Skin lesion classification with Convolution Neural Network

Roch Fedorowicz

BIAI project raport

Contents

Project topic

1.1 General assumptions

The project was aiming to create a convolutional neural network that would be able to classify skin lesions into 2 different classes: malignant and benign and then into 8 classes of specific skin conditions. As input data, a data-set consisting of around 25 thousand images was used. As a result, two separate final models should be created trained, and tested giving satisfactory outcomes.

Analysis of the task

2.1 Possible approaches to solve the problem

To solve the problem of classification any fundamental or more compound classifier should be theoretically an option to solve it. However, data derived from the data set are quite large even for singular training examples, and therefore all fundamental approaches will struggle to divide data in the best possible way. During the phase of finding the best solution following classifiers were considered.

2.1.1 Linear Discriminant Analysis

This approach uses multidimensional space and a newly created axis. It attempts to divide data along these axes. All classes have to be the most distinguishable along one axis so the classification can be performed. It is easy to implement but assumes that the variance of data is equal among all the classes and poorly performs with a bigger number of features that directly influence the number of space dimensions.

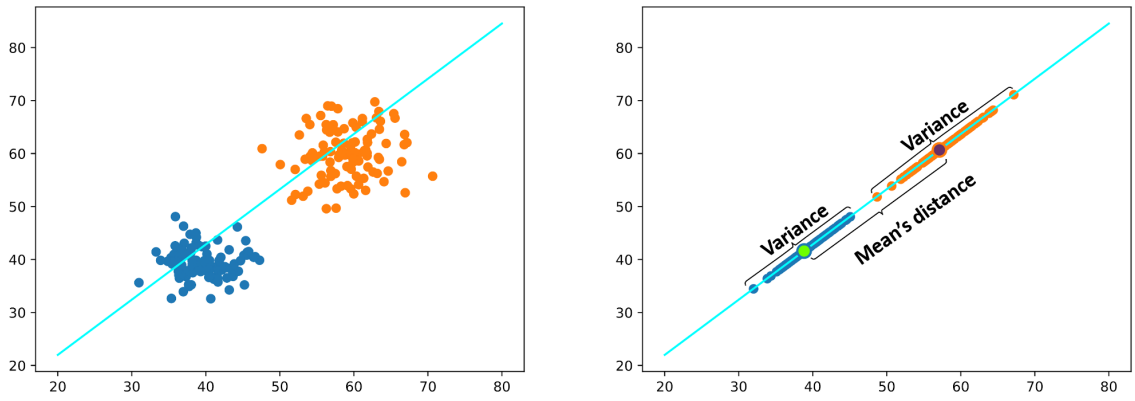


Figure 2.1: Example of dividing 2-dimensional feature space using LDA

2.1.2 Support Vector Machine

This approach uses multidimensional space and supporting vectors to set boundaries between data. It attempts data division by finding the most outlined element of the group and using it as a reference to create a supporting vector and then set a boundary. In terms of multi-class classification, it divides it into smaller problems of binary classifications between each pair of classes. It performs better with the high dimensionality of data than LDA, but it still assumes that all data are linear separably.

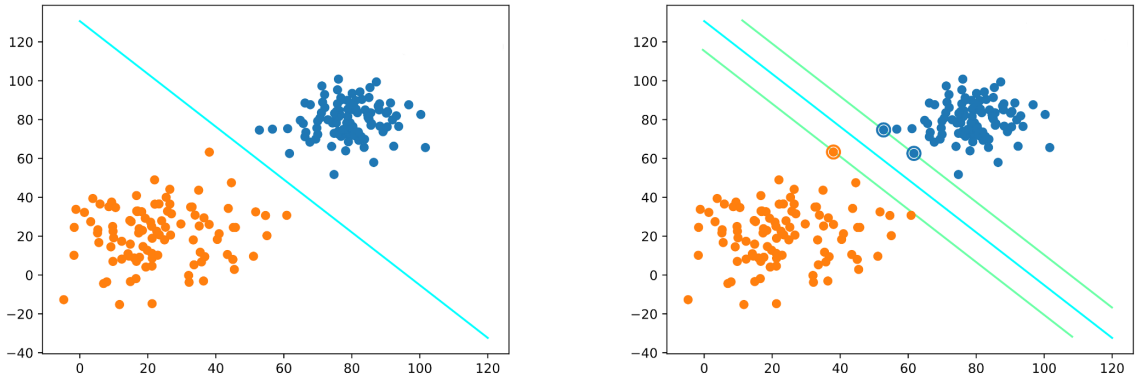


Figure 2.2: Example of dividing 2-dimensional feature space using SVM

2.1.3 Random Forest Algorithm

This approach uses a lot of smaller classifiers (known as decision trees) to calculate the compound prediction of the belongingness of data to a particular class. The single decision tree looks like the nested if structure that on every next level, whether based on particular parameter data should be classified to a particular class or the other next level should be evaluated. Then having a lot of these simple classifiers this attempt manages to classify data even when they are not linear separably. However, to achieve high accuracy of such a classifier on compound data set a lot of trees would be created, and therefore it would be extremely ineffective.

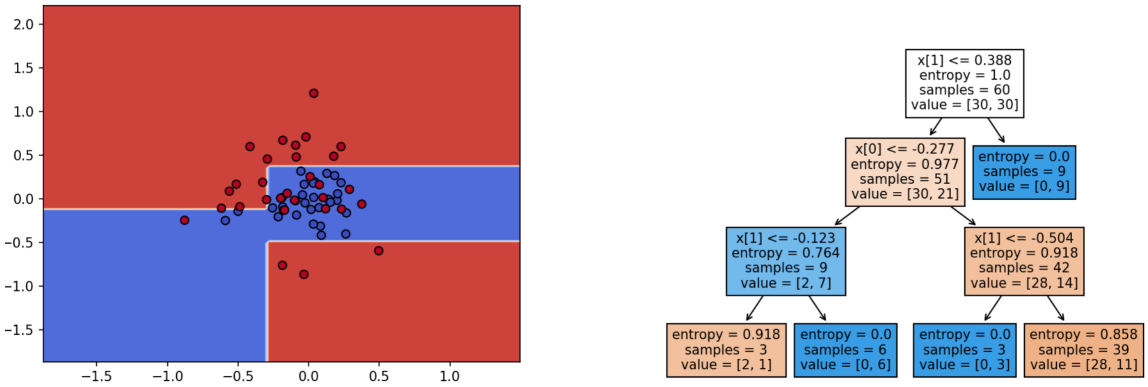


Figure 2.3: Example of single decision tree dividing 2-dimensional feature space

2.1.4 Classic (Artificial) Neural Network

This approach uses neurons - simple calculation elements - that are stored in layers and then in a neural network. These classifiers could basically transform any kind of input data into desired output data. The problem with a classical neural network for image classification is that it would be highly ineffective for any bigger images as input data for such a classical neural network would be equal to height times width times the number of color channels.

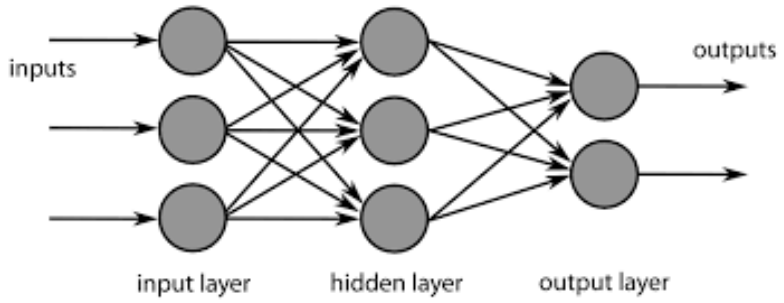


Figure 2.4: Architecture of artificial neural network

2.1.5 Convolutional Neural Network

This approach is similar to ANN, but before the bare calculation will take place the features are extracted from data by a special type of layers interlaced one by another - creating adaptive image filtering. These types are convolution layers responsible for feature extraction and polling layers responsible for making features independent from the position in the image. The whole process is handled in 2 dimensions so it fits perfectly the 2-dimensional nature of data representing images. After that extracted features are passed to the normal layers conducting calculations in order to predict data-class relation. This last part works exactly the same as in ANN.

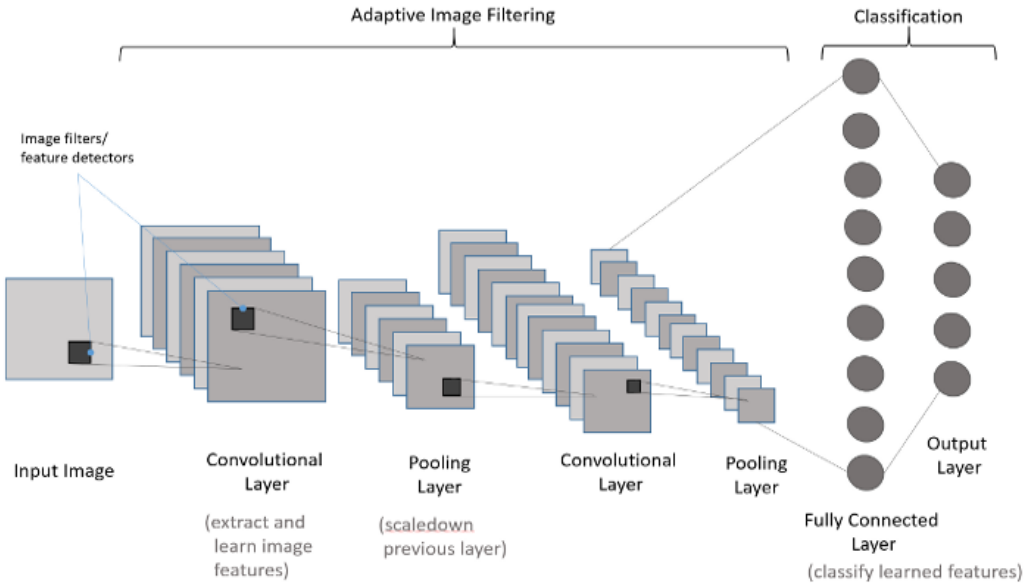


Figure 2.5: Architecture of convolutional neural network used for image classification

This way of solving the problem was ultimately chosen, because CNN are one of the best classifiers that could be used for image classification as they were designed directly to solve this problem.

