بانام خالق نکته

عنوان:

گزارش امتحان پایان ترم درس پیاده سازی بهینه شبکه های عصبی عمیق

دانشجو:

هدیه مفتخری رستم خانی

اساتید راهنما:

استاد حمید سربازی آزاد

استاد هاجر فلاحتی

بهمن ۱۴۰۱

بخش ۱ : پیاده سازی تکنیک differential

توجه : فایل عکس icon.png در فولدر ارسالی قرار گرفته ، لطفا قبل از اجرای کد در colab عکس مذکور را در مسیر sample_data/ در colab بارگزاری کنید.

در این بخش به پیاده سازی تکنیک input differential روی یک تابع convolution می پردازیم . (کد های این بخش در فایل final_exam در بخشی تحت عنوان Input differential in convolution قرار گرفته)

در ابتدا مسیر برنامه را به فولدر Sample_data/ تغییر می دهیم تا فایل های تصویر را از این پوشه بتوانیم بخوانیم :

```
%cd sample_data/
```

سپس در تابع main پارامتر هایی که قرار است در عمل convolution از آن ها استفاده کنیم را مقدار دهی می کنیم ، متغیر num_filter_channels در اینجا تعداد چنل های فیلتر و عکس و متغیر num_filters تعداد فیلتر هایی است که روی یک عکس قرار است اعمال بشود .

```
#identification
padding = 0
strides = 1
num_filters = 1
kernel_size = 3
num_filter_channels = 3
```

سپس ابعاد عکس ورودی را تعیین می کنیم :

```
#image(s) dimention
width = 300
height = 360
```

حال با استفاده از کد زیر و کتابخانه cv2 تصویری که در مسیر Sample_data/ بارگزاری کرده بودیم را خوانده و نمایش می دهیم :

```
#reading image(s)
img = cv2.imread('icon.png')
img = cv2.resize(img,(width,height))
cv2_imshow(img)
```

با توجه به اینکه تصویری که توسط تابع imread خوانده می شود به صورت (b,g,r) است ، ما آن را با استفاده از کد زیر به (r,g,b) تبدیل می کنیم :

```
pixels = []
for i in range (height):
  for j in range (width):
    pixels.append(img[i,j])

#RGB streams are stored in different array to facilitate the process of co
nvlution
b = []
g = []
r = []

for i in pixels:
  b.append(i[0])
  g.append(i[1])
  r.append(i[2])
```

سپس برای اینکه برنامه حالت داینامیک داشته باشد یعنی برای ورودی با تعداد چنل های متفاوت کار کند ، سه ماتریس r ، g و b را به ترتیب به عنوان چنل های ۰ ، ۱ و ۲ تعریف می کنیم :

```
img = np.zeros((num_filter_channels,height,width))

img[0,:,:] = np.array(r).reshape(height,width)
img[1,:,:] = np.array(g).reshape(height,width)
img[2,:,:] = np.array(b).reshape(height,width)
```

با استفاده از کد زیر اگر padding تعریف شده توسط برنامه نویس مخالف صفر باشد در هر دو جهت افقی و عمودی به اندازه دو برابر padding (چون هم به راست اضافه می شود هم چپ و هم به بالای عکس اضافه می شود و هم پایین آن) به ابعاد عکس اضافه می شود :

```
# Apply Equal Padding to All Sides
if padding != 0:
  img_padded = np.zeros((num_filter_channels,height + padding*2, width + p
adding*2))
  for i in range(num_filter_channels):
    img_padded[i,int(padding):int(-1 * padding), int(padding):int(-
1 * padding)] = img[i,:,:]
```

```
    print(img_padded[i].shape)
else:
  img_padded = np.zeros((num_filter_channels,height,width))
  for i in range(num_filter_channels):
    img_padded[i,:,:] = img[i,:,:]
```

حال کانال های مختلف فیلتر دلخواه را با مقادیر دلخواه با کد زیر مقدار دهی می کنیم :

```
b_kernel = np.array([0.11 ,0.11 ,0.11 ,0.11 ,0.11 ,0.11 ,0.11 ,0.11 ,0.11
]).reshape(kernel_size,kernel_size)
g_kernel = np.array([0.11 ,0.11 ,0.11 ,0.11 ,0.11 ,0.11 ,0.11 ,0.11 ,0.11
]).reshape(kernel_size,kernel_size)
r_kernel = np.array([0.11 ,0.11 ,0.11 ,0.11 ,0.11 ,0.11 ,0.11 ,0.11 ,0.11
]).reshape(kernel_size,kernel_size)
```

