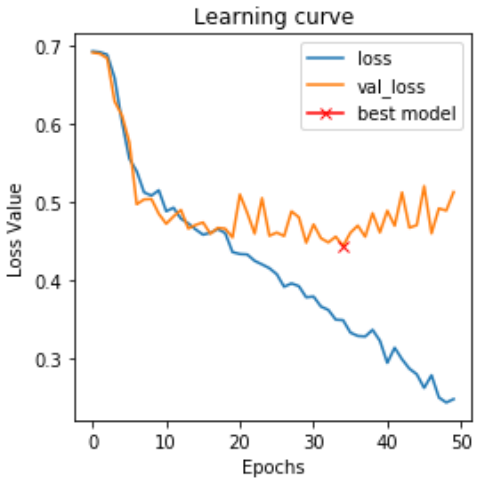
Laboratory Assignment 2

Regularization Techniques

Task1a) Employ the AlexNet model with the following architecture: five convolutional layers (base=8), followed by three dense layers (64,64,1), three max-pooling layers after 1st, 2nd, and 5th blocks; LR=0.0001, batch size=8, Adam optimizer, and image size=(128,128,1). Train this model for 50 epochs on skin images. What is the final training accuracy?

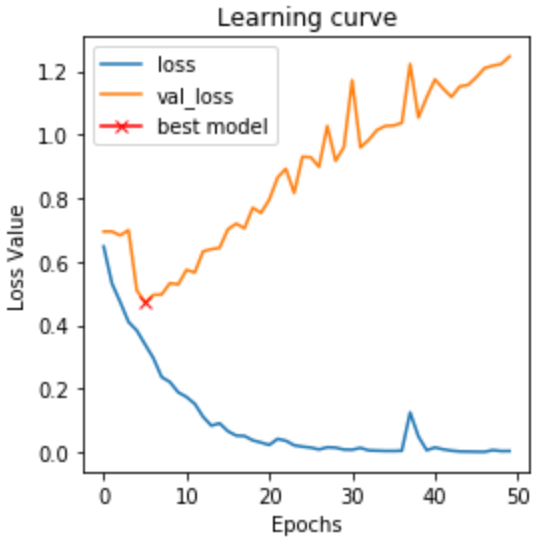


Without Batch Normalization

Epoch 50/50
1000/1000 [==============================] - 1s 792us/sample - loss: 0.2471 - accuracy: 0.9030 - val_loss: 0.5128 - val_accuracy: 0.8050

Task1b) With the same model and same settings, now, insert a batch normalization layer at each convolutional blocks (right after convolution layer and before activation function). At which epoch do you observe the same training accuracy as task1a?
Epoch 8/50
1000/1000 [==============================] - 1s 1ms/sample - loss: 0.2370 - accuracy: 0.9080 - val_loss: 0.4969 - val_accuracy: 0.8250

What is the final training accuracy? What is the effect of batch normalization layer?

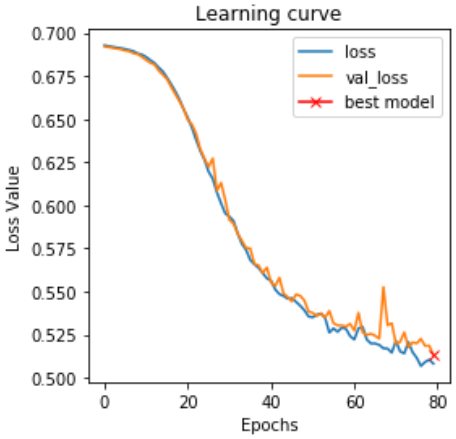


With Batch Normalization

Epoch 50/50
1000/1000 [==============================] - 1s 1ms/sample - loss: 0.0043 - accuracy: 1.0000 - val_loss: 1.2460 - val_accuracy: 0.7900

The effect of batch normalization is stabilize the learning by speeding up the process. Basically the role is to transfer the output data of the layers into the range 0-1. It was observed that with adding batch normalization it reaches the training accuracy as _

