Project 2

March 6, 2023

0.1 Importing Libraries

[ ]:
```
# For this project, stride is always 1 and padding is set to valid.
import numpy as np
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras import layers
from tensorflow.keras import optimizers
```

```
2023-03-06 16:05:53.129781: I tensorflow/core/platform/cpu_feature_guard.cc:193]
This TensorFlow binary is optimized with oneAPI Deep Neural Network Library
(oneDNN) to use the following CPU instructions in performance-critical
operations:  SSE4.1 SSE4.2
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
```

1 Neuron Convolution Class

[ ]:
```
#Only for the neurons in convolutional layer
class NeuronConv:
    def __init__(self, activation, kernelSize, lr, weights,bias=None):
        self.output = None
        self.input = None
        self.delta = None

        self.activation = activation
        self.kernelSize = kernelSize
        self.lr = lr
        self.weights = weights # 3D in the right shape
        self.bias = 0 if bias is None else bias# scalar: a single value



    # This method returns the activation of the net
    # Since the activation functions in all convolution layers are sigmoid in␣
 ↪this project, here we only provide one way.
    def activate(self,activation,net):
        if activation == 'sigmoid':
```

```python
            output = 1 / (1 + np.exp(-1 * net))
        else:
            output = net

        return output


    # Calculate the output of the neuron should save the input and output for
↪back-propagation.
    def calculate(self,input): #input should have the same size as kernel.
↪(channels,H,W)

        dimension =np.shape(self.weights)
        channels = dimension[0]

        #create an empty matrix to contain the internal steps
        matrix = np.zeros(self.weights.size).reshape(dimension)

        for c in range(channels): #channels
            for i in range(self.kernelSize): # Row
                for j in range(self.kernelSize): #Column
                    matrix[c,i,j] = input[c,i,j] * self.weights[c,i,j]
        output = np.sum(matrix)+self.bias

        self.output=self.activate(self.activation,output)

        return self.output


    # This method returns the derivative of the activation function with
↪respect to the net
    def activationderivative(self):
        if self.activation =='sigmoid':
            act_der = self.output*(1-self.output)
        else: #linear
            act_der=1
        return act_der

    def calcpartialderivative(self, wtimesdelta):  # wtimesdelta is a matrix
        self.delta = wtimesdelta * self.activationderivative()
        return self.delta
```

