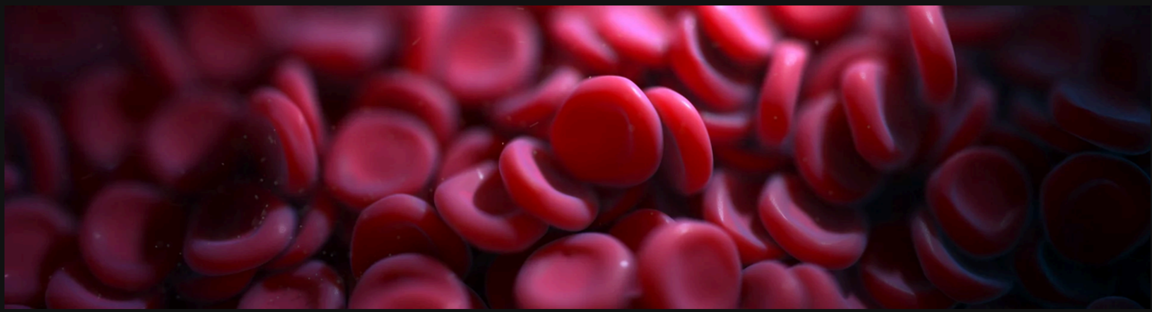
🩸 Blood Cell Cancer Prediction - Classification & Segmentation by Alexander Daniel Rios



Overview

Acute lymphocytic leukemia (ALL) is a type of cancer of the blood and bone marrow (the spongy tissue inside bones where blood cells are made).

Acute lymphocytic leukemia is the most common type of cancer in children, and treatments result in a good chance for a cure. Acute lymphocytic leukemia can also occur in adults, though the chance of a cure is greatly reduced.

Causes

Acute lymphocytic leukemia occurs when a bone marrow cell develops changes (mutations) in its genetic material or DNA. A cell's DNA contains the instructions that tell a cell what to do. Normally, the DNA tells the cell to grow at a set rate and to die at a set time. In acute lymphocytic leukemia, the mutations tell the bone marrow cell to continue growing and dividing.

When this happens, blood cell production becomes out of control. The bone marrow produces immature cells that develop into leukemic white blood cells called lymphoblasts. These abnormal cells are unable to function properly, and they can build up and crowd out healthy cells.

ALL Subtypes

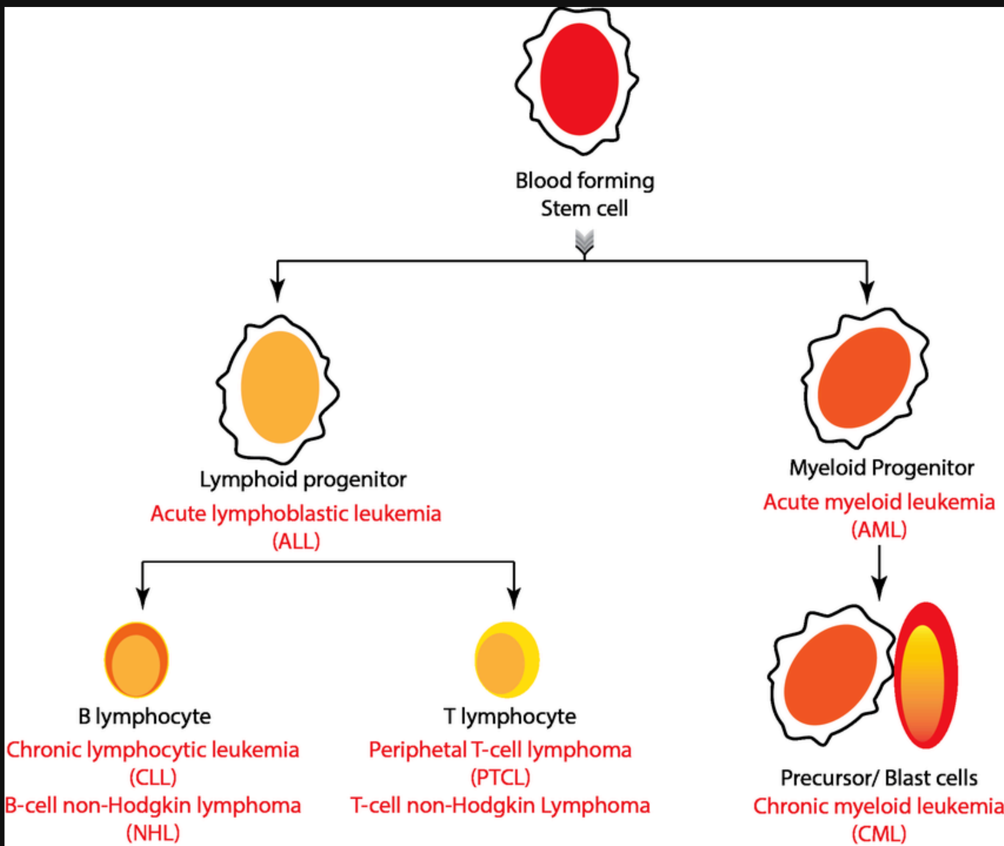
Doctors classify acute lymphoblastic leukemia (ALL) into subtypes by using various tests. It's important to get an accurate diagnosis since your subtype plays a large part in deciding the type of treatment you'll receive. Depending on your ALL subtype, the doctor will determine

- The type of drug combination needed for your treatment
- The length of time you'll need to be in treatment

- Other types of treatment that may be needed to achieve the best outcomes

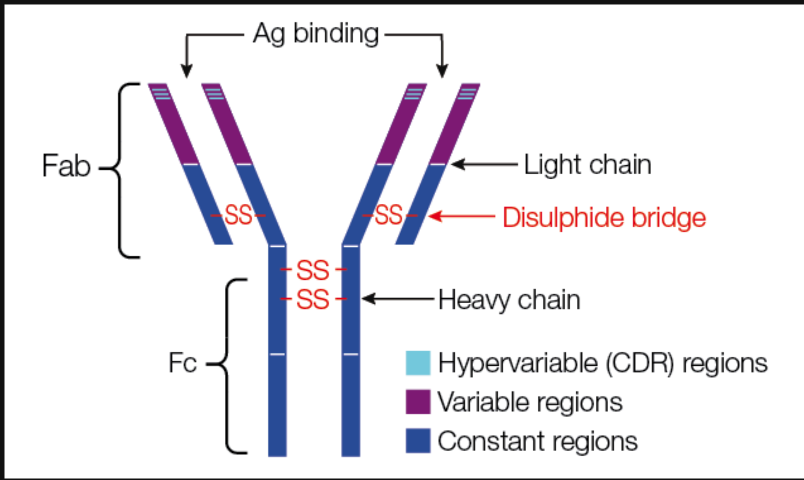
Based on immunophenotyping of the leukemia cell, the World Health Organization (WHO) classifies ALL into two main subtypes.

- **B-cell lymphoblastic leukemia/lymphoma**: This subtype begins in immature cells that would normally develop into B-cell lymphocytes. This is the most common ALL subtype. Among adults, B-cell lineage represents 75 percent of cases.
- **T-cell lymphoblastic leukemia**: This subtype of ALL originates in immature cells that would normally develop into T-cell lymphocytes. This subtype is less common, and it occurs more often in adults than in children. Among adults, T-cell lineage represents about 25 percent of cases.



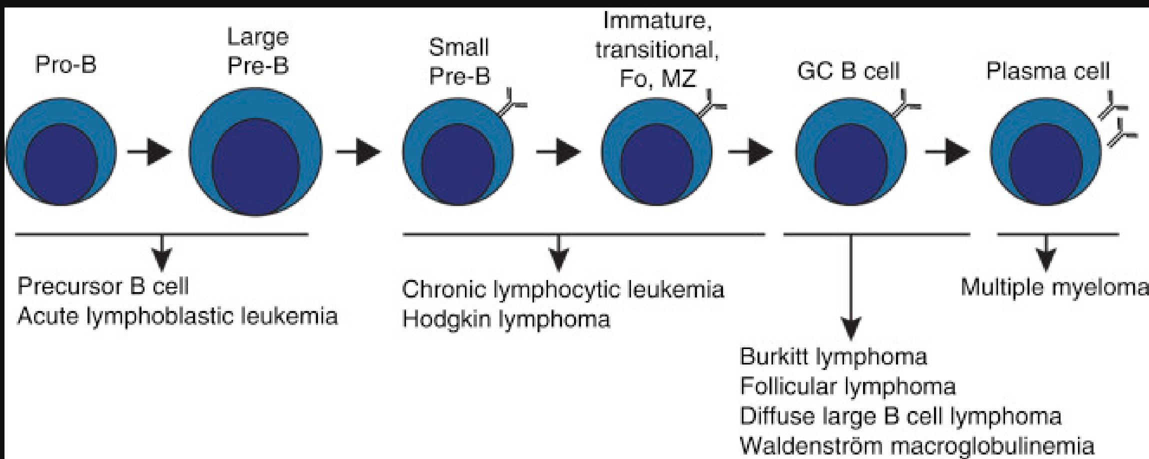
The final task of the lymphocytes (B cells) developed in the bone marrow (BM) is the production of antigen-specific immunoglobulins (IGS), which function as antibodies. IGS are proteins secreted by or present on the surface of B cells, assembled from identical pairs of heavy (H) and light (L) chains.

Antibody structure



Before being capable of producing Ag-specific IGS, B cells must undergo a number of transformations, first in the BM and subsequently in the lymph nodes (LNs). The first stages of B-cell development occur in the BM, where pro-B cells first rearrange the Ig H chain gene to become pre-B cells. Pre-B cells continue this somatic recombination process by rearranging the L chain to become immature B cells, expressing IgM on their surface.

B-cell development



About the dataset

In this project I used the following dataset: Blood cell cancer all 4 class

You can download this dataset with the following code:

```
!pip install kagglehub

import kagglehub
blood_cell_cancer_all_4class_path =
kagglehub.dataset_download('mohammadamireshraghi/blood-cell-
cancer-all-4class')
```

The images of this dataset were prepared in the bone marrow laboratory of Taleqani Hospital (Tehran, Iran).

This dataset consisted of **3242 PBS images** from 89 patients suspected of ALL, whose blood samples were prepared and stained by skilled laboratory staff. This dataset is divided into two classes *benign* and *malignant*. The former comprises hematogenous, and the latter is the ALL group with three subtypes of malignant lymphoblasts: **Early Pre-B**, **Pre-B**, and **Pro-B ALL**.

All the images were taken by using a Zeiss camera in a microscope with a 100x magnification and saved as JPG files. A specialist using the flow cytometry tool made the definitive determination of the types and subtypes of these cells.

Table of Contents

From Kaggle to Colab environment

In [ ]:

```
# Cell only for Colab

!pip install kagglehub

import kagglehub
import os
import shutil

blood_cell_cancer_all_4class_path = kagglehub.dataset_download('mohammadamir
```

```
for dir in ["kaggle", "kaggle/input", "kaggle/working", "kaggle/working/mode
    if not os.path.exists(dir):
        os.mkdir(dir)

target_dir = 'kaggle/input'

file_names = os.listdir(blood_cell_cancer_all_4class_path)

for file_name in file_names:
    shutil.move(os.path.join(blood_cell_cancer_all_4class_path, file_name), t
```

Installing packages

In [ ]:

```
!pip install mplcyberpunk
```

Importing packages

In [ ]:

```
import numpy as np
import pandas as pd
import os
import random

import keras
import tensorflow as tf
from tensorflow.keras import backend as K
from tensorflow.keras.backend import epsilon
from tensorflow.keras.utils import load_img
from tensorflow.data.experimental import cardinality
from tensorflow.image import (random_flip_left_right,
                              random_flip_up_down,
                              random_brightness,
                              random_contrast,
                              random_jpeg_quality,
                              random_saturation,
                              resize)

from tensorflow.keras import Model, Input
from tensorflow.keras.layers import (Conv2D,
                                     Dense,
                                     Dropout,
                                     Flatten,
                                     GlobalAveragePooling2D,
                                     MaxPooling2D,
                                     BatchNormalization,
                                     ReLU,
                                     Conv2DTranspose,
                                     Concatenate)
```

```
from tensorflow.keras.regularizers import L2
from tensorflow.keras.losses import CategoricalCrossentropy, Dice
from tensorflow.keras.metrics import AUC, BinaryIoU, BinaryAccuracy
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import Callback, EarlyStopping, ReduceLROnPla
from tensorflow.keras.applications import VGG16
from tensorflow.keras.utils import plot_model

from sklearn.model_selection import train_test_split
from sklearn.utils.class_weight import compute_class_weight
from sklearn.metrics import f1_score, accuracy_score, precision_score, recal

import cv2

from IPython.display import clear_output

import matplotlib.pyplot as plt
import mplcyberpunk
plt.style.use("cyberpunk")
```

Setting for reproducibility

In [ ]:
```
seed_value = 42
os.environ['PYTHONHASHSEED'] = str(seed_value)
random.seed(seed_value)
np.random.seed(seed_value)
```

Gathering image pathfiles

In [ ]:
```
imgs_path = []
classes = []

data_dir = "/kaggle/input/blood-cell-cancer-all-4class/Blood cell Cancer [AL
for dirpath, dirnames, filenames in os.walk(data_dir):
    if len(classes) == 0:
        classes = dirnames
    for filename in filenames:
        imgs_path.append(os.path.join(dirpath, filename))
```

From images metadata to Pandas DataFrame

In [ ]:
```
def get_class(x):
    for class_ in classes:
        if class_ in x:
            return class_

df_imgs = pd.DataFrame(data=imgs_path, columns=["pathfiles"])
df_imgs["type_cell"] = df_imgs.map(get_class)
```

```python
widths = []
heights = []
for row in df_imgs.iterrows():
    h, w = load_img(row[1].pathfiles).size
    widths.append(w)
    heights.append(h)

df_imgs["width"] = widths
df_imgs["heigth"] = heights

df_imgs
```

Out[ ]:

	pathfiles	type_cell	width	heigth
0	/kaggle/input/blood-cell-cancer-all-4class/Blo...	[Malignant] early Pre-B	768	1024
1	/kaggle/input/blood-cell-cancer-all-4class/Blo...	[Malignant] early Pre-B	768	1024
2	/kaggle/input/blood-cell-cancer-all-4class/Blo...	[Malignant] early Pre-B	768	1024
3	/kaggle/input/blood-cell-cancer-all-4class/Blo...	[Malignant] early Pre-B	768	1024
4	/kaggle/input/blood-cell-cancer-all-4class/Blo...	[Malignant] early Pre-B	768	1024
...	...	...	...	...
3237	/kaggle/input/blood-cell-cancer-all-4class/Blo...	Benign	768	1024
3238	/kaggle/input/blood-cell-cancer-all-4class/Blo...	Benign	768	1024
3239	/kaggle/input/blood-cell-cancer-all-4class/Blo...	Benign	768	1024
3240	/kaggle/input/blood-cell-cancer-all-4class/Blo...	Benign	1920	2560
3241	/kaggle/input/blood-cell-cancer-all-4class/Blo...	Benign	768	1024

3242 rows × 4 columns

Setting for normalization of image sizes

In [ ]:
```python
df_imgs.width.value_counts()
```

Out[ ]:
```
width
768     3196
1920      46
Name: count, dtype: int64
```

The common factors between 768 and 1920 are [1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, 128, 192, 384], therefore we will resize these images to 192.