حال آرایه ای به نام array_filters تعریف می کنیم ، که در آن فیلتر هایی که قرار است روی عکس ورودی اعمال شود را می ریزیم :

```
array_filters = np.zeros((num_filters,num_filter_channels, kernel_size, ke
rnel_size))
```

در این مرحله چون کانال های فیلتر برای ما از قبل مشخص است ، کانال های r وg وb را به ترتیب در خانه ۰ و ۱و۲ درآرایه c که آرایه چنل های یک فیلتر است می ریزیم و سپس فیلتری که ساختیم را در خانه اول آرایه نگه دارنده فیلتر ها می ریزیم :

```
#filter channels
c = np.zeros((num_filter_channels, kernel_size, kernel_size))
c[0,:] = r_kernel
c[1,:] = g_kernel
c[2,:] = b_kernel

array_filters[0,:,:,:] = c
```

حال ابعاد عکس خروجی را حساب می کنیم :

```
#calculate 2Dconvolution
out_height = int(((height - kernel_size + 2 * padding) / strides) + 1)
out_width = int (((width - kernel_size + 2 * padding) / strides) + 1)
```

در مرحله بعد تابع conv_2d را تعریف می کنیم ، این تابع در واقع عکس ورودی ، ابعاد عکس (ارتفاع و عرض) آن و یک کانال از یک فیلتر (kernel) را می گیرد و فیلتر را روی عکس اعمال می کند تنها تفاوت عملکرد این تابع با تابع covolution عادی این است که در اینجا kernel که روی عکس حرکت می کند در هر مرحله در

یک window ضرب می شود که مقدار ماتریس window اگر اولین window باشد (window نماینده) همان مقدار اولیه آن است و اگر window های بعدی باشد برابر با اختلاف window کنونی با window اولیه (نماینده) است. در نهایت ضرب window در kernel درون متغیر prod ریخته ی شود و حاصل جمع خانه های prod یک درایه از ماتریس خروجی(out_matrix) را تولید می کند در نهایت شکل out_matrix که یک آرایه خطی هست را به شکل یک ماتریس با ابعاد out_height وout_width که قبل تر حساب کردیم در می آوریم ، کد این بخش به صورت زیر است :

```
def conv_2d (input_img , height , width , kernel):
  output_matrix = []
  for i in range (0,(height - kernel_size  + 2 * padding + 1),strides):
    for j in range (0,(width - kernel_size  + 2 * padding + 1),strides):
      if(i == j == 0):
        window = input_img[i:i+kernel_size , j:j+kernel_size]
        temp = window
      else :
        temp = np.subtract(window, input_img[i:i+kernel_size , j:j+kernel_
size])

      prod = np.multiply(temp , kernel)
      output_matrix.append(np.sum(prod))
  output_matrix = (np.array(output_matrix).reshape(out_height,out_width))
  return output_matrix
```

در این مرحله نوبت به آن رسیده که برای عکس ورودی به تعداد فیلتر ها ، و به ازای هر فیلتر به تعداد کانال های عکس تابع conv_2d را صدا بزنیم و در نهایت کانال های متناظر برای فیلتر های مختلف در عکس خروجی را جمع کنیم و در آرایه feature_map ، بریزیم ، که این کار را با تابع convolution انجام می دهیم :

```
def convolution(input_map, kernal, front_delta=None, deriv=False):
  C, W, H = img_padded.shape#(batch,channel,w,h)
  K_NUM, K_C, K_W, K_H = kernal.shape
  print('kernal.shape',kernal.shape)
  print('img_padded.shape',img_padded.shape)
  if deriv == False:
    feature_map = np.zeros((K_NUM, W-K_W+1, H-K_H+1))
    for kId in range(K_NUM):
        for cId in range(C):
            feature_map[kId] += \
              conv_2d(input_map[cId],W,H, kernal[kId,cId,:,:])
    return feature_map
```