2.2 Available data-sets to model the problem

Most of the data-sets considered to model the problem consisted of several classes being:

- ak - Actinic Keratosis
- bcc - Basal Cell Carcinoma
- df - Dermatofibroma
- mel - Melanoma
- nv - Melanocytic nevus
- bkl - benign keratosis-like lesions (Solar Lentigines / Seborrheic Keratoses and Lichen Planus-like Keratoses)
- scc - Squamous Cell Carcinoma
- vasc - vascular lesions (Angiomas, Angiokeratomas, Pyogenic Granulomas and Hemorrhage)

2.2.1 Skin Cancer ISIC

A data-set consisting of only 2357 images from 7 classes. It was collected by International Skin Imaging Collaboration and therefore has high quality. The most represented class has 5 times more representatives than the less represented one.

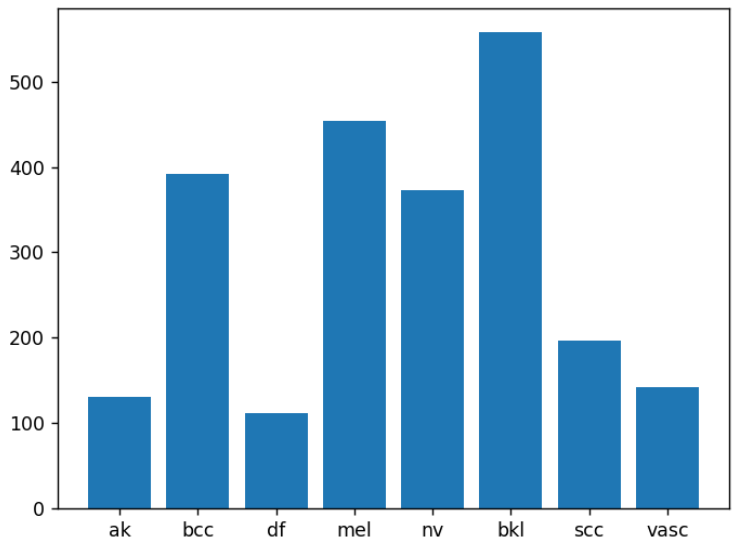


Figure 2.6: Number of classes belonging to particular groups in data-set

2.2.2 Skin Cancer MNIST: HAM10000

A fairly big data-set covering 7 different classes. It consisted of 10 000 images, but the disproportion between the most represented class and the less represented class was around 60 ratio.

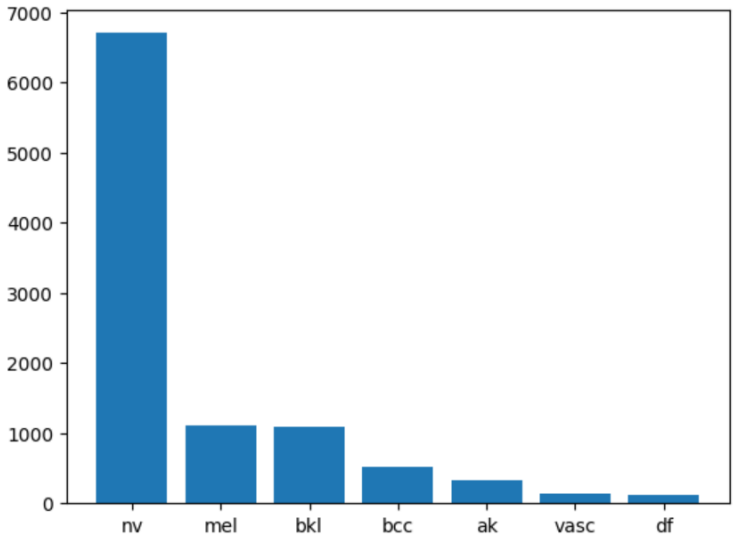
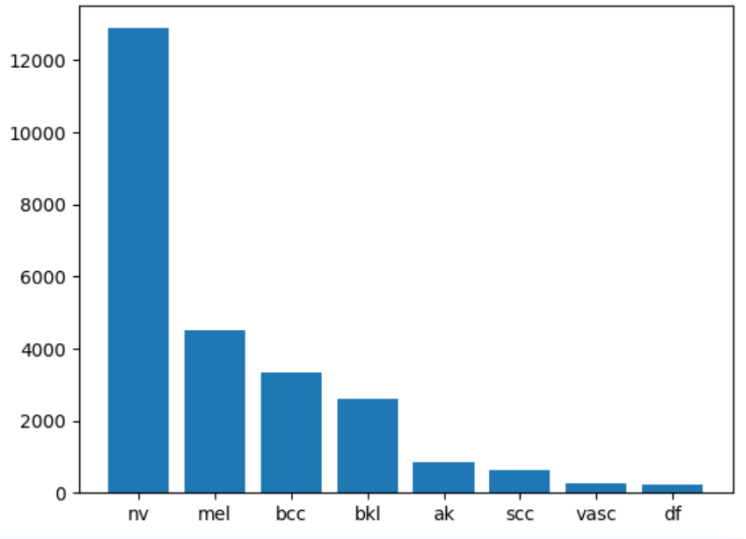


Figure 2.7: Number of classes belonging to particular groups in data-set

2.2.3 Skin Lesion Images for Melanoma Classification

A chosen data set that covers all 8 classes. It still is as imbalanced as the HAM10000 data-set, but it consists of 25 331 images, and therefore even though it is unbalanced the least represented class from this data-set has more representatives than the most represented class from the Skin Cancer ISIC data-set (that was the most balanced data-set from all considered).

Figure 2.8: Number of classes belonging to particular groups in data-set



2.3 Available tools and implementation approach

As there are many possible libraries in several languages only the most popular one was considered.

2.3.1 OpenNN with C++

This library used for C++ language provides speed execution of learning algorithms for NN and simple user-friendly API. It is the faster of considered, but data handling and preprocessing in C++ language besides being really fast is also quite complex.

2.3.2 PyTorch with Python

Python library used to solve deep-learning problems. It is easy to use, compact, and good at memory management. However, as it is Python native library it is also the slowest considered.

2.3.3 Tensorflow with Python

This library is originally written in C++ language which makes it quite fast in terms of speed execution. It was then wrapped to be available to use in other languages like Python or JavaScript. This ultimately allows Tensorflow to be a universal and fast library that besides simple API for building NN provides data preprocessing algorithms. Also, as a language Python was used to simplify all data handling-related operations.

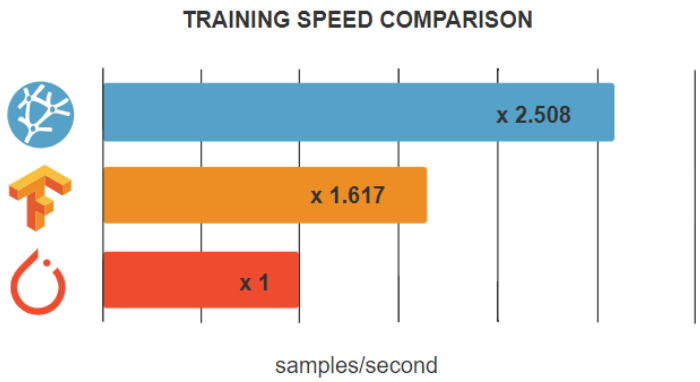


Figure 2.9: Execution speed of considered libraries - `https://www.opennn.net/`

Internal and external specification of the software solution

3.1 Basics of Tensorflow Python API

3.1.1 keras.utils.image_dataset_from_directory

A class that allows loading data-set consisting of images from a directory. It was used during the initial approach. The drawback of loading the data set with this function is that after connecting loaded data to the data pipeline all the process of data augmentation has to be handled as a part of created NN model.

3.1.2 keras.preprocessing.image.ImageDataGenerator

A class that allows generating data-set based on indicated one. The main advantage of this approach is that it provides a way to disconnect the data augmentation process from the learning process. Using this function also enables image data to be better augmented.

3.1.3 keras.Sequential

This simple class allows the creation of NN using the sequential architecture of layers. It enables also squeeze of ready-to-use models into sequential architecture and treats them as the first layer of such it. Due to that, it is widely used to fine-tune pre-trained models. It returns keras.Model object that then could be used to data classification.

3.1.4 keras.Model.compile

Before using the model it has to be compiled. During this function the model is actually built and chosen loss function, optimizer and metrics are applied. Loss function and optimizer has direct influence on training of NN, and metrics enables to specify NN attributes that should be watched over during training.

3.1.5 keras.Model.fit

When model is already compiled, it can be used for training. Here data pipeline are connected to loaded data-set and number of epochs are chosen.

