

Research Topics in AI - Assignment 2

Case Study Implementation - Report

Submitted By:

Kalyani Prashant Kawale
Student ID: 21237189
k.kawale1@nuigalway.ie

Anshika Mathur
Student ID: 21238296
a.mathur2@nuigalway.ie

Harshitha Bengaluru Raghuram
Student ID: 21235396
h.bengalururaghuram1@nuigalway.ie

April 1, 2022

Task 1: Contribution by Team Members

The implementation, testing, and reporting on the results of the proposed layer along with a custom max pooling layer, and coding for the model was done by Kalyani Kawale, who was also responsible for the overall formatting of the report. Anshika Mathur was responsible for the pre-processing of the data, the coding for the same and report section on pre-processing was worked upon by her. Harshitha Bengaluru Raguram worked on the tensorboard deployment and interpreting the results of the runs, which were also reported by her in the report. The hyperparameter decisions were taken after discussions on the successful and failed training results in collaboration by all team members.

Task 2: Implementation of the Proposed Layer

Custom Layer

Description:

- Based on the specified requirements for the proposed layer, a custom layer was developed by creating a subclass of the **Layer** class of **Keras API** which runs on the **Tensorflow** library [1].
- The layer named **Custom_Layer** which inherits from the **Layer** class is responsible for initializing weight matrices depending upon the number of filters or kernels configured for the layer.
- The weights are initialized using the **add_weight** method with **trainable** parameter set to true for enabling weight update during training including during backpropagation, in the build definition of the **Custom_Layer** to delay the initialization until input shape is known [1].
- The input image for each sample in the training data is then used to generate feature maps by multiplying the receptive fields of shape 3X3, taken from the same position of the input image and the weight matrices corresponding to each filter and the weighted sum is calculated for each such pair of receptive fields taken with a stride of 1 to create the output maps or the feature maps.
- The methods **tf.matmul** and **tf.reduce_sum** are used to calculate the weighted sum value for each pixel in the output map.

Code:

```
# Creating the custom layer class
class Custom_Layer(keras.layers.Layer):
    # Initializing the filters property to store the number of filters or weight
    # matrices to be used for the current layer, the filter property needs in the
    # CustomLayer constructor when being used in the model
    def __init__(self, filters):
        super(Custom_Layer, self).__init__()
        self.filters = filters
    # Initializing weights using add_weight during model building step and setting the
    # trainable parameter to true, to ensure weight updation during training
    def build(self, input_shape):
        self.w = self.add_weight(
            shape=(self.filters, input_shape[-1], input_shape[-1]),
            initializer="random_normal",
            trainable=True,
        )
    # Defining the layer computation steps
    def call(self, inputs):
        # Getting total number of samples
        num_samples = tf.shape(inputs)[0]
        # Getting the channel shape set by previous layers
        previous_maps = tf.shape(inputs)[1]
        # Initializing out_maps to save the feature_maps for each weight matrix
        output_maps = tf.zeros((num_samples, self.filters,
            tf.shape(inputs)[2]-2, tf.shape(inputs)[3]-2))
        # Setting the axes values for matrix traversal
        axis0 = tf.shape(output_maps)[0]
        axis1 = tf.shape(output_maps)[1]
        axis2 = tf.shape(output_maps)[2]
        axis3 = tf.shape(output_maps)[3]
        # For all (n) samples, for each feature/ weight matrix (k) of current layer,
        # calculating the feature map
        for n in range(axis0):
            for k in range(axis1):
                # Setting the from and to row and column indices for getting 3X3 kernel
                x1 = tf.constant(0)
                y1 = tf.constant(3)
                x2 = tf.constant(0)
                y2 = tf.constant(3)
                for i in range(axis2):
```

```

# Re-setting the from and to row indices for getting 3X3 kernel,
# after being moved to the end of matrix
x1 = tf.constant(0)
y1 = tf.constant(3)
for j in range(axis3):
    weighted_sum = tf.constant(0.0)
    # for all channels or previous feature maps (feature), calculating the weighted sum
    for feature in range(previous_maps):
        weighted_sum += tf.math.reduce_sum(tf.matmul(inputs[n, feature, x1:y1, x2:y2],
            self.w[k, x1:y1, x2:y2]))
    # using sparse tensor to generate matrix with feature map value
    # to be set into output map
    sparse_tensor = tf.SparseTensor(indices=[[n, k, i, j]],
        values=[weighted_sum],
        dense_shape=[axis0, axis1, axis2, axis3])
    # updating output map pixel for current receptive field
    output_maps = output_maps + tf.sparse.to_dense(sparse_tensor)
    # increasing row-wise stride
    x1 += 1
    y1 += 1
    # increasing column-wise stride
    x2 += 1
    y2 += 1
# Reshaping output_maps to maintain shape during training
return tf.reshape(output_maps, (axis0, axis1, axis2, axis3))

```

Custom Pooling Layer

Description:

- To design the model for training the given data-set of alpaca images, the proposed architecture in the specification was followed and max pooling was performed on the feature maps for applying down-scaling of the maps to only focus on the most important pixels.
- While, Keras provides a **MaxPooling2D** layer, the design of the custom layer in above section requires the input to be shaped by **channel first** approach, however due to the computation requirement of a GPU by the **MaxPooling2D** layer with **channel first data_format**, a custom pooling layer was designed instead using the same sub-classing method described in above section to support max-pooling on the output of the **Custom_Layer**.
- The layer named **Custom_Pooling_Layer** uses a 2X2 pool size, with a stride of 2 and reduces the feature map size to half its original dimensions, by performing max pooling on the feature maps. For matrices with odd number of dimensions the pixels in the last row and last column are ignored.
- Kera's **MaxPooling2D** layer was also experimented with, but due to the computation limits of RAM, the **Custom_Pooling_Layer** was used to perform Max Pooling in the designed network model.

Code:

```

# Creating the custom pooling layer class by inheriting from Keras Layer class
class Custom_Pooling_Layer(keras.layers.Layer):
    # Initializing the Custom_Pooling_layer
    def __init__(self):
        super(Custom_Pooling_Layer, self).__init__()
    # Defining the max pooling computation to be performed on each feature map
    def call(self, inputs):
        # Initializing the pooling output size, which is half of original input size
        pool_output_size = tf.cast(tf.shape(inputs)[2]/2, tf.int32)
        # Initializing pooled_maps with the max pooling output map shape,
        # with channel first data format
        pooled_maps = tf.zeros((tf.shape(inputs)[0], tf.shape(inputs)[1],
            pool_output_size, pool_output_size))
        # Initializing the range of indices with a stride size of 2 to be traversed
        # to get the 2X2 sub-matrix on which pooling is performed
        end = tf.shape(inputs)[2]

