

روش بیان شده در مقاله سه مرحله اصلی دارد:

- ۱- مرتب سازی مجدد المان های دیتاست
- ۲- کد گذاری پیش گویانه
- ۳- کد گذاری بی نظمی ها

در این رویکرد یک مدل پیشگویی روی کل دیتاست آموزش داده میشود،

مرتب سازی مجدد المان های دیتاست

برای افزایش دقت پیشگویی ابتدا المان های دیتاست را مجددا مرتب میکنیم، روش کدگذاری پیش گویانه هنگامی بیشترین کارایی را دارد که خطای پیش بینی در نزدیکی صفر تمرکز داشته باشد. تجربه نشان داده که مرتب سازی داده ها براساس کم کردن فاصله اقلیدسی بین داده های متوالی، باعث فشردگی بهتری در مقایسه با هنگامی که داده ها به طور تصادفی مرتب شده اند میشود.

برای رسیدن به این هدف نویسندها مقاله، روشی را به نام (kNN-MST (k nearest neighbor min spanning tree) توسعه داده اند.

ابتدا شایان توجه است که مرتب سازی داده ها به منظور بیشینه کردن شباهت بین داده های متوالی، معادل پیدا کردن یک مسیر همیلتونی کمینه در گراف کامل نزدیک ترین همسایه ها است.

محاسبه یک پیمایش بهینه سراسری روی یک گراف همسایگی یک مسئله MetricTSP (Metric Traveling Saleman Problem) است؛ اگرچه این مسئله یک مسئله NP-hard است ولی تقریب های خوبی نیز وجود دارند. پیمایش یک درخت پوشای کمینه (MST) گراف کامل همسایگی، راه حلی است که از نظر هزینه دو برابر به صرفه تر است. البته محاسبه MST روی گراف همسایگی کامل یک دیتاست بزرگ ممکن است از نظر محاسباتی غیر ممکن باشد؛ در نتیجه از رویکرد زیر استفاده میکنیم:

Lemma 1. Let $G = (V, E)$. For all $1 \leq k \leq |V|$, if $k\text{NN}(G)$ is connected, then a MST of $k\text{NN}(G)$ is a MST of G .

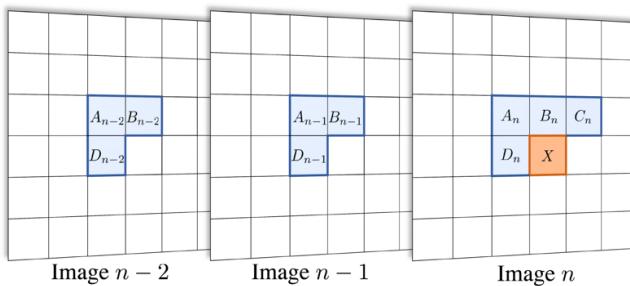
این لم بیان میکند که در گراف G با V گره و E ضلع، اگر k را در محدوده فوق الذکر انتخاب کنیم، در صورتی که گراف k نزدیک ترین همسایه G متصل باشد، برای پیدا کردن درخت پوشای کمینه G کافی است درخت پوشای کمینه $k\text{NN}(G)$ را محاسبه کنیم.

نویسندها مقاله به روشی اشاره میکنند که توانسته $k\text{NN}(G)$ را با مرتبه $O(\log n)$ محاسبه کند.

کد گذاری پیش گویانه

ایده اصلی کد گذاری پیش گویانه استفاده از ویژگی های نمونه های همسایه و همچنین ویژگی های