (only chris has reported improvements) and right now I cannot train everything again.

An alternative to upsampling is class weights:

<https://scikit-learn.org/stable/modules/generated/sklearn.utils.class_weight.compute_class_weight.html>

But not proven to work in this competition.

# Image and Tabular Data

Ideas explained here:

<https://www.kaggle.com/cdeotte/image-and-tabular-data-0-915>

# Out-of-the-box Kernels

<https://www.kaggle.com/cdeotte/triple-stratified-kfold-with-tfrecords>

* 0.9454
* This is a great Kernel from Chris Deotte using Triple (or more) Stratified KFold CV with TFRecords with EfficientNetB6.
* It gives you the possibility to have submissions straight forward with a super good score. And the flexibility to tune and improve even more the results. Using 5 folds it takes 2.2h of TPU time to train and generate the submission file.
* You can control which size images are loaded, which efficientNets are used, and whether external data is used. You can experiment with different data augmentation, model architecture, loss, optimizers, and learning schedules. The TFRecords contain meta data, so you can input that into your CNN too.
* **Notice that this kernel doesn’t use the metadata!** We have metadata in the TFRecords but we are not using it after reading the tfrecord. Notice that we only return the image and the label, not the other features.
* **(no) I think that MaxPooling should be better than avg pooling.** [**https://www.machinecurve.com/index.php/2020/01/30/what-are-max-pooling-average-pooling-global-max-pooling-and-global-average-pooling/**](https://www.machinecurve.com/index.php/2020/01/30/what-are-max-pooling-average-pooling-global-max-pooling-and-global-average-pooling/)

<https://www.kaggle.com/agentauers/incredible-tpus-finetune-effnetb0-b6-at-once>

* 0.9419
* This is a kernel using Tensorflow and TFRecords.
* Data TFRecords 256x256. I think it doesn’t use external data.
* The model is an ensembling of EF0, EF1… up to EF6. Then it takes the predictions of each one and computes the mean.
* Callback learning rate (adjusted depending on the epoch)
* Multiple image augmentations using tensorflow.
* It doesn’t make any CV. Just one fold to train the data with all training data.
* It makes the predictions on the test data but after augmenting the test data. Notice that this is valid since we have all the test set and we only need to deliver the csv file with its predictions.
* **Conclusion**: Probably high score due to overfitting. But I don’t expect it to be general. It doesn’t do CV, lots of parameters, only one resolution of images, no external data, and the data augmentation is quite simple.

<https://www.kaggle.com/graf10a/efficientnet-bn-tabular-features-tf-cv5-512x512>

* CNN with tabular data used as an input for the CNN as well, ie combining images and tabular data.
* It turns out that this method doesn’t work for this competition.
* It is better to have both models predicting separately and then ensemble the predictions with a simple average.

<https://www.kaggle.com/cdeotte/rapids-cuml-knn-find-duplicates>

* CNN output as embedding to train a KNN.
* It is useful to find similar images (duplicates), but not as a model to make the final submission.
* Extracting CNN embeddings is a useful skill by itself. Once you have embeddings in a dataframe, you can train any ML model to classify Melanoma images using embeddings dataframe and ignoring the original images. And you can add more features like meta features to your embeddings for improved accuracy. I will demonstrate training a simple ML model using embeddings.
* However, all the notebooks using embeddings have reported to have low scores.

<https://www.kaggle.com/hiramcho/melanoma-efficientnetb6-with-attention-mechanism>

* This is a notebook pretty similar than Chris but a bit improved.
* However not too much improvement.
* I cannot train everything again.

# 

# Submission file

The submission is sorted by image\_name.

submission = submission.sort\_values('image\_name')

Before converting to\_csv

For each image\_name in the test set, you must predict the probability (target) that the sample is malignant. The file should contain a header and have the following format:

# Offline training

Offline training is allowed by all competitions. Kaggle doesn’t want its competitors to feel constrained by the resource limitations. In terms of inference, we need to distinguish two types of competitions. Kernels only competitions and normal competitions. Kernels only competitions force u to have the inference pipeline in a kernel, ie the predictions made in a kernel. In kernel only competitions uploading a static submission to Kernels (as opposed to source code or a model capable of predicting) will cause your code to fail when the Kernel is rerun on the held-out test set.

