Name: Soumitra Koustubh Manavi Student ID: 22220805 MSc Artificial Intelligence (1MAI1)
Name: Jash Prakash Rana Student ID: 22222806 MSc Artificial Intelligence (1MAI1)
Name: Pratik Anil Rahood Student ID: 22210643 MSc Artificial Intelligence (1MAI1)

Research Topics in AI

Assignment 2

Group No. 16

Contribution:

Soumitra Koustubh Manavi (22220805):

- Proposed Layer Idea using Subclassing.
- Loading, Resizing, & Preprocessing of Dataset using OpenCV.
- Analysis & implementation of a 2D CNN network of the same depth.

Pratik Anil Rahood (22210643):

- Proposed layer implementation.
- TensorBoard configuration and development along with analysis.
- 3D Data code changes research.

Jash Prakash Rana (22222806):

- Proposed layer debugging.
- Hyperparameter fine-tuning & increase in depth of the proposed model network.
- 1D Data code changes research.

Teamwork:

- Training the Proposed & 2D CNN model with baseline hyperparameters & discussion of modifications for compatible resource allocation
- Simulations and visualisation generation and analysis.
- Research different optimizers and appropriate hyperparameters and loss metrics.
- Report generation.

Aim: In this assignment, the goal is to build a network like a Convolutional Neural Network (CNN). The difference is that instead of a Convolution layer in a CNN, you must introduce a new layer where a new partially shared kernel calculates the weighted sum each time.

Theory/Working:

Figure 1 image shows the proposed model with a weight matrix equal to the input image size. The weight kernel updates the weighted value for all neurons. Whereas in the CNN layer, a weighted kernel like 3x3 slides over the whole input image to generate an output feature map.

To build the proposed model, the difference is in the weight kernel and weight matrix. A weight matrix from which each time a locally position-oriented kernel is used for calculating the weighted sum operation for the proposed model. A weight matrix from which each time a locally position-oriented kernel is used for calculating the weighted sum operation for the proposed model.

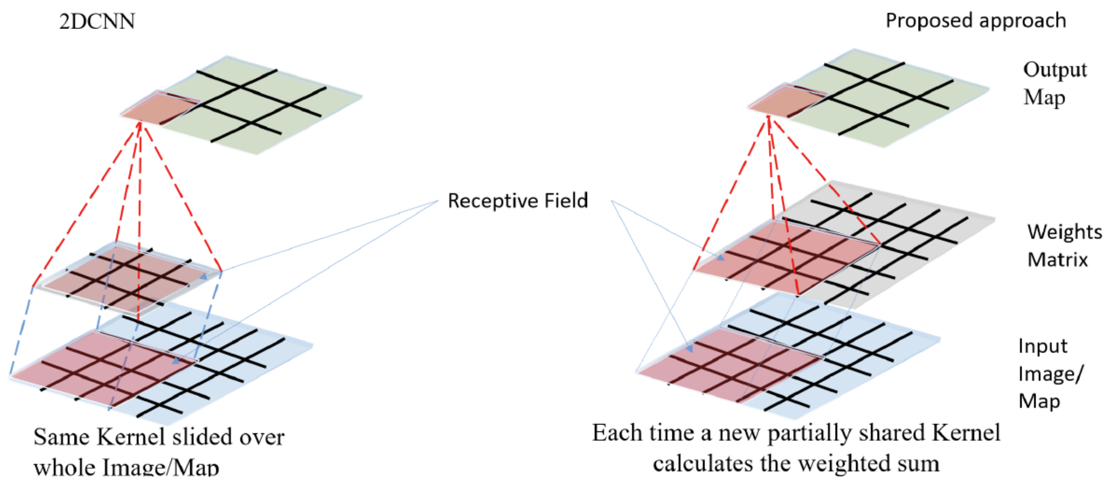

2DCNN
Proposed approach
Output Map
Receptive Field
Weights Matrix
Input Image/ Map
Same Kernel slided over whole Image/Map
Each time a new partially shared Kernel calculates the weighted sum

Fig 1: Difference between CNN and Proposed Layer
Source: Dr. Ihsan Ullah's PPT "CNN vs new Network approach"

In the 2D CNN model, the convolution layer of 3x3,5x5 or 7x7 weight kernel is used as a weighted kernel that slides over the entire input image and generates an output feature map. The weight kernel is randomly initialized. Multiple output feature maps can be generated using more than one weight kernel.

The proposed approach introduces a randomly initialized weight matrix that is of equal size to the input image. The receptive field from each matrix is selected as a weighted kernel that will do a dot product with the same position receptive field in the input image to calculate a weighted output value for an output neuron. This process is repeated for all the neurons on the kernel in the output map. The backpropagation algorithm is used through the same receptive field to backpropagate the error calculated. The neuron calculates the gradient using the same weights in the respective receptive fields. This helps in computing the gradient of the loss function with respect to the inputs.

2D Convolutional Neural Network Implementation :

- Loading the dataset using OpenCV and resizing the images using OpenCV.
- Splitting the data for training, validation, and testing.
- Rescaling the image data by 1/255. This step is a normalization technique for scaling images to a range of [0,1].[2]
- Use of **to_categorial** by Keras that converts the labels or class vectors to binary value matrix.
- Model Selection for the creation of 2D CNN consists of a Sequential Model with an input image shape of 320x320. The convolutional layers with 16 filters each of size 3x3 and ReLU activation function are added followed by the max-pooling layer with a pool size of 2x2.

- The Conv2D layer creates a convolutional kernel that produces a tensor of outputs convolving with the input layer.
- The max pooling layer down samples the input along its dimensions of height and width for each channel of the input. The pooling window is shifted by strides along each dimension.
- Another convolutional layer with 12 filters each of size 3x3 and ReLU activation function is added followed by a max pooling layer of pool size 2x2.
- Another convolutional layer with 8 filters each of size 3x3 and ReLU activation function is added followed by a max pooling layer of pool size 2x2.
- Flatten is used to flatten the input that converts 2D feature vectors to 1D feature vectors.
- The Dense layer is used that say returns a 2-D tensor with the shape of batch size and units along with activation function as SoftMax for two possible labels alpaca or not alpaca.

Advancements to the model:

Early Stopping: used to stop training when monitored metrics like loss stops decreasing. The patience states the no. of epochs observed with no improvements. The training stops after that no. of epochs. If set to true, the model restores best weights from each epoch for the monitored quantity.

Compile Strategy: The model calculates the cross-entropy loss between labels and prediction by compiling the model setting the loss of Categorical cross-entropy. The optimizer used is Adam.

