Case studies assignment 2

Report the following:

1. **At first page, write your names, roll numbers, and group number, and then in 50-150 words, report who did what in the assignment e.g., who came up with the idea of how to implement it, who implemented it (as a whole or part of it), who debugged it, who gave suggestions (and what suggestion/s), who wrote the report, who did which simulations.**

Answer:

Group number: 07

Team members	**Student id**
Thomas Kuruvilla	22231052
Mahima Sharma	22225563
Ankit Juneja	22221327

We all three worked together in every part equally, however there is and detailed overview of tasks: Data pre-processing has been taken care by Ankit Juneja. Mahima Sharma and Thomas Kuruvilla has been completed report writing for this. Thomas Kuruvilla came up with the idea of the proposed layer and Ankit Juneja and Mahima Sharma has contributed the calculations for the proposed layer and implementation of it using python code and each one of us has contributed our results in report writing. Thomas Kuruvilla and Mahima Sharma has trained the proposed model, and each of us(Ankit, Thomas and Mahima) has taken a different setting of hyperparameters(Three different hyperparameters). Ankit Juneja debugged it and completed the report writing for the same. Ankit has completed the training of the convolutional neural network of the same depth and performed debugging. Mahima Sharma and Thomas Kuruvilla completed the report writing and comparisons. Mahima Sharma and Thomas Kuruvilla has taken care of the tensor board and Ankit Juneja completed the report writing for this. Thomas Kuruvilla has performed data visualisations and Mahima Sharma completed the report writing for this.

2. Explain how your group implemented the layer within existing library (keras, TensorFlow, PyTorch, or other) or from scratch [3 Marks]

Answer: Firstly we have defined a custom layer class called Proposed_layer in Keras, which inherits from the keras.layers.Layer class. The Proposed_layer class creates a convolutional layer that performs a sliding window operation on the input tensor and applies a set of learned filters to produce an output tensor.

The Proposed_layer class has three methods: __init__, build, and call.

In **__init__** we have initialized the number of filters in the layer, which is passed as an argument to the constructor method. The super() function is used to call the constructor of the parent class.

In the **build** method, we used to create the layer's variables, including the kernel or filter weights. The kernel is a weight tensor with random normal initialization, and its shape is (filters, input_shape[-1], input_shape[-1]), where input_shape is the shape of the input tensor.

In **call** method we perform the forward pass of the layer. It takes the input tensor and applies the learned filters to produce an output tensor. The output tensor has the same shape as the input tensor, except for the last two dimensions, which are reduced by 2 due to the sliding window operation.

Also, the call method starts by getting the number of samples in the batch and the number of feature maps in the input tensor. It then initializes the output tensor with zeros and gets the dimensions of the output tensor.

Next, the call method loops over each sample in the batch and each filter in the layer. It initializes the starting indices for the sliding window and loops over each row and column in the input feature maps. It initializes the weighted sum to 0 and loops over each feature map in the input tensor. It performs a 2D convolution between the kernel and the input feature map and adds the result to the weighted sum. It then creates a sparse tensor with the weighted sum and the corresponding indices and adds the sparse tensor to the output tensor. It moves the sliding window one column to the right and one row down and repeats the process until the entire input tensor has been processed.

Finally, the call method reshapes the output tensor to match the input shape and returns the output tensor.

Overall, the Proposed_layer class defines a convolutional layer that performs a sliding window operation on the input tensor and applies a set of learned filters to produce an output tensor. This can be useful for our assignment for alpaca and not alpaca dataset.

```
# Define a custom layer class that inherits from keras.layers.Layer
class proposed_layer(keras.layers.Layer):

    # Define the constructor method
    def __init__(self, filters):
      # Call the constructor of the parent class
      super(proposed_layer, self).__init__()
      # Initialize the number of filters
      self.filters = filters

    # Define the build method to create the layer's variables
    def build(self, input_shape):
      # Create a weight tensor with random normal initialization
      # The shape of the weight tensor is (filters, input_shape[-
1], input_shape[-1])
      self.kernel = self.add_weight(
        shape=(self.filters, input_shape[-1], input_shape[-1]),
        initializer="random_normal",
        trainable=True,
      )

    # Define the call method to perform the forward pass of the layer
    def call(self, inputs):
      # Get the number of samples in the batch
      num_samples = tf.shape(inputs)[0]
      # Get the number of feature maps in the input
      previous_maps = tf.shape(inputs)[1]
      # Initialize the output tensor with zeros
      # The shape of the output tensor is (num_samples, filters, input_
shape[2]-2, input_shape[3]-2)
      features_maps = tf.zeros((num_samples, self.filters, tf.shape(inp
uts)[2]-2, tf.shape(inputs)[3]-2))
      # Get the dimensions of the output tensor
      a0 = tf.shape(features_maps)[0]
      a1 = tf.shape(features_maps)[1]
      a2 = tf.shape(features_maps)[2]
      a3 = tf.shape(features_maps)[3]

      # Loop over each sample in the batch
      for n in range(a0):
        # Loop over each filter in the layer
        for k in range(a1):
          # Initialize the starting indices for the sliding window
          x1 = 0
          x2 = 0
          # Loop over each row in the input feature maps
          for i in range(a2):
            # Reset the starting index for the sliding window
            x1 = 0
            # Loop over each column in the input feature maps
            for j in range(a3):
              # Initialize the weighted sum to 0
              weighted_sum = tf.constant(0.0)
              # Loop over each feature map in the input
              for feature in range(previous_maps):
                # Perform a 2D convolution between the kernel and the i
nput feature map
                # Add the result to the weighted sum
                weighted_sum += tf.math.reduce_sum(tf.matmul(inputs[n,
feature, x1:x1+3, x2:x2+3], self.kernel[k, x1:x1+3, x2:x2+3]))
              # Create a sparse tensor with the weighted sum and the co
rresponding indices
              sparse_tensor = tf.SparseTensor(indices=[[n, k, i, j]],
                        values=[weighted_sum],
                        dense_shape=[a0, a1, a2, a3])
              # Add the sparse tensor to the output tensor
              features_maps = features_maps + tf.sparse.to_dense(sparse
_tensor)
              # Move the sliding window one column to the right
              x1 += 1
            # Move the sliding window one row down
            x2 += 1

      # Reshape the output tensor to match the input shape
```

```
        return tf.reshape(features_maps, (a0, a1, a2, a3))
```

3. Report the successful implementation of the layer and complete model with a figure in tensorboard. [10 Marks]

Answer: We build the model that has a specific architecture that consists of several layers that are defined in the __init__ function. The layers of the model are defined in the constructor of the custom_model class.

Firstly, the super(custom_model, self).__init__() method calls the constructor of the parent class.

Next, several layers are defined, including three custom layers (prop1, prop2, and prop3) that have 5, 3, and 2 filters respectively instead of 16, 12 and 8 because our laptops RAM has crashed several times when we ran with these filters. These layers are followed by an activation layer (actv) with the ReLU activation function and a max pooling layer (maxp) with a pool size of 2x2.

We have also added the flatten layer (flatten) and a fully connected layer (dense1) with 16 units and a ReLU activation function.

Finally, the output layer (output_layer) has one neuron and the softmax activation function.