On the other hand, the rest of the competitions allow you to make the inference offline and just submit the csv. Notice that the csv includes the predictions of all test set. The public score is computed with 30% of this test set. The final score is computed using 100%.

Always read the rules of the competitions for clarification.

# Run with GPU locally

**Better to use Colab TPU.**

I can run this on GPU. Make the following changes. In code cell 4, change DEVICE = 'TPU to DEVICE = 'GPU. And change IMG\_SIZES to IMG\_SIZES = [128]\*FOLDS. If you have memory problems consider changing BATCH\_SIZES = [32]\*FOLDS to a number less than 32. And consider changing EFF\_NETS = [6,6,6,6,6] to a number less than 6. Lastly, in code cell 10, change ds = ds.shuffle(1024\*8) to ds = ds.shuffle(2048).

# Colab training using Free TPU

In order to use data for ur colab project, u need to import the data first in the environment. This is similar to how kaggle works. However notice that there is a disk limitation of 100gb. Be careful to not overpass it. Only upload what u need for each execution.

Wait. There is a better way than doing it manually without having to download the data locally: **Using the Kaggle api. I have prepared it in a notebook.**

**Actually you can reach a limit of usage and you don’t know when you will receive the restart. It happened to me. However, you just need another google account to run the notebooks from there.**

## Data from GCS

Apart from that, we also need to use the data from the GCS (public datasets are there).

The way to do it is the If you run KaggleDataSets().get\_gcs\_path(file\_path) in a Kaggle notebook, you can just copy and paste the outputted path into a Colab notebook as the GCS\_PATH. U will receive sth like gs://{bucket\_name}/

I train on Colab TPU whenever I'm out of Kaggle TPU for the week: training on Colab is slower than training on Kaggle. Kaggle TPU is 2x more powerful.

Two caveats: the Kaggle API gets you a bucket in the same region as your Kaggle TPU which might not be the region you want. Also, The Kaggle API treats this GCS bucket as a cache. It can erase and recreate it u

## Execution time on Colab

*Google Colab notebooks have an idle timeout of 90 minutes and absolute timeout of 12 hours. This means, if user does not interact with his Google Colab notebook for more than 90 minutes, its instance is automatically terminated. Also, maximum lifetime of a Colab instance is 12 hours.*

We need to think about sth in order to avoid being closed automatically every 90 min and therefore, break our training pipeline.

## Prevent Colab from disconnecting

Open your Chrome DevTools by pressing F12 or ctrl+shift+i on Linux and enter the following JavaScript snippet in your console:

function KeepClicking(){

console.log("Clicking");

document.querySelector("colab-connect-button").click()

}

setInterval(KeepClicking,60000)

This function makes a click on the connect-button every 60 seconds. Thus, Colab thinks that the notebook is not idle and you don’t have to worry about being disconnected!

## Store the weights in the personal google drive storage

Store the weights in the personal google drive storage in order to not lose them during the restart of the colab sessions.

Be careful when selecting the monitor! If u select sth different than val\_loss, must be in accordance to out metric in compile. Ie if in compile I have auc metric, then I should write val\_auc.

Source: <https://stackoverflow.com/questions/58682098/keras-callback-modelcheckpoint-doesnt-save-weights>

# Does the batch size needs to be the same in fit and predict?

Not necessarily.

Source: <https://machinelearningmastery.com/use-different-batch-sizes-training-predicting-python-keras/>

If we do predict with the same batch as in training, we may expect needing the same ram for both training and predicting. However, if we already have the model trained (we have the weights), we can upload the weights and do predictions with a lower batch (even batch size 1 predictions).

Notice that the batch size in inference will only affect the speed of the inference but not the predicted values. We will obtain the same predicted values despite the batch size of inference.