2 Convolution Layer Class

[ ]:
```
# A Convolutional layer
class ConvolutionalLayer:

    # initialize with the number of neurons in the layer, their activation,the␣
 ↪input size,the learning rate
    # inputDimension (C,H,W)  3D matrix
    # bias will be a 2D matrix (C, 1)
    # weight should be 4D (fliter sequence, c,H,W)
    def __init__(self, numOfKernels, kernelSize, activation, inputDimension,lr,␣
 ↪weights, bias=None):

        self.inputDimension = inputDimension
        self.kernelSize = kernelSize
        self.weights = np.random.random((numOfKernels,self.
 ↪inputDimension[0],kernelSize, kernelSize)) if weights is None else weights.
 ↪reshape(numOfKernels,self.inputDimension[0],kernelSize, kernelSize)
        self.bias = np.zeros((numOfKernels)) if bias is None else bias
        self.lr = lr

        #the output dimension = (channels,height, width)
        # Although in this project width is equal to heights, we can still␣
 ↪write them separately for future more generalized application;
        # channels = numOfKernels
        # width = (width_input-width_kernel)/stride +1
        self.width = int((inputDimension[2]-kernelSize)/1+1)
        self.height = int((inputDimension[1]-kernelSize)/1+1)
        self.channels = numOfKernels # output channels != input channels
        self.numOfOutputNeuron = int(self.width*self.height*self.channels)␣
 ↪#total # of output neurons
        self.numOfOutputNeuron_OneLayer = int(self.width*self.height)#  # of␣
 ↪output neurons in one feature map

        self.outputDimension =np.array((self.channels,self.height,self.width))␣
 ↪#Will be the dimension of input of next layer


        # Store the output of this layer, matrix )
        self.output = np.zeros(self.numOfOutputNeuron).reshape((self.channels,␣
 ↪self.height,self.width))

        # create the output layer
        self.neurons= [] # list of list [[output of filter 1],[Output of filter␣
 ↪2],...]
```

```python
        for i in range(self.channels): # equal to the filter number
            neurons= []
            for j in range(self.numOfOutputNeuron_OneLayer):
                neurons.append(NeuronConv(activation,kernelSize,lr, self.
↪weights[i], self.bias[i]))
            self.neurons.append(neurons)

    def get_outputDimension(self):
        return self.outputDimension

    def calculate(self,input): # input is 3D ( channelsOfInput,widthOfInput,␣
↪heightOfInput)
        self.input = input
        total_output =[]
        # Separate the entire input into a list of input for each output neuron
        input_lst = []
        for i in range(self.height): #get the data of each row
            for j in range(self.width): #column
                input_lst.append(self.input[:, i:(i+self.kernelSize),j:(j+self.
↪kernelSize)])

        for i in range(self.channels):
            output_layer=[]
            for j in range(self.numOfOutputNeuron_OneLayer):
                output_layer.append(self.neurons[i][j].calculate(input_lst[j]))
            output_matrix_onelayer = np.array(output_layer).reshape((self.
↪height, self.width))
            total_output.append(output_matrix_onelayer)

        self.output = np.array(total_output)
        return self.output

    # This method calculates the partial derivative for each weight and returns␣
↪the delta*w to be used in the previous layer
    # Compared with the fully connected neuron, the update of the learning␣
↪parameter occurs at the ConvolutionalLayer
    def calcwdeltas(self, wtimesdelta):  # wtimesdelta is a matrix
        #Same logic as calculate the output
        total_delta=[]

        for i in range(self.channels):# Num of kernel
            delta_layer =[]
            for j in range(self.numOfOutputNeuron_OneLayer):
```

```
            delta_layer.append(self.neurons[i][j].
↪calcpartialderivative(wtimesdelta[i,j//self.width,j%self.width]))

        delta_layer = np.array(delta_layer).reshape((self.height,self.
↪width))

        total_delta.append(delta_layer)
    self.delta = np.array(total_delta)

    #Add padding to delta Since padding is 'same'.
    # (new-kernealsize)/stride +1 = initial
    # new = (initial-1)*stride + kernelSize
    new_width = self.inputDimension[2]-1+self.kernelSize
    new_height = self.inputDimension[1]-1+self.kernelSize

    # flipped the weights of each layer per filter
    new_weights = np.zeros(self.weights.size).reshape(np.shape(self.
↪weights))
    for i in range(np.shape(self.weights)[0]): #different filters
        for j in range(np.shape(self.weights)[1]): #different channels␣
↪corresponding to the input channels
            new_weights[i,j]= self.weights[i,j].T


    lst2=[]
    for i in range(self.inputDimension[0]):
        lst = []
        for j in range(np.shape(self.delta)[0]):
            new_padding_matrix = np.zeros((1,new_height,new_width))
            new_padding_matrix[0,self.kernelSize-1:(self.kernelSize-1+self.
↪height),self.kernelSize-1:(self.kernelSize-1+self.width) ] = self.delta[j,:,:
↪]
            lst.append(ConvolutionalLayer(1, self.kernelSize, 'linear', np.
↪shape(new_padding_matrix), self.lr,
                               np.expand_dims(new_weights[j,i],axis=0),␣
↪bias=None).calculate(new_padding_matrix))
        lst = np.array(lst)

        lst2.append(np.sum(lst, axis=0).reshape(self.inputDimension[1],self.
↪inputDimension[2]))
    lst2= np.array(lst2)
    wtimesdelta = lst2

    #update weights
    self.updateweight()
```

```
        return wtimesdelta


    #numOfKernels, kernelSize, activation, inputDimension, lr, weights, bias):
    def updateweight(self):
        total_deltas=[]#4D #of filters, channels

        for i in range(self.delta.shape[0]):
            new_deltas = []
            for j in range(self.inputDimension[0]):
                new_deltas.append(self.delta[i,:,:])
            new_deltas = np.array(new_deltas)
            total_deltas.append(new_deltas)
        total_deltas =np.array(total_deltas)

        for i in range(self.delta.shape[0]):
            self.weights[i] -= self.lr * ConvolutionalLayer(1, self.delta.
↪shape[-1],'linear', self.inputDimension, self.lr, np.
↪expand_dims(total_deltas[i],axis=0), None).calculate(self.input)

        self.bias -= self.lr * np.sum(self.delta)

    def print_weights(self,layer_idx):
        print('layer {} is a Conv layer, weights shape = {} and bias shape = {}.
↪ Weights and bias are as follows:'.format(layer_idx, self.weights.shape,␣
↪self.bias.shape))
        print(self.weights)
        print(self.bias)
```

