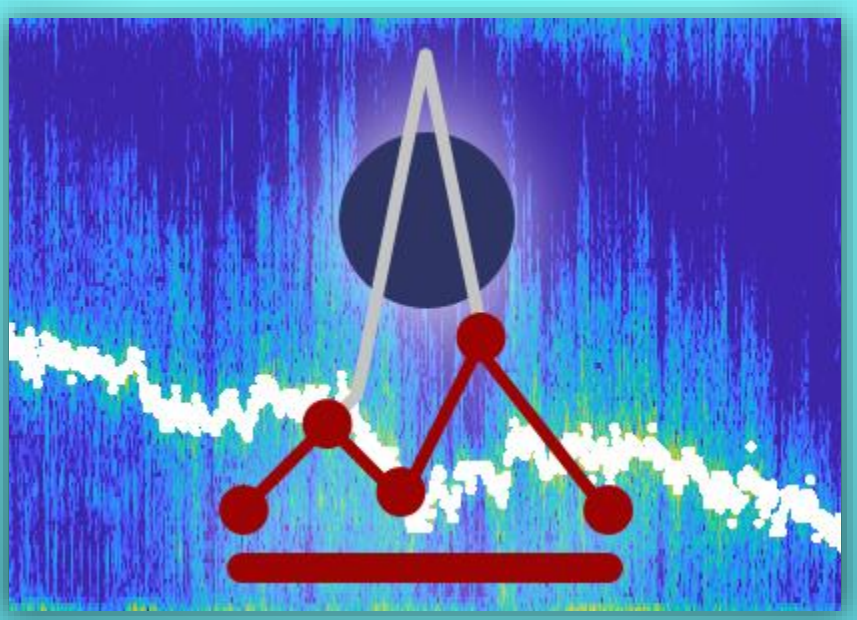
MAFAT Radar Challenge

Can you distinguish between humans and animals in radar tracks?



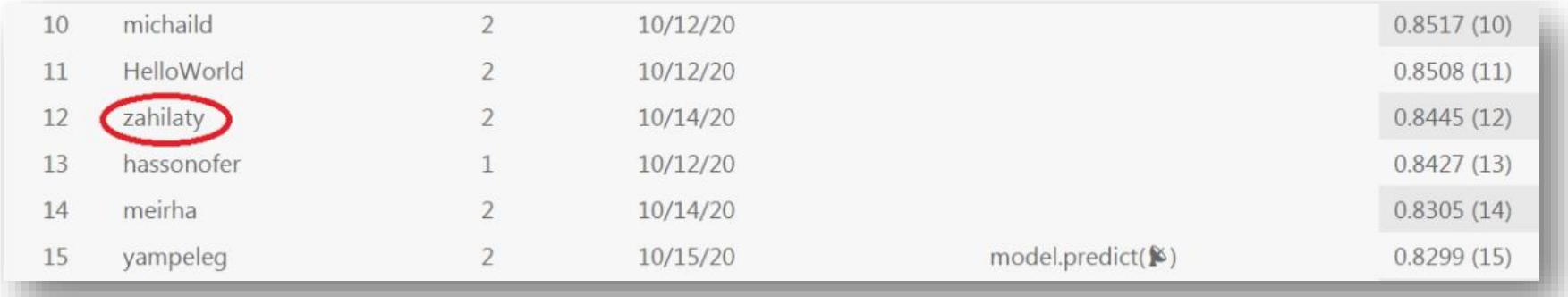
1.11.2020 - Solution by Zahi Laty

Part 1: Introduction

- *About the contest*
- *The data*
- *Base model*
- *The main challenges*
- *Presentation agenda*

About the contest

- CodaLab contest that was organized by MAFAT
- First phase: July 15, 2020 - Oct. 8, 2020
- Second phase: Oct. 8, 2020 - Oct. 15, 2020
- **More than 1,000 registered participants**
- **$40,000 Reward**
- Winners: GSI Technology (0.91), Axon pulse (0.90)
- https://competitions.codalab.org/competitions/25389



10	michaild	2	10/12/20		0.8517 (10)
11	HelloWorld	2	10/12/20		0.8508 (11)
12	zahilaty	2	10/14/20		0.8445 (12)
13	hassonofer	1	10/12/20		0.8427 (13)
14	meirha	2	10/14/20		0.8305 (14)
15	yampeleg	2	10/15/20	model.predict( )	0.8299 (15)

The Data – First phase

- Training data set contains **6,656** radar segments
- Labeled as either **animals** (0) or **humans** (1)
- About **50,000** segments of **human in controlled environment**
- About **50,000** segments of **background** without any target
- Collected from ground-mounted radars, from **4 different locations**

The Data – Second phase

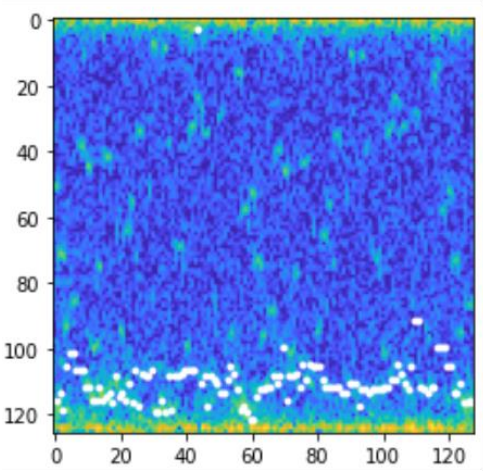
- "Public test set" become available, with **102 hard** examples
- New data was collected from new locations (not seen in phase 1)
- This requires "out of distribution" generalization

The Training Data Set

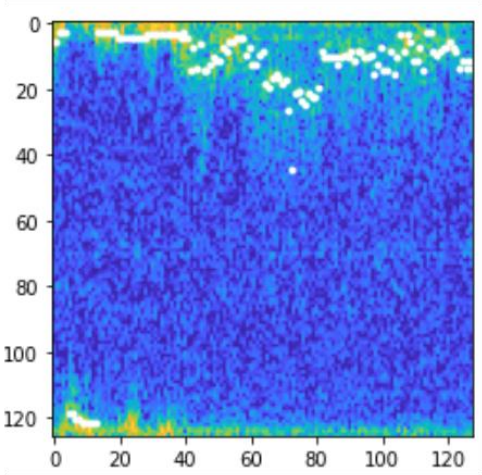
- The Data is basically I,Q samples after match-filter (“compressed pulse”)
- MAFAT provided us with a pre-process script to create this “spectrograms”
- X – Time (not range)
- Y – Frequency \ Velocity
- Z – Amplitude

Does the micro-Doppler variation contains enough information?

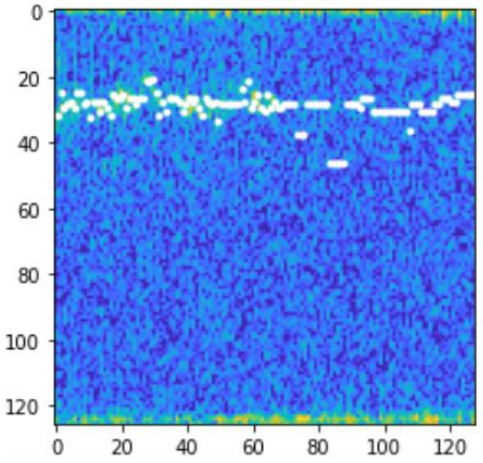
Animal - Low SNR



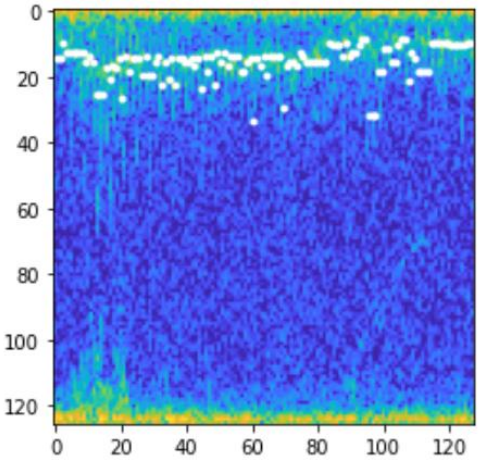
Animal - High SNR



Human - Low SNR



Human - High SNR



Domain knowledge (or speculations..)

What MAFAT has told us:

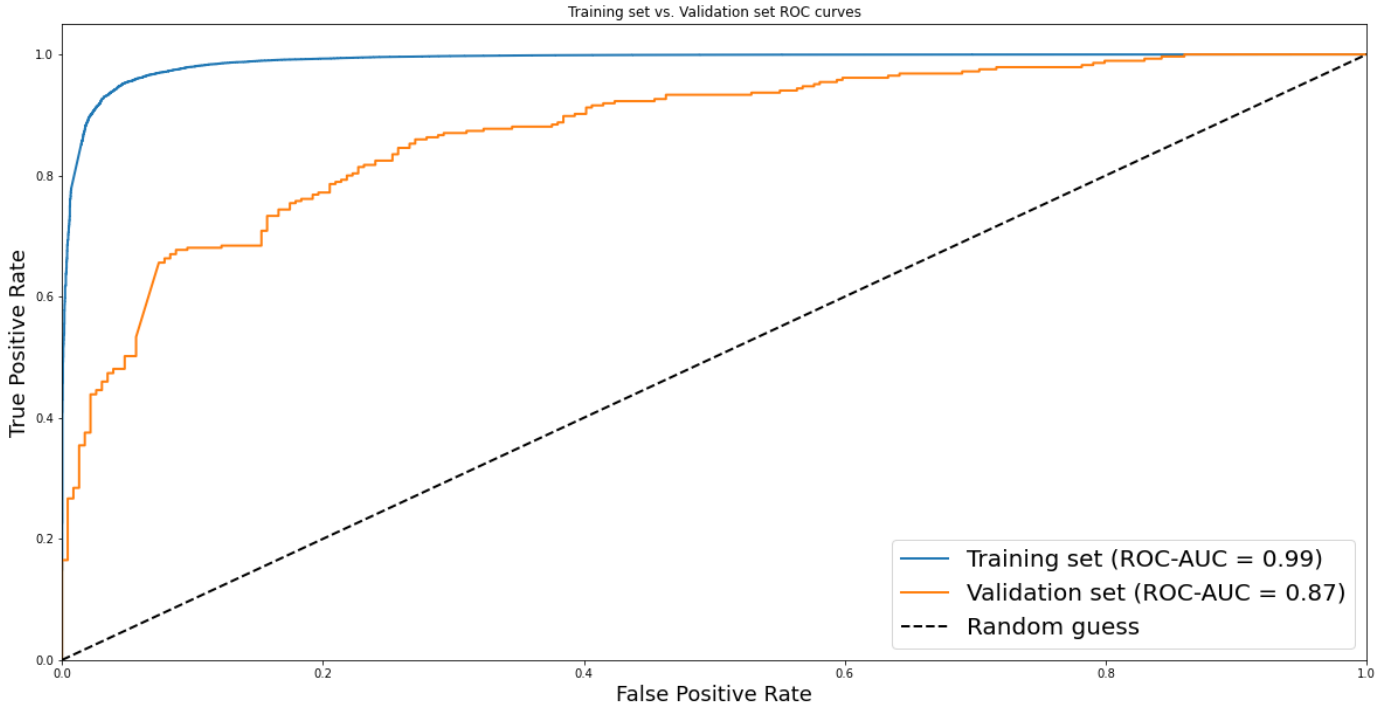
- Each track contains one or more segments
- Each segment contains 32 “slow time units” X 128 “fast time units”.
- FFT should be done along the “fast time” axis.

What we can deduce:

- **This is probably a Doppler radar** (no range information).
- Around 10 cells of difference between human walk (~ 1m/s) and clutter, so the **velocity resolution ~= 0.1 m/s.**
- The spectrogram columns are not coherent, thus, smart coherent processing (using the phase along the slow time as an input) might not be possible.
- **"It is what it is” – we are in the data realm from now on.**

Base model

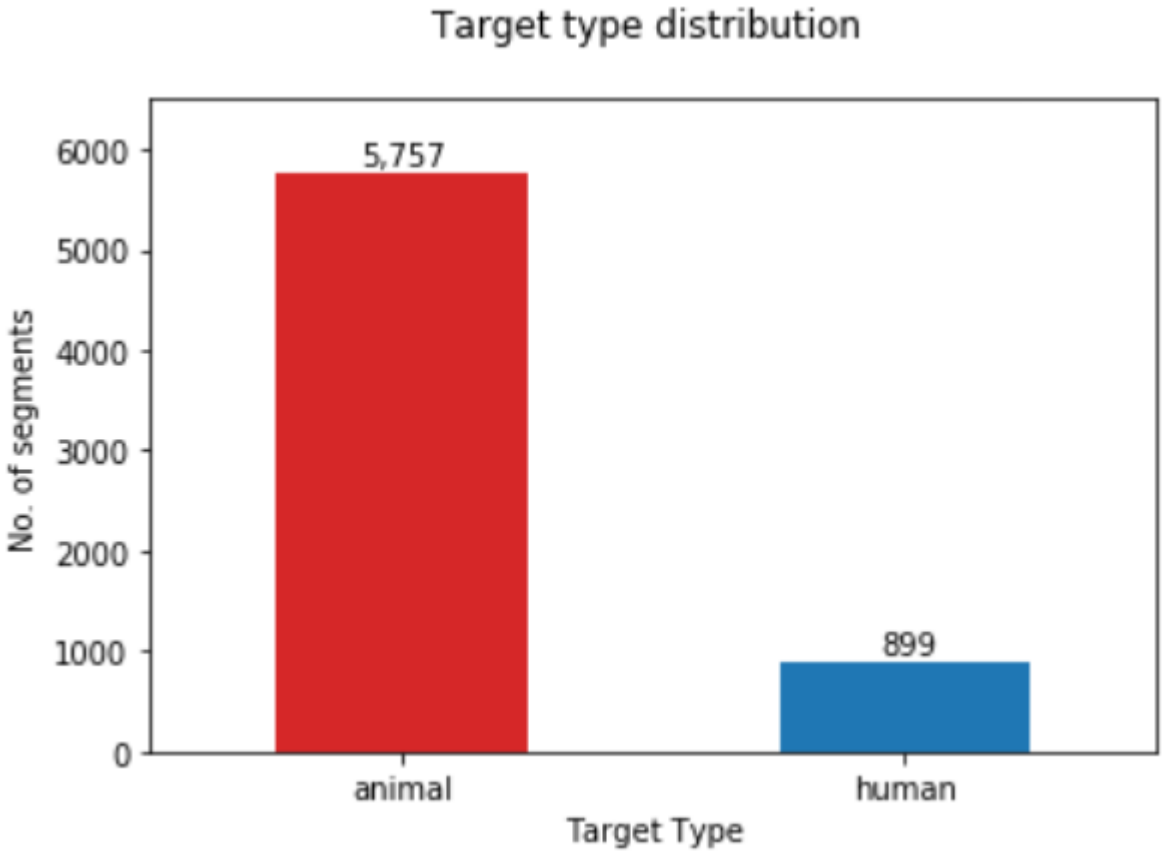
- MAFAT also provided us with a base model:
- Conv2D -> Max Pulling -> Conv2D -> Max Pulling -> 3 layers FCN



Unfortunately, on the "hard" test set, it achieved merely **AUC = 0.78**

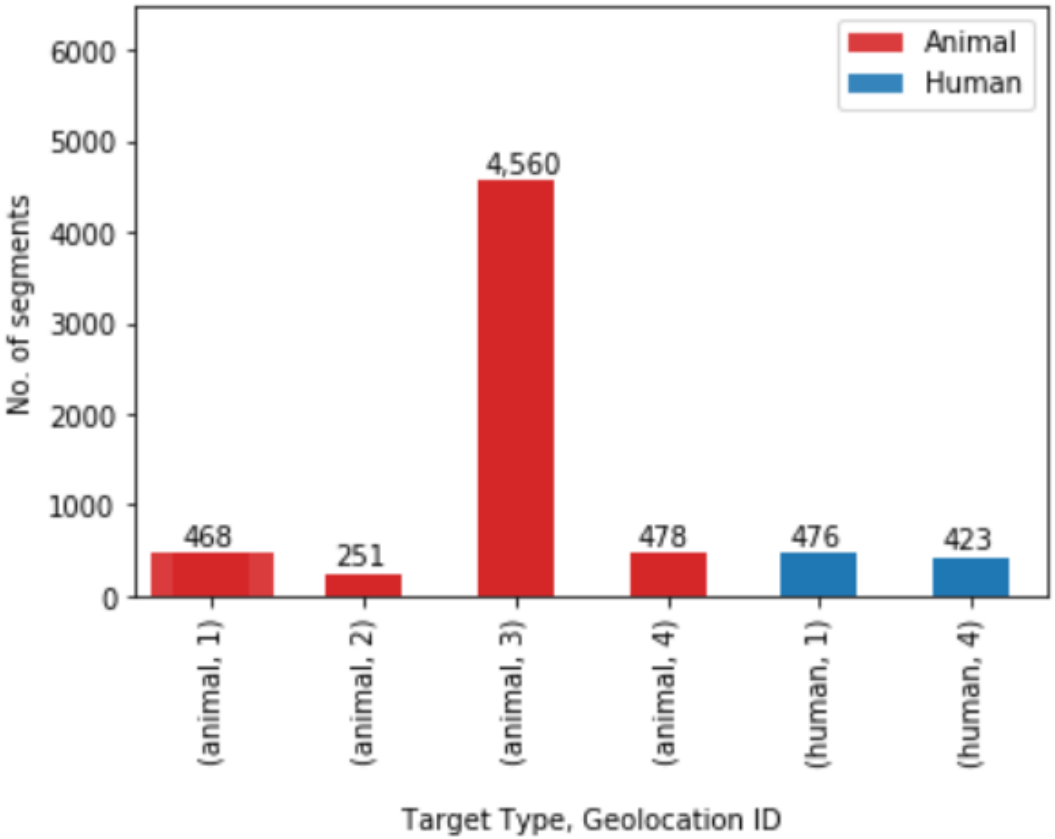
- It can be improved by 6-8% with hyper parameters tuning

Why is it so hard?



- Training dataset is **unbalanced**, while the test set is balanced
- We do have more human samples, but from “different distribution”

Why is it so hard?



- The **geographical diversity** is low (especially for the “human” class)
- Moreover: different SNR, low resolution etc.

Presentation Agenda

1. Introduction
2. Pre-process
3. The optimizer
4. The metric \ loss function
5. Dealing with unbalanced data
6. Architectures
7. Tips and tricks

Part 2: Pre-process

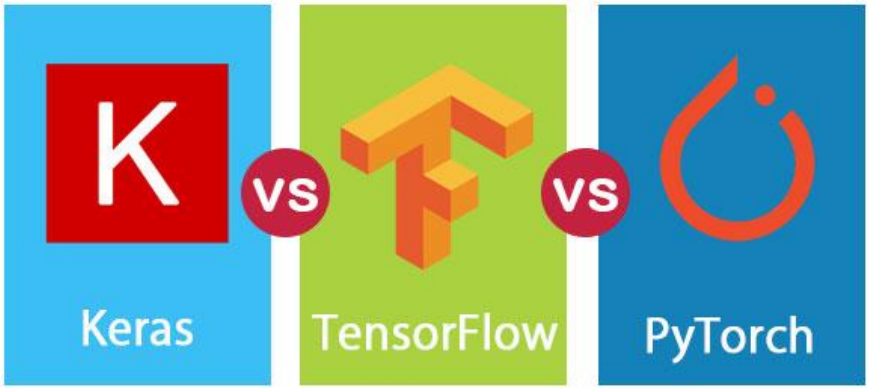
- *From Keras to PyTorch*
- *Creating the spectrograms*
- *Abs and phase*
- *Data split*
- *Shuffle*

The war of the frameworks

Although I am highly biased about this subject, my reasons to switch **from Keras to PyTorch** were:

- I don't care about inference running time (this model is not going to production..)
- I do need maximum flexibility and fast prototyping to test new ideas
- I will need to copy a lot of code from github

The shifting took a lot of time, and some "bugs" were detected here and there, but it was worth it.



Pre-process steps

1. FFT on the “Fast-time” axis (inside “fft”).
2. Extract the magnitude \ power [dB] (inside “fft”).
3. With or w/o Hann window. I used “domain knowledge”.
4. Feature scaling and DC removal (inside “normlize”).

```
def data_preprocess(data,axis=0):

    X = []
    for i in range(len(data['iq_sweep_burst'])):
        iq = fft(data['iq_sweep_burst'][i])
        #iq = np.fft.fft(hann(data['iq_sweep_burst'][i]), axis=axis)
        iq = max_value_on_doppler(iq,data['doppler_burst'][i])
        iq = normalize(iq)
        X.append(iq)

    data['iq_sweep_burst'] = np.array(X)
    if 'target_type' in data:
        data['target_type'][data['target_type'] == 'animal'] = 0
        data['target_type'][data['target_type'] == 'human'] = 1
        data['target_type'][data['target_type'] == 'empty'] = 2
    return data
```

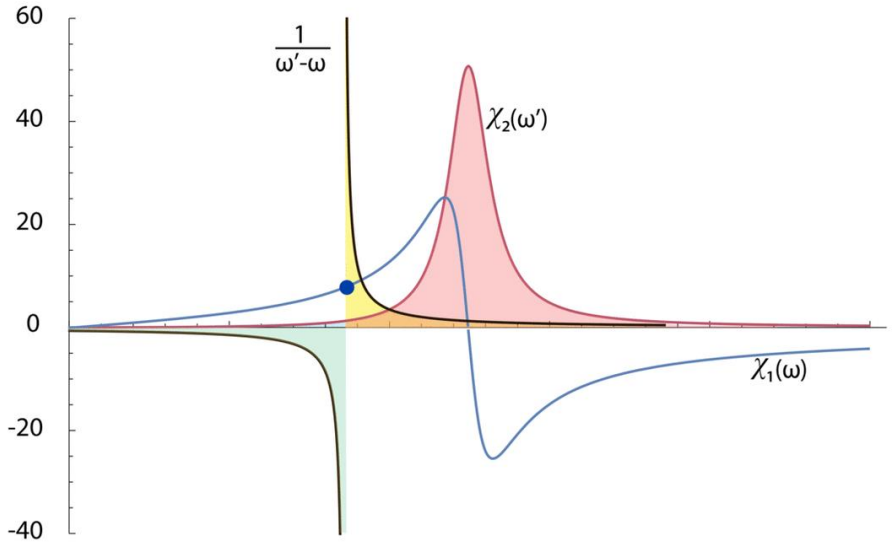
The code MAFAT provided is not efficient , but I don’t care – I need to run it only once, so.. **choose your fights!**

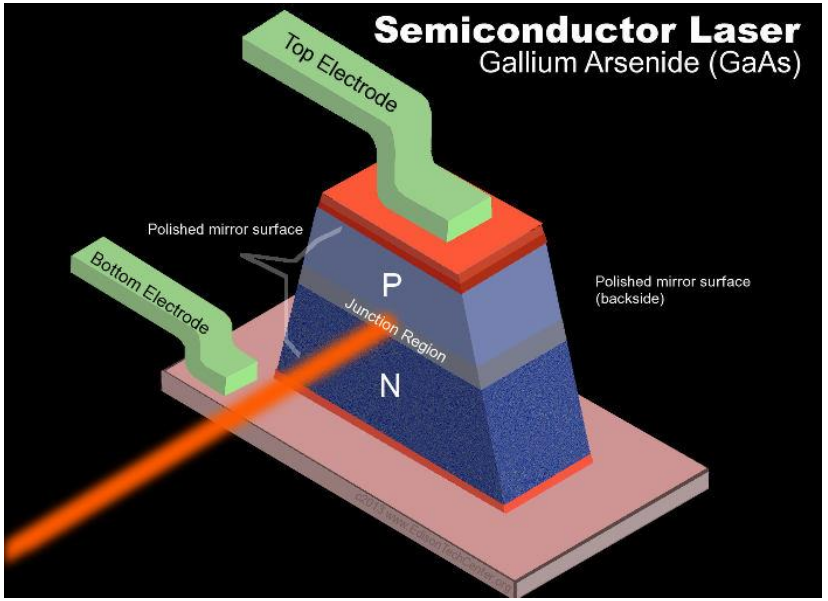
But.. What about the phase?!

1. **Literature survey** about complex networks found articles with inconclusive conclusions about the improvement.
2. From my experience **residual phase is very tricky** – avoid it if you can!
3. **I tried to use it**, by creating 2 head network to process the amplitude and residual phase. I didn’t get a significant difference.
4. It seems that **we can manage without it** (and this assumption might be backed up with mathematical prove)

Kramers–Kronig relations

$$\chi_1(\omega) = \frac{1}{\pi}\mathcal{P}\int_{-\infty}^{\infty} \frac{\chi_2(\omega')}{\omega' - \omega}\, d\omega' \qquad \chi_2(\omega) = -\frac{1}{\pi}\mathcal{P}\int_{-\infty}^{\infty} \frac{\chi_1(\omega')}{\omega' - \omega}\, d\omega',$$





If it is a "real world" problem amplitude and phase must be coupled.

We are not "throwing information".

Data split

- Choosing the right cross validation set is critical – even more than choosing the architecture
- **On the one hand:** test the NN with samples it didn't see before to make sure it can generalize.
- **On the other hand:** If we don't provide diverse-data, the NN won't learn much.
- **Cross validation size** should be chosen according to the desired resolution.
- I took **196 samples** (~0.5% resolution) from **locations 1 and 4** to create a **balanced** cross validation set

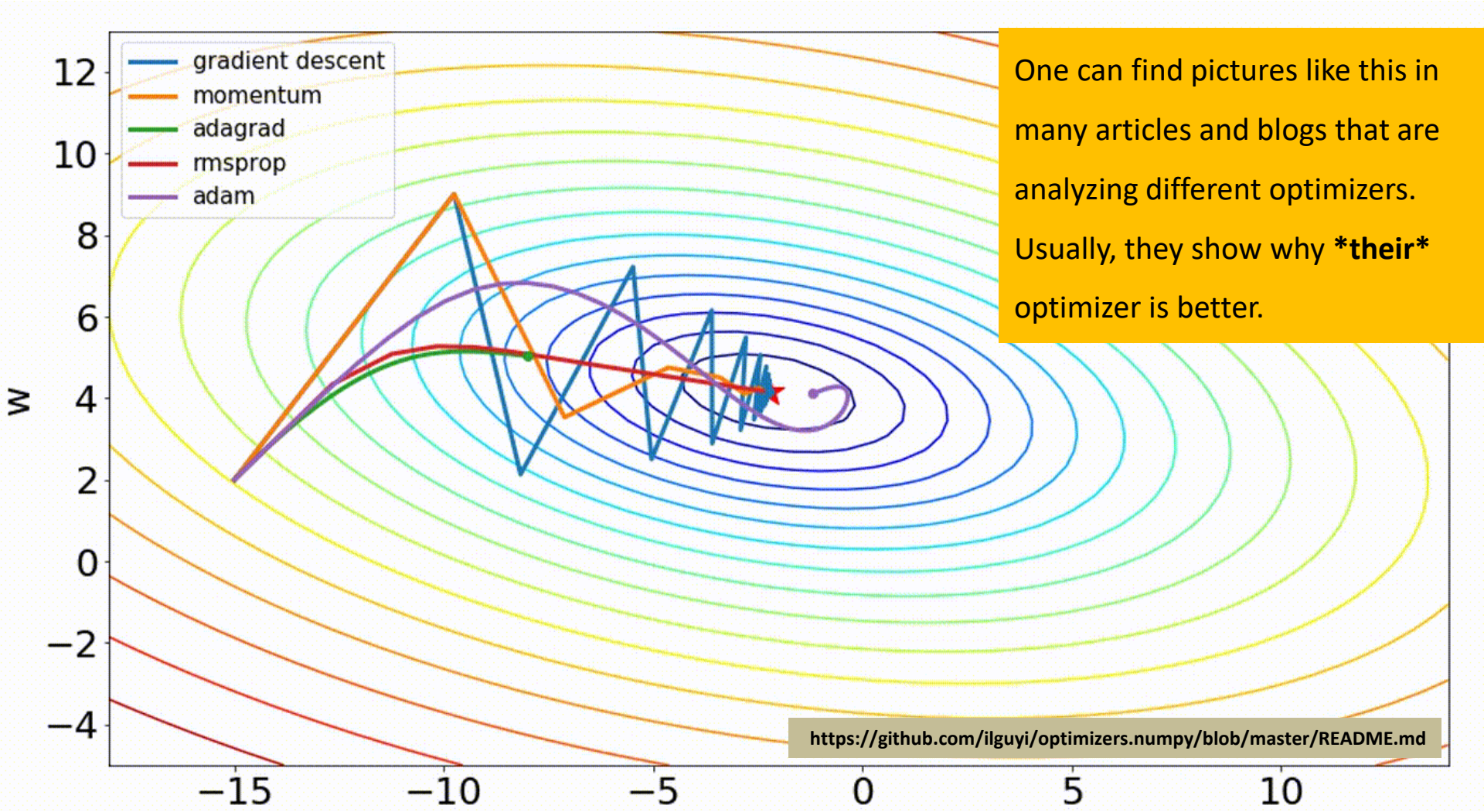
Why 16? Stay tuned..

```
idx = ((data['geolocation_id'] == 4) | (data['geolocation_id'] == 1)) & (data['segment_id'] % 16 == 0)
```

Part 3: The optimizer

- *Adam vs SGD*
- *Learned parameters*
- *Learning rate*
- *Regularization*

Optimizers



One can find pictures like this in many articles and blogs that are analyzing different optimizers. Usually, they show why ***their*** optimizer is better.

https://github.com/ilguyi/optimizers.numpy/blob/master/README.md

Adam & SGD – my conclusions

SGD

- Choose it if you an expert in optimization or want to break the current SOTA
- Learning rate should be determined according to the batch size
- Tune the LR scheduler
- Can have better generalization (allegedly..)

References:

Daniel soudry, Train longer, generalize better: closing the generalization gap in large batch training of neural networks, https://arxiv.org/abs/1705.08741

Adam & SGD – my conclusions

ADAM

- Choose it if this is your first time training model for this task
- It is a very “forgiving” optimizer
- it “normalizes” both direction (“momentum”) and step size (“RMS-prop”)
- Learning rate should be known: around ~3e-4
- LR scheduler doesn’t really matter (but I will use it for different purpose)

References:

http://karpathy.github.io/2019/04/25/recipe/

My Choice

Tried several LR

Convolution and FC parameters should be regularized diffrently

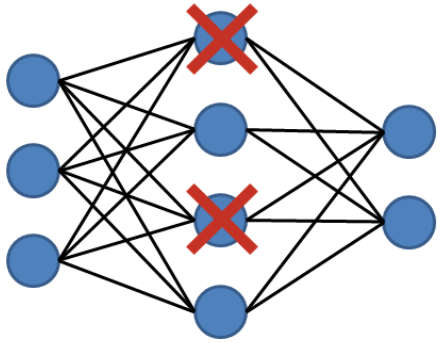
```
######## Hyper-params #########
Lr = 0.001 #0.001 #0.5
Betas = (0.9,0.999) #momentum and RMSprop of adam optimizer
BatchSize = 256

weight_conv, bias_conv, weight_fc, bias_fc = GetWeightBiasParams(net)

#optimizer = torch.optim.Adam([{'params': weight_conv, 'weight_decay':0},{'params':
bias_conv, 'weight_decay':1e-3},{'params': weight_fc, 'weight_decay':1e-
3},{'params': bias_fc, 'weight_decay':0}], lr=Lr, betas=Betas)

optimizer = torch.optim.Adam([{'params': weight_conv, 'weight_decay':0},{'params':
bias_conv, 'weight_decay':0},{'params': weight_fc, 'weight_decay':0},{'params':
bias_fc, 'weight_decay':0}], lr=Lr, betas=Betas)

#optimizer = torch.optim.SGD([{'params': weight_conv, 'weight_decay':
bias_conv, 'weight_decay':0},{'params': weight_fc, 'weight_decay':0},
bias_fc, 'weight_decay':0}], lr=Lr,momentum=0.6)


scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer, milestones=[2,15],
gamma=0.5)
```

Regularization will be done in other means

Under debate…

Regularization

- The use of weight decay \ L2 regularization seems to be more rare in modern deep learning architectures, and for a reason.
- Modern CNN have many parameters, so they can basically learn “a lot”, and the classical **Bias-variance trade off does not always remain true**
- The use of **dropout creates “mini ensembles”** and proved to be more useful
- **Batch normalization** has a slight regularization effect as well



References:

https://openai.com/blog/deep-double-descent/

https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/

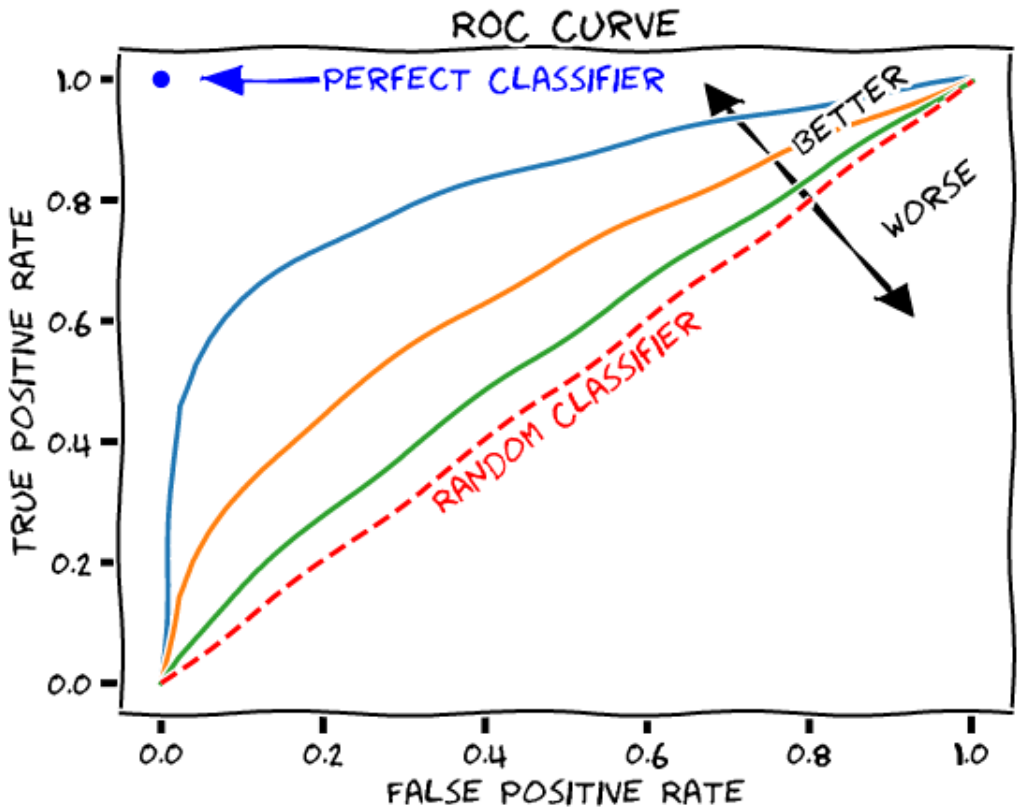
1.11.2020

Part 4: ROC - AUC

- *History*
- *Metric vs loss*
- *Alternatives*

ROC-AUC

- *"The ROC curve was first used during World War II for the analysis of radar signals before it was employed in signal detection theory."*
- https://en.wikipedia.org/wiki/Receiver_operating_characteristic



Each point on the curve represents a "threshold" value, so all curves will start at (0,0) and end at (1,1)

Metric vs loss

The problem

- ROC-AUC is the metric that was used in this contest
- We want to win, and therefore we will optimize our AUC, right?
- **The problem: Optimizers uses derivatives but the AUC is not differentiable**
- I.e. not every metric can serve as loss function (for example, this problem also exist For F1 score)

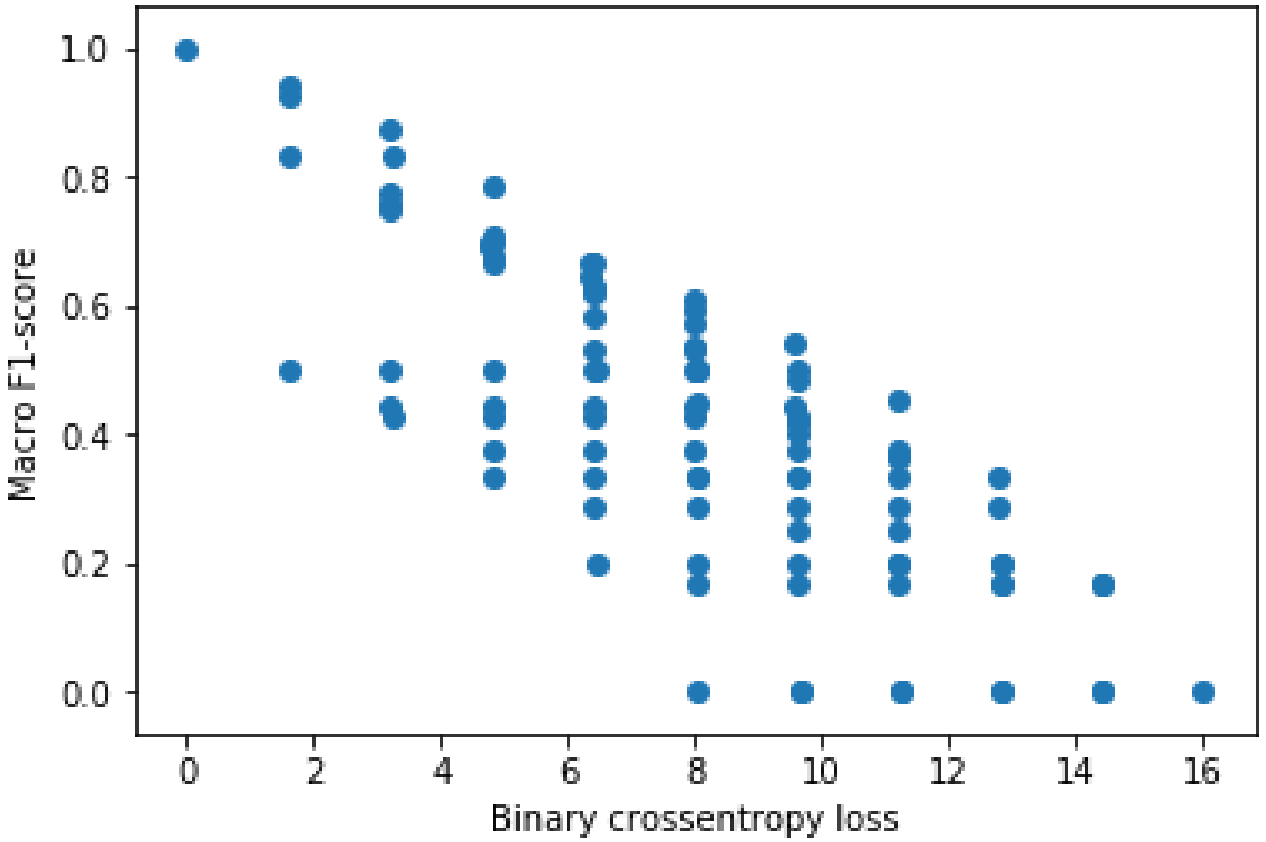
Why is this a problem?

- If we are using BCE, **we are training sub-optimal classifier** by definition
- Our data is very unbalanced, but the use of metric that integrates the TP and FP ratio could have overcome this unbalance

Metric vs loss

Why is this a problem?

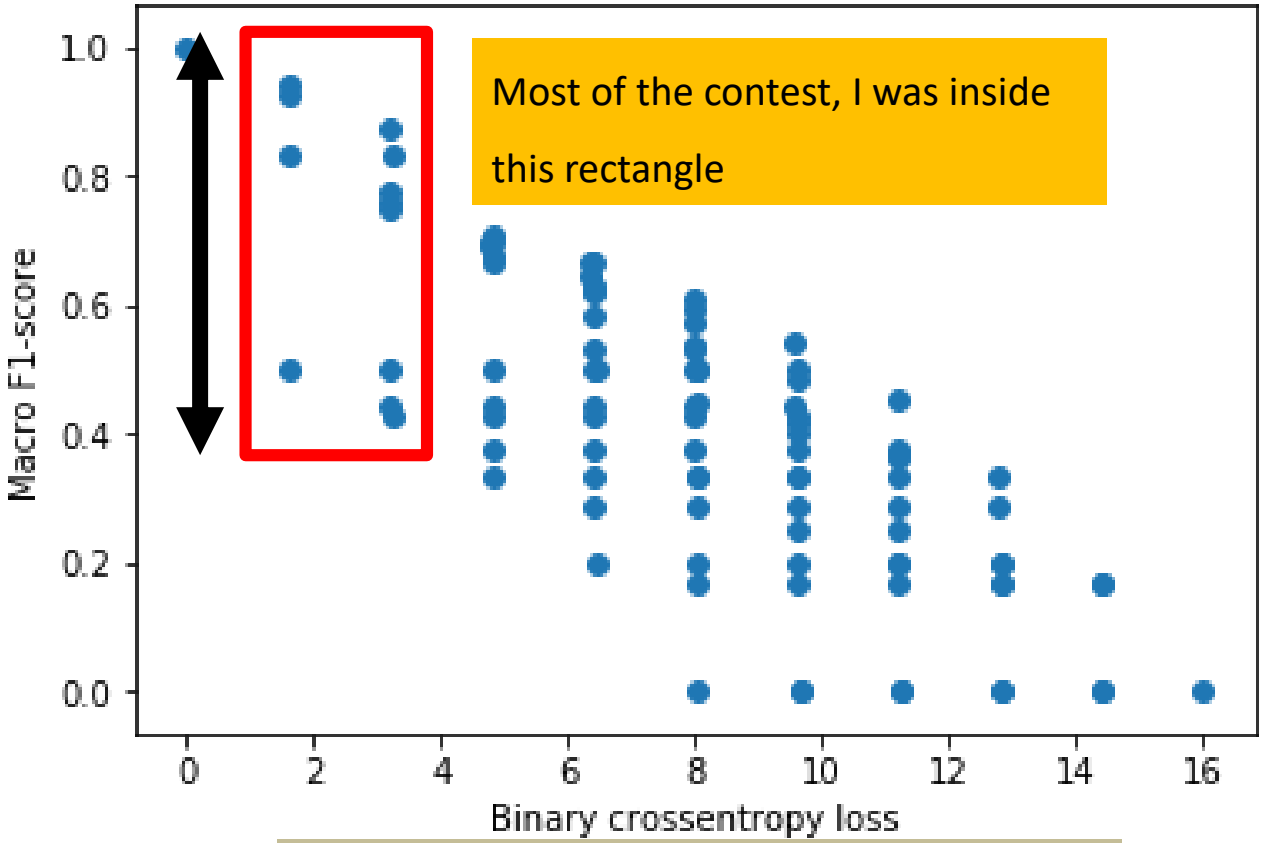
- **Not only we are training a sub optimal estimator, it's not even "consistent"!**



https://www.kaggle.com/rejpalcz/best-loss-function-for-f1-score-metric

Overlook the AUC

I would not recommend the AUC even as a metric for checkpoints saving (will be explained latter) or for making any decisions about the model because it is "unreliable"



https://www.kaggle.com/rejpalcz/best-loss-function-for-f1-score-metric

Alternatives?

- Some papers suggest some sort of approximations to get a **differentiable AUC.** Unfortunately, this is not really working and they end up with "reinventing" some new log-loss
- It seems like we are doomed to use "conventional" loss functions that will train sub-optimal classifiers, but **maybe we can still address the unbalance issue?**

References (bad ones…):

- https://gist.github.com/SuperShinyEyes/dcc68a08ff8b615442e3bc6a9b55a354
- https://www.kaggle.com/rejpalcz/best-loss-function-for-f1-score-metric
- Online AUC maximization, 2011, https://ink.library.smu.edu.sg/sis_research/2351/
- Learning Structured Models with the AUC Loss and Its Generalizations http://proceedings.mlr.press/v33/rosenfeld14.html

Yet another failed attempt

The values here are defined with probabilities rather than with thresholds

```
def MyF1_loss(pred, actual):
    epsilon = 1e-7
    tp = ((1-actual) * (1-pred)).sum()
    tn = ((actual) * (pred)).sum()
    fp = ((actual) * (1-pred)).sum()
    fn = ((1-actual) * (pred)).sum()
    precision = tp / (tp + fp + epsilon)
    recall = tp / (tp + fn + epsilon)
    f1 = 2 * (precision * recall) / (precision + recall + epsilon)
    return 1-f1
```

Part 5: Unbalanced data

- *Weights*
- *Focal loss*
- *Data augmentation*

Dealing with unbalance dataset

You are probably already familiar with:

- Weighted loss
- Sampling techniques
- Data augmentation

But I would like to talk about:

- Choosing the validation set
- Loss function

Cross validation test

Might be very trivial, but:

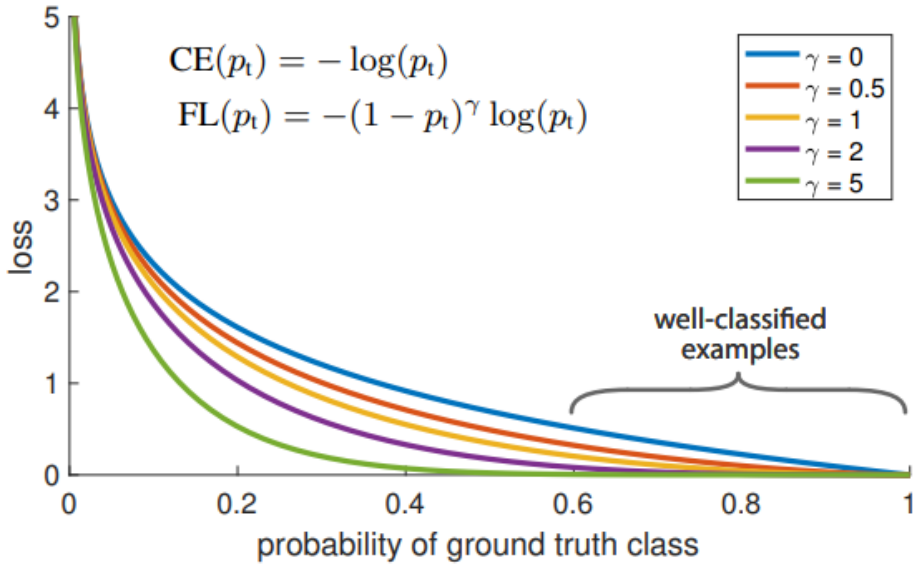
1. ***Your dev and test set should come from the same distribution***
2. ***Choose test set to reflect data you want to do well on***

Andrew Ng, "machine learning yearning"

MAFAT told us we will be tested on balanced set (50% animal , 50% human) – so one must choose a balanced CV set!

The loss function

- We can’t optimize exactly what we need (F1 score , AUC)
- **Weighted loss** is the first thing I tried, but it’s not enough
- Then I started to use **focal loss** (FAIR, 2018)



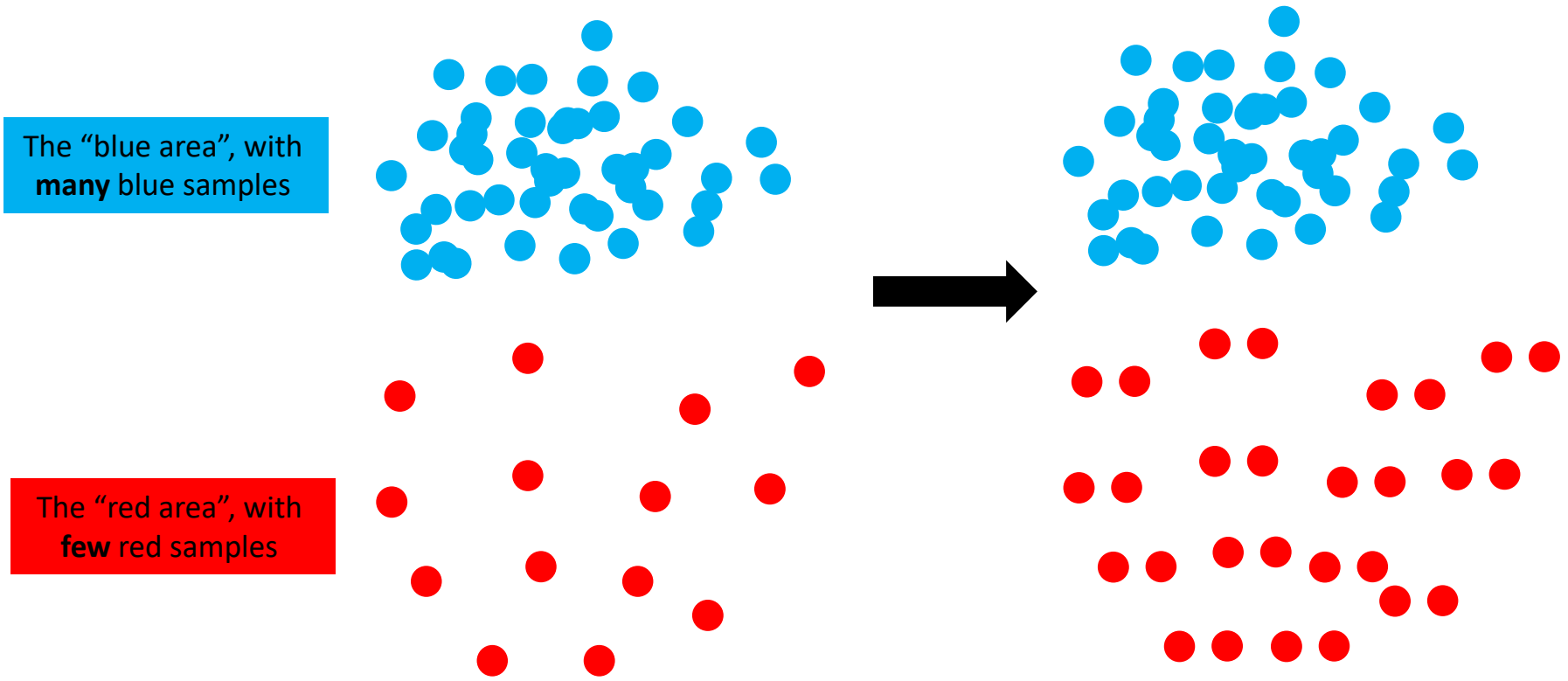
It seems that they took the old and good BCE loss, and just added a non-linear factor, so what's the trick?

Don’t forget!

```
loss = torch.mean(-alpha * ((1 - prediction) ** gamma) * torch.log(prediction+torch.tensor(1e-10)) * y
- (1 - alpha) * (prediction ** gamma) * torch.log(1 - prediction+torch.tensor(1e-10)) * (1 - y))
```

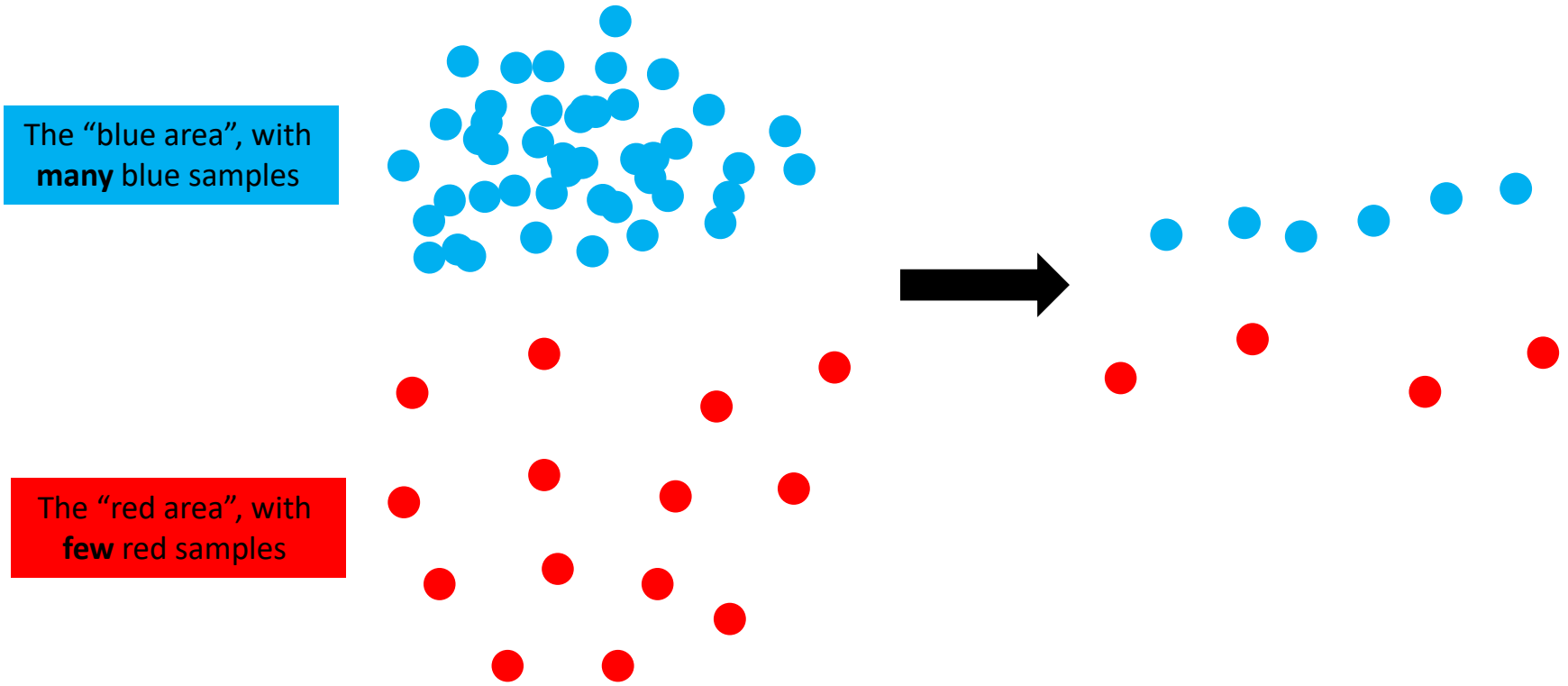
Why does it works better?

Weighted loss, could be thought of as **duplicating our entire minority data set** by some factor:



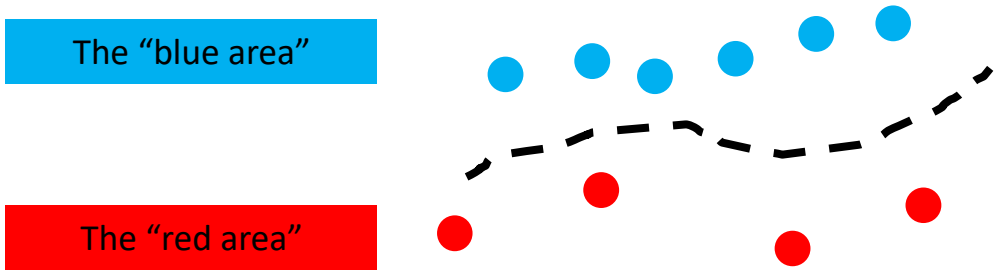
Why does it works better?

Focal loss does something better – **it is eliminating the effect of remote samples.**



My own interpretation

- **Although it is not written anywhere, I believe that this loss works just like SVM**. We are ending up with the “front line” samples, and the perceptron is drawn at the right place
- Moreover, once you use focal loss, **data augmentation is not that important anymore**



Data augmentation (Naive)

- Augmentation of the minority set can help us dealing with unbalanced data (anyway, it is always a “good practice”).
- Torchvision provides us **built-in augmentations**, but not all of them make sense in the scope of radar maps.
- At the end, I used only **time\Doppler flips** (and I am not sure about it…)

```
####### Add Augmented pictures for animals #########
if AugmentionFlag is True:
    VelocityFlip = np.flip(train_x[train_y==0,:,:],axis=1)
    TimeFlip = np.flip(train_x[train_y==0,:,:],axis=2)
    print("Adding augmantaion of %d VelocityFlip and %d TimeFlip" % (VelocityFlip.shape[0], TimeFlip.shape[0]))
    train_x = np.concatenate((train_x, VelocityFlip, TimeFlip), axis=0, out=None)
    train_y = np.concatenate((train_y, np.zeros(VelocityFlip.shape[0]+TimeFlip.shape[0],)), axis=0, out=None)
```

Data augmentation (Advanced)

- I tried to use this paper but without success: *"Deep Representation Learning on Long-tailed Data: A Learnable Embedding Augmentation Perspective":*

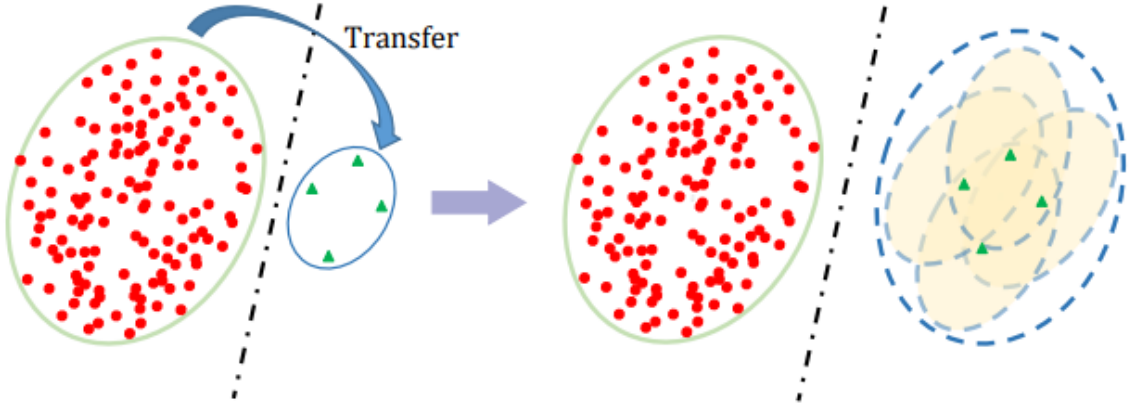
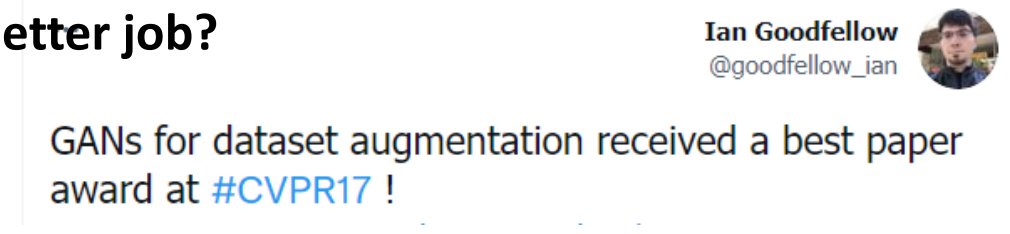


Figure 3. We transfer the intra-class angular distribution learned from the head class to the tail class.

- **Maybe GAN could have done a better job?**



1.11.2020

Part 6: Architectures

- *My resources*
- *Res-Net 18 can do it!*
- *EfficientNet*
- *A friend at need is a friend indeed*

My resources

Problem:

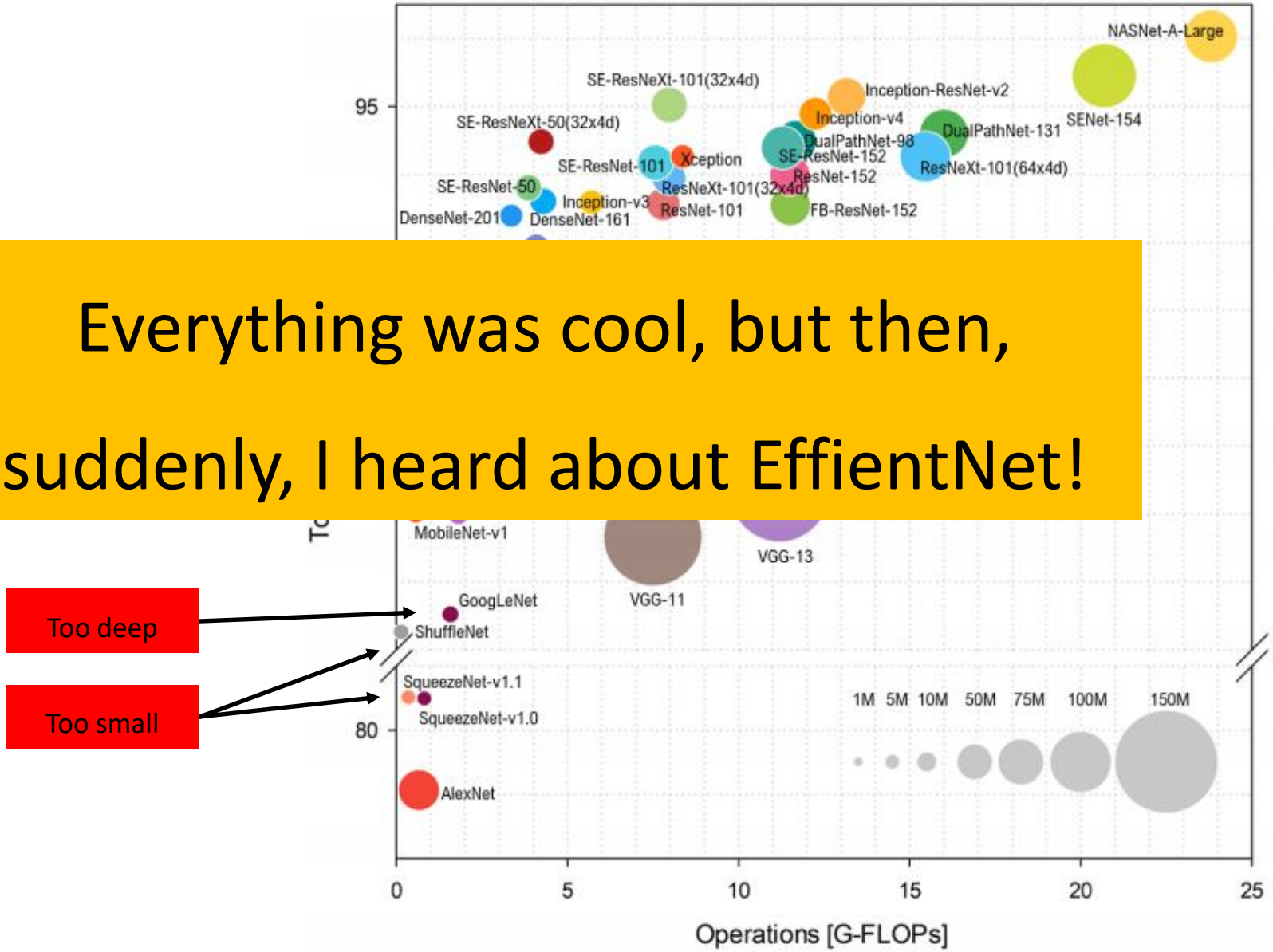
- Somewhere between 100-200 working hours
- **GTX-960** – 2015 SOTA!!

Solution:

- Networks with reasonable depth (running time)
- Networks with reasonable number of parameters (batch size)
- **With PyTorch one can try new architectures with few lines of code**

```
def Adjust_ResNet18():

    MyResNet18 = torchvision.models.resnet18(pretrained=True)
    weight = MyResNet18.conv1.weight[:,1,:,:].clone()

        MyResNet18.conv1 = nn.Conv2d(1, 64, kernel_size=(7, 3), stride=(2, 2), padding=(3, 3), bias=False) #changed number of channels from 3 to 1. The
        rest is the same

    with torch.no_grad():
                MyResNet18.conv1.weight = torch.nn.Parameter(weight.reshape(weight.shape[0], 1, weight.shape[1], weight.shape[2]))

    MyResNet18.fc = nn.Sequential(nn.Dropout(p=0.25), nn.Linear(in_features=512, out_features=64, bias=True),nn.Dropout(p=0.0),nn.Linear(in_features=64,
out_features=1, bias=True), nn.Sigmoid())
    MyResNet18 = MyResNet18.float()
    return MyResNet18
```

Architectures I tried



Everything was cool, but then,

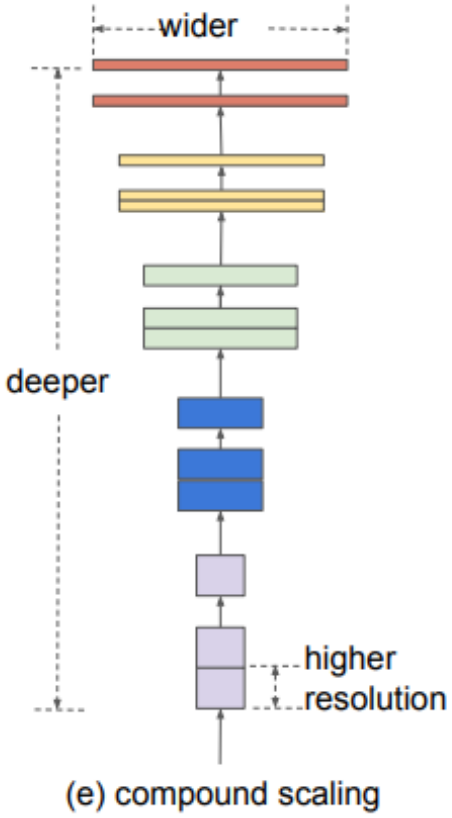
suddenly, I heard about EffientNet!

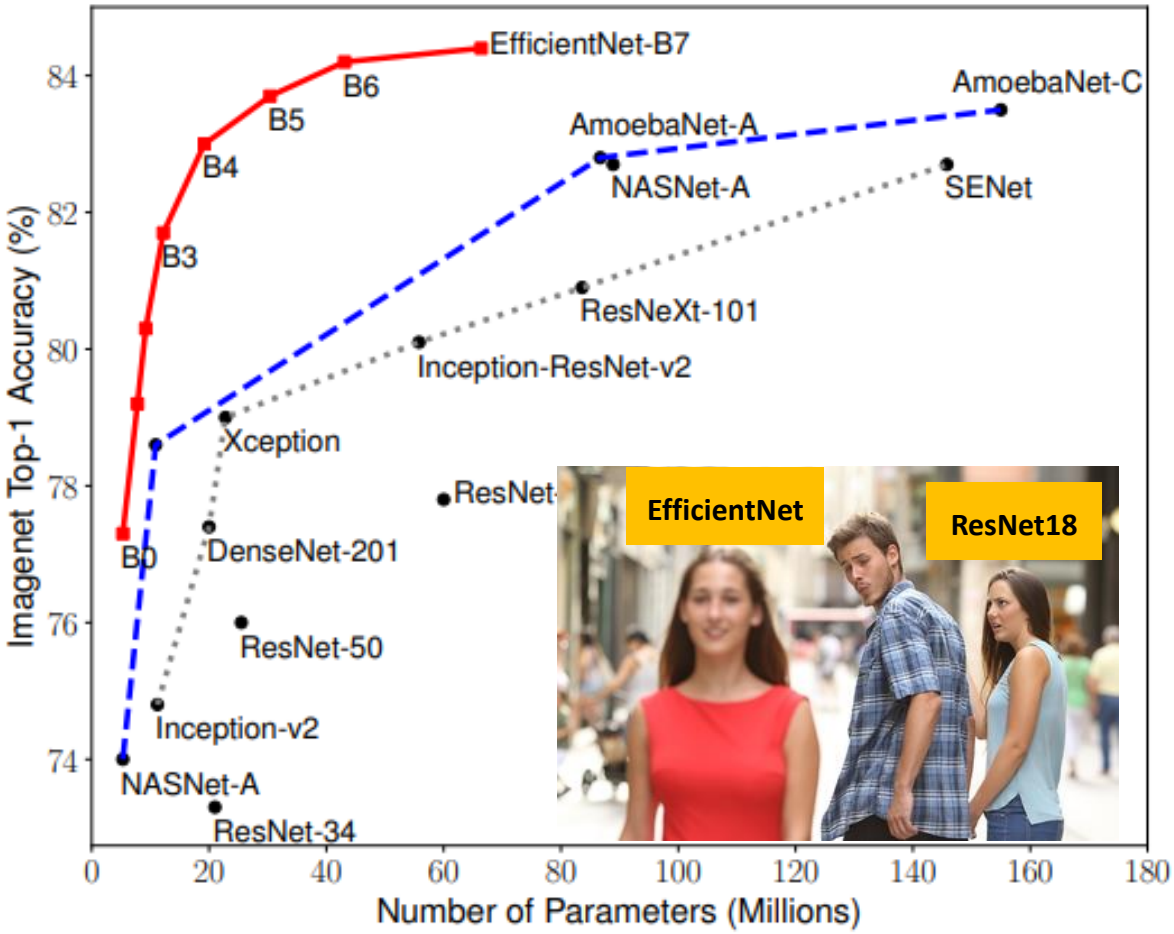
Too deep

Too small

EfficientNet 2019

It seems google have found the perfect net in terms of performance \ memory ratio, and I had to try it..





EfficientNet

- With a little help from a friend who owns GTX 1070 Ti, I tested EfficientNet at the last moment(!), just to find out that I got the same AUC as in ResNet18…

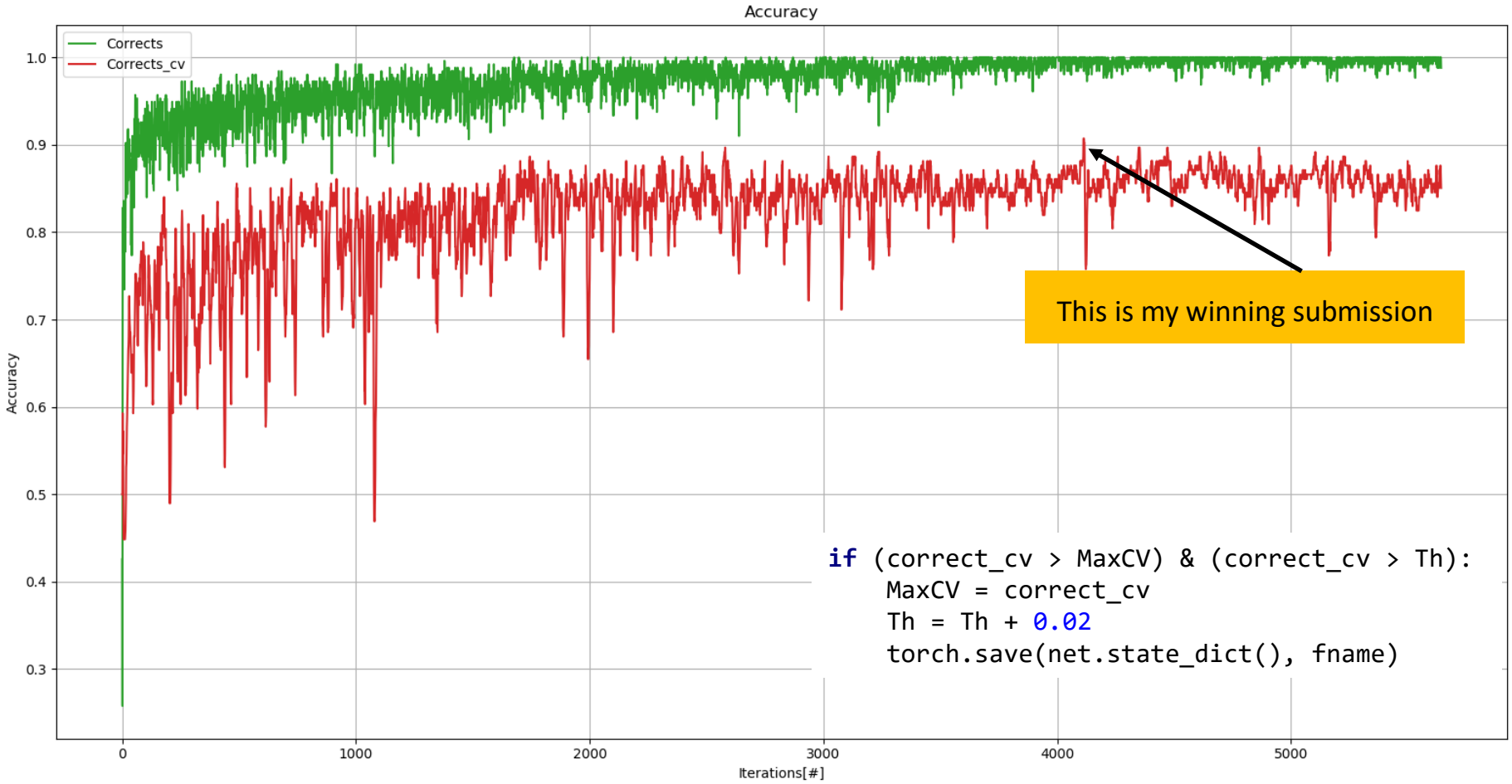
- **Conclusion 1: ImageNet is not the real world**
- **Conclusion 2: Networks architectures are not that important (**disclaimer: changing the kernel size with domain knowledge does helps)

Part 7: Tricks and tips

- *Checkpoints and randomness*
- *Stick with your CV set*
- *Ensembles*
- *Things that didn't worked*
- *Don't look at the score table!*

Checkpoints and randomness

Do “something” (like changing the learning rate) to add some noise, and save checkpoints if the CV score is high

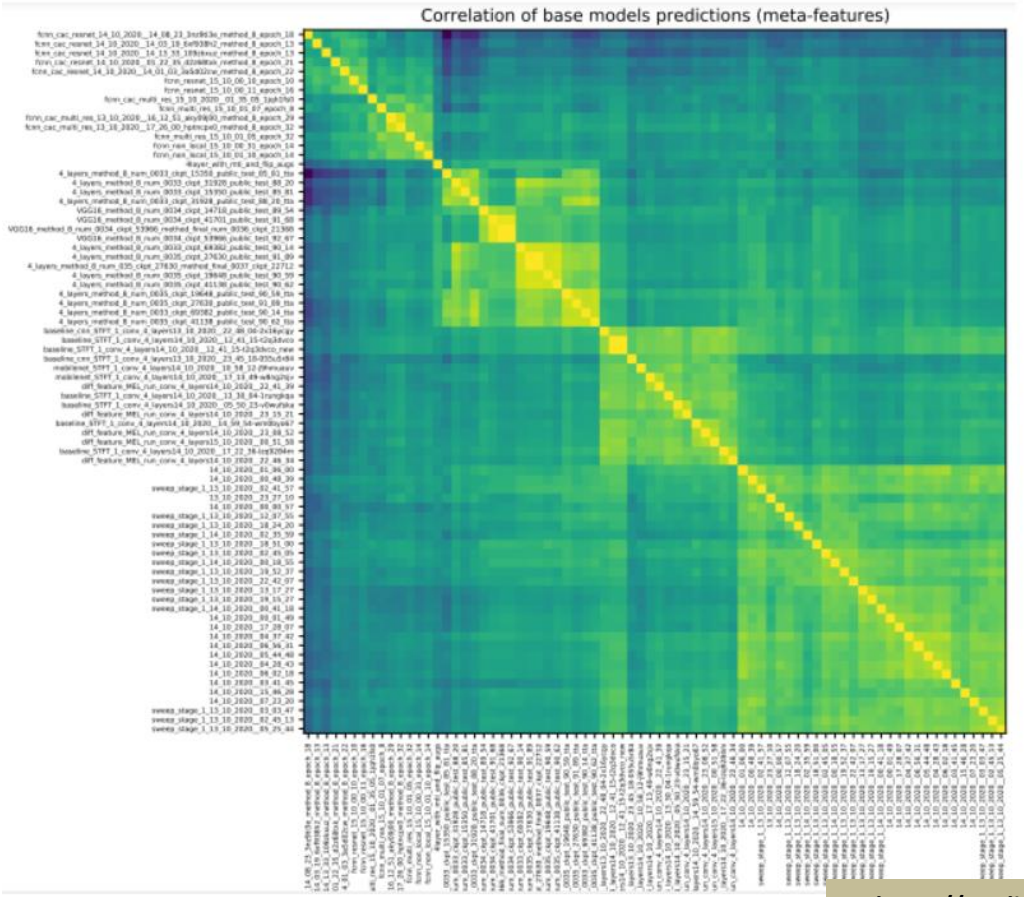


Stick with your CV set

- In two phase contests, the test set is released at the end of the first phase, and a common tactic is to use it as a new CV set
- I believe this is a mistake. For months we have been making decisions with our CV set. **If we replace it with a different one we won't know the impact of our new decisions.**

Ensembles

- If the classifiers are uncorrelated – O.K.
- If the classifiers are correlated – use only the best one



Predictions cross correlation matrix of our 76 models

https://medium.com/axon-pulse/mafat-radar-challenge-solution-by-axon-pulse

Things that didn’t work as expected

- Expanding the problem by classifying the background pictures as well
- Adding noise from background images as an augmentation
- Exploiting the signal phase
- Representation learning and triples Loss
- **Looking at the table too many times in a day!**

Thank you!