# Distribution of melanoma cases in Public dataset

<https://www.kaggle.com/cpmpml/number-of-public-melanoma-is-78-or-77>

Very interesting, we could expect 60 melanomas to obtain 1.8% part of melanomas in public LB, but there 77, +17 what is significantly differ.

# Ensembling

* Do not trust too much the LB score. Trust more the CV score.
* (done) People are doing nesting blending: blendings of blendings.
* (done) Simple average (mean) of predictions is being used and works well.
* (done) Weighted average can be interesting. For instance weight the values of the model with higher LB or CV x3 wrt the other model values.
* (no) Apart from bagging submissions, I should consider stacking approaches (using the predictions of models as an input of other models).
* (done) MinMax ensemble used in 2018 melanoma competition: <https://www.kaggle.com/c/siim-isic-melanoma-classification/discussion/167465>
  + People report that simple average (mean) works better than this method.
  + It requires experimentation setting the thresholds and I do not have many tries before the deadline.
  + Summary: Notice that it does the average of some predictions files first to generate a csv (uses log transformation here). Then it uses the generated file + other files as an input of the min max ensemble.
  + The MinMax technique does the following: if all your models agree that it's above a certain threshold (cutoff\_lo) then predict the max value among our csv (for that row), if they predict bellow a certain threshold predict the min along your csv, else predict what you consider to be your best model (ie we select the value of our csv that we consider our best).
  + **All the submissions files used for the competition are shit. Anything reusable.**
* (done) MinMax ensemble actually used in this competition:

<https://www.kaggle.com/paklau9/minmax-highest-public-lb-9619>

* 0.9619 of LB.
* **The submissions used for ensembling are shitty.**
* (done) Ensembling using Avg, Rank, Pow Avg and Weighted Avg (Bayesian Optimization)

<https://www.kaggle.com/steubk/simple-oof-ensembling-methods-for-classification>

* Loads three models similar to chris with different image sizes 2 with 384 and 1 with 512.
* It uses the oof predictions values. Ie the csv with the predictions made per each fold, where we still haven’t average them through the folds to create the submission.csv. **This limits the kernel to only be able to use models with the same cv fold strategy.**
* **Actually I can use it. The oof csv are only used to keep knowing the CV score after blending. We use the oof predictions which are in our training set in order to compute the final oof. But once we know the oof, we use the test set and apply the same technique to generate the submission.csv.**
* (done) Ensembling with tabular data:

<https://www.kaggle.com/cdeotte/image-and-tabular-data-0-915>

* Post processing ensembling:

<https://www.kaggle.com/khoongweihao/post-processing-technique-c-f-1st-place-jigsaw>

* **Everyone reports earnings in LB so it should be one of my submissions.**
* **All the submissions that it uses are shitty.**
* **Summary:** This is one technique that already won one1st place in the past. The main idea is computing the difference of predictions for each sample between pairs of submission files, then averaging those differences in order to push the predictions towards the correct direction. Once you have for instance, 4 differences between submission files, you average them, and then you use this average + you best submission in order to improve the results.
* It may look like a bit of overfitting of the public score. But if the public and private distributions are similar, we should expect nice results.
* (done) I need to check if I can ensemble more models to my model. It is better to have models with oof, which is not the case of what people share. It is better for me to create my models.
* Based on last year solution: <https://www.kaggle.com/c/siim-isic-melanoma-classification/discussion/154683>

If I want to win, I need to create more CNN with different EF levels, combining image sizes and also combining different data augmentation.