In [ ]:
```python
df_imgs.heigth.value_counts()
```

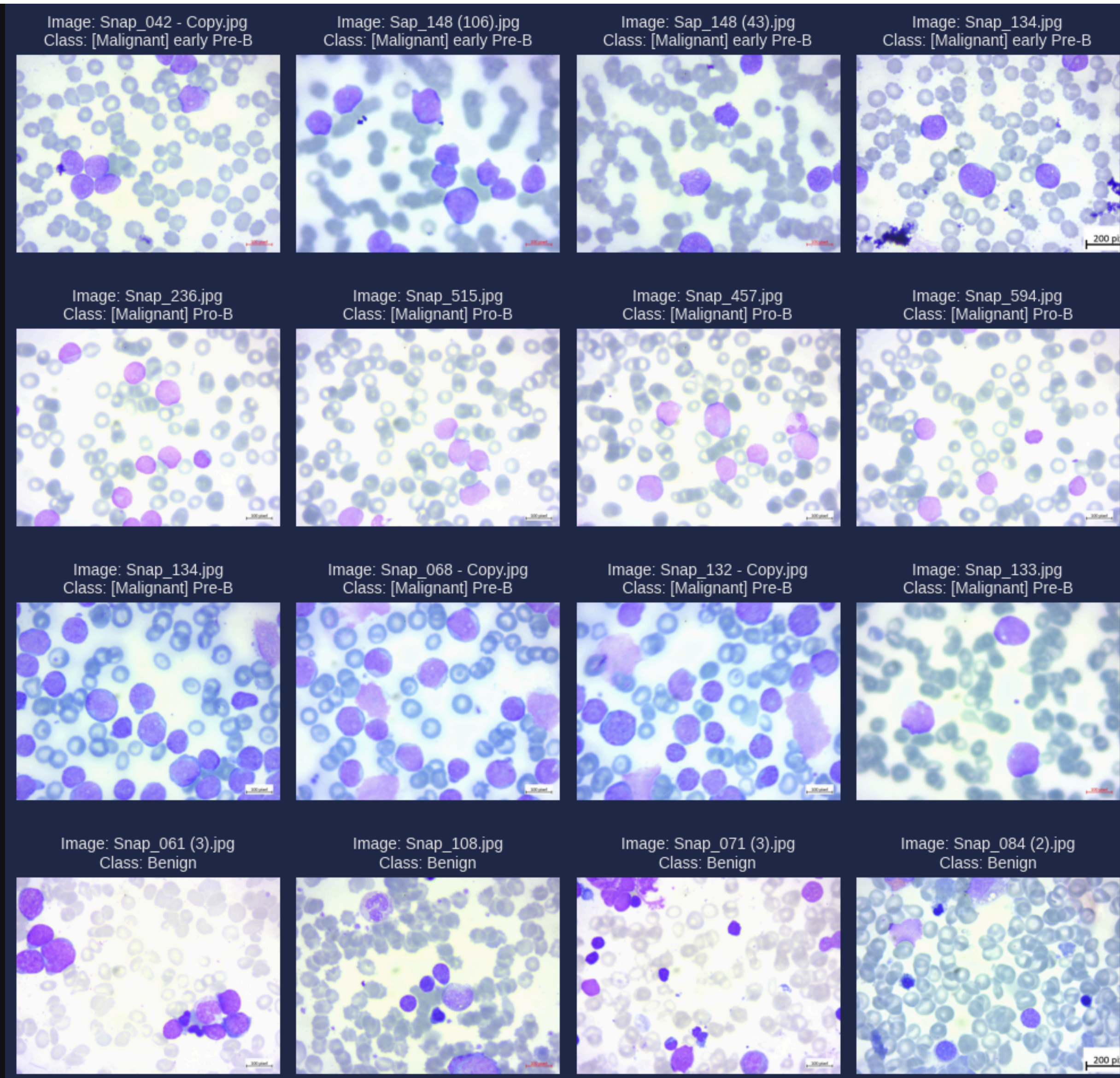
Out[ ]:
```
heigth
1024    3196
2560      46
Name: count, dtype: int64
```

The common factors between 1024 and 2560 are [1, 2, 4, 8, 16, 32, 64, 128, 256, 512], therefore we will resize these images to 256.

In [ ]:
```
img_shape = (192, 256)
```

EDA: Visualization of the image dataset

In [ ]:
```
def random_img(df, cell=None, n=4):
    if cell:
        aux = df[df["type_cell"] == cell]
    else:
        aux = df
    return aux.loc[random.sample(list(aux.index), n)]

plt.subplots(4, 4, figsize=(10, 10))

for x, class_ in enumerate(classes):
    imgs = random_img(df_imgs, cell=class_, n=4)
    for i, row in enumerate(imgs.iterrows()):
        plt.subplot(4, 4, (x*4)+(i+1))
        plt.imshow(load_img(row[1].pathfiles))
        plt.title(f"Image: {row[1].pathfiles.split('/')[-1]}\nClass: {class_}
        plt.axis("off")
plt.tight_layout()
```

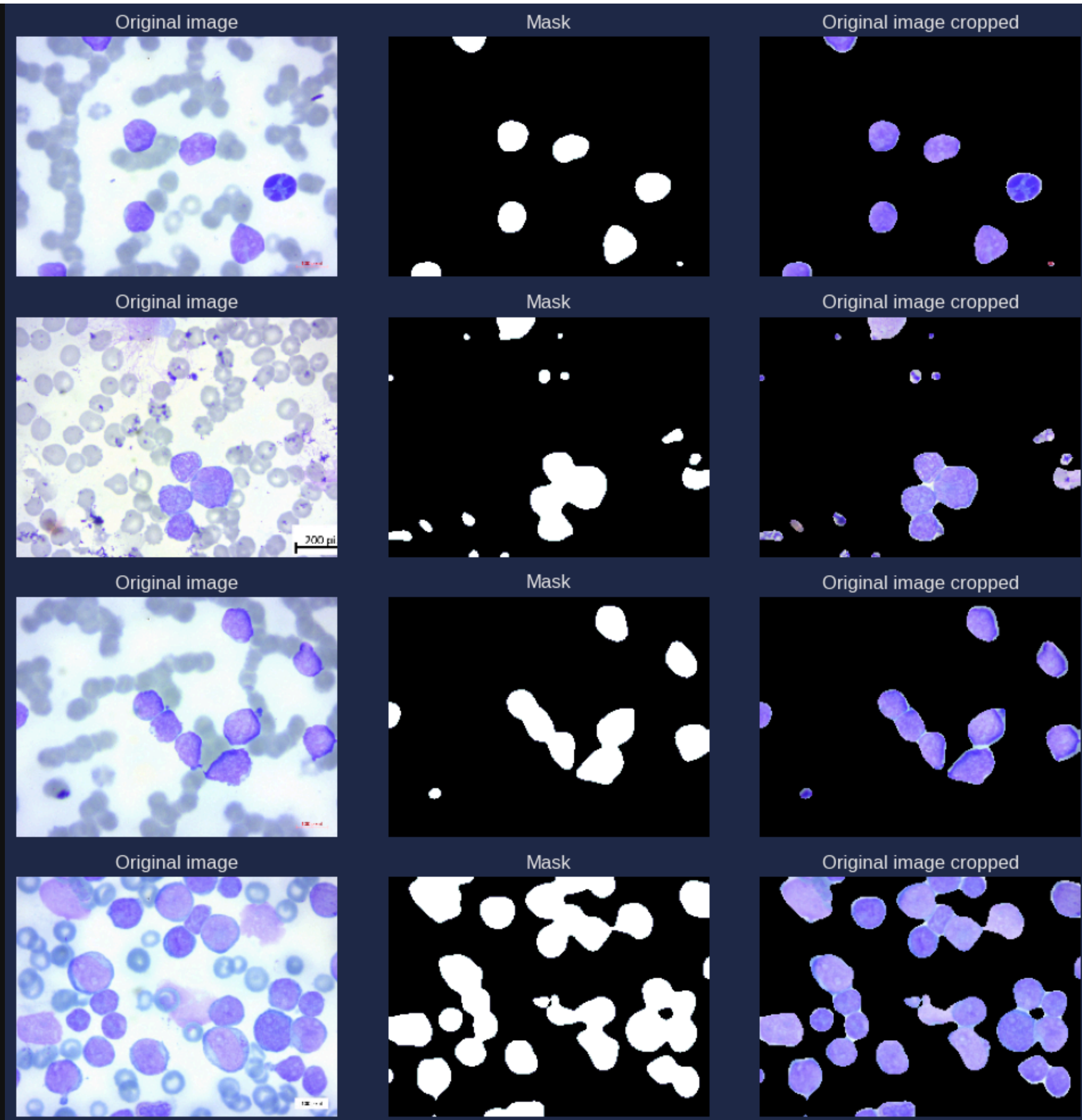


Extract the masks from the images

In [ ]:

```
def get_image(pathfile):
    return cv2.cvtColor(cv2.imread(pathfile), cv2.COLOR_BGR2RGB)

def RGB2LAB(image):
    image_lab = cv2.cvtColor(image, cv2.COLOR_RGB2LAB)
    l, a, b = cv2.split(image_lab)
    return l, a, b

def get_mask(image):
    l, a, b = RGB2LAB(image)
    a_blur = cv2.GaussianBlur(a, (19, 19), 0)
    _, thresh_img = cv2.threshold(a_blur, 200, 255, cv2.THRESH_BINARY+cv2.TH
    kernel = np.ones((1, 1),np.uint8)
    return cv2.morphologyEx(thresh_img, op=cv2.MORPH_CLOSE, kernel=kernel, i
```

```
def apply_mask(image, mask):
    return cv2.bitwise_and(image, image, mask=mask)

plt.subplots(4, 3, figsize=(10, 10))
imgs = random_img(df_imgs, n=4)

for i, row in enumerate(imgs.iterrows()):
    # Read image
    image = get_image(pathfile=row[1].pathfiles)
    image = resize(image, size=img_shape, method="nearest", antialias=True).
    mask = get_mask(image)
    image_cropped = apply_mask(image, mask)

    plt.subplot(4, 3, (i*3)+1)
    plt.imshow(image)
    plt.axis("off")
    plt.title("Original image")

    plt.subplot(4, 3, (i*3)+2)
    plt.imshow(mask, cmap="bone")
    plt.axis("off")
    plt.title("Mask")

    plt.subplot(4, 3, (i*3)+3)
    plt.imshow(image_cropped)
    plt.axis("off")
    plt.title("Original image cropped");

plt.tight_layout()
plt.show()
```



Creating training, validation and testing image set

In [ ]:
```
df_full_train, df_test = train_test_split(df_imgs, test_size=0.15, stratify=
df_train, df_val = train_test_split(df_full_train, test_size=1-(7/8.5), stra
```

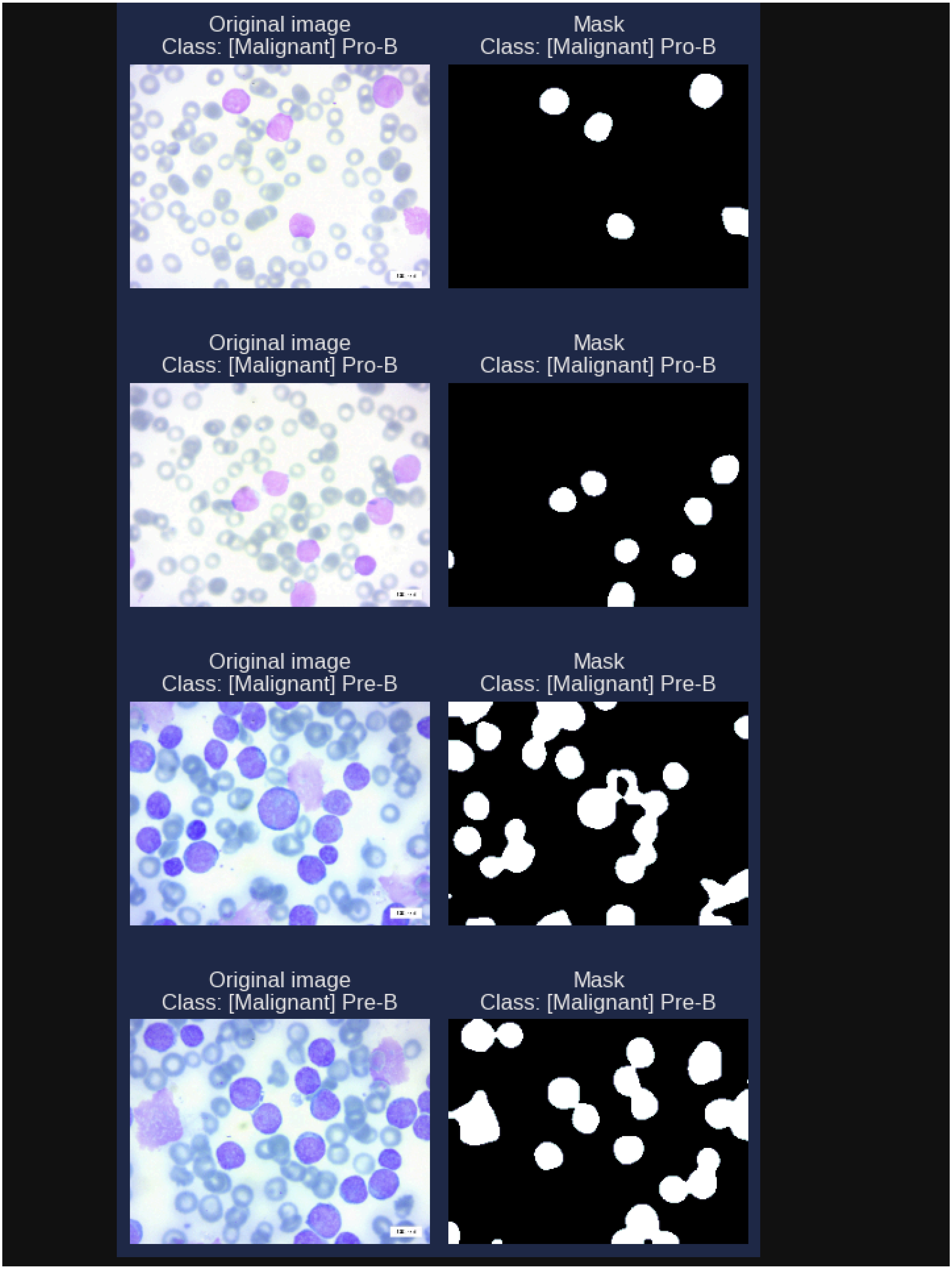
Creating TFRecord files for contain our data