3 MaxPoolingLayer Class

[ ]:
```
class MaxPoolingLayer:
    def __init__(self, kernelSize, inputDimension): #inputDimension (channels,␣
↪height, width)

        self.kernelSize = kernelSize
        self.inputDimension = np.array(inputDimension)
        self.initial_input_number = np.prod(inputDimension)
        # For this project, stride is always the same as the filter size; no␣
↪padding.
        self.height = int((inputDimension[1]-kernelSize)/kernelSize + 1)
        self.width = int((inputDimension[2]-kernelSize)/kernelSize + 1)
        self.channels =int(inputDimension[0])
        total_neuron = int(self.channels * self.height * self.width)
```

```
        self.output = np.zeros(self.channels * self.height * self.width).
↪reshape((self.channels,self.height,self.width))

        #Install the max value position (max position=1; others =0 )
        self.position = np.zeros(self.initial_input_number).reshape(self.
↪inputDimension) #For backpropagation

        # will be the inputdimension of next layer
        self.dimension = np.shape(self.output)

    def get_outputDimension(self):
        return self.output.shape

    def calculate(self, input): #inputDimension (channels, height, width)

        for c in range(self.channels):
            for i in range(self.height):
                for j in range(self.width):
                     matrix =input[c, int(i*self.kernelSize):int((i+1)*self.
↪kernelSize), int(j*self.kernelSize):int((j+1)*self.kernelSize)]
                     self.output[c,i,j] = matrix.max()
                     #The sequence number of the max number in each kernel
                     k=np.argmax(matrix[c])# Will be the number from 0 to␣
↪kernelSize*kernelSize-1
                     #the position of each selected max value can be writen as␣
↪below
                     self.position[c,int((i*self.kernelSize + k) // self.
↪kernelSize), int((j*self.kernelSize + k)% self.kernelSize)] = 1
        return self.output

    def calcwdeltas(self, wtimesdelta):
        expanded_matrix = np.zeros(self.initial_input_number).reshape(self.
↪inputDimension)

        for c in range(self.channels):
            for i in range(self.height):
                for j in range(self.width):
                    expanded_matrix[c,int(i*self.kernelSize):int((i+1)*self.
↪kernelSize), int(j*self.kernelSize):int((j+1)*self.kernelSize)]=␣
↪wtimesdelta[c,i,j]

        wtimesdelta = np.zeros(self.initial_input_number).reshape(self.
↪inputDimension)
        for c in range(self.inputDimension[0]):
            for i in range( self.inputDimension[1]):
                for j in range(self.inputDimension[2]):
```

```
                    wtimesdelta[c,i,j] = expanded_matrix[c,i,j] *self.
↪position[c,i,j]

        return wtimesdelta

    def print_weights(self, layer_idx):
        print('layer {} is a Max Pooling layer without weights'.
↪format(layer_idx))
```

4 Flatten Layer Class

[ ]:
```
class FlattenLayer:
    def __init__(self,inputDimension):
        self.inputDimension = inputDimension
        self.output= None

    def get_outputDimension(self):
        return np.prod(self.inputDimension)

    def calculate(self, input):
        self.output = input.flatten()
        self.output = np.expand_dims(self.output,0)
        return self.output

    def calcwdeltas(self, wtimesdelta):
        wtimesdelta = wtimesdelta.reshape(self.inputDimension)
        return wtimesdelta

    def print_weights(self, layer_idx):
        print('layer {} is a Flatten layer without weights'.format(layer_idx))
```

5 Neuron Class

[ ]:
```
class Neuron:
    # initilize neuron with activation type, number of inputs, learning rate,␣
↪and possibly with set weights
    def __init__(self, activation, input_num, lr, weights=None):
        self.activation = activation
        self.input_num = input_num
        self.lr = lr

        # if weights is not set, random set [0,1) in shape of (1, input_num)
        if weights is None:
            self.weights = np.random.randn(input_num)
            self.bias = np.random.randn()
```

```
        else:
            self.weights = weights[:-1]
            self.bias = weights[-1]

        # some self defined terms
        self.output = None
        self.input = None
        self.delta = None


    def print_weight(self, layer_idx, neu_idx):
        print('layer {} is a FC layer, weights shape = {} and bias shape = {}.␣
↪Weights and bias are as follows:'.format(layer_idx, self.weights.shape, self.
↪bias.shape))
        print(self.weights)
        print(self.bias)

    # This method returns the activation of the net
    def activate(self, net):
        if self.activation == 0:  # linear
            output = net
        else:  # logistic-sigmoid
            output = 1 / (1 + np.exp(-1 * net))
        return output

    # Calculate the output of the neuron should save the input and output for␣
↪back-propagation.
    def calculate(self, input):
        self.input = input
        net = np.dot(input, self.weights) + self.bias
        self.output = self.activate(net)  # a scalar/ a number

        return self.output

    # This method returns the derivative of the activation function with␣
↪respect to the net
    def activationderivative(self):
        # errorder is the derivative of loss/error with respect to the output
        if self.activation == 1:  # logistic
            act_der = self.output * (1 - self.output)
        else:  # linear
            act_der = 1
        return act_der  # a scalar

    # This method calculates the partial derivative for each weight and returns␣
↪the delta*w to be used in the previous layer
    def calcpartialderivative(self, wtimesdelta):  # wtimesdelta is a matrix
```

```
        # Be careful about the last hidden layer wtimesdelta should be␣
↪error_der times actder only
        delta = wtimesdelta * self.activationderivative()

        self.cal_der_w = delta * self.input  # One node only have one delta␣
↪value.
        self.cal_der_b = delta * 1

        # update wtimesdelta
        wtimesdelta = delta * self.weights

        # update weights
        self.updateweight()
        return wtimesdelta  # output a vetctor

    # Simply update the weights using the partial derivatives and the learning␣
↪weight
    def updateweight(self):
        # print('updateweight',self.weights.shape, self.cal_der_w.shape)
        self.weights -= self.lr * self.cal_der_w[0]
        self.bias -= self.lr * self.cal_der_b
```