3.2 Main training scripts

3.2.1 Initial approach

Initial approach involved loading data-set, building own model and performing data augmentation during training phase. That resulted in long training time even for simple model. Besides this own-built models did not performed qualitatively enough to be considered good at classification (even for binary classification problem).

```
# Loading data-set from images in directory
train_ds, val_ds = keras.utils.image_dataset_from_directory(
    dataset_path,
    validation_split = 0.2,
    subset = "both",
    ...)

# Creating own model coresponding to typicall architecture of CNN
model = Sequential([
    # Simple data augmentation
    layers.RandomFlip("horizontal",
        input_shape=(img_height, img_width, 3)),
    layers.RandomRotation(0.1),
    layers.RandomZoom(0.1),
    layers.Rescaling(1./255,
        input_shape=(img_height, img_width, 3)),
    ...

    # Adaptive image filter
    layers.Conv2D(16, 3, padding='same', activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(32, 3, padding='same', activation='relu'),
    layers.MaxPooling2D(),
    layers.Conv2D(64, 3, padding='same', activation='relu'),
    layers.MaxPooling2D(),
    ...

    # Last layers predicting what class should data belong to
    layers.Dropout(0.2),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(number_of_classes)
])

# Compiling model
model.compile(optimizer = 'adam',
              loss = 'sparse_categorical_crossentropy',
              metrics = ['accuracy'])

# Training model
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=20)
```

3.2.2 Genetic algorithm - model wrapper class

Class used for wrapping model to make it more compact for genetic algorithm.

```
class MyModel:

    # Constructor
    def __init__(self, img_height, img_width, features, dropout,
        learning_rate, mutation_rate, base_model = None):

        # Create new empty base_model if it was not passed
        if self.base_model == None:
            self.base_model = ...

        # Create new model based on passed parameters, that
        # will reused base_model
        self.model = tensorflow.keras.models.Sequential([
            ...
            self.base_model,
            ...])

        # Compile model
        self.model.compile(...)

    # Function used to produce new offspring, mutations dependant
    # on mutation_rate (or external_mutation_rate if provided)
    # also takes place inside this function
    def produceOffspring(self, external_mutation_rate = None):
        ...
        return MyModel(..., newFeature, newDropout, newLR, newMR,
            copy.deepcopy(self.base_model))

    # Function allowing training of embedded model
    def train(self, train_ds, val_ds, epochs):
        self.history = self.model.fit(...)

    # Function returning accuracy of model, it was used as
    # an fitness function for genetic algorithm
    def getAcc(self):
        ...
        return max(self.history.history['val_accuracy'])
```

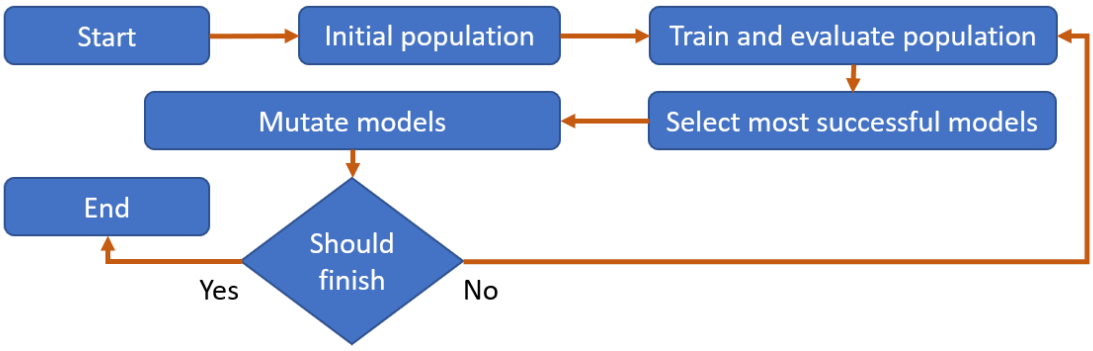


Figure 3.1: Flow chart for genetic algorithm

3.2.3 Genetic algorithm implementation

Genetic algorithm implemented for own models (like in initial approach). It lacks typical for genetic algorithm phase of crossover, and therefore it implements asexual reproduction - offspring inherit mutated genes only from single parent. As a rule the most successful model produce 7 children, second-most successful - 2, and third-most successful only 1.

```
# Creating parent model that has 128 neurons in second-last layer
# with 0.3 dropout, learning rate of 0.001 and mutation rate of 0.1
parent_model = MyModel(img_height, img_width, 128, 0.3, 0.0001, 0.1)
my_models = []

# Creating 10 first random models with external mutation rate of
# 0.8, that will provide diversified first generation based on
# stated parent model
for i in range(0, 10):
    my_models.append(parent_model.produceOffspring(0.8))

# Repeat process for 10 generations
for i in range(0, 10):
    print(f"============ {i + 1} generation ============")
    # Prescreen models in generation before training started
    for model in my_models:
        print(model)

    # Train models for 10 epochs
    ...
    for model in my_models:
        model.train(train_ds, val_ds, 10)
        ...

    # Screen models after training finished
    ...
    my_models = sorted(my_models, key = lambda my_model:
        my_model.getAcc(), reverse = True)
    for model in my_models:
        print(model)

    # Reproduce generation if it is not the last generation
    if i != 9:
        new_my_models = []
        ...
        # Create 7 children of the most successful model
        for i in range(7):
            new_my_models.append(my_models[0].produceOffspring())
            ...
        ...
        # Create 2 children of the second-most successful model
        for i in range(2):
            new_my_models.append(my_models[1].produceOffspring())
            ...
        ...
        # Create 1 child of the third-most successful model
        for i in range(1):
            new_my_models.append(my_models[2].produceOffspring())
            ...
        my_models = new_my_models
```

3.2.4 Latter approach

Latter approach changed way model was built and instead of building own models, pretrained models of complex architecture was used. That allowed to import crucial part of models with pretreined weights inside of it and adjust them to particular classification problem by building own fully connected layers. Also way the model was trained changed, because after fine-tuning model for 40 epochs, training in the whole model would be enabled.

```
# Generator that allows to augment data before it is feed to model
datagen = keras.preprocessing.image.ImageDataGenerator(
        zoom_range = 0.1,
        width_shift_range=0.1,
        rescale=1./255,
        ...)

# Load and generate new data-set
train_generator = datagen.flow_from_directory(
        data_set_path + '/train',
        ...)
validation_generator = ...

# Import an existing model with pretrained weights and do
# not include fully connected layers, that allows to reuse
# model for new purpose
base_model = keras.applications.Xception(include_top = False,
                                         weights ='imagenet',
                                         ...)

# Disable training in base model to perform fine-tuning
for layer in base_model.layers:
    layer.trainable = False

# Create new model that will reuse pretrained model as feature
# extractor, fully connected layer are adjusted to specific problem
model = Sequential([
    base_model,
    layers.Flatten(),
    ...
    layers.Dense(number_of_classes, activation='softmax')
])

# Create own optimizer
optimizer = keras.optimizers.legacy.Adam(learning_rate=0.001, ...)

# Create callback that will reduce learining rate by 0.5 every time
# accuracy stopped improving for 5 epochs in a row
learning_rate_reduction = tf.keras.callbacks.ReduceLROnPlateau(
    monitor='val_accuracy', patience=5 factor=0.5, ...)

# Create callback that will stop training if accuracy stopped
# improving for 20 epochs in a row
early_stopping_monitor = tf.keras.callbacks.EarlyStopping(
    patience=20, monitor='val_accuracy', ...)

# Compile model
model.compile(optimizer = optimizer, ...)

```

```
# Train model
history = model.fit(train_generator,
                    epochs = 30,
                    validation_data = validation_generator,
                    callbacks=[learning_rate_reduction,
                        early_stopping_monitor])
```

3.3 Data structures

The whole data-sets after being loaded was stored as tf.data.Dataset. This tensorflow class creates very effective data pipelines that stores data as input tensors and could be further easily used in training function for created models. However, when it comes to data stored in computer memory, they have to follow significant hierarchy of being grouped into specific folders in order to be used correctly by Tensorflow:

data/
- train/
  - class_A/
    - class_A001.jpg
    - class_A002.jpg
    - ...
  - class_B/
    - class_B001.jpg
    - class_B002.jpg
    - ...
  - ...
- validation/
  - class_A/
    - class_A001.jpg
    - class_A002.jpg
    - ...
  - class_B/
    - class_B001.jpg
    - class_B002.jpg
    - ...
  - ...

Experiments

4.1 Binary classification

4.1.1 Mitigating overfitting by data augmentation inside of the model

First created model has basically no data augmentation - internal or external. It caused model to stop improve accuracy on validation part of data-set.

