



Medical data analysis

Emil, Mads, Thomas & Yulii





Heart disease
Unbalanced data

Data

Heart disease

dtypes: booleans, strings, integers, and floats

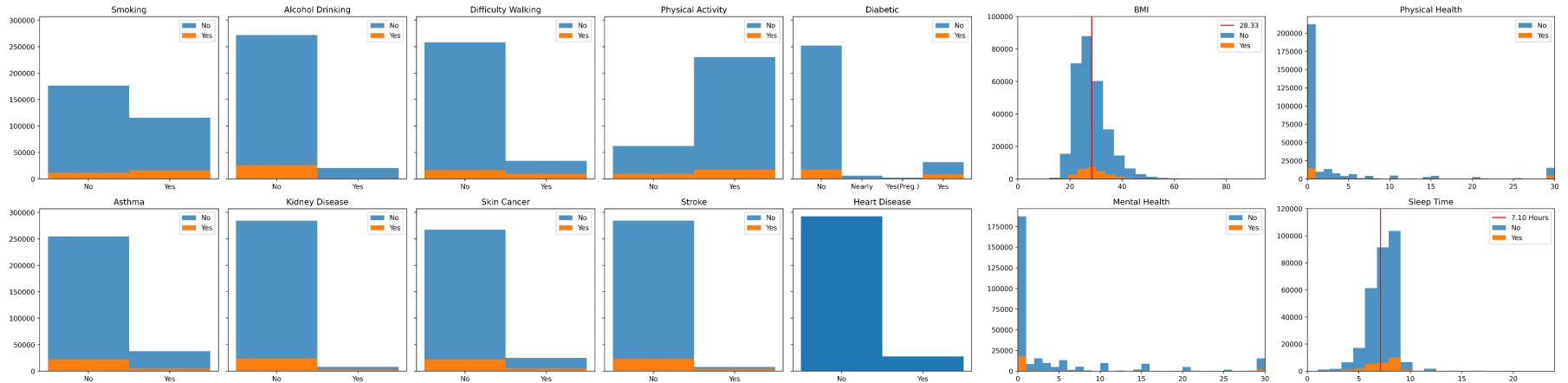
- **Cleaned:**
 - 319.795 data points (18 features)
 - 8,6 % positive cases
- **Raw:**
 - 401.958 data points (279 features)
 - 10,6 % positive cases

Clean data

- Quick classifier 91.4 % accuracy (8.6 % cases)
- Over- and under sampling → Drop to ~71 % accuracy

Clean data

➤ Results: Complete feature overlap



Raw data

- Easy part
 - Categorical variables, binary choice → Numerical variables
 - Remove filling data (16 features)
- Tricky part
 - NAN value ~48 % (260 features) → 11 % (132 features)

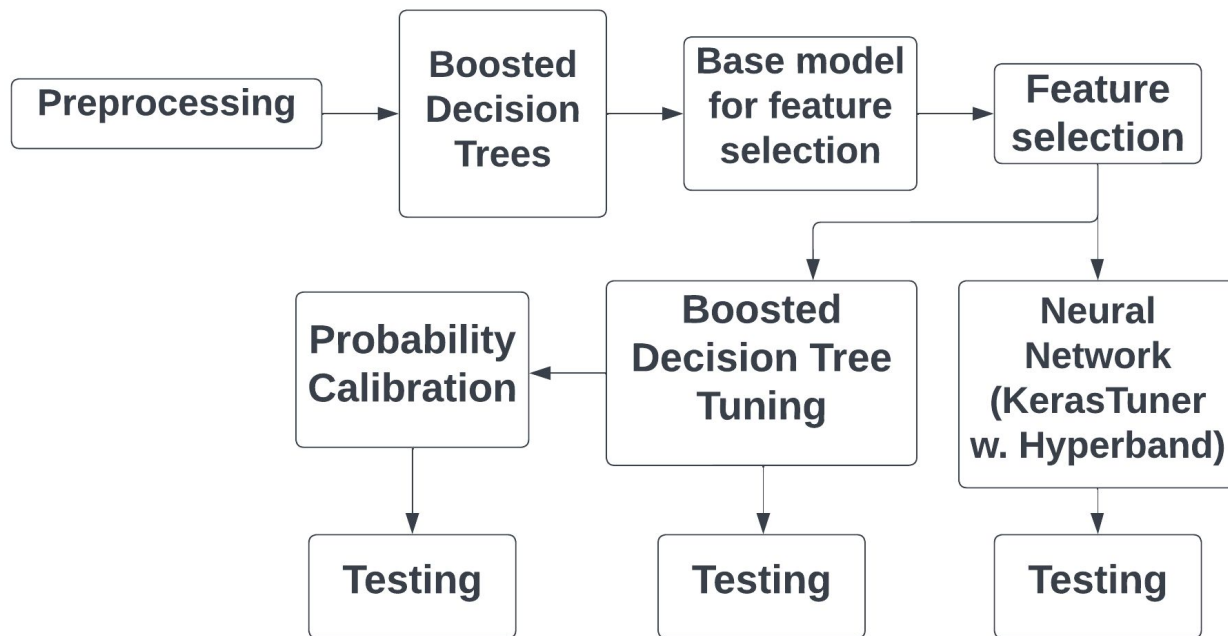
Challenges

- Raw data set contains a lot of confounders, a lot of manual inspection is required to make an unbiased model.
- Choice of metric is important, true negatives are undiagnosed patients and true positives are costly due to the costs of heart disease tests in the U.S.

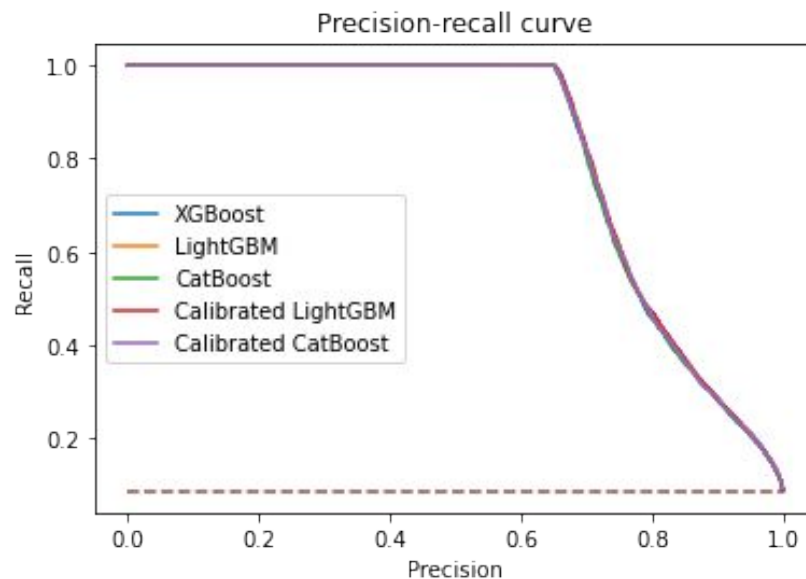
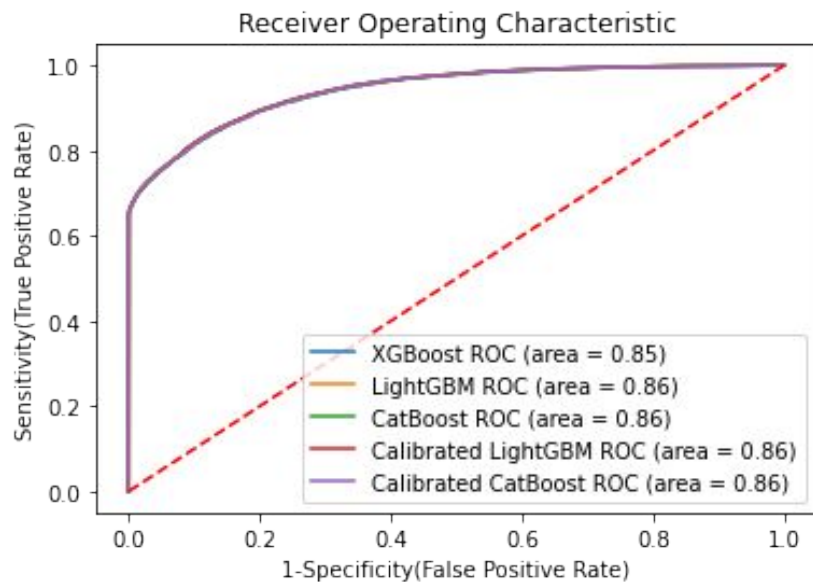
Methods

- Boosted decision trees turned out to be the best
 - CatBoost, LightGBM and XGBoost
 - Keras HyperTune (Model from scratch)
- Class weight balancing, SMOTE, and undersampling
- The average precision score turned out to be the best evaluation metric
 - Balanced Accuracy, F1 and F2 scores = not good enough
- Feature selection from SHAP and Feature Importance.
- Two turn HP optimization initial one + final micro tuning