* I should try to ensemble with other models as well:
* <https://www.kaggle.com/ipythonx/tresnet-hp-gpu-dedicated-net-grad-accumulation-tta/comments?select=submission.csv>
* <https://www.kaggle.com/vbhargav875/efficientnet-b5-b6-b7-tf-keras?select=submission_EfficientNet_B7_Blend_6.csv>

Blend of EN6 and EN7 using images

# Ensembling submissions I can use

* My 6 EFN models with the different resolutions.
* Tabular\_data:
  + 1. <https://www.kaggle.com/titericz/simple-baseline> 0.6928 (no xgboost)
  + 2. <https://www.kaggle.com/datafan07/eda-modelling-of-the-external-data-inc-ensemble/log?select=external_tabular_predicts.csv> (uses xgboost and removes overfitting features like image height and width)
  + <https://www.kaggle.com/datafan07/analysis-of-melanoma-metadata-and-effnet-ensemble/log?select=meta_with_img_data.csv> CV 0.87 LB 0.85 (xgboost meta + extra features V24, V88…). It can produce overfitting in case that you use for ensembling. However it is worth the try.
* I need to ensemble more CNNs apart from mine that contain oof (or even generate them):
  + (no off?) <https://www.kaggle.com/datafan07/analysis-of-melanoma-metadata-and-effnet-ensemble/log?select=blended_effnets.csv>

Blend of EFN b3 b4 b5 with 384 img size.

Blended\_effnets.csv

* Other ensembles:
  + <https://www.kaggle.com/paklau9/minmax-highest-public-lb-9619?select=submission.csv>

MinMax 0.9619

* min\_max\_9526
* Blended\_effnets
* Submission\_mean (as best model)
* Unknown (?)
* <https://www.kaggle.com/solomonk/minmax-ensemble-0-9526-lb?select=submission.csv>

MinMax 0.9526

* This is based on another ensembles.
* <https://www.kaggle.com/truonghoang/stacking-ensemble-on-my-submissions?select=submission_median.csv>

Submission\_mean and submission\_median.

Generate submission mean and submission median based on ensembling several submissions files (it ensembles submissions from lots of models and they are not all efficient nets.

* This one can be interesting.
* <https://www.kaggle.com/kmldas/new-basline-np-log2-ensemble-top-10>

Ensemble of VGG and Resnet.

* It does not even reference the submissions so cannot be recreated.

# Ensembling Results

128: OOF auc:0.8961

192: OOF auc:0.9073

256: OOF auc:0.9178

384: OOF auc:0.9183

512: OOF auc:0.909

768: OOF auc:0.9281

OOF avg\_auc:0.9392804359619394

OOF pow\_auc:0.9376314120377729

OOF rank\_auc:0.9407371943630771

bo\_avg auc:0.94066086704086

128\_ef0\_01 0.06058943675481898

128\_ef1\_01 0.08979899318290918

128\_ef3\_00 0.11575558252263451

128\_ef6\_11 0.358684911074849

192\_ef0\_01 0.08166750871395734

192\_ef1\_01 0.0867064690760494

192\_ef3\_00 0.4061190051379864

192\_ef6\_11 0.5956901017244923

256\_ef4\_01 0.16326442323691126

256\_ef6\_11 0.6438771445145423

384\_ef6\_01 0.8577346279879365

384\_ef6\_11 0.8574068875756895

512\_ef6\_11 0.6571336908866899

768\_ef5\_11 0.816690565760001

n\_iter = 40

OOF bo\_avg\_auc:0.940662582261584

OOF bo\_pow\_auc:0.9392334817946204

OOF bo\_rank\_auc:0.9426736785604539

n\_iter = 200

OOF bo\_avg\_auc:0.9406923306210155

OOF bo\_pow\_auc:0.9393016082177511

OOF bo\_rank\_auc:0.9430502231100156

Init\_points = 200, n\_iter = 200

OOF bo\_avg\_auc:0.9411691619822828

OOF bo\_pow\_auc:0.9395330558141936

OOF bo\_rank\_auc:0.9430980348876968

Init\_points = 300, n\_iter = 300

OOF bo\_avg\_auc:0.941180686121522 LB 0.9438

OOF bo\_pow\_auc:0.9395294109701552

OOF bo\_rank\_auc:0.9430980348876968 LB 0.9431

MinMax: this is the best I achieve playing with cutoffs.

