SOI 2024: Auburn Waves
Team Name: The Crue

Machine Learning Model to Identify Martian Terrains

Data Analysis and Processing

The training set provided contains **6201** images of the Mars Terrain ranging across **8 different classes**. The images have a shape of **227 X 227** and have only one channel **(black and white).**
The csv file provided maps the images to their class names. We mapped these class names to integers ranging from 0-7 as follows:

Class	Label
Bright Dune	0
Dark Dune	1
Spider	2
Impact Ejecta	3
Slope Streak	4
Swiss Cheese	5
Crater	6
Other	7

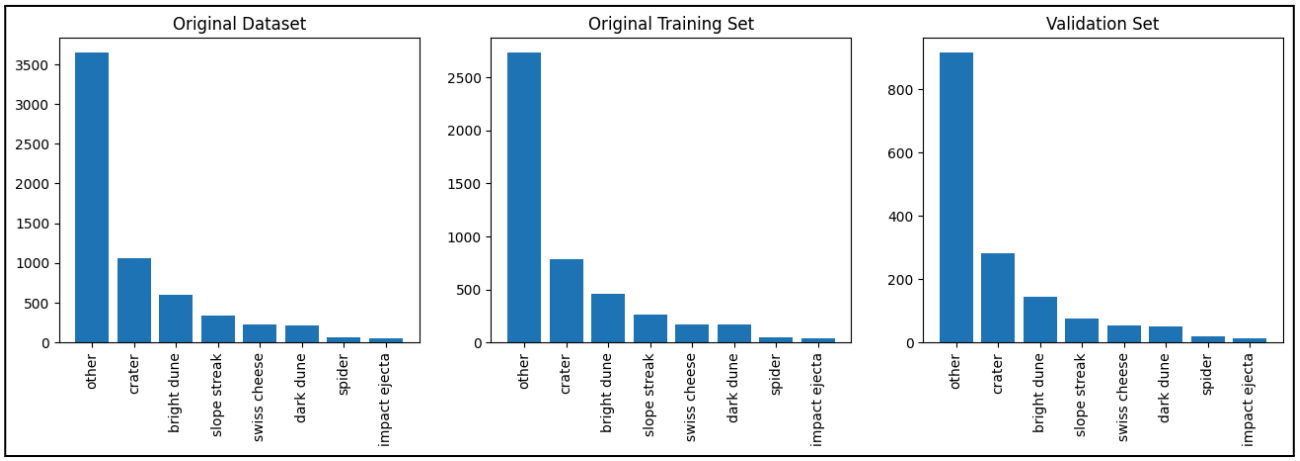
Reading the CSV:

Libraries Used: Pandas
The CSV file was imported as a **Pandas DataFrame Object**. It contained two columns: “File Name” and “Class”. These two columns were extracted as Python Lists. The contents of the File Name list were concatenated with the name of the dataset folder, to get a list of paths that can be used to directly access the images. The classes in the Class list were replaced with their corresponding integer

labels. After which we moved on to splitting the dataset into a **Training Set** and **Validation Set.**

Splitting the Set, Visualising, and Upsampling Data:

Libraries used: Pandas, Scikit Learn
The lists of filenames and their labels were split using the train_test_split function in sklearn, to obtain the Training and Validation sets.
The distribution of the training and validation sets obtained is as follows:

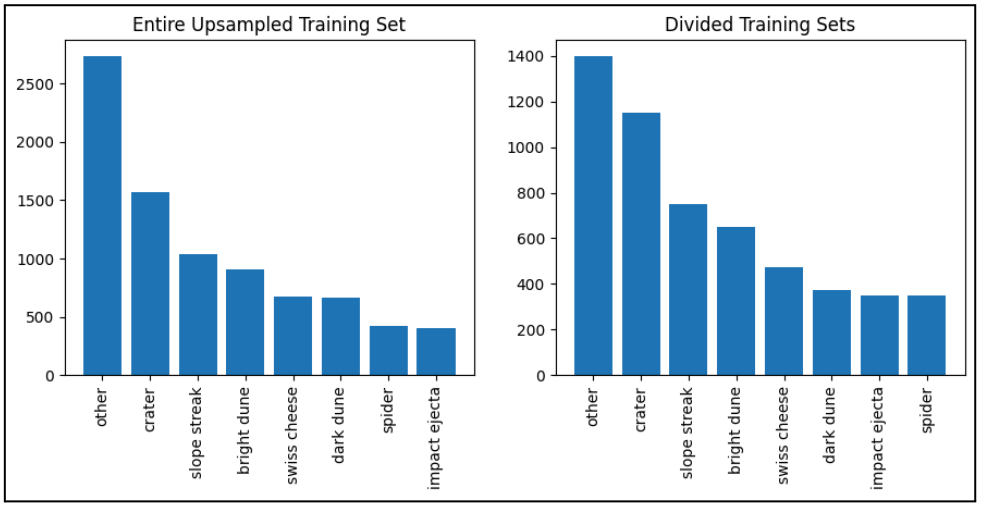


From these bar graphs it is clearly visible that the dataset is **skewed.** It contains a large number of images from the **“other”** class, which will cause the model to be biased towards classifying data as “other”, and not get enough training data from the rest of the classes.

To ensure uniform data distribution, we **upsampled** the lower-frequency classes in the training set,to obtain a training set having **8.4k** images. This operation was limited to the training set to avoid affecting the validation set. The dataset was then split into two training sets, each with **5500 images** and a considerable overlap, to further normalise class frequencies.

We decided not to change the validation dataset in order to maintain the expected distribution of the unlabelled test_dataset, and get similar outlooks from the validation set.

The obtained distributions are:



Label	Entire Dataset	Validation Set	Original Training Set	Upsampled Training Set	Final Training Sets
0	597	143	454	908	650
1	216	50	166	664	375
2	66	19	47	423	350
3	51	14	37	407	350
4	335	75	260	1040	750
5	223	54	169	676	475
6	1062	280	782	1564	1150
7	3651	916	2735	2735	1400
Total	**6201**	**1551**	**4650**	**8417**	**5500**

We now have two Training Dataframes and one Validation Dataframe. The 2 training Dataframes are created so that we have a small data set of 5.5k images, with a somewhat uniform distribution of classes.

Creating the dataset and Dataloaders:

Libraries used: PyTorch and TorchVision
The Dataframes containing the image paths and their labels are passed into a PyTorch dataset class to import the images and build the image dataset.

The following transforms are applied to the images:

1. **Resize:** All images are resized to 200x200 pixels

2. **Random Flips:** Images are flipped randomly, vertically and/or horizontally every time they are retrieved with a probability of 60% for each flip. This helps to augment the limited data
3. The images are then converted to **Pytorch Tensors**
4. The **contrast** of the images is **increased by 80%** in order to highlight the lines in the images more.

The Model

Model Architecture - CNN:

We have used a Convolutional Neural Network (CNN) for the purpose of image classification. Our CNN model consists of **5 convolutional layers**, each followed by **ReLU activation** and **batch normalisation.** We have included both **max pooling** and **average pooling** layers in our model to progressively reduce the spatial dimensions of the data. Finally, the extracted features are passed through a **fully connected layer** that outputs predictions for **8 classes.**
The details of each component of our model are as follows:

Convolutional Layers:

1. **Conv Layer 1**
    - Input Channels: 1 (Grayscale images)
    - Output Channels: 32
2. **Conv Layer 2**
    - Input Channels: 32
    - Output Channels: 64
3. **Conv Layer 3**
    - Input Channels: 64
    - Output Channels: 512
4. **Conv Layer 4**
    - Input Channels: 512
    - Output Channels: 512
5. **Conv Layer 5**

- Input Channels: 512
- Output Channels: 256

Each convolution layer has a kernel size of 3 * 3, and a padding of one, in order to maintain the spatial dimension of the input data. The activations for all these layers are ReLU functions. In addition, each layer contains a batch normalisation sub-layer, in order to stabilise and speed up the training.

Pooling Layers:

With a kernel size of 2 * 2 , we have utilised a combination of the following pooling layers in our model:

1. **Max Pooling**
2. **Average Pooling**

Classifier Layer:

1. **Flatten Layer:** Converts the multi-dimensional tensor (256 output channels) from the final convolutional layer to a 1D tensor for the fully connected layers.
2. **Fully Connected Layer 1:** Maps the flattened vector (256 * 3 * 3) to a vector of size 128, reducing dimensionality, followed by a ReLU activation
3. **Dropout Layer**: Randomly sets 10% of the connections to the next layer to zero during training to prevent overfitting.
4. **Fully Connected Layer 2:** Consolidates the 128 neuron activation to 8 final output logits, that will later be softmaxed to get the final output labels.