در مرحله آخر خروجی تابع convolution را که در main آن را صدا زده بودیم با دستورات زیر چاپ می کنیم:

```
diff_conv_img = convolution(img_padded,array_filters)
# Plotting output images of convolution layer to compare with faulty out
put
plt.figure(figsize=(30,36))
plt.subplot(321)
plt.imshow(diff_conv_img.squeeze())
plt.axis('off')
plt.show()
```

بخش ۲ : پیاده سازی تکنیک LSH

توجه : فایل عکس icon.png در فولدر ارسالی قرار گرفته ، لطفا قبل از اجرای کد در colab عکس مذکور را در مسیر /sample_data در colab بارگزاری کنید.

در این بخش به پیاده سازی تکنیک input Clustering روی یک تابع convolution می پردازیم .

(کد های این بخش در فایل final_exam در بخشی تحت عنوان Input clustering using LSH قرار گرفته)

در پیاده سازی تکنیک LSH کد های بخش های ابتدایی برنامه که مربوط به ذخیره عکس و اضافه کردن padding به آن است مشابه قسمت قبل است ، قسمت هایی که تفاوت کرده :

۱) اندازه kernel به ۷*۷ تغییر یافته
۲) ابعاد عکس به ۳۸*۳۸ تغییر یافته

بخش جدید پیاده سازی تکنیک LSH است به این منظور ابتدا دو ماتریس (v1,v2) با اندازه برابر اندازه kernel (در اینجا ۷*۷) که اعداد داخل آن با مقادیر رندوم بین -۱ و ۱ پر شده با استفاده از کد زیر می سازیم :

```
#defining 2 random vectors(array ellements ۱- or 1)
v1 = np.random.uniform(low=-1, high=1, size=(kernel_size,kernel_size))
v2 = np.random.uniform(low=-1, high=1, size=(kernel_size,kernel_size))
```

حال برای پیاده سازی تکنیک LSHنیاز به یافتن نماینده در هر گروه داریم بنابراین ابتدا رو داده(های) ورودی که برای training هستند (در اینجا ما یک عکس را در نظر گرفتیم) با استفاده از ۳ حلقه for تو در تو که حلقه اول شمارش را روی تعداد کانال انجام می دهد و دو حلقه بعدی روی سطر و ستون عکس جلو می روند ، به ازای هر کانال در عکس ، ابتدا پنجره مورد نظر را در متغیر temp می ریزیم و سپس temp را در v1 می ریزیم و حاصل جمع خانه های ماتریس حاصل را در sum1 می ریزیم ، همین کار را برای v2 انجام می دهیم و حاصل جمع را در sum2 می ریزیم ، سپس sum1 وsum2 را به تابع sign_to_cluster که عملکرد آن را جلوتر توضیح می دهیم پاس می هیم تا شماره گروه winow را برگرداند (00,01,10یا ۱۱) ، شماره گروه window را درون متغیر cluster_num می ریزیم سپس window مورد نظر را با ماتریس گروه متناظر آن در آرایه clusters_matrix که به ازای هر کانال مجموع ماتریس های گروه های مختلف را نگه می دارد ، جمع می کنیم و در نهایت در آرایه cntr_cluster_members که به ازای هر کانال ، تعداد اعضای هر گروه را نگه می دارد ، شمارنده گروهی که window به آن تعلق دارد را یک شماره افزایش می دهیم :

```
cntr_cluster_members = np.zeros((num_filter_channels,4))
clusters_matix = np.zeros((num_filter_channels,4,kernel_size,kernel_size))

#computing clusters
for h in range(len(c)):
  for i in range (0,(height - kernel_size  + 2 * padding + 1),strides):
    for j in range (0,(width - kernel_size  + 2 * padding + 1),strides):
      temp = img_padded[h,i:i+kernel_size , j:j+kernel_size]
      prod1 = np.multiply(temp , v1)
      sum1 = np.sum(prod1)
      prod2 = np.multiply(temp , v2)
      sum2 = np.sum(prod2)
      cluster_number = sign_to_cluster(sum1,sum2) #define cluster
      clusters_matix[h,cluster_number,:,:] += np.add(temp,clusters_matix[h
,cluster_number,:,:]) #add member of each class to the others
      cntr_cluster_members[h,cluster_number] +=1 #count member of each clu
ster
```