```

```

# Checking if input feature map has odd or even dimensions,
# ignoring last column and row if shape is odd
if end%2 != 0:
    end = tf.shape(inputs)[2] - 1
indices = tf.range(0, end, 2)
# Setting the axes values for matrix traversal
axis0 = tf.shape(pooled_maps)[0]
axis1 = tf.shape(pooled_maps)[1]
axis2 = tf.shape(pooled_maps)[2]
axis3 = tf.shape(pooled_maps)[3]
# For each sample (n), for each feature k, performing max pooling
all_pools = tf.TensorArray(tf.float32, size=0, dynamic_size=True)
for n in range(axis0):
    n_pool = tf.TensorArray(tf.float32, size=0, dynamic_size=True)
    for k in range(axis1):
        pool = tf.TensorArray(tf.float32, size=0, dynamic_size=True)
        for index1 in indices:
            for index2 in indices:
                # max pooling applied to receptive field of size 2X2
                pool = pool.write(pool.size(),
                                tf.reduce_max(inputs[n][k][index1:index1+2, index2:index2+2]))
            # saving pooled output for each feature map
            n_pool = n_pool.write(n_pool.size(), tf.reshape(pool.stack(),
                (pool_output_size, pool_output_size)))
        # saving pooled output for each sample in the data-set
        all_pools = all_pools.write(all_pools.size(), tf.stack(n_pool.stack()))
# Assigning the final max-pooling output in tensorflow compatible form
pooled_maps = tf.stack(all_pools.stack())
# Reshaping the output for maintaining consistent shape size while training
return tf.reshape(pooled_maps, (axis0, axis1, axis2, axis3))

```

Task 3: Implementation of Layer and Complete Model

Building the Complete Neural Network

Description:

- The neural network designed below has **3 Proposed Layers, 2 Max Pooling Layers**, and follows the sequence of layers mentioned in the assignment specification with an addition of the Keras **Flatten** layer for reducing the 2 dimensional inputs into 1 dimensional vector, however, due to computation limitations, the **weight matrices** used for each of the 3 layers are **5, 3, and 2** respectively, instead of 16, 12, 8.
- The **image size** used to train the model was **28x28**, which the system is able to **successfully train** with, however, increasing the size any further takes over 2 hours for one epoch.
- The neural network was tested with 16, 12, 8 layers, but the system could not complete the training process with the limited resources.
- The system is able to train with 16, 12, 8 layers when image size is reduced to 8x8 pixels, however, this results in huge loss of information and the neural network with 3 Proposed Layers and 2 Max Pooling layer need a minimum of 28x28 sized image, as any lesser sized image gets reduced to single pixelated image 1x1 by the time it reaches the last Custom/Proposed Layer, resulting in unsuccessful training.

Code:

```

# Building the neural network model according to the assignment specification
def build_model():
    # Initializing the model using Sequential model class of Keras
    model = keras.models.Sequential()
    # Adding custom layer with 5 weight matrices to generate 5 feature maps
    model.add(Custom_Layer(filters=5))
    # Adding Relu Activation
    model.add(layers.Activation(activations.relu))
    model.add(Custom_Pooling_Layer()) # Max Pooling Custom Layer
    # Adding custom layer with 3 weight matrices to generate 3 feature maps

```

```

model.add(Custom_Layer(filters=3))
# Adding Relu Activation
model.add(layers.Activation(activations.relu))
model.add(Custom_Pooling_Layer()) # Max Pooling Custom Layer
# Adding custom layer with 2 weight matrices to generate 2 feature maps
model.add(Custom_Layer(filters=2))
# Adding Relu Activation
model.add(layers.Activation(activations.relu))
# Adding Flatten layer
model.add(layers.Flatten())
# Adding Fully Connected layer with 16 neurons
model.add(layers.Dense(units=16, activation="relu"))
# Adding Output Layer with one neuron and activation softmax
# for predicting class probability
model.add(layers.Dense(units=1, activation="softmax"))
return model

# Initialising results list to store accuracy
# of each run with different hyperparameters
results = []
# Initialising histories list to store run history
# of each run with different hyperparameters
# to be used in plotting learning curves
histories = []
# Building and saving the neural network in model_nn
# using build_model method defined above
model_nn = build_model()

```

Neural Network Model Implementation & Results

Defining Common Method to Compile, Train and Evaluate Model:

The following method `model_run` defines the steps to,

1. Compile the neural network model with specified hyperparameters using Keras Model's **compile** method with **BinaryCrossEntropy** loss as the dataset contains two classes *alpaca* and *not alpaca*.
2. Perform training on the image data-set using Keras Model's **fit** method.
3. Evaluate the model against Test data using Keras Model's **evaluate** method.
4. Save the accuracy, loss and history for the run.

```

# Method to compile, train and evaluate the model
def model_run(alpha, batch_size, epoch, adam=False):
    # Initialising run_result dictionary
    run_result = {"Learning Rate": alpha, "Batch Size": batch_size, "Epoch": epoch,
                  "Train Accuracy": 0.0, "Test Accuracy": 0.0}
    # Checking and setting the optimizer to be used for training
    if adam:
        optimizer = tf.keras.optimizers.Adam(learning_rate=alpha)
    else:
        optimizer = tf.keras.optimizers.SGD(learning_rate=alpha)
    # Compiling the model with given optimizer
    model_nn.compile(optimizer=optimizer,
                     loss='BinaryCrossentropy',
                     metrics=['accuracy'])
    # Initialising and setting up directory to store run files for tensorboard
    log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
    tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)
    # Training the model
    model_history = model_nn.fit(X_train, y_train,
                                epochs=epoch,
                                batch_size=batch_size,
                                validation_data=(X_valid, y_valid),

```

```

callbacks=[tensorboard_callback])
# Saving the training accuracy obtained from the History object returned by fit method
print("Saving Training Data Accuracy...")
train_acc = model_history.history['accuracy']
# Evaluating test data using the trained model
print("Evaluating Test Data:")
test_loss, test_acc = model_nn.evaluate(X_test, y_test)
# Saving results
run_result["Train Accuracy"] = f'{round(np.mean(train_acc) * 100, 4)} %'
run_result["Test Accuracy"] = f'{round(test_acc * 100, 4)} %'

return model_history, run_result

```

Model Implementation & Computation Graph:

The model created using **build_model** method was trained with all data samples using the **model_run** method, with the default parameter settings (batch size None and Learning Rate 0.01) of Stochastic Gradient Descent optimizer provided by Keras **Optimizer** class as follows,

```

# checking the model for successful implementation
# Default hyperparameter values for SGD optimizer and Keras Model Fit function
model_history, run_result = model_run(alpha=0.01, batch_size=None, epoch=5)

# Saving the run results for testing the layer and model implementation
histories.append(model_history)
results.append(run_result)
print("For the default settings of Stochastic Gradient Descent
Optimizer the designed model has following accuracies:")
print(f"Training Accuracy: {run_result['Train Accuracy']}")
print(f"Test Accuracy: {run_result['Test Accuracy']}")

```

<p>For the default settings of Stochastic Gradient Descent Optimizer the designed model has following accuracies: Training Accuracy: 55.0459% Test Accuracy: 56.3636%</p>

Figure 1: Results for training of the neural network model designed.