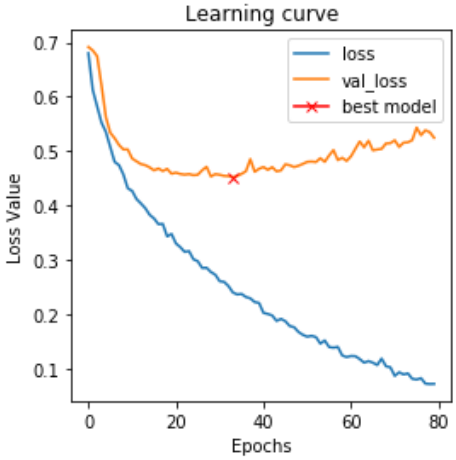
Task1c) Train again the same model with exact same parameters except LR = 0.00001 and epochs = 80 with and without batch normalization layers. Focus on validation accuracy.



Without Batch Normalization

Epoch 80/80

1000/1000 [==============================] - 1s 890us/sample - loss: 0.5082 - accuracy: 0.7600 - val_loss: 0.5133 - val_accuracy: 0.7750



With Batch Normalization

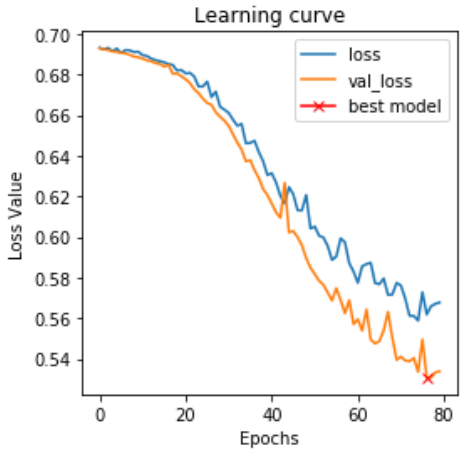
Epoch 80/80
1000/1000 [==============================] - 1s 1ms/sample - loss: 0.0726 - accuracy: 0.9900 - val_loss: 0.5247 - val_accuracy: 0.7800

Which model resulted in higher validation accuracy?

The model with batch normalization resulted higher validation accuracy.

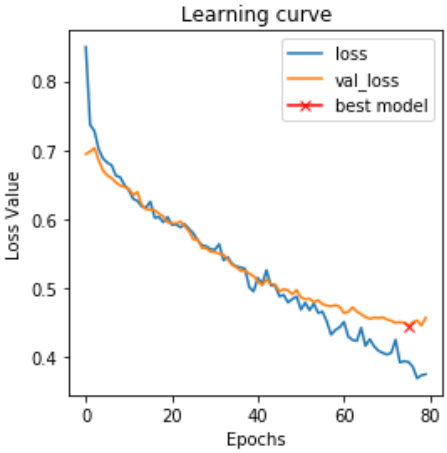
Which model resulted in higher generalization power? How do you justify the effect of batch normalization?

Task2) Use the same model as task 1c with the exact same parameters. Add drop out layers after the first two fully connected layers (right after activation layers with drop out rate of 0.4). Run the model with/without batch normalization layers and discuss the observed results.



Without Batch Normalization

Epoch 80/80
1000/1000 [==============================] - 1s 872us/sample - loss: 0.5678 - accuracy: 0.7290 - val_loss: 0.5338 - val_accuracy: 0.7600



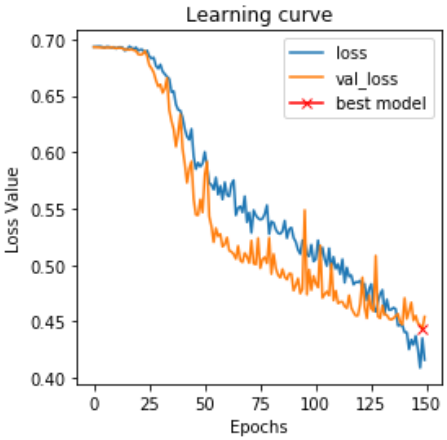
With Batch Normalization

Epoch 80/80
1000/1000 [==============================] - 1s 1ms/sample - loss: 0.3749 - accuracy: 0.8450 - val_loss: 0.4565 - val_accuracy: 0.8450

How does drop out layer help your model?