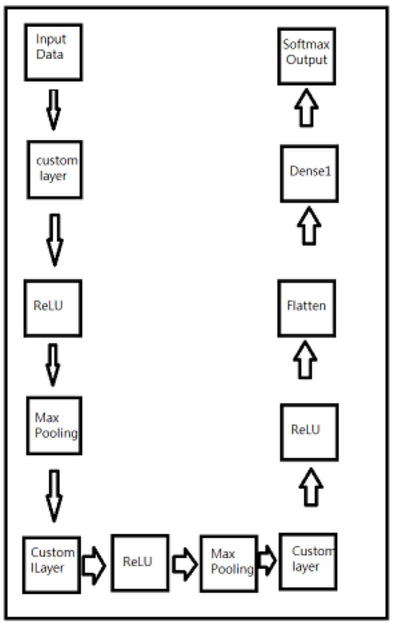
The forward pass of the model is defined in the call method. In the call method, the input data is passed through the layers of the model, in the order they were defined in the __init__ method.

The input is passed through the first custom layer (prop1), followed by an activation layer (actv) with the ReLU activation function, and then through a max pooling layer (maxp) with a pool size of 2x2. The output of the max pooling layer is passed through the second custom layer (prop2), another activation layer (actv), and then another max pooling layer (maxp) with a pool size of 2x2.

The output of the second max pooling layer is passed through the third custom layer (prop3) and another activation layer (actv). The output of the third custom layer is then flattened (flatten) and passed through the fully connected layer (dense1).

Finally, the output of the fully connected layer is passed through the output layer (output_layer) with the softmax activation function. The output of the model is returned by the call method.

Diagram of implementation:

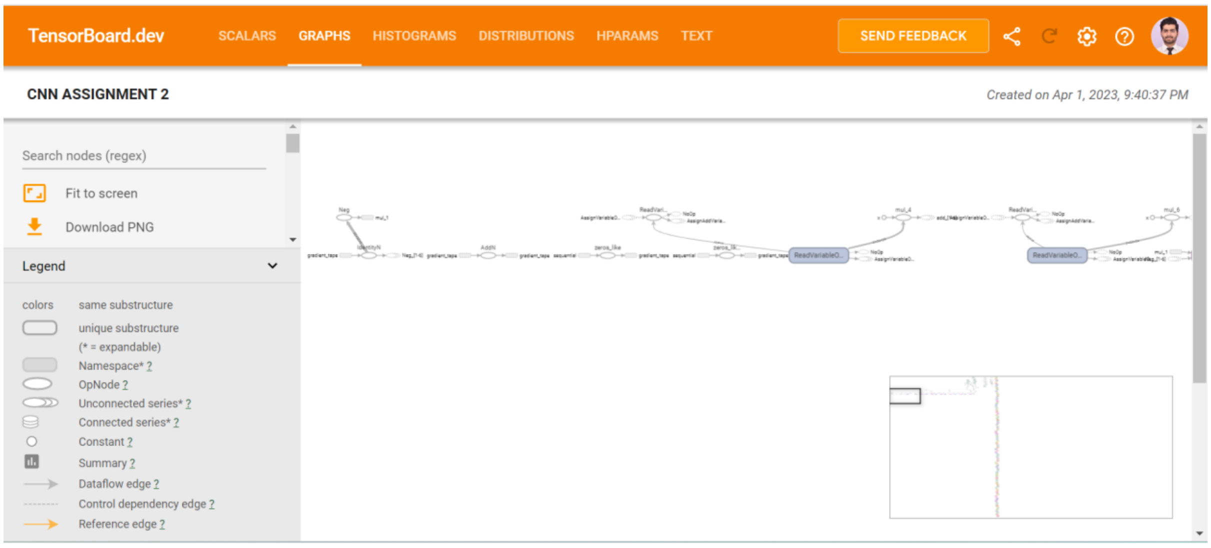


Code:

```
class custom_model(tf.keras.Model):
    # Define the model architecture
    def __init__(self):

        # Call the constructor of the parent class
        super(custom_model, self).__init__()

        # Define the layers of the model
        self.prop1 = Proposed_layer(filters=5)   # First custom layer w
ith 5 filters
        self.prop2 = Proposed_layer(filters=3)   # Second custom layer
with 3 filters
        self.prop3 = Proposed_layer(filters=2)   # Third custom layer w
ith 2 filters
        self.actv = Activation(activations.relu)   # Activation layer w
ith ReLU activation function
        self.maxp = MaxPooling2D(pool_size=(2,2))  # Max pooling layer
with a pool size of 2x2

        self.flatten = Flatten()   # Flatten layer
        self.dense1 = Dense(units=16, activation="relu")  # Fully conne
cted layer with 16 units(neurons) and ReLU activation function
        self.output_layer = Dense(units=1, activation="softmax")  # Out
put layer with 1 unit(neuron) and softmax activation function

    # Define the forward pass of the model
    def call(self, inputdata):

        x = self.prop1(inputdata)  # Pass the input through the first c
ustom layer
```

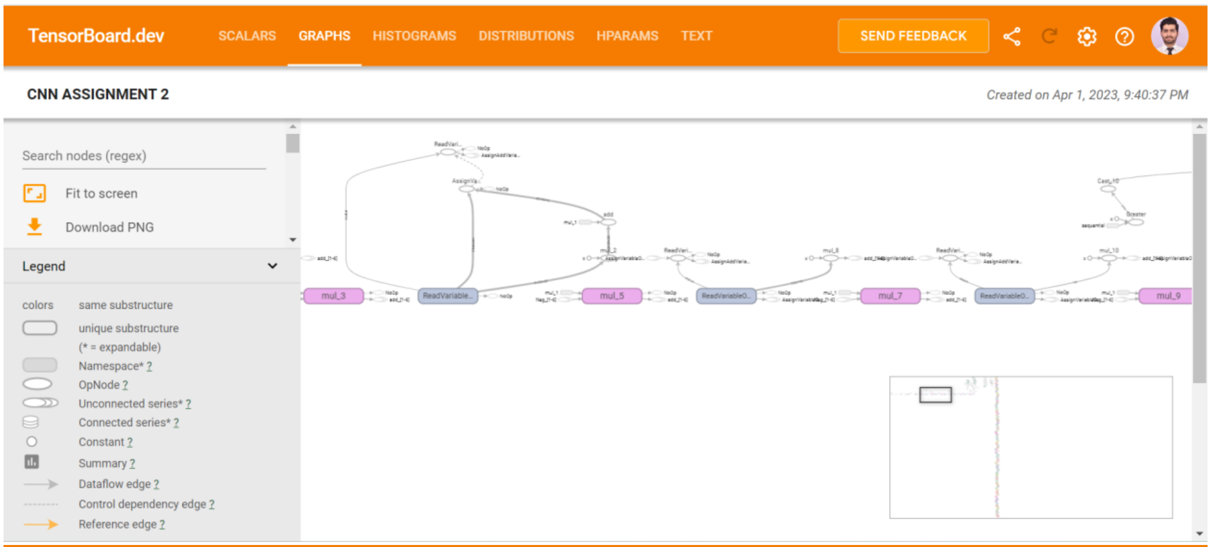
```
        x = self.actv(x)   # Apply the ReLU activation function to the output of the first custom layer
        x = self.maxp(x)   # Apply max pooling to the output of the first custom layer
        x = self.prop2(x)   # Pass the output of the max pooling layer through the second custom layer
        x = self.actv(x)   # Apply the ReLU activation function to the output of the second custom layer
        x = self.maxp(x)   # Apply max pooling to the output of the second custom layer
        x = self.prop3(x)   # Pass the output of the max pooling layer through the third custom layer
        x = self.actv(x)   # Apply the ReLU activation function to the output of the third custom layer
        x = self.flatten(x)   # Flatten the output of the third custom layer
        x = self.dense1(x)   # Pass the flattened output through the fully connected layer
        x = self.output_layer(x)   # Pass the output of the fully connected layer through the output layer with softmax activation function

        return x   # Return the output of the model
```