- The model when trained with SGD optimizer with default values for 5 epochs was able to train and evaluate all the images divided into train, validation and test data-sets of size 28X28 and gave for 5 epochs **average training accuracy of 55.0459%, average validation accuracy of 62.96%, and the test accuracy of 56.3636%.**
- The following command was used to upload all the run logs onto **Tensorboard**,

```

!tensorboard dev upload --logdir ./logs \
  --name "RTAI\Assignment2-All Experiments" \
  --description "Results of runs for general sgd configuration, \
  different hyperparameters for 3 layers of custom layer, \
  increase in width, decrease in depth and cnn run" \
  --one\_shot

```

- The computation graph generated by **Tensorflow** for the designed model is as follows,

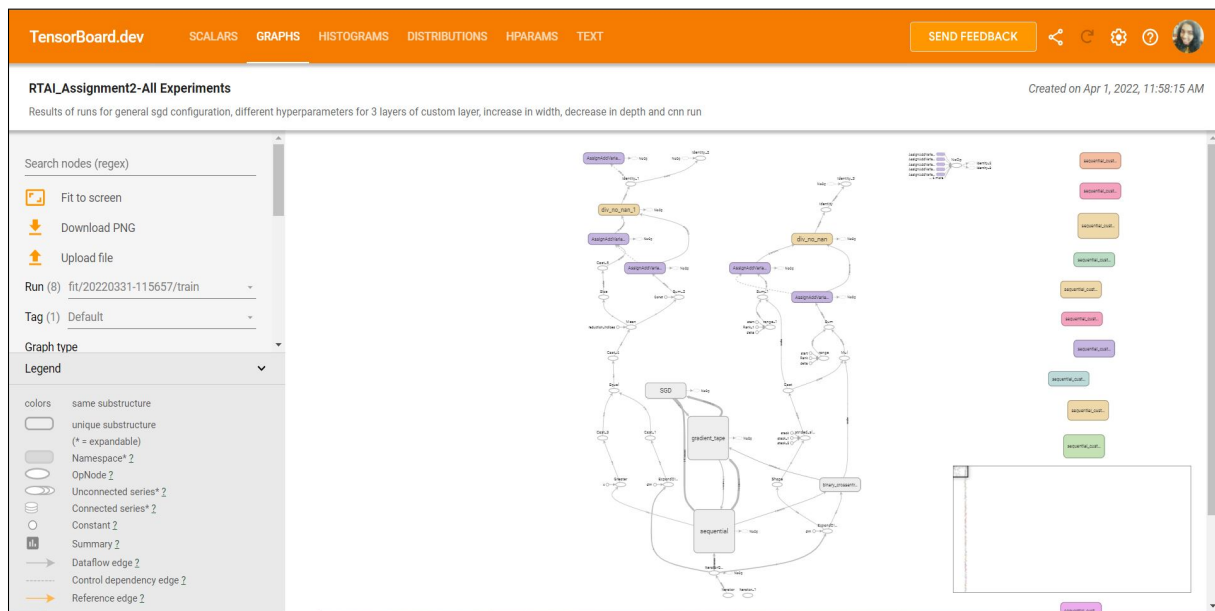


Figure 2: Model Computation Graph in Tensorboard

- The expanded view of the sequential layer of the computation graph generated by **Tensorflow** for the designed model is as follows,

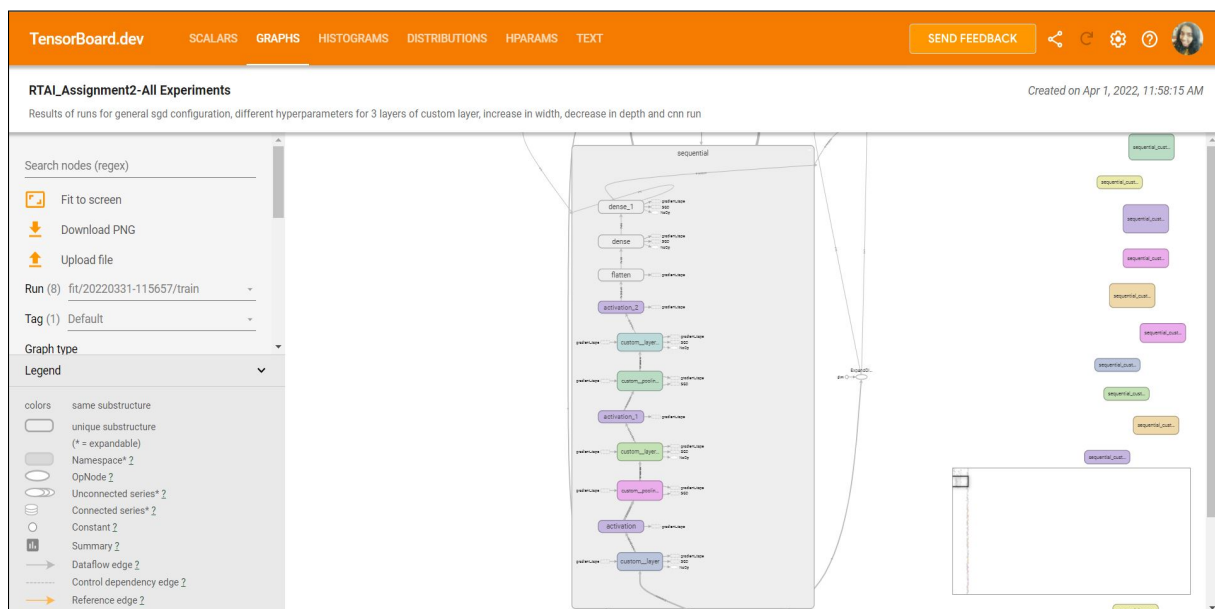


Figure 3: Model Layers as generated in Tensorboard

The results and graphs for all runs performed uploaded to Tensorboard can be found at: [RTAI_Assignment2-All Experiments](#)

Task 4: Pre-Processing & Splitting of Data-Set

Loading & Gray-scaling

Description:

- All the images in the data-set were converted to gray-scale while reading with the **imread** function of cv library.
- Gray-scaling helped in reducing the complexity of the code, as only a single color channel had to be handled while processing and training, which helped with the limited computation power available for the processing.