OOF MinMax\_auc:0.9334485514987787

Ensemble with tabular data:

Submission\_bo\_avg\_300\_t2x1.csv 0.9457

Submission\_bo\_avg\_300\_t2x1\_t1x1.csv 0.9468

**Submission\_bo\_avg\_300\_t2x2.csv 0.9469**

Submission\_bo\_avg\_300\_t3x2.csv 0.9445

**Using 9 best models:**

Adding a lot of models with a low oof doesn't help. It is better to reduce the amount of models just keeping the best ones.

models = [

# "128\_ef0\_01",

# "128\_ef1\_01",

# "128\_ef3\_00",

# "128\_ef4\_01",

# "128\_ef6\_11",

#

# "192\_ef0\_01",

# "192\_ef1\_01",

# "192\_ef3\_00",

# "192\_ef4\_01",

"192\_ef6\_11",

#

# "256\_ef3\_01",

# "256\_ef4\_00",

# "256\_ef4\_01",

"256\_ef6\_11",

#

"384\_ef6\_00",

"384\_ef6\_01",

"384\_ef6\_11",

#

"512\_ef5\_00",

"512\_ef5\_01",

"512\_ef6\_11",

#

"768\_ef5\_11",

]

OOF avg\_auc:0.9423117133870458

OOF rank\_auc:0.9447312198204902

OOF pow\_auc:0.9398256617495754

**OOF avg\_auc:0.9433087926341561**

OOF rank\_auc:0.945549567708097

OOF pow\_auc:0.9410292106913354

**bo\_avg auc:0.94431278636481**

192\_ef6\_11 0.290857070847954

256\_ef6\_11 0.4629979108665405

384\_ef6\_00 0.22824785603673126

384\_ef6\_01 0.12181708787466183

384\_ef6\_11 0.7821094568547555

512\_ef5\_00 0.3494241395013379

512\_ef5\_01 0.7648588746194755

512\_ef6\_11 0.30688679485516057

768\_ef5\_11 0.9787643456782964

OOF post\_processing\_auc:0.932890541956684

### Using best 10 models:

**OOF avg\_auc:0.9449595853796944**

**OOF rank\_auc:0.9468988836110712**

OOF pow\_auc:0.9427253495847637

**bo\_avg auc:0.9463006199826108**

192\_ef6\_11 0.1562575216063946

256\_ef6\_11 0.6343279945916444

384\_ef6\_00 0.2765983002342486

384\_ef6\_01 0.24840484228576354

384\_ef6\_11 0.8972516026732116

512\_ef5\_00 0.214332718832293

512\_ef5\_01 0.8645845664226072

512\_ef6\_11 0.3035272922729195

768\_ef5\_11 0.9912283222805577

768\_ef5\_01 0.980929021292084

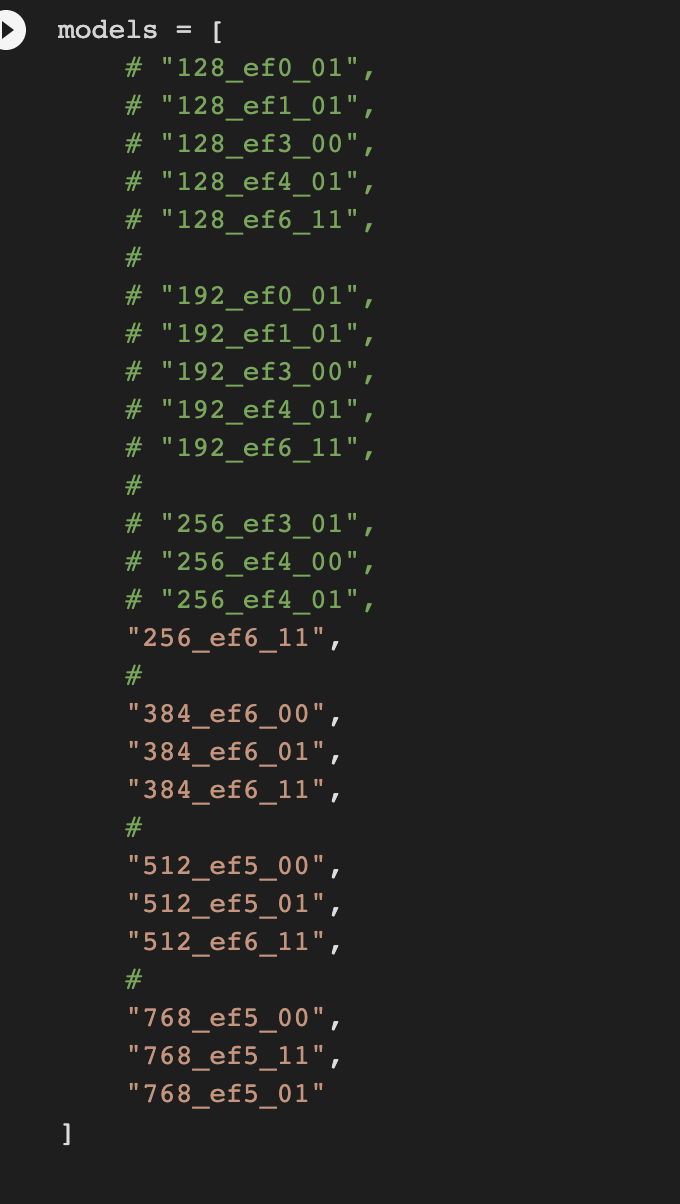
**Submission\_bo\_avg\_10m\_1000\_200\_t2x2 LB 0.9533**

