Skin Cancer Detection and Classification on HAM10000 with 100% Accuracy

I use Kaggle's Notebook for building this model because of easy access to GPUs and TPUs.

The Dataset :- The Dataset was HAM10000. It consists of 10015 images together in two different folders named part 1 and part 2. There are some CSV files containing pixelwise data about the images but I won't be using them in this project. There is also a csv consisting of metadata about the images. The metadata for each image can be identified by the image id attribute, which is actually the filename for that image. The metadata had information about Lesion_id, age, gender, localisation of the lesion in the body, the method involved in the diagnosis and the type of the diagnostic. The type of the diagnostic is what I will be targeting as labels for classification on this project.

Preparing the data :- I start by first reading the metadata csv using pandas. I name the data frame cancer. Upon inspecting the data using info() method, I find that the age column contains some null values, which I replace with the mean age. It was not needed in the end; I could have chosen to drop the age column and everything would have been still fine. Next, I add a column to the data frame containing the paths of the image file about which the metadata is featured in the data frame. I do it by first creating a dictionary of the image paths and the image ids and then mapping the image id column to this dictionary. Also, labels are extracted from the 'dx' column which contains the type of diagnostic and then one-hot encoding is run on it. I find that there are 7 different classes of diagnostic.

```
In [3]: cancer=pd.read_csv("/kaggle/input/skin-cancer-mnist-ham10000/HAM10000_metadata.csv")

In [4]: cancer.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10015 entries, 0 to 10014
Data columns (total 7 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   lesion_id     10015 non-null  object 
 1   image_id      10015 non-null  object 
 2   dx            10015 non-null  object 
 3   dx_type       10015 non-null  object 
 4   age           9958 non-null   float64
 5   sex           10015 non-null  object 
 6   localization  10015 non-null  object 
dtypes: float64(1), object(6)
memory usage: 547.8+ KB
```

In [6]:
```
cancer.head()
```

Out[6]:

	lesion_id	image_id	dx_type	age	sex	localization	path
0	HAM_0000118	ISIC_0027419	histo	80.0	male	scalp	/kaggle/input/skin-cancer-mnist-ham10000/HAM10...
1	HAM_0000118	ISIC_0025030	histo	80.0	male	scalp	/kaggle/input/skin-cancer-mnist-ham10000/HAM10...
2	HAM_0002730	ISIC_0026769	histo	80.0	male	scalp	/kaggle/input/skin-cancer-mnist-ham10000/HAM10...
3	HAM_0002730	ISIC_0025661	histo	80.0	male	scalp	/kaggle/input/skin-cancer-mnist-ham10000/HAM10...
4	HAM_0001466	ISIC_0031633	histo	75.0	male	ear	/kaggle/input/skin-cancer-mnist-ham10000/HAM10...

In [7]:
```
labels.head()
```

Out[7]:

	dx_akiec	dx_bcc	dx_bkl	dx_df	dx_mel	dx_nv	dx_vasc
0	0	0	1	0	0	0	0
1	0	0	1	0	0	0	0
2	0	0	1	0	0	0	0

Next, I take the path column and the convert the labels data frame to a list of one-hot vectors and the zip them both and convert that to a list. Thus, I get a preliminary dataset containing the paths of the images and the labels for them. I shuffle the data and then unzip it and then, I use the from_tensor_slices() method to get our data in the form of tensors.

In [8]:
```
preliminary_data=list(zip(cancer['path'],labels.values.tolist()))
```

In [9]:
```
random.shuffle(preliminary_data)
paths,labels=zip(*preliminary_data)
```

In [10]:
```
data=tf.data.Dataset.from_tensor_slices((list(paths),list(labels)))
```

Then, I write a function name final_data to which I map the dataset to get our final dataset. The function takes the path and label as input, reads the image and decodes it and resizes it to 90 x 120 from 450x600. The values are from 0 to 255, so to prepare it to feed it into the model, I scale it from 0 to 1. Then, I return the decoded image data and the labels and thus our final dataset is ready.

In [11]:
```
def final_data(path,label):
    image = tf.io.read_file(path)
    image = tf.image.decode_jpeg(image,channels=3)
    image = tf.image.resize(image,[90,120])
    image = image/255
    return image,label
```

In [12]:
```
data=data.map(final_data).prefetch(30)
```

But, Now I will have to split the data in train, test and validation sets as Ill and that is exactly what I do.

In [13]:
```
train_size=round(0.8*10015)
val_size=round(0.1*10015)
test_size=10015-train_size-val_size
```

In [14]:
```
train=data.take(train_size)
val=data.skip(train_size)
test=data.skip(train_size)
val=data.take(val_size)
test=data.take(test_size)
```

With the dataset now prepared and split, it is time to get to model building.

However, before getting to the model building part, I would like to specify on thing. This notebook is being written on Kaggle so that I can access the GPU for faster training. I will be using dual T4 GPUs provided by Kaggle for this project. I will be using the mirrored strategy to distribute computing across both GPU's as TensorFlow by default uses only the first GPU.

In [16]:
```
mirrored_strategy = tf.distribute.MirroredStrategy(devices=["/gpu:0", "/gpu:1"])
```

Choosing the Correct kind of model :- Since the metadata contained information about age, sex, method for diagnosis, lesion id and localisation, I was initially tempted to go for a very complex model which took 4 different sets of inputs, age, one-hot encoded categorical-features(sex, method for diagnosis, localisation), lesion id which will go to an embedding layer and the image input which will go to a layers of CNNs or for transfer learning on a pretrained model like ResNet or Xception. However, the architecture of this model seemed to be a bit too complex for the task at hand and I thought, why not just try a relatively simple model which takes just the image input, pass it to a layer of convolutions, add some dropout and batch normalization and see how it goes. In case it didn't perform well, I could always build the multi-input model. And, as it will turn out, that this relatively simple model did perform well and achieved an accuracy of 100% on the test set.

Building and training the model :- I begin by specifying the strategy(which is the mirrored strategy in this case) by calling the scope() method. Then, I create a sequential model. The tf.keras module and its component classes are used extensively in order to build this model.

The first layer is a 2D-Convolution with 30 filters and kernel size 5x5 and stride 1x1.Just to specify, I have kept stride as 1x1 throughout all the Convolutional layers in the model as I want minimum loss of data while also being able to reduce dimensionality. I don't use any Zero- padding so padding is set to 'valid'. The reason behind it is that Zero-padding helps in stronger representation of pixels near the boundaries of the images in the deeper-layers of the models but since majority of our data regarding lesions is in the inner part of the images, I won't be needing it. Thus, the same padding value is retained throughout the model. I use the relu activation function throughout the model except the output layer which happens to be the standard practice.

```
with mirrored_strategy.scope():
    model=tf.keras.Sequential()
    model.add(tf.keras.layers.Conv2D(30,(5,5),strides=(1,1),padding='valid',activation='relu',input_shape=(90,120,3)))
    model.add(tf.keras.layers.Conv2D(30,(3,3),strides=(1,1),padding='valid',activation='relu'))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.MaxPool2D(pool_size=(2,2),strides=None,padding='valid'))
    model.add(tf.keras.layers.Conv2D(20,(3,3),strides=(1,1),padding='valid',activation='relu'))
    model.add(tf.keras.layers.Conv2D(15,(3,3),strides=(1,1),padding='valid',activation='relu'))
    model.add(tf.keras.layers.Conv2D(15,(3,3),strides=(1,1),padding='valid',activation='relu'))
    model.add(tf.keras.layers.GroupNormalization(groups=3))
    model.add(tf.keras.layers.MaxPool2D(pool_size=(2,2),strides=None,padding='valid'))
```

Note:- The shapes mentioned here are for each individual instance and not the entire batch.

Now the input shape in the first layer is (90,120,3) which is changed to (86,116,30) after passing through the first layer. Next, I add another 2D-Convolution layer with 30 filters and 3x3 kernel. This further concentrates the features, albeit by a small margin. This layer has an output shape of (84,114,30). At this stage, I add a batch normalisation layer and 2D- MaxPool layer with a pool size of 2x2 to capture the strongest features and discard the rest. This layer reduces the shape of the first 2-dimensions by half thus output shape at this stage is (42,57,30). Then, I add 3 more 2D Convolution layers one after another with kernel size of 3x3 & 20,15 and 15 filters each. The output shape after the last such layer happens to be (36,51,15). At, this stage I add a Group normalization layer with three groups which divides the 15 channels into three groups of 5 each and performs normalization across each group. Then, again I add a max-pooling layer with pool-size (2,2) to capture the strongest features and discard the rest while also reducing the output shape to (18,25,15).

I add one more 2D-Convolution layer with 10 filters and kernel size of 3x3 giving an output of shape (16,23,10) and then flatten and Batch Normalize the

output of this layer using a Flatten layer and a Batch Normalization layer. After flattening and batch normalization, the output takes a shape of (3680).

```
    model.add(tf.keras.layers.GroupNormalization(groups=3))
    model.add(tf.keras.layers.MaxPool2D(pool_size=(2,2),strides=None,padding='valid'))
    model.add(tf.keras.layers.Conv2D(10,(3,3),strides=(1,1),padding='valid',activation='relu'))
    model.add(tf.keras.layers.Flatten())
    model.add(tf.keras.layers.Normalization())
    model.add(tf.keras.layers.Dense(256,activation='relu'))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Dropout(0.1))
    model.add(tf.keras.layers.Dense(128,activation='relu'))
    model.add(tf.keras.layers.Dense(7,activation='softmax'))
    model.compile(optimizer="Adam",loss='categorical_crossentropy',metrics=['accuracy'])
```

I connect it to a dense layer with 256 neurons and then add Batch Normalization and dropout to reduce any chances of overfitting. I then add a Dense layer with 128 neurons and finally connect it with the output Dense layer with 7 neurons with a softmax activation as there are 7 classes of diagnostic lesions in our task. I compile the model and I are ready to move to training, but first I must write some callbacks. Also, below is the architecture of the model.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 86, 116, 30)       2280

 conv2d_1 (Conv2D)           (None, 84, 114, 30)       8130

 batch_normalization (BatchN  (None, 84, 114, 30)      120
 ormalization)

 max_pooling2d (MaxPooling2D  (None, 42, 57, 30)       0
 )

 conv2d_2 (Conv2D)           (None, 40, 55, 20)        5420

 conv2d_3 (Conv2D)           (None, 38, 53, 15)        2715

 conv2d_4 (Conv2D)           (None, 36, 51, 15)        2040

 group_normalization (GroupN  (None, 36, 51, 15)       30
 ormalization)

 max_pooling2d_1 (MaxPooling  (None, 18, 25, 15)       0
 2D)

 conv2d_5 (Conv2D)           (None, 16, 23, 10)        1360

 flatten (Flatten)           (None, 3680)              0

 normalization (Normalizatio  (None, 3680)             7361
 n)

 dense (Dense)               (None, 256)               942336

 batch_normalization_1 (Batc  (None, 256)              1024
 hNormalization)

 dropout (Dropout)           (None, 256)               0

 dense_1 (Dense)             (None, 128)               32896

 dense_2 (Dense)             (None, 7)                 903

=================================================================
Total params: 1,006,615
Trainable params: 998,682
Non-trainable params: 7,933
```

Callbacks:- I will be writing 3 very important callbacks, 2 of which are pretty standard, Early Stopping and Model saving checkpoint. The Third one which I will write is a learning rate scheduler which reduces the learning rate as the training progresses. I had also written a memory clear callback which I might have used in case the GPU was becoming too overloaded but didn't need that.

```
with mirrored_strategy.scope():
    checkpoint= tf.keras.callbacks.ModelCheckpoint(filepath='/kaggle/working/skin_cancer_detection7.h5',save_weights_only=Fa
                                                   monitor='val_accuracy',save_best_only=True,save_freq="epoch",)
    early_stopping= tf.keras.callbacks.EarlyStopping(monitor='val_loss',patience=10,restore_best_weights=True)
    def lr_scheduler(epoch,lr,epochs=50):
        initial=1e-3
        if epoch<epochs*0.1:
            return initial
        elif epoch>epochs*0.1 and epoch<epochs*0.25:
            lr*=tf.math.exp(-0.1)
            return lr
        else:
            lr*=tf.math.exp(-0.008)
            return lr
    lr_scheduling=tf.keras.callbacks.LearningRateScheduler(lr_scheduler)
    # Memory clear on every epoch
    class RAM_reclaim(tf.keras.callbacks.Callback):
        def on_epoch_end(self,epoch,logs=None):
            gc.collect()
    ram_reclaim=RAM_reclaim()
```

Now, I fit the model and train it. I have kept batch size 60 and it indeed performed better with this large batch size instead of smaller batch sizes of 8,12,16,20,28 and 32 which I tried earlier and didn't work out this good.

n [21]:
```
with mirrored_strategy.scope():
    history = model.fit(train.batch(60),epochs=50,validation_data=val.batch(60),
                        callbacks=[checkpoint,early_stopping,lr_scheduling],shuffle=True)
```

After training, if I go through the training logs, I can observe that there's not much difference betIen training accuracy and validation accuracy, thus the model has not overfitted.

Next, I move to testing. I load the best save version of the model and pass it the entire test set to get the predictions.

]:
```
model1=tf.keras.models.load_model("/kaggle/working/skin_cancer_detection7.h5")
```

]:
```
predictions=model1.predict(test.batch(len(test)))
```

```
1/1 [==============================] - 6s 6s/step
```

]:
```
predictions[0]
```

]:
```
array([3.25512839e-10, 2.28691076e-11, 2.16605383e-08, 2.73373257e-09,
       1.31461335e-11, 1.00000000e+00, 3.13129661e-11], dtype=float32)
```

The predictions are in probability form so I write a function named outputs to convert it into the labels form.

```
def outputs(x):
    a = np.zeros(x.shape)
    a[np.where(x==np.max(x))] = 1
    return a
```

```
for i in range(len(predictions)):
    predictions[i]=outputs(predictions[i])
```

```
predictions
```

```
array([[0., 0., 0., ..., 0., 1., 0.],
       [0., 0., 0., ..., 0., 1., 0.],
       [0., 0., 0., ..., 1., 0., 0.],
       ...,
```

Also, the labels are batched in the test set, so I extract the labels and then calculate the accuracy score, which as it turns out to be is 100%.

```
from sklearn.metrics import accuracy_score
```

```
y_test = np.concatenate([y for x, y in test.batch(len(test))], axis=0)
```

```
y_test
```

```
array([[0, 0, 0, ..., 0, 1, 0],
       [0, 0, 0, ..., 0, 1, 0],
       [0, 0, 0, ..., 1, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 1, 0],
       [0, 0, 0, ..., 0, 1, 0],
       [0, 0, 0, ..., 1, 0, 0]], dtype=int32)
```

```
accuracy_score(predictions,y_test)
```

```
1.0
```