Code:

The data-set of images was loaded and converted into gray-scale using the following code,

```
import numpy as np
import matplotlib.pyplot as plt
import os
import cv2 # Used for image processing operations
import tensorflow as tf

# Setting the directory names from which images are to be loaded
Data_Dir = "./dataset"
Categories = ["alpaca", "not alpaca"]

# Loading of an image from dataset to test if loading works
for category in Categories:
    path = os.path.join(Data_Dir, category) # Path to alpaca or not alpaca dir
    print(path)
    for Img in os.listdir(path):
        # Reading image into Img_array array with conversion into grayscale
        Img_array = cv2.imread(os.path.join(path,Img), cv2.IMREAD_GRAYSCALE)
        # Displaying the image to ensure that it has been converted into gray-scale
        plt.imshow(Img_array, cmap = "gray")
        plt.show()
        break
    break
```

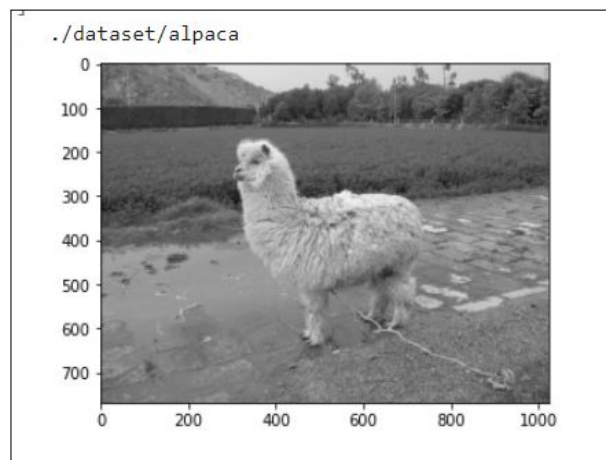


Figure 4: Alpaca Image Loaded from Data-Set and Converted into Gray-Scale Image

Image Resizing

Description:

- The neural network requires numerical inputs which are provided in the form of equal sized matrices of pixel values for each image in the data-set.
- The image size used for the given task was 28x28, all the images were resized to the said dimension using **resize** function of **cv2 library**.
- Apart from 28x28 the system was also tested on images of size 8x8, 16x16, 32x32. The neural network worked well on 8x8 and 16x16 images, but these cases the layers used in the network had to be limited to a single CustomLayer and single CustomPoolingLayer due to the downscaling of the image size that happens as the image moves through the

network. With data-set size fixed at 32x32 dimension, which could have provided better refined features, the neural network ran extremely slowly, hence the final size determined to enable running the system for more than one epoch was 28x28.

Code:

The following code is used to resize the images to 28x28 size,

```
# Resizing of the image to 28x28
Img_Size = 28
New_Array = cv2.resize(Img_array, (Img_Size,Img_Size))
# Checking image distortion after resizing
plt.imshow(New_Array, cmap = 'gray')
plt.show()
```

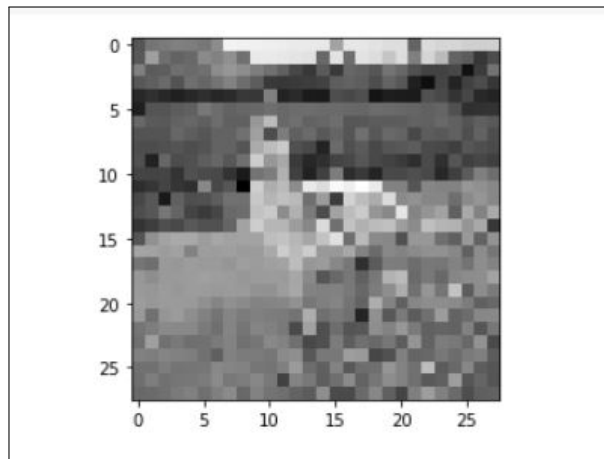


Figure 5: Image from figure 4 resized to 28x28

Splitting into Train, Test, Validation Sets

Description:

- Following cells combines the code from above gray-scale and re-size sections to read all images from the data-set, converts the data-set array into tensorflow compatible format (float32 from uint32) and splits **70%** of the data into **training set**, **15%** into **validation set** and **15%** into **testing set**.
- The validation data is configured into the training by setting the **validation_data** of the **fit** method to help avoid over-fitting.
- The **testing set** was used to evaluate the trained model.
- The limited data-set did not provide good accuracy after running the model, which could have been improved by using data augmentation methods such as zooming the image, rotating or flipping the image, however, due to the already limited computation resources, the model was trained on the 327 original re-sized images.

Code:

Loading and splitting of pre-processed data into Train, Test and Validation is done using the following code,

```
# Initialising training data array
Training_Data = []
# Method to load and preprocess all images
```

```

def create_TrainData():
    for category in Categories:
        # Path to alpaca or not alpaca directories
        path = os.path.join(Data_Dir, category)
        # Assigning number to classes using their index,
        # i.e., alpaca = [0] not alpaca = [1]
        Class_Num = Categories.index(category)
        # Iterating through all the images
        for Img in os.listdir(path):
            try:
                # Reading and converting the images to gray-scale
                Img_array = cv2.imread(os.path.join(path,Img), cv2.IMREAD_GRAYSCALE)
                # Performing resizing operation
                New_Array = cv2.resize(Img_array, (Img_Size,Img_Size))
                # Appending new array and class to the list
                Training_Data.append([New_Array, Class_Num])
            # Exception is raised for cases such as if the img is distorted
            except Exception as e:
                pass

create_TrainData()

# Shuffling of data for better training
import random
random.shuffle(Training_Data)

X = [] # Features
y = [] # Labels
# Splitting the data into features and labels
for features, labels in Training_Data:
    X.append(features)
    y.append(labels)
# Converting the arrays to numpy
X = np.array(X)
y = np.array(y)
# Creating a copy to be used for CNN model
X_copy = np.copy(X)
# Re-shaping the images to get channel first format
X = np.array(X).reshape(-1, 1, Img_Size, Img_Size)
# Casting images to tensorflow supported format
X = tf.cast(X, tf.float32)

# Setting set sizes to get 2/3rd (70%) data for training, and remaining 1/3rd (30%)
# is further divided into 15% validation and 15% test
dataset_len = len(X)
valid_test_size = int(dataset_len / 3)
valid_size = int(valid_test_size / 2)

X_train = X[: (dataset_len - valid_test_size)]
y_train = y[: (dataset_len - valid_test_size)]

X_valid = X[(dataset_len - valid_test_size):
((dataset_len - valid_test_size) + valid_size)]
y_valid = y[(dataset_len - valid_test_size):

```

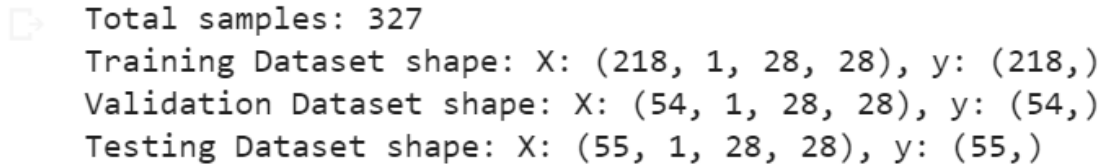
```

((dataset_len - valid_test_size) + valid_size)]

X_test = X[((dataset_len - valid_test_size) + valid_size):]
y_test = y[((dataset_len - valid_test_size) + valid_size):]

print(f"Total samples: {dataset_len}")
print(f"Training Dataset shape: X: {X_train.shape}, y: {y_train.shape}")
print(f"Validation Dataset shape: X: {X_valid.shape}, y: {y_valid.shape}")
print(f"Testing Dataset shape: X: {X_test.shape}, y: {y_test.shape}")