In [ ]:
```
def createTFRecord(pathfile, dataset, size_img):
    #options = tf.io.TFRecordOptions(compression_type="GZIP", compression_le
    writer = tf.io.TFRecordWriter(pathfile,
```

```
                                        #options=options
                                       )
    cnt_written_img = 0
    for index, row in dataset.iterrows():
        image = get_image(row.pathfiles)
        image = resize(image, size=img_shape, method="nearest", antialias=Tr
        mask = np.array(get_mask(image))

        class_ = np.argmax(np.array(classes) == row.type_cell)
        image_data, mask_data = image.tobytes(), mask.tobytes()

        example = tf.train.Example(features=tf.train.Features(feature={
            'image': tf.train.Feature(bytes_list=tf.train.BytesList(value=[i
            'mask': tf.train.Feature(bytes_list=tf.train.BytesList(value=[ma
            'class': tf.train.Feature(int64_list=tf.train.Int64List(value=[c
        }))

        writer.write(example.SerializeToString())
        cnt_written_img += 1
    return cnt_written_img

def parse(feature):
    features = tf.io.parse_single_example(
        feature,
        features={
        'image': tf.io.FixedLenFeature([], tf.string),
        'mask': tf.io.FixedLenFeature([], tf.string),
        'class': tf.io.VarLenFeature(tf.int64),
    })
    image = tf.reshape(tf.io.decode_raw(features['image'], out_type=tf.uint8
    mask = tf.reshape(tf.io.decode_raw(features['mask'], out_type=tf.uint8),
    class_ = tf.sparse.to_dense(features["class"])
    return image, mask, class_

autotune = tf.data.AUTOTUNE
create_tensorflow_record = True
DIR_TFRECORD = "/kaggle/working/blood_cell_cancer_with_mask"

if create_tensorflow_record == True:
    createTFRecord(pathfile=DIR_TFRECORD+"_train.tfrecord", dataset=df_train,
    createTFRecord(pathfile=DIR_TFRECORD+"_val.tfrecord", dataset=df_val, siz
    createTFRecord(pathfile=DIR_TFRECORD+"_test.tfrecord", dataset=df_test, s

train_data = tf.data.TFRecordDataset(DIR_TFRECORD+"_train.tfrecord")
train_data = train_data.map(parse, num_parallel_calls=autotune)

val_data = tf.data.TFRecordDataset(DIR_TFRECORD+"_val.tfrecord")
val_data = val_data.map(parse, num_parallel_calls=autotune)

test_data = tf.data.TFRecordDataset(DIR_TFRECORD+"_test.tfrecord")
test_data = test_data.map(parse, num_parallel_calls=autotune)
```

Let's take a preview

In [ ]:
```
plt.subplots(4, 2, figsize=(5, 10))
for i, data in enumerate(train_data.take(4)):
    plt.subplot(4, 2, (i*2)+1)
    plt.imshow(data[0])
    plt.axis("off")
    plt.title(f"Original image\nClass: {classes[data[2][0]]}")
    plt.subplot(4, 2, (i*2)+2)
    plt.imshow(data[1], cmap="bone")
    plt.axis("off")
    plt.title(f"Mask\nClass: {classes[data[2][0]]}")
plt.tight_layout()
```


Original image
Class: [Malignant] Pro-B
Mask
Class: [Malignant] Pro-B
Original image
Class: [Malignant] Pro-B
Mask
Class: [Malignant] Pro-B
Original image
Class: [Malignant] Pre-B
Mask
Class: [Malignant] Pre-B
Original image
Class: [Malignant] Pre-B
Mask
Class: [Malignant] Pre-B

Loading the images for image classification

In [ ]:
```
batch_size = 32

def classification_task(image, mask, class_):
    class_ohe = tf.one_hot(indices=tf.squeeze(class_, axis=0), depth=4)
    return tf.cast(image, dtype=tf.float32)/255, tf.cast(class_ohe, dtype=tf.

train_data_clf = train_data.map(classification_task, num_parallel_calls=autot
train_data_clf = train_data_clf.batch(batch_size)
train_data_clf = train_data_clf.prefetch(autotune)

val_data_clf = val_data.map(classification_task, num_parallel_calls=autotune)
val_data_clf = val_data_clf.batch(batch_size)
val_data_clf = val_data_clf.prefetch(autotune)

test_data_clf = test_data.map(classification_task, num_parallel_calls=autotun
test_data_clf = test_data_clf.batch(batch_size)
test_data_clf = test_data_clf.prefetch(autotune)
```

Loading the images for image segmentation

In [ ]:
```
def segmentation_task(image, mask, class_):
    return tf.cast(image, dtype=tf.float32)/255, tf.cast(mask, dtype=tf.float3

train_data_sg = train_data.map(segmentation_task, num_parallel_calls=autotun
train_data_sg = train_data_sg.batch(batch_size)
train_data_sg = train_data_sg.prefetch(autotune)

val_data_sg = val_data.map(segmentation_task, num_parallel_calls=autotune)
val_data_sg = val_data_sg.batch(batch_size)
val_data_sg = val_data_sg.prefetch(autotune)

test_data_sg = test_data.map(segmentation_task, num_parallel_calls=autotune)
test_data_sg = test_data_sg.batch(batch_size)
test_data_sg = test_data_sg.prefetch(autotune)
```

Loading the images for both tasks

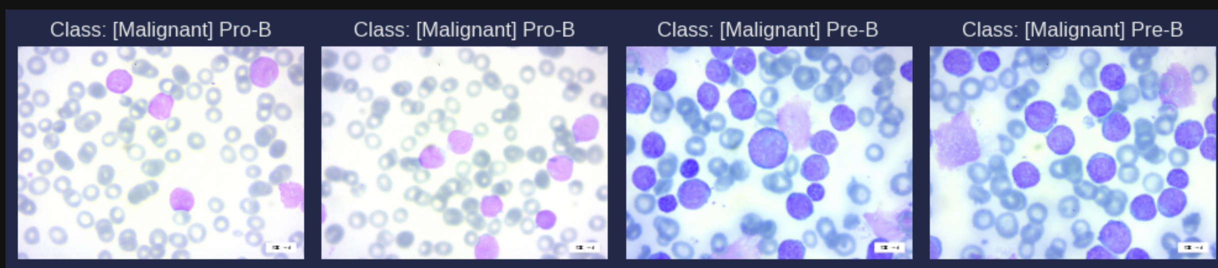
In [ ]:
```
def both_task(image, mask, class_):
    class_ohe = tf.one_hot(indices=tf.squeeze(class_, axis=0), depth=4)
    return tf.cast(image, dtype=tf.float32)/255, {"segmentation": tf.cast(mas

train_data_both = train_data.map(both_task, num_parallel_calls=autotune)
train_data_both = train_data_both.batch(batch_size)
train_data_both = train_data_both.prefetch(autotune)

val_data_both = val_data.map(both_task, num_parallel_calls=autotune)
val_data_both = val_data_both.batch(batch_size)
val_data_both = val_data_both.prefetch(autotune)
```

```
test_data_both = test_data.map(both_task, num_parallel_calls=autotune)
test_data_both = test_data_both.batch(batch_size)
test_data_both = test_data_both.prefetch(autotune)
```

Training image visualization

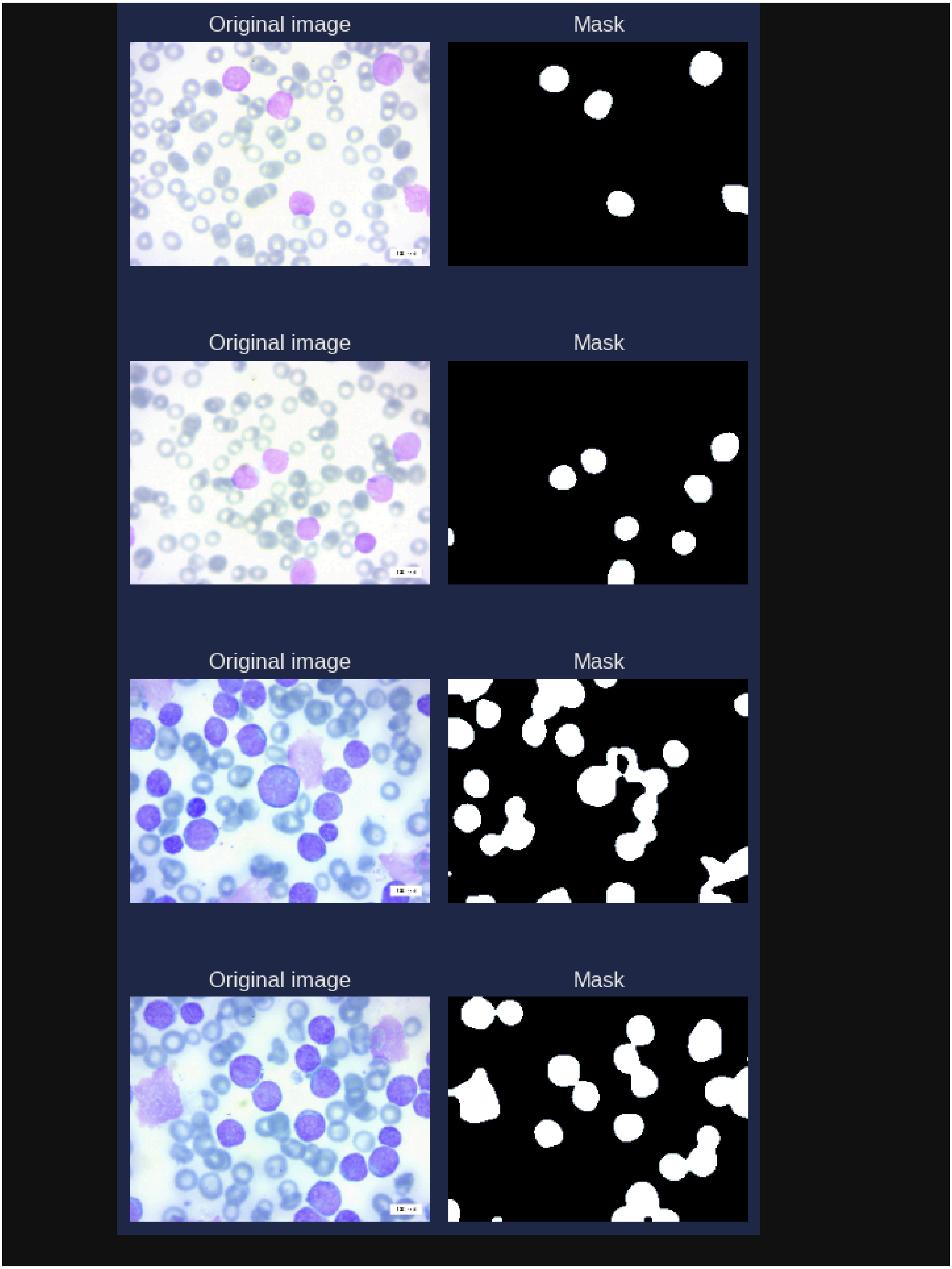
For classification

In [ ]:
```
plt.subplots(1, 4, figsize=(10, 4))
for i, (image, class_ohe) in enumerate(train_data_clf.take(1).unbatch()):
    if i == 4:
        break
    plt.subplot(1, 4, i+1)
    plt.imshow(tf.cast(image*255, tf.uint8))
    plt.title(f"Class: {classes[np.argmax(class_ohe)]}")
    plt.axis("off")
plt.tight_layout()
```



For segmentation

In [ ]:
```
plt.subplots(4, 2, figsize=(5, 10))
for i, (image, mask) in enumerate(train_data_sg.take(1).unbatch()):
    if i == 4:
        break
    plt.subplot(4, 2, (i*2)+1)
    plt.imshow(image)
    plt.axis("off")
    plt.title(f"Original image")
    plt.subplot(4, 2, (i*2)+2)
    plt.imshow(mask, cmap="bone")
    plt.axis("off")
    plt.title(f"Mask")
plt.tight_layout()
```


Original image
Mask
Original image
Mask
Original image
Mask
Original image
Mask

Handling imbalance image dataset

Calculating the class weights for the classes

In [ ]:
```
df_train.type_cell.value_counts()
```

Out[ ]:
```
type_cell
[Malignant] early Pre-B    685
[Malignant] Pre-B          668
[Malignant] Pro-B          557
Benign                     358
Name: count, dtype: int64
```

In [ ]:
```
import seaborn as sns
plt.figure(figsize=(15, 2))
sns.countplot(df_train, y="type_cell", alpha=1, order=df_train.type_cell.val
plt.ylabel("Type cell");
```

In [ ]:
```
class_weight = dict(zip([i for i, _ in enumerate(classes)], compute_class_we
class_weight
```

Out[ ]:
```
{0: 0.8277372262773722,
 1: 1.0179533213644525,
 2: 0.8488023952095808,
 3: 1.5837988826815643}
```

Calculating the class weights for the masks

In [ ]:
```
pixels_one = 0
pixels_zero = 0

for image, mask, class_ in train_data:
    aux = tf.reduce_sum(mask/255)
    pixels_one += aux
    pixels_zero += (mask.shape[0] * mask.shape[1]) - aux

total_pixels = pixels_one + pixels_zero
print(pixels_one, pixels_zero, total_pixels)
```

```
tf.Tensor(12867587.0, shape=(), dtype=float32) tf.Tensor(98609190.0, shape=(),
dtype=float32) tf.Tensor(111476776.0, shape=(), dtype=float32)
```

```
In [ ]: class_weight_pixel = {0:(total_pixels/(2*pixels_zero)).numpy(), 1:(total_pix
        class_weight_pixel
```

```
Out[ ]: {0: 0.56524533, 1: 4.3316894}
```

Diagnosing Acute Lymphoblastic Leukemia (ALL) by classifying and segmenting images of blood cells affected by cancer

Image classification

Creating a baseline model for image classification

```
In [ ]: def conv2d(filters):
            return Conv2D(filters, kernel_size=(2, 2), padding='same', strides=(1, 1)

        def block_conv(inp, filters=128, dropout=0.4):
            x = conv2d(filters)(inp)
            x = ReLU()(x)
            x = BatchNormalization()(x)

            x = conv2d(filters//2)(x)
            x = ReLU()(x)
            x = BatchNormalization()(x)

            x = MaxPooling2D(pool_size=(2, 2))(x)
            return Dropout(dropout)(x)

        def classification_base_architecture(inp, name_output=None):
            x = block_conv(inp, filters=128, dropout=0.4)
            x = block_conv(x, filters=128, dropout=0.4)

            x = GlobalAveragePooling2D()(x)
            x = Flatten()(x)
            x = Dense(32, activation="relu")(x)
            x = Dense(16, activation="relu")(x)
            x = Dense(len(classes), activation="softmax", name=name_output)(x)
            return x

        def get_model():
            tf.keras.backend.clear_session()

            inp = Input(shape=img_shape+(3,))
            x = classification_base_architecture(inp)
            model = Model(inputs=inp, outputs=x)

            model.compile(loss=CategoricalCrossentropy(), optimizer=Adam(learning_ra
```

```python
    return model

model_base = get_model()
```

In [ ]:
```python
model_base.summary()
```

Model: "functional"

Layer (type)	Output Shape
input_layer (InputLayer)	(None, 192, 256, 3)
conv2d (Conv2D)	(None, 192, 256, 128)
re_lu (ReLU)	(None, 192, 256, 128)
batch_normalization (BatchNormalization)	(None, 192, 256, 128)
conv2d_1 (Conv2D)	(None, 192, 256, 64)
re_lu_1 (ReLU)	(None, 192, 256, 64)
batch_normalization_1 (BatchNormalization)	(None, 192, 256, 64)
max_pooling2d (MaxPooling2D)	(None, 96, 128, 64)
dropout (Dropout)	(None, 96, 128, 64)
conv2d_2 (Conv2D)	(None, 96, 128, 128)
re_lu_2 (ReLU)	(None, 96, 128, 128)
batch_normalization_2 (BatchNormalization)	(None, 96, 128, 128)
conv2d_3 (Conv2D)	(None, 96, 128, 64)
re_lu_3 (ReLU)	(None, 96, 128, 64)
batch_normalization_3 (BatchNormalization)	(None, 96, 128, 64)
max_pooling2d_1 (MaxPooling2D)	(None, 48, 64, 64)
dropout_1 (Dropout)	(None, 48, 64, 64)
global_average_pooling2d (GlobalAveragePooling2D)	(None, 64)
flatten (Flatten)	(None, 64)
dense (Dense)	(None, 32)
dense_1 (Dense)	(None, 16)
dense_2 (Dense)	(None, 4)