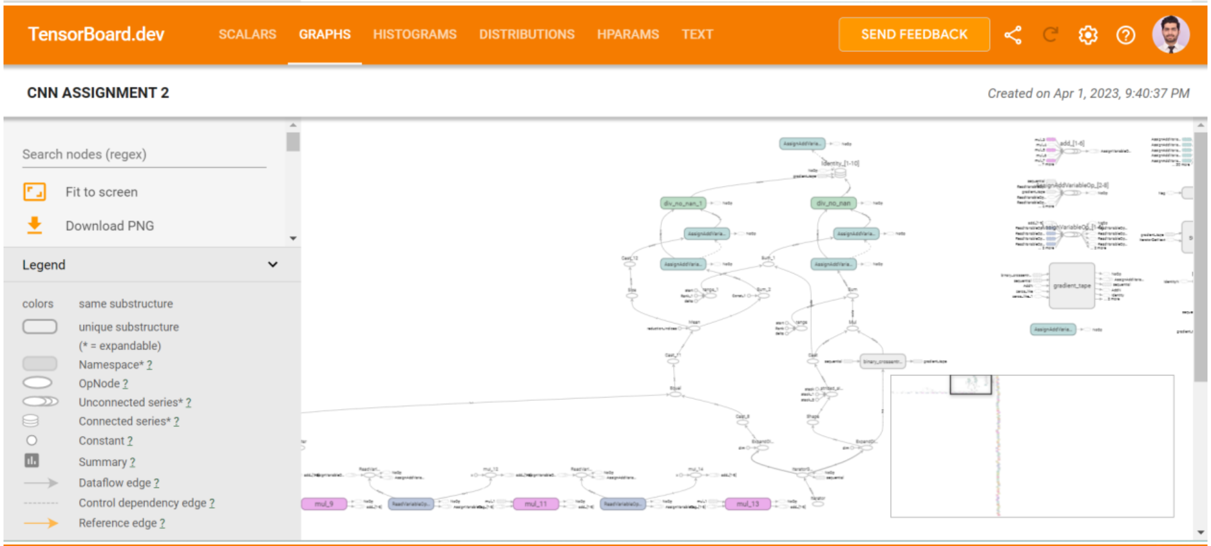
TesnsorBoard Graphs:

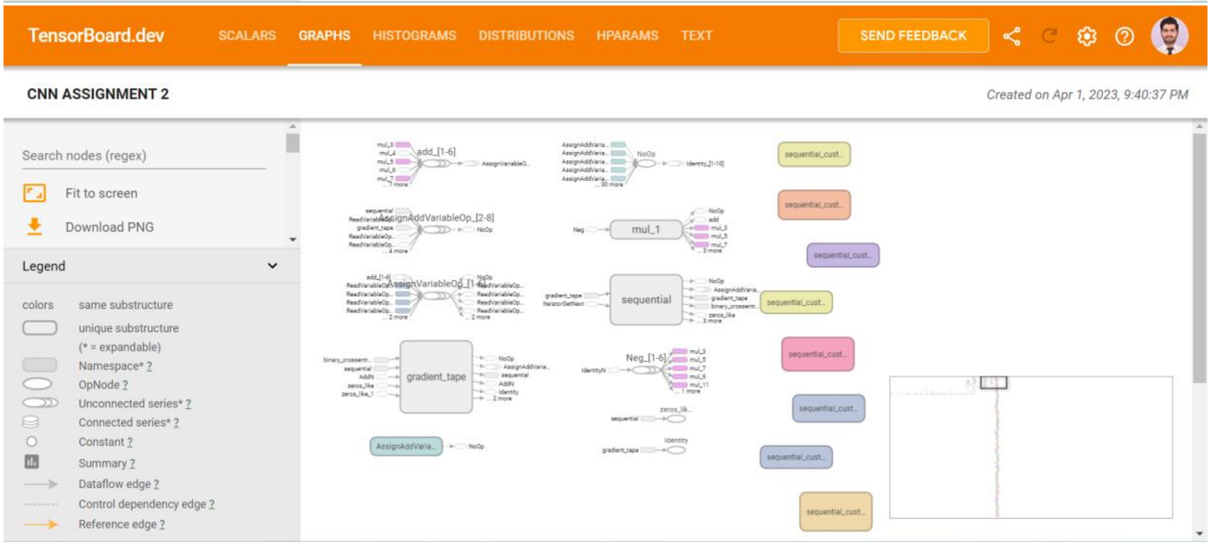


CNN ASSIGNMENT 2

Created on Apr 1, 2023, 9:40:37 PM







[TesnorBoard link for diagrams]

4. Load and distribute the data for training, validation, and testing [2 Marks]

Answer: Firstly, we have defined the path to the directory containing the image dataset. This is followed by defining the image size to be used, which is 28 in this case. A list of categories in the dataset is then defined, with the categories being "alpaca" and "not alpaca".

Second, we have created an empty list to hold the training data, which is later filled by the images that are loaded and preprocessed using the load_data() function. This function iterates over the categories and loads each image, and converts it to grayscale, and resizes it to the desired size of 28x28 pixels. The plt.imshow() function is used to display each image after it has been resized.

The loaded and preprocessed images are then appended to the training_data list along with their respective labels, which are assigned based on the index of the category in the categories list.

Once all the images have been loaded and preprocessed, the training_data list is shuffled randomly to ensure that the order of the images does not impact the model's training. The training data is then split into features and labels using a for loop and the append() method.

The feature and label lists are then converted to numpy arrays using the np.array() method, and the feature array is reshaped to have a channel-first format. This is followed by casting the images to a TensorFlow-supported format using the tf.cast() method, and the shapes of the feature and label arrays are printed.

Finally, our training data is split into training (70%), validation(15%), and test sets(15%) using slicing operations on the numpy arrays. The shapes of the training, validation, and test data are printed to confirm that the data has been split correctly.