پس از محاسبه مجموع ماتریس های گروه های مختلف به ازای هر کانال می توانیم با تابع زیر در هر گروه میانگین هر درایه را در ماتریس با توجه به تعدادی که در ماتریس cntr_cluster_member ذخیره کردیم ، محاسبه کنیم :

```
#calculate mean in each cluster
for h in range(len(c)):
  for i in range(4):
    print("clusters_matix : ",i,clusters_matix[h,i,:])
```

```
    print("cntr_cluster_members : ",cntr_cluster_members[h,i])
    clusters_matix[h,i,:] = clusters_matix[h,i,:] / cntr_cluster_members[h
,i]
```

معرفی تابع sign_to_cluster : این تابع دو عدد را می گیرد و تعیین می کند و اگر بزرگتر از ۰ بودند عدد ۰ و اگر کوچکتر یا مساوی بودند عدد ۱ را به آن ها نسبت می دهد در نهایت دو عدد دودویی داریم که عدد دسیمال متناظر آن را این تابع بر می گرداند ، به طور مثال sum1= -123و sum2= ۱2را می گیرد و عدد 1 را بر می گرداند ، از این تابع در گروه بندی window ها استفاده شده :

```
def sign_to_cluster(sum1,sum2):
    if (sum1 <= 0 and sum2 <= 0) :
      return 0
    elif  (sum1 <= 0 and sum2 > 0) :
      return 1
    elif  (sum1 > 0 and sum2 <= 0) :
      return 2
    else :
      return 3
```

حال به تعریف تابع conv_2d می پردازیم . ، این تابع در واقع عکس ورودی ، ابعاد عکس (ارتفاع و عرض) آن ، یک کانال از یک فیلتر (kernel) و شماره کانالی که قرار است فیلتر را روی آن اعمال کند می گیرد .

تنتفاوت عملکرد این تابع با تابع covolution عادی این است که در اینجا kernel که روی عکس حرکت می کند در هر مرحله در یک window ضرب می شود که ماتریس window ابتدا در v1 وv2 ضرب می شود تا با استفاده از تابع sign_to_cluster گروه بندی آن مشخص شود ، سپس kernel به جای window اصلی در ماتریس نماینده گروهی که window به آن تعلق دارد ضرب می شود ، در نهایت حاصل جمع درایه های ماتریس حاصل ضرب یک درایه از ماتریس خروجی (output_matrix) را تشکیل می دهد ، در آخر هم ماتریس خروجی بعد از reshap شدن برگردانده می شود:

```
def conv_2d (input_img , height , width , kernel,channel_num):
  output_matrix = []
  for i in range (0,(height - kernel_size  + 2 * padding + 1),strides):
    for j in range (0,(width - kernel_size  + 2 * padding + 1),strides):
      window = input_img[i:i+kernel_size , j:j+kernel_size]
      prod1 = np.multiply(window , v1)
      sum1 = np.sum(prod1)
      prod2 = np.multiply(window , v2)
```

```
        sum2 = np.sum(prod2)
        cluster_number = sign_to_cluster(sum1,sum2)
        prod = np.multiply(clusters_matix[channel_num,cluster_number,:,:] ,
kernel)
        output_matrix.append(np.sum(prod))
    output_matrix = (np.array(output_matrix).reshape(out_height,out_width))
    return output_matrix
```

در این مرحله نوبت به آن رسیده که برای عکس ورودی به تعداد فیلتر ها ، و به ازای هر فیلتر به تعداد کانال های عکس تابع conv_2d را صدا بزنیم و در نهایت کانال های متناظر برای فیلتر های مختلف در عکس خروجی را جمع کنیم و در آرایه feature_map ، بریزیم ، که این کار را با تابع convolution انجام می دهیم :

```
def convolution(input_map, kernal, front_delta=None, deriv=False):
    C, W, H = img_padded.shape#(batch,channel,w,h)
    K_NUM, K_C, K_W, K_H = kernal.shape
    print('kernal.shape',kernal.shape)
    print('img_padded.shape',img_padded.shape)
    if deriv == False:
      feature_map = np.zeros((K_NUM, W-K_W+1, H-K_H+1))
      for kId in range(K_NUM):
          for cId in range(C):
              feature_map[kId] += \
                conv_2d(input_map[cId],W,H, kernal[kId,cId,:,:])
      return feature_map
```