Implementation of Model based on Proposed Layer using Keras:

Proposed layer:

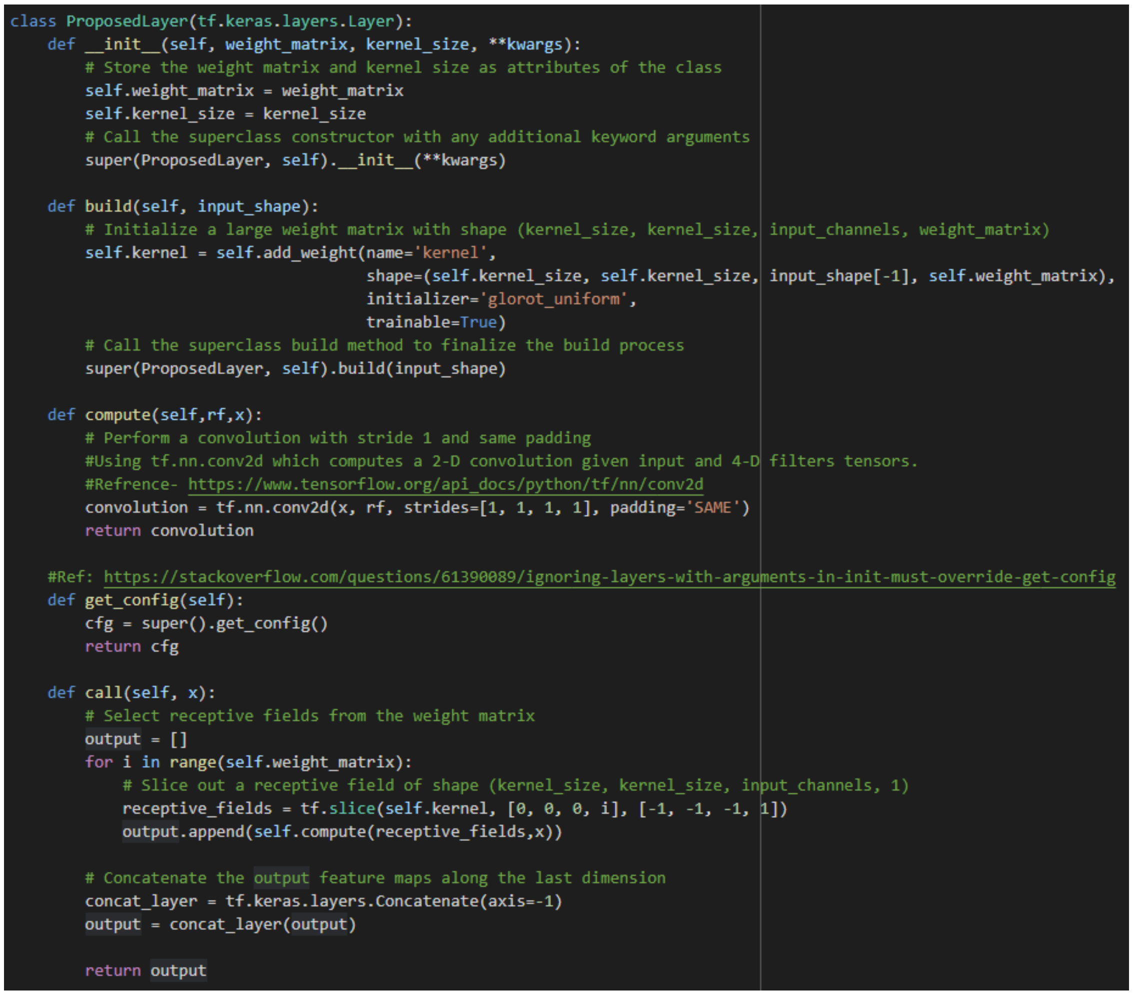
We have loaded the dataset using OpenCV and resized the images using OpenCV. Splitting the data for training, validation, and testing is done by using train-test-split, and then Rescaling of the image data is done by dividing by 255. This step is a normalization technique for scaling images to a range of [0,1].

The proposed layer implementation is done with the help of the Keras subclassing guide [4]. In the new model, the proposed layer implementation consists of a weight matrix that is of equal size to the input image and the kernel size of (3,3). Each output channel of the weight kernel is selected as a matrix in the receptive field. Then for each receptive field, we slice the weight matrix to get a receptive field of the input shape and append it to a list. This gives us a list of n receptive fields.

Next, we compute the dot product of each receptive field with the input vector using "**tf.nn.conv2d**" [5]. For each receptive field, we compute a 2D convolution between the input image x and the receptive field using "**tf.nn.conv2d**" [5]. The stride [1, 1, 1, 1] is set, which means that we move the receptive field by 1 pixel in both the horizontal and vertical directions. The padding parameter is set to **'SAME'**, which means that we add padding to the input image so that the output has the same shape as the input. We then append each output vector to the output list. Finally, we concatenate the output vector along the channel dimension using "**tf.concat**." This gives us a final output tensor that has n channels corresponding to the n receptive fields.

When the model is trained using the compile and fit techniques of the Sequential model, backpropagation occurs automatically in this scenario (Sequential model). During training, the gradients of the loss with respect to the weights and biases are produced using TensorFlow's automated differentiation, which is based on the difference between the predicted and actual labels. The optimizer then modifies the parameters based on these gradients to reduce loss.

This procedure is continued over several epochs until the loss converges or stops dramatically reducing. The **"ProposedLayer"** class automatically handles computing gradients during backpropagation because it inherits from **"tf.keras.layers.Layer."** For example, TensorFlow computes the gradients of the loss with respect to each layer's inputs using the chain rule of calculus and then propagates those gradients backward through the layers to compute the gradients with respect to each layer's parameters.



```
class ProposedLayer(tf.keras.layers.Layer):
    def __init__(self, weight_matrix, kernel_size, **kwargs):
        # Store the weight matrix and kernel size as attributes of the class
        self.weight_matrix = weight_matrix
        self.kernel_size = kernel_size
        # Call the superclass constructor with any additional keyword arguments
        super(ProposedLayer, self).__init__(**kwargs)

    def build(self, input_shape):
        # Initialize a large weight matrix with shape (kernel_size, kernel_size, input_channels, weight_matrix)
        self.kernel = self.add_weight(name='kernel',
                                      shape=(self.kernel_size, self.kernel_size, input_shape[-1], self.weight_matrix),
                                      initializer='glorot_uniform',
                                      trainable=True)
        # Call the superclass build method to finalize the build process
        super(ProposedLayer, self).build(input_shape)

    def compute(self,rf,x):
        # Perform a convolution with stride 1 and same padding
        #Using tf.nn.conv2d which computes a 2-D convolution given input and 4-D filters tensors.
        #Refrence- https://www.tensorflow.org/api_docs/python/tf/nn/conv2d
        convolution = tf.nn.conv2d(x, rf, strides=[1, 1, 1, 1], padding='SAME')
        return convolution

    #Ref: https://stackoverflow.com/questions/61390089/ignoring-layers-with-arguments-in-init-must-override-get-config
    def get_config(self):
        cfg = super().get_config()
        return cfg

    def call(self, x):
        # Select receptive fields from the weight matrix
        output = []
        for i in range(self.weight_matrix):
            # Slice out a receptive field of shape (kernel_size, kernel_size, input_channels, 1)
            receptive_fields = tf.slice(self.kernel, [0, 0, 0, i], [-1, -1, -1, 1])
            output.append(self.compute(receptive_fields,x))

        # Concatenate the output feature maps along the last dimension
        concat_layer = tf.keras.layers.Concatenate(axis=-1)
        output = concat_layer(output)

        return output
```

Fig 2. Proposed layer Implementation.
Source: Keras new layers and Subclassing [4]