Submission\_rank\_10m.csv LB 0.9492

Submission\_post\_pro\_9648\_t2x1 LB 0.9575

**Submission\_avg\_10m\_t2x1 LB 0.9554**

### Using best 10 models (2):

****

**Removing 192:**

**OOF avg\_auc:0.9451073087645474**

**OOF rank\_auc:0.9469087461302343**

**OOF pow\_auc:0.9428702321352929**

256\_ef6\_11: OOF auc:0.9178

384\_ef6\_00: OOF auc:0.9029

384\_ef6\_01: OOF auc:0.9038

384\_ef6\_11: OOF auc:0.9183

512\_ef5\_00: OOF auc:0.9083

512\_ef5\_01: OOF auc:0.9223

512\_ef6\_11: OOF auc:0.909

768\_ef5\_11: OOF auc:0.9281

768\_ef5\_01: OOF auc:0.9332

**Adding 768 5F 00:**

256\_ef6\_11: OOF auc:0.9178

384\_ef6\_00: OOF auc:0.9029

384\_ef6\_01: OOF auc:0.9038

384\_ef6\_11: OOF auc:0.9183

512\_ef5\_00: OOF auc:0.9083

512\_ef5\_01: OOF auc:0.9223

512\_ef6\_11: OOF auc:0.909

768\_ef5\_00: OOF auc:0.915

768\_ef5\_11: OOF auc:0.9281

768\_ef5\_01: OOF auc:0.9332

**OOF avg\_auc:0.9452538529351526**

**OOF rank\_auc:0.9472924463662539**

OOF pow\_auc:0.9429062517704965

submission\_avg\_10m\_2.csv

**bo\_avg auc:0.9464783061294859**

256\_ef6\_11 0.46297069598692975

384\_ef6\_00 0.10405192017581666

384\_ef6\_01 0.44428940992379107

384\_ef6\_11 0.843492180404384

512\_ef5\_00 0.08255222535064477

512\_ef5\_01 0.346645721187575

512\_ef6\_11 0.12347749933136387

768\_ef5\_00 0.5056723937844688

768\_ef5\_11 0.5380402226110541

768\_ef5\_01 0.9473567274466644

submission\_bo\_avg\_10m\_2\_1000\_20.csv

### Using previous best models + upsampling models

**Adding 384 upsampling helps a lot**

192\_ef6\_11: OOF auc:0.9073