6 Fully Connected Layer Class

[ ]:
```
# A fully connected layer
class FullyConnected:
    # initialize with the number of neurons in the layer, their activation,the␣
↪input size,
    # the learning rate and a 2d matrix of weights (or else initilize randomly)
    def __init__(self, numOfNeurons, activation, input_num, lr, weights=None):
        self.numOfNeurons = numOfNeurons
        self.neurons = []  # neurons works as a new list containing all Neurons
        self.output_shape = numOfNeurons
        if weights is None:
            for i in range(numOfNeurons):
                self.neurons.append(Neuron(activation, input_num, lr))
        else:
            for i in range(numOfNeurons):
                self.neurons.append(Neuron(activation, input_num, lr, weights))

    def get_outputDimension(self):
        return (self.numOfNeurons, 1)

    def print_weights(self, layer_idx):
        for neuron_index in range(self.numOfNeurons):
            self.neurons[neuron_index].print_weight(layer_idx, neuron_index)
```

```
    # calcualte the output of all the neurons in the layer and return a vector␣
↪with those values (go through the neurons and call the calcualte() method)
    def calculate(self, input):  # input:(2,)
        self.output_vector = []  # list
        for i in range(self.numOfNeurons):
            self.output_vector.append(self.neurons[i].calculate(input))
        return np.array(self.output_vector)  # array

    # given the next layer's w*delta, should run through the neurons calling␣
↪calcpartialderivative() for each (with the correct value),
    # sum up its ownw*delta, and then update the wieghts (using the␣
↪updateweight() method).
    # should return the sum of w*delta. - A vector
    def calcwdeltas(self, wtimesdelta):
        lst = []

        for i in range(self.numOfNeurons):
            lst.append(self.neurons[i].calcpartialderivative(wtimesdelta[i]))
        lst = np.array(lst)

        wtimesdelta = np.sum(lst, axis=0)
        return wtimesdelta
```

7 Neural Network Class

```
[ ]: class NeuralNetwork:
    # Initialize the instance instance variables 'inputSize', 'activation',␣
↪'loss', 'lr' and 'weights'
    def __init__(self, inputSize, activation, loss, lr, weights=None):
        self.loss = loss # Set loss function
        self.activation = activation # Set activation function
        self.lr = lr # Set learning rate
        self.layers = [] # Initialize list of layers
        self.inputSize = inputSize # Set input size


    # Add a new layer to the neural network and it takes two arguments, 'type'␣
↪and 'layer_param'
    def addLayer(self,type,layer_param):

        # If this is the first layer
        if self.layers == []:
            input_shape = self.inputSize # Set input shape to input size
        else:
```

```
            input_shape = self.layers[-1].get_outputDimension() # Otherwise,␣
↪set input shape to output shape of previous layer

        # If the layer type is convolutional
        if type == 'conv':
            self.layers.append(ConvolutionalLayer(layer_param['numOfKernels'],␣
↪layer_param['kernelSize'],layer_param['activation']
                                                 ,input_shape, self.lr,␣
↪layer_param['weights'],layer_param['bias'])) # Add a convolutional layer to␣
↪the list of layers

        # If layer type is max pooling
        elif type == 'max':
            self.layers.append(MaxPoolingLayer(layer_param['poolingSize'],␣
↪input_shape)) # Add a max pooling layer to the list of layers

        # If layer type is flatten
        elif type == 'fl':
            self.layers.append(FlattenLayer(input_shape)) # Add a flattern␣
↪layer to the list of layers

        # Otherwise, assume fully connected layer
        else:
            self.layers.append(FullyConnected(layer_param['numOfNeurons'],␣
↪layer_param['activation'], input_shape, self.lr,
                                                     layer_param['weights']))

    # Feed the input data through the network and obtain the output of the last␣
↪layer
    def calculate(self, input):
        for j in range(input.shape[0]):
            tmp_output = input[j] # Initialize output to current input

            # For each layer in the list of layers, calculate the output of the␣
↪layer
            for i in range(len(self.layers)):
                tmp_output = self.layers[i].calculate(tmp_output)

        return tmp_output


    # Return the calculated loss
    def calculateloss(self, yp, y):
        error = 0 # Initial error

        # For each prediction in the prediction array
```

```
        for i in range(yp.shape[0]):
            if self.loss == 0: # if the loss function is mean squared error
                error += 1 / 2 * np.sum((yp[i] - y[i]) ** 2)
            else:  # Otherwise, assume cross-entropy loss function
                error += np.sum(np.nan_to_num(-y[i] * np.log(yp[i]) - (1 -␣
↪y[i]) * np.log(1 - yp[i])))

        return error / yp.shape[0] # Return the average error over all␣
↪predictions

    # Compute the derivative of the loss with respect to the final output of␣
↪the neural network, 'yp', given the true target values 'y'.
    def lossderiv(self, yp, y):

        if self.loss == 0: # if loss function is mean squared error
            error_der = 0
            for i in range(yp.shape[0]):
                error_der += (yp[i] - y[i])
        else: # Otherwise, assume cross-entropy loss function

            if self.activation == 0:  # Add softmax
                error_der = 0
                for i in range(yp.shape[0]):
                    error_der += ((yp[i] - y[i]) / yp[i] / (1 - yp[i])) * yp[i]
            else:
                error_der = 0
                for i in range(yp.shape[0]):
                    error_der = np.nan_to_num(((yp[i] - y[i]) / yp[i] / (1 -␣
↪yp[i])))

        return (1 / yp.shape[0]) * error_der

    # Train the neural network model
    def train(self, x, y, num_epochs=1000):
        error = []
        for epoch_index in range(num_epochs):
            output_last = self.calculate(x)
            yp = output_last
            if error == []:
                past_loss = 1e+10
            else:
                past_loss = current_loss

            current_loss = self.calculateloss(yp, y)

            if np.abs(current_loss - past_loss) <= 1e-8:
               break
```

```
            error.append(current_loss)
            error_der = self.lossderiv(yp, y)* 2
            wtimesdelta = error_der
            for layer_index in range(len(self.layers)-1, -1, -1):
                wtimesdelta = self.layers[layer_index].calcwdeltas(wtimesdelta)

        return yp, error[-1]


    def print_weights(self):
        for layer_index in range(len(self.layers)):
            self.layers[layer_index].print_weights(layer_index)
```