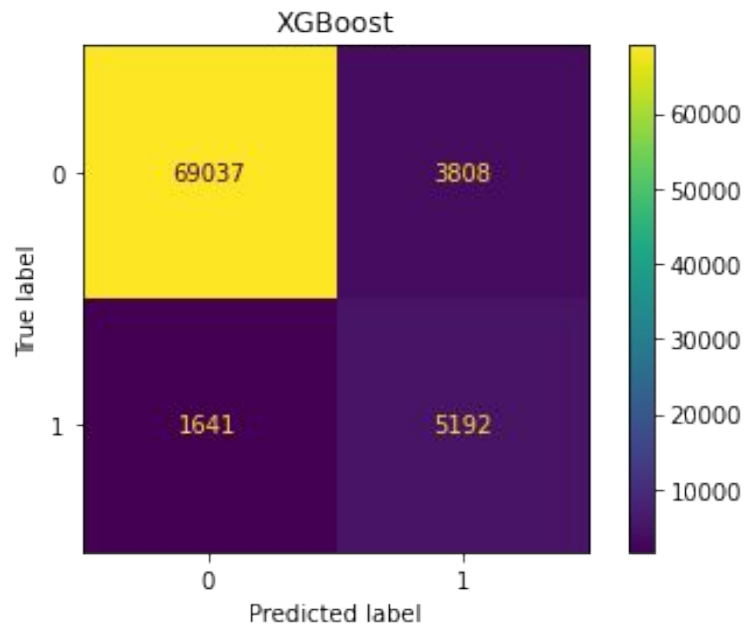
Main Method Framework



Results: ROC- & Precision-Recall Curve





Best model: XGBoost



- Accuracy score: 0.93
- F1-score: 0.65
- Recall score: 0.75
- Precision score: 0.58

Conclusion

- Undersampling was the preferred method, however introduces bias
- Difficult dataset for predictive analysis
 - Too much class overlap
 - Requires a lot of manual inspection and cleaning
 - Difficult to reduce the number of false negatives without increasing false positives.
 - Confounders
- Could be used for descriptive analysis.



COVID-19 predictions CNN

Data

8-bit grayscale images [256x256]

- 33.920 chest X-Ray images
 - 35,2 % **COVID-19** cases
 - 33,2 % **Non-COVID** cases (pneumonia)
 - 31,5 % **Healthy** cases (normal)
- with premade masks for lungs



Multi-classification problem: divide the images into the 3 classes

Chest X-Rays

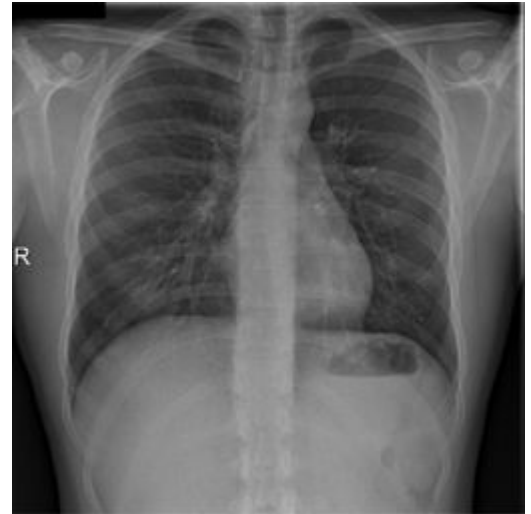
Normal



Pneumonia



COVID-19



Chest X-Rays - using masks

Normal



Pneumonia



COVID-19



Data splits

Train

21715 images - 64 %

- 6849 **Normal** 32 %
- 7208 **Non-COVID** 33 %
- 7658 **COVID-19** 35 %

Validation

5417 images - 16 %

- 1712 **Normal** 32 %
- 1802 **Non-COVID** 33 %
- 1903 **COVID-19** 35 %

Test

6788 images - 20 %

- 2140 **Normal** 32 %
- 2253 **Non-COVID** 33 %
- 2395 **COVID-19** 35 %

CNNs for image classification

A simple CNN:

input → 2 × (Conv2d → ReLU → MaxPool2d → Dropout) → Conv2d → ReLU → Flatten → Linear → **output**

May be improved upon in several ways but serves as a baseline.

NB: Images are normalized by subtracting the *global* mean across images and dividing by the standard deviation (see appendix)

CNNs for image classification

Reasons to use pre-trained models

- Superior network architecture
- Trained on millions of images from the ImageNet* database
- General image recognition features will 'hopefully' transfer to our data set, meaning better initial weights than random and faster training

What we chose**

VGG-19: 144M parameters

ResNet-152: 60.2M parameters

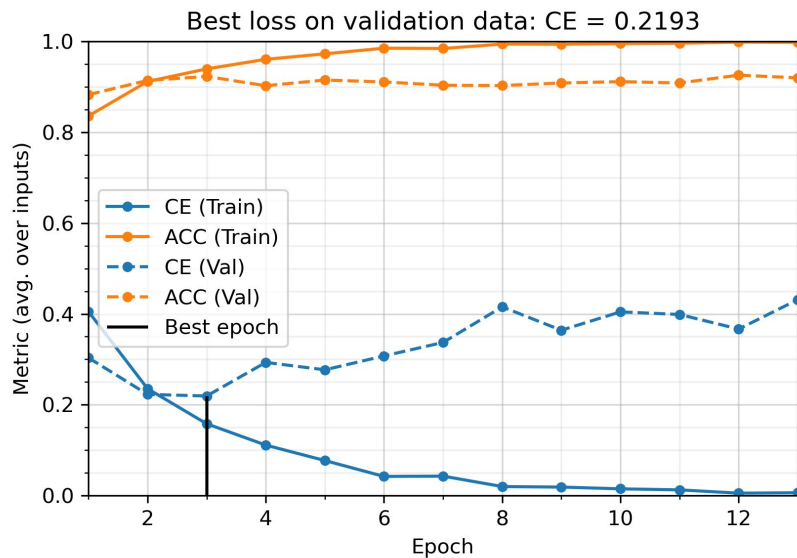
ConvNeXt-Tiny: 28.6M parameters

EfficientNet-B4: 19.3M parameters

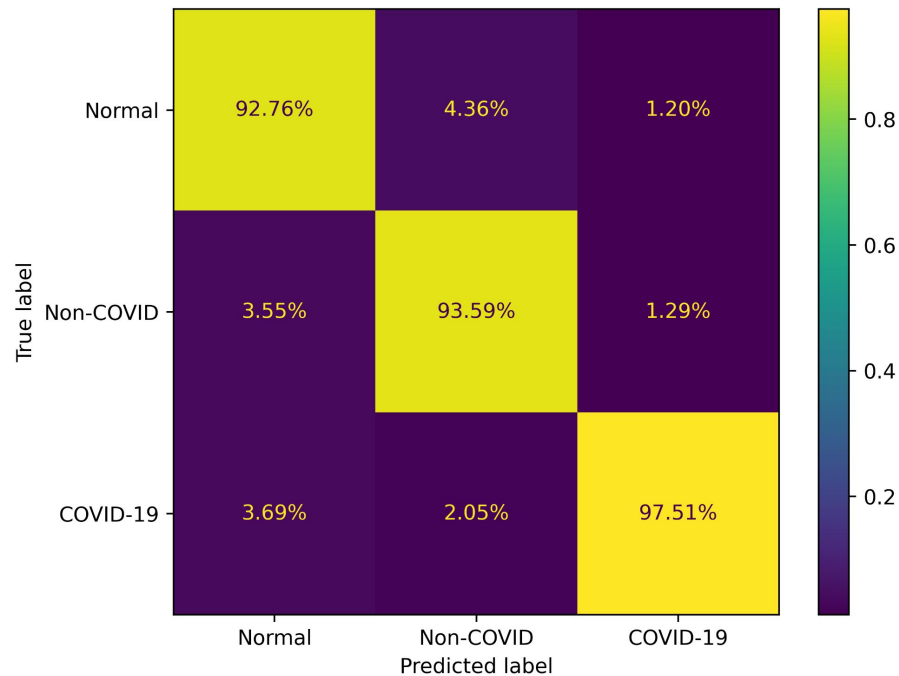
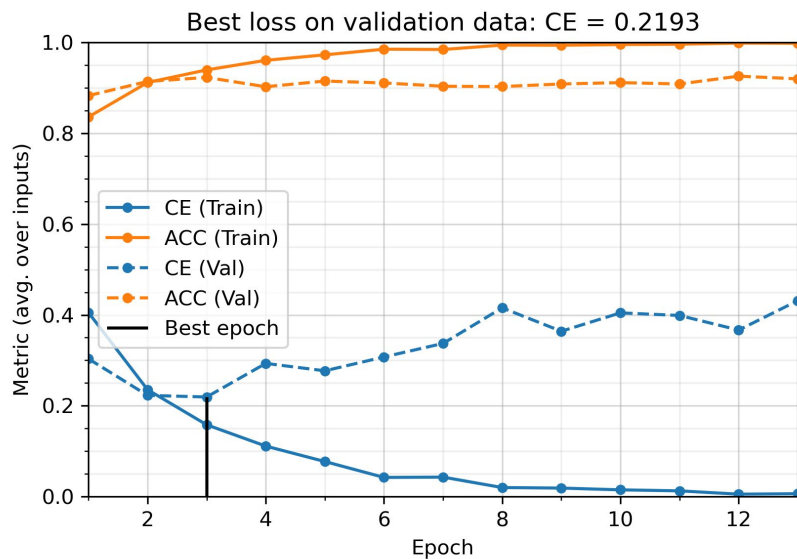
* <https://www.image-net.org/>

** Models are implemented in PyTorch with downloadable pre-trained weights: <https://pytorch.org/vision/stable/models.html>

ConvNeXt-Tiny: Training Losses



ConvNeXt-Tiny: Training Losses and Confusion Matrix



Some predictions...