<u>Final Proposed Model:</u>

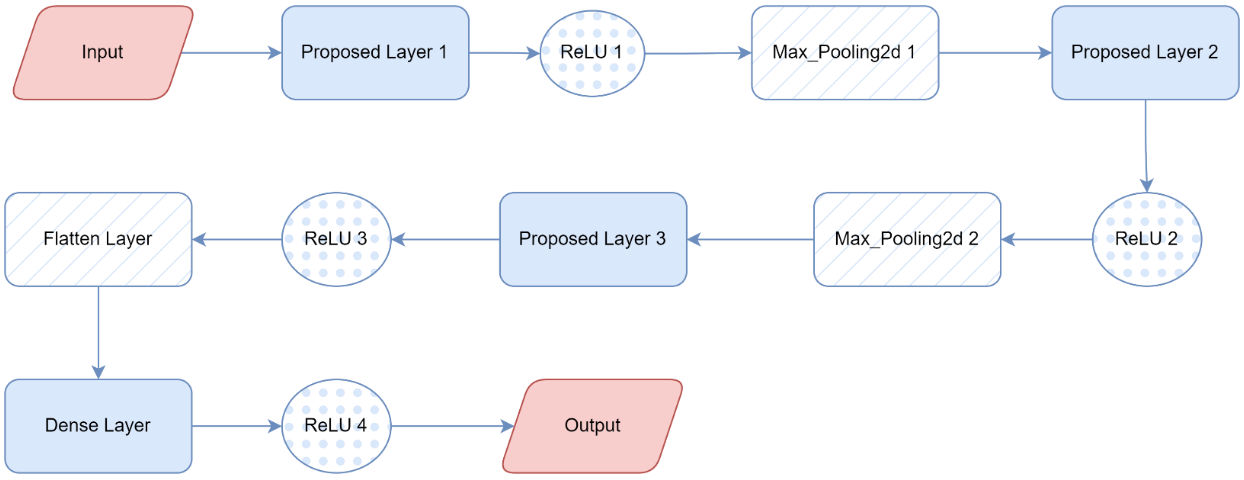
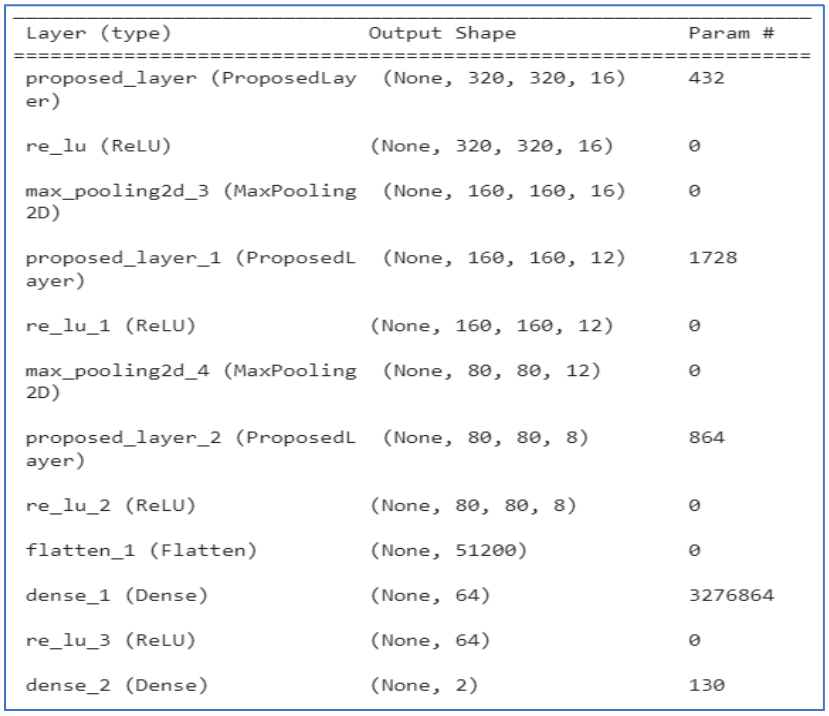


Fig 3(a). Flow Chart of Proposed Model

1) At ProposedLayer1 we have used 16 weight matrices to get 16 feature maps while using a 3x3 receptive field and stride of 1.
2) At ProposedLayer2 we have used 12 weight matrices to get 12 feature maps while using a 3x3 receptive field and stride of 1.

3) At ProposedLayer3 we have used 8 weight matrices to get 8 feature maps while using a 3x3 receptive field and stride of 1.
4) At ActivationLayer1, ActivationLayer2, and ActivationLayer3 we have used ReLU activation.
5) At PoolingLayer1 and PoolingLayer2, we have used 2x2 receptive field.



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 proposed_layer (ProposedLay  (None, 320, 320, 16)     432
 er)

 re_lu (ReLU)                (None, 320, 320, 16)      0

 max_pooling2d_3 (MaxPooling  (None, 160, 160, 16)     0
 2D)

 proposed_layer_1 (ProposedL  (None, 160, 160, 12)     1728
 ayer)

 re_lu_1 (ReLU)              (None, 160, 160, 12)      0

 max_pooling2d_4 (MaxPooling  (None, 80, 80, 12)       0
 2D)

 proposed_layer_2 (ProposedL  (None, 80, 80, 8)        864
 ayer)