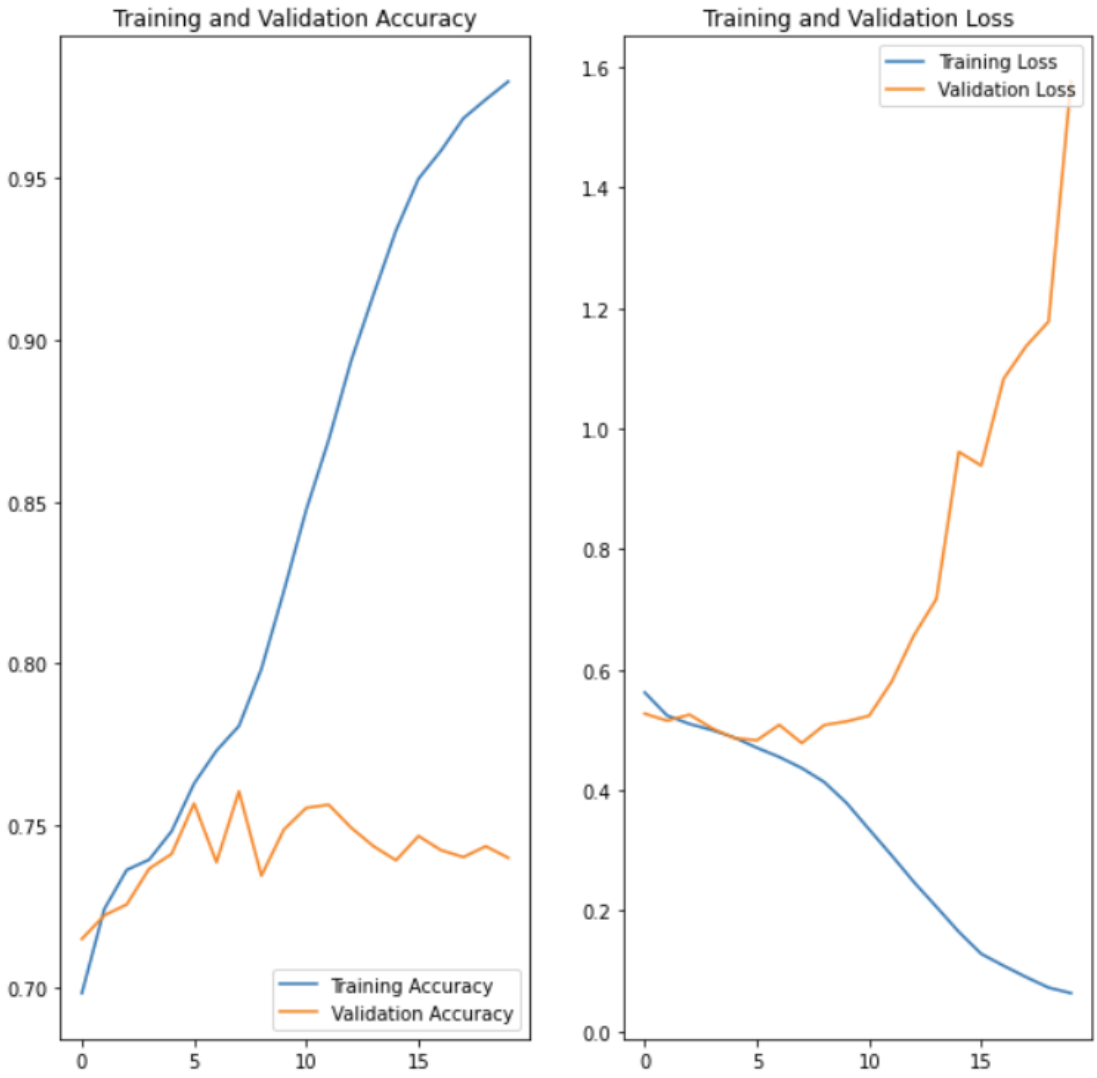


Figure 4.1: Accuracy and loss on training and validation data-set

After introducing simple data augmentation inside model big gap between accuracy on validation and training data-set disappeared, but overall accuracy did not improve.

```
    # Simple data augmentation
    layers.RandomFlip("horizontal",
        input_shape=(img_height, img_width, 3)),
    layers.RandomRotation(0.1),
    layers.RandomZoom(0.1),
    layers.Rescaling(1./255,
        input_shape=(img_height, img_width, 3)),
    ...
```

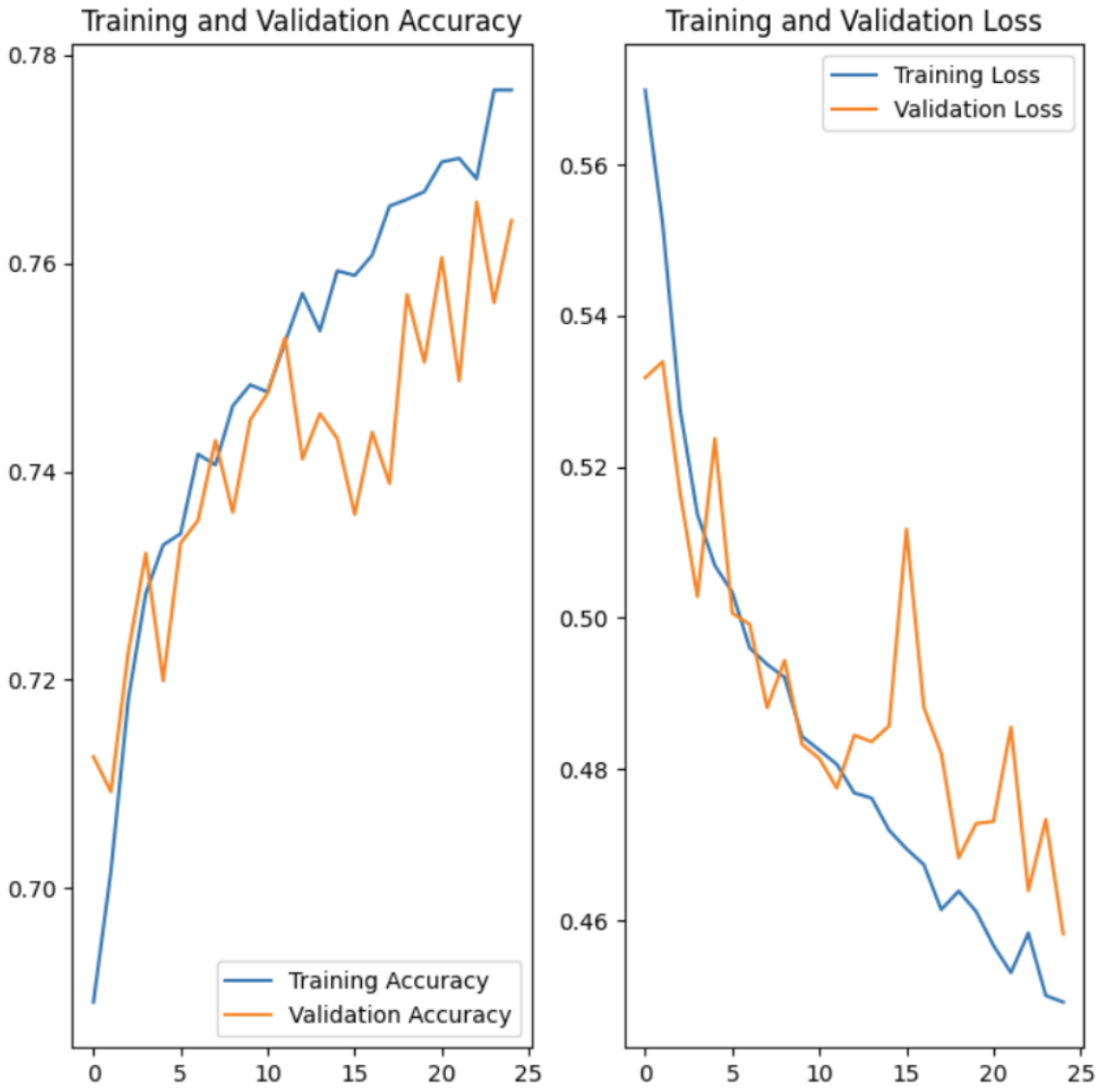


Figure 4.2: Accuracy and loss on training and validation data-set

4.1.2 Changing own built model to pretrained one

After building simple augmentation inside the model, pretrained model VGG19 was tested. At this point no fine-tuning was tested. Core part of VGG19 was adopted as base model, new fully connected layers was created and model was tested as a whole, but comparing to the own built model no big improvement among accuracy was observed.

```
vgg19_base = VGG19(include_top = False, weights = 'imagenet')

model = Sequential([
    ...
    vgg19_base,
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dropout(0.5),
    layers.Dense(2, activation='softmax')
])
```

Figure 4.3: Accuracy and loss on training and validation data-set

4.1.3 Genetic algorithm test

After running script in section 3.2.3 following outcomes were shout:

```
Top 3 models:
Features: 112, Dropout: 0.30066694639732416, LR:
    0.00010268824811419098, MR: 0.10969526221432405, Accuracy:
    0.788195788860321
Features: 116, Dropout: 0.3307050879739871, LR:
    0.00010355447549191398, MR: 0.11192529719018227, Accuracy:
    0.7878010272979736
Features: 114, Dropout: 0.2946322658174671, LR:
    0.00010783922249108772, MR: 0.10973000996323085, Accuracy:
    0.7872088551521301
```

The best model was saved and gave another 30 epochs for training. However, something similar to first experiment is observed and even though parameters in training set are getting better, they stuck on validation set. Maybe simple data augmentation was not enough to mitigate problem of overfitting for that data.

