Homework Assignment 3

By Marc Mailloux

Imports:

```
from PIL import Image
import numpy as np
np.random.seed(333)
```

1st Class Convolution Layer:

```
 class Convolution_layer:


    def __init__(self):
        #calling the data from the source
        self.data = "ass3images/image_0729.jpg"
        #opening the data
        self.transfer = Image.open(self.data)
        #creating an array of the given image
        self.image = np.array(self.transfer)
        #setting the filter sizes for the weights
        self.layer_size = 3
        self.filter_size = 3
        #creating the weights using the he intialization
        self.w1 = np.random.rand(self.layer_size, self.filter_size) * np.sqrt(
2 / self.filter_size)
        self.w2 = np.random.rand(self.layer_size, self.filter_size) * np.sqrt(
2 / self.filter_size)
        self.w3 = np.random.rand(self.layer_size, self.filter_size) * np.sqrt(
2 / self.filter_size)
        # print ( w1.shape)
```

```
        #combining the weights to one 3D tensor
        self.weights = np.concatenate((self.w1, self.w2, self.w3))

        self.weights = np.reshape(self.weights, (3, 3, 3))
       #creating the bias
        self.b = np.random.randn(1, 1, 3)

    #creating a function that will pad out image as needed
    def zero_padding(self,X, pad):
        self.padding = np.pad(X, ((pad, pad), (pad, pad), (0, 0)), 'constant',
constant_values=0)

        return self.padding
    #creating a function that will take one step in a convolution
    def single_step_conv(self,prev_slice, W, b):
        # convolutions
        self.had = (prev_slice * W) + b
        self.z = np.sum(self.had)

        return self.z
    #uses the single step conv to iterate over a 3D tensor
    def conv_forward(prev, W, b, hyperParams):
        # print(prev.shape)
        (n_H_prev, n_W_prev, n_C_prev) = prev.shape
        num_models = 1
        (f, f, n_C) = W.shape

        stride = hyperParams['stride']
        pad = hyperParams['pad']
        #formula for convolution/correlation
        n_H = int((n_H_prev - f + 2 + pad) / stride) + 1
        n_W = int((n_H_prev - f + 2 + pad) / stride) + 1

        z = np.zeros((n_H, n_W, n_C))

        prev_pad = cnn.zero_padding(prev, pad)
```

```
        #itarting through the 3d tensor
        for h in range(n_H):

            for w in range(n_W):

                for c in range(n_C):
                    creating the bounding box that will be convolving
                    vert_start = h * stride
                    vert_end = vert_start + f
                    horiz_start = w * stride
                    horiz_end = horiz_start + f
                    # print('b4')

                    # print(prev_pad.shape)
                    using the window values I just created to slice our image
and out put to z
                    slice_prev = prev_pad[vert_start:vert_end, horiz_start:hor
iz_end, :]
```

For some reason my convolving made my shape increase by size by 1. so it would go from 500,500,3 to 501,501,3 to 502,502,3 and so on. I had to implement this because when the shape was not the same the multiplication of course could not take place here. So this solved it for now and I figured would be okay since it would be a very low amount of samples that would be discarded.

```
                    if slice_prev.shape == (3, 2, 3) or slice_prev.shape == (2
, 3, 3) or slice_prev.shape == (
                    2, 2, 3) or slice_prev.shape == (2, 1, 3) or slice_prev.sh
ape == (1, 2, 3) or slice_prev.shape == (
                    1, 1, 3):
                        break
```

```
                    z[h, w, c] = cnn.single_step_conv(slice_prev, W[..., c], b
[..., c])

        assert (z.shape == (n_H, n_W, n_C))

        cache = (prev, W, b, hyperParams)
        #storing variables as single output
        return z, cache
```

Creating the Backwards Propagtion function

```
    def conv_backwards(a):

        gradient, cache = a
        (prev, W, b, hyperparams) = cache

        (n_H_prev, n_W_prev, n_C_prev) = prev.shape

        (f, n_C_prev, n_C) = W.shape

        stride = hyperparams['stride']
        pad = hyperparams['pad']

        (n_H, n_W, n_C) = gradient.shape

        dprev = np.zeros((n_H_prev, n_W_prev, n_C_prev))
        dW = np.zeros((f, f, n_C_prev, n_C))
        db = np.zeros((1, 1, 1, n_C))

        prev_pad = zero_padding(prev, pad)
        dprev_pad = zero_padding(dprev, pad)

        for h in range(n_H):

            for w in range(n_W):
```

```
                for c in range(n_C):

                    vert_start = h * stride
                    vert_end = vert_start + f
                    horiz_start = w * stride

                    horiz_end = horiz_start + f

                    slice = prev_pad[vert_start:vert_end, horiz_start:horiz_en
d, :]

                    dprev_pad[vert_start:vert_end, horiz_start:horiz_end, :] +
= W[..., c] * gradient[h, w, c]
                    if dprev_pad.shape == (3, 2, 3) or dprev_pad.shape == (2,
3, 3) or dprev_pad.shape == (
                    2, 2, 3) or dprev_pad.shape == (2, 1, 3) or dprev_pad.shap
e == (1, 2, 3) or dprev_pad.shape == (
                    1, 1, 3):
                        break
                    dW[..., c] += slice * gradient[h, w, c]
                    db[..., c] += gradient[h, w, c]

                    # z[h, w, c] = single_step_conv(slice, W[..., c], b[..., c
])

        dprev[:, :, :] = dprev_pad[pad:-pad, pad:-pad, :]
        assert (dprev.shape == (n_H_prev, n_W_prev, n_C_prev))

        return dprev, dW, db
```

2nd Class: Relu Class

```
class Relu(object):
```

```
    def activation(incoming):
        x , y = incoming
        activate= (np.maximum((x), 0), y)
        return activate
```

Creating the first convolution Layer function and also creating a function that can convolute n times as long as it the input has been convoluted before.

```
def conv_layer1(image):
    #creating the dictionary for our stride and padding parameters
    hyp = {'stride':1 , 'pad':1}
    #calling the weights from the cnn class
    weights = cnn.weights
    b = cnn.b
    #convoluting using the forward convolution function in the convolution lay
er class
    conv1 = Ccnn.conv_forward(image ,weights , b , hyp )
    newim ,cache  = conv1
    #outputting the image and cache of variables
    return newim ,cache

def conv_layer_n(a):

    x , y = a

    (prev, W, b, hyperParams) = y

    conv= Convolution_layer.conv_forward(x ,W , b , hyperParams )
    newim , cache = conv
    return newim, cache
```

Creating the hidden Layer of 8 Covolutions follwed by a Relu

```
# calls all previous functions
def hidden_layer(image):
    print('starting')
    conv1 = conv_layer1(image)
    print('passed through conv1')
    conv1_re = Relu.activation(conv1)
    print('passed through relu')
    conv2 = conv_layer_n(conv1_re)
    print('passed through conv2 and relu')
    conv3 = (conv_layer_n(Relu.activation(conv2)))
    print('layer 4')
    conv4 = (conv_layer_n(Relu.activation(conv3)))
    print('layer 5')
    conv5 = (conv_layer_n(Relu.activation(conv4)))
    print('layer 6')
    conv6 = (conv_layer_n(Relu.activation(conv5)))
    print('layer 7')
    conv7 = (conv_layer_n(Relu.activation(conv6)))
    print('layer 8')
    conv8 = (conv_layer_n(Relu.activation(conv7)))
    print('final Layer')
    final = conv_layer_n(conv8)
    #this is for outputing the image to test to for it grabbing features.
    x, b = final

    r = x[:, :, 0]
    g = x[:, :, 1]
    b = x[:, :, 2]

    from PIL import Image

    rgb = np.stack((r, g, b))
    img = Image.fromarray(x, 'RGB')
```

```
    # img.save('my.png')
    img.show()

    print('workeddd')

    return final


cnn = Convolution_layer()
print(cnn.image.shape)
#splitting data
hl = hidden_layer(cnn.image)
x , y = hl

print('Starting Backprob')
b = cnn.conv_backwards(x)
#cant get it to get past the backprop :(
```