8 Verifying Results with Keras

In this section, we will verify the results of our implementation with Keras.

8.1 Example 1

[ ]:
```
def example1():

    # set random seed
    np.random.seed(0)

    # create input and output for keras model
    keras_input = np.random.rand(1, 5, 5, 1)
    keras_output = np.random.rand(1)

    our_input = keras_input.reshape(1, 5, 5)
    our_label = [keras_output]

    keras_conv_w = np.random.rand(3, 3, 1, 1)
    keras_conv_b = np.random.rand(1)
    keras_fc_w = np.random.rand(9, 1)
    keras_fc_b = np.random.rand(1)

    # create input and output for our model
    our_conv_w = keras_conv_w.reshape(1, 1, 3, 3)
    our_conv_b = keras_conv_b.reshape(1)
    our_fc_w = keras_fc_w.reshape(9)
    our_fc_b = keras_fc_b.reshape(1)

    our_fc_w = np.hstack((our_fc_w, our_fc_b))
```

```
    return keras_input,keras_output,keras_conv_w, keras_conv_b, keras_fc_w,␣
↪keras_fc_b, \
           our_input,our_label,our_conv_w, our_conv_b,our_fc_w
```

8.1.1 Keras Model

[ ]:
```
keras_input, keras_output, keras_conv_w, keras_conv_b, keras_fc_w, keras_fc_b, \
our_input, our_label, our_conv_w, our_conv_b, our_fc_w=example1()

input = keras_input
output = keras_output
print('Output before backprop:', output)

print('----- Keras Model after Backpropagation (Example 1)-----')
model = Sequential()
model.add(layers.Conv2D(1, 3, input_shape=(5, 5, 1), activation='sigmoid'))
model.add(layers.Flatten())
model.add(layers.Dense(1, activation='sigmoid'))

model.layers[0].set_weights([keras_conv_w, keras_conv_b])
model.layers[2].set_weights([keras_fc_w,keras_fc_b])

np.set_printoptions(precision=5)
sgd = optimizers.SGD(learning_rate=100) #change suggested by Prof Amir
model.compile(loss='MSE', optimizer=sgd, metrics=['accuracy'])
history = model.fit(input, output, batch_size=1, epochs=1)
print('Output from Keras model after backpropagation is', model.predict(input))

print('1st conv layer weight is ')
print(model.get_weights()[0].reshape(1, 1, 3, 3))
print('1st conv layer weight is ')
print(model.get_weights()[1])
print('FC layer weight is ')
print(model.get_weights()[2].reshape(9))
print('FC layer bias is ')
print(model.get_weights()[3])
```

```
Output before backprop: [0.63992102]
----- Keras Model after Backpropagation (Example 1)-----

2023-03-06 16:05:55.908608: I tensorflow/core/platform/cpu_feature_guard.cc:193]
This TensorFlow binary is optimized with oneAPI Deep Neural Network Library
(oneDNN) to use the following CPU instructions in performance-critical
operations:  SSE4.1 SSE4.2
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.

1/1 [==============================] - 0s 225ms/step - loss: 0.1269 - accuracy:
0.0000e+00
```

```
1/1 [==============================] - 0s 57ms/step
Output from Keras model after backpropagation is [[0.94939]]
1st conv layer weight is
[[[[ 0.10745  0.91042  0.49045]
   [ 0.38067  0.23018  0.73869]
   [ 0.42771  0.53635 -0.01125]]]]
1st conv layer weight is
[0.56307]
FC layer weight is
[ 0.34338  0.34923  0.67778  0.42426  0.08943  0.1702   0.43194 -0.20615
  0.39757]
FC layer bias is
[0.39325]
```