```



```

Total samples: 327
Training Dataset shape: X: (218, 1, 28, 28), y: (218,)
Validation Dataset shape: X: (54, 1, 28, 28), y: (54,)
Testing Dataset shape: X: (55, 1, 28, 28), y: (55,)

```

Figure 6: Data-Set Splits

Task 5: Training Model with Different Hyperparameters

Description:

- The model created in task 3 was trained with different hyperparameters to evaluate its performance.
- Due to the computation complexity of the created **Custom_Layer** the epochs had to be limited to 5, but the model was trained and evaluated with the different configurations of optimizer, learning rate to test difference in convergence, and batch sizes.
- The optimizers used are **Stochastic Gradient Descent** and **Adam**, provided by Keras **Optimizer** class.
- The **learning rates** tested are **0.05** and **0.1**, these were selected as the default 0.01 learning rate of SGD used in task 3 was reducing the loss very slowly, while it could have been due to less number of epochs and small data-set, increased learning rates were instead used to verify the effect of learning rates on the convergence.
- The **batch sizes** tested with are **20** and **50**, again due to the small number of images being trained upon.

Code:

```

# Setting the learning rate alpha and batch sizes
alphas = [0.05, 0.1]
batch_sizes = [20, 50]

```

Setting 1:

```

# 0.05, 20 hyperparameter values for SGD optimizer
model_history, run_result = model_run(alpha=alphas[0],
batch_size=batch_sizes[0], epoch=5, adam=False)
histories.append(model_history)
results.append(run_result)

```

```

Epoch 1/5
11/11 [=====] - 636s 57s/step - loss: 0.6921 - accuracy: 0.5505 - val_loss: 0.6881 - val_accuracy: 0.6296
Epoch 2/5
11/11 [=====] - 589s 54s/step - loss: 0.6914 - accuracy: 0.5505 - val_loss: 0.6857 - val_accuracy: 0.6296
Epoch 3/5
11/11 [=====] - 585s 54s/step - loss: 0.6907 - accuracy: 0.5505 - val_loss: 0.6839 - val_accuracy: 0.6296
Epoch 4/5
11/11 [=====] - 616s 57s/step - loss: 0.6899 - accuracy: 0.5505 - val_loss: 0.6821 - val_accuracy: 0.6296
Epoch 5/5
11/11 [=====] - 574s 53s/step - loss: 0.6895 - accuracy: 0.5505 - val_loss: 0.6807 - val_accuracy: 0.6296
Saving Training Data Accuracy...
Evaluating Test Data:
2/2 [=====] - 61s 21s/step - loss: 0.6878 - accuracy: 0.5636

```

Figure 7: 5 Epoch Run for Hyper-parameter setting 1

Setting 2:

```

# 0.05, 20 hyperparameter values for ADAM optimizer
model_history, run_result = model_run(alpha=alphas[0],
batch_size=batch_sizes[0], epoch=5, adam=True)
histories.append(model_history)
results.append(run_result)

```

```

Epoch 1/5
11/11 [=====] - 607s 54s/step - loss: 0.6954 - accuracy: 0.5505 - val_loss: 0.6786 - val_accuracy: 0.6296
Epoch 2/5
11/11 [=====] - 615s 57s/step - loss: 0.6877 - accuracy: 0.5505 - val_loss: 0.6691 - val_accuracy: 0.6296
Epoch 3/5
11/11 [=====] - 649s 60s/step - loss: 0.6888 - accuracy: 0.5505 - val_loss: 0.6701 - val_accuracy: 0.6296
Epoch 4/5
11/11 [=====] - 618s 57s/step - loss: 0.6899 - accuracy: 0.5505 - val_loss: 0.6673 - val_accuracy: 0.6296
Epoch 5/5
11/11 [=====] - 588s 54s/step - loss: 0.6887 - accuracy: 0.5505 - val_loss: 0.6725 - val_accuracy: 0.6296
Saving Training Data Accuracy...
Evaluating Test Data:
2/2 [=====] - 63s 22s/step - loss: 0.6855 - accuracy: 0.5636

```

Figure 8: 5 Epoch Run for Hyper-parameter setting 2

Setting 3:

```

# 0.1, 50 hyperparameter values for SGD optimizer
model_history, run_result = model_run(alpha=alphas[1],
batch_size=batch_sizes[1], epoch=5, adam=False)
histories.append(model_history)
results.append(run_result)

```

```

Epoch 1/5
5/5 [=====] - 737s 142s/step - loss: 0.6881 - accuracy: 0.5505 - val_loss: 0.6715 - val_accuracy: 0.6296
Epoch 2/5
5/5 [=====] - 845s 174s/step - loss: 0.6887 - accuracy: 0.5505 - val_loss: 0.6713 - val_accuracy: 0.6296
Epoch 3/5
5/5 [=====] - 899s 165s/step - loss: 0.6881 - accuracy: 0.5505 - val_loss: 0.6705 - val_accuracy: 0.6296
Epoch 4/5
5/5 [=====] - 742s 146s/step - loss: 0.6884 - accuracy: 0.5505 - val_loss: 0.6718 - val_accuracy: 0.6296
Epoch 5/5
5/5 [=====] - 741s 147s/step - loss: 0.6881 - accuracy: 0.5505 - val_loss: 0.6724 - val_accuracy: 0.6296
Saving Training Data Accuracy...
Evaluating Test Data:
2/2 [=====] - 65s 23s/step - loss: 0.6854 - accuracy: 0.5636

```

Figure 9: 5 Epoch Run for Hyper-parameter setting 3

Setting 4:

```

# 0.1, 50 hyperparameter values for ADAM optimizer
model_history, run_result = model_run(alpha=alphas[1],
batch_size=batch_sizes[1], epoch=5, adam=True)
histories.append(model_history)
results.append(run_result)

```

```

Epoch 1/5
5/5 [=====] - 766s 147s/step - loss: 0.6920 - accuracy: 0.5505 - val_loss: 0.6690 - val_accuracy: 0.6296
Epoch 2/5
5/5 [=====] - 736s 145s/step - loss: 0.6873 - accuracy: 0.5505 - val_loss: 0.6802 - val_accuracy: 0.6296
Epoch 3/5
5/5 [=====] - 729s 144s/step - loss: 0.6893 - accuracy: 0.5505 - val_loss: 0.6774 - val_accuracy: 0.6296
Epoch 4/5
5/5 [=====] - 739s 146s/step - loss: 0.6919 - accuracy: 0.5505 - val_loss: 0.6667 - val_accuracy: 0.6296
Epoch 5/5
5/5 [=====] - 729s 144s/step - loss: 0.6891 - accuracy: 0.5505 - val_loss: 0.6674 - val_accuracy: 0.6296
Saving Training Data Accuracy...
Evaluating Test Data:
2/2 [=====] - 63s 22s/step - loss: 0.6850 - accuracy: 0.5636