در مرحله آخر خروجی تابع convolution را که در main آن را صدا زده بودیم با دستورات زیر چاپ می کنیم:

```
    LSH_conv_img = convolution(img_padded,array_filters)
    # Plotting output images of convolution layer to compare with faulty out
    put
    plt.figure(figsize=(30,36))
    plt.subplot(321)
    plt.imshow(LSH_conv_img.squeeze())
    plt.axis('off')
    plt.show()
```

محاسبه دقت روش differential :

برای مقایسه دقت روش differential با حالتی که عملیات convolution به صورت عادی اجرا می شود ، (کد پیاده سازی convolution عادی در فایل final_exam تحت عنوان normal convolution implementation قرار گرفته).

ابتدا تابع زیر را برای پیاده سازی convolution عادی در نظر می گیریم ، تمامی کد های پیاده سازی convolution عادی با روش differential یکسان است فقط تابع زیر تفاوت دارد که به جای محاسبه differential خود window را در نظر می گیرد :

```
def conv_2d (input_img , height , width , kernel):
  output_matrix = []
  for i in range (0,(height - kernel_size  + 2 * padding + 1),strides):
    for j in range (0,(width - kernel_size  + 2 * padding + 1),strides):
      temp = input_img[i:i+kernel_size , j:j+kernel_size]
      prod = np.multiply(temp , kernel)
      output_matrix.append(np.sum(prod))
  output_matrix = (np.array(output_matrix).reshape(out_height,out_width))
  return output_matrix
```

حال با استفاده از روش MSE و با استفاده از کد زیر عکس های خروجی دو روش عادی و differential را با هم مقایسه می کنیم و می بینیم که دقت روش differential ٪۱۰۰ است و خروجی ها کاملا با هم یکسان هستند :

```
#compare two images calculate accuracy
print('diff_conv_img.shape',diff_conv_img.shape)
print('simple_conv_img.shape',simple_conv_img.shape)

err = np.sum(pow(abs((diff_conv_img.astype("float") - simple_conv_img.asty
pe("float"))),2))
err /= float(diff_conv_img.shape[1] * diff_conv_img.shape[2])
print('accuracy',(1-err)*100,'%')
```

محاسبه دقت روش LSH :

برای مقایسه دقت روش LSH با حالتی که عملیات convolution به صورت عادی اجرا می شود ، (کد پیاده سازی convolution عادی برای این بخش در فایل final_exam تحت عنوان simple little convolution قرار گرفته).

ابتدا تابع زیر را برای پیاده سازی convolution عادی در نظر می گیریم ، تمامی کد های پیاده سازی convolution عادی با روش LSH یکسان است فقط دیگر نیاز به clustering نداریم و همچنین تابع زیر تفاوت دارد که به جای محاسبه گروه و انتخاب نماینده برای هر window که در kernel ضرب شود از خود window اصلی استفاده می کنیم :

```
def conv_2d (input_img , height , width , kernel):
  output_matrix = []
  for i in range (0,(height - kernel_size  + 2 * padding + 1),strides):
    for j in range (0,(width - kernel_size  + 2 * padding + 1),strides):
      temp = input_img[i:i+kernel_size , j:j+kernel_size]
      prod = np.multiply(temp , kernel)
      output_matrix.append(np.sum(prod))
  output_matrix = (np.array(output_matrix).reshape(out_height,out_width))
  return output_matrix
```

حال با استفاده از روش MSE و با استفاده از کد زیر عکس های خروجی دو روش عادی و LSH را با هم مقایسه می کنیم و می بینیم که دقت روش LSH ۰٪ است و خروجی ها اصلا با هم یکسان نیستند :

```
#compare two images calculate accuracy
print('LSH_conv_img.shape',LSH_conv_img.shape)
print('simple_little_conv_img.shape',simple_little_conv_img.shape)
#MSE
err = np.sum(pow(abs((LSH_conv_img.astype("float") - simple_little_conv_im
g.astype("float"))),2))
err /= float(simple_conv_img.shape[1] * simple_conv_img.shape[2])
if(err == float('inf')):
  print('accuracy',0,'%')
else:
   print('accuracy',(1-err)*100,'%')
```

