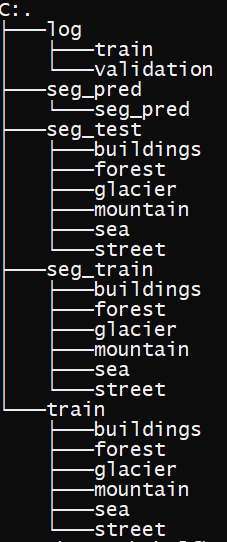
***Image Classification***

Image Classification is one of the methods of classification which has been processed by Deep learning. In the given problem set we have to classify buildings, glacier, forest, mountain, sea and street. Every has 150\*150 pixels in form of RGB. Image Classification is little bit tricky part because process needs complex model but we have to deal with RAM memory as well as processors. Mostly use the GPU. So, the classification directory image is this:

Now, I had made one more train folder so that I try the subset part of seg\_train because the size of images is high. Next data loading part is done in the code. And normalization should be done before inserting data to input that is the reason, we have to delete the input with 255 for ‘RGB’ type image. My most concern is regarding how to create the model. At first, we have the labels the image. Always remember for classification of images try to use Label encoder instead of one hot encoder because order does not important in this process and it take the large amount of memory.

So, to make the model for image classification we mostly use the Convolution Neural network. The reason behind this is CNN tries to compress the data and use less memory as well as time to compute. The working is for multi-channel images, a different kernel is applied to each channel, and the outputs are added together pixel-wise. Checking out the following articles to gain a better understanding of convolutions:

1. [Intuitively understanding Convolutions for Deep Learning] (https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1) by Irhum Shafkat

2. [Convolutions in Depth]

(https://sgugger.github.io/convolution-in-depth.html) by Sylvian Gugger (this article implements convolutions from scratch)

There are certain advantages offered by convolutional layers when working with image data:

Fewer parameters: A small set of parameters (the kernel) is used to calculate outputs of the entire image, so the model has much fewer parameters compared to a fully connected layer.

Sparsity of connections: In each layer, each output element only depends on a small number of input elements, which makes the forward and backward passes more efficient.

Parameter sharing and spatial invariance: The features learned by a kernel in one part of the image can be used to detect similar pattern in a different part of another image.

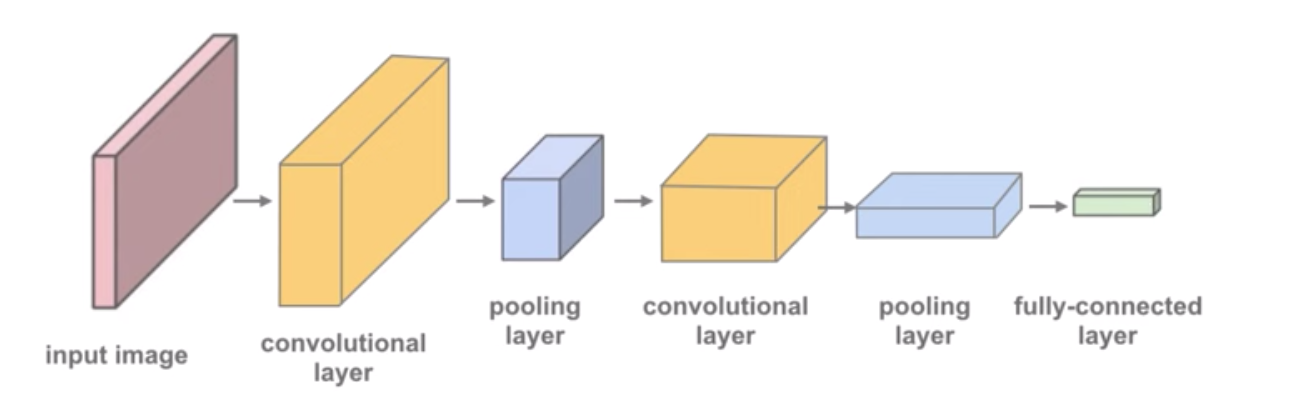
We will also use a [max-pooling] 

**<img src="https://computersciencewiki.org/images/8/8a/MaxpoolSample2.png" style="max-width:400px;">**

(https://computersciencewiki.org/index.php/Max-pooling\_/\_Pooling) layers to progressively decrease the height & width of the output tensors from each convolutional layer.