Sample code:

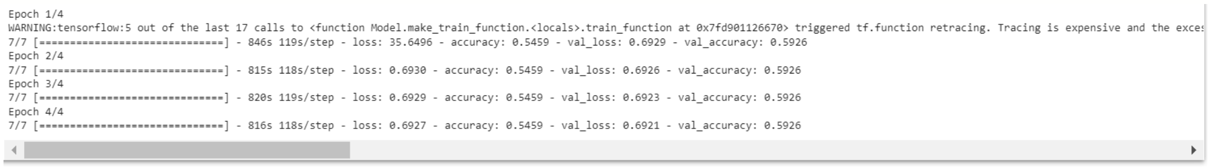
```
# Define the path to the directory containing the image dataset

data_dir = "/content/drive/MyDrive/dataset"

# Define the image size to be used
img_size = 28

# Define the list of categories in the dataset
categories = ["alpaca", "not alpaca"]

# Create an empty list to hold the training data
training_data = []
print("All images of alpaca  and not alpaca after reducing their dimins
ions.")
# Define a function to load and preprocess all images in the dataset
def load_data():
    # Iterate over the categories
    for category in categories:
        # Get the path to the category directory
        category_path = os.path.join(data_dir, category)
        # Get the class label for this category
        class_label = categories.index(category)
        # Iterate over the images in the category directory
        for img_name in os.listdir(category_path):
            try:
                # Load the image and convert it to grayscale
                img_path = os.path.join(category_path, img_name)
                img_original = cv2.imread(img_path, cv2.IMREAD_GRAYSCAL
E)
                # Resize the image to the desired size
                img = cv2.resize(img_original, (img_size, img_size))
                plt.imshow(img, cmap = "gray")
                plt.show()
                # Add the image and its label to the training data list
                training_data.append((img, class_label))

            except Exception as e:
                pass

# Load the data
load_data()

# Shuffle the training data
random.shuffle(training_data)

# Split the training data into features and labels
Features = []
y = []
for features, label in training_data:
    Features.append(features)
    y.append(label)

# Convert the feature and label lists to numpy arrays
Features = np.array(Features)
y = np.array(y)

# Reshape the feature array to have channel first format
Features = np.array(Features).reshape(-1, 1, Img_Size, Img_Size)
# Casting images to tensorflow supported format
Features = tf.cast(Features, tf.float32)
# Print the shapes of the feature and label arrays
data_length = len(Features)
validation_test_size = data_length // 3
validation_size = validation_test_size // 2

train_Features = Features[:data_length - validation_test_size]
train_y = y[:data_length - validation_test_size]

valid_Features = Features[data_length - validation_test_size:data_lengt
h - validation_test_size + validation_size]
```

```
valid_y = y[data_length - validation_test_size:data_length - validation
_test_size + validation_size]

test_Features = Features[data_length - validation_test_size + validatio
n_size:]
test_y = y[data_length - validation_test_size + validation_size:]

print(f"Features + Label: {data_length}")
print(f"Train data: Features: {train_Features.shape}, y: {train_y.shape
}")
print(f"Val data: Features: {valid_Features.shape}, y: {valid_y.shape}"
)
print(f"Test data : Features: {test_Features.shape}, y: {test_y.shape}"
)
```

5. Train the model with the dataset provided in this assignment folder and report the training and testing performance against various hyper parameters (e.g. learning rate = 0.1, batch size = 32, epochs = 500,) using stochastic gradient decent [9 Marks].

Answer: We have test the performance against different hyper parameter:
First, we have run in default hyper parameters:

```
# Set the learning rate of optimizer to 0.01

alpha = 0.01

# Set the optimizer to Stochastic Gradient Descent (SGD)
adam = False

# Set the batch size to None (which means that the default batch size w
ill be used)
batchSize = None

# Set the number of epochs to 4
epochs = 4

# Instantiate a custom model
Proposed_Model = custom_model()

# Train the model and store the training history in History_data
History_data = modelHistory(Proposed_Model, alpha, adam, batchSize, epo
chs)
```

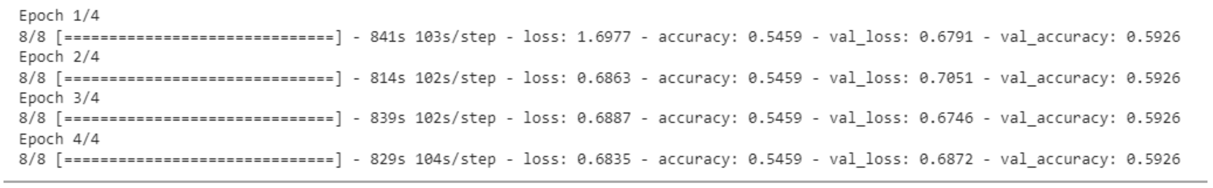


```
Epoch 1/4
WARNING:tensorflow:5 out of the last 17 calls to <function Model.make_train_function.<locals>.train_function at 0x7fd901126670> triggered tf.function retracing. Tracing is expensive and the exce
7/7 [==============================] - 846s 119s/step - loss: 35.6496 - accuracy: 0.5459 - val_loss: 0.6929 - val_accuracy: 0.5926
Epoch 2/4
7/7 [==============================] - 815s 118s/step - loss: 0.6930 - accuracy: 0.5459 - val_loss: 0.6926 - val_accuracy: 0.5926
Epoch 3/4
7/7 [==============================] - 820s 119s/step - loss: 0.6929 - accuracy: 0.5459 - val_loss: 0.6923 - val_accuracy: 0.5926
Epoch 4/4
7/7 [==============================] - 816s 118s/step - loss: 0.6927 - accuracy: 0.5459 - val_loss: 0.6921 - val_accuracy: 0.5926
```

It shows the training progress of the model, the training loss starts very high (35.6496) but decreases significantly in subsequent epochs, indicating that the model is learning. The training accuracy remains constant at 0.5459, which suggests that the model is not improving its predictions. The validation loss and accuracy remain constant at 0.6926 and 0.5926 respectively, which means that the model is not overfitting the training data.

Second Hyperparameters (configured manually),(Hyper-Parameter: Alpha = 0.001,optimiser = sgd,batchsize = 30,epochs = 4)

```
# Define the hyperparameters
alpha = 0.001
adam = False
batchSize = 30
epochs = 4

# Create an instance of the custom model
Proposed_Model = custom_model()

# Train the model with the given hyperparameters and save the training
history
History_data = modelHistory(Proposed_Model, alpha, adam, batchSize, epo
chs)
```

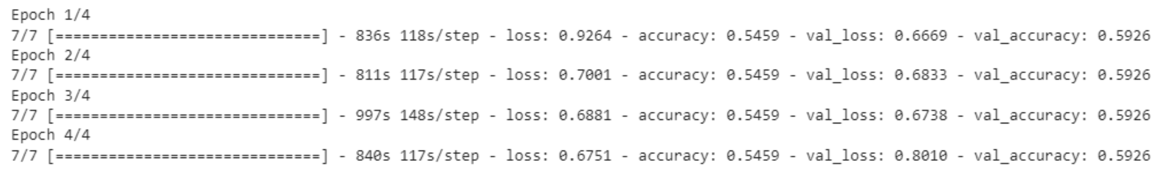


```
Epoch 1/4
8/8 [==============================] - 841s 103s/step - loss: 1.6977 - accuracy: 0.5459 - val_loss: 0.6791 - val_accuracy: 0.5926
Epoch 2/4
8/8 [==============================] - 814s 102s/step - loss: 0.6863 - accuracy: 0.5459 - val_loss: 0.7051 - val_accuracy: 0.5926
Epoch 3/4
8/8 [==============================] - 839s 102s/step - loss: 0.6887 - accuracy: 0.5459 - val_loss: 0.6746 - val_accuracy: 0.5926
Epoch 4/4
8/8 [==============================] - 829s 104s/step - loss: 0.6835 - accuracy: 0.5459 - val_loss: 0.6872 - val_accuracy: 0.5926
```

Above output indicates that the model achieved an accuracy of 54.59% on the training set and 59.26% on the validation set for all 4 epochs. However, the loss and accuracy values did not change much throughout the training process, which may indicate that the model did not learn much from the training data.