Truth = 2. Pred = 2



Truth = 0. Pred = 0



Truth = 0. Pred = 0



Truth = 0. Pred = 0



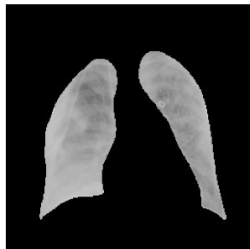
Truth = 2. Pred = 0



Truth = 1. Pred = 1



Truth = 1. Pred = 1



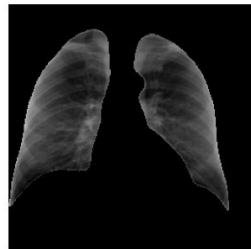
Truth = 0. Pred = 0



Truth = 1. Pred = 1



Truth = 0. Pred = 0



Truth = 1. Pred = 1



Truth = 2. Pred = 2



Truth = 1. Pred = 1



Truth = 2. Pred = 2



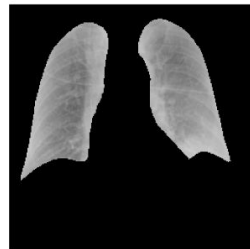
Truth = 0. Pred = 0



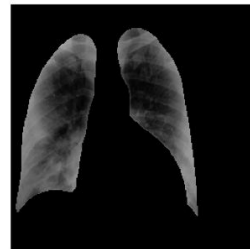
Truth = 2. Pred = 2



Truth = 2. Pred = 2



Truth = 2. Pred = 2



Best models on test-data (Multi Classification)

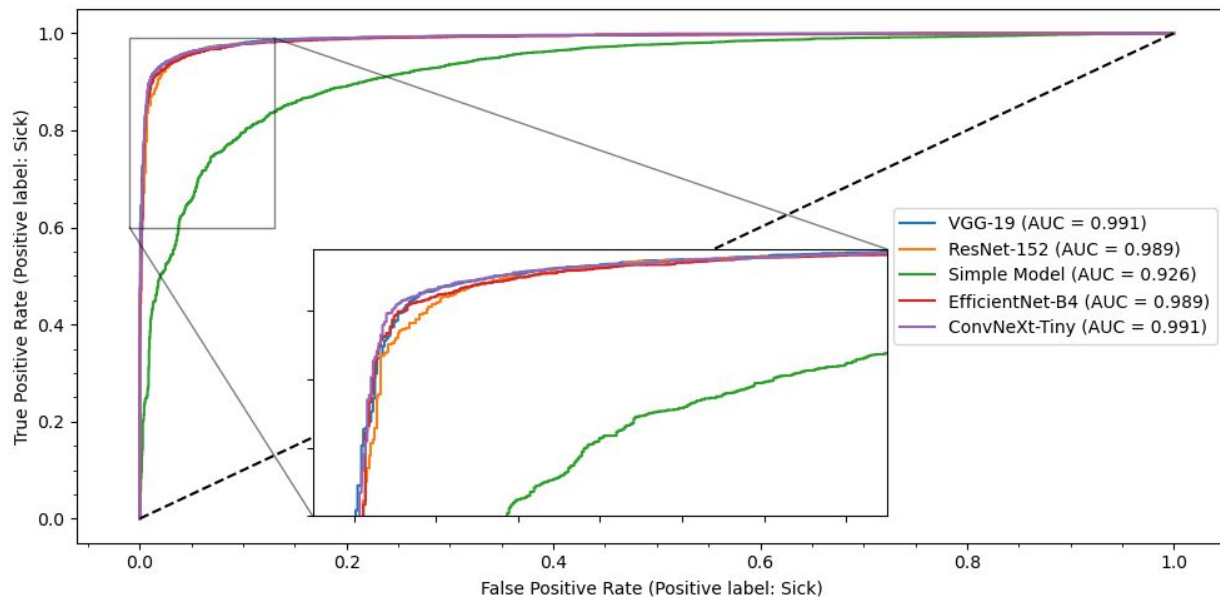
Model	Cross entropy loss	Accuracy	# Params	Scheduler	Image augmentation
VGG-19	0.156	94.6%	144M	ExponentialLR (lr_init = 1e-4, gamma = 0.6)	Horizontal/vertical flip
ResNet-152	0.173	94.0%	60.2M	ExponentialLR (lr_init = 1e-4, gamma = 0.6)	RandomAugment
ConvNeXt-Tiny	0.152	94.7%	28.6M	CosineAnnealingWarmRestarts (lr_init = 1e-4, lr_min = 0, T_0 = 3, T_mult = 2, gamma = 1)	None
EfficientNet-B4	0.168	94.4%	19.3M	CosineAnnealingWarmRestarts (lr_init = 1e-3, lr_min = 1e-4, T_0 = 4, T_mult = 2, gamma = 0.8)	None
Simple model	0.591	76.1%	0.8M	ExponentialLR (lr_init = 1e-3, gamma = 0.9)	None

Binary Classification (sick or not sick)

Combine classes 'Non-COVID' and 'COVID' to form a binary classification problem:

- Positive Label → Covid and Pneumonia (Sick)
- Negative Label → Normal (Not Sick)

Pre-trained networks close to the information limit

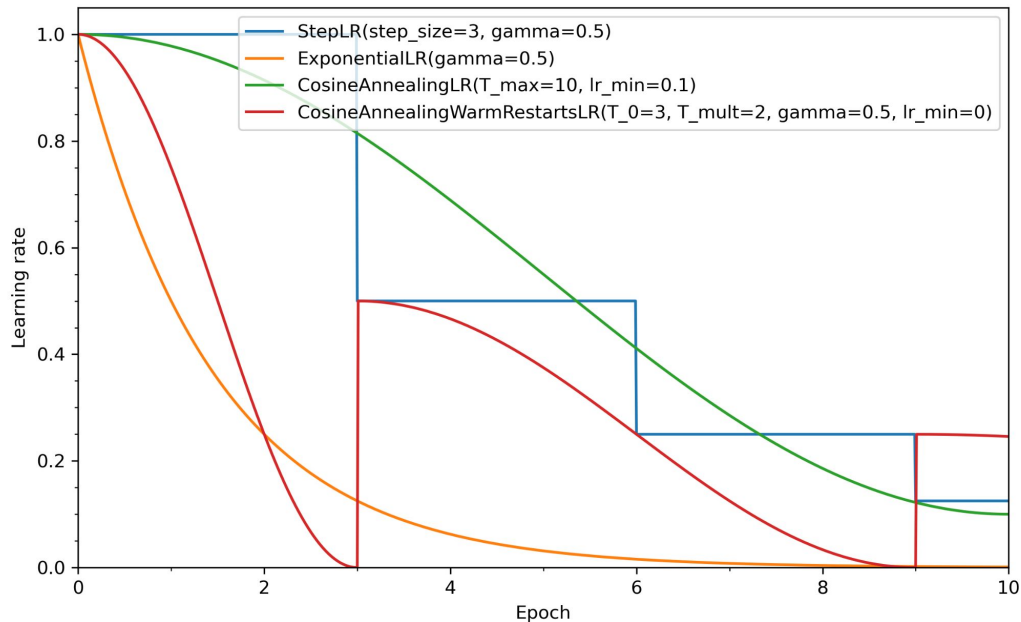


Things we've tried...

Different learning rate schedulers

- StepLR
- ExponentialLR
- CosineAnnealingLR
- CosineAnnealingWarmRestarts (+ modification with exp. decay)

In practice there was little difference between the quality of the final models



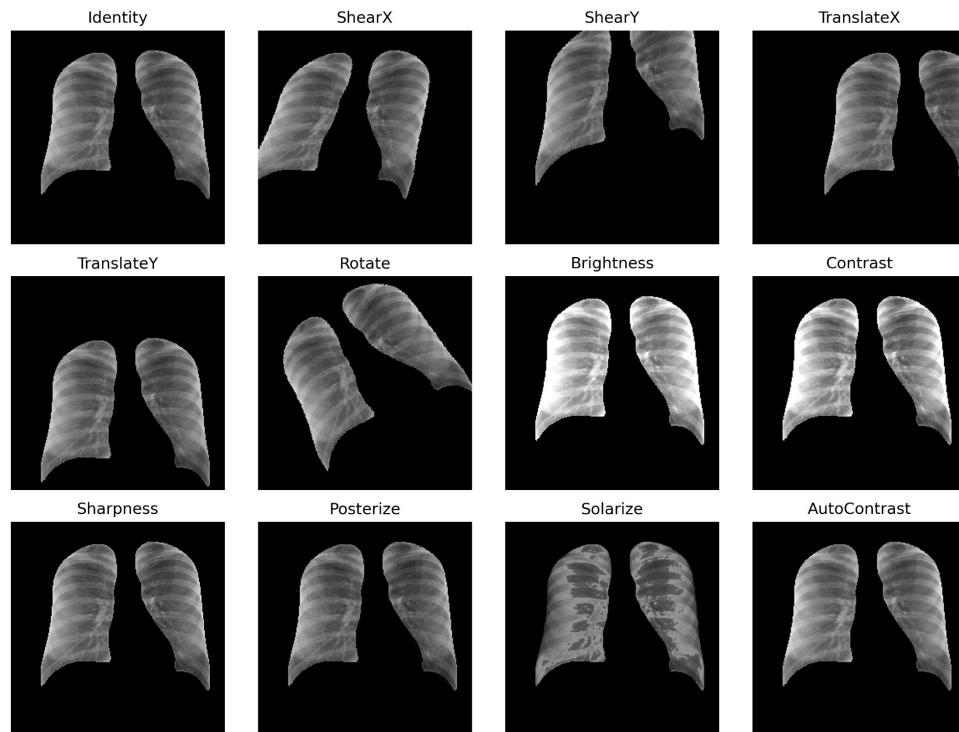
Things we've tried...