Total params: 104,436 (407.95 KB)

Trainable params: 103,668 (404.95 KB)

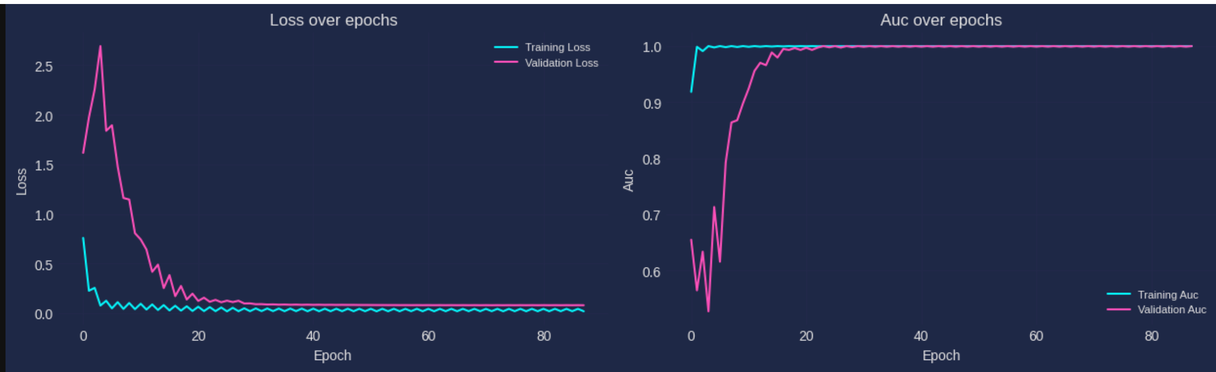
Non-trainable params: 768 (3.00 KB)

Let's train

```
In [ ]: def train_model(model, train_data, val_data, epochs=100, version="base"):
            checkpoint_filepath = f'/kaggle/working/models/model_{version}.keras'
            checkpoint = ModelCheckpoint(checkpoint_filepath, monitor='val_loss', sav
            early = EarlyStopping(monitor='val_loss', patience=8, verbose=1, restore
            reduce = ReduceLROnPlateau(monitor='val_loss', patience=3)
            csvlogger = CSVLogger(f"/kaggle/working/histories/history_model_{version

            history = model.fit(train_data,
                                  batch_size=batch_size,
                                  epochs=epochs,
                                  callbacks=[early,
                                             reduce,
                                             checkpoint,
                                             csvlogger],
                                  validation_data=val_data,
                                  class_weight=class_weight,
                                  steps_per_epoch=len(df_train)//batch_size,
                                  validation_steps=len(df_val)//batch_size,
                                  validation_batch_size=batch_size
                                 )
            return history
```

```
In [ ]: history_base = train_model(model_base, train_data_clf, val_data_clf, epochs=
```

```
In [ ]: def delete_empty_subplots(fig, axs):
            for ax_row in axs:
                for ax in ax_row:
                    if ax.title.get_text() == "":
                        fig.delaxes(ax)
            return fig, axs

        def show_history(version, parameters=None):
            df_hist = pd.read_csv(f"/kaggle/working/histories/history_model_{version
            fig, axs = plt.subplots(2, 4, figsize=(25, 7))

            for i, col in enumerate(parameters):
                plt.subplot(2, 4, i+1)
                param = col[0].replace('_', ' ').title()
                plt.plot(df_hist.index, df_hist[col[0]], label=f"Training {param}")
                plt.plot(df_hist.index, df_hist[col[1]], label=f"Validation {param}"
                plt.ylabel(col[0].capitalize().replace("_", " "))
                plt.xlabel("Epoch")
                plt.legend(fontsize=8)
                plt.grid(True, alpha=0.3, which="both")
                plt.title(f"{param} over epochs")

            fig, axs = delete_empty_subplots(fig, axs)
            plt.tight_layout()
            plt.show()
            return

        show_history(version="base", parameters=[["loss", "val_loss"],["auc", "val_a
```



Evaluating the baseline model for image classification

In [ ]:
```
model_base.evaluate(test_data_clf)
```

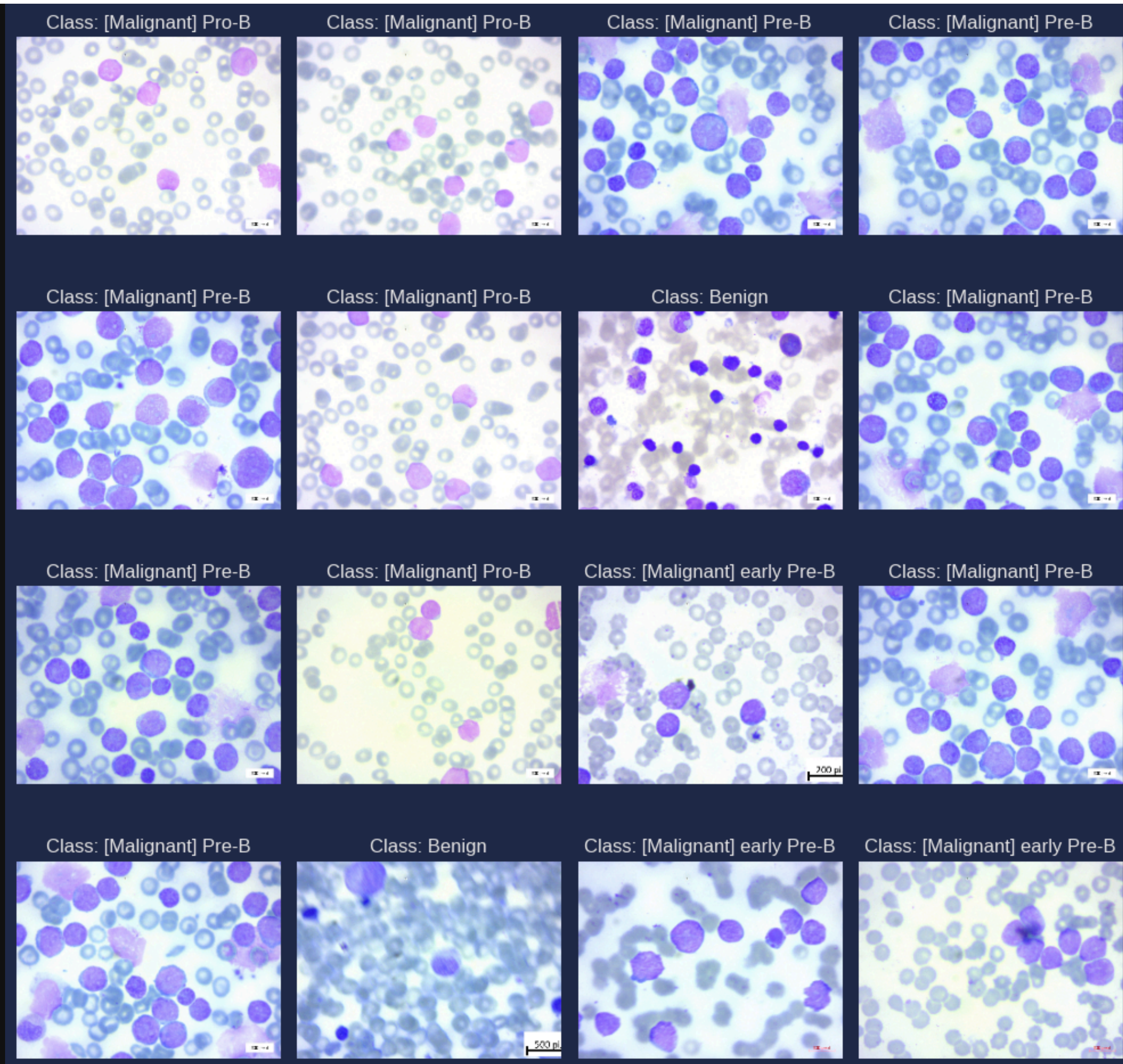
```
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 35ms/step - auc: 0.9999 - loss: 0.0631
```

```
/usr/lib/python3.10/contextlib.py:153: UserWarning: Your input ran out of data; interrupting training. Make sure that your dataset or generator can generate at least `steps_per_epoch * epochs` batches. You may need to use the `.repeat()` function when building your dataset.
  self.gen.throw(typ, value, traceback)
```

Out[ ]:
```
[0.06927863508462906, 0.9997674822807312]
```

Applying data augmentation technique on images

In [ ]:
```
def data_augmentation(image, mask, class_):
    img = random_flip_left_right(image, seed=seed_value)
    img = random_flip_up_down(img, seed=seed_value)
    img = random_brightness(img, max_delta=0.01, seed=seed_value)
    img = random_contrast(img, lower=0.5, upper=1, seed=seed_value)
    img = random_saturation(img, lower=0.5, upper=1, seed=seed_value)
    class_ohe = tf.one_hot(indices=tf.squeeze(class_, axis=0), depth=4)
    return tf.cast(image, dtype=tf.float32)/255, tf.cast(class_ohe, dtype=tf.

train_data_clf_aug = train_data.map(data_augmentation, num_parallel_calls=au
train_data_clf_aug = train_data_clf_aug.batch(batch_size)
train_data_clf_aug = train_data_clf_aug.prefetch(autotune)

plt.subplots(4, 4, figsize=(10, 10))
for i, (image, class_ohe) in enumerate(train_data_clf_aug.take(1).unbatch())
    if i == 16:
        break
    plt.subplot(4, 4, i+1)
    plt.imshow(tf.cast(image*255, tf.uint8))
    plt.title(f"Class: {classes[np.argmax(class_ohe)]}")
    plt.axis("off")
plt.tight_layout()
```



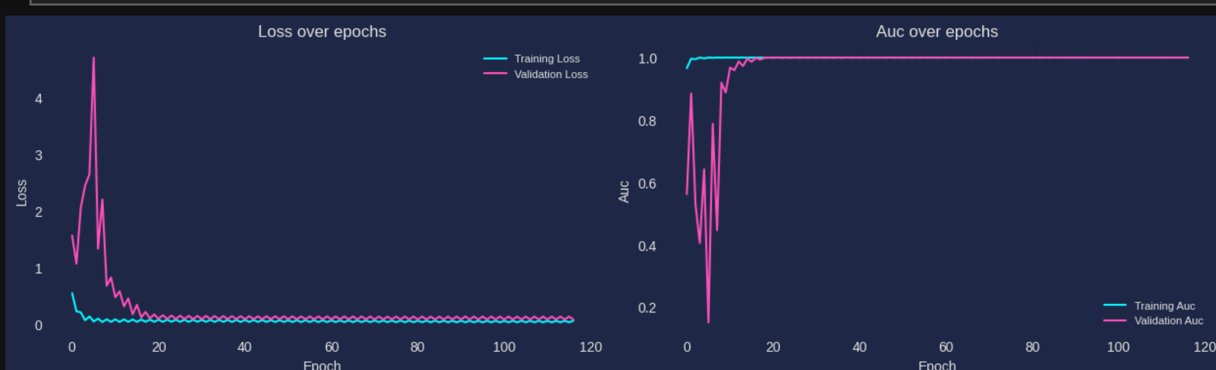
Training a baseline model with data augmentation

In [ ]:

```
model_base_aug = get_model()
history_base_aug = train_model(model_base_aug, train_data_clf_aug, val_data_
```

In [ ]:

```
show_history(version="base_aug", parameters=[["loss", "val_loss"],["auc", "v
```



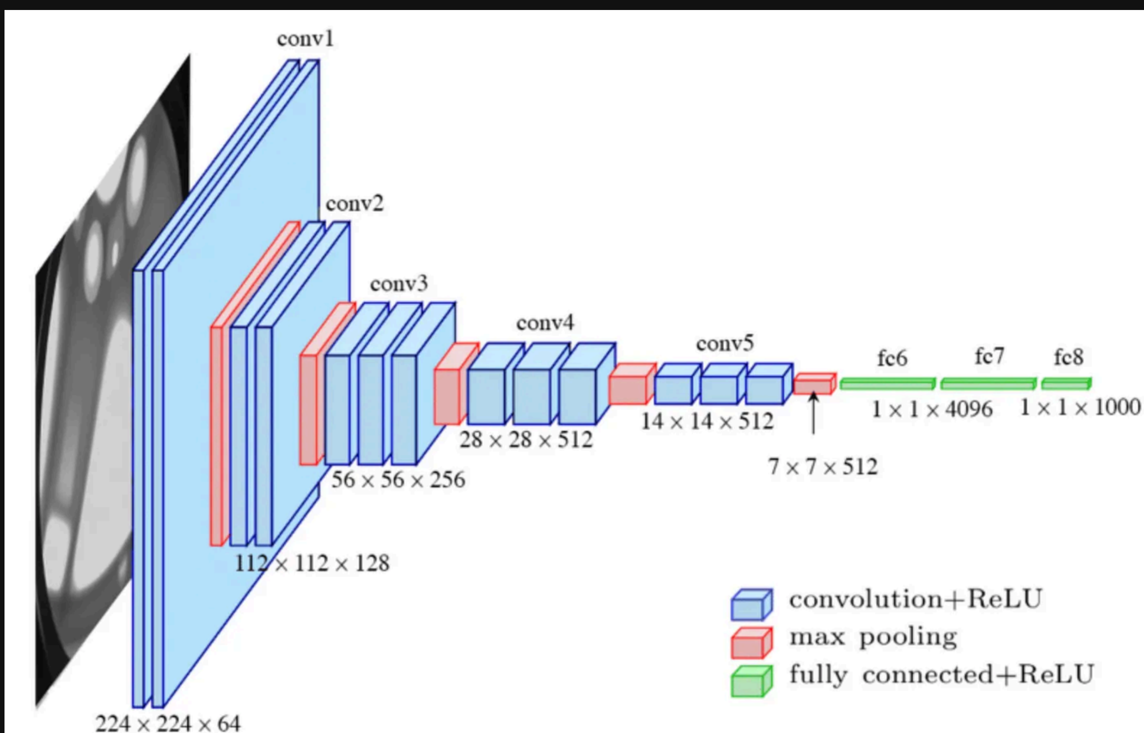
Evaluating the baseline model with data augmentation

In [ ]:
```
model_base_aug.evaluate(test_data_clf)
```

```
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 34ms/step - auc: 0.9995 - loss: 0.0655
/usr/lib/python3.10/contextlib.py:153: UserWarning: Your input ran out of data; interrupting training. Make sure that your dataset or generator can generate at least `steps_per_epoch * epochs` batches. You may need to use the `.repeat()` function when building your dataset.
  self.gen.throw(typ, value, traceback)
```

Out[ ]:
```
[0.07973799854516983, 0.9987912774085999]
```

Transfer learning using the architecture proposed by the Visual Geometry Group (VGG)



In [ ]:
```
def get_backbone():
    backbone = VGG16(include_top=False, weights='imagenet', input_shape=img_
    backbone.trainable = False
    return backbone

def classification_w_backbone_architecture(inp, name_output=None):
    backbone = get_backbone()
    x = backbone(inp, training=False)
    x = Flatten()(x)
    x = Dense(32, activation="relu")(x)
    x = Dense(16, activation="relu")(x)
    x = Dense(len(classes), activation="softmax", name=name_output)(x)
```