256\_ef6\_11: OOF auc:0.9178

384\_ef6\_00: OOF auc:0.9029

384\_ef6\_01: OOF auc:0.9038

384\_ef6\_11: OOF auc:0.9183

384\_ef3\_01\_upsample: OOF auc:0.9191

512\_ef5\_00: OOF auc:0.9083

512\_ef5\_01: OOF auc:0.9223

512\_ef6\_11: OOF auc:0.909

768\_ef5\_00: OOF auc:0.915

768\_ef5\_11: OOF auc:0.9281

768\_ef5\_01: OOF auc:0.9332

**OOF avg\_auc:0.9454844965218808**

**OOF rank\_auc:0.9477529831306434**

OOF pow\_auc:0.9428464870483951

**Adding 512 upsample doesn’t help**

192\_ef6\_11: OOF auc:0.9073

256\_ef6\_11: OOF auc:0.9178

384\_ef6\_00: OOF auc:0.9029

384\_ef6\_01: OOF auc:0.9038

384\_ef6\_11: OOF auc:0.9183

384\_ef3\_01\_upsample: OOF auc:0.9191

512\_ef5\_00: OOF auc:0.9083

512\_ef5\_01: OOF auc:0.9223

512\_ef6\_11: OOF auc:0.909

512\_ef3\_01\_upsample: OOF auc:0.9185

768\_ef5\_00: OOF auc:0.915

768\_ef5\_11: OOF auc:0.9281

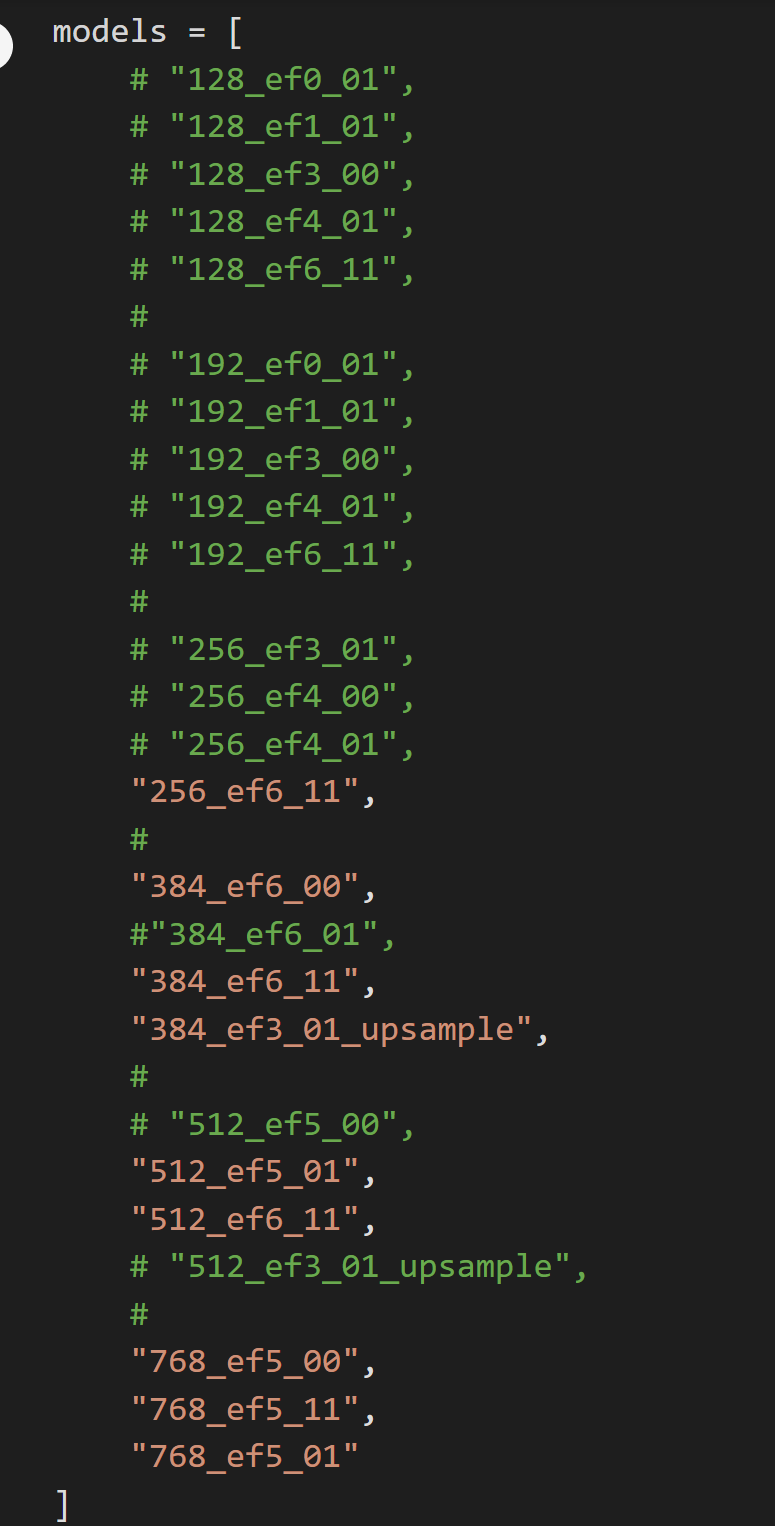
768\_ef5\_01: OOF auc:0.9332

OOF avg\_auc:0.9452446336237613

OOF rank\_auc:0.9477534387361481

OOF pow\_auc:0.9426124666208666

### New selection best models:



**OOF avg\_auc:0.9458945414762078**

OOF rank\_auc:0.948570017802383

OOF pow\_auc:0.9431350729352052

**bo\_avg auc:0.9468190454464348**

256\_ef6\_11 0.2659660658766113

384\_ef6\_00 0.1309044213226167

384\_ef6\_11 0.9812503767271975

384\_ef3\_01\_upsample 0.2075305626842464

512\_ef5\_01 0.6315699217746747

512\_ef6\_11 0.09740425196040792

768\_ef5\_00 0.1741459775016021