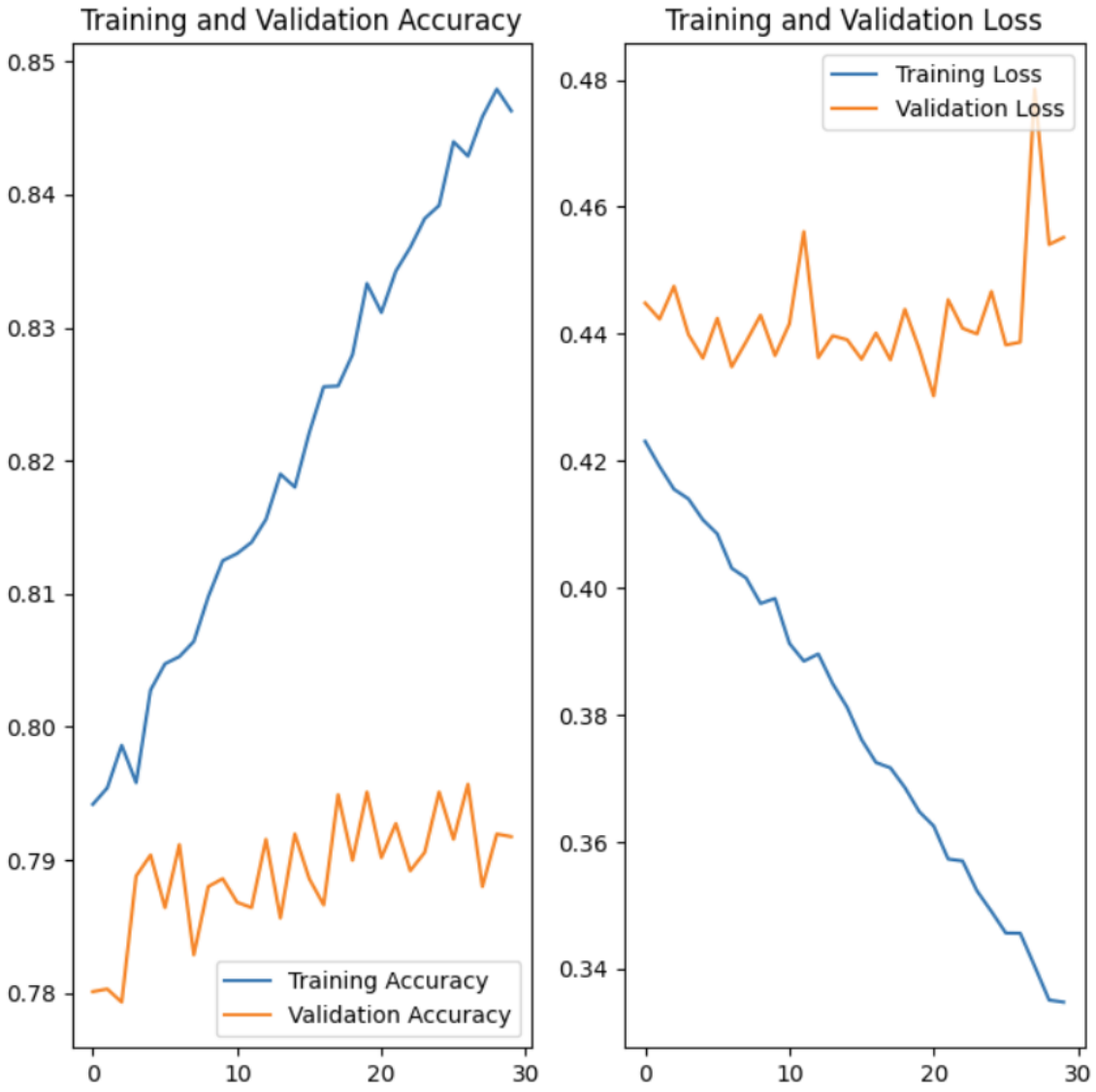


Figure 4.4: Accuracy and loss on training and validation data-set

4.1.4 Re-balancing data-set

At that point a lot of factors that could cause model to stop improve on validation data-set was considered. One of possible negative feature of data-set was a quite huge imbalance between 2 classes. One of the class had 2 times more representatives and after concluding that both classes was balanced by under-sampling more numerous class. Unfortunately, this also did not bring expected results - probably cause for reaching plateau on validation data-set was somewhere else.

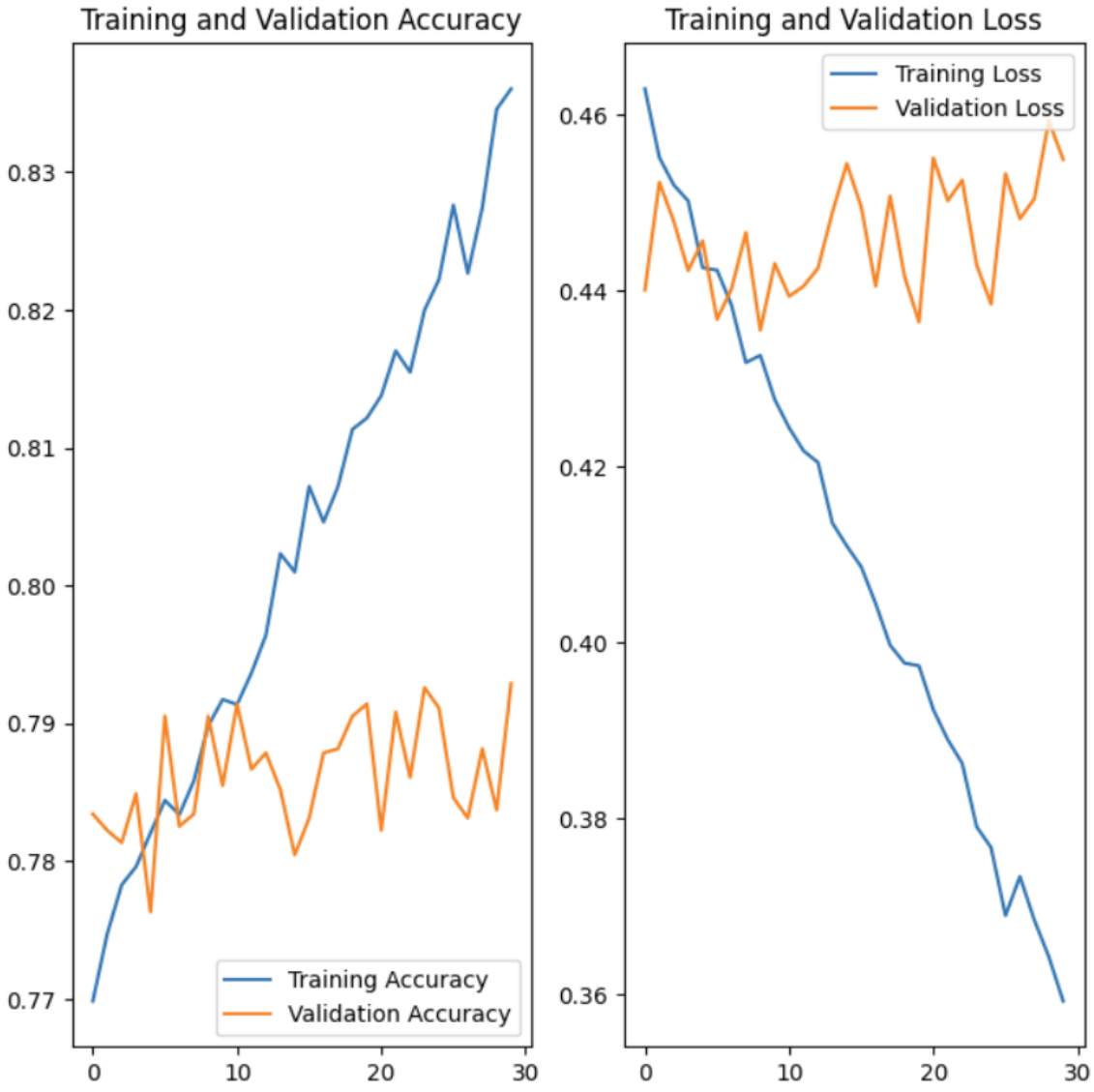


Figure 4.5: Accuracy and loss on training and validation data-set

4.1.5 Changing way data augmentation was performed

After checking several factors there was still one more to check - poor data augmentation. As images of skin lesion typically has circular shape rotating or flipping them do not augment data well. Therefore, additional class for data augmentation that performs it separately from training was something that needed to be tested.

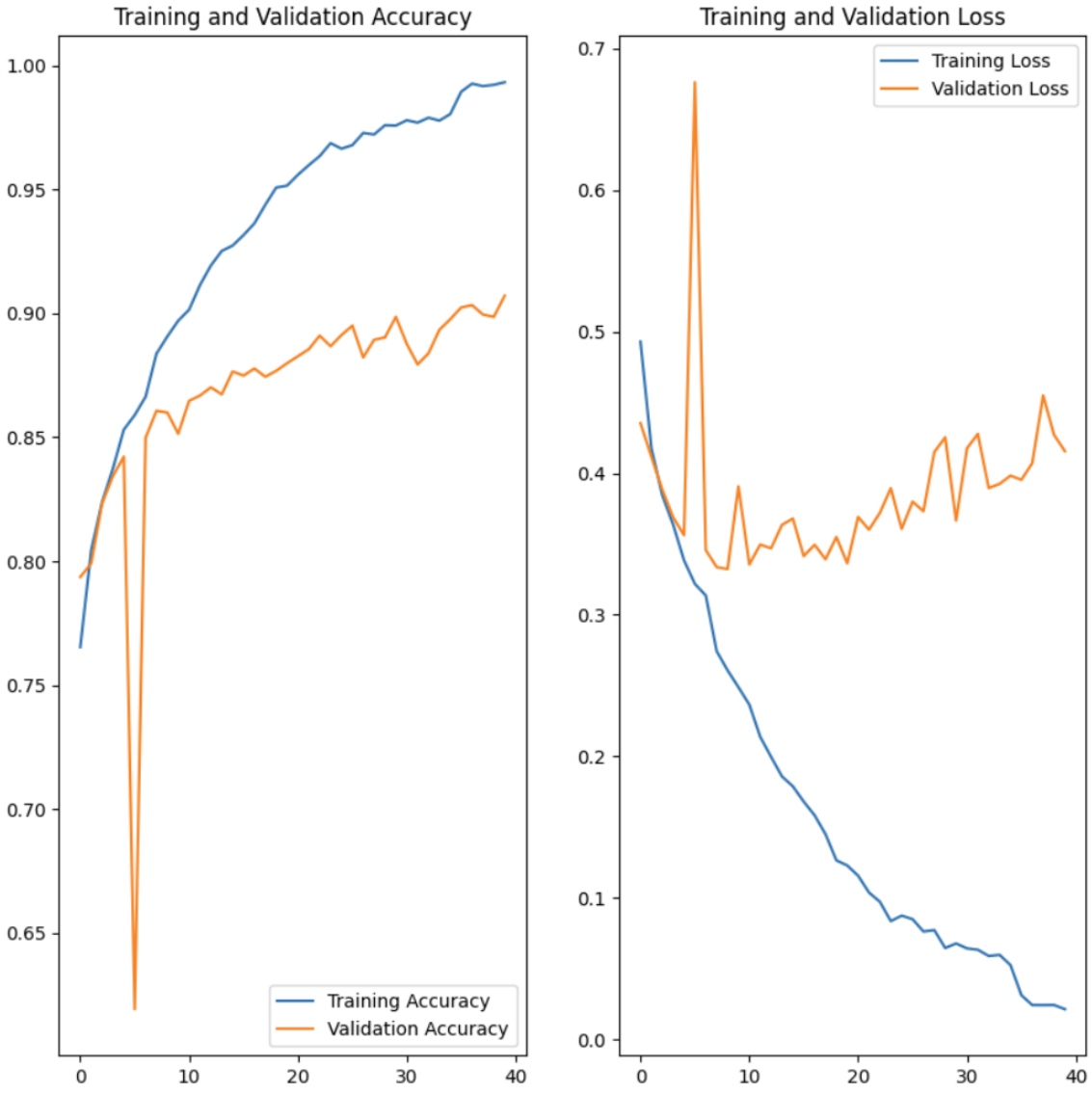


Figure 4.6: Accuracy and loss on training and validation data-set of the second step