Dropout layer is basically used to drop the neurons from the hidden layers so that if we don't have sufficient amount of data then by doing so we avoid overfitting and improving generalization.

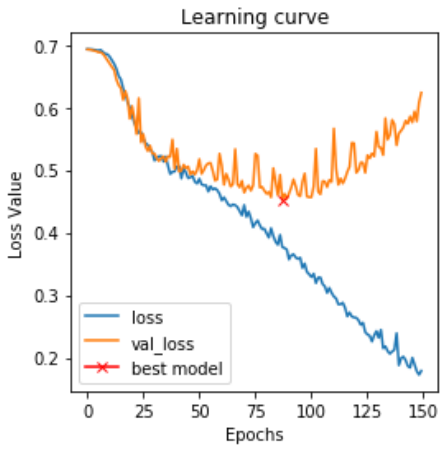
Task3) Use the same model as task1c but set the Base parameters as 64 and remove the batch normalization layers. Instead, insert spatial dropout layers at each convolutional blocks (after activation function, and before max-pooling). Set the dropout rate of spatial drop out layers as 0.1 and the rate of 0.4 for the drop out layers after the first fully connected layers. Then let the model runs for 150 epochs with LR=0.00001. Save the loss and accuracy values for the validation data.



Without Batch Normalization, With Spatial Dropout

Epoch 150/150
1000/1000 [==============================] - 2s 2ms/sample - loss: 0.4161 - accuracy: 0.8110 - val_loss: 0.4543 - val_accuracy: 0.8250

Then, run the same model with the same settings but remove all the drop out layers.



Without Batch Normalization, Without Spatial Dropout

Epoch 150/150
1000/1000 [==============================] - 2s 2ms/sample - loss: 0.1790 - accuracy: 0.9360 - val_loss: 0.6236 - val_accuracy: 0.7600

Which models perform better?

The model with dropout layer performs better because it prevents overfitting.

Which models resulted in higher generalization power? In general, discuss how the drop out layers would be helpful?

Dropout prevents overfitting due to deactivating some neurons in the training randomly so that it doesn't map the same input as an output thereby improving generalization

Task4) Try to improve the performance of VGG16 model while prevent it from overfitting by using batch normalization, spatial dropout, and/or drop out techniques. Tune the model parameters to achieve the best classification accuracy over the validation set and save the observed results for both skin and bone data.

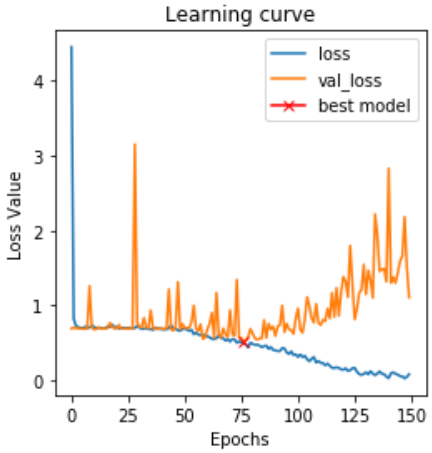
On Skin data:

With following parameters:

n_epochs = 150
Batch_Size = 8
LR = 0.00001
Base = 32
loss='sparse_categorical_crossentropy', optimizer=adam, metrics=['accuracy']