So our input layer is similar as the numpy array format that has been generated for training. As our pixels shape is (150, 150, 3). This is our input shape for CNN.

The `Conv2d` layer transforms a 3-channel image to a 16-channel \*feature map\*, and the `MaxPool2d` layer halves the height and width. The feature map gets smaller as we add more layers, until we are finally left with a small feature map, which can be flattened into a vector. We can then add some fully connected layers at the end to get vector of size 10 for each image.



So first input after input later in CNN is kernel size and filters. Filters should be more than 3 as this represents for RGB. And kernel weights are glorot uniform or depending on need. Later decide the strides that how much kernel matrix(window) should shift as I have kept the kernel matrix size as (4, 4) I put the strides as 1. I make three layers of filters [64,64,128]. This can be changed as per the accuracy of your model. I used sic CNN layers. One max pooling layer and one Average pooling layer.

I flatten the CNN kernel matrix and then I added one dense layer of 64 or more as per the ram size and depending on the variables created. And for out put layer we use dens layer and input is number of classes as per the number of classes that we have to classify. The activation function “RELU” is mostly used. But for last layer, Softmax is used to differentiate in six classes. If you are doing binary classification better activation function is Sigmoid.

Link for getting knowledge about activation function:

“**https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6**”

For compiling the model we use the three things one is loss function other is optimizer and the last is learning metrics.

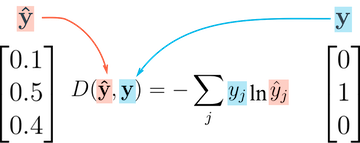
The below reading text is for evaluation of model.

“Accuracy is an excellent way for us (humans) to evaluate the model. However, we can't use it as a loss function for optimizing our model using gradient descent for the following reasons:

1. It's not a differentiable function. ‘tf.model.layers’ and `==` are both non-continuous and non-differentiable operations, so we can't use the accuracy for computing gradients w.r.t the weights and biases.

2. It doesn't take into account the actual probabilities predicted by the model, so it can't provide sufficient feedback for incremental improvements.

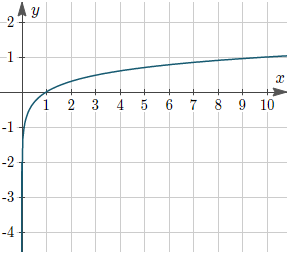
For these reasons, accuracy is often used as an evaluation metric for classification, but not as a loss function. A commonly used loss function for classification problems is the “sparse-cross-entropy”(when you use Label Encoding, if you are using one-hot encoding best method of evaluation is “cross-entropy” ), which has the following formula:



While it looks complicated, it's actually quite simple:

For each output row, pick the predicted probability for the correct label. E.g., if the predicted probabilities for an image are `[0.1, 0.3, 0.2, ...]` and the correct label is `1`, we pick the corresponding element `0.3` and ignore the rest.

Then, take the [logarithm](**https://en.wikipedia.org/wiki/Logarithm)** of the picked probability. If the probability is high, i.e., close to 1, then its logarithm is a very small negative value, close to 0. And if the probability is low (close to 0), then the logarithm is a very large negative value. We also multiply the result by -1, which results is a large positive value of the loss for poor predictions.



Finally, take the average of the cross entropy across all the output rows to get the overall loss for a batch of data.

Unlike accuracy, cross-entropy is a continuous and differentiable function. It also provides useful feedback for incremental improvements in the model (a slightly higher probability for the correct label leads to a lower loss). These two factors make cross-entropy a better choice for the loss function.

For optimizers best is “ADAM”. But for more reading about optimizers you can refer this:

“**https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6**”