Different augmentations for training set

- AutoAugment
 - RandAugment
 - AugMix
 - TrivialAugmentWide
 - Random Horizontal + Vertical Flip (Prob = 0.5)
- 

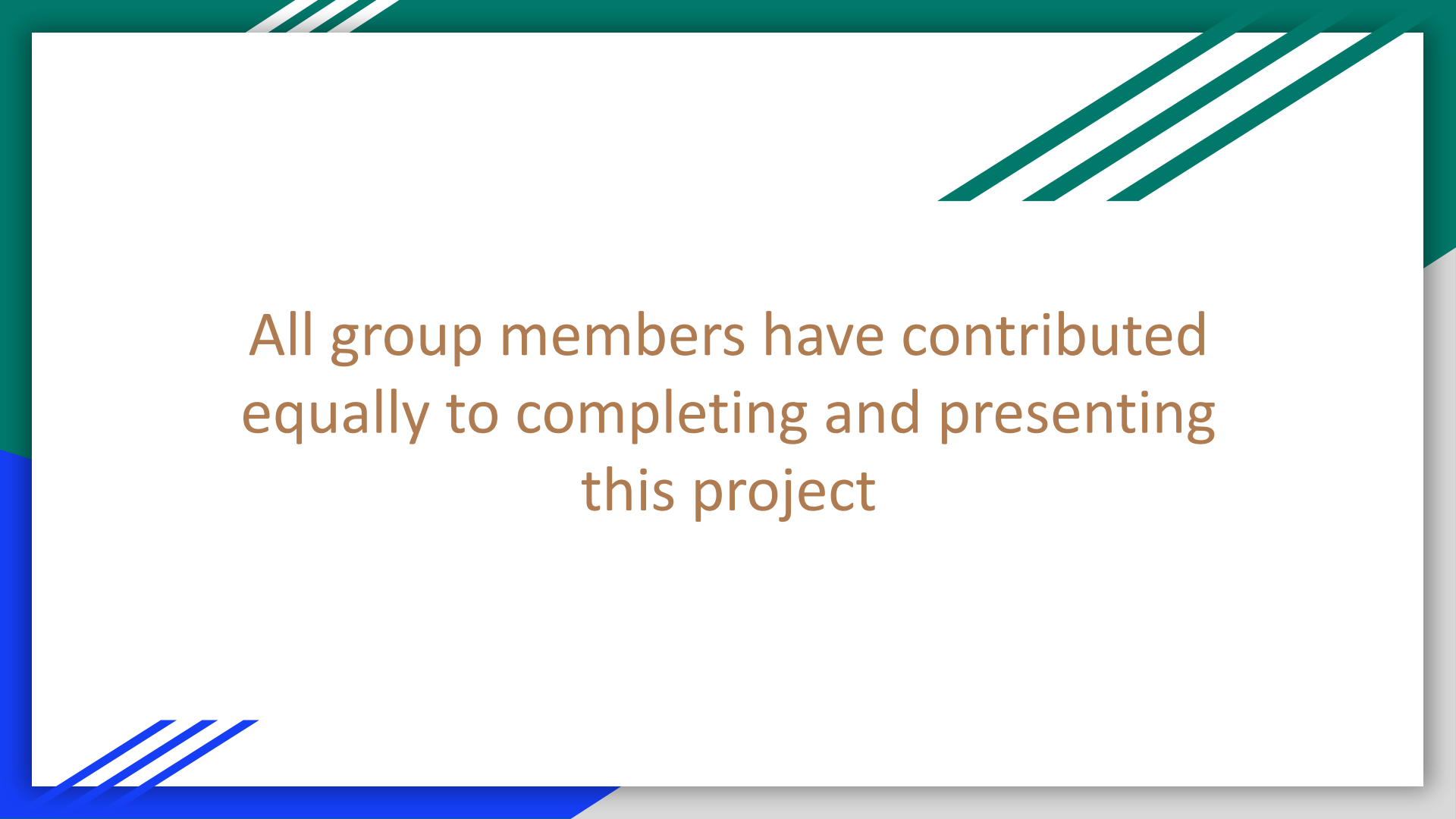
Miscellaneous (but also poor)

- Label-smoothing in loss function
- Weight decay in Adam optimizer



Conclusion

- The pre-trained CNNs we tried were surprisingly powerful in classifying the images and in particular detecting COVID-19.
- The simple model couldn't keep up with the tried and tested CNNs
- Our optimization attempts didn't improve the models much



All group members have contributed
equally to completing and presenting
this project



Appendix

Links to the used data sets

Heart disease (cleaned):

<https://www.kaggle.com/datasets/kamilpytlak/personal-key-indicators-of-heart-disease>

Heart disease (raw):

https://www.cdc.gov/brfss/annual_data/annual_2020.html?fbclid=IwAR072yYfrbYrfTkYHbB-auzfdhGAHHk2zfoN69SDEbaDKPHu-ilU61583F8

Chest X-Rays (COVID-QU-Ex Dataset): <https://www.kaggle.com/datasets/anasmohammedtahir/covidqu>

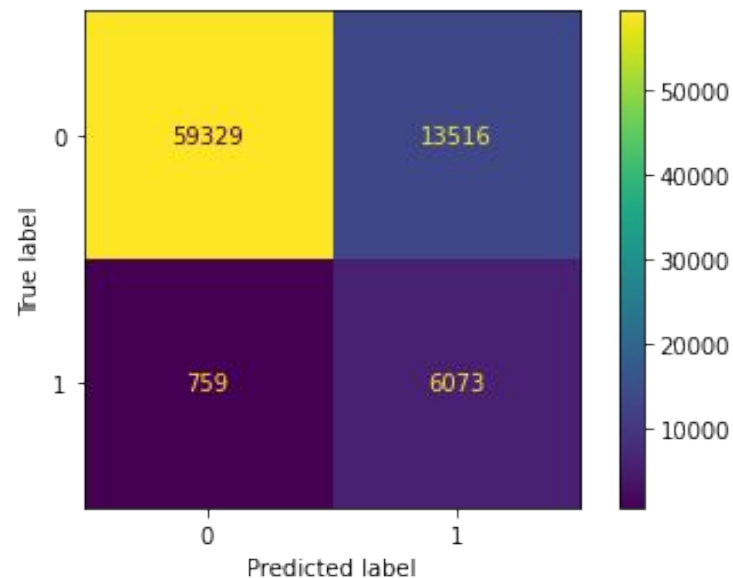
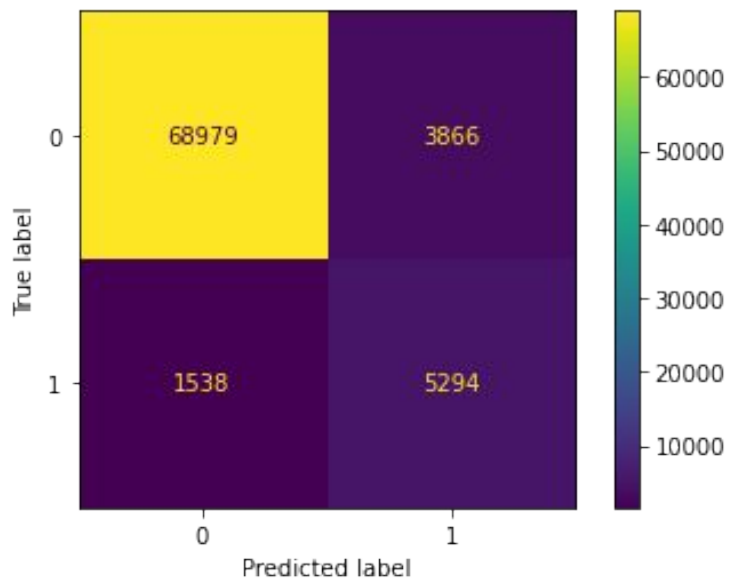


Appendix (Heart disease)