Third Hyperparameters (configured manually),(Hyper-Parameter: Alpha= .01 ,batchsize=None ,optimiser=adam ,epochs=4)

```
# Setting hyperparameters for the model
alpha = 0.01
adam = True
batchSize = None
epochs = 4

# Creating a custom model
Proposed_Model = custom_model()

# Training the model and saving the history
History_data = modelHistory(Proposed_Model,alpha,adam,batchSize,epochs)
```

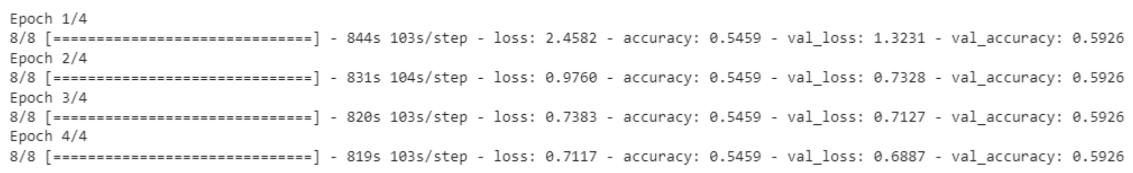


```
Epoch 1/4
7/7 [==============================] - 836s 118s/step - loss: 0.9264 - accuracy: 0.5459 - val_loss: 0.6669 - val_accuracy: 0.5926
Epoch 2/4
7/7 [==============================] - 811s 117s/step - loss: 0.7001 - accuracy: 0.5459 - val_loss: 0.6833 - val_accuracy: 0.5926
Epoch 3/4
7/7 [==============================] - 997s 148s/step - loss: 0.6881 - accuracy: 0.5459 - val_loss: 0.6738 - val_accuracy: 0.5926
Epoch 4/4
7/7 [==============================] - 840s 117s/step - loss: 0.6751 - accuracy: 0.5459 - val_loss: 0.8010 - val_accuracy: 0.5926
```

In this case, model was able to achieve a training accuracy of 0.5459, but the validation accuracy was 0.5926, which suggests that the model may be overfitting. Additionally, the validation loss increased in the last epoch, which could indicate that the model is not generalizing well to new data. Further analysis of the model and its performance may be necessary to determine if any changes need to be made to improve its accuracy and performance.

Fourth Hyperparameters (configured manually),(Hyper-Parameter: Alpha= .01 ,batchsize=None ,optimiser=adam ,epochs=4)

```
# Setting the values for alpha, adam, batch size, and epochs
alpha = 0.001
adam = True
batchSize = 30
epochs = 4

# Creating a custom model
Proposed_Model = custom_model()

# Fitting the model with the specified hyperparameters and storing the
training history
History_data = modelHistory(Proposed_Model, alpha, adam, batchSize, epo
chs)
```



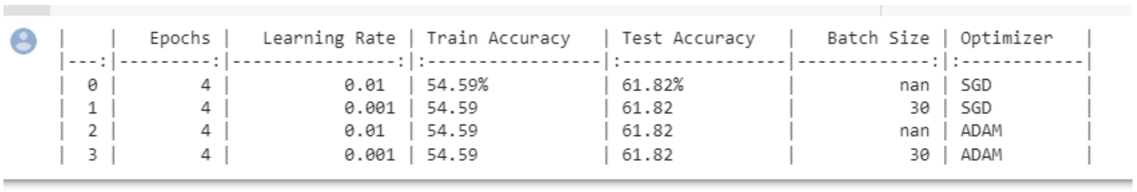
```
Epoch 1/4
8/8 [==============================] - 844s 103s/step - loss: 2.4582 - accuracy: 0.5459 - val_loss: 1.3231 - val_accuracy: 0.5926
Epoch 2/4
8/8 [==============================] - 831s 104s/step - loss: 0.9760 - accuracy: 0.5459 - val_loss: 0.7328 - val_accuracy: 0.5926
Epoch 3/4
8/8 [==============================] - 820s 103s/step - loss: 0.7383 - accuracy: 0.5459 - val_loss: 0.7127 - val_accuracy: 0.5926
Epoch 4/4
8/8 [==============================] - 819s 103s/step - loss: 0.7117 - accuracy: 0.5459 - val_loss: 0.6887 - val_accuracy: 0.5926
```

For these hyperparameters, it shows that training loss and accuracy start at high values and remain relatively constant throughout the training process, indicating that the model is not learning well. The validation loss and accuracy are also constant, and the accuracy is only slightly above 50%, which suggests that the model is not performing well and it appears that model is underfitted.

6. 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, even if they are not good. Also show their training and testing performance in the form of a graph [7 Marks].

Answer:

Attached below table provides the results of four different training configurations. The configurations vary in terms of the number of epochs, learning rate, batch size, and optimizer used. The training and testing accuracies are reported for each configuration:



```
|    |   Epochs |   Learning Rate | Train Accuracy    | Test Accuracy    |   Batch Size | Optimizer   |
|---:|---------:|----------------:|:------------------|:-----------------|-------------:|:------------|
|  0 |        4 |           0.01  | 54.59%            | 61.82%           |          nan | SGD         |
|  1 |        4 |           0.001 | 54.59             | 61.82            |           30 | SGD         |
|  2 |        4 |           0.01  | 54.59             | 61.82            |          nan | ADAM        |
|  3 |        4 |           0.001 | 54.59             | 61.82            |           30 | ADAM        |
```

Detailed Answer: The four configurations used to train the model are described below:

Configuration 0:

Epochs: 4

Learning Rate: 0.01

Batch Size: Not specified

Optimizer: SGD

Train Accuracy: 54.59%

Test Accuracy: 61.82%

Configuration 1:

Epochs: 4

Learning Rate: 0.001

Batch Size: 30

Optimizer: SGD

Train Accuracy: 54.59%

Test Accuracy: 61.82%

Configuration 2:

Epochs: 4

Learning Rate: 0.01

Batch Size: Not specified

Optimizer: ADAM

Train Accuracy: 54.59%

Test Accuracy: 61.82%

Configuration 3:

Epochs: 4

Learning Rate: 0.001

Batch Size: 30

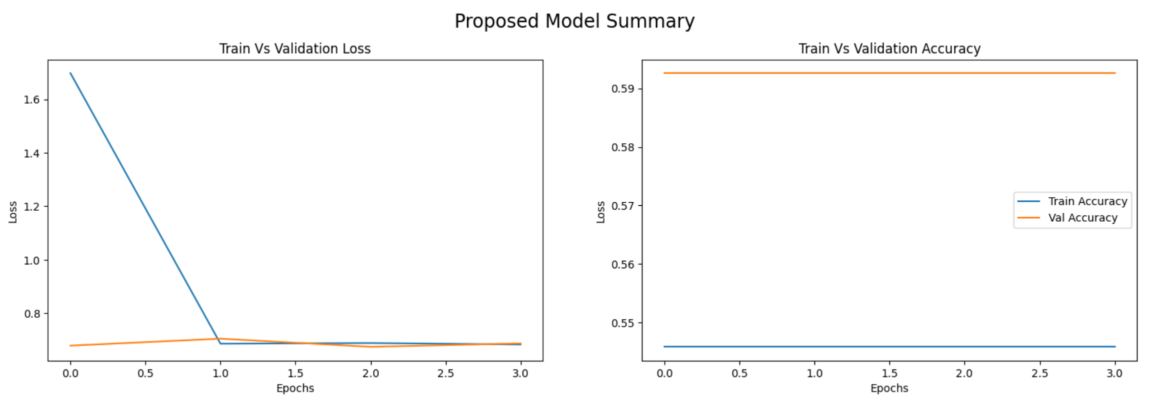
Optimizer: ADAM

Train Accuracy: 54.59%

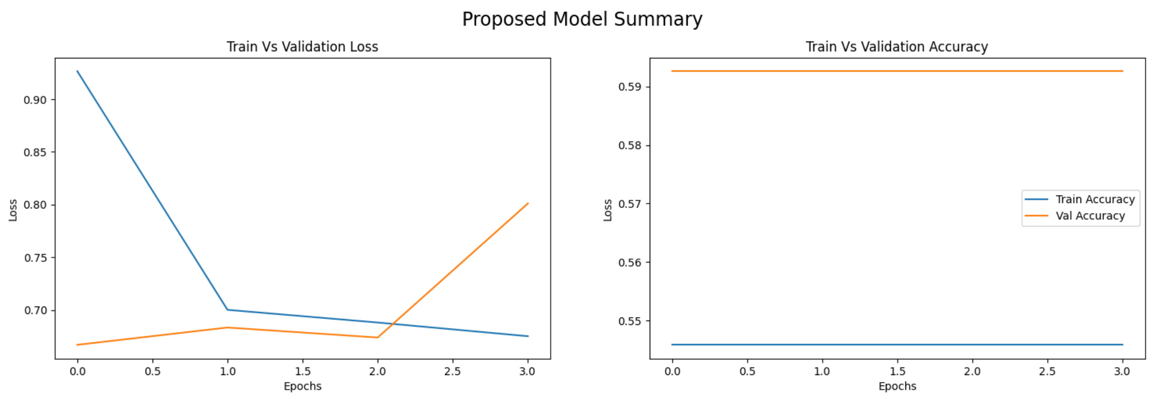
Test Accuracy: 61.82%

In all configurations, the training accuracy and test accuracy are almost identical, indicating that our model is not overfitting to the training data. The use of different optimizers (SGD and ADAM) did not result in any significant differences in the accuracy scores.

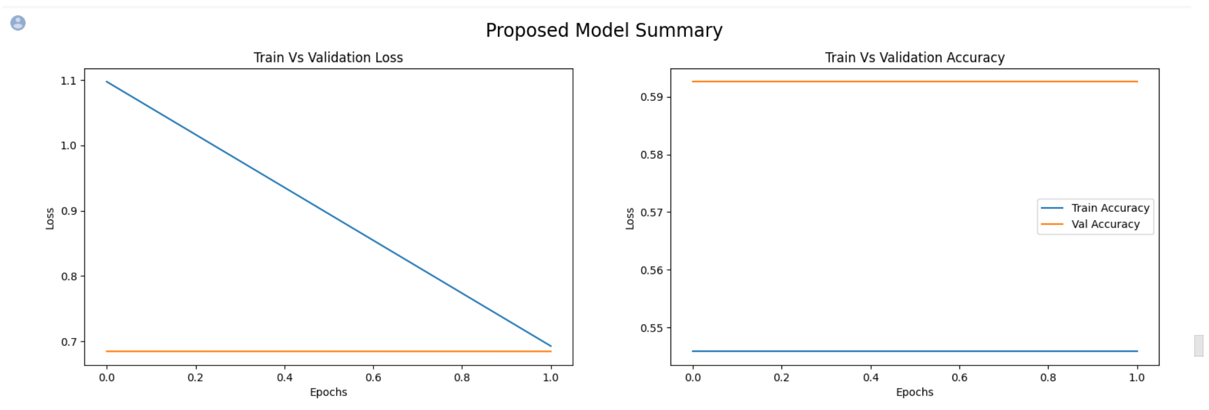
However, the use of a lower learning rate of 0.001 (Configuration 1 and 3) did not affect the accuracy scores but may have resulted in slower convergence during training. Additionally, specifying a batch size of 30 (Configuration 1 and 3) did not appear to affect the accuracy scores either.



In the above diagram we showed the loss (y-axis) over the four epochs (x-axis). The loss seems to fluctuate around a value of 0.68, with no significant decrease over the course of the four epochs. This indicates that the model is not learning very effectively and is not improving its predictions significantly as training continues. The accuracy is also consistently around 54.59%, indicating that the model is not performing well on the training data.



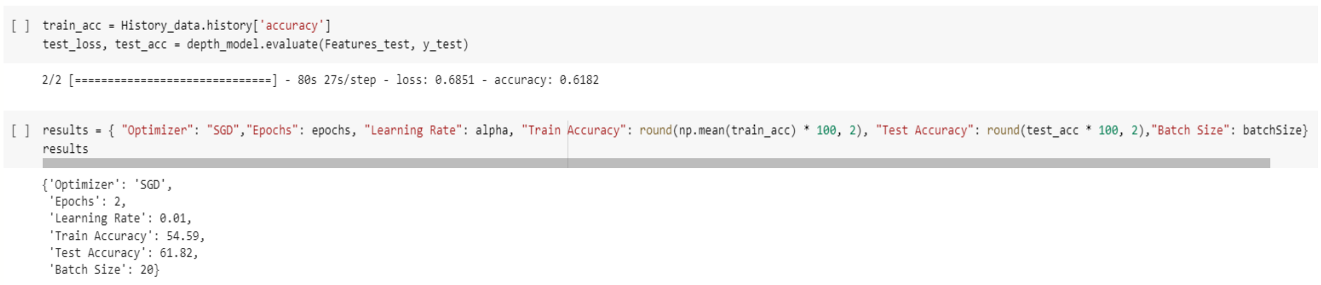
In above model it is trained for 4 epochs and model was trained using batches of size 7 and the training data was processed in steps. The model was evaluated on validation data after each epoch. The training started with a loss of 0.9264 and an accuracy of 54.59%. After 4 epochs, the final loss was 0.6751 and the final accuracy was still 54.59%. The validation loss started at 0.6669 and increased to 0.8010, while the validation accuracy remained constant at 59.26%.



By analyzing the above figures, it shows model is trained using a learning rate of 0.01 and a batch size of 20 and it shows the training and validation loss and accuracy for each epoch. The model achieved a training loss of 1.0971 and a training accuracy of 0.5459 in the first epoch, and a training loss of 0.6930 and a training accuracy of 0.5459 in the second epoch. The validation loss and accuracy remained almost the same for both epochs at 0.6849 and 0.5926, respectively.