Epoch 150/150

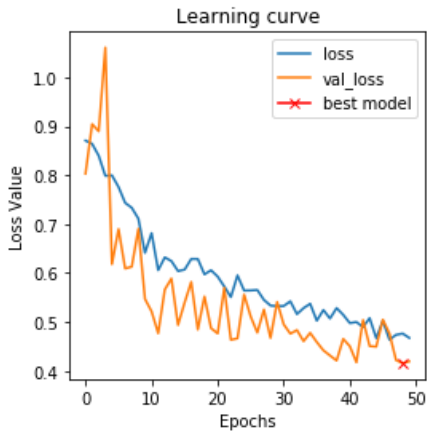
1000/1000 [==========] - 3s 3ms/sample - loss: 0.0852 - accuracy: 0.9720 - val_loss: 1.1070 - val_accuracy: 0.7250



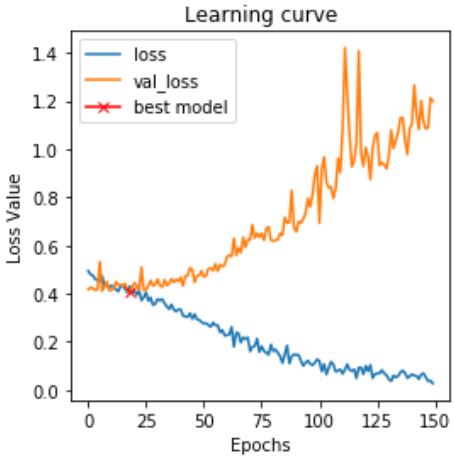
With following parameters:

n_epochs = 50
Batch_Size = 8
LR = 0.00001
Base = 32
loss='sparse_categorical_crossentropy', optimizer=Adam(lr=LR), metrics=['accuracy']

Epoch 50/50
1000/1000 [==========] - 3s 3ms/sample - loss: 0.4675 - accuracy: 0.7890 - val_loss: 0.4195 - val_accuracy: 0.8450

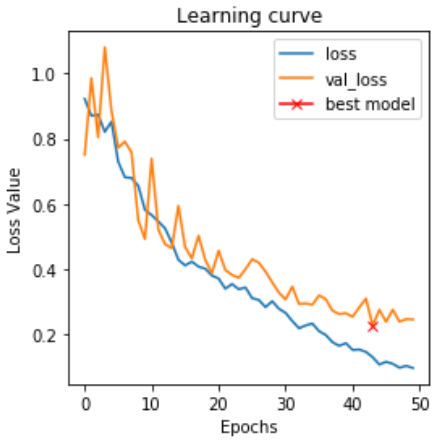


Epoch 150/150
1000/1000 [==============================] - 3s 3ms/sample - loss: 0.0270 - accuracy: 0.9920 - val_loss: 1.1994 - val_accuracy: 0.7900



On Bone Data:
With the parameter setting as:
n_epochs = 50 and 150
Batch_Size = 8
LR = 0.00001
Base = 32
model_VGG.compile(loss='sparse_categorical_crossentropy', optimizer=Adam(lr=LR), metrics=['accuracy'])

Epoch 50/50
1072/1072 [==============================] - 3s 3ms/sample - loss: 0.0983 - accuracy: 0.9571 - val_loss: 0.2469 - val_accuracy: 0.8929



Data Augmentation

Task5a) Read the following codes to find out how they work. Then, change the following parameters scale_factor, Angle, Min and Max percentile to see their effects.

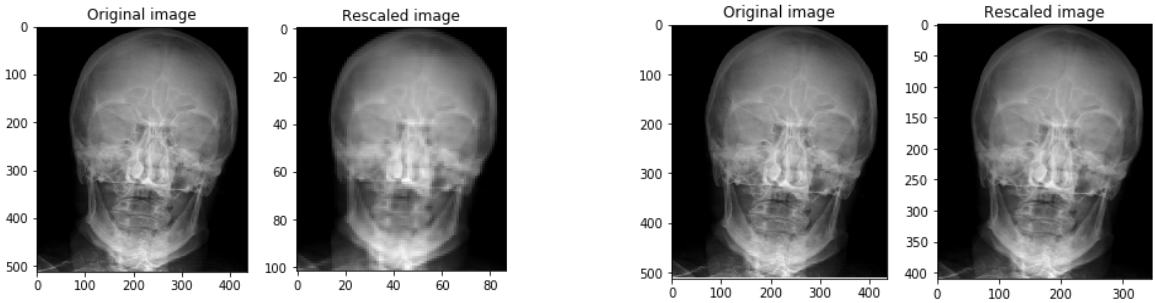


Fig: Rescaled with factor of 0.2 and 0.8 respectively

Rescaling is the best way to deal with different sizes of images in the dataset, to match the desired dimension available in the dataset.