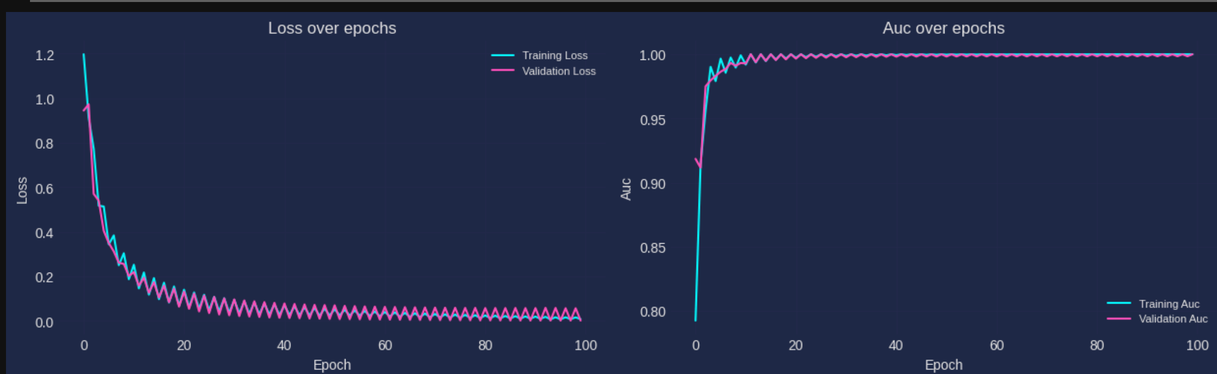
```python
    return x

def get_model_w_backbone():
    tf.keras.backend.clear_session()

    inp = Input(shape=img_shape+(3,))
    x = classification_w_backbone_architecture(inp)
    model = Model(inputs=inp, outputs=x)

    model.compile(loss=CategoricalCrossentropy(), optimizer=Adam(learning_ra
    return model

model_backbone = get_model_w_backbone()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applicat
ions/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5
58889256/58889256 ━━━━━━━━━━━━━━━━━━━━ 2s 0us/step
```

Let's train

In [ ]:
```python
history_backbone = train_model(model_backbone, train_data_clf_aug, val_data_
```

In [ ]:
```python
show_history(version="backbone", parameters=[["loss", "val_loss"],["auc", "v
```



Evaluating the model with a VGG backbone

In [ ]:
```python
model_backbone.evaluate(test_data_clf)
```

```
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 58ms/step - auc: 0.9996 - loss:
0.0469
```

```
/usr/lib/python3.10/contextlib.py:153: UserWarning: Your input ran out of dat
a; interrupting training. Make sure that your dataset or generator can genera
te at least `steps_per_epoch * epochs` batches. You may need to use the `.rep
eat()` function when building your dataset.
  self.gen.throw(typ, value, traceback)
```

Out[ ]:
```
[0.051316503435373306, 0.9995214939117432]
```

Evaluation metrics for classification task

In [ ]:
```python
def get_scores(y_true, y_pred, y_pred_proba):
    return {'AUC_ROC':roc_auc_score(y_true, y_pred_proba, multi_class='ovr')
            'F1_Score':f1_score(y_true, y_pred, average='weighted'),
            'Accuracy':accuracy_score(y_true, y_pred),
            'Precision':precision_score(y_true, y_pred, average='weighted'),
            'Recall':recall_score(y_true, y_pred, average='weighted')}

for i, (img, class_) in enumerate(test_data_clf):
    if i == 0:
        y_true = class_.numpy()
    else:
        y_true = np.concatenate((y_true, class_.numpy()), axis=0)

scores = []
for model in [model_base, model_base_aug, model_backbone]:
    y_test_pred_proba = model.predict(test_data_clf, verbose=0)
    y_test_pred = tf.math.round(y_test_pred_proba)
    scores.append(get_scores(y_true, y_test_pred, y_test_pred_proba))

comparison = pd.DataFrame(data=scores, index=["Baseline Model", "Baseline Mo
comparison.style.highlight_max(color = 'green', axis = 0).highlight_min(colo
```

```
/usr/lib/python3.10/contextlib.py:153: UserWarning: Your input ran out of dat
a; interrupting training. Make sure that your dataset or generator can genera
te at least `steps_per_epoch * epochs` batches. You may need to use the `.rep
eat()` function when building your dataset.
  self.gen.throw(typ, value, traceback)
/usr/lib/python3.10/contextlib.py:153: UserWarning: Your input ran out of dat
a; interrupting training. Make sure that your dataset or generator can genera
te at least `steps_per_epoch * epochs` batches. You may need to use the `.rep
eat()` function when building your dataset.
  self.gen.throw(typ, value, traceback)
```

Out[ ]:

	AUC_ROC	F1_Score	Accuracy	Precision	Recall
Baseline Model	0.999683	0.985588	0.983573	0.987708	0.983573
Baseline Model with DA	0.999239	0.983538	0.981520	0.985767	0.981520
Model with VGG16	0.999667	0.979537	0.979466	0.980103	0.979466

Image segmentation

Creating a baseline model for image segmentation

In [ ]:
```python
def encoder(inp, filters):
    x1 = block_conv(inp, filters=filters, dropout=0.4)
    x2 = block_conv(x1, filters=filters, dropout=0.4)
    return x1, x2

def bottleneck(inp, filters):
```

```
    x = conv2d(filters)(inp)
    x = ReLU()(x)
    x = BatchNormalization()(x)

    x = conv2d(filters)(inp)
    x = ReLU()(x)
    return BatchNormalization()(x)

def conv2dtranspose(filters):
        return Conv2DTranspose(filters=filters, activation="relu", kernel_size

def up_block(inp, output_enc, filters, dropout=0.4):
    x = conv2d(filters)(inp)
    x = Concatenate()([inp, output_enc])
    x = ReLU()(x)
    x = BatchNormalization()(x)

    x = conv2d(filters)(inp)
    x = ReLU()(x)
    x = BatchNormalization()(x)

    x = conv2dtranspose(filters)(x)
    return Dropout(dropout)(x)

def decoder(inp, ouput_block1_enc, ouput_block2_enc, filters):
    x1 = up_block(inp, ouput_block2_enc, filters, dropout=0.4)
    x2 = up_block(x1, ouput_block1_enc, filters, dropout=0.4)
    return x2

def segmentation_architecture(inp, name_output=None):
    ouput_block1_enc, ouput_block2_enc = encoder(inp, filters=128)
    x = bottleneck(ouput_block2_enc, filters=128)
    x = decoder(x, ouput_block1_enc, ouput_block2_enc, filters=128)
    x = Conv2D(filters=1, activation="sigmoid", kernel_size=(2, 2), padding='
    return x

def get_model_segmentation():
    tf.keras.backend.clear_session()

    inp = Input(shape=img_shape+(3,))
    x = segmentation_architecture(inp)
    model = Model(inputs=inp, outputs=x)

    model.compile(loss=Dice(),
                  optimizer=Adam(learning_rate=1e-3),
                  metrics=[BinaryIoU(), BinaryAccuracy()])
    return model

model_base_sg = get_model_segmentation()
```

In [ ]:

```
model_base_sg.summary()
```

Model: "functional"

Layer (type)	Output Shape
input_layer (InputLayer)	(None, 192, 256, 3)
conv2d (Conv2D)	(None, 192, 256, 128)
re_lu (ReLU)	(None, 192, 256, 128)
batch_normalization (BatchNormalization)	(None, 192, 256, 128)
conv2d_1 (Conv2D)	(None, 192, 256, 64)
re_lu_1 (ReLU)	(None, 192, 256, 64)
batch_normalization_1 (BatchNormalization)	(None, 192, 256, 64)
max_pooling2d (MaxPooling2D)	(None, 96, 128, 64)
dropout (Dropout)	(None, 96, 128, 64)
conv2d_2 (Conv2D)	(None, 96, 128, 128)
re_lu_2 (ReLU)	(None, 96, 128, 128)
batch_normalization_2 (BatchNormalization)	(None, 96, 128, 128)
conv2d_3 (Conv2D)	(None, 96, 128, 64)
re_lu_3 (ReLU)	(None, 96, 128, 64)
batch_normalization_3 (BatchNormalization)	(None, 96, 128, 64)
max_pooling2d_1 (MaxPooling2D)	(None, 48, 64, 64)
dropout_1 (Dropout)	(None, 48, 64, 64)
conv2d_5 (Conv2D)	(None, 48, 64, 128)
re_lu_5 (ReLU)	(None, 48, 64, 128)
batch_normalization_5 (BatchNormalization)	(None, 48, 64, 128)
conv2d_7 (Conv2D)	(None, 48, 64, 128)
re_lu_7 (ReLU)	(None, 48, 64, 128)
batch_normalization_7 (BatchNormalization)	(None, 48, 64, 128)
conv2d_transpose (Conv2DTranspose)	(None, 96, 128, 128)
dropout_2 (Dropout)	(None, 96, 128, 128)

conv2d_9 (Conv2D)	(None, 96, 128, 128)
re_lu_9 (ReLU)	(None, 96, 128, 128)
batch_normalization_9 (BatchNormalization)	(None, 96, 128, 128)
conv2d_transpose_1 (Conv2DTranspose)	(None, 192, 256, 128)
dropout_3 (Dropout)	(None, 192, 256, 128)
conv2d_10 (Conv2D)	(None, 192, 256, 1)

Total params: 399,361 (1.52 MB)

Trainable params: 397,825 (1.52 MB)

Non-trainable params: 1,536 (6.00 KB)

In [ ]:
```
def show_predictions(dataset=None):
    for image, _ in dataset.shuffle(100).take(1):
        pred = model_base_sg.predict(image)
        pred_mask = pred[0] >= 0.5
        plt.subplots(1, 2, figsize=(6, 3))

        plt.subplot(1, 2, 1)
        plt.imshow(image[0])
        plt.axis("off")
        plt.title(f"Original image")

        plt.subplot(1, 2, 2)
        plt.imshow(pred_mask, cmap="bone")
        plt.axis("off")
        plt.title(f"Mask")

        plt.tight_layout()
        plt.show()
        break
    return

class DisplayCallback(Callback):
    def on_epoch_end(self, epoch, logs=None):
        clear_output(wait=True)
        show_predictions(dataset=val_data_sg)
        print ('\nSample Prediction after epoch {}\n'.format(epoch+1))
        return
```

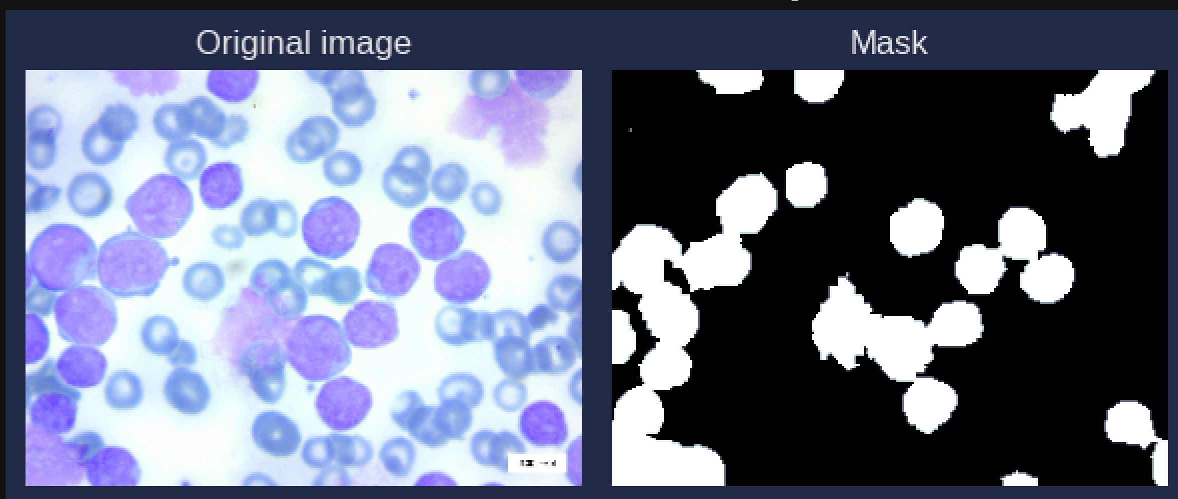
Let's train

In [ ]:
```
def train_model_segmentation(model, train_data, val_data, epochs=100, versio
    checkpoint_filepath = f'/kaggle/working/models/model_{version}.keras'
    checkpoint = ModelCheckpoint(checkpoint_filepath, monitor='val_loss', sav
    early = EarlyStopping(monitor='val_loss', patience=8, verbose=1, restore
    reduce = ReduceLROnPlateau(monitor='val_loss', patience=3)
    csvlogger = CSVLogger(f"/kaggle/working/histories/history_model_{version
```

```python
    history = model.fit(train_data,
                        batch_size=batch_size,
                        epochs=epochs,
                        callbacks=[early,
                                   reduce,
                                   checkpoint,
                                   csvlogger,
                                   DisplayCallback()],
                        validation_data=val_data,
                        class_weight=class_weight_pixel,
                        steps_per_epoch=len(df_train)//batch_size,
                        validation_steps=len(df_val)//batch_size,
                        validation_batch_size=batch_size
                       )
    return history
```

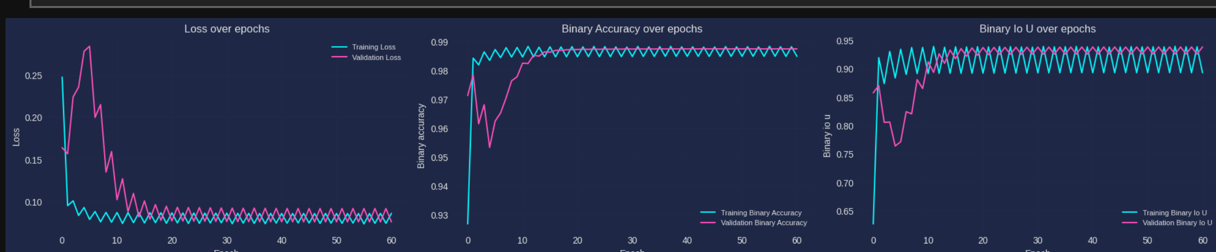
In [ ]:
```python
history_base_sg = train_model_segmentation(model_base_sg, train_data_sg, val
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step
```




```
Sample Prediction after epoch 61

70/70 ━━━━━━━━━━━━━━━━━━━━ 19s 269ms/step - binary_accuracy: 0.
9849 - binary_io_u: 0.9139 - loss: 0.0848 - val_binary_accuracy: 0.9873 - val
_binary_io_u: 0.9375 - val_loss: 0.0755 - learning_rate: 1.0000e-13
Epoch 61: early stopping
Restoring model weights from the end of the best epoch: 53.
```

In [ ]:
```python
show_history(version="base_segmentation", parameters=[["loss", "val_loss"],
                                                      ["binary_accuracy", "v
                                                      ["binary_io_u", "val_b
```



Evaluating the baseline model for segmentation

In [ ]:
```
result_test_sg = model_base_sg.evaluate(test_data_sg)
result_test_sg
```

```
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 83ms/step - binary_accuracy: 0.98
51 - binary_io_u: 0.9295 - loss: 0.0845
```