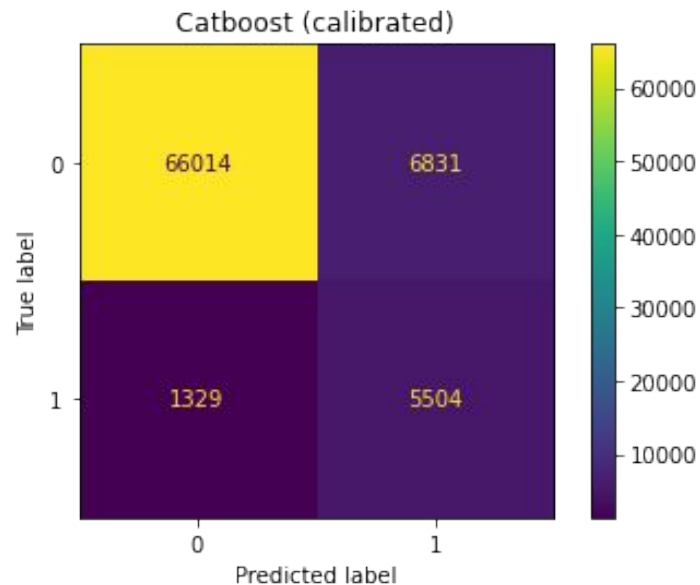
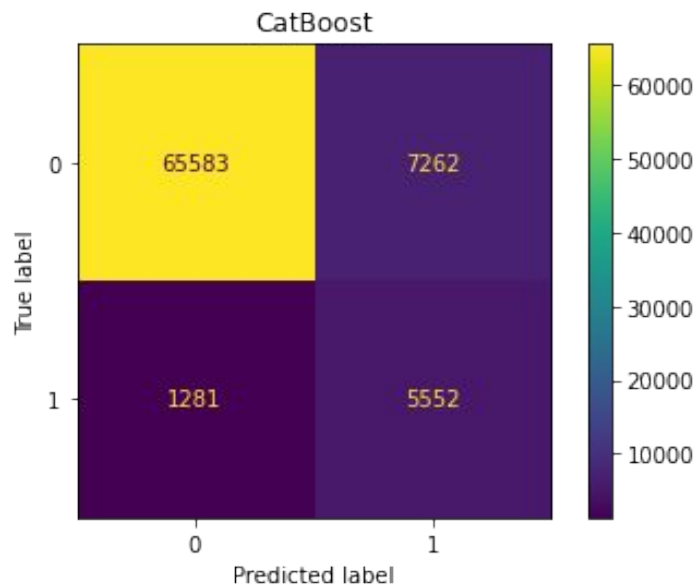
Technical details

- All Boosted Decision tree parameters were tuned with Optuna.
- Binary cross entropy was used as a loss function for the Neural Network.
- Many models were deleted since they were poor predictors, i.e basic tree algorithms, svms and logistic regression.
- Making models for each sex did not work any better.
- Verstack was used to estimate the performance of F1, AUC and weighted F1. They did not perform well.
- Features were selected using SHAP and feature importance on LightGBM.

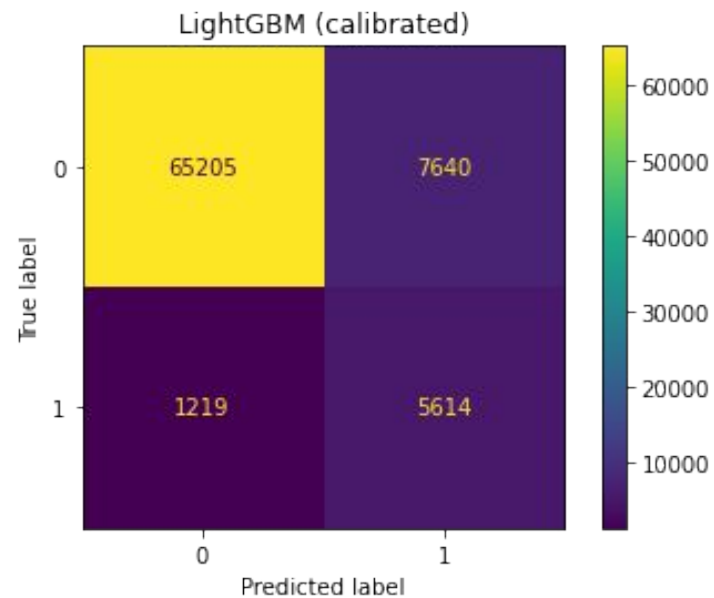
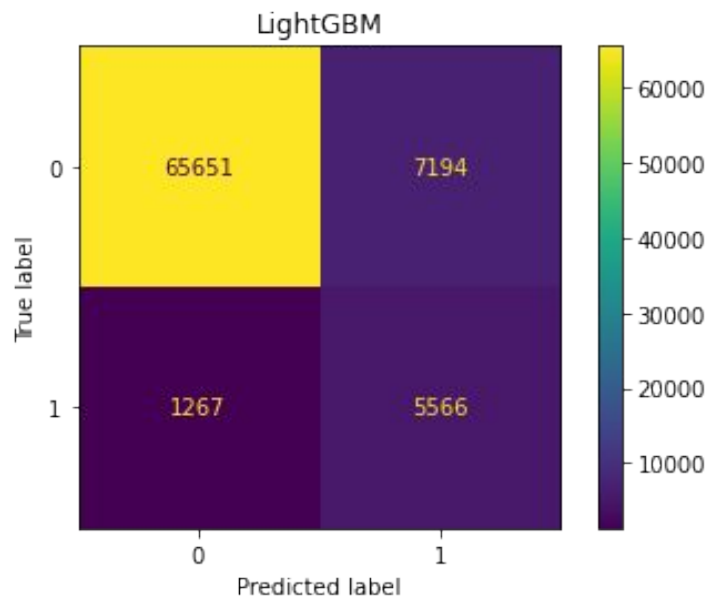
Metric comparison: Average Precision vs F2 (validation set)



Calibration comparison: CatBoost

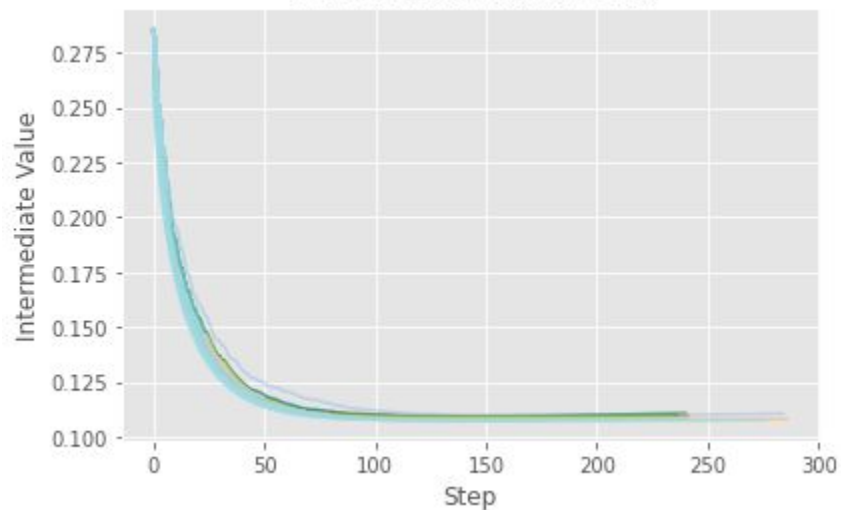


Calibration comparison LightGBM

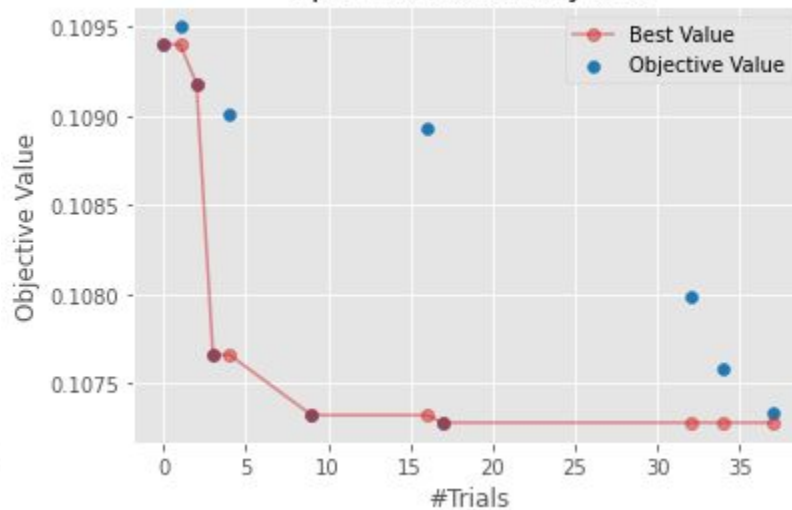


Optimization results: Verstack (AUC)