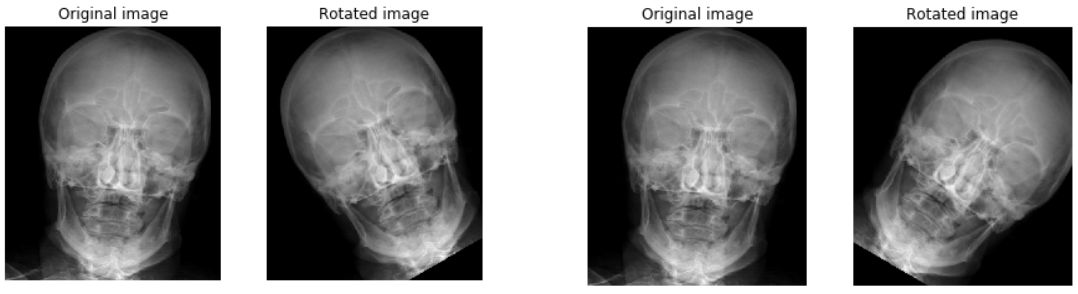


Fig: Rotation with the angle factor of 30° and -30°

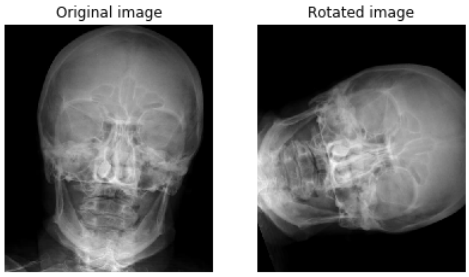


Fig: Rotation with angle factor of -70°

One key thing to note about this operation is that image dimensions may not be preserved after rotation. If the image is a square, rotating it at right angles will preserve the image size. If it's a rectangle, rotating it by 180 degrees would preserve the size. Rotating the image by finer angles will also change the final image size.

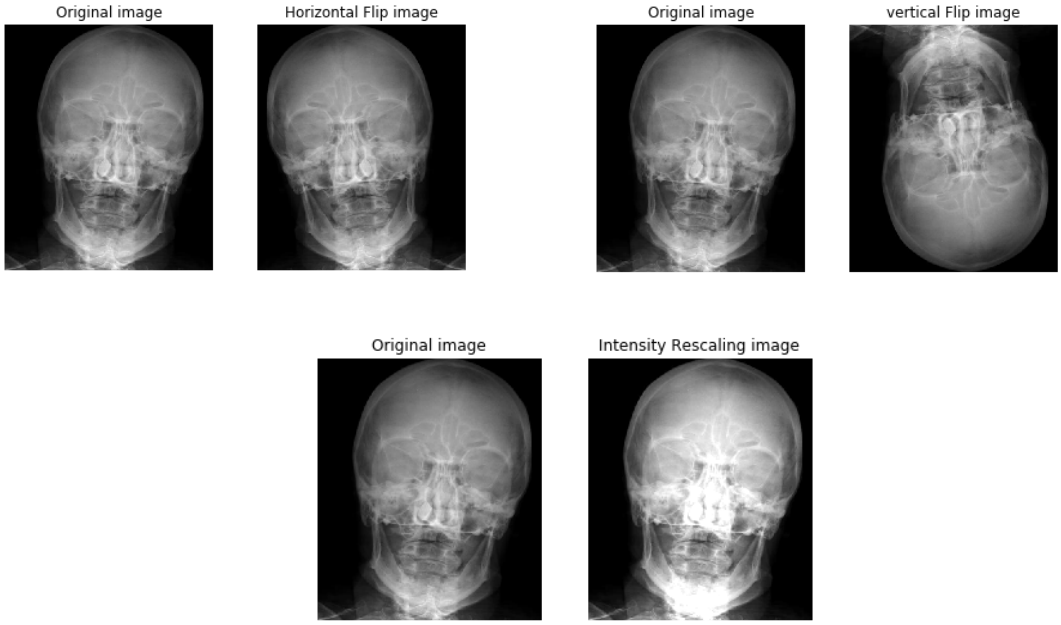
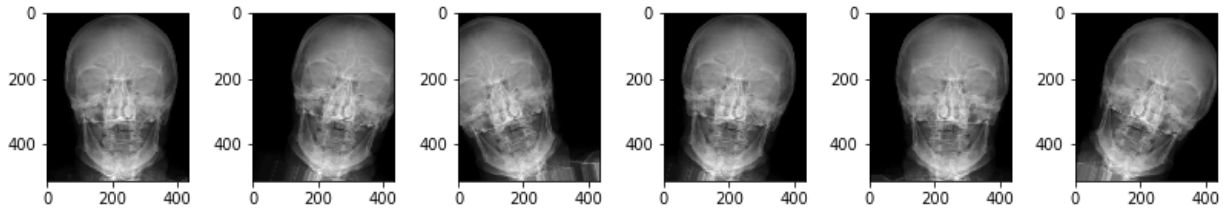


Fig: Intensity scaling

With all these data augmentation techniques we learn how to increase the dataset samples if we have limited data. However, we have to keep this in mind that the

technique which we apply to the dataset to make it bigger should resemble the real world scenario else our model will not learn properly the features and the prediction of accuracy will not be accurate.

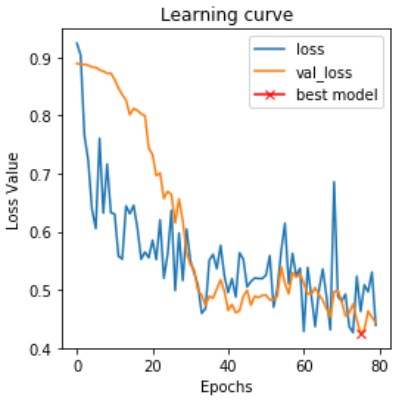
Task5b) A practical way to perform data augmentation is to develop a generator. The following code is an example of how you can generate random augmented images with tensorflow built-in generator. Search for other options of the ImageDataGenerator and apply them in code and try to find out how it works.



Task6) Develop a framework for training the models with augmenting the training data. Apply image augmentations on training data including rotation_range=10, width_shift_range=0.1, height_shift_range=0.1, rescale=1./255, and horizontal_flip=True. However, for validation data you just need to apply rescale=1./255.

Use the AlexNet model with the batch normalization layers, and drop out layers for the first two dense layers(rate=0.4). Set the Base parameter as 64, and assign 128 neurons for the first dense layer and 64 for the second one. LR=0.00001, batch-size=8 and train the model on skin images for for 80 epochs. Does data augmentation help to improve the model performance? Why?

With binary_accuracy and 3 channels:

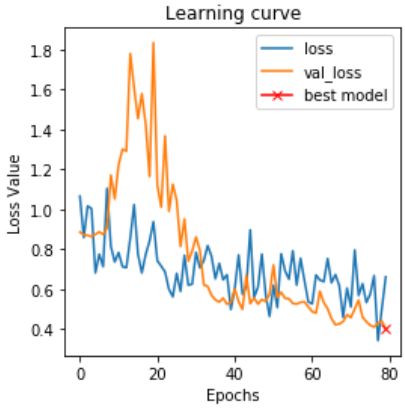


Epoch 80/80 16/15 [==============================] - 1s 47ms/step - loss: 0.4390 - binary_accuracy: 0.8359 - val_loss: 0.4437 - val_binary_accuracy: 0.8438

Task7) Repeat task6 for VGG model for both skin and bone data set.

For Skin Dataset:

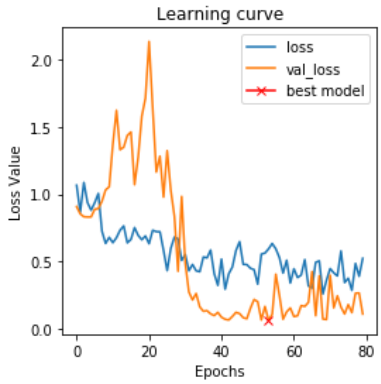
Epoch 80/80
16/15 [==============================] - 1s 53ms/step - loss: 0.6623 - accuracy: 0.6953 - val_loss: 0.4028 - val_accuracy: 0.8438



For Bone Dataset:

Epoch 80/80

18/17 [==============================] - 7s 414ms/step - loss: 0.5237 - accuracy: 0.8542 - val_loss: 0.1114 - val_accuracy: 0.9375



Transfer Learning

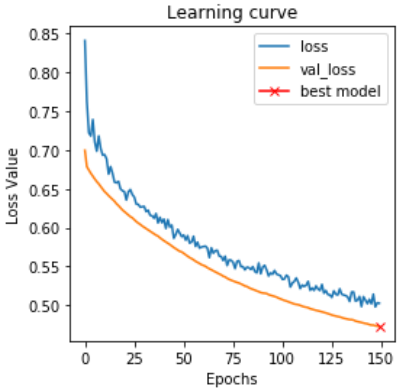
Task8) In this task, you will employ a pretrained VGG16 model which was trained on ImageNet database. By fine tuning this model, you will classify the skin and bone data set again. To do so, as the first step, the pretrained model should be loaded. Training and validation data should passed through the layers and the model output should be saved to be fed, later, to the new layer. Finally, only the new added layers will be trained. Follow the code and complete it as:

Task9) Train the fine-tuned model with skin and bone images. Do you observe some changes in model performance? Describe how transfer learning would be useful for training? How can you make sure that the results are reliable?

For Skin Dataset:

Epoch 150/150

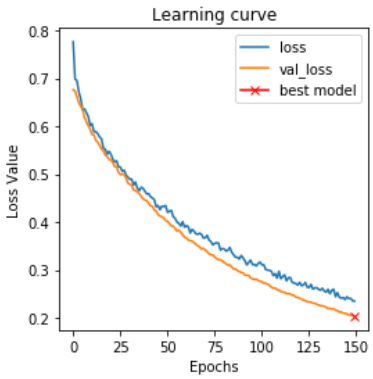
1000/1000 [==============================] - 1s 539us/sample - loss: 0.5024 - accuracy: 0.7900 - val_loss: 0.4724 - val_accuracy: 0.8700



For Bone Dataset:

Epoch 150/150

1112/1112 [==============================] - 1s 491us/sample - loss: 0.2352 - accuracy: 0.9622 - val_loss: 0.2036 - val_accuracy: 1.0000



On comparing the plots achieved in task 7 and task 9 we observe that model performs better when transfer learning is implemented.Transfer learning is basically a method where a model developed for a task is reused as the starting point for a model on a second task if the dataset is small and the last layer of the model is changed with the new dataset.

Visualizing Activation Maps

Task10) Back to bone image classification. Design a VGG16 with 2 neurons at the last year so that the activation function should be set as "softmax" and categorical cross entropy as loss function. (Remember to name the last convolutional layer as name = 'Last_ConvLayer'). Train the model with data augmentation techniques and after the model learned, follow the implementation below and interpret the observed results:

VGG results:

Epoch 80/80

18/17 [==============================] - 8s 455ms/step - loss: 0.2879 - accuracy: 0.8819 - val_loss: 0.2921 - val_accuracy: 0.9375

After trying several times we couldn’t find out how to finish task 10. We were getting the following error:

RuntimeError: Attempting to capture an EagerTensor without building a function.