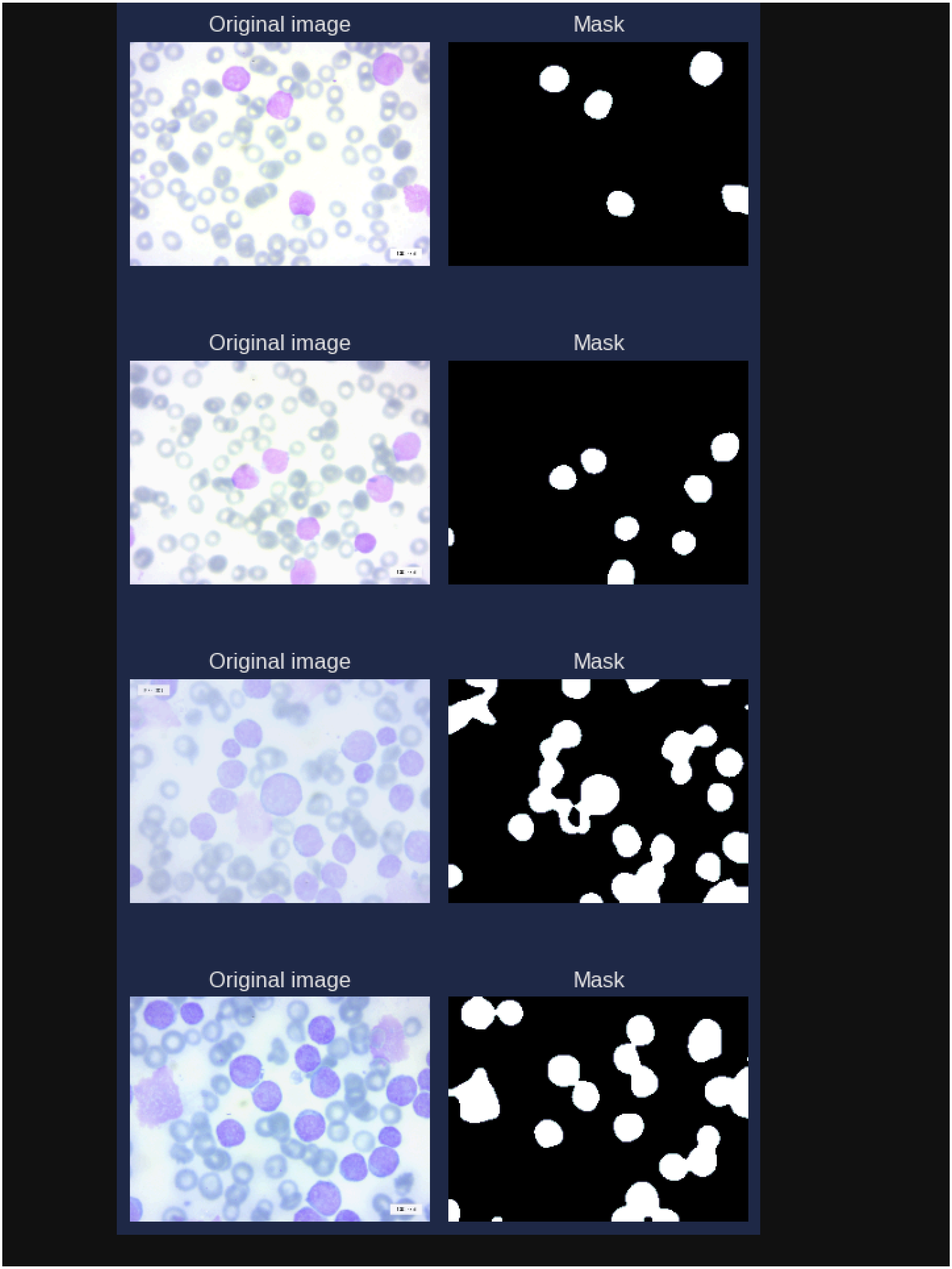
```
/usr/lib/python3.10/contextlib.py:153: UserWarning: Your input ran out of dat
a; interrupting training. Make sure that your dataset or generator can genera
te at least `steps_per_epoch * epochs` batches. You may need to use the `.rep
eat()` function when building your dataset.
  self.gen.throw(typ, value, traceback)
```

Out[ ]:
```
[0.08604717999696732, 0.9265201091766357, 0.9849867820739746]
```

Applying data augmentation technique on images and masks

In [ ]:
```
def data_transformation(image, mask, class_):
    img = random_brightness(image, max_delta=0.01, seed=seed_value)
    img = random_contrast(img, lower=0.5, upper=1, seed=seed_value)
    img = random_saturation(img, lower=0.5, upper=1, seed=seed_value)
    aux = tf.concat([img, mask], -1)

    aux_ = random_flip_left_right(aux, seed=seed_value)
    aux_ = random_flip_up_down(aux_, seed=seed_value)

    return tf.cast(aux_[..., :3], tf.float32)/255, tf.cast(aux_[..., 3:], tf.

train_data_sg_aug = train_data.map(data_transformation, num_parallel_calls=a
train_data_sg_aug = train_data_sg_aug.batch(batch_size)
train_data_sg_aug = train_data_sg_aug.prefetch(autotune)
```

In [ ]:
```
plt.subplots(4, 2, figsize=(5, 10))
for i, (image, mask) in enumerate(train_data_sg_aug.take(1).unbatch()):
    if i == 4:
        break
    plt.subplot(4, 2, (i*2)+1)
    plt.imshow(image)
    plt.axis("off")
    plt.title(f"Original image")
    plt.subplot(4, 2, (i*2)+2)
    plt.imshow(mask, cmap="bone")
    plt.axis("off")
    plt.title(f"Mask")
plt.tight_layout()
```

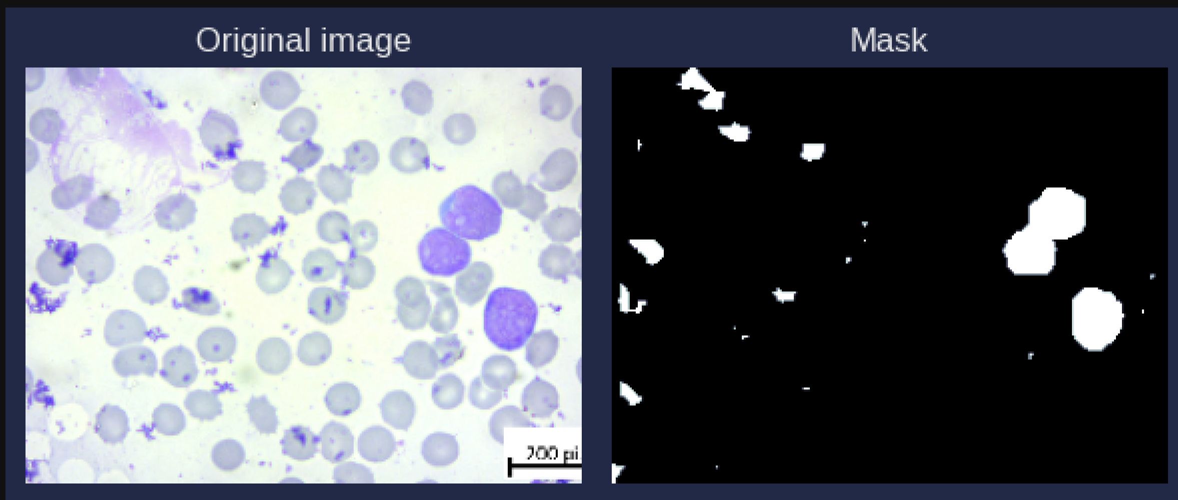


Original image
Mask
Original image
Mask
Original image
Mask
Original image
Mask

Training a baseline model with data augmentation

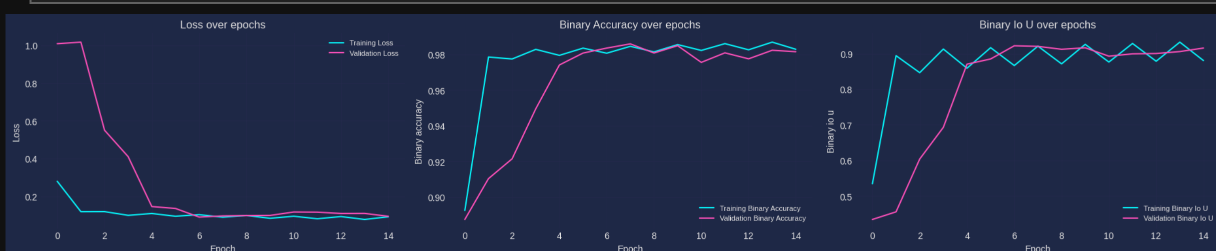
In [ ]:
```
model_base_sg_aug = get_model_segmentation()
history_base_sg_aug = train_model_segmentation(model_base_sg_aug, train_data
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 70ms/step
```




```
Sample Prediction after epoch 15

70/70 ━━━━━━━━━━━━━━━━━━━━ 19s 270ms/step - binary_accuracy: 0.
9830 - binary_io_u: 0.9041 - loss: 0.0910 - val_binary_accuracy: 0.9816 - val
_binary_io_u: 0.9156 - val_loss: 0.0941 - learning_rate: 1.0000e-05
Epoch 15: early stopping
Restoring model weights from the end of the best epoch: 7.
```

In [ ]:
```
show_history(version="base_segmentation_aug", parameters=[["loss", "val_loss
                                                           ["binary_accuracy", "v
                                                           ["binary_io_u", "val_b
```



Evaluating the baseline model for image segmentation with data augmentation

In [ ]:
```
result_test_sg_aug = model_base_sg_aug.evaluate(test_data_sg)
result_test_sg_aug
```

```
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 82ms/step - binary_accuracy: 0.98
14 - binary_io_u: 0.9150 - loss: 0.0982
```

```
/usr/lib/python3.10/contextlib.py:153: UserWarning: Your input ran out of dat
a; interrupting training. Make sure that your dataset or generator can genera
te at least `steps_per_epoch * epochs` batches. You may need to use the `.rep
eat()` function when building your dataset.
  self.gen.throw(typ, value, traceback)
```

Out[ ]: [0.09920085221529007, 0.9123088121414185, 0.9814990758895874]

Evaluation metrics for segmentation task

In [ ]:

```python
for i, (img, mask) in enumerate(test_data_sg):
    aux = tf.reshape(mask,[-1])
    if i == 0:
        y_true = aux.numpy()
    else:
        y_true = np.concatenate((y_true, aux.numpy()), axis=0)

scores = []
for model in [model_base_sg, model_base_sg_aug]:
    y_test_pred_proba = model.predict(test_data_sg, verbose=0)
    y_test_pred_proba = tf.reshape(y_test_pred_proba,[-1])
    y_test_pred = tf.math.round(y_test_pred_proba)
    result_test = model.evaluate(test_data_sg)
    score = get_scores(y_true, y_test_pred, y_test_pred_proba)
    score["BinaryIOU"] = result_test[1]
    score["BinaryAccuracy"] = result_test[2]
    scores.append(score)

comparison = pd.DataFrame(data=scores, index=["Baseline Model", "Baseline Mo
comparison.style.highlight_max(color = 'green', axis = 0).highlight_min(colo
```

```
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 82ms/step - binary_accuracy: 0.98
51 - binary_io_u: 0.9295 - loss: 0.0845
/usr/lib/python3.10/contextlib.py:153: UserWarning: Your input ran out of dat
a; interrupting training. Make sure that your dataset or generator can genera
te at least `steps_per_epoch * epochs` batches. You may need to use the `.rep
eat()` function when building your dataset.
  self.gen.throw(typ, value, traceback)
16/16 ━━━━━━━━━━━━━━━━━━━━ 1s 82ms/step - binary_accuracy: 0.98
14 - binary_io_u: 0.9150 - loss: 0.0982
```

Out[ ]:

	AUC_ROC	F1_Score	Accuracy	Precision	Recall	BinaryIOU	BinaryAccurac
Baseline Model	0.987874	0.984820	0.984987	0.984809	0.984987	0.926520	0.98498
Baseline Model with DA	0.989966	0.981490	0.981499	0.981481	0.981499	0.912309	0.98149

Image segmentation and classification

Creating a baseline model for image segmentation and classification

In [ ]:
```
def get_model_both():
    tf.keras.backend.clear_session()

    inp = Input(shape=img_shape+(3,))
    seg_output = segmentation_architecture(inp, name_output="segmentation")
    clf_output = classification_w_backbone_architecture(inp, name_output="cla

    model = Model(inputs=inp, outputs={"segmentation":seg_output,
                                       "classification":clf_output})

    model.compile(loss={"segmentation":Dice(),
                        "classification":CategoricalCrossentropy()},
                  optimizer=Adam(learning_rate=1e-3),
                  metrics={"segmentation":[BinaryIoU(), BinaryAccuracy()],
                           "classification":[AUC()]}
                 )
    return model

model_base_both = get_model_both()
```

In [ ]:
```
model_base_both.summary()
```

Model: "functional"

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 192, 256, 3)	0
conv2d (Conv2D)	(None, 192, 256, 128)	1,664
re_lu (ReLU)	(None, 192, 256, 128)	0
batch_normalization (BatchNormalization)	(None, 192, 256, 128)	512
conv2d_1 (Conv2D)	(None, 192, 256, 64)	32,832
re_lu_1 (ReLU)	(None, 192, 256, 64)	0
batch_normalization_1 (BatchNormalization)	(None, 192, 256, 64)	256
max_pooling2d (MaxPooling2D)	(None, 96, 128, 64)	0
dropout (Dropout)	(None, 96, 128, 64)	0
conv2d_2 (Conv2D)	(None, 96, 128, 128)	32,896
re_lu_2 (ReLU)	(None, 96, 128, 128)	0
batch_normalization_2 (BatchNormalization)	(None, 96, 128, 128)	512
conv2d_3 (Conv2D)	(None, 96, 128, 64)	32,832
re_lu_3 (ReLU)	(None, 96, 128, 64)	0
batch_normalization_3 (BatchNormalization)	(None, 96, 128, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 48, 64, 64)	0
dropout_1 (Dropout)	(None, 48, 64, 64)	0
conv2d_5 (Conv2D)	(None, 48, 64, 128)	32,896
re_lu_5 (ReLU)	(None, 48, 64, 128)	0
batch_normalization_5 (BatchNormalization)	(None, 48, 64, 128)	512
conv2d_7 (Conv2D)	(None, 48, 64, 128)	65,664
re_lu_7 (ReLU)	(None, 48, 64, 128)	0
batch_normalization_7 (BatchNormalization)	(None, 48, 64, 128)	512
conv2d_transpose (Conv2DTranspose)	(None, 96, 128, 128)	65,664

dropout_2 (Dropout)	(None, 96, 128, 128)	0
conv2d_9 (Conv2D)	(None, 96, 128, 128)	65,664
vgg16 (Functional)	(None, 512)	14,714,688
re_lu_9 (ReLU)	(None, 96, 128, 128)	0
flatten (Flatten)	(None, 512)	0
batch_normalization_9 (BatchNormalization)	(None, 96, 128, 128)	512
dense (Dense)	(None, 32)	16,416
conv2d_transpose_1 (Conv2DTranspose)	(None, 192, 256, 128)	65,664
dense_1 (Dense)	(None, 16)	528
dropout_3 (Dropout)	(None, 192, 256, 128)	0
classification (Dense)	(None, 4)	68
segmentation (Conv2D)	(None, 192, 256, 1)	513

Total params: 15,131,061 (57.72 MB)
Trainable params: 414,837 (1.58 MB)
Non-trainable params: 14,716,224 (56.14 MB)

Let's plot the model

In [ ]:
```
plot_model(model_base_both, expand_nested=True, dpi=50, show_shapes=True)
```

Function for display some predictions during the training

In [ ]:
```
def show_predictions(dataset=None):
    for image, _ in dataset.shuffle(100).take(1):
        pred = model_base_both.predict(image)
        pred_mask, pred_class = pred["segmentation"][0], pred["classification
        pred_mask = pred_mask >= 0.5

        plt.subplots(1, 2, figsize=(6, 3))

        plt.subplot(1, 2, 1)
        plt.imshow(image[0])
        plt.axis("off")
        plt.title(f"Original image\nClass: {classes[np.argmax(pred_class)]}"

        plt.subplot(1, 2, 2)
        plt.imshow(pred_mask, cmap="bone")
        plt.axis("off")
        plt.title(f"Mask\nClass: {classes[np.argmax(pred_class)]}")
```

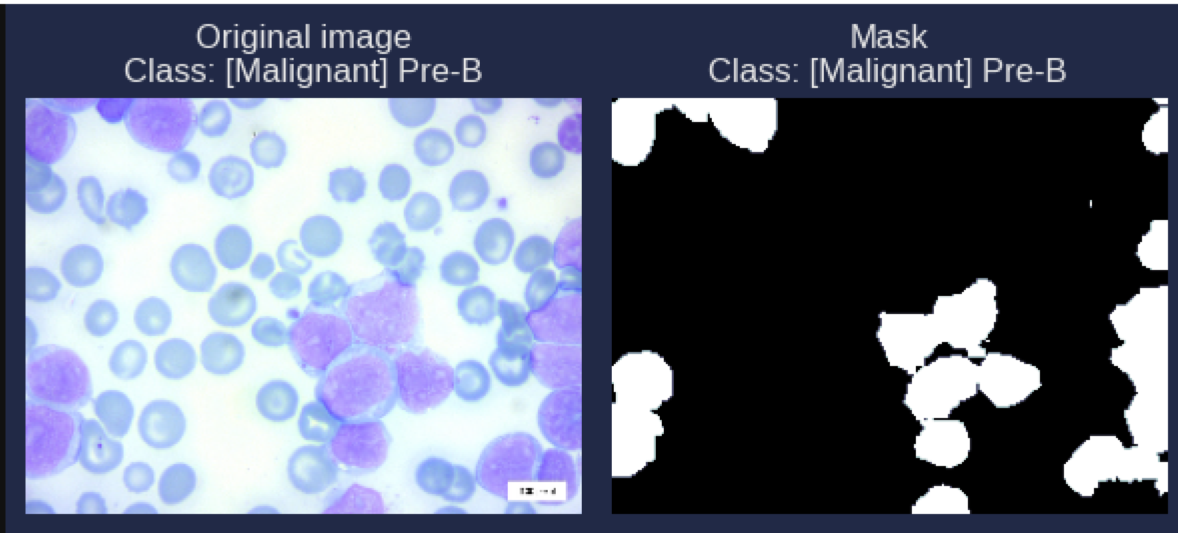
```python
        plt.tight_layout()
        plt.show()
        break
    return

class DisplayCallback(Callback):
    def on_epoch_end(self, epoch, logs=None):
        clear_output(wait=True)
        show_predictions(dataset=val_data_sg)
        print ('\nSample Prediction after epoch {}\n'.format(epoch+1))
        return
```

Let's train

In [ ]:
```python
def train_model_both(model, train_data, val_data, epochs=100, version="base"
    checkpoint_filepath = f'/kaggle/working/models/model_{version}.keras'
    checkpoint = ModelCheckpoint(checkpoint_filepath, monitor='val_loss', sav
    early = EarlyStopping(monitor='val_loss', patience=8, verbose=1, restore
    reduce = ReduceLROnPlateau(monitor='val_loss', patience=3)
    csvlogger = CSVLogger(f"/kaggle/working/histories/history_model_{version

    history = model.fit(train_data,
                        batch_size=batch_size,
                        epochs=epochs,
                        callbacks=[early,
                                   reduce,
                                   checkpoint,
                                   csvlogger,
                                   DisplayCallback()],
                        validation_data=val_data,
                        steps_per_epoch=len(df_train)//batch_size,
                        validation_steps=len(df_val)//batch_size,
                        validation_batch_size=batch_size
                       )
    return history
```

In [ ]:
```python
history_base_both = train_model_both(model_base_both, train_data_both, val_d
```

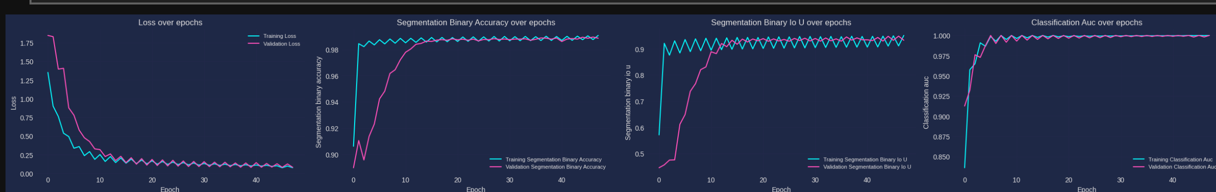
```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step
```




```
Sample Prediction after epoch 48

70/70 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - classification_auc: 1.
0000 - loss: 0.0758 - segmentation_binary_accuracy: 0.9907 - segmentation_bin
ary_io_u: 0.9509 - val_classification_auc: 1.0000 - val_loss: 0.0811 - val_seg
mentation_binary_accuracy: 0.9886 - val_segmentation_binary_io_u: 0.9326 - le
arning_rate: 1.0000e-05
Epoch 48: early stopping
Restoring model weights from the end of the best epoch: 40.
```

In [ ]:

```python
show_history(version="seg_clf", parameters=[["loss", "val_loss"],
                                            ["segmentation_binary_accuracy",
                                            ["segmentation_binary_io_u", "va
                                            ["classification_auc", "val_class
```



Evaluating the baseline model for image segmentation and classification

In [ ]:

```python
model_base_both.evaluate(test_data_both, return_dict=True)
```

```
16/16 ━━━━━━━━━━━━━━━━━━━━ 2s 137ms/step - classification_auc:
0.9993 - loss: 0.1381 - segmentation_binary_accuracy: 0.9867 - segmentation_b
inary_io_u: 0.9380
/usr/lib/python3.10/contextlib.py:153: UserWarning: Your input ran out of dat
a; interrupting training. Make sure that your dataset or generator can genera
te at least `steps_per_epoch * epochs` batches. You may need to use the `.rep
eat()` function when building your dataset.
  self.gen.throw(typ, value, traceback)
```

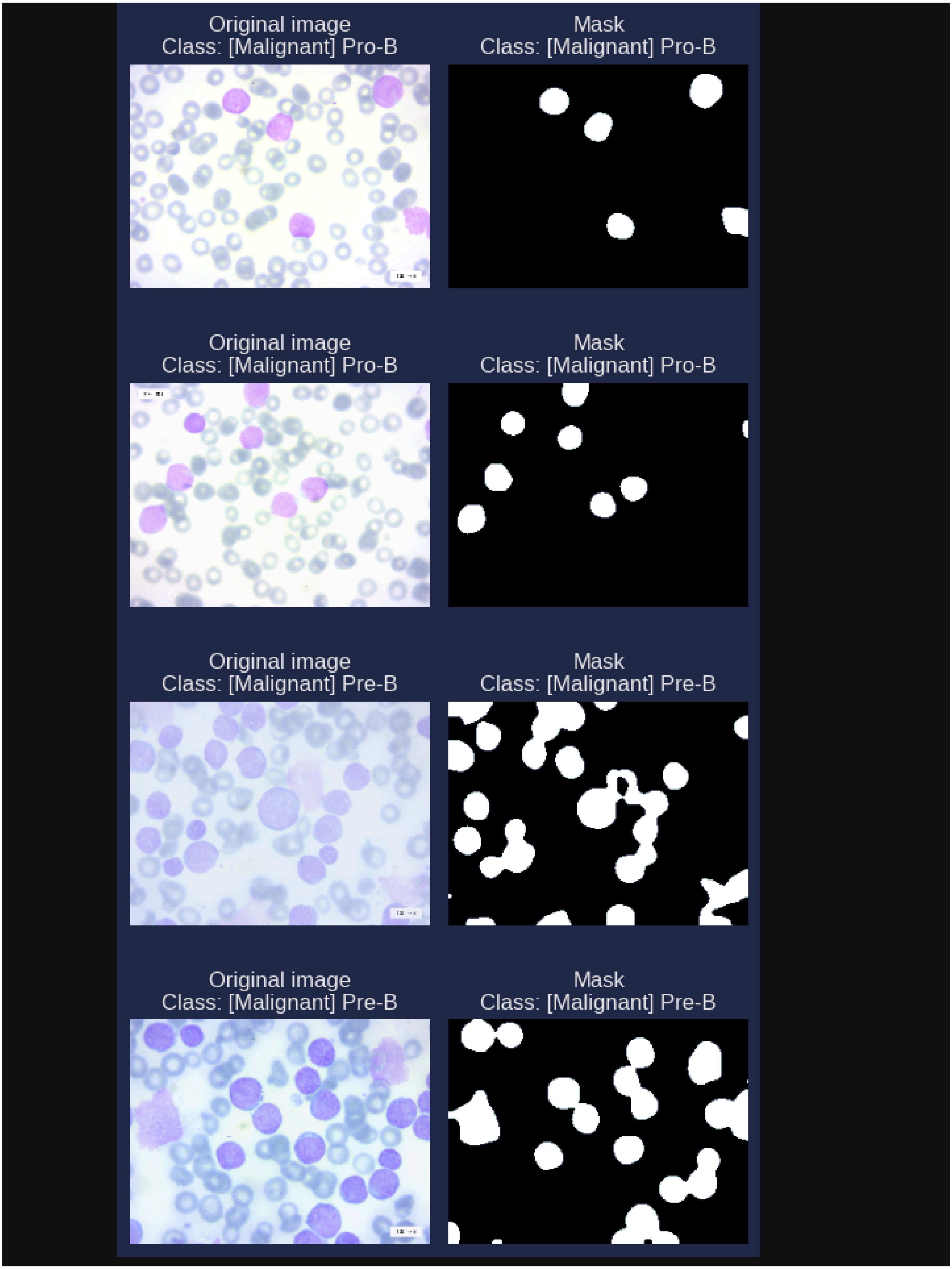
Out[ ]:

```
{'classification_auc': 0.9992965459823608,
 'loss': 0.13611197471618652,
 'segmentation_binary_accuracy': 0.9866564869880676,
 'segmentation_binary_io_u': 0.935377299785614}
```

Applying data augmentation technique

In [ ]:
```
def data_transformation_both(image, mask, class_):
    img = random_brightness(image, max_delta=0.01, seed=seed_value)
    img = random_contrast(img, lower=0.5, upper=1, seed=seed_value)
    img = random_saturation(img, lower=0.5, upper=1, seed=seed_value)
    aux = tf.concat([img, mask], -1)

    aux_ = random_flip_left_right(aux, seed=seed_value)
    aux_ = random_flip_up_down(aux_, seed=seed_value)

    class_ohe = tf.one_hot(indices=tf.squeeze(class_, axis=0), depth=4)

    return tf.cast(aux_[..., :3], tf.float32)/255, {"segmentation":tf.cast(au

train_data_both_aug = train_data.map(data_transformation_both, num_parallel_
train_data_both_aug = train_data_both_aug.batch(batch_size)
train_data_both_aug = train_data_both_aug.prefetch(autotune)
```

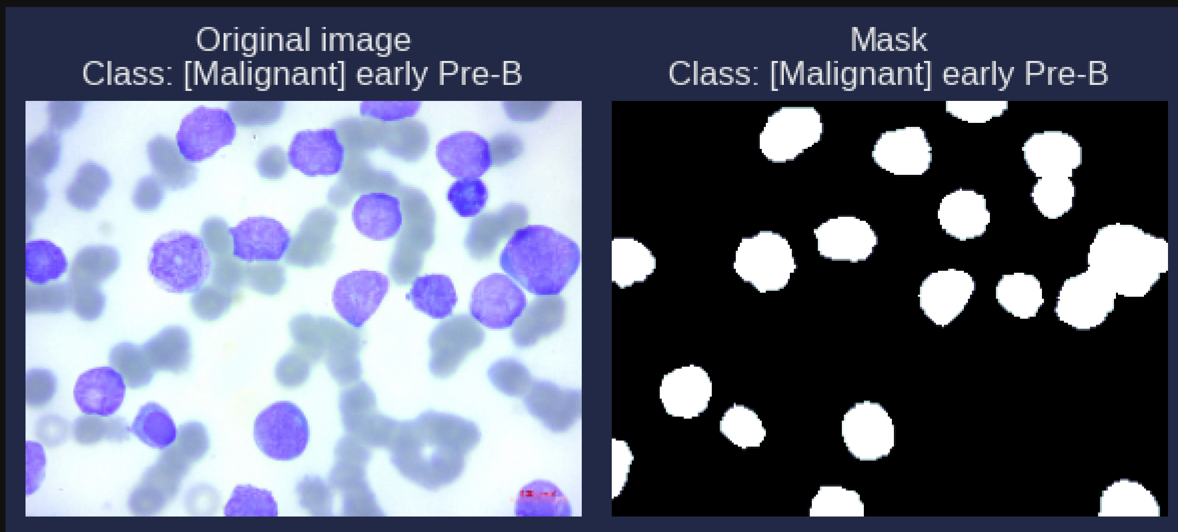
In [ ]:
```
plt.subplots(4, 2, figsize=(5, 10))
for i, (image, pred) in enumerate(train_data_both_aug.take(1).unbatch()):
    mask, class_ = pred["segmentation"], pred["classification"]
    if i == 4:
        break
    plt.subplot(4, 2, (i*2)+1)
    plt.imshow(image)
    plt.axis("off")
    plt.title(f"Original image\nClass: {classes[np.argmax(class_)]}")

    plt.subplot(4, 2, (i*2)+2)
    plt.imshow(mask, cmap="bone")
    plt.axis("off")
    plt.title(f"Mask\nClass: {classes[np.argmax(class_)]}")
plt.tight_layout()
```


Original image
Class: [Malignant] Pro-B
Mask
Class: [Malignant] Pro-B
Original image
Class: [Malignant] Pro-B
Mask
Class: [Malignant] Pro-B
Original image
Class: [Malignant] Pre-B
Mask
Class: [Malignant] Pre-B
Original image
Class: [Malignant] Pre-B
Mask
Class: [Malignant] Pre-B

Let's train

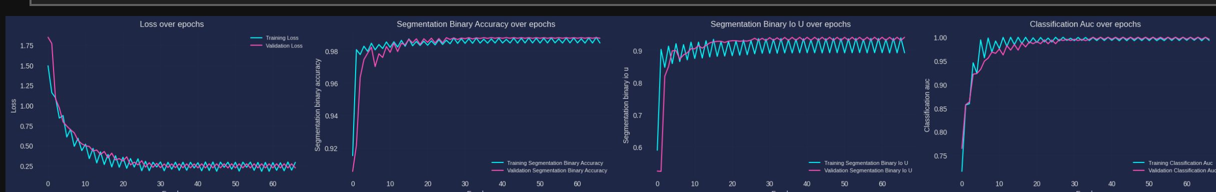
In [ ]:
```
model_base_both_aug = get_model_both()
history_base_both_aug = train_model_both(model_base_both_aug, train_data_bot
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step
```




```
Sample Prediction after epoch 67

70/70 ━━━━━━━━━━━━━━━━━━━━ 24s 338ms/step - classification_auc:
0.9935 - loss: 0.2861 - segmentation_binary_accuracy: 0.9849 - segmentation_b
inary_io_u: 0.9139 - val_classification_auc: 0.9959 - val_loss: 0.2210 - val_s
egmentation_binary_accuracy: 0.9877 - val_segmentation_binary_io_u: 0.9405 -
learning_rate: 1.0000e-13
Epoch 67: early stopping
Restoring model weights from the end of the best epoch: 59.
```