Intermediate Values Plot

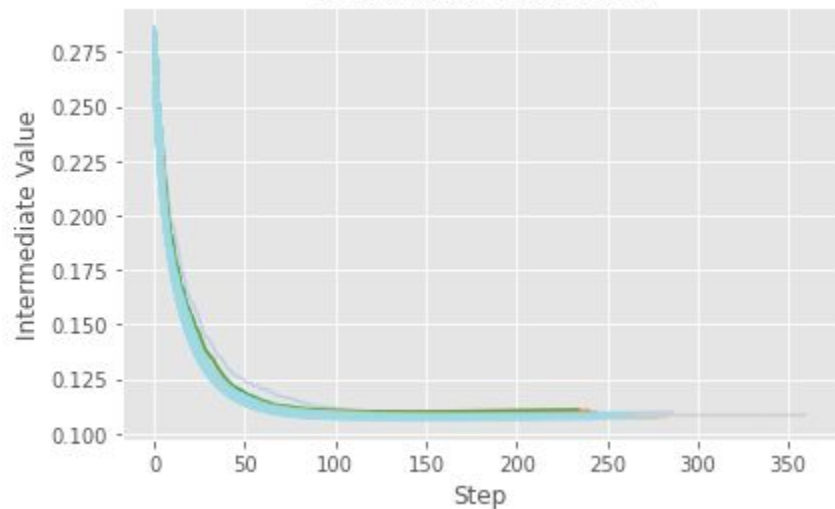


Optimization History Plot

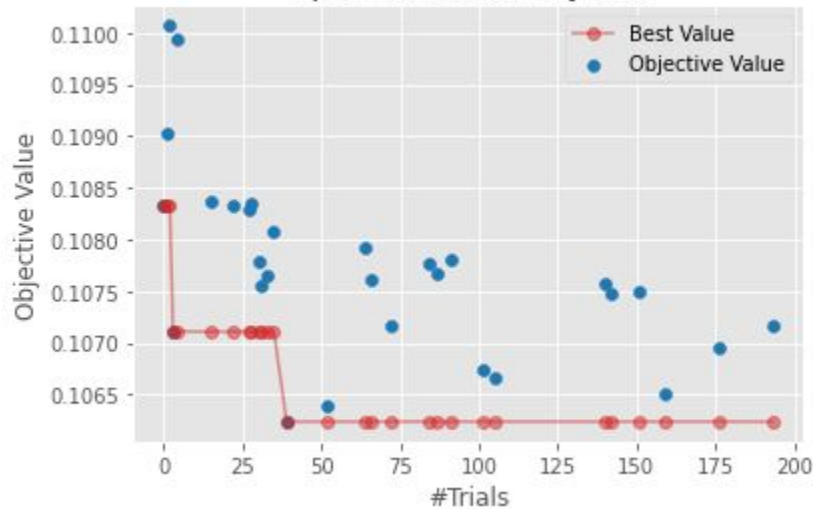


Optimization results: Verstack (F1)

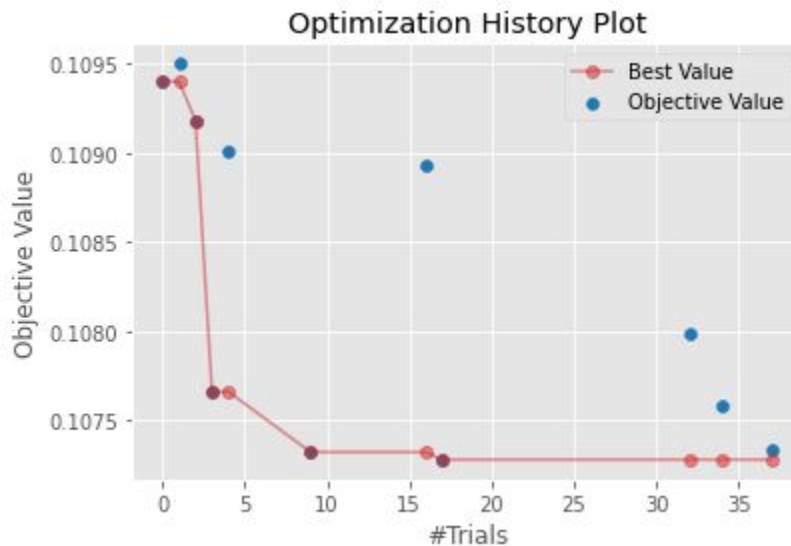
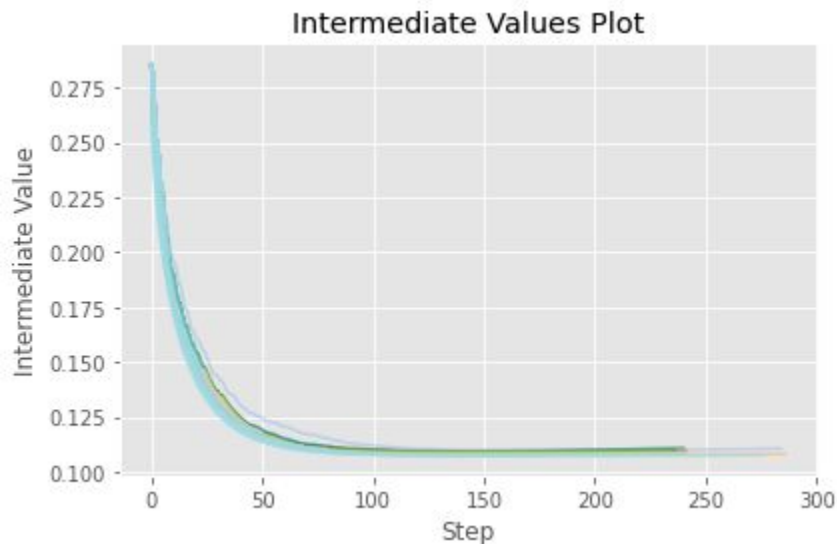
Intermediate Values Plot



Optimization History Plot



Optimization results: Verstack (Weighted F1)



Best 18 features (LightGBM as Base model)

```
['PNEUVAC4', 'COLNTEST', 'CHECKUP1', 'LASTPAP2', 'DIABETE4', 'DRDXAR2', 'EMPLOY1', 'SLEPTIM1',  
'WEIGHT2', 'CHCCOPD2', 'PERSDOC2', 'FLSHTMY3', 'POORHLTH', 'CVDINFR4', 'GENHLTH', 'WTKG3',  
'BMI5', 'AGE80']
```

Discussion

- All Boosted Decision Trees have identical performance on the test set.
- Undersampling was the preferred method to deal with imbalance, but it also introduces bias.
- Average Precision score was the preferred metric score, it reduced the tradeoff between false positives and false negatives.
- The performance of the Neural Network was poor, high bias - high variance tradeoff..
- Achieved better feature choice with features extracted from model compared to SHAP values.
- Final hyperparameter optimization was redundant. Base classifiers with small adjustments worked better.



Appendix (COVID-19)

How we normalized the image data

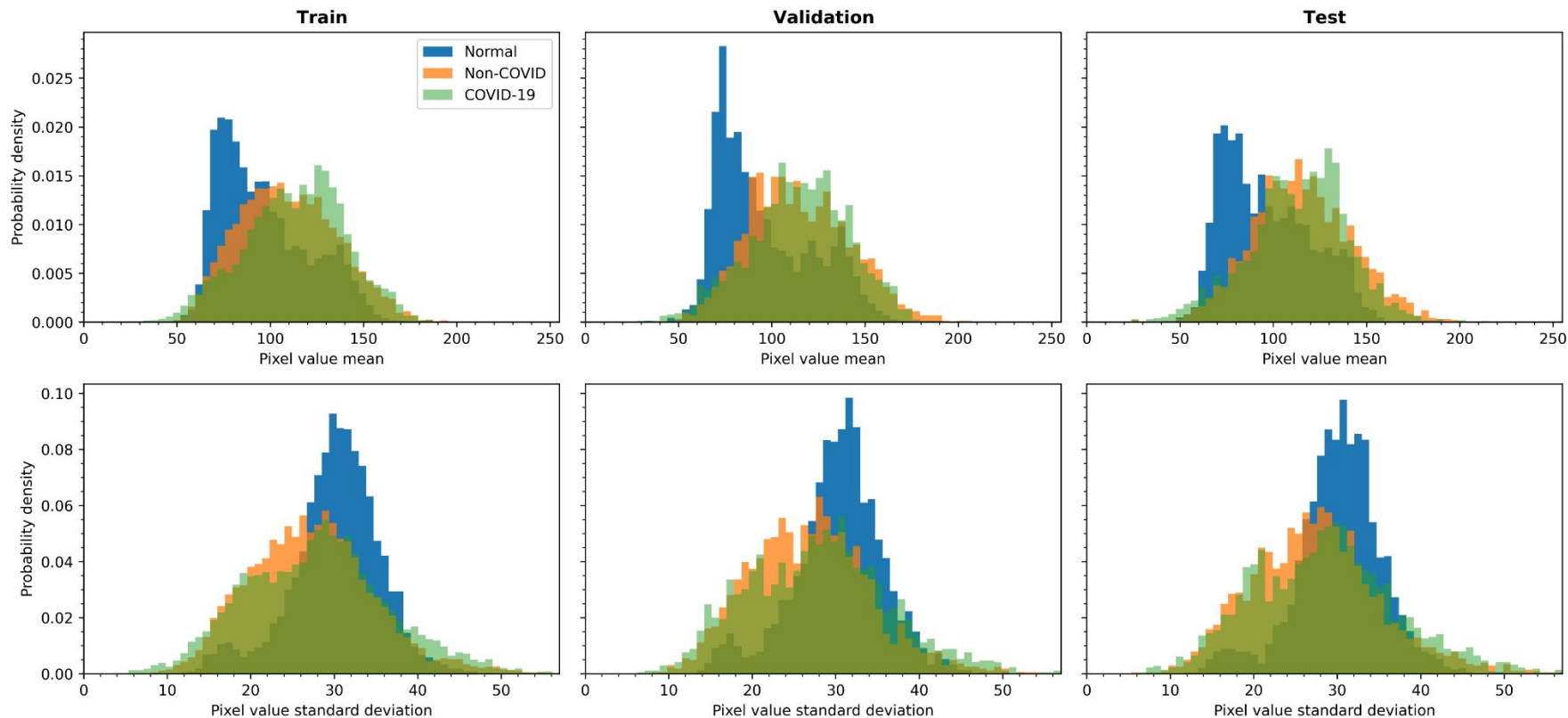
The following three slides show the distribution of means and standard deviations across all 33.920 images divided in train, validation, and test data splits.

The 'mean of the means' is subtracted from all pixel values in each image and divided by the 'mean of the standard deviations' to normalize the data to values of order 1 and to match the pre-trained networks.

There clearly is already some information that can separate the classes – this is preserved by this simple transformation (the distributions are identically looking, just on a different scale).

The distributions are similar across the 3 splits: train, validation, test. This is, of course, a good sign that they don't seem to be biased – at least not when studying means and standard deviations.

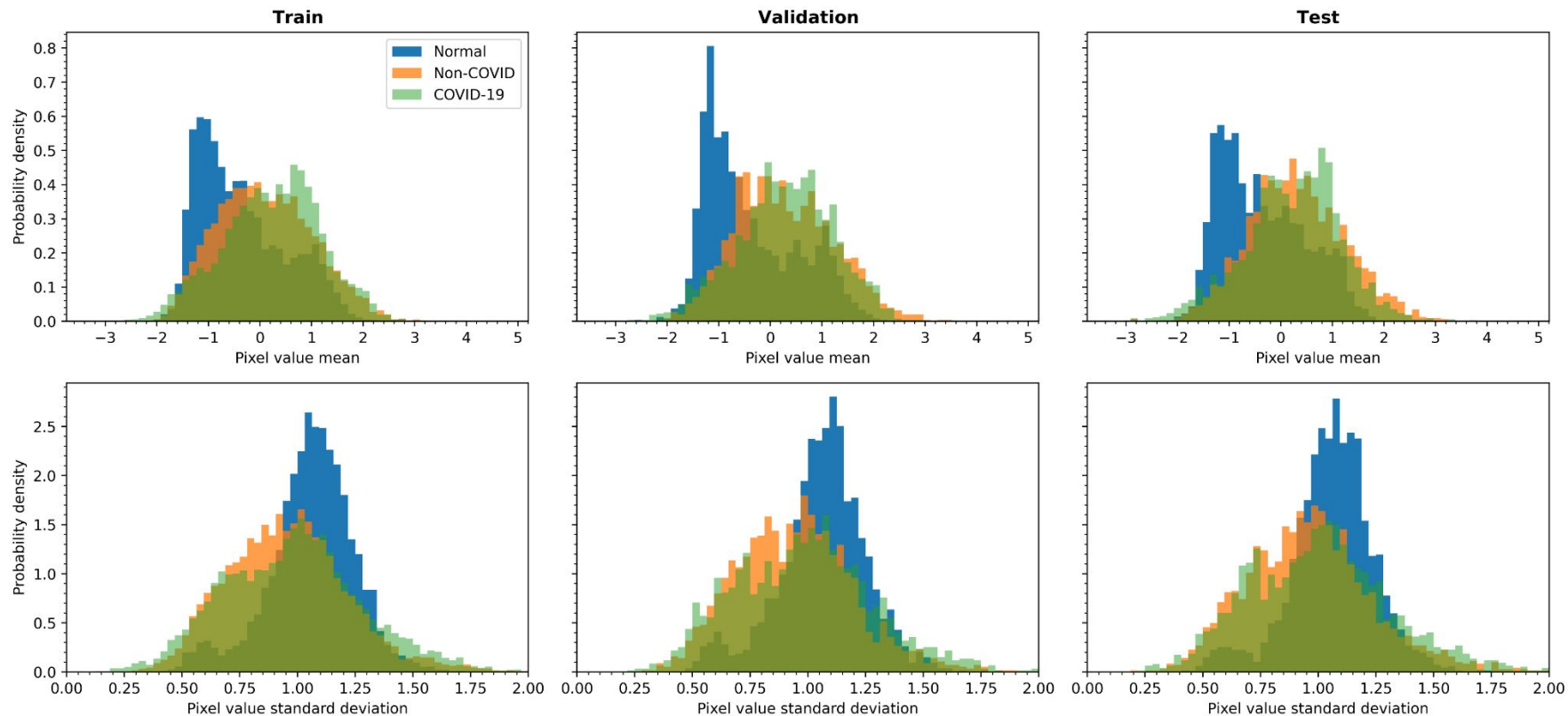
Before normalization



Before normalization



After normalization



'Simple model' architecture

PyTorch implementation:

```
from torch import nn
```

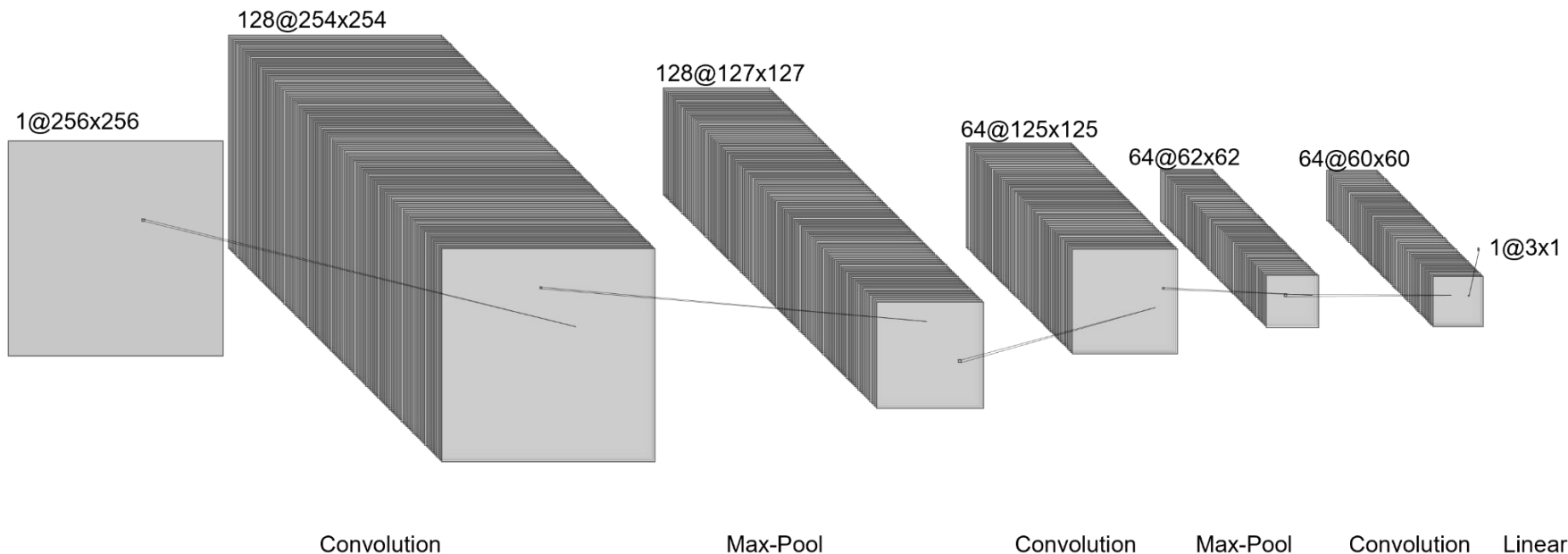
```
simple_model = nn.Sequential(nn.Conv2d(1, 128, 3),  
                             nn.ReLU(),  
                             nn.MaxPool2d(2),  
                             nn.Dropout(0.3),  
                             nn.Conv2d(128, 64, 3),  
                             nn.ReLU(),  
                             nn.MaxPool2d(2),  
                             nn.Dropout(0.5),  
                             nn.Conv2d(64, 64, 3),  
                             nn.ReLU(),  
                             nn.Flatten(),  
                             nn.Linear(64 * 60 * 60, 3)  
)
```

Layer (type:depth-idx)	Output Shape	Param #
Sequential	[32, 3]	--
Conv2d: 1-1	[32, 128, 254, 254]	1,280
ReLU: 1-2	[32, 128, 254, 254]	--
MaxPool2d: 1-3	[32, 128, 127, 127]	--
Dropout: 1-4	[32, 128, 127, 127]	--
Conv2d: 1-5	[32, 64, 125, 125]	73,792
ReLU: 1-6	[32, 64, 125, 125]	--
MaxPool2d: 1-7	[32, 64, 62, 62]	--
Dropout: 1-8	[32, 64, 62, 62]	--
Conv2d: 1-9	[32, 64, 60, 60]	36,928
ReLU: 1-10	[32, 64, 60, 60]	--
Flatten: 1-11	[32, 230400]	--
Linear: 1-12	[32, 3]	691,203

=====
Total params: 803,203
Trainable params: 803,203
Non-trainable params: 0
Total mult-adds (G): 43.81
=====
Input size (MB): 8.39
Forward/backward pass size (MB): 2429.04
Params size (MB): 3.21
Estimated Total Size (MB): 2440.64
=====

PyTorch nn documentation: <https://pytorch.org/docs/stable/nn.html>

'Simple model' architecture



Technical details

The optimizer 'Adam' was used for all the models during training and the best models stated in the table on slide [24](#) are therefore using this optimizer.

The loss function 'Cross Entropy Loss' was chosen as this is commonly used and well suited for multi classification problems.