That finally led to expected results. From this point scripts similar to seen in 3.2.4 was run. Even though new way for data augmentation was one of the biggest change here, also fine-tuning as well as adaptive learning was introduced here using callbacks that actively moderated learning rate. The successful model was using Xception architecture as an feature extractor. The model was trained in 2 steps:

- First step - pure fine-tuning (only fully connected layers are able to be trained)
- Second step - full training (all layers are enabled to be trained)

4.1.6 Mistake during training

As this was also the point where besides approach, environment was switched from Google Colab do Paperspace Notebooks, imbalanced data set was used by accident to training this model. That questioned the whole quality of trained model results. Models trained on imbalanced data-sets generally tends to learn to classify everything as the most numerous class. To check if imbalanced data-set influenced training of the newest model confusion matrix along with additional parameters were checked.

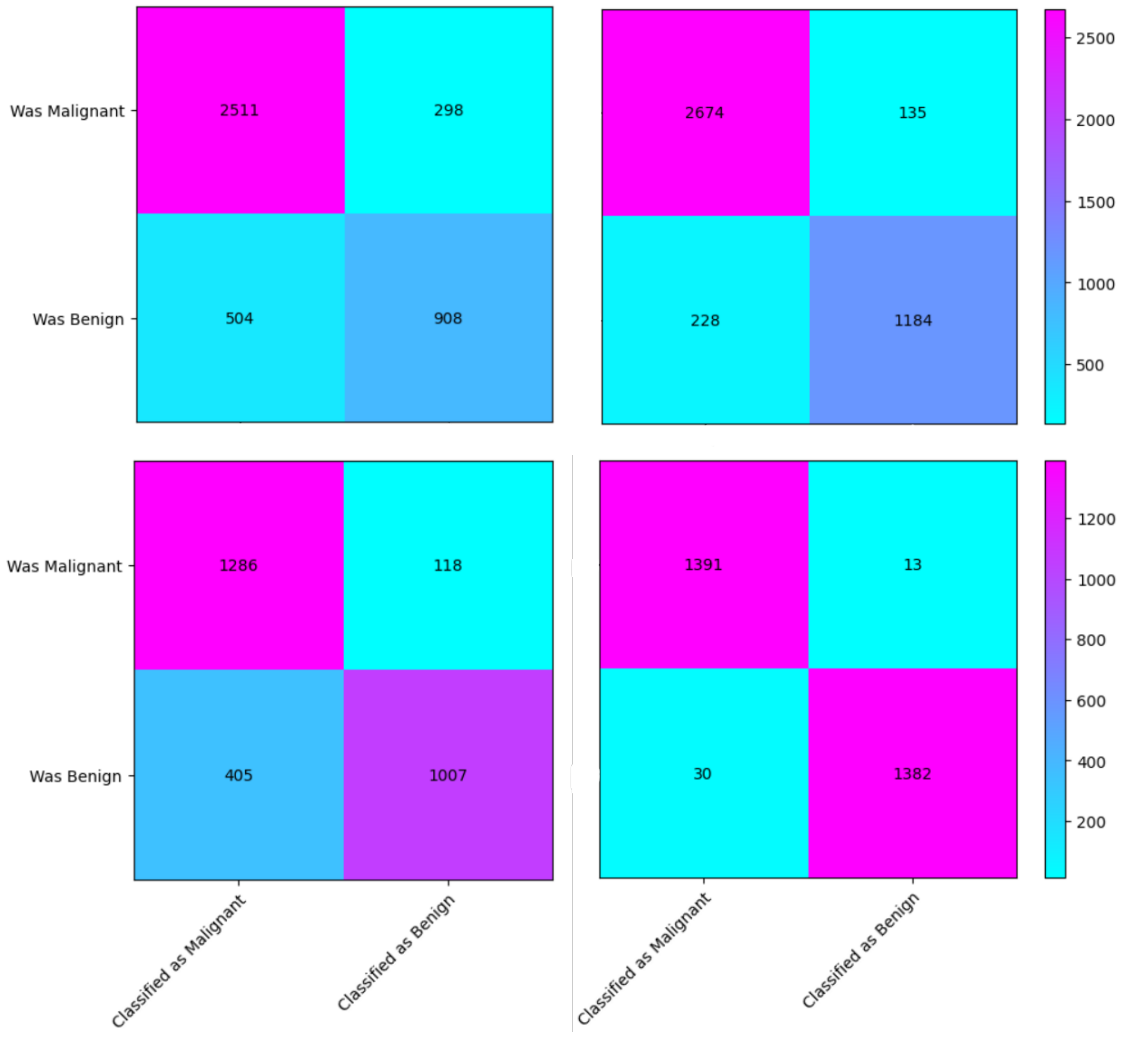


Figure 4.7: Confusion matrix for the best binary classification model, first column is the confusion matrix of this model after first step (pure fine-tuning) of training and second column the confusion matrix of this model after second step of training, first row represent performance on imbalanced data-set and second row performance in balanced data-set, therefore colorbars are unique for rows

Checking validation on balanced data-set turned out to yield even better results. When checking precision, recall and F-1 score they was **0.991**, **0.979** and **0.985** respectively.

4.2 Multiclass classification

4.2.1 Attempt to repeat successful performance of binary classifier

Hence training binary classification model on imbalanced data-set yield surprisingly satisfying results, attempt to repeat the same training environment for multiclass classification model were made. Looking just at the accuracy and loss values from the last 40 epochs of training, it can be assumed that model achieved accuracy of roughly about 85 %.

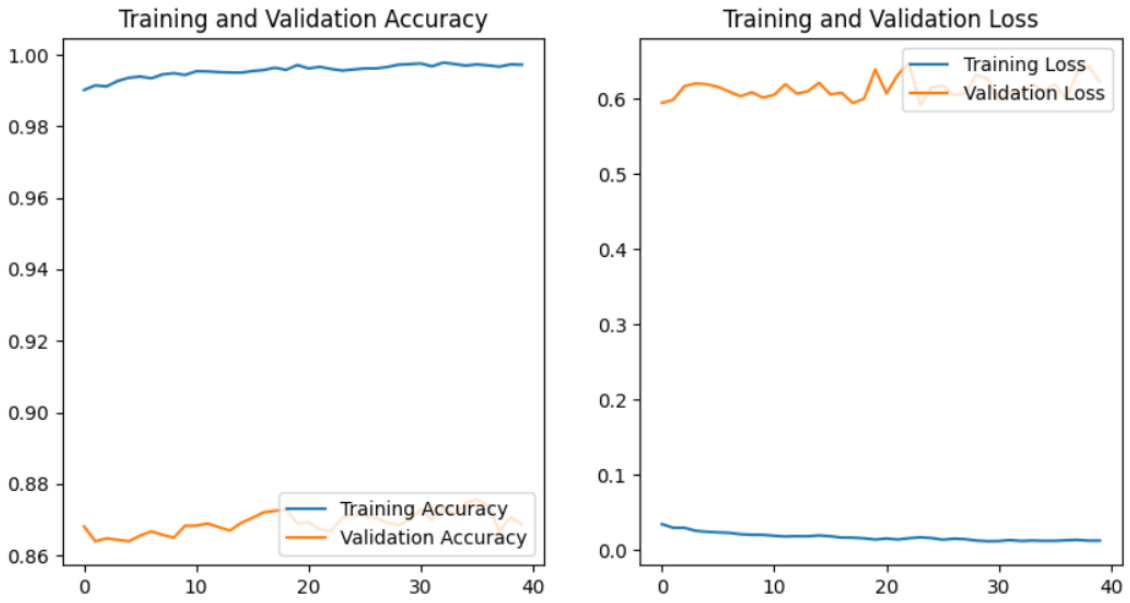


Figure 4.8: Accuracy and loss on training and validation data-set of the second step

Unfortunately, for non-binary classification difference between the most and the least numerous class was significantly bigger. That probably led to problem of achieving better results for validation or unbalanced data-set than for balanced data-set - that is the opposite way as it worked for binary-classification. Conclusion of that experiment is interesting and shows that when data imbalance is small, then it does not always influence quality of classifier.

Additional validation step showed that accuracy ultimately was only about 71 % on balanced data-set.

```
Evaluating model on imbalanced data-set after first step:
159/159 [==============================] - accuracy: 0.8577
Evaluating model on imbalanced data-set after second step:
159/159 [==============================] - accuracy: 0.8695
Evaluating model on balanced data-set after first step:
126/126 [==============================] - accuracy: 0.7143
Evaluating model on balanced data-set after second step:
126/126 [==============================] - accuracy: 0.7123
```

4.2.2 Rebalancing multiclass data-set

Rebalancing data-set and performing training on it led to small improvement in accuracy in validation data-set.