 re_lu_2 (ReLU)              (None, 80, 80, 8)         0

 flatten_1 (Flatten)         (None, 51200)             0

 dense_1 (Dense)             (None, 64)                3276864

 re_lu_3 (ReLU)              (None, 64)                0

 dense_2 (Dense)             (None, 2)                 130
```

Fig 3(b). Proposed Model Summary

Advancements to the model:

Early Stopping: used to stop training when monitored metrics like loss stops decreasing. The patience states the no. of epochs observed with no improvements. The training stops after that no. of epochs. If set to true, the model restores the best weights from each epoch for the monitored quantity.

Compile Strategy: The model calculates the cross-entropy loss between labels and prediction by compiling the model setting the loss of Categorical cross-entropy. The optimizer used is Adam.

Successful implementation of the layer and complete model with a figure in the Tensor Board:

The Google TensorFlow team created TensorBoard as a machine learning visualization tool. With interactive visualizations, graphs, and dashboards, users may explore, visualize, and understand the behaviour of machine learning models [6].

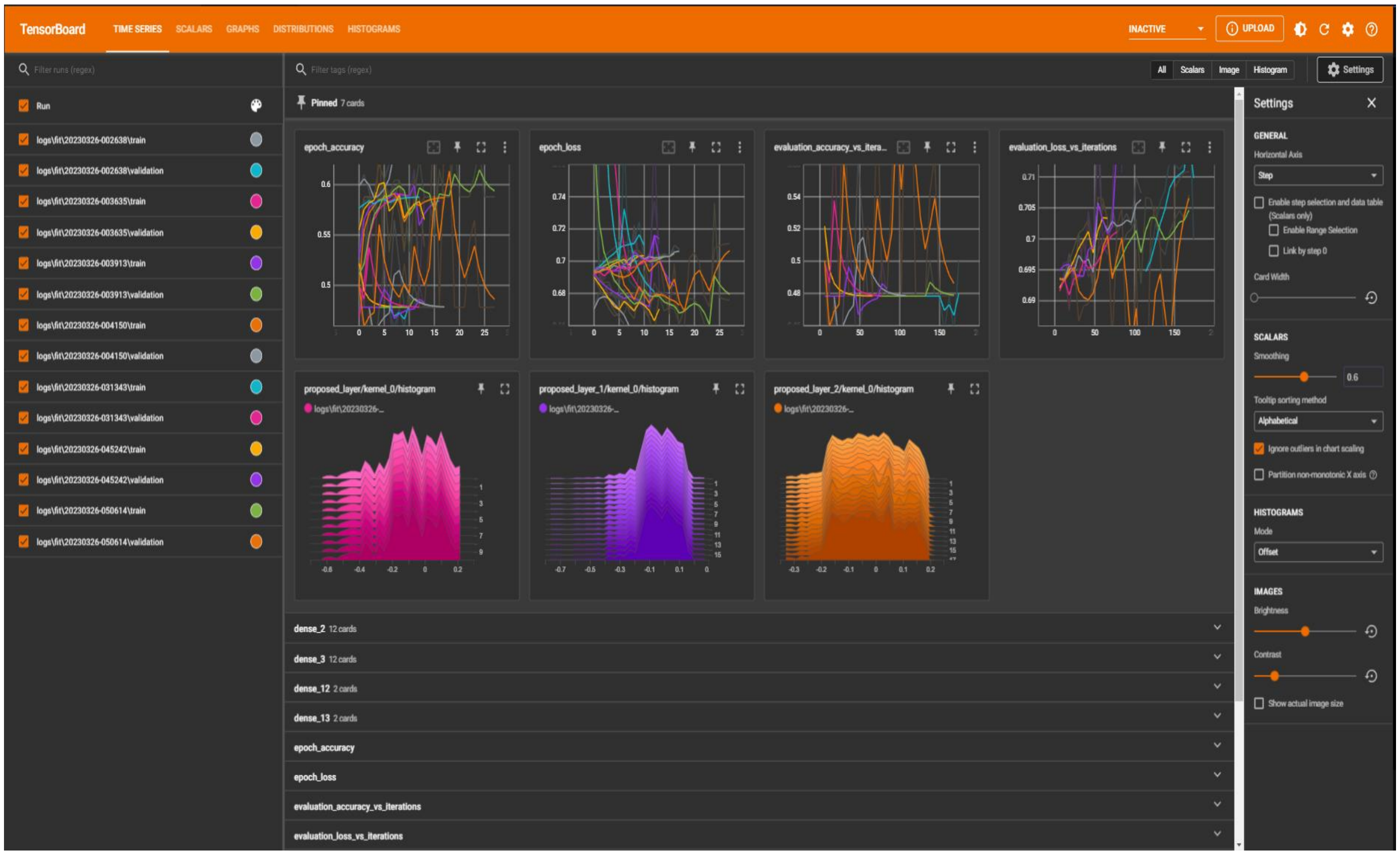
TensorBoard may show a variety of data regarding a model's performance during training, including histograms of weights and biases, scalar numbers like loss and accuracy, and representations of the model's internal structure. It can also be used to compare several models, monitor development over time, and troubleshoot performance problems with the model [6].

TensorBoard reads training-related log files, including the data needed to produce the visuals. Although it can be used with other machine learning libraries, the tool is commonly integrated into the TensorFlow framework and is accessible through a web browser [6].

In conclusion, TensorBoard is a potent tool for understanding the behaviour of machine learning models and can assist researchers and developers in improving the performance of their models [6].

So, we use Tensor Board, a visualization toolkit for machine learning experimentation and visualization. Tensor Board allows tracking and visualizing metrics such as loss and accuracy, visualizing the model graph, histograms, displaying images, etc.

Figure 3 shows the Tensor Board data which is called via the call back method [3]. The Tensor Board is called during the model.fit when the training of the said model is done using the callback parameter. It records the data by creating a folder and creating data suitable which can be usable at any time. The figure shows accuracy, loss, evaluation by epochs, and histograms showing the proposed layer's kernel values found during training.

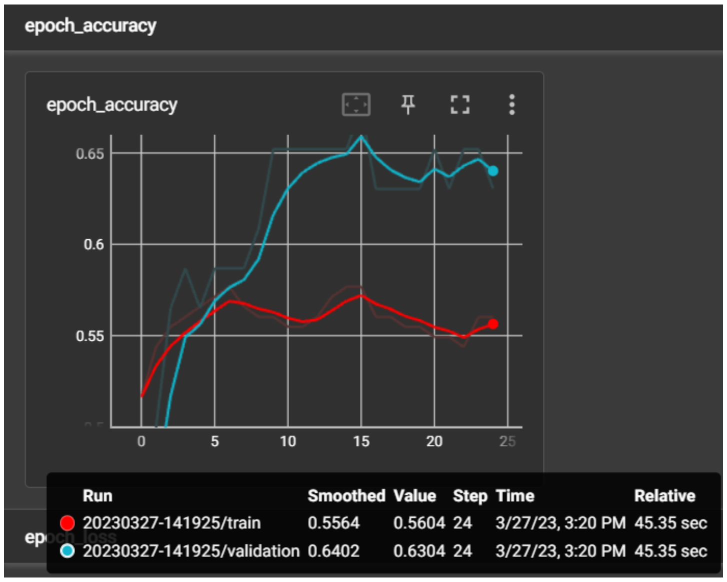


Run
logs\fit\20230326-002638\train
logs\fit\20230326-002638\validation
logs\fit\20230326-003635\train
logs\fit\20230326-003635\validation
logs\fit\20230326-003913\train
logs\fit\20230326-003913\validation
logs\fit\20230326-004150\train
logs\fit\20230326-004150\validation
logs\fit\20230326-031343\train
logs\fit\20230326-031343\validation
logs\fit\20230326-045242\train
logs\fit\20230326-045242\validation
logs\fit\20230326-050614\train
logs\fit\20230326-050614\validation

Settings
GENERAL
Horizontal Axis: Step
SCALARS
Smoothing: 0.6
Tooltip sorting method: Alphabetical
[✓] Ignore outliers in chart scaling
HISTOGRAMS
Mode: Offset

Fig 4. Tensor Board showing complete model training vs validation with Proposed Layers

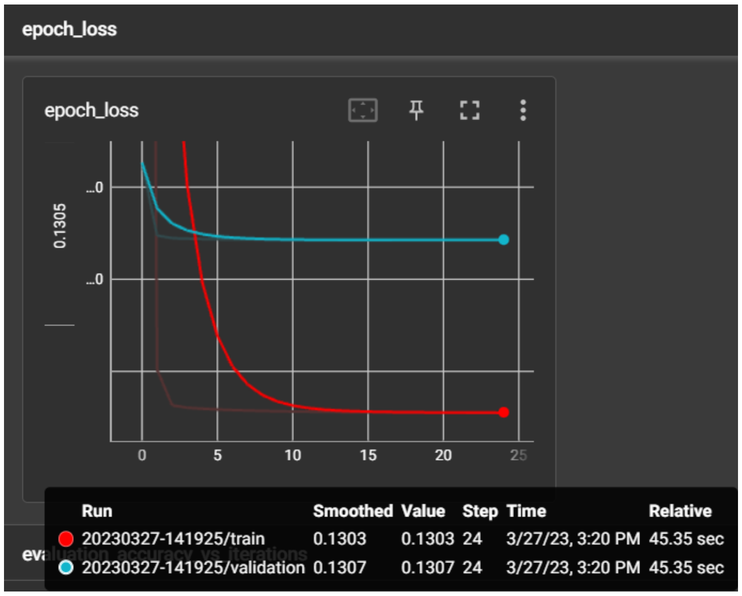
The fig.4(a) shows epoch accuracy of the proposed model on the train and validation data. As can be seen, the accuracy increased for both the train and validation data with each epoch, with the best accuracy coming in at about 66 percent and 57 percent for the train and validation data, respectively.



Run	Smoothed	Value	Step	Time	Relative
🔴 20230327-141925/train	0.5564	0.5604	24	3/27/23, 3:20 PM	45.35 sec
🔵 20230327-141925/validation	0.6402	0.6304	24	3/27/23, 3:20 PM	45.35 sec

Fig 4(a). Epoch Accuracy for one set of hyperparameters

The fig. 4 (b) shows epoch loss of the proposed model for both the train and validation data. As can be seen, the loss decreased with each passing epoch for both the train and validation data, and convergence was reached for both sets of data at epochs 8 and 15, respectively.



Run	Smoothed	Value	Step	Time	Relative
● 20230327-141925/train	0.1303	0.1303	24	3/27/23, 3:20 PM	45.35 sec
● 20230327-141925/validation	0.1307	0.1307	24	3/27/23, 3:20 PM	45.35 sec

Fig 4(b). Epoch Loss for one set of hyperparameters

Load and distribute the data for training, validation, and testing