```

Figure 10: 5 Epoch Run for Hyper-parameter setting 4

Task 6: Analysis and Display of Experiment Results

Following table provides a summary of the Average Training Accuracy and Loss obtained by training the model designed in task 3 with different parameters using the fit method with training and validation data sets, the Testing Accuracy and Loss are computed by evaluating each model setting on the 15% testing data-set,

Overall Results

	Optimizer	Batch Size	Epoch	Learning Rate	Test Accuracy	Test Loss	Avg. Train Accuracy	Avg. Train Loss
0	SGD (Default)	nan	5	0.01	56.3636%	0.6921	55.0459%	0.6927
1	SGD	20	5	0.05	56.3636%	0.6878	55.0459%	0.6907
2	ADAM	20	5	0.05	56.3636%	0.6855	55.0459%	0.6901
3	SGD	50	5	0.1	56.3636%	0.6854	55.0459%	0.6882
4	ADAM	50	5	0.1	56.3636%	0.685	55.0459%	0.6899

Figure 11: Result Table for Each Run Including the Initial Default SGD Run

Individual Results and Learning Curves:

The model with 3 Custom_layers with 5, 3 and 2 filters/ weight matrices were trained with 5 different parameter settings, each ran for 5 epochs, the results of which are as follows, **Default**

Setting used in Task 3:

- The SGD optimizer with its default parameter settings of learning rate set at 0.01 and batch size None, gave **average training accuracy of 55.0459%, average validation accuracy of 62.96%, and the test accuracy of 56.3636%.**
- It was observed that the training and validation accuracy dropped from around 71% at the beginning of the run to 55.0459% for each epoch.
- However, the difference could be seen in loss where the average training loss was obtained of 0.6927 and testing loss settled at 0.6921.

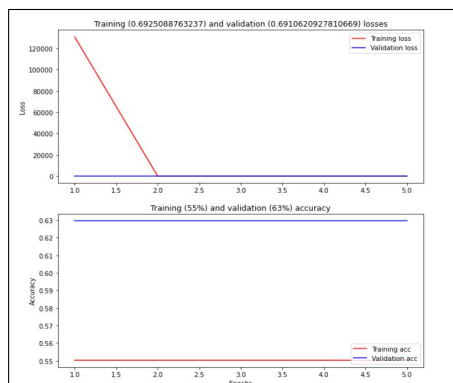


Figure 12: Default Setting SGD Learning Curve

Setting 1:

- Setting 1 ran the SGD optimizer with learning rate of 0.05 and batch size 20, which gave average training accuracy, average validation accuracy, and the test accuracy same as default setting.
- Similar to above setting it was observed that the training and validation accuracy dropped from around 71% at the beginning of the run to 55.0459% for each epoch.
- Here the average training loss was obtained of 0.6907 and testing loss settled at 0.6878 which dropped compared to the default setting.

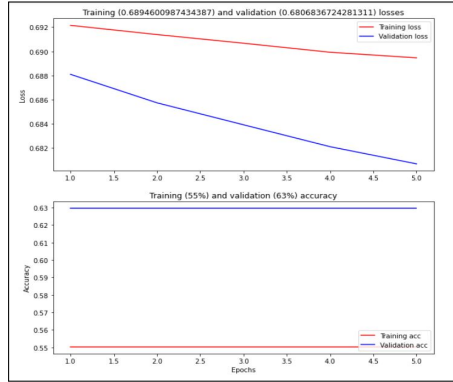


Figure 13: SGD, Alpha 0.05 and Batch Size 20 Learning Curve

Setting 2:

- Setting 1 ran the ADAM optimizer with learning rate of 0.05 and batch size 20, which gave same accuracy results as previous settings.
- The average training loss and test loss dropped further to 0.6901 and 0.6855 respectively compared to the above two settings.

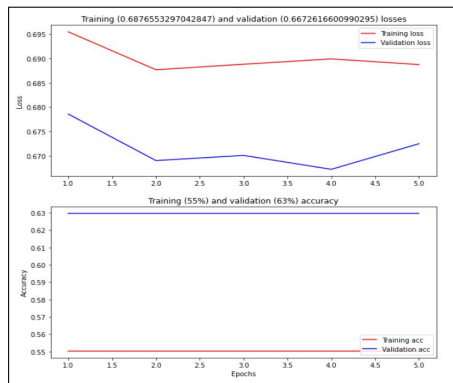


Figure 14: ADAM, Alpha 0.05 and Batch Size 20 Learning Curve

Setting 3:

- Setting 3 ran the SGD optimizer again but with increased learning rate of 0.1 and batch size 50, because the convergence was very slow, which still gave same accuracy results as previous settings.
- The average training loss and test loss however dropped even further to 0.6882 and 0.6854 respectively compared to all the above settings.

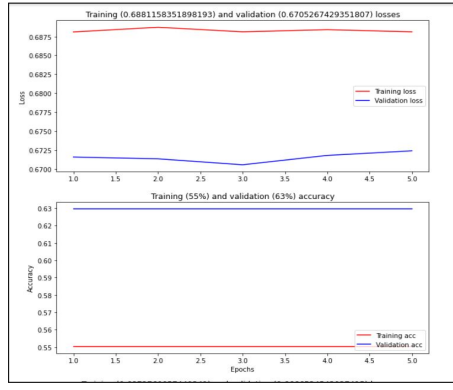


Figure 15: SGD, Alpha 0.1 and Batch Size 50 Learning Curve

Setting 4:

- Setting 4 ran the ADAM optimizer with learning rate of 0.1 and batch size 50, giving same accuracy results.
- The average training loss of 0.6899 was lower than setting 1 and 2, but greater than setting 3 and test loss however dropped even further to 0.6850.

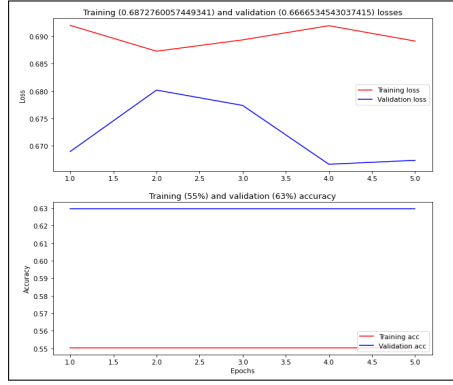


Figure 16: ADAM, Alpha 0.1 and Batch Size 50 Learning Curve

- Overall, all configurations gave almost similar results, which could be due to smaller dataset and less number of epochs, as well as smaller difference in the number of filters in each proposed layer, increasing these factors may provide improved results.
- It was also observed that the runs with SGD had a steady drop in loss as compared to the ADAM optimizer.
- Due to the lack of change in accuracy, the drop in loss has been considered as a measure to select the best configuration (SGD with 0.1 learning rate and 50 batch size) for the next task.