768\_ef5\_11 0.7594219531880363

768\_ef5\_01 0.9915194642762638

200 it:

**bo\_avg auc:0.9469610871626396**

256\_ef6\_11 0.40921633963145265

384\_ef6\_00 0.20253548445712322

384\_ef6\_11 0.9727948252923402

384\_ef3\_01\_upsample 0.30080789681055853

512\_ef5\_01 0.3294978327079098

512\_ef6\_11 0.14523897998132285

768\_ef5\_00 0.2596547616442776

768\_ef5\_11 0.4773995450667823

768\_ef5\_01 0.9743656739661126

# 

# Final 5 submissions

**Last submissions done:**

OOF avg\_auc:0.9449595853796944

bo\_avg auc:0.9463006199826108

**Submission\_bo\_avg\_10m\_1000\_200\_t2x2 LB 0.9533**

Submission\_rank\_10m.csv LB 0.9492

Submission\_post\_pro\_9648\_t2x1 LB 0.9575

**Submission\_avg\_10m\_t2x1 LB 0.9554**

**Now we need to try:**

OOF avg\_auc:0.9458945414762078

bo\_avg auc:0.9469610871626396

submission\_avg\_best9m.csv

submission\_bo\_avg\_best9m\_1000\_200.csv

Adding tabular:

**Submission\_avg\_best9m\_t2x1 0.9552**

**Submission\_avg\_best9m\_t2x2 0.9550**

Submission\_bo\_avg\_best9m\_1000\_200\_t2x1 0.9512

**Submission\_bo\_avg\_best9m\_1000\_200\_t2x2 0.9526**

order:

Avg with t2x1

Bo\_avg with t2x2

Bo\_avg with t2x1

Avg with t2x2

If it is not better than before, we have a problem.

From here I should select one or two of the submissions.

### All models:

OOF avg\_auc:0.9390447539143347

OOF rank\_auc:0.9406033267456351

OOF pow\_auc:0.9370840422242319

submission\_avg\_all\_models.csv

Last submit could be

All models bo\_avg with best configuration of tabular