8.1.2 Our Model

[ ]:
```
input = our_input
label = our_label

conv_layer_param = {'numOfKernels':1, 'kernelSize':3, 'activation':'sigmoid',␣
  ↪'weights': our_conv_w, 'bias':our_conv_b}
fl_layer_param = {}
fc_layer_param = {'numOfNeurons': 1, 'activation': 1,'weights': our_fc_w}

print('----- Our Model after Backpropagation (Example 1) -----')

model = NeuralNetwork(input.shape, activation='sigmoid', loss=0, lr=100)
model.addLayer('conv', conv_layer_param)
model.addLayer('fl', fl_layer_param)
model.addLayer('fc', fc_layer_param)
yp, error = model.train(np.array([input]), np.array(label), num_epochs=1)
print(f"Output from our model after backpropagation is: {model.calculate(np.
  ↪array([input]))}")
model.print_weights()
```

```
----- Our Model after Backpropagation (Example 1) -----
Output from our model after backpropagation is: [[0.94972]]
layer 0 is a Conv layer, weights shape = (1, 1, 3, 3) and bias shape = (1,).
Weights and bias are as follows:
[[[[ 0.10745  0.91042  0.49045]
   [ 0.38067  0.23018  0.73869]
   [ 0.42771  0.53635 -0.01125]]]]
[0.56307]
layer 1 is a Flatten layer without weights
layer 2 is a FC layer, weights shape = (9,) and bias shape = (1,). Weights and
bias are as follows:
[ 0.34338  0.34923  0.67778  0.42426  0.08943  0.1702   0.43194 -0.20615
  0.39757]
```

```
[0.39325]
```

8.2 Example 2

```
[ ]: def example2():
         # set random seed
         np.random.seed(0)


         # create input and output for keras model
         keras_input = np.random.rand(1, 7, 7, 1)
         keras_output = np.random.rand(1)

         our_input = keras_input.reshape(1, 7, 7)
         our_label = [keras_output]

         keras_conv1_w = np.random.rand(3, 3, 1, 2)
         keras_conv1_b = np.random.rand(2)
         keras_conv2_w = np.random.rand(3, 3, 2, 1)
         keras_conv2_b = np.random.rand(1)
         keras_fc_w = np.random.rand(9, 1)
         keras_fc_b = np.random.rand(1)

         # create input and output for our model
         our_conv1_w = keras_conv1_w.reshape(2, 1, 3, 3)
         our_conv1_b = keras_conv1_b.reshape(2)
         our_conv2_w = keras_conv2_w.reshape(1, 2, 3, 3)
         our_conv2_b = keras_conv2_b.reshape(1)
         our_fc_w = keras_fc_w.reshape(9)
         our_fc_b = keras_fc_b.reshape(1)

         our_fc_w = np.hstack((our_fc_w, our_fc_b))

         return keras_input,keras_output,keras_conv1_w, keras_conv1_b,␣
      ↪keras_conv2_w, keras_conv2_b, keras_fc_w, keras_fc_b, \
                 our_input,our_label,our_conv1_w, our_conv1_b,our_conv2_w,␣
      ↪our_conv2_b,our_fc_w
```

```
[ ]: keras_input, keras_output, keras_conv1_w, keras_conv1_b, keras_conv2_w,␣
      ↪keras_conv2_b, keras_fc_w, keras_fc_b, \
     our_input, our_label, our_conv1_w, our_conv1_b, our_conv2_w, our_conv2_b,␣
      ↪our_fc_w = example2()
```