نتیجه ای که از این بخش می توان گرفت این است که تعداد گروه ها در خوشه بندی روی دقت تاثیر مسقیم دارد به این معنا که هر چه تعداد گروه ها بیشتر باشد دقت بالاتر می رود ، به طور مثال برای حالتی که به تعداد هر window یک گروه داشته باشیم الگوریتم عملا با پیاده سازی عادی تفاوتی نمی کند و نماینده هر گروه خود window انتخابیست ولی در یک مثال دیگر اگر کلا دو گروه داشته باشیم دقت به نسبت پیاده سازی ما که با ۴ گروه است بسیار کمتر می شود.

محاسبه تعداد مقادیر ذخیره شده روش differential :

در فاز اولیه برای پیاده سازی تکنیک differential فقط به اندازه تعداد خانه های یک window نیاز به فضای ذخیره سازی داریم چون فقط لازم است window اصلی را نگه داریم و بقیه window ها که خوانده می شوند اختلافشان با window اصلی محاسبه شود ، اما اگر در یک فاز دیگر بخواهیم به تعداد عکس ها در هر batch نگاه کنیم و تعداد فیلتر هایی که نگه می داریم ، به تعداد عکس ها * ابعاد عکس هایی که می خواهیم عملیات convolution را روی آن ها پیاده سازی کنیم و همچنین به اندازه ذخیره کردن فیلتر ها فضا نیاز داریم.

محاسبه تعداد مقادیر ذخیره شده روش LSH :

در فاز اولیه برای پیاده سازی تکنیک LSH چون به تعداد کانال ها و به ازای هر کانال چند گروه(در اینجا ۴ گروه) داریم که هر گروه فقط یک window نماینده نگه می دارد ، تعداد مقادری ذخیره شده :

تعداد کانال * تعداد دسته ها * اندازه window

ولی در فاز دیگر اگر بخواهیم batch های عکس های ورودی و فیلتر ها را هم در نظر بگیریم ، به تعداد عکس ها * ابعادشان بعلاوه تعداد فیلتر ها * ابعادشان فضا نیاز داریم.

تعداد ضرب در روش differential:

تعداد ضرب های این روش با تعداد ضرب های روش convolution عادی برابر است و برابر با فرمول زیر است:

ابعاد ماتریس خروجی (W_out * H_out) * تعداد کانال ها (c) * ابعاد kernel (L*S)

تعداد ضرب در روش LSH:

A = محاسبه گروه ها در فاز اولیه = ابعاد ماتریس خروجی (W_out * H_out) * تعداد وکتور ها (۲) * سایز window(L*S)

B = محاسبه گروه برای window ها در هر عکس ورودی = ابعاد ماتریس خروجی (W_out * H_out) * تعداد وکتور ها (۲) * سایز window(L*S)

C = عملیات convolution روی عکس = ابعاد ماتریس خروجی (W_out * H_out) * تعداد کانال ها (c) * ابعاد kernel (L*S)

تعداد ضرب ها = A+B+C

تعداد جمع ها در روش differential:

A = عملیات convolution روی عکس = ابعاد ماتریس خروجی (W_out * H_out) * تعداد کانال ها (c) * ابعاد kernel (L*S)

B = به ازای هر window تفریق روی هر خانه از window داریم = ابعاد ماتریس خروجی (W_out * H_out) * ابعاد kernel (L*S) منهای(-) ابعاد window (window اصلی تفریق ندارد)

تعداد جمع ها = A+B

تعداد جمع ها در روش LSH:

A = شمارش window ها برای میانگین گیری = ابعاد ماتریس خروجی (W_out * H_out)

B = جمع Window ها با اعضای گروه خودشان = ابعاد ماتریس خروجی (W_out * H_out) * ابعاد kernel (L*S)

C = محاسبه گروه برای window ها در هر عکس ورودی = ابعاد ماتریس خروجی (W_out * H_out) * تعداد وکتور ها (۲) * سایز window(L*S)

D = عملیات convolution روی عکس = ابعاد ماتریس خروجی (W_out * H_out) * تعداد کانال ها (c) * ابعاد kernel (L*S)

تعداد جمع ها = A+B+C+D

پایان