Forward Pass - Data flow through the network:

1. **Conv Layer 1**: Input tensor with 1 channel passes through Conv Layer 1, ReLU activation, and batch normalisation, resulting in 32 output channels.
2. **Conv Layer 2**: The output from Conv Layer 1 goes through Conv Layer 2, yielding 64 output channels, followed by max pooling.

3. **Conv Layer 3**: The output from Conv Layer 2 passes through Conv Layer 3, resulting in 512 output channels, followed by average pooling.
4. **Conv Layer 4**: The output from Conv Layer 3 goes through Conv Layer 4 iteratively, followed by average pooling and multiple max pooling operations, leading to 512 output channels.
5. **Conv Layer 5**: The output from Conv Layer 4 passes through Conv Layer 5, resulting in 256 output channels.
6. **Flattening**: The output from Conv Layer 5 is flattened into a 1D tensor.
7. **Fully Connected Layer 1**: The flattened output passes through the first fully connected layer, reducing the feature size to 128, followed by ReLU activation and dropout for regularisation.
8. **Fully Connected Layer 2**: The resulting output passes through the second fully connected layer, mapping the features to 8 output classes for classification.

Training and Results

Model Training:

- **Loss Function**: We have used cross-entropy loss to measure the error between predicted and actual class labels.
- **Optimizer:** Adam optimizer with an initial learning rate of 0.0002 is used to update model parameters based on computed gradients.
- **Epoch Handling:** In order to equally train our model on both the training datasets we created, the training loader alternated between the two datasets every epoch.

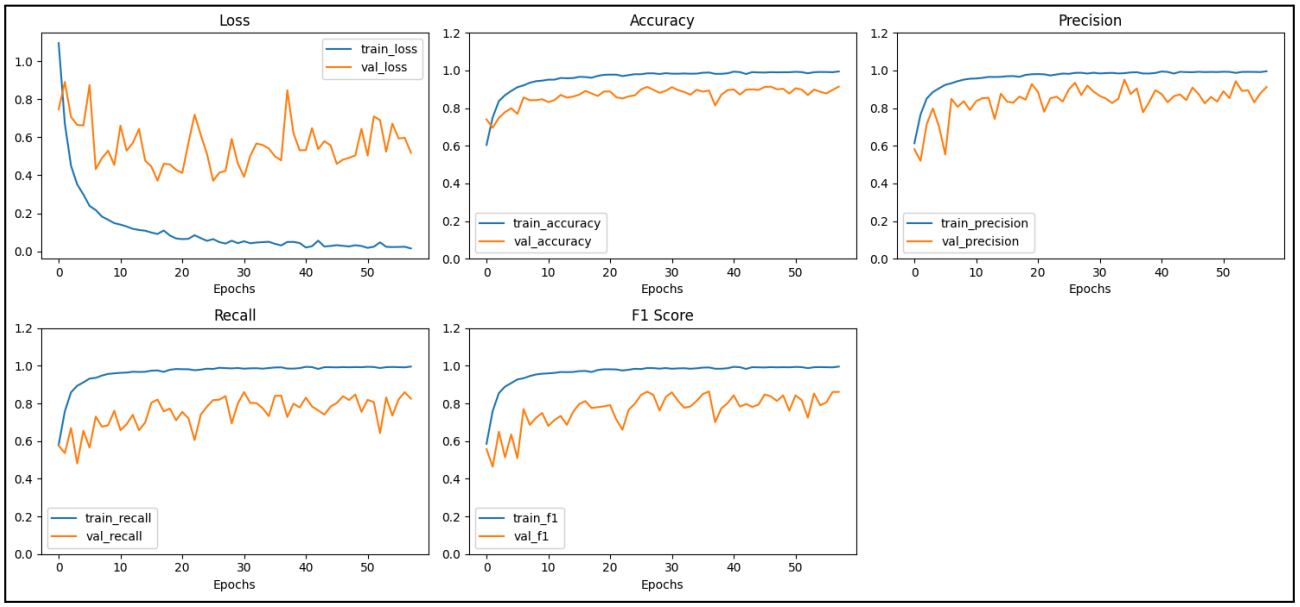
- **Model Saving:** After each epoch, we saved the training results to a file. This allowed us to pause and resume training if needed.