```
def load_images(file):
    input_path = './dataset/' + file
    filenames = [input_path + filename for filename in os.listdir(input_path)]
    images = [cv2.imread(filename) for filename in filenames]
    return images
```

```
def resize_images(images):
    resized_image_list=[]
    labels=[]
    for i in images:
        resize_image = cv2.resize(i, dsize=(320, 320))
        resized_image_list.append(resize_image)
    return resized_image_list
```

Fig 5. Data Loading & Resizing using OS and CV2 libraries.

- The loading of data with OpenCV provided the input path. The images are resized to 320x320 using OpenCV.

Train-Test split:

```
from sklearn.model_selection import train_test_split
from keras.utils import to_categorical
#reference : rescaling :
# https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html

X_train, X_test, y_train, y_test = train_test_split(data/255,
                                                    to_categorical(labels),
                                                    test_size=0.3,
                                                    random_state=101)
```

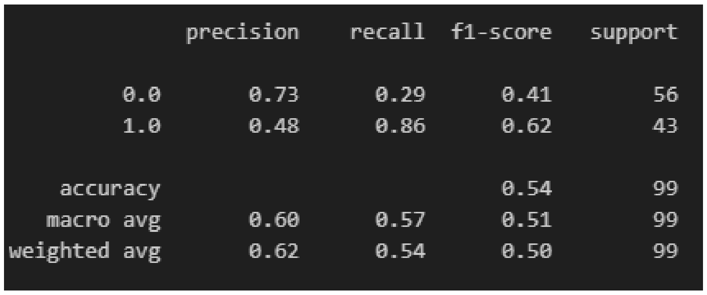
Fig 6. Train Test Split and Normalization

- The data is split into train test data using the sklearn train test split. The validation set is further split with size 0.2 of the training data in the model fit of keras.

<u>Training of the model with the dataset provided in this assignment folder and reporting the training and testing performance against hyperparameters (e.g., learning rate = 0.1, batch size = 32, epochs = 500,) using stochastic gradient descent.</u>

Training the model was done with 70% of the data for the training set and 30% for testing. We also took a 20% validation set from the training set itself while calling the fit method on the proposed model. As per figure 2, the output stage contained Sigmoid activation which gave an output between 0 and 1 for the test set.

The basic model running with the Proposed Layer has the above hyperparameters and Stochastic Gradient Descent as the optimizer while calculating loss using binary cross-entropy. As per figure 7, the model works all right, and the model is overfitting due to data being low (142 images of the alpaca class and 185 images of the not alpaca class). The problem can be solved using image augmentation, where we can rotate, skew, zoom, and play with light and dark images to get some more data made artificially but still showing alpacas. Early Stopping was used with patience = 10 to save resources of the system.



```
              precision    recall  f1-score   support

         0.0       0.73      0.29      0.41        56
         1.0       0.48      0.86      0.62        43