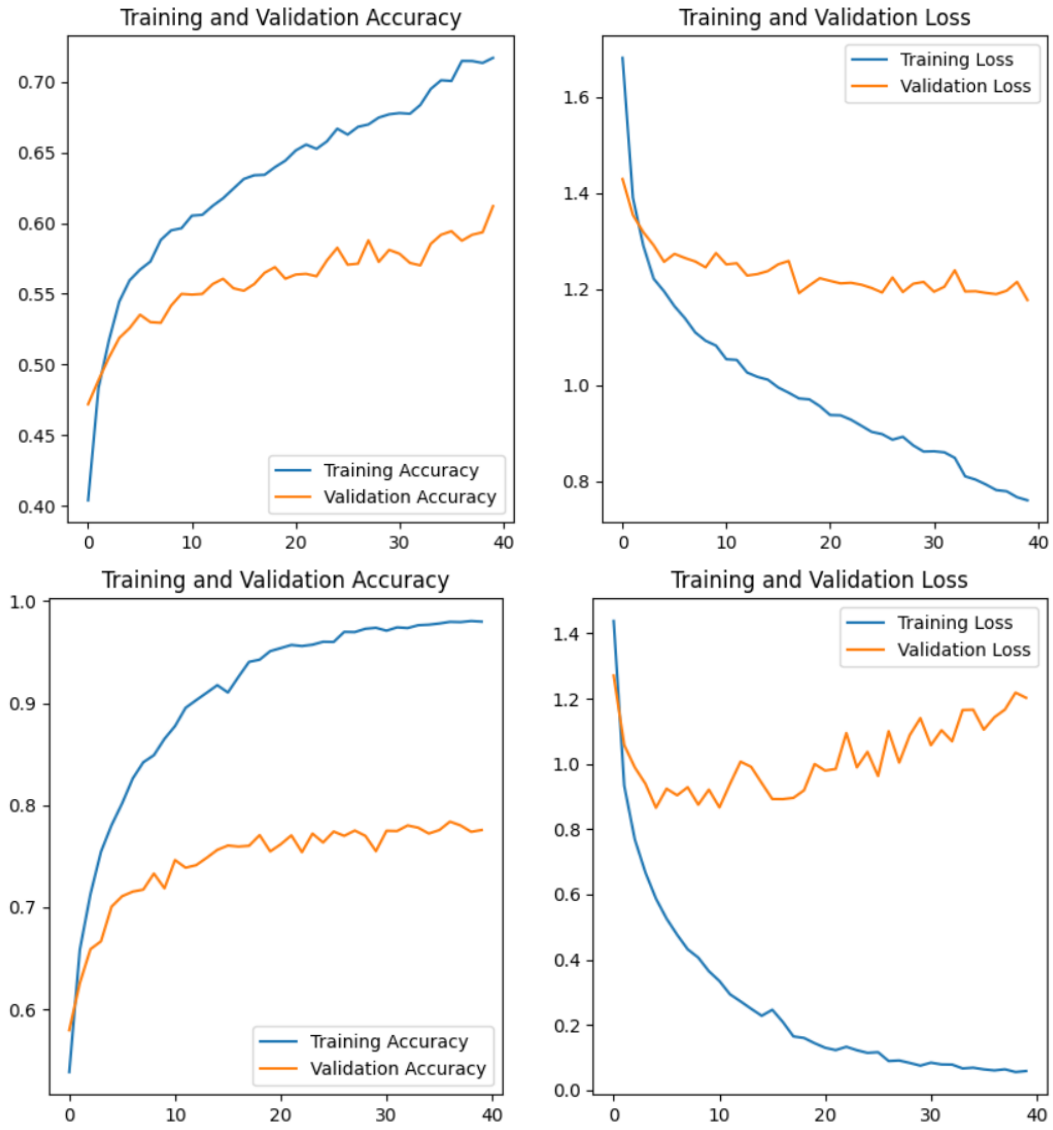


Figure 4.9: Accuracy and loss on training and validation data-set, in first row for first step and in second row for second step of training

Additional validation step showed that accuracy ultimately raised to about 78 % on balanced data-set.

```
Evaluating model on imbalanced data set after first step:
159/159 [==============================] - accuracy: 0.6236
Evaluating model on imbalanced data set after second step:
159/159 [==============================] - accuracy: 0.7444
Evaluating model on balanced data set after first step:
126/126 [==============================] - accuracy: 0.6105
Evaluating model on balanced data set after second step:
126/126 [==============================] - accuracy: 0.7827
```

4.2.3 Changing number of neurons used for feature classification in model

As there theoretically should be bigger gradation of features distinguishable between classes, more neurons was added to fully-connected layers.

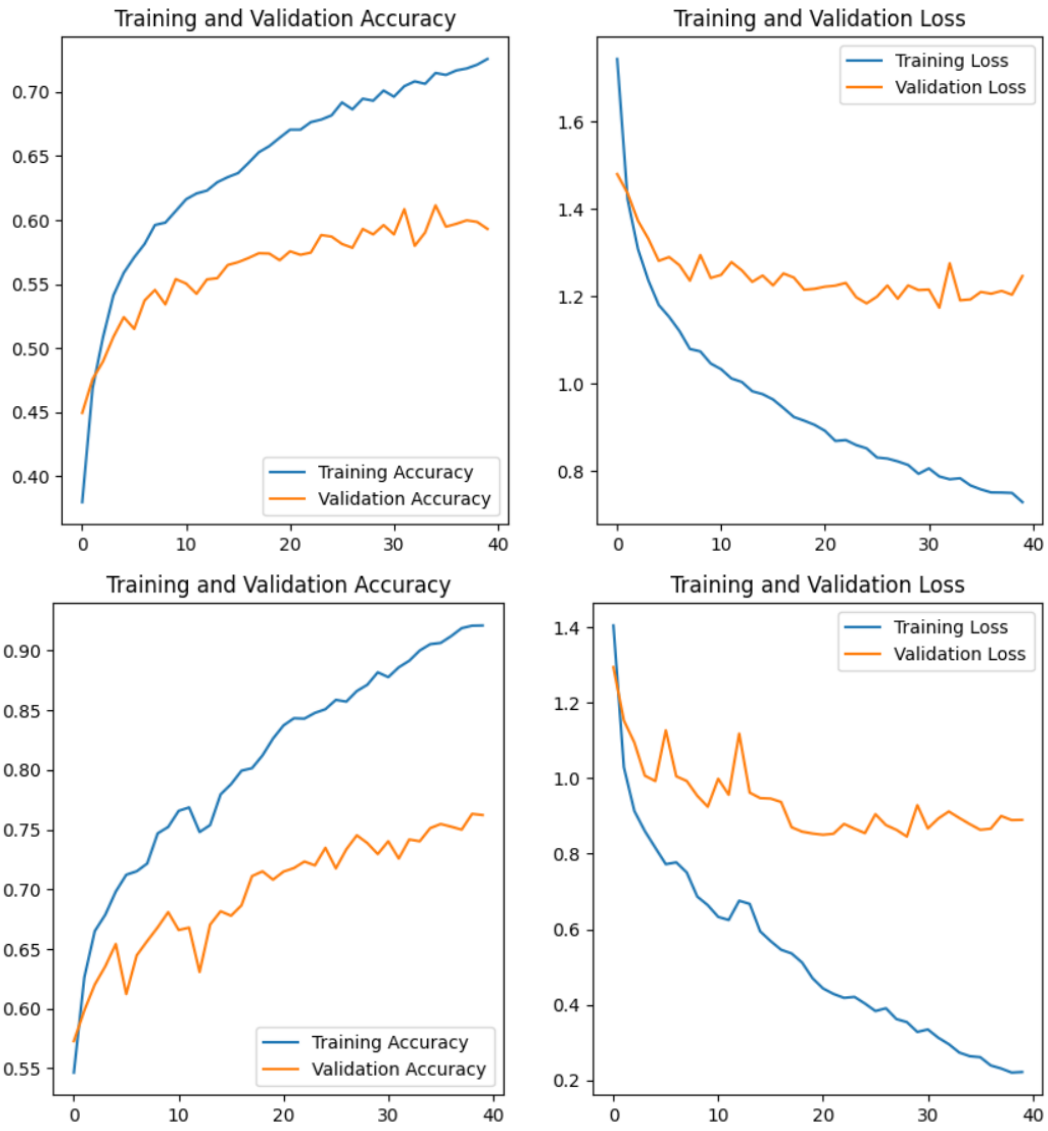


Figure 4.10: Accuracy and loss on training and validation data-set, in first row for first step and in second row for second step of training

Accuracy of model was not any bigger. Additional validation step showed that ultimately accuracy was even slightly below 78 % on balanced data-set.

```
Evaluating model on imbalanced data set after first step:
159/159 [==============================] - accuracy: 0.6281
Evaluating model on imbalanced data set after second: 0.7499
Evaluating model on balanced data set after first step:
126/126 [==============================] - accuracy: 0.5913
Evaluating model on balanced data set after second step:
126/126 [==============================] - accuracy: 0.7740
```

4.2.4 Confusion matrices for multiclass model

In order to find how data imbalance influenced learning process of multi-class model confusion matrices were created. Actual data balancing technique can also be verified here.