Training Results:

We trained the model for more than 55 epochs and obtained the following final metrics:

Train Loss	0.0152	**Validation Loss**	0.519
Train Accuracy	**0.9953**	**Val Accuracy**	**0.9143**
Train Precision	0.9960	**Val Precision**	0.9123
Train Recall	0.9964	**Val Recall**	0.8256
Train F1 Score	0.9962	**Val F1 Score**	0.8618

The following are the plots of the metrics over the training duration:

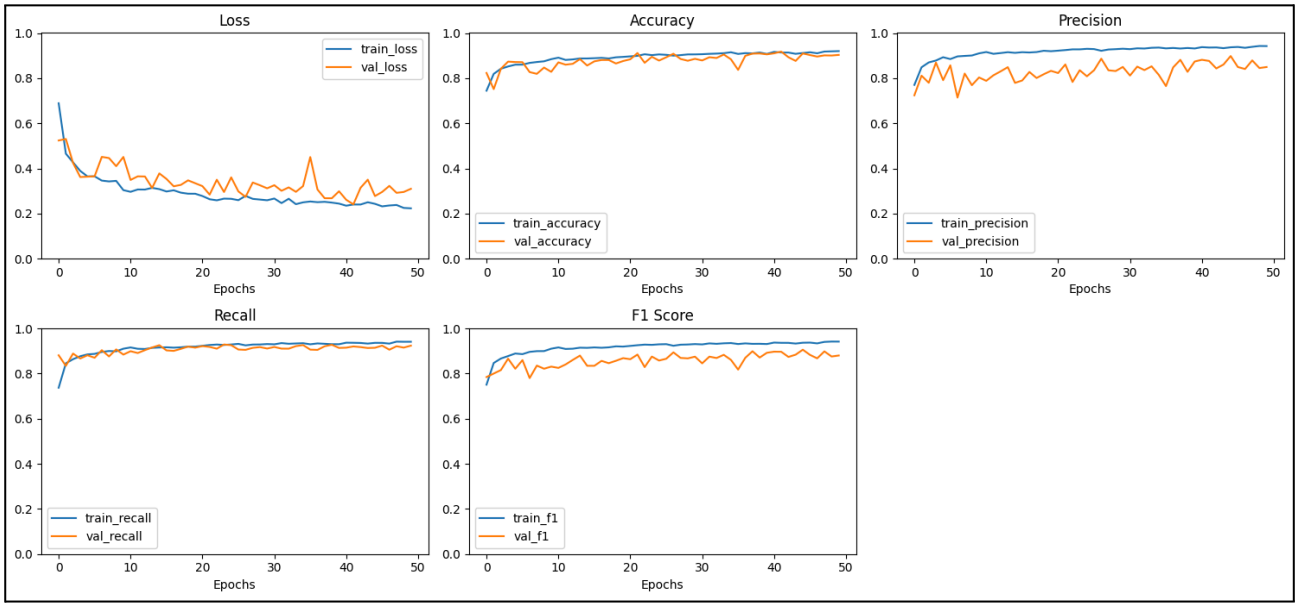


With a validation accuracy of 91% and a good validation F1 score, we went ahead to use this model for inference.

Alternate Approaches:

This model wasn't the first model we tried. Several models were trained and their accuracies and other metrics were compared in order to choose the best one.
We also tried some Transfer Learning techniques, and fine tuned pretrained models. We worked on pretrained cutting edge models,

namely **VGG-16** and **ResNet50**. Only the classification layers of these models were trained on our dataset. Although these models have much more parameters and layers than our models, the results were not very different from our own model.
Illustrated below are the training curves of the VGG-16 pretrained models, whose last fully connected layer was replaced with our custom layer.



The best accuracy that this model could reach was **90%**. Seeing the similar results, **we decided to go with our own custom model** rather than the pretrained one, since it was more original and was much smaller in size and number of parameters than the VGG-16 model.