    accuracy                           0.54        99
   macro avg       0.60      0.57      0.51        99
weighted avg       0.62      0.54      0.50        99
```

Fig 7. Classification Report for Proposed Model where Class 1 = Alpacas

As per figure 8, the early stopping was working on the lowest validation loss as opposed to training loss, and we can clearly see the overfitting of the model due to fewer data. The green dashed line showing the model parameter chosen for the proposed model was found at epoch = 17. It is always a good idea to get more image data so that we can train the model better, but it always consumes more resources and time and so for this experiment we are satisfied with the results as shown.

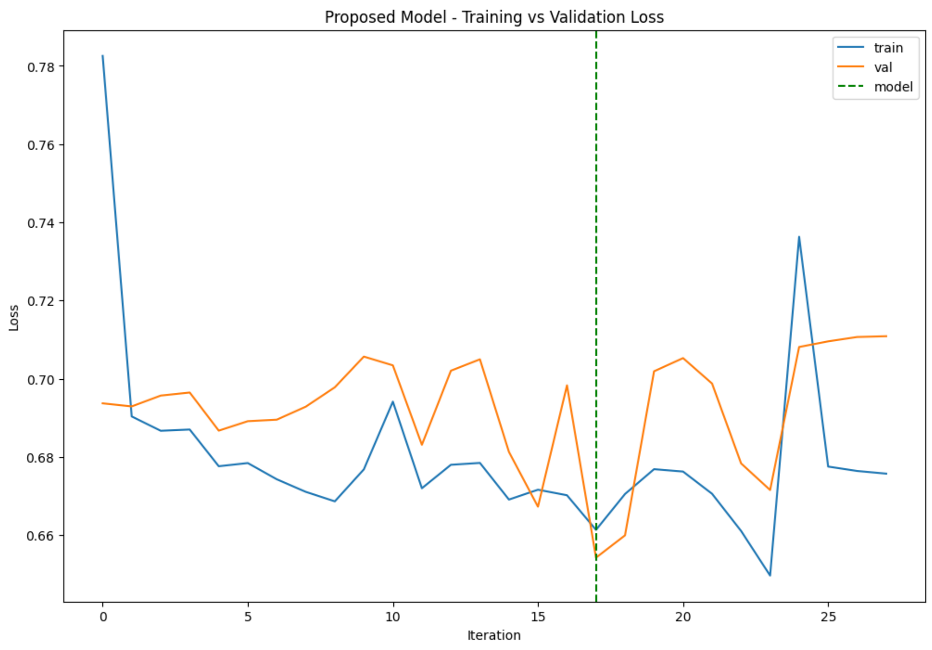
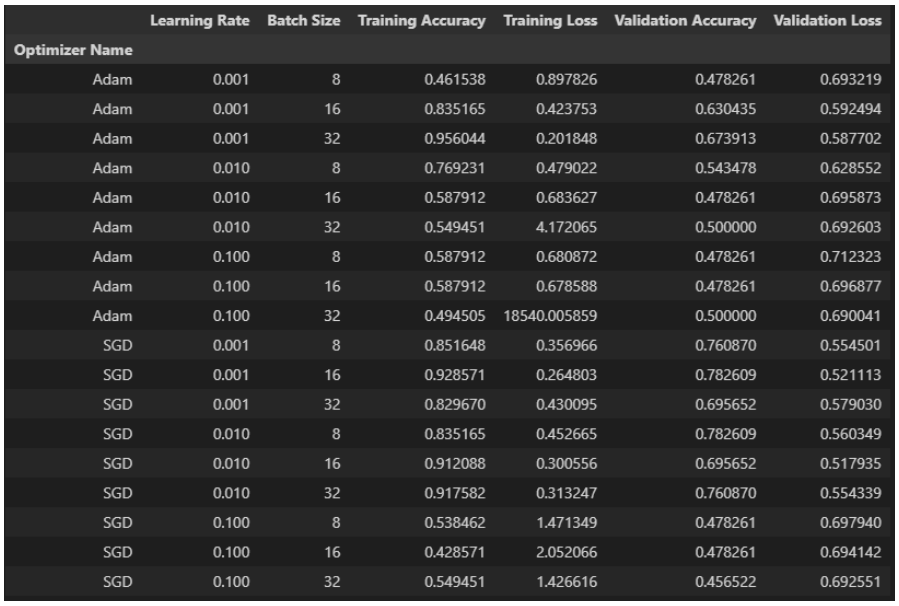


Fig 8. Training vs Validation Loss for Proposed Layer and Model Saving.

__Report the analysis of the model against changes in network depth and width as well as changing hyper parameters i.e., show how performance increase or decrease by changing learning rate {0.01, 0.001, or other}, batch sizes {8, 16, 50, or others depending on your system/laptop capacity}, epochs {100, 200, 700, 1000 or other depending on the model performance}, optimizers e.g., Adam or other factors.__
__Report all results in a table.__

The hyperparameter tuning took place without the use of Keras Tuner, so we have manually checked for losses and accuracy as shown in figure 9. As seen, Adam and SGD were the optimizers used, while taking learning rates as [0.1, 0.01, 0.00] and batch size as [8, 16, 32]. Epochs were not tested to save resources and early stopping gave good results. The early stopping took place according to the lowest validation loss.

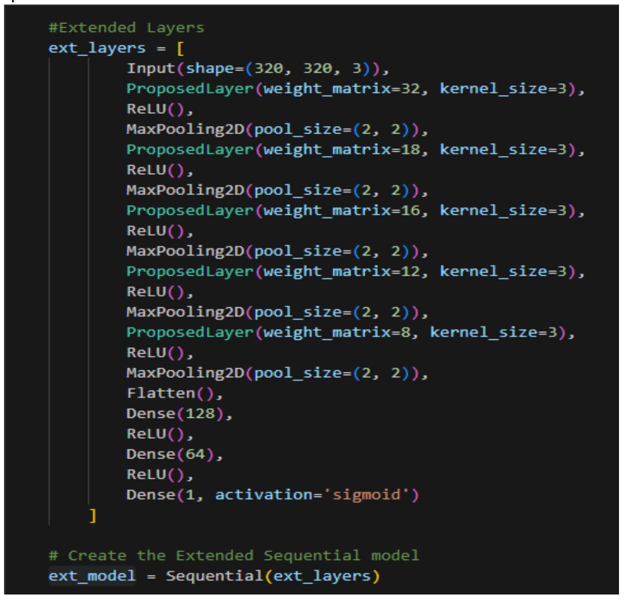
The best case we found looking at the table is at index 11 with SGD optimizer, 0.001 learning rate, and batch size 16, where we see the validation loss to be ~0.52 and training loss to be ~0.26 while validation accuracy is ~0.78 and training accuracy is ~0.93. This compensates for an underperforming model and shows promising results for actual use.



	Learning Rate	Batch Size	Training Accuracy	Training Loss	Validation Accuracy	Validation Loss
Optimizer Name						
Adam	0.001	8	0.461538	0.897826	0.478261	0.693219
Adam	0.001	16	0.835165	0.423753	0.630435	0.592494
Adam	0.001	32	0.956044	0.201848	0.673913	0.587702
Adam	0.010	8	0.769231	0.479022	0.543478	0.628552
Adam	0.010	16	0.587912	0.683627	0.478261	0.695873
Adam	0.010	32	0.549451	4.172065	0.500000	0.692603
Adam	0.100	8	0.587912	0.680872	0.478261	0.712323
Adam	0.100	16	0.587912	0.678588	0.478261	0.696877
Adam	0.100	32	0.494505	18540.005859	0.500000	0.690041
SGD	0.001	8	0.851648	0.356966	0.760870	0.554501
SGD	0.001	16	0.928571	0.264803	0.782609	0.521113
SGD	0.001	32	0.829670	0.430095	0.695652	0.579030
SGD	0.010	8	0.835165	0.452665	0.782609	0.560349
SGD	0.010	16	0.912088	0.300556	0.695652	0.517935
SGD	0.010	32	0.917582	0.313247	0.760870	0.554339
SGD	0.100	8	0.538462	1.471349	0.478261	0.697940
SGD	0.100	16	0.428571	2.052066	0.478261	0.694142
SGD	0.100	32	0.549451	1.426616	0.456522	0.692551

Fig 9. Data frame showing hyperparameter tuning analysis.

The model was also extended as seen in figure 10 and we used more layers for training. Unfortunately, it didn't help in getting a better score even though the promising results and the hyperparameters from figure 9 were taken. The results of the training are shown in figure 11, and this tells us that the data plays a pivotal role in training any model from scratch and needs a good training set and size to dominate the performance and metrics of the neural network.

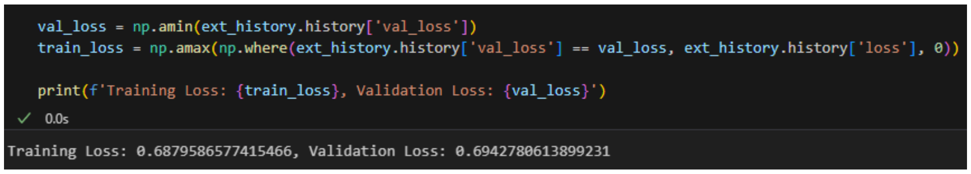


```
#Extended Layers
ext_layers = [
        Input(shape=(320, 320, 3)),
        ProposedLayer(weight_matrix=32, kernel_size=3),
        ReLU(),
        MaxPooling2D(pool_size=(2, 2)),
        ProposedLayer(weight_matrix=18, kernel_size=3),
        ReLU(),
        MaxPooling2D(pool_size=(2, 2)),
        ProposedLayer(weight_matrix=16, kernel_size=3),
        ReLU(),
        MaxPooling2D(pool_size=(2, 2)),
        ProposedLayer(weight_matrix=12, kernel_size=3),
        ReLU(),
        MaxPooling2D(pool_size=(2, 2)),
        ProposedLayer(weight_matrix=8, kernel_size=3),
        ReLU(),
        MaxPooling2D(pool_size=(2, 2)),
        Flatten(),
        Dense(128),
        ReLU(),
        Dense(64),
        ReLU(),
        Dense(1, activation='sigmoid')
    ]

# Create the Extended Sequential model
ext_model = Sequential(ext_layers)
```

Fig 10. Extension of the Proposed Model.

The ideology is good to work with smaller image sizes, but preprocessing and getting more data or augmentation may work to improve model performance, but its good to see that the model is still capable of creating okay performance as compared to CNN which is an inbuilt class of TensorFlow. The experimentation comes to a succession with the idea of kernel shape like the input and proposing this idea leads to even better experiments like changing the receptive field size, making a custom dense layer, or using conv1D after getting a flattened output.



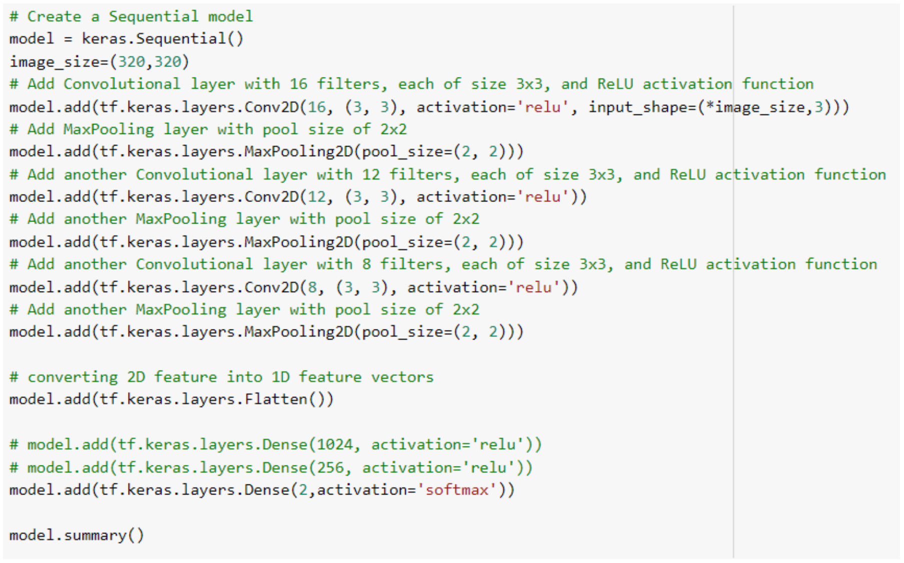
```
val_loss = np.amin(ext_history.history['val_loss'])
train_loss = np.amax(np.where(ext_history.history['val_loss'] == val_loss, ext_history.history['loss'], 0))

print(f'Training Loss: {train_loss}, Validation Loss: {val_loss}')
✓ 0.0s
Training Loss: 0.6879586577415466, Validation Loss: 0.6942780613899231
```

Fig 11. Training and Validation Loss of Extended Model.

<u>Report what result you achieved after training a Convolutional Neural Network of same depth</u>

The training a Convolutional Neural Network was done with same depth using the following depth :



```
# Create a Sequential model
model = keras.Sequential()
image_size=(320,320)
# Add Convolutional layer with 16 filters, each of size 3x3, and ReLU activation function
model.add(tf.keras.layers.Conv2D(16, (3, 3), activation='relu', input_shape=(*image_size,3)))
# Add MaxPooling layer with pool size of 2x2
model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
# Add another Convolutional layer with 12 filters, each of size 3x3, and ReLU activation function
model.add(tf.keras.layers.Conv2D(12, (3, 3), activation='relu'))
# Add another MaxPooling layer with pool size of 2x2
model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))
# Add another Convolutional layer with 8 filters, each of size 3x3, and ReLU activation function
model.add(tf.keras.layers.Conv2D(8, (3, 3), activation='relu'))
# Add another MaxPooling layer with pool size of 2x2
model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2)))

# converting 2D feature into 1D feature vectors
model.add(tf.keras.layers.Flatten())

# model.add(tf.keras.layers.Dense(1024, activation='relu'))
# model.add(tf.keras.layers.Dense(256, activation='relu'))
model.add(tf.keras.layers.Dense(2,activation='softmax'))

model.summary()
```

Fig 12. CNN Model Implementation using Sequential Model.

The 2D CNN trained for specific parameters give Tensor board as shown below:

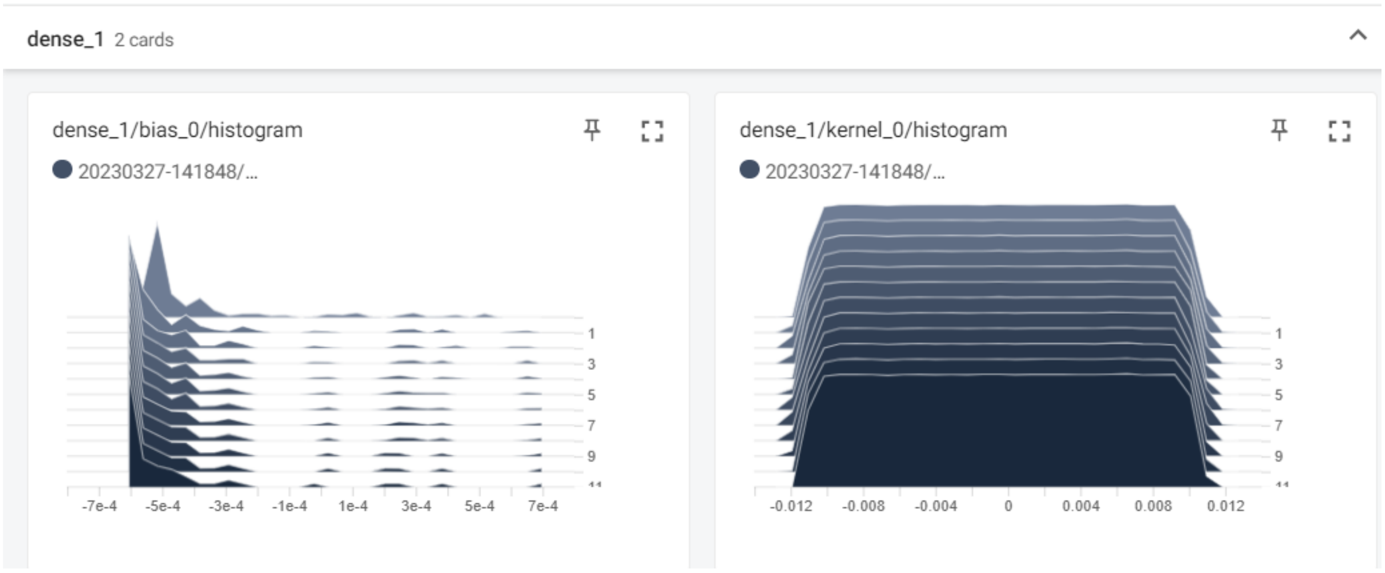


Fig 13(a). TensorBoard showing dense layer 1's histograms.

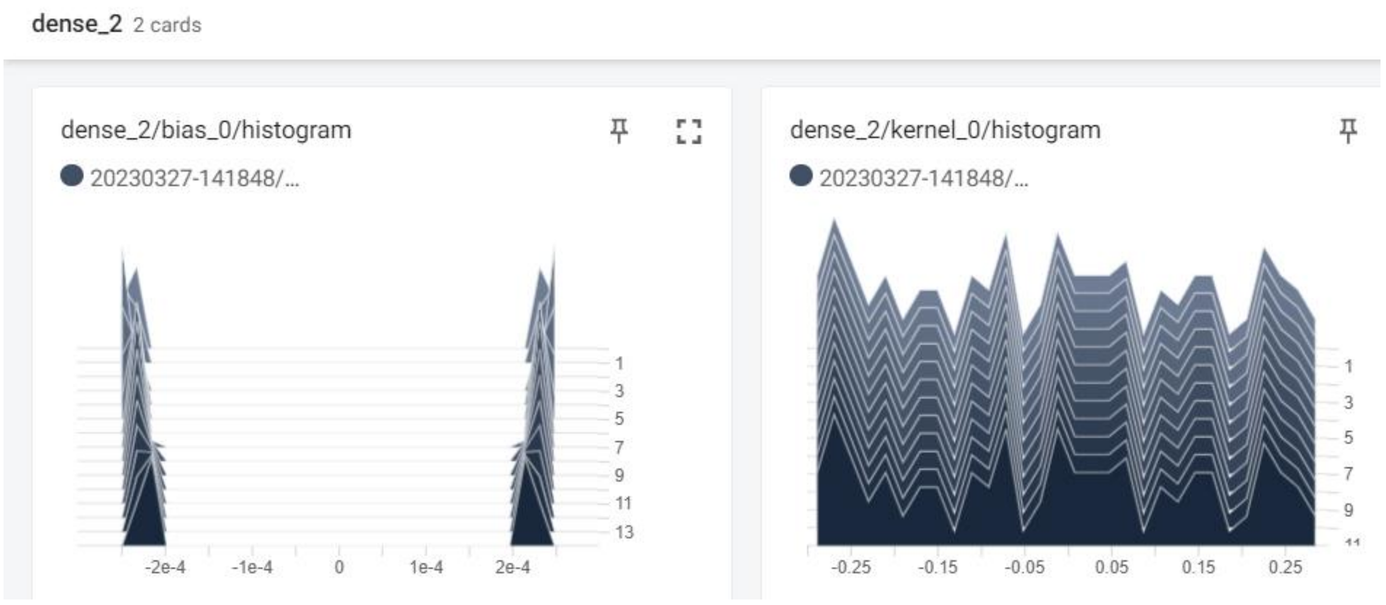


Fig 13(b). TensorBoard showing dense layer 2's histograms.

The 2D CNN model is trained and training and validation loss is calculated using the Adam optimizer. The model is selected for suitable values of training and validation loss as shown in the figure below.

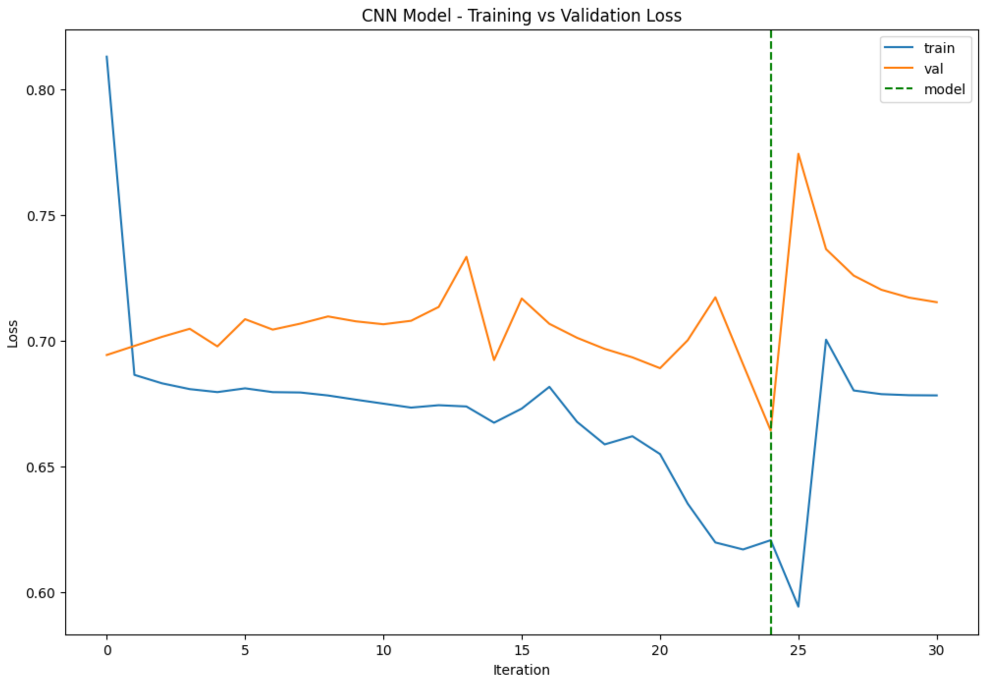
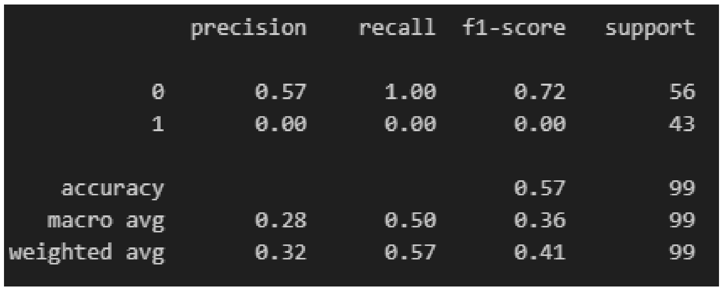


Fig 14. CNN Model's Training and Validation Loss with Model Selection.

The classification report for 2D CNN model does not predict the alpaca images as the f1 score is zero. The proposed model implementation works better in this case. The macro average is about 36% for the model whereas the macro average of proposed model is around 54% and performs well over 2D CNN.