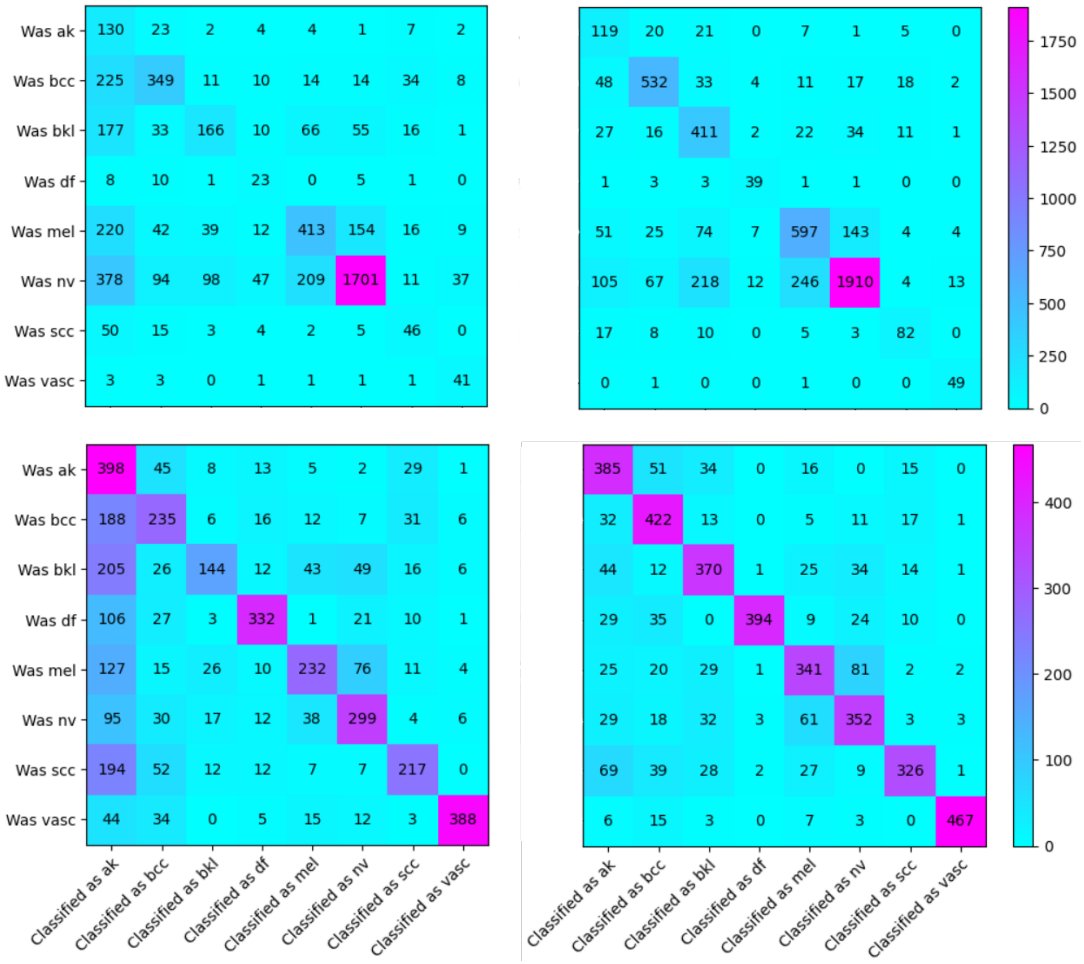


Figure 4.11: Confusion matrix for the best multiclass classification model, first column is the confusion matrix of this model after first step (pure fine-tuning) of training and second column the confusion matrix of this model after second step of training, first row represent performance on imbalanced data-set and second row performance in balanced data-set, therefore colorbars are unique for rows

The most valuable information could be found in the second row, second column matrix, as it is confusion matrix for the ultimately trained model validated on balanced data-set. The poorest recognizable classes there are scc, mel and nv. At bar chart 2.8 cardinality of **scc** is one of the smallest among classes, which could be aligned with scc being the poorest recognizable class. This would suggest augmentation of the data-set during balancing was too excessive for the least numerous classes and images ended up being not that similar. However, the second and the third poorest recognizable class are one of the best numerous ones. At the other hand this could suggest that augmentation of the classes with the smallest cardinality was too limited and only highly numerous classes had enough difference among itself to pose a problem during classification for model. Therefore as symptoms of too excessive and too limited data augmentation could be seen then probably data balancing was

conducted properly and it does not favored any class type (regarding cardinality).

In this matrix it could be also observed that model did not prioritized any particular class regarding cardinality. If results in matrix were sum up along columns, then it would also be noticeable that the most frequent classified classes are ak and bcc that were not the lest nor the most numerous classes.

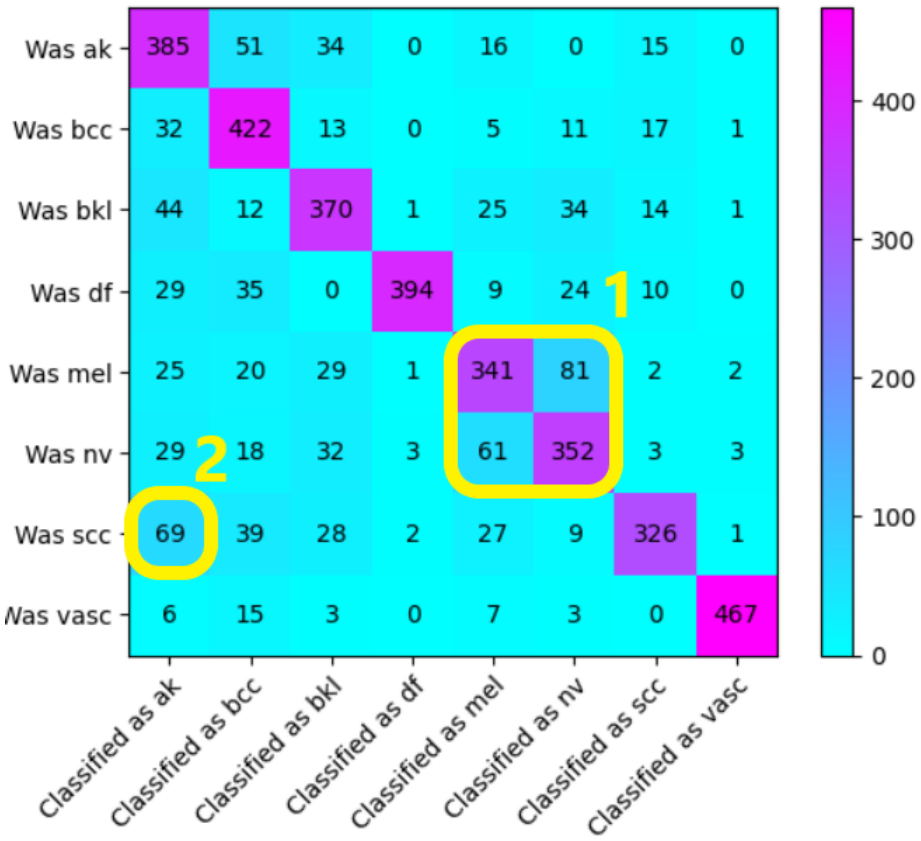


Figure 4.12: Class-class dependency problems in confusion matrix for the ultimately trained model validated on balanced data-set

When it comes to class-class dependency, two curios mismatches are evident. The first one is common mistake between classyfing images from class mel as nv, and the way around. If the binary classification was mapped just from multiclass model predication, then this mismatch would lead to mistaking malignant change (nv) with cancer change (mel) and that is highly undesired. The second problem is frequent classification of scc as ak. Again in the same scenario of generating binary predication for multiclass model that could lead to classifying benign change (scc) as malignant one (ak).

Project summary

5.1 Conclusion

The general assumptions of the project has been met as two separate models were created and tested. Measured results of the best model for each of classifications were 98 % percent of accuracy for binary classification and 78 % percent of accuracy for 8-class classification problem. Creating separate model to solve binary classification problem was better approach than mapping multiclass prediction to binary classification, due to common mismatches between malignant and benign classes that occurred during validation of multiclass classification model. The quality of data-set is crucial for proper training and validation of the problem - the better input is, the better model can perform. When it comes to binary classification model, as prove that model really focuses on changes itself - not color of outer skin, or any other parameter - grad cam algorithm was used to create a heat map of the most influential parts of images regarding class prediction.

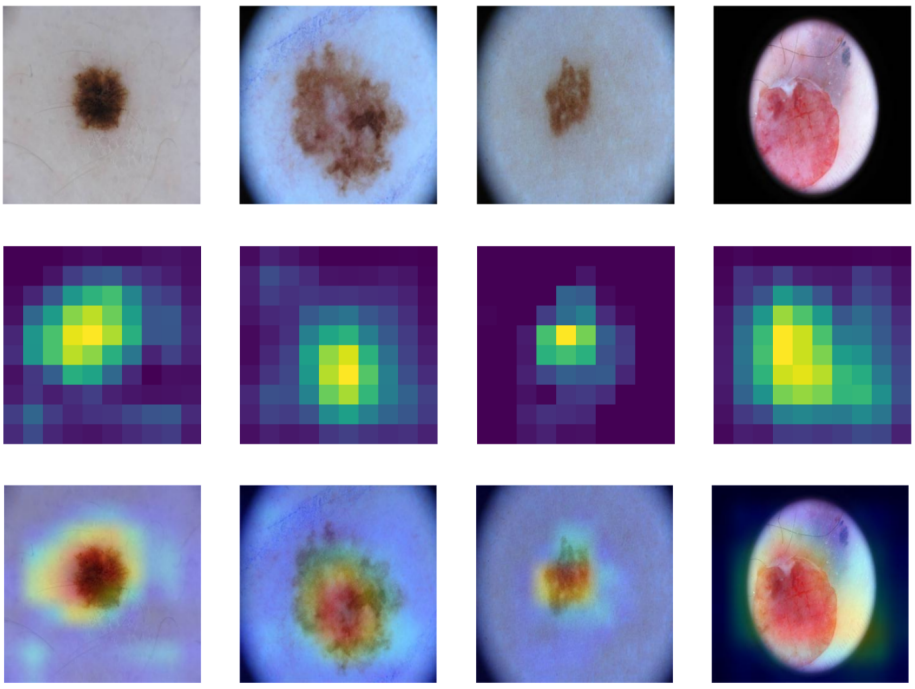


Figure 5.1: The most influential part of images regarding class prediction

References

6.1 Materials' link

- https://www.tensorflow.org
- https://numpy.org/
- https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms
- https://www.ibm.com/topics/random-forest
- https://towardsdatascience.com/linear-discriminant-analysis-explained

6.2 Project's link

https://github.com/rochfedorowicz/SkinLesionClassification