In [ ]:
```
show_history(version="seg_clf_aug", parameters=[["loss", "val_loss"],
                                                ["segmentation_binary_accuracy",
                                                ["segmentation_binary_io_u", "va
                                                ["classification_auc", "val_class
```



Evaluating the baseline model for image segmentation and classification with DA

In [ ]:
```
model_base_both_aug.evaluate(test_data_both, return_dict=True)
```

```
16/16 ━━━━━━━━━━━━━━━━━━━━ 2s 137ms/step - classification_auc:
0.9916 - loss: 0.2695 - segmentation_binary_accuracy: 0.9858 - segmentation_b
inary_io_u: 0.9340
```

```
/usr/lib/python3.10/contextlib.py:153: UserWarning: Your input ran out of data; interrupting training. Make sure that your dataset or generator can generate at least `steps_per_epoch * epochs` batches. You may need to use the `.repeat()` function when building your dataset.
  self.gen.throw(typ, value, traceback)
```

Out[ ]:
```
{'classification_auc': 0.9911665916442871,
 'loss': 0.2748297154903412,
 'segmentation_binary_accuracy': 0.9857146739959717,
 'segmentation_binary_io_u': 0.9311850666999817}
```

Evaluation metrics for both tasks

In [ ]:
```python
for i, (img, output) in enumerate(test_data_both):
    mask, class_ = output["segmentation"], output["classification"]
    aux = tf.reshape(mask,[-1])
    if i == 0:
        y_true_class = class_.numpy()
        y_true_mask = aux.numpy()
    else:
        y_true_class = np.concatenate((y_true_class, class_.numpy()), axis=0
        y_true_mask = np.concatenate((y_true_mask, aux.numpy()), axis=0)

scores_mask = []
scores_class = []
for model in [model_base_both, model_base_both_aug]:
    y_test_pred_proba = model.predict(test_data_both, verbose=0)

    y_test_pred_proba_class = y_test_pred_proba["classification"]
    y_test_pred_class = tf.math.round(y_test_pred_proba_class)
    scores_class.append(get_scores(y_true_class, y_test_pred_class, y_test_p

    y_test_pred_proba_mask = y_test_pred_proba["segmentation"]
    y_test_pred_proba_mask = tf.reshape(y_test_pred_proba_mask,[-1])
    y_test_pred_mask = tf.math.round(y_test_pred_proba_mask)
    score = get_scores(y_true_mask, y_test_pred_mask, y_test_pred_proba_mask
    result_test = model.evaluate(test_data_both, return_dict=True)
    score["BinaryIOU"] = result_test["segmentation_binary_io_u"]
    score["BinaryAccuracy"] = result_test["segmentation_binary_accuracy"]
    scores_mask.append(score)
```

```
16/16 ━━━━━━━━━━━━━━━━━━━━ 2s 134ms/step - classification_auc: 0.9993 - loss: 0.1381 - segmentation_binary_accuracy: 0.9867 - segmentation_binary_io_u: 0.9380
/usr/lib/python3.10/contextlib.py:153: UserWarning: Your input ran out of data; interrupting training. Make sure that your dataset or generator can generate at least `steps_per_epoch * epochs` batches. You may need to use the `.repeat()` function when building your dataset.
  self.gen.throw(typ, value, traceback)
16/16 ━━━━━━━━━━━━━━━━━━━━ 2s 134ms/step - classification_auc: 0.9916 - loss: 0.2695 - segmentation_binary_accuracy: 0.9858 - segmentation_binary_io_u: 0.9340
```

```
/usr/lib/python3.10/contextlib.py:153: UserWarning: Your input ran out of dat
a; interrupting training. Make sure that your dataset or generator can genera
te at least `steps_per_epoch * epochs` batches. You may need to use the `.rep
eat()` function when building your dataset.
  self.gen.throw(typ, value, traceback)
```

For classification task

In [ ]:
```
comparison = pd.DataFrame(data=scores_class, index=["Baseline Model", "Basel
comparison.style.highlight_max(color = 'green', axis = 0).highlight_min(colo
```

Out[ ]:

	AUC_ROC	F1_Score	Accuracy	Precision	Recall
Baseline Model	0.999364	0.979420	0.977413	0.981942	0.977413
Baseline Model with DA	0.989956	0.932592	0.926078	0.943388	0.926078

For segmentation task

In [ ]:
```
comparison = pd.DataFrame(data=scores_mask, index=["Baseline Model", "Baseli
comparison.style.highlight_max(color = 'green', axis = 0).highlight_min(colo
```

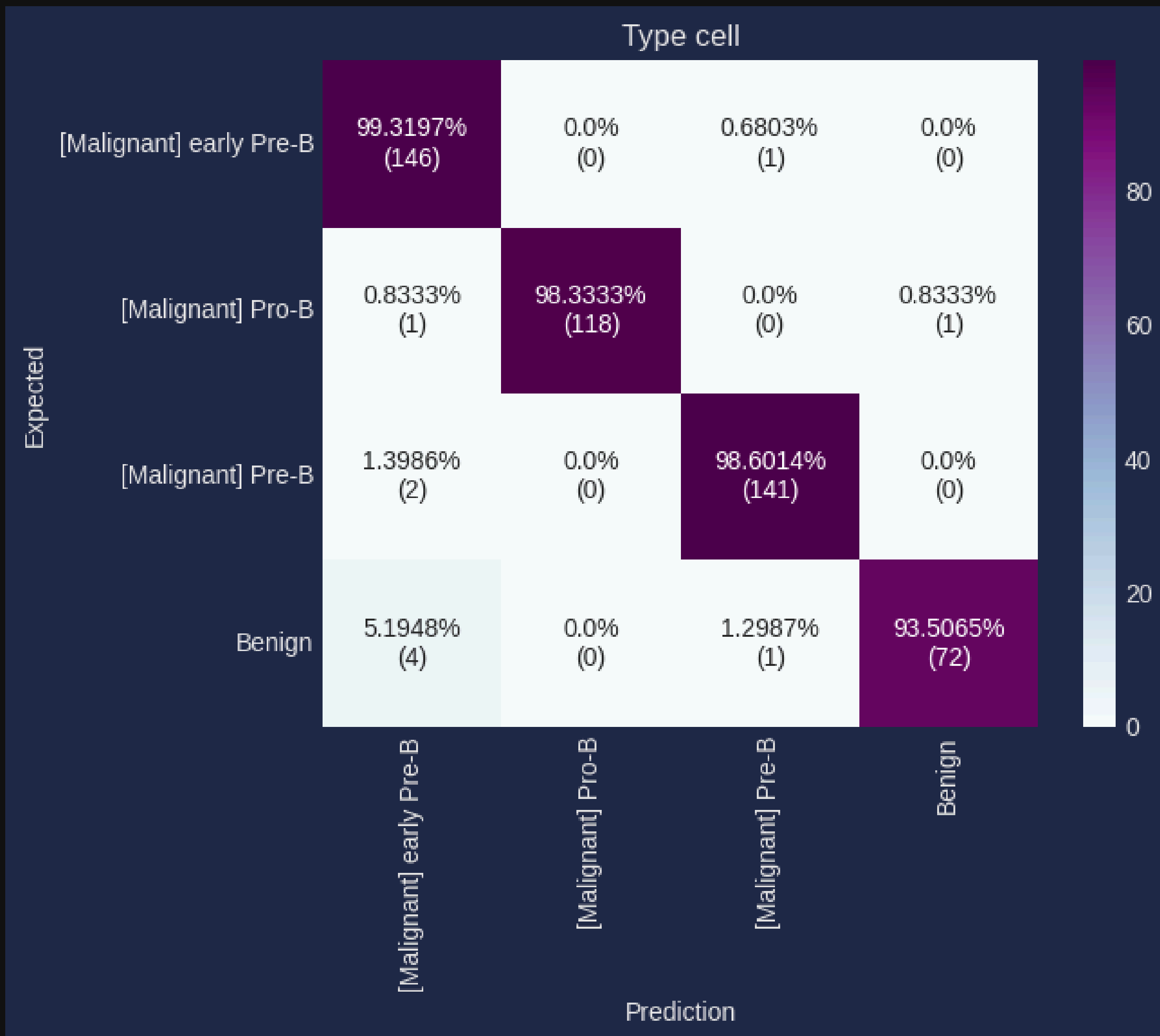
Out[ ]:

	AUC_ROC	F1_Score	Accuracy	Precision	Recall	BinaryIOU	BinaryAccurac
Baseline Model	0.988337	0.986632	0.986656	0.986612	0.986656	0.935377	0.98665
Baseline Model with DA	0.987248	0.985703	0.985715	0.985692	0.985715	0.931185	0.98571

Confusion Matrix for classification

In [ ]:
```
for i, (img, output) in enumerate(test_data_both):
    class_ = tf.argmax(output["classification"], axis=1)
    if i == 0:
        y_true = class_.numpy()
    else:
        y_true = np.concatenate((y_true, class_.numpy()), axis=0)

y_test_pred_proba = model_base_both.predict(test_data_both, verbose=0)
y_test_pred_proba = y_test_pred_proba["classification"]
y_test_pred = tf.argmax(y_test_pred_proba, axis=1)

def my_cm(y_true, y_pred, title):
    cm_val = confusion_matrix(y_true, y_pred)
    cm_pgs = np.round(confusion_matrix(y_true, y_pred, normalize='true')*100

    formatted_text = (np.asarray([f"{pgs}%\n({val})" for val, pgs in zip(cm_

    sns.heatmap(cm_pgs, annot=formatted_text, fmt='', cmap='BuPu', yticklabe
    plt.title(title)
```

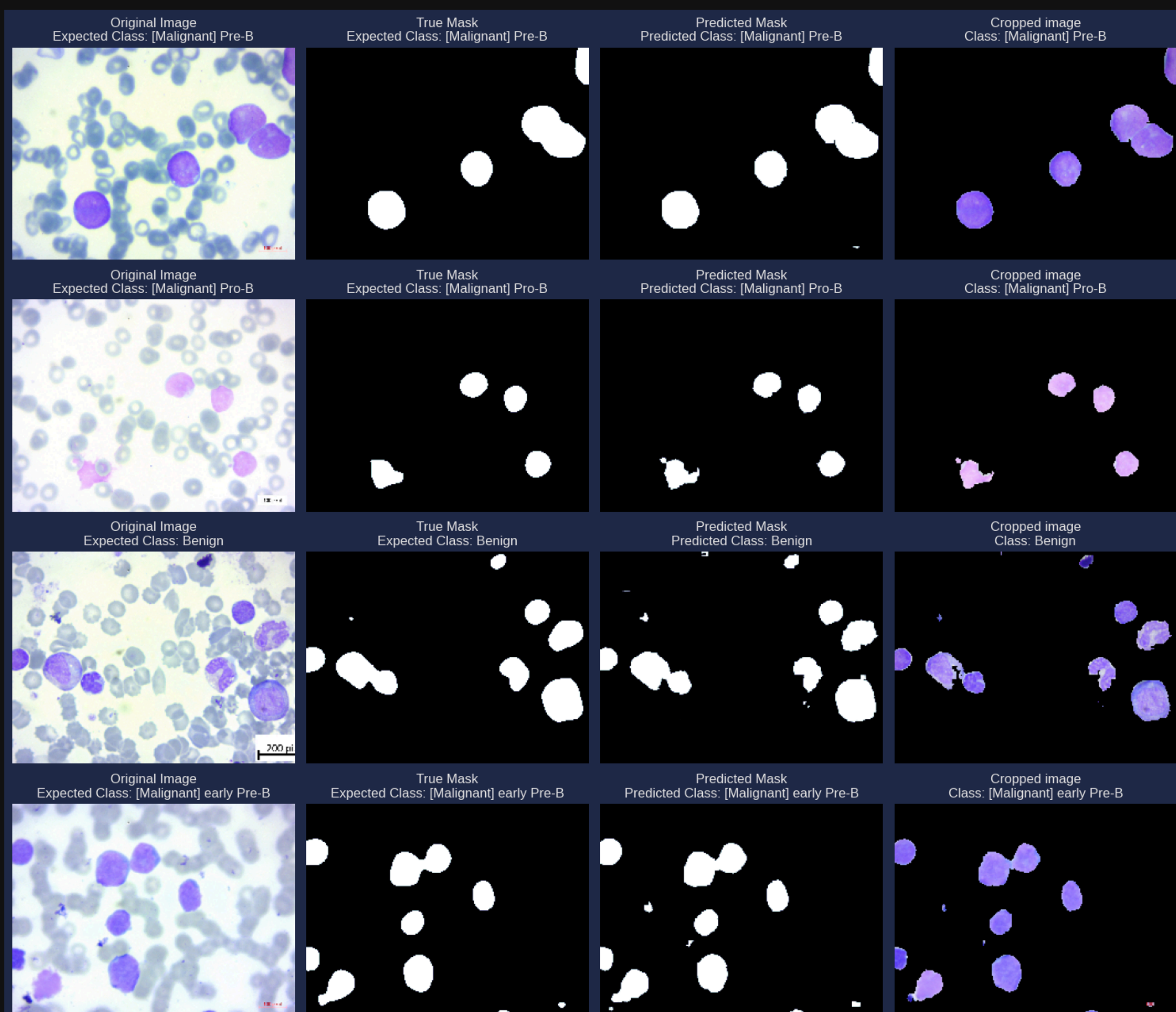
```
    plt.xlabel("Prediction")
    plt.ylabel("Expected")
    return

my_cm(y_true, y_test_pred, title="Type cell")
```



Let's take a look at the predictions.

In [ ]:
```
plt.subplots(4, 4, figsize=(15, 13))
for i, (image, output) in enumerate(test_data_both.shuffle(100).take(1).unbatc
    if i == 4:
        break

    pred = model_base_both.predict(tf.expand_dims(image, axis=0), verbose=0)

    class_ = np.argmax(output["classification"])
    pred_class_proba = pred["classification"]
    pred_class = tf.argmax(pred_class_proba, axis=1)

    mask = output["segmentation"]
    pred_mask = tf.squeeze(pred["segmentation"], axis=0)
```

```
    image_cropped = apply_mask(tf.cast(image*255, tf.uint8).numpy(), tf.cast

    plt.subplot(4, 4, (i*4)+1)
    plt.imshow(image)
    plt.axis("off")
    plt.title(f"Original Image\nExpected Class: {classes[class_]}")

    plt.subplot(4, 4, (i*4)+2)
    plt.imshow(mask, cmap="bone")
    plt.axis("off")
    plt.title(f"True Mask\nExpected Class: {classes[class_]}")

    plt.subplot(4, 4, (i*4)+3)
    plt.imshow(pred_mask, cmap="bone")
    plt.axis("off")
    plt.title(f"Predicted Mask\nPredicted Class: {classes[int(pred_class)]}"

    plt.subplot(4, 4, (i*4)+4)
    plt.imshow(image_cropped)
    plt.axis("off")
    plt.title(f"Cropped image\nClass: {classes[int(pred_class)]}")

plt.tight_layout()
```



Converting Notebook to PDF

In [ ]:
```
!apt-get install texlive-xetex texlive-fonts-recommended texlive-plain-gener
!pip install pypandoc nbconvert[webpdf]
!playwright install
!playwright install-deps

from google.colab import drive
from IPython.display import clear_output
drive.mount('/content/drive')
clear_output(wait=False)
```

In [ ]:
```
%%capture
!jupyter nbconvert --to webpdf /content/drive/MyDrive/ML_Zoomcamps_2024/Caps
```

Thank you for your time! 😃