Task 7: Width, Depth Variations and Comparison with CNN

The best parameters selected in task 6, i.e. SGD optimizer with 0.1 alpha and 50 batch size was applied to train each of the following configurations,

Increase in Width

Description:

- Due to computation complexity to increase the width of network designed in task 3, only the number of filters in the first proposed layer was increased from 5 to 6.

- Other configurations ran, but failed to complete training were increase in each proposed layer by 1, i.e. 6, 4, and 3 filters and two layers with increased filter size and one with size of 2.

Code:

```
# Increasing width
model_width = keras.models.Sequential()
# Increased filters from 5 to 6
model_width.add(Custom_Layer(filters=6))
model_width.add(layers.Activation(activations.relu))
model_width.add(Custom_Pooling_Layer())
model_width.add(Custom_Layer(filters=3))
model_width.add(layers.Activation(activations.relu))
model_width.add(Custom_Pooling_Layer())
model_width.add(Custom_Layer(filters=2))
model_width.add(layers.Activation(activations.relu))
model_width.add(layers.Flatten())
model_width.add(layers.Dense(units=16, activation="relu"))
model_width.add(layers.Dense(units=1, activation="softmax"))
# Assign best hyperparameters
model_width.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.1),
                    loss='BinaryCrossentropy',
                    metrics=['accuracy'])

log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir,
histogram_freq=1)

model_history = model_width.fit(X_train, y_train,
                               epochs=5,
                               batch_size=50,
                               validation_data=(X_valid, y_valid),
                               callbacks=[tensorboard_callback])

print("Saving Training Data Accuracy...")
train_acc = model_history.history['accuracy']
train_loss = model_history.history['loss']
print("Evaluating Test Data:")
test_loss, test_acc = model_width.evaluate(X_test, y_test)
```

Run Result:

```
Epoch 1/5
5/5 [=====] - 951s 187s/step - loss: 29766390004500660224.0000 - accuracy: 0.5642 - val_loss: 0.6921 - val_accuracy: 0.5370
Epoch 2/5
5/5 [=====] - 864s 171s/step - loss: 0.6908 - accuracy: 0.5642 - val_loss: 0.6914 - val_accuracy: 0.5370
Epoch 3/5
5/5 [=====] - 874s 173s/step - loss: 0.6894 - accuracy: 0.5642 - val_loss: 0.6910 - val_accuracy: 0.5370
Epoch 4/5
5/5 [=====] - 881s 173s/step - loss: 0.6888 - accuracy: 0.5642 - val_loss: 0.6906 - val_accuracy: 0.5370
Epoch 5/5
5/5 [=====] - 942s 189s/step - loss: 0.6877 - accuracy: 0.5642 - val_loss: 0.6905 - val_accuracy: 0.5370
Saving Training Data Accuracy...
Evaluating Test Data:
2/2 [=====] - 77s 26s/step - loss: 0.6828 - accuracy: 0.6000
```

Figure 17: 5 Epoch Run for Increased Width

Learning Curve:

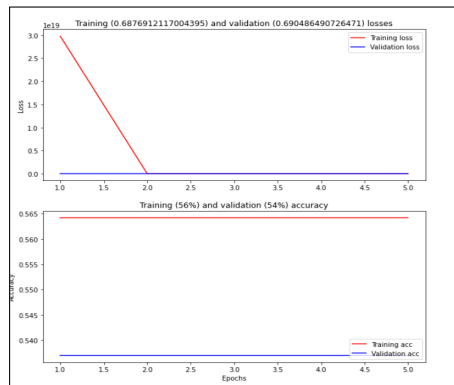


Figure 18: Learning Curve for Increased Width Configuration

Decrease in Depth

Description:

- The image size selected for reducing complexity of 28x28 cannot be used as an input for more than 3 Custom_Layer and 2 Max Pooling layers, because the size drops to single pixel, hence instead the depth was reduced to a single Custom_Layer with one Max Pooling layer.

Code:

Decreasing depth, testing the model with one custom layers instead of 3

```
model_depth = keras.models.Sequential()
model_depth.add(Custom_Layer(filters=5))
model_depth.add(layers.Activation(activations.relu))
model_depth.add(Custom_Pooling_Layer())
model_depth.add(layers.Flatten())
model_depth.add(layers.Dense(units=16, activation="relu"))
model_depth.add(layers.Dense(units=1, activation="softmax"))
# Assign best hyperparameters
model_depth.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.1),
                    loss='BinaryCrossentropy',
                    metrics=['accuracy'])
```

```
log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)
```

```
model_history = model_depth.fit(X_train, y_train,
                               epochs=5,
                               batch_size=50,
                               validation_data=(X_valid, y_valid),
                               callbacks=[tensorboard_callback])
```

```
print("Saving Training Data Accuracy...")
train_acc = model_history.history['accuracy']
train_loss = model_history.history['loss']
print("Evaluating Test Data:")
test_loss, test_acc = model_depth.evaluate(X_test, y_test)
```

Run Result:

```
Epoch 1/5  
5/5 [=====] - 654s 127s/step - loss: 10514130272256.0000 - accuracy: 0.5642 - val_loss: 2567042.7500 - val_accuracy: 0.5370  
Epoch 2/5  
5/5 [=====] - 625s 125s/step - loss: 791052.1875 - accuracy: 0.5642 - val_loss: 0.6923 - val_accuracy: 0.5370  
Epoch 3/5  
5/5 [=====] - 617s 122s/step - loss: 0.6912 - accuracy: 0.5642 - val_loss: 0.6914 - val_accuracy: 0.5370  
Epoch 4/5  
5/5 [=====] - 633s 126s/step - loss: 0.6896 - accuracy: 0.5642 - val_loss: 0.6912 - val_accuracy: 0.5370  
Epoch 5/5  
5/5 [=====] - 634s 126s/step - loss: 0.6894 - accuracy: 0.5642 - val_loss: 0.6909 - val_accuracy: 0.5370  
Saving Training Data Accuracy...  
Evaluating Test Data:  
2/2 [=====] - 53s 18s/step - loss: 0.6856 - accuracy: 0.6000
```

Figure 19: 5 Epoch Run for Decreased Depth

Learning Curve:

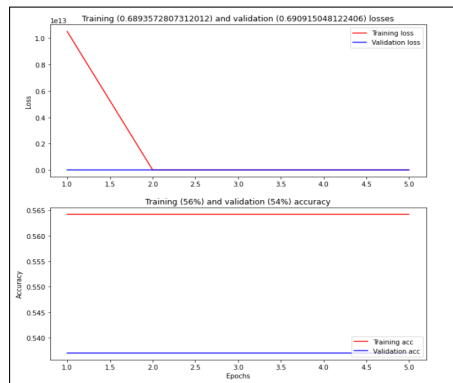


Figure 20: Learning Curve for Decreased Depth Configuration

Comparing the Model by replacing Custom_Layer with Conv2D Keras Layer

Description:

- To compare the exact model configuration with a CNN network, the Custom_Layer in task 3 model was replaced by **Conv2D** layer and Custom_Pooling Layer was replaced by **MaxPooling2D** layer.
- The 3 Conv2D layers had 5, 3 and 2 filters respectively.