Depth Increase:

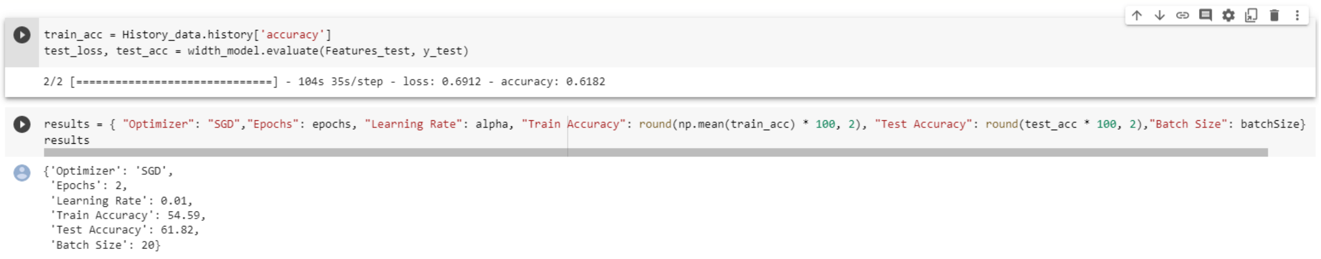
```
# Define a custom model class called "custom_modelwithdepth" that inher
its from tf.keras.Model
class custom_modelwithdepth(tf.keras.Model):

    # Define the constructor function for the model
    def __init__(self):
        # Call the constructor of the parent class
        super(custom_modelwithdepth, self).__init__()

        # Define the layers of the model as instance variables
        # using a custom layer called "Proposed_layer" with different n
umber of filters
        self.prop1 = Proposed_layer(filters=5)
        self.prop2 = Proposed_layer(filters=3)
        self.prop3 = Proposed_layer(filters=2)

        # Define the activation function layer
        self.actv = Activation(activations.relu)

        # Define the max pooling layer
        self.maxp = MaxPooling2D(pool_size=(2,2))

        # Define the flattening layer
        self.flatten = Flatten()

        # Define two dense (fully connected) layers with 16 and 8 units
 and relu activation
        self.dense1 = Dense(units=16, activation="relu")
        self.dense2 = Dense(units=8, activation="relu")
        # Define the output layer with 1 unit and softmax activation
        self.output_layer = Dense(units=1, activation="softmax")

    # Define the call function for the model
    def call(self, inputdata):

        # Apply the first custom layer to the input data
        x = self.prop1(inputdata)

        # Apply the activation function to the output of the first laye
r
        x = self.actv(x)

        # Apply the max pooling layer to the output of the activation f
unction
        x = self.maxp(x)

        # Apply the second custom layer to the output of the max poolin
g layer
        x = self.prop2(x)

        # Apply the activation function to the output of the second lay
er
        x = self.actv(x)

        # Apply the max pooling layer to the output of the activation f
unction
        x = self.maxp(x)

        # Apply the third custom layer to the output of the max pooling
 layer
        x = self.prop3(x)

        # Apply the activation function to the output of the third laye
r
        x = self.actv(x)

        # Flatten the output of the activation function
        x = self.flatten(x)

        # Apply the first dense layer to the flattened output
        x = self.dense1(x)

        # Apply the second dense layer to the output of the first dense
 layer
        x = self.dense2(x)
        # Apply the output layer to the output of the second dense laye
r
        x = self.output_layer(x)

        # Return the output of the output layer
        return x
```



```
[ ] train_acc = History_data.history['accuracy']
    test_loss, test_acc = depth_model.evaluate(Features_test, y_test)

    2/2 [==============================] - 80s 27s/step - loss: 0.6851 - accuracy: 0.6182

[ ] results = { "Optimizer": "SGD","Epochs": epochs, "Learning Rate": alpha, "Train Accuracy": round(np.mean(train_acc) * 100, 2), "Test Accuracy": round(test_acc * 100, 2),"Batch Size": batchSize}
    results

    {'Optimizer': 'SGD',
     'Epochs': 2,
     'Learning Rate': 0.01,
     'Train Accuracy': 54.59,
     'Test Accuracy': 61.82,
     'Batch Size': 20}
```

Width Increase:

```
# Define a custom Keras model called "custom_modelwithwidth"
class custom_modelwithwidth(tf.keras.Model):

    def __init__(self):
        super(custom_modelwithwidth, self).__init__()

        # Create three instances of a custom layer called "Proposed_lay
er" with different filter sizes
        self.prop1 = Proposed_layer(filters=6)
        self.prop2 = Proposed_layer(filters=4)
        self.prop3 = Proposed_layer(filters=2)

        # Define an activation function for the model to use (ReLU)
        self.actv = Activation(activations.relu)

        # Define a pooling layer for the model to use (MaxPooling2D wit
h pool size 2x2)
        self.maxp = MaxPooling2D(pool_size=(2,2))

        # Define a flatten layer for the model to use
        self.flatten = Flatten()

        # Create two dense layers with 16 and 1 output units, respectiv
ely, and ReLU and softmax activation functions
        self.dense1 = Dense(units=16, activation="relu")
        self.output_layer = Dense(units=1, activation="softmax")

    def call(self, inputdata):

        # Pass the input data through the three custom layers, with ReL
U activations and max pooling in between
        x = self.prop1(inputdata)
        x = self.actv(x)
        x = self.maxp(x)
        x = self.prop2(x)
        x = self.actv(x)
        x = self.maxp(x)
        x = self.prop3(x)
        x = self.actv(x)

        # Flatten the output from the custom layers and pass it through
the two dense layers, with ReLU and softmax activations, respectively
        x = self.flatten(x)
        x = self.dense1(x)
        x = self.output_layer(x)

        # Return the final output of the model
        return x
```

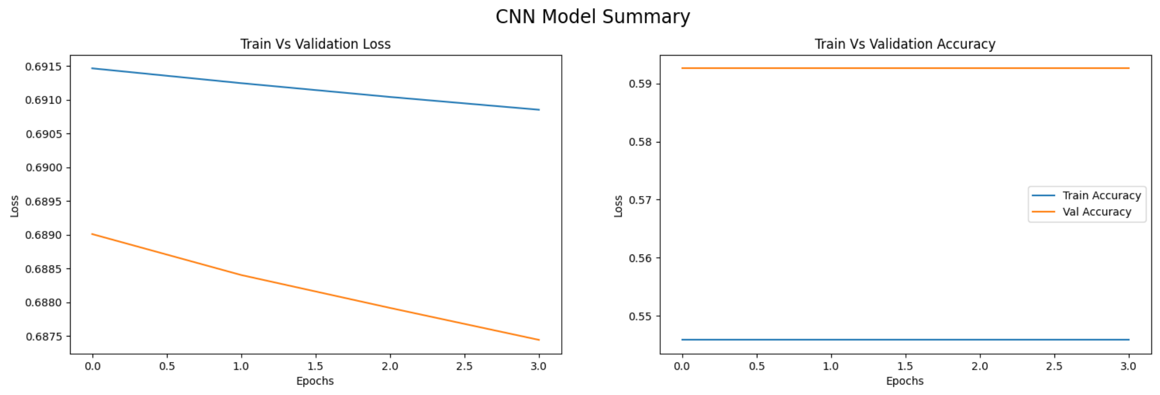


```
train_acc = History_data.history['accuracy']
test_loss, test_acc = width_model.evaluate(Features_test, y_test)

2/2 [==============================] - 104s 35s/step - loss: 0.6912 - accuracy: 0.6182
```

```
results = { "Optimizer": "SGD","Epochs": epochs, "Learning Rate": alpha, "Train Accuracy": round(np.mean(train_acc) * 100, 2), "Test Accuracy": round(test_acc * 100, 2),"Batch Size": batchSize}
results
```

```
{'Optimizer': 'SGD',
 'Epochs': 2,
 'Learning Rate': 0.01,
 'Train Accuracy': 54.59,
 'Test Accuracy': 61.82,
 'Batch Size': 20}
```