8.2.1 Keras Model

[ ]:
```
input = keras_input
output = keras_output
print('label:', output)

print('-----Keras Model after Backpropagation (Example 2)-----')
model = Sequential()

model.add(layers.Conv2D(2, 3, input_shape=(7, 7, 1), activation='sigmoid'))
model.add(layers.Conv2D(1, 3, activation='sigmoid'))
model.add(layers.Flatten())
model.add(layers.Dense(1, activation='sigmoid'))

model.layers[0].set_weights([keras_conv1_w, keras_conv1_b])
model.layers[1].set_weights([keras_conv2_w,keras_conv2_b])
model.layers[3].set_weights([keras_fc_w, keras_fc_b])


np.set_printoptions(precision=5)
sgd = optimizers.SGD(learning_rate=100)
model.compile(loss='MSE', optimizer=sgd, metrics=['accuracy'])
history = model.fit(input, output, batch_size=1, epochs=1)
print('keras model output after backprop is', model.predict(input))

print('1st conv layer weight is ')
print(model.get_weights()[0].reshape(2, 1, 3, 3))
print('1st conv layer bias is ')
print(model.get_weights()[1])
print('2nd conv layer weight is ')
print(model.get_weights()[2].reshape(1, 2, 3, 3))
print('2nd conv layer bias is ')
print(model.get_weights()[3])
print('FC layer weight is ')
print(model.get_weights()[4].reshape(9))
print('FC layer bias is ')
print(model.get_weights()[5])
```

```
label: [0.36371]
-----Keras Model after Backpropagation (Example 2)-----
1/1 [==============================] - 0s 173ms/step - loss: 0.3931 - accuracy:
0.0000e+00
1/1 [==============================] - 0s 42ms/step
keras model output after backprop is [[0.00104]]
1st conv layer weight is
[[[[0.56994 0.43787 0.98811]
   [0.10116 0.20855 0.16035]
   [0.65284 0.25239 0.46602]]]
```

```
 [[[0.24351 0.15868 0.10935]
   [0.65611 0.1374  0.19631]
   [0.36792 0.82073 0.09616]]]]
1st conv layer bias is
[0.83742 0.09441]
2nd conv layer weight is
[[[[0.97344 0.46631 0.9737 ]
   [0.60247 0.73619 0.03673]
   [0.27976 0.11789 0.29311]]

  [[0.11632 0.3149  0.41187]
   [0.06113 0.69016 0.56356]
   [0.26302 0.52019 0.09156]]]]
2nd conv layer bias is
[0.57278]
FC layer weight is
[-0.2247  -0.83551 -0.48674 -1.02226 -0.43772 -0.86473 -0.97078 -0.56753
 -1.13398]
FC layer bias is
[-0.32594]
```

8.2.2 Our Model

```
[ ]: print('----- Our Model after Backpropagation (Example 2) -----')
input = our_input
label = our_label

conv1_layer_param = {'numOfKernels': 2, 'kernelSize': 3, 'activation':␣
 ↪'sigmoid', 'weights': our_conv1_w, 'bias': our_conv1_b}
conv2_layer_param = {'numOfKernels': 1, 'kernelSize': 3, 'activation':␣
 ↪'sigmoid', 'weights': our_conv2_w, 'bias': our_conv2_b}
fl_layer_param = {}
fc_layer_param = {'numOfNeurons': 1, 'activation': 1, 'weights': our_fc_w}

model = NeuralNetwork(input.shape, activation='sigmoid', loss=0, lr=100)
model.addLayer('conv', conv1_layer_param)
model.addLayer('conv', conv2_layer_param)
model.addLayer('fl', fl_layer_param)
model.addLayer('fc', fc_layer_param)
yp, error = model.train(np.array([input]), np.array(label), num_epochs=1)
print(f"Output from our model after backpropagation is: {model.calculate(np.
 ↪array([input]))}")
model.print_weights()
```

```
----- Our Model after Backpropagation (Example 2) -----
Output from our model after backpropagation is: [[0.00104]]
```

```
layer 0 is a Conv layer, weights shape = (2, 1, 3, 3) and bias shape = (2,).
Weights and bias are as follows:
[[[[0.56992 0.43826 0.98801]
   [0.1017  0.20854 0.16094]
   [0.65281 0.25293 0.46601]]]


 [[[0.24378 0.15826 0.10963]
   [0.65578 0.13753 0.19584]
   [0.3681  0.82037 0.09638]]]]
[0.83607 0.09422]
layer 1 is a Conv layer, weights shape = (1, 2, 3, 3) and bias shape = (1,).
Weights and bias are as follows:
[[[[0.97157 0.46366 0.97167]
   [0.59993 0.73423 0.03417]
   [0.27788 0.11521 0.29107]]

  [[0.11384 0.31299 0.40918]
   [0.05923 0.68744 0.56159]
   [0.26046 0.51826 0.08887]]]]
[0.57312]
layer 2 is a Flatten layer without weights
layer 3 is a FC layer, weights shape = (9,) and bias shape = (1,). Weights and
bias are as follows:
[-0.22442 -0.83527 -0.48649 -1.02196 -0.43751 -0.86449 -0.97057 -0.56737
 -1.13374]
[-0.32562]
```

8.3 Example 3

[ ]:
```python
def example3():
    # set random seed
    np.random.seed(0)

    # create input and output for keras model
    keras_input = np.random.rand(1, 8, 8, 1)
    keras_output = np.random.rand(1)

    our_input = keras_input.reshape(1, 8, 8)
    our_label = [keras_output]

    keras_conv_w = np.random.rand(3, 3, 1, 2)
    keras_conv_b = np.random.rand(2)
    keras_fc_w = np.random.rand(18, 1)
    keras_fc_b = np.random.rand(1)

    # create input and output for our model
```

```
    our_conv_w = keras_conv_w.reshape(2, 1, 3, 3)
    our_conv_b = keras_conv_b.reshape(2)
    our_fc_w = keras_fc_w.reshape(18)
    our_fc_b = keras_fc_b.reshape(1)

    our_fc_w = np.hstack((our_fc_w, our_fc_b))
    return keras_input,keras_output,keras_conv_w, keras_conv_b, keras_fc_w,␣
↪keras_fc_b, \
           our_input,our_label,our_conv_w, our_conv_b,our_fc_w
```