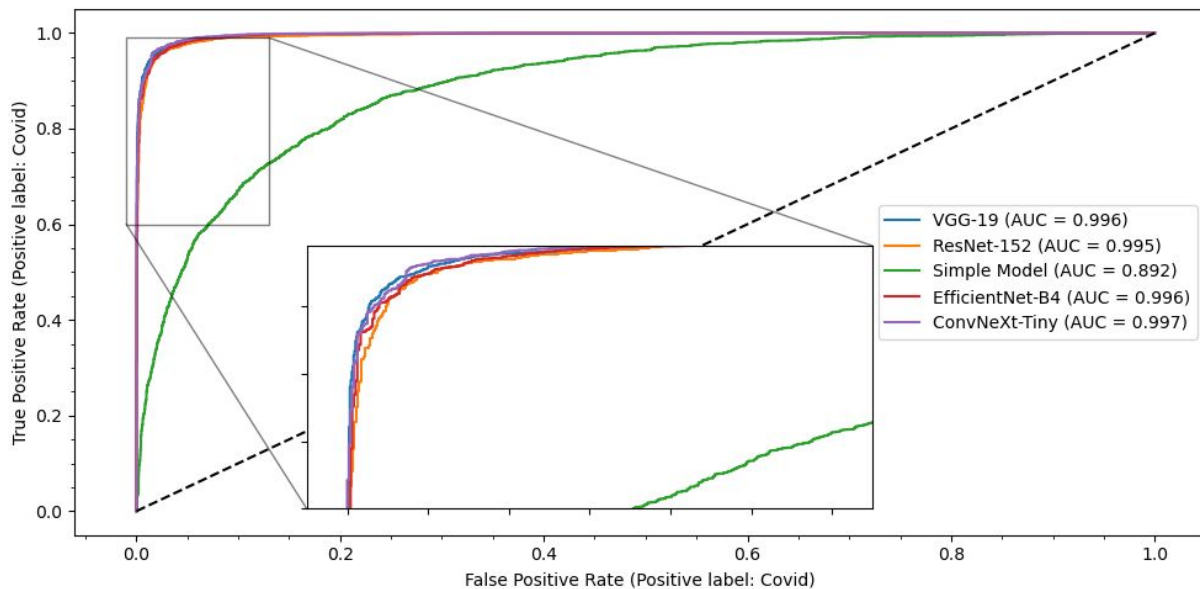
Due to high memory requirement we were forced to load the images onto the GPU in batches with a batch size of 32 images, which contributed to a fairly long training duration. In addition, the pre-trained CNNs were fairly large, which contributed further. The epoch duration was approximately 5-10 minutes depending on the particular model and image augmentation used.

We employed early stopping during training with a patience of 10 epochs, meaning that we continue training as long as the cross entropy loss on the validation set has decreased in the last 10 epochs. The best model (lowest loss on validation set) was saved. Early stopping seemed to be the best regularization/overfitting preventative method, we tried.

Binary Classification (Covid or no Covid)

Negative label → Normal and
Pneumonia

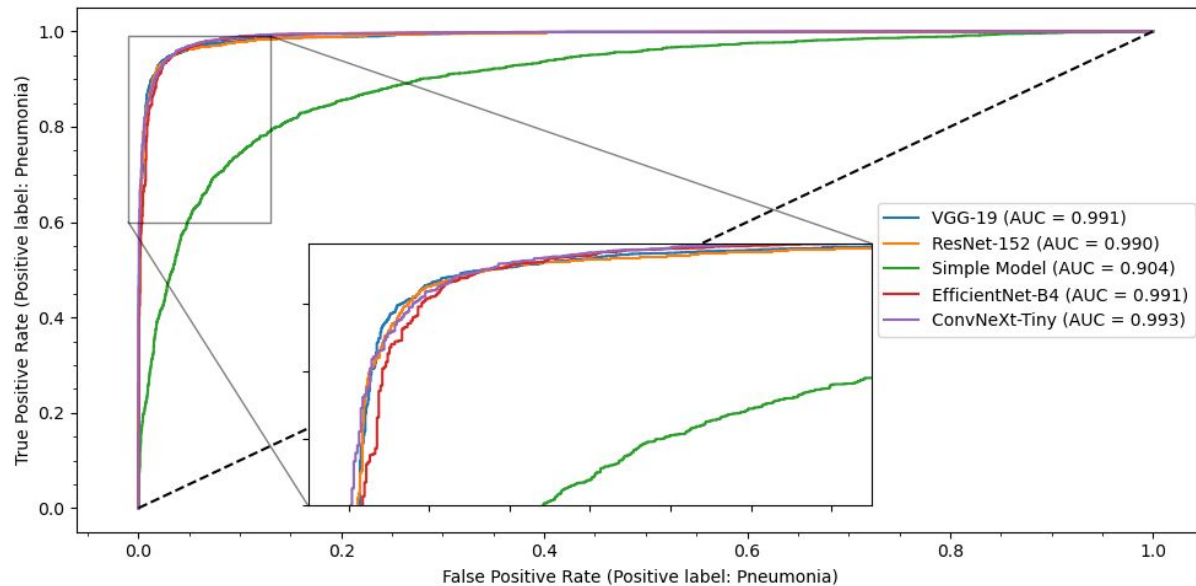
Positive label → Covid



Binary Classification (Pneumonia or no Pneumonia)

Negative label → Normal and Covid

Positive label → Pneumonia



Comparison of image augmentations

Model	Cross entropy loss	Accuracy	Image augmentation
ConvNeXt-Tiny	0.152	94.7%	None
ConvNeXt-Tiny	0.154	94.9%	AugMix
ConvNeXt-Tiny	0.156	94.4%	RandAugment
ConvNeXt-Tiny	0.180	93.5%	AutoAugment
ConvNeXt-Tiny	0.189	93.3%	TrivialAugmentWide

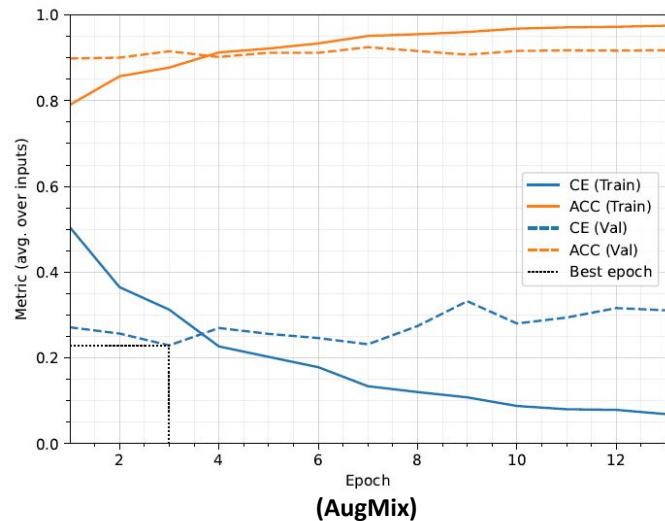
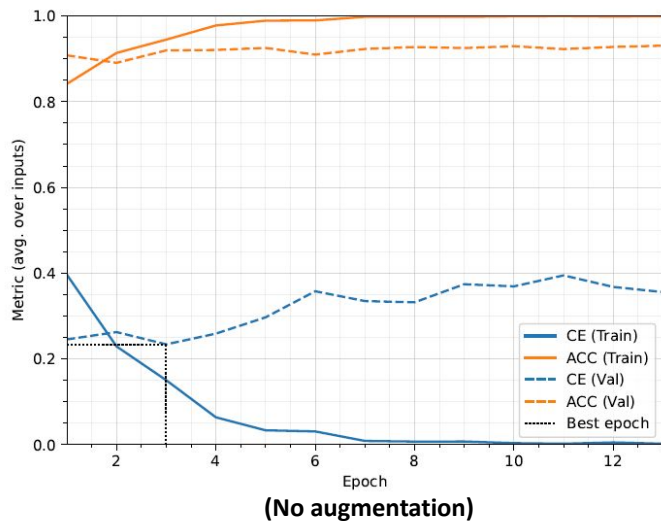
We compared different image augmentation techniques from the PyTorch library. The best cross entropy loss was with no image augmentation but AugMix gave higher accuracy.

The top three are all pretty close and would probably fall within the margin of error if we cross-validated or reran the experiments but due to hour long training times this was not done.

NB: We used the same scheduler for all: CosineAnnealingWarmRestarts(lr_init=1e-4, lr_min=0, T_0 = 3, T_mult = 2, gamma = 1)

Effect of image augmentation on training

On the two figures below, the ConvNeXt-Tiny pre-trained network is trained with scheduler StepLR($lr_init=1e-4$, $step_size=3$, $gamma=0.5$) but with no image augmentation on the left and the 'AugMix' augmentation scheme on the right. We see that without augmentations, the loss converges faster towards zero (eventually reaching almost 100% accuracy); the network needs to learn more images with AugMix (though the generated images are very similar). The validation loss during the AugMix model training tends to be lower, indicating less overfitting, though more data is needed to draw that conclusion in general.



Effect of initializing pre-trained weights

When training from the ImageNet weights in the pre-trained CNNs, we noticed that

- training required **lower learning rate**,
- loss was initially lower and **converged faster**, meaning **faster overall training**