```
              precision    recall  f1-score   support

           0       0.57      1.00      0.72        56
           1       0.00      0.00      0.00        43

    accuracy                           0.57        99
   macro avg       0.28      0.50      0.36        99
weighted avg       0.32      0.57      0.41        99
```

Fig 15. CNN Model's Classification Report with 1 being Alpaca Class.

Change of code to work over 1-Dimensional data?

To modify the code to work with 1-dimensional data, we would need to make the following changes:

1) Replace Conv2D with Conv1D.
2) MaxPooling1D should be used instead of MaxPooling2D.
3) Make the input one-dimensional by changing its shape.
4) Modify the shape of the kernel in the ProposedLayer class to have a shape of (kernel size, input channels, and weight matrix).
5) Modify the call method of the ProposedLayer class to perform 1D convolutions instead of 2D convolutions.

There should be two components in the strides parameter for a 1D convolution operation as there are only two spatial dimensions that the filter or kernel can move along. Applying a 1D convolution operation with stride=1 on an input tensor of shape (batch size, sequence length, and input channels), for instance

Changes to the code: To match the input shape of the 1-dimensional data, the weight matrix shape should be changed to (kernel size, input channels, weight matrix). In order to make the input tensor 3D for convolution, it should also be modified by adding a new dimension using the tf.expand dims function. The resulting tensor should then be restored to its original shape by using tf.squeeze to remove the extra dimension.

Change of code to work over 3-Dimensional data?

To modify the code to work with 3-dimensional data, we would need to make the following changes:

1) Replace Conv2D with Conv3D.
2) Replace MaxPooling2D with MaxPooling3D.
3) Modify the shape of the input to be three-dimensional.
4) Modify the shape of the kernel in the ProposedLayer class to have a shape of (kernel size, kernel size, input channels, weight matrix).
5) Modify the call method of the ProposedLayer class to perform 3D convolutions instead of 2D convolutions.

Moreover, the strides argument would have five items instead of four if we were implementing a 3D convolution process. These elements would represent the strides along each of the three spatial dimensions (width, height, and depth), as well as the batch size and number of channel dimensions. Additionally, the TensorFlow function tf.nn.conv2d should be changed to tf.nn.conv3d, which performs a 3D convolution operation on a batch of 3D input tensors with a set of 3D filters, like tf.nn.conv2d, which performs a 2D convolution operation on a batch of 2D input tensors with a set of 2D filters.

Conclusion :

The results for the implementation shown above show great promise to improve the dataset provided the increase in the size of the dataset. The results when compared show that our proposed model works better over the traditional CNN model, and with some additional changes, can work even wonders and we can predict the model's accuracy to go beyond 85%. Data augmentation will be a major consideration for anyone who wants to proceed further with this experimentation, and this model is also flexible enough to be trained on other tasks i.e., regression, classification, and unsupervised learning (nearest neighbours, etc.). The impact of this model may prove to be even better when taken into the consideration every time some programmers may want to increase their model's performance and can become a debate amongst professionals when trying to take into a real use case. The proposed model can be termed as computationally efficient using a weight matrix of the size of the input shape provided by the given dataset. The advantage is that the estimated error is backpropagated using the backpropagation algorithm via the same receptive field. The identical weights in each of the receptive fields are used by the neuron to calculate the gradient. This aids in calculating the loss function's gradient with respect to the inputs.

References

[1] D. I. Ullah, "CNN vs New Network approach," University of Galway, Galway, Ireland, 2022.

[2] F. Chollet, "Building powerful image classification models using very little data," The Keras Blog, 05 June 2016. [Online]. Available: https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html.

[3] "Get started with TensorBoard," TensorFlow, 11 February 2023. [Online]. Available: https://www.tensorflow.org/tensorboard/get_started.

[4] F. Chollet, "Making new layers and models via subclassing," Keras, 13 April 2020. [Online]. Available: https://keras.io/guides/making_new_layers_and_models_via_subclassing/.

[5] "tf.nn.conv2d," TensorFlow, 23 March 2023. [Online].

[6] "TensorBoard: TensorFlow's visualization toolkit," TensorFlow, [Online]. Available: https://www.tensorflow.org/tensorboard.