Code:

```
# CNN  
model_cnn = keras.models.Sequential()  
model_cnn.add(layers.Conv2D(5, kernel_size=(3,3)))  
model_cnn.add(layers.Activation(activations.relu))  
model_cnn.add(layers.MaxPooling2D(pool_size=(2,2)))  
model_cnn.add(layers.Conv2D(3, kernel_size=(3,3)))  
model_cnn.add(layers.Activation(activations.relu))  
model_cnn.add(layers.MaxPooling2D(pool_size=(2,2)))  
model_cnn.add(layers.Conv2D(2, kernel_size=(3,3)))  
model_cnn.add(layers.Activation(activations.relu))  
model_cnn.add(layers.Flatten())  
model_cnn.add(layers.Dense(units=16, activation="relu"))  
model_cnn.add(layers.Dense(units=1, activation="softmax"))  
  
# Assign best hyperparameters  
model_cnn.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.1),  
                  loss='BinaryCrossentropy',
```

```

metrics=['accuracy'])

log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)

model_history = model_cnn.fit(X_train_copy, y_train_copy,
                              epochs=5,
                              batch_size=50,
                              validation_data=(X_valid_copy, y_valid_copy),
                              callbacks=[tensorboard_callback])

print("Saving Training Data Accuracy...")
train_acc = model_history.history['accuracy']
train_loss = model_history.history['loss']
print("Evaluating Test Data:")
test_loss, test_acc = model_cnn.evaluate(X_test_copy, y_test_copy)

```

Run Result:

```

Epoch 1/5
5/5 [=====] - 2s 174ms/step - loss: 151.3446 - accuracy: 0.5642 - val_loss: 0.6922 - val_accuracy: 0.5370
Epoch 2/5
5/5 [=====] - 0s 66ms/step - loss: 0.6912 - accuracy: 0.5642 - val_loss: 0.6913 - val_accuracy: 0.5370
Epoch 3/5
5/5 [=====] - 0s 76ms/step - loss: 0.6895 - accuracy: 0.5642 - val_loss: 0.6907 - val_accuracy: 0.5370
Epoch 4/5
5/5 [=====] - 0s 60ms/step - loss: 0.6881 - accuracy: 0.5642 - val_loss: 0.6907 - val_accuracy: 0.5370
Epoch 5/5
5/5 [=====] - 0s 60ms/step - loss: 0.6879 - accuracy: 0.5642 - val_loss: 0.6905 - val_accuracy: 0.5370
Saving Training Data Accuracy...
Evaluating Test Data:
2/2 [=====] - 0s 11ms/step - loss: 0.6827 - accuracy: 0.6000

```

Figure 21: 5 Epoch Run for CNN

Learning Curve:

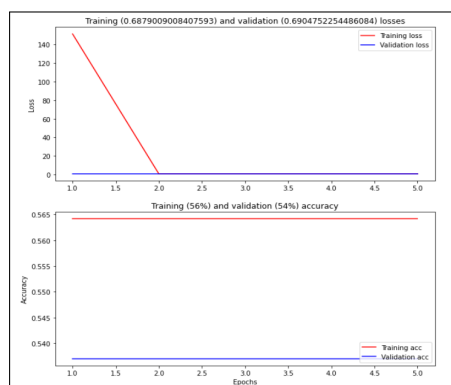


Figure 22: Learning Curve for CNN Configuration

Overall Comparison Results:

- The average training accuracy and test accuracy for all the above three configurations was **56.422%** and **60.0%** which is better than all settings described in task 5 and task 6.
- Apart from accuracy the other results were more or less similar to previous settings.
- The validation accuracy however dropped to approximately 53% which is far smaller than the average validation accuracy obtained by task 5 and 6 settings.

- It was also observed that the training loss for the first epoch for each of these configurations of the model was extremely high, which could be related to the colab settings, rather than the algorithm configurations.

Following table summarizes the results for changes in width, depth and comparison with CNN network,

	Configuration	Learning Rate	Batch Size	Epoch	Train Accuracy	Test Accuracy	Train Loss	Test Loss
0	Increase in Width	0.1	50	5	56.422%	60.0%	5.95328e+18	0.6828
1	Decrease in Depth	0.1	50	5	56.422%	60.0%	2.10283e+12	0.6856
2	Conv2D Layer	0.1	50	5	56.422%	60.0%	30.8203	0.6827

Figure 23: Increased Width, Decreased Depth and CNN Configuration Results

Following resources were referred to perform the given tasks,

References

- [1] François Chollet. (2020). Making new layers and models via subclassing. Available at: https://keras.io/guides/making_new_layers_and_models_via_subclassing/
- [2] Convolutional Neural Networks. (2020). Available at: <https://www.ibm.com/cloud/learn/convolutional-neural-networks>
- [3] François Chollet. (2020). Image classification from scratch. Available at: https://keras.io/examples/vision/image_classification_from_scratch/
- [4] SGD. Available at: <https://keras.io/api/optimizers/sgd/>
- [5] Adjust Single Value within Tensor – TensorFlow. Available at: <https://stackoverflow.com/questions/34685947/adjust-single-value-within-tensor-tensorflow>
- [6] Working with sparse tensors Available at: https://www.tensorflow.org/guide/sparse_tensor
- [7] tf.keras.layers.Reshape. Available at: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Reshape
- [8] WARNING:tensorflow:Layer my_model is casting an input tensor from dtype float64 to the layer's dtype of float32, which is new behavior in TensorFlow 2. Available at: <https://stackoverflow.com/questions/59400128/warningtensorflowlayer-my-model-is-casting-an-input-tensor-from-dtype-float64>
- [9] Introduction to Tensors. Available at: <https://www.tensorflow.org/guide/tensor>
- [10] tf.range. Available at: https://www.tensorflow.org/api_docs/python/tf/range
- [11] tf.math.reduce_max. Available at: https://www.tensorflow.org/api_docs/python/tf/math/reduce_max
- [12] InaccessibleTensorError when appending to list, while looping. Available at: <https://github.com/tensorflow/tensorflow/issues/37512>
- [13] Get started with TensorBoard. Available at: https://www.tensorflow.org/tensorboard/get_started#:text=TensorBoard%20is%20a%20tool%20for,dimensional%20space%2C%20and%20much%20more.
- [14] Model training APIs. Available at: https://keras.io/api/models/model_training_apis/