7. Report what result you achieved after training a Convolutional Neural Network of same depth [5 Marks].



```
Epoch 1/4
11/11 [==============================] - 1s 44ms/step - loss: 0.6915 - accuracy: 0.5459 - val_loss: 0.6890 - val_accuracy: 0.5926
Epoch 2/4
11/11 [==============================] - 0s 26ms/step - loss: 0.6912 - accuracy: 0.5459 - val_loss: 0.6884 - val_accuracy: 0.5926
Epoch 3/4
11/11 [==============================] - 0s 30ms/step - loss: 0.6910 - accuracy: 0.5459 - val_loss: 0.6879 - val_accuracy: 0.5926
Epoch 4/4
11/11 [==============================] - 0s 24ms/step - loss: 0.6909 - accuracy: 0.5459 - val_loss: 0.6874 - val_accuracy: 0.5926
```

```
{'Epochs': 4,
 'Learning Rate': 0.01,
 'Train Accuracy': 54.59,
 'Test Accuracy': 61.82,
 'Batch Size': 20}
```

Answer: After training the CNN with same dept we have seen that the training data was run for 4 epochs with a batch size of 20, and the results of each epoch are shown in the output. The training accuracy is around 54%, and the validation accuracy is around 59%. This indicates that the model is not performing very well on the given dataset, as the accuracy values are not significantly higher than random guessing. Further experimentation with different hyperparameters and architectures may be needed to improve the model's performance.

CNN Code:

```
normalmodel = keras.Sequential(
    [
        # Add a 2D convolution layer with 5 filters and a kernel size o
f 3x3
        layers.Conv2D(5, kernel_size=(3,3)),
        # Add a ReLU activation layer
        layers.Activation(activations.relu),
        # Add a max pooling layer with a pool size of 2x2
        layers.MaxPooling2D(pool_size=(2,2)),
        # Add another 2D convolution layer with 3 filters and a kernel
size of 3x3
        layers.Conv2D(3, kernel_size=(3,3)),
        # Add another ReLU activation layer
        layers.Activation(activations.relu),
        # Add another max pooling layer with a pool size of 2x2
        layers.MaxPooling2D(pool_size=(2,2)),
        # Add another 2D convolution layer with 2 filters and a kernel
size of 3x3
        layers.Conv2D(2, kernel_size=(3,3)),
        # Add another ReLU activation layer
        layers.Activation(activations.relu),
        # Flatten the output of the previous layer
        layers.Flatten(),
        # Add a fully connected layer with 16 units and a ReLU activati
on
        layers.Dense(16, activation="relu"),
        # Add a final output layer with 1 unit and a softmax activation
        layers.Dense(1, activation="softmax")
    ]
)
alpha = 0.01 # learning rate for the optimizer
adam = False # boolean value to specify whether to use the Adam optimiz
er or not
batchSize = 20 # size of the batches for training
epochs = 4 # number of epochs for training

# creating a directory path for storing the TensorBoard logs with curre
nt timestamp
```

```
log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-
%H%M%S")
# creating a callback for TensorBoard to log training information
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir,
histogram_freq=1)

# compiling the normalmodel with the selected optimizer, loss function
and metrics
normalmodel.compile(optimizer=selectoptimizer(alpha,adam),
                    loss='BinaryCrossentropy',
                    metrics=['accuracy'])

# training the model using the fit method with given parameters
model_history = normalmodel.fit(Features_train_copy, y_train_copy,
               epochs=epochs,
               batch_size=batchSize,
               validation_data=(Features_valid_copy, y_valid_copy),
               callbacks=[tensorboard_callback])
```

8. How to change the code to work over 1-Dimensional data? [2 Marks]

Answer:

In our code, we have a custom layer called Custom Layer that inherits from the Keras Layer class. This layer takes a 3D input tensor and applies a set of weight matrices to it to generate output feature maps.

1. The first modification that we need to make is to the shape of the weight matrix w. Since there is only one spatial dimension in the input, we need to modify the shape of the weight matrix to be (self.filters, input_shape[-1]).
2. Next, we need to update the calculation of the output feature map size. Instead of subtracting 2 from the height and width dimensions of the input, we now need to subtract 2 from the length dimension.
3. We also need to modify the nested loop that iterates over the height and width dimensions of the input to iterate over the length dimension instead.
4. Finally, we need to replace the matrix multiplication operation tf.matmul(inputs[n, feature, x1:y1, x2:y2], self.w[k, x1:y1, x2:y2]) with the vector dot product operation tf.tensordot(inputs[n, feature, x1:y1], self.w[k, x1:y1], axes=[-1, 0]). This is because we are now dealing with 1D vectors instead of 2D matrices.

9. How to change the code to work over 3-Dimensional data? [2 Marks]

Answer:

To modify the code to handle 3D inputs, we need to make a few changes to the proposed model. So, we need to modify the way we loop over the input tensor and the weight tensor to perform the convolution operation. These are the changes that can be included in for 3-D:

1. Adding an extra dimension to the weight tensor to accommodate the additional depth dimension in the input tensor and modifying the shape of the Feature maps tensor to include an extra dimension for the depth dimension.

2. Then, adding an additional loop to iterate over the depth dimension of the input tensor and updating the indexing of the input tensor and the weight tensor to include the depth dimension.
3. Then we can modify the shape of the sparse_tensor to include the additional depth dimension and update the shape of the feature maps tensor in the return statement to include the additional depth dimension.

References:

1. https://www.learndatasci.com/tutorials/convolutional-neural-networks-image-classification/
2. https://keras.io/guides/making_new_layers_and_models_via_subclassing/
3. Stochastic gradient descent: https://keras.io/api/optimizers/sgd/
4. sparse tensors : https://www.tensorflow.org/guide/sparse_tensor
5. https://keras.io/examples/vision/image_classification_from_scratch/
6. https://stackoverflow.com/questions/59400128/warningtensorflowlayer-my-modelis-casting-an-input-tensor-from-dtype-float64
7. Understanding Tensors: https://www.tensorflow.org/guide/tensor
8. TensorBoard : https://www.tensorflow.o
9. Model training APIs: https://keras.io/api/models/model_training_apis/