MODEL USED :Densenet (custom built)

I used Densenet for this task.Similar to Residual Networks that add connection from the preceding layer, Dense Nets add connections to all the preceding layers, to produce a Dense Block. The problem that Residual Nets addressed was the vanishing gradient problem that was due to many layers of the network. This helped build bigger and more efficient networks and reduce the error rate on classification tasks on ImageNet. So the idea of Densely Connected Networks, is that every layer is connected to all its previous layers and its succeeding ones, thus forming a Dense Block.

About "Densenet.ipynb" file:

● Class **Dense_Block_Transposed**:

This class contains all the functions needed to implement the dense block.

Here 5 layers of densenet are implemented.

2.) Each time there is a Transposedconvolution operation of the previous layer, it is followed by concatenation of the tensors. This is allowed as the channel dimensions, height and width of the input stay the same after convolution with a kernel size 3×3 and padding 1.

3.)I am going to implement a Dense Network with 3 Dense Blocks and 3 Transition Layers. In the Transition Layers, I make use of the average pooling operation which reduces the size of its input.

4.)From the experiment above transpose convolutions seem to learn meaningful results and have comparable accuracy with regular convolutions, even for classification tasks.But for transpose convolution is used more for upsampling purposes in GANs.I just tried it...ML is full of experimentation...

5. Also I used leakyReLU as the activation function for this experimentation.

- **def get_train_valid_loader:**

If using CUDA, num_workers should be set to 1 and pin_memory to True.

Params:

- batch_size: how many samples per batch to load
- random_seed: fix seed for reproducibility.
- valid_size: percentage split of the training set used for the validation set.
- shuffle: whether to shuffle the train/validation indices.
- num_workers: number of subprocesses to use when loading the dataset.
- pin_memory: whether to copy tensors into CUDA pinned memory. Set it to “true” if using GPU.

Returns

- **train_loader:** training set iterator.
- **valid_loader:** validation set iterator.

- **def get_test_loader**: This is also same as get_train_loader....it returns test_loader.

USED PARAMS:

- batch_size = 64
- momentum = 0.95
- learning_rate = 0.001
- nr_classes = 10
- nr_epochs = 100
- Used SGD as the optimizer and CrossEntropy as the loss function

- **def log_training_results:** Takes in the trainer object and logs the accuracy and loss after every epoch....it is built using the pytorch-ignite package.
- **def log_validation_results:** Takes in the trainer object and logs the accuracy and loss after every epoch....it is built using the pytorch-ignite package
- **def trainer.run(train_loader,max_epochs):** takes in the train_loader and the max_epochs as the params and runs the training process.
- **def apply_test_transforms(input_img):** applies transforms for the input test image.........
- **densenet.eval( ):** For evaluation purpose
- **def predict(densenet,filepath,show_img=False,url=False):**gives ouot predcition for the image passed.

I also tried inferencing with python for my learning purpose