[ ]:
```
keras_input, keras_output, keras_conv_w, keras_conv_b, keras_fc_w, keras_fc_b, \
        our_input, our_label, our_conv_w, our_conv_b, our_fc_w=example3()
```

8.3.1 Keras Model

[ ]:
```
input = keras_input
output = keras_output
print('label:', output)

print('-----Keras Model after Backpropagation (Example 3)-----')
model = Sequential()

model.add(layers.Conv2D(2, 3, input_shape=(8, 8, 1), activation='sigmoid'))
model.add(layers.MaxPooling2D(2))
model.add(layers.Flatten())
model.add(layers.Dense(1, activation='sigmoid'))

model.layers[0].set_weights([keras_conv_w, keras_conv_b])
model.layers[3].set_weights([keras_fc_w,keras_fc_b])

np.set_printoptions(precision=5)


sgd = optimizers.SGD(learning_rate=100)
model.compile(loss='MSE', optimizer=sgd, metrics=['accuracy'])
history = model.fit(input, output, batch_size=1, epochs=1)
print('Keras model output after backprop is',model.predict(input))

print('1st conv layer weight is ')
print(model.get_weights()[0].reshape(2, 1, 3, 3))
print('1st conv layer bias is ')
print(model.get_weights()[1])
print('FC layer weight is ')
print(model.get_weights()[2].reshape(18))
print('FC layer bias is ')
print(model.get_weights()[3])
```

```
label: [0.19658]
```

```
-----Keras Model after Backpropagation (Example 3)-----
1/1 [==============================] - 0s 154ms/step - loss: 0.6451 - accuracy:
0.0000e+00
1/1 [==============================] - 0s 38ms/step
Keras model output after backprop is [[0.99954]]
1st conv layer weight is
[[[[0.3613  0.81864 0.08867]
   [0.83532 0.08839 0.97368]
   [0.46092 0.9748  0.59834]]]


 [[[0.73691 0.03401 0.28045]
   [0.11179 0.29337 0.11395]
   [0.31612 0.40792 0.06221]]]]
1st conv layer bias is
[0.68129 0.56232]
FC layer weight is
[ 0.23038  0.48557  0.05873  0.53793  0.8948   0.28099  0.63399  0.09414
  0.68126  0.25137  0.14858  0.54896 -0.01321  0.79247 -0.02798  0.6407
  0.23616  0.69747]
FC layer bias is
[0.92381]
```

8.3.2 Our Model

[ ]:
```
print('----- Our Model after Backpropagation (Example 3) -----')
input = our_input
label = our_label

conv_layer_param = {'numOfKernels': 2, 'kernelSize': 3, 'activation':␣
 ↪'sigmoid', 'weights': our_conv_w, 'bias': our_conv_b}
max_layer_param = {'poolingSize': 2}
fl_layer_param = {}
fc_layer_param = {'numOfNeurons': 1, 'activation': 1, 'weights': our_fc_w}

model = NeuralNetwork(input.shape, activation='sigmoid', loss=0, lr=100)
model.addLayer('conv', conv_layer_param)
model.addLayer('max', max_layer_param)
model.addLayer('fl', fl_layer_param)
model.addLayer('fc', fc_layer_param)
yp, error = model.train(np.array([input]), np.array(label), num_epochs=1)
print(f"Output from our model after backpropagation is: {model.calculate(np.
 ↪array([input]))}")
model.print_weights()
```

```
----- Our Model after Backpropagation (Example 3) -----
Output from our model after backpropagation is: [[0.99949]]
layer 0 is a Conv layer, weights shape = (2, 1, 3, 3) and bias shape = (2,).
```

```
Weights and bias are as follows:
[[[[0.36844 0.81934 0.09561]
   [0.83743 0.09477 0.9759 ]
   [0.46767 0.97559 0.60441]]]


 [[[0.73612 0.03587 0.27964]
   [0.11721 0.2927  0.11492]
   [0.31595 0.41138 0.06064]]]]
[0.68531 0.55944]
layer 1 is a Max Pooling layer without weights
layer 2 is a Flatten layer without weights
layer 3 is a FC layer, weights shape = (18,) and bias shape = (1,). Weights and
bias are as follows:
[ 0.22502  0.48283  0.05397  0.53595  0.88895  0.27833  0.62816  0.09185
  0.67719  0.25242  0.14616  0.55032 -0.01551  0.79188 -0.03224  0.64272
  0.23377  0.69929]
[0.92134]
```

[ ]: