1. Who are you (mini-bio) and what do you do professionally?

My name is ___. I live in Russia, ___. I'm Ph.D in microelectronics field. Currently I'm working in Institute of Designing Problems in Microelectronics (part of Russian Academy of Sciences) as Leading Researcher. I often take part in machine learning competitions. I have extensive experience with GBM, Neural Nets and Deep Learning as well as with development of CAD programs in different programming languages (C/C++, Python, TCL/TK, PHP etc).

2. What motivated you to compete in this challenge?

I always wanted to try Conv3D neural net models and it was good competition to use them. On the basis of competition I prepared set of independent modules for fasten 3D neural net models development and training.

3. High level summary of your approach: what did you do and why?

The solution that gives the best result is divided into the following steps:

Stage 1 [region of interest (ROI) extraction]: For each video, only the part that is marked with an orange border in the video is extracted. To increase the speed of training, this part was saved as a numpy array in a zipped pkl file. Zip, allowed to significantly reduce the disk space requirements.

Generated files:

../modified_data/roi_parts/train/*.mp4.pklz
../modified_data/roi_parts/test/*.mp4.pklz

File with code: preproc_data/r01_extract_roi.py

Stage 2 [Preparation of 5KFold cross-validation]: In my solution, I did not use all available videos (1.4 TB). I downloaded the micro dataset, and also downloaded all the available videos from stalled == 1 and additionally a large number of videos with stalled == 0 label. In total, I had 51490 videos, where 49603 with stalled == 0 and 1887 videos with stalled == 1. I split them uniformly into 5 folds using the standard KFold function from the sklearn module.

Note 1: I think using the full set of videos should improve the result.

Note 2: You can skip this step and use my split file.

Generated files:

../modified_data/kfold_split_large_v2_5_42.csv

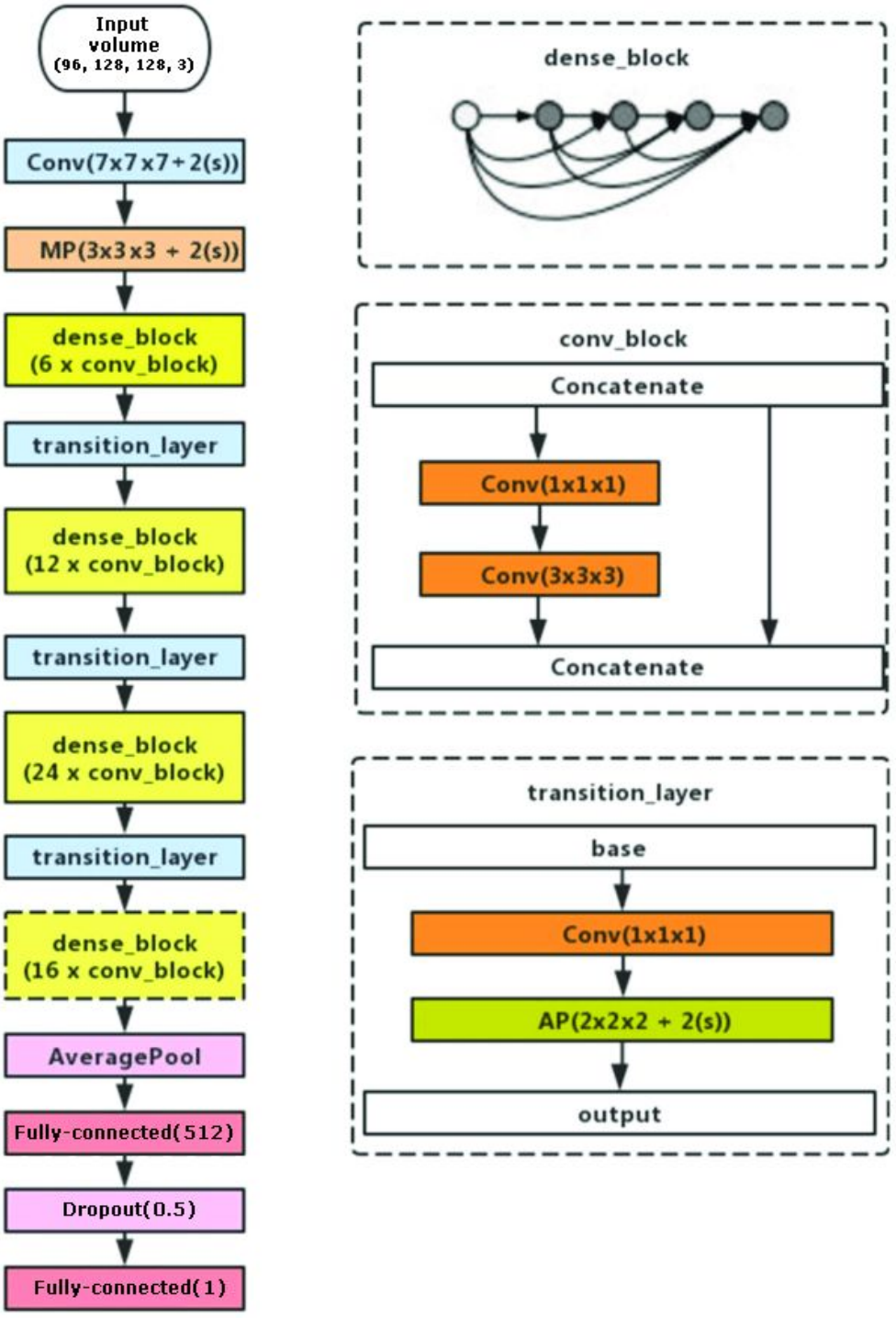
File with code: preproc_data/r03_gen_kfold_split.py

Stage 3 [training of DenseNet121 3D]: This is the longest and most time-consuming stage of calculations. As the main neural network, I used a neural network built on the principles of the DenseNet121 network, which is used to classify images, but in a 3D version. I also converted the ImageNet weights into a 3D version, so that I would not start from scratch.

At the output of the network after GlobalAveragePooling, an additional classification fully connected layer with 512 elements was added, followed by a Dropout layer with a probability of 0.5, to reduce overfitting. There was little data with stalled == 1 and it was likely that the network would start remembering the features of these videos instead of being properly generalized. At the very end, there was a fully connected layer with one neuron (since we only have one class) and sigmoid activation (see img. 1).

I used the previously generated 5KFold cross-validation. That is, I trained 5 models, separately for each fold.

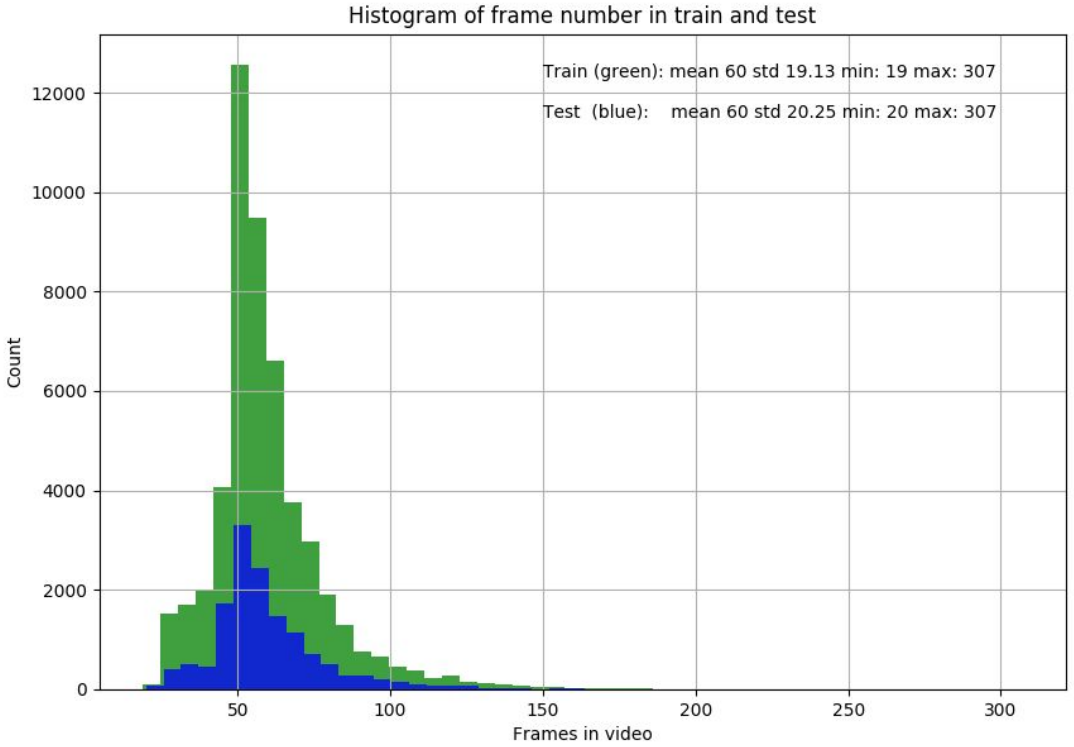
The input of the neural network was feed with 3D volumes with input shape (96, 128, 128, 3). The first dimension (96) here is time, the second dimension (128) is the height and the third dimension (128) is the width, the last dimension is the number of channels. The choice of this size was due to the following factors:

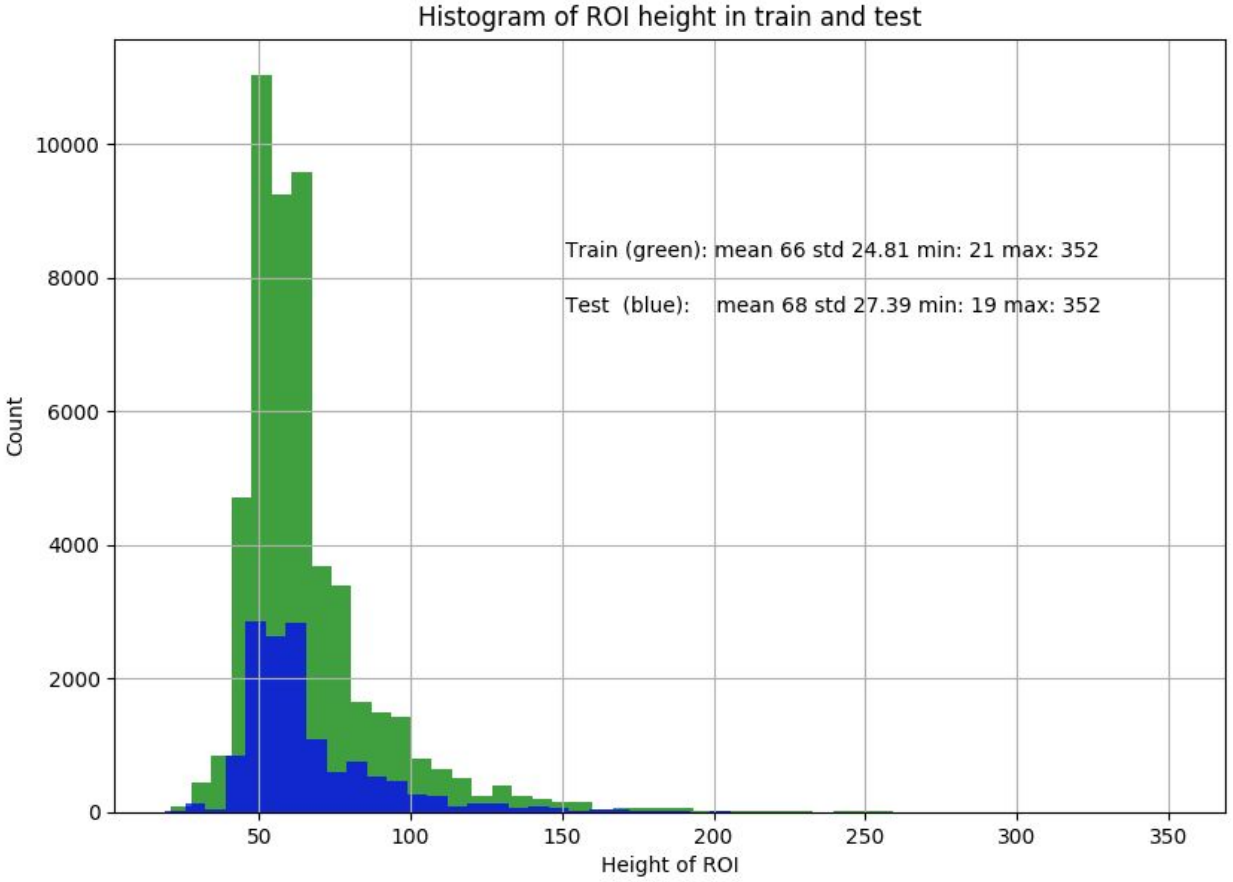


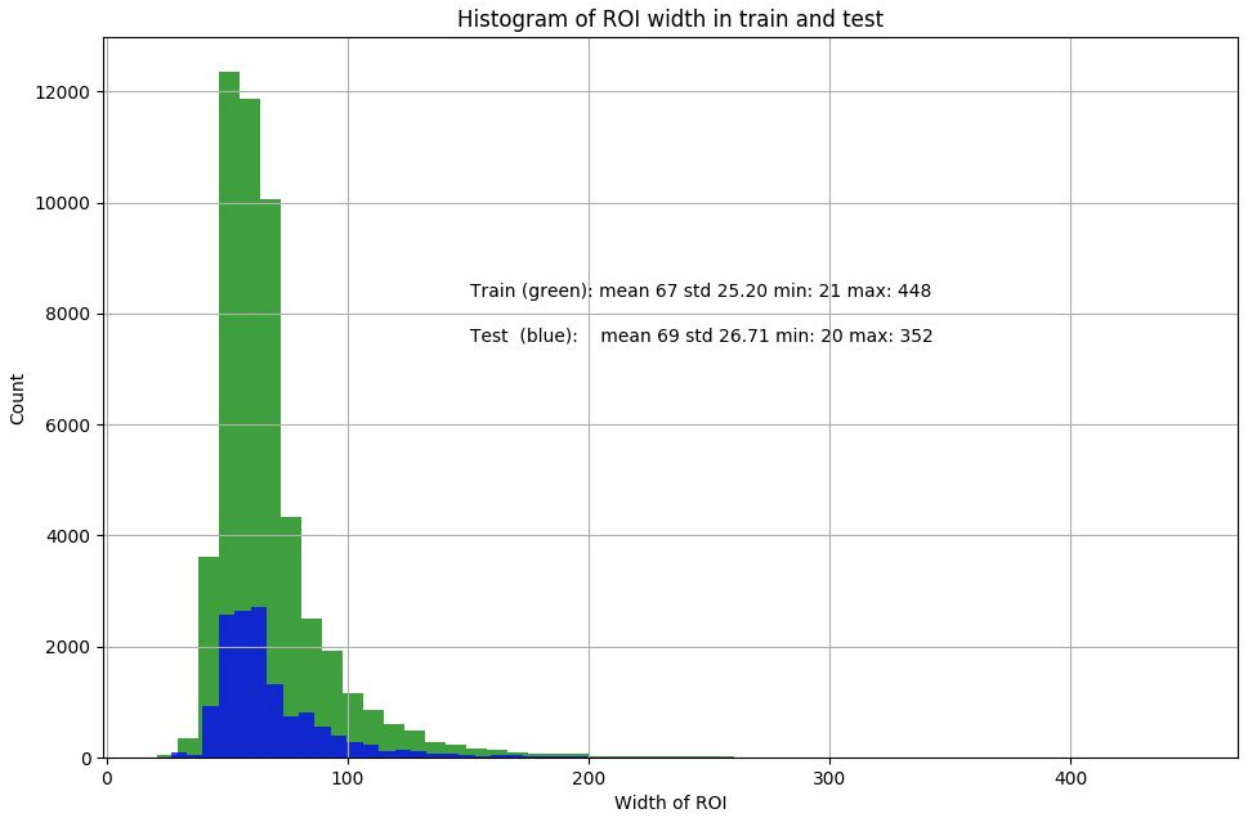
Img. 1 Structure of DenseNet121 3D

- This shape allowed using batch = 6 on one GPU, the shapes which were larger either did not fit into memory at all, either the batch size was too small.

- At the same time, after analyzing the size of the ROIs, I found out that most of the ROI video fits into this size, even without reducing the size of the cube (see images below).


Histogram of frame number in train and test
Train (green): mean 60 std 19.13 min: 19 max: 307
Test (blue): mean 60 std 20.25 min: 20 max: 307
Count
Frames in video
0
2000
4000
6000
8000
10000
12000
50
100
150
200
250
300


Histogram of ROI height in train and test
Train (green): mean 66 std 24.81 min: 21 max: 352
Test (blue): mean 68 std 27.39 min: 19 max: 352
Count
Height of ROI
0
2000
4000
6000
8000
10000
50
100
150
200
250
300
350



I ran the neural network training three times.

1) First run started with imagenet weights and used the MCC metric for validation. For validation, the ratio stalled == 1 to stalled == 0 was used as 1 to 1.

2) Second run started with the weights obtained at the previous step, but used the ROC AUC metric for validation. Also validation ratio changed: the ratio stalled == 1 to stalled == 0 is used as 1 to 4.

3) A week before the end of the competition, I decided to test the hypothesis that the test data contained only tier1 videos (and this hypothesis turned out to be correct (?)). So I excluded all non-tier1 videos from the training and validation. And I finished training the neural network, starting with the weights obtained at the second stage. The validation also maximized the ROC AUC metric. For validation, the ratio stalled == 1 to stalled == 0 is used as 1 to 10 (which is closer to the ratio on LeaderBoard).

Note 1: It is possible that only one run will be enough for training (according to scheme number 3), but I did not have time to test this option.

Note 2: Since in stages 2 and 3 we start with the weights of the previous stage, the training goes much faster than the first stage.

General settings for the training process:

1) Loss function: binary_crossentropy
2) Optimizer: Adam. Initial learning rate: 0.0001. LR multiplied on 0.95 after 3 epochs without improvements.
3) Each batch in average consists of 25% of (stalled == 1) and 75% of (stalled == 0) to partially capture large imbalance of dataset classes.
4) Each batch created randomly and epoch is just 1000 batches (so epoch is not related to dataset size).

To artificially increase the size of the dataset (this is especially critical for videos with stalled == 1), a large set of augmentations was used. For this, the volumentations library was used, partially rewritten to increase the speed of processing, and also some new useful augmentations were added.

The following set of augmentations was used:

- Rotate((-10, 10), (0, 0), (0, 0), p=0.3) – rotation by a random angle -10 - 10 degrees only along the zero axis.
- ElasticTransform((0, 0.25), interpolation=2, p=0.1) – random elastic deformation
- RandomCropFromBorders(crop_value=0.15, p=0.4) – from 0 up to 15% of the data is randomly removed from each side.
- RandomDropPlane(plane_drop_prob=0.1, axes=(0, 1, 2), p=0.3) – single planes randomly removed from the volume
- Resize(patch_size, interpolation=1, always_apply=True, p=1.0) – each volume is converted to the size of the neural network input using bilinear interpolation (scipy.zoom)
- Flip(0, p=0.5) – random volume reflection along the zero axis
- Flip(1, p=0.5) – random volume reflection along the first axis
- Flip(2, p=0.5) – random volume reflection along the second axis
- RandomRotate90((1, 2), p=0.5) – random rotation by an angle multiple of 90 degrees around the zero axis
- GaussianNoise(var_limit=(0, 5), p=0.2) – adding random Gaussian noise
- RandomGamma(gamma_limit=(0.5, 1.5), p=0.2) – random gamma change

Generated files: neural net model *.h5 files in following directories:

../models/net_v13_3D_roi_regions_densenet121_r31_train_3D_model_dn121.py_kfold_split_large_v2_5_42/

../models/net_v14_d121_auc_large_valid_r31_train_3D_model_dn121.py_kfold_split_large_v2_5_42/

../models/net_v20_d121_only_tier1_finetune_r31_train_3D_model_dn121.py_kfold_split_large_v2_5_42/

Files with code:

net_v13_3D_roi_regions_densenet121/r31_train_3D_model_dn121.py

net_v14_d121_auc_large_valid/r31_train_3D_model_dn121.py

net_v20_d121_only_tier1_finetune/r31_train_3D_model_dn121.py

Stage 4 [inference for test set]: At the last stage, five models received after training (by the number of folds):

- Reduced by BatchNorm Fusion operation (using KITO)
- Combined into one model with a common input

Each test video is then run using this combined model. The test time augmentation (TTA) approach is used, where all 8 possible video reflections on all axes are fed into neural network and then the prediction is averaged.

The MCC metric was used on LB, and in the problem it was required to predict not probabilities, but binary stalled label. The selection of the optimal THR value was carried out using LeaderBoard. In my case the optimal value for threshold was about 0.7.

Generated files:

../features/DN121_net_v20_d121_only_tier1_finetune_test.csv – raw probabilities

../subm/submission.csv – final submission file

File with code: net_v20_d121_only_tier1_finetune/r42_process_test.py

4. Copy and paste the 3 most impactful parts of your code and explain what each does and how it helped your model.

Code sample 1

preproc_data/r01_extract_roi_parts.py:

```
def extract_roi_parts(type):
    out_path = OUTPUT_PATH + 'roi_parts/'
    if not os.path.isdir(out_path):
        os.mkdir(out_path)
```

```
    out_path = out_path + '{}/'.format(type)
    if not os.path.isdir(out_path):
        os.mkdir(out_path)

    files = glob.glob(INPUT_PATH + '{}/*.mp4'.format(type))
    print(len(files))
    res = dict()
    frames_in_video = []
    for f in files:
        cache_path = out_path + os.path.basename(f) + '.pklz'
        if os.path.isfile(cache_path):
            continue
        cap = cv2.VideoCapture(f)
        length = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
        current_frame = 0
        frame_list = []
        print('ID: {} Video length: {}'.format(os.path.basename(f), length))
        min_x = 10000000
        min_y = 10000000
        max_x = -10000000
        max_y = -10000000
        while (cap.isOpened()):
            ret, frame = cap.read()
            if ret is False:
                break
            th = cv2.inRange(frame, (9, 13, 104), (98, 143, 255))
            points = np.where(th > 0)
            p2 = zip(points[0], points[1])
            p2 = [p for p in p2]
            rect = cv2.boundingRect(np.float32(p2))
            frame_list.append(frame.copy())
            if rect[1] < min_x:
                min_x = rect[1]
            if rect[0] < min_y:
                min_y = rect[0]
            if rect[1] + rect[3] > max_x:
                max_x = rect[1] + rect[3]
            if rect[0] + rect[2] > max_y:
                max_y = rect[0] + rect[2]

        frame_list = np.array(frame_list, dtype=np.uint8)
        frame_list = frame_list[:, min_y:max_y, min_x:max_x, :]
        # show_image(frame_list[0])
        save_in_file(frame_list, cache_path)
        print(frame_list.shape, min_x, max_x, min_y, max_y)
```

This code extracts only ROI part of volume. Training on full videos is waste of time and resources.

Code sample 2

net_v20_d121_only_tier1_finetune/r31_train_3D_model_dn121.py:

```
def get_augmentation_full(patch_size):
    return Compose([
        Rotate((-10, 10), (0, 0), (0, 0), p=0.3),
        ElasticTransform((0, 0.25), interpolation=2, p=0.1),
```

```
        RandomCropFromBorders(crop_value=0.15, p=0.4),
        RandomDropPlane(plane_drop_prob=0.1, axes=(0, 1, 2), p=0.3),
        Resize(patch_size, interpolation=1, always_apply=True, p=1.0),
        Flip(0, p=0.5),
        Flip(1, p=0.5),
        Flip(2, p=0.5),
        RandomRotate90((1, 2), p=0.5),
        GaussianNoise(var_limit=(0, 5), p=0.2),
        RandomGamma(gamma_limit=(0.5, 1.5), p=0.2),
    ], p=1.0)
```

This function return set of augmentations applied to each video, which allows to greatly increasing the train dataset.

Code sample 3

net_v20_d121_only_tier1_finetune/r42_process_test.py:

```
def get_cube_pred_v2(model, cube, preproc_input):
    valid_aug = get_augmentation_valid(SHAPE_SIZE[:3])
    data = {'image': cube}
    aug_data = valid_aug(**data)
    cube = aug_data['image']

    cubes_to_pred = []
    for i in range(8):
        cout = np.zeros(SHAPE_SIZE, dtype=np.uint8)
        if i == 0:
            cout[:cube.shape[0], :cube.shape[1], :cube.shape[2],
:cube.shape[3]] = cube.copy()
        elif i == 1:
            cout[:cube.shape[0], :cube.shape[1], :cube.shape[2],
:cube.shape[3]] = cube[::-1, :, :].copy()
        elif i == 2:
            cout[:cube.shape[0], :cube.shape[1], :cube.shape[2],
:cube.shape[3]] = cube[:, ::-1, :].copy()
        elif i == 3:
            cout[:cube.shape[0], :cube.shape[1], :cube.shape[2],
:cube.shape[3]] = cube[:, :, ::-1].copy()
        elif i == 4:
            cout[:cube.shape[0], :cube.shape[1], :cube.shape[2],
:cube.shape[3]] = cube[::-1, ::-1, :].copy()
        elif i == 5:
            cout[:cube.shape[0], :cube.shape[1], :cube.shape[2],
:cube.shape[3]] = cube[:, ::-1, ::-1].copy()
        elif i == 6:
            cout[:cube.shape[0], :cube.shape[1], :cube.shape[2],
:cube.shape[3]] = cube[::-1, :, ::-1].copy()
        elif i == 7:
            cout[:cube.shape[0], :cube.shape[1], :cube.shape[2],
:cube.shape[3]] = cube[::-1, ::-1, ::-1].copy()
        cubes_to_pred.append(cout.copy())

    cubes = np.array(cubes_to_pred)
    cubes = preproc_input(cubes.astype(np.float32))
    preds = model.predict(cubes)
    preds = np.array(preds).mean(axis=1)
```

```
    pred = np.squeeze(preds)
    return pred
```

This is the code for Test Time Augmentation. In this function, all eight possible reflections applied to single volume. All this allows us to get more precise predictions of the neural network after averaging. This function used for all neural networks at the inference stage.

5. Please provide the machine specs and time you used to run your model.

- CPU (model): AMD Ryzen Threadripper 2920X 12-Core
- GPU (model or N/A): NVIDIA GeForce GTX 1080 Ti 11 GB
- Memory (GB): 128 GB
- OS: Windows 10
- Train duration: ~7-8 days (time for run at single GPU. Folds can be trained in parallel on several GPUs. It will speed up training x5 times)
- Inference duration: ~7 hours

6. What are some other things you tried that didn't necessarily make it into the final workflow (quick overview)?

- I found out that test set probably contains only tier1 data at final week of competition. So, almost all my previously trained models became too weak. With training on all data I was able to achieve only 0.81-0.82 MCC on public LB, while fine-tuning DenseNet121 using only tier1 data gave me 0.84-0.86 MCC.
- I tried to optimize ROC AUC directly with special Experimental binary cross entropy with ranking loss function: https://gist.github.com/jerheff/8cf06fe1df0695806456 but it didn't give me noticeable difference at local validation.

7. Did you use any tools for data preparation or exploratory data analysis that aren't listed in your code submission?

All analysis and data preparation were done in Python. I didn't use any outside tools and didn't make any additional annotation in training dataset.

8. How did you evaluate performance of the model other than the provided metric, if at all?

- At first I run validation after each epoch and calculate MCC at each threshold starting from 0.01 up to 0.99 with step 0.01 and uses best found MCC as score. It's actually not very stable metric. It's varies a lot from step to step.
- After I switch to ROC AUC metric for validation. It didn't require THR to calculate and measure overall quality of predictions.

My local MCC on validation didn't match the LB, but direction was the same e.g. better MCC on local gave better MCC on Leaderboard. To output binary values I mostly used LB for checking. At the end I found out that optimal number of stalled == 1 on LB was around 600-700.

9. Anything we should watch out for or be aware of in using your model (e.g. code quirks, memory requirements, numerical stability issues, etc.)?

The code for training neural nets requires powerful GPUs like NVIDIA GTX 1080 Ti. And it's computationally expensive. All other parts of code should be run fast and with lower requirements. I'd recommend using powerful CPU because augmentations requires a lot of CPU power. Also SSD hard drive is critical, since during neural net training code read many data from disc and it can be bottleneck in case of default HDD. Minimum RAM memory requirement for training is 64 GB. It's possible to use several GPUs for training different folds.

Code is split in independent parts, has logical structure and print useful debug information. So it should be easy to run on your side.

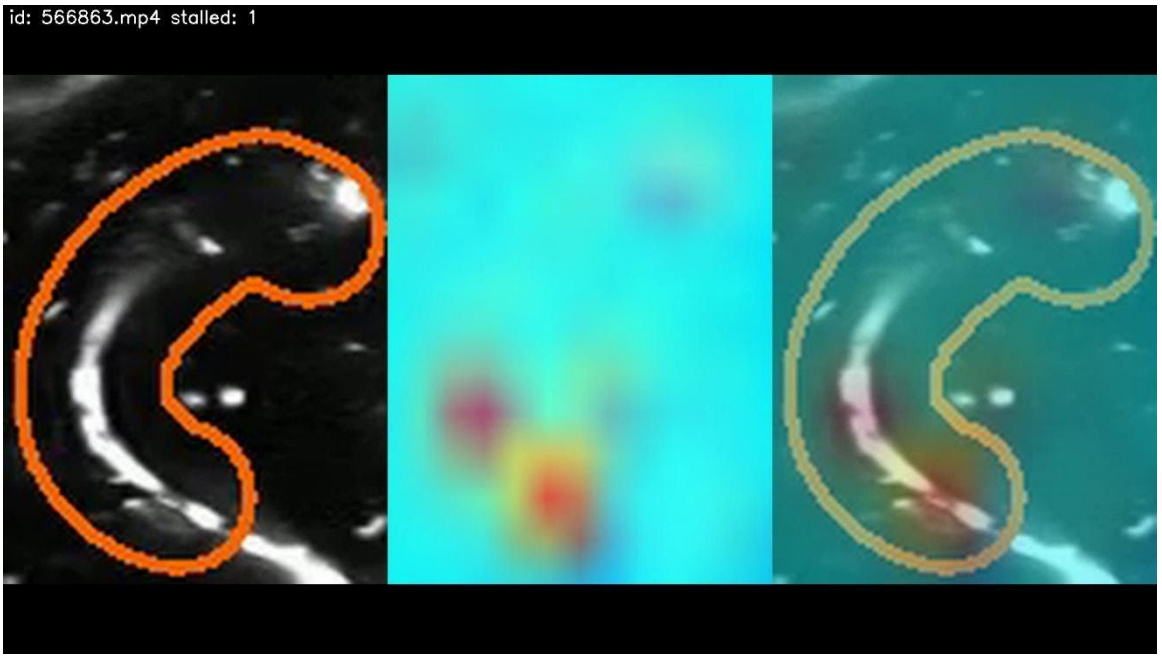
10. Do you have any useful charts, graphs, or visualizations from the process?

It's possible to create HeatMaps using latest BatchNormaliztion layer before GlobalAvgPooling or GlobalMaxPooling. We are interested in places which:

1) Logits mostly define value after where maxAvgPooling – we can use mean() value here
2) Logits which changes often - we can use std() there.
3) Also it's possible to check where the largest values appear, so we can use max() for it.

I created video which is available on youtube:

https://youtu.be/k7s5DCzvKj8



There are several stalled == 0 and stalled == 1 videos used. Frame of each video is divided into 3 parts left to right: original video (only ROI part used), heatmap, heatmap projected on the original video.

Blue – std, green – max, red – mean.

11. If you were to continue working on this problem for the next year, what methods or techniques might you try in order to build on your work so far? Are there other fields or features you felt would have been very helpful to have?

1) I'd definitely try to use full 1.4 TB dataset
2) I didn't explore more efficient neural net models. Probably EfficientNet3D could work here.
3) Probably there are some additional augmentations can be applied to increase quality of model.
4) Ensembles of several neural net models most likely increase the quality of model.
5) It's probably possible to use model which trained on 2D frames and then second model which uses predictions from each frame to generate overall prediction for full video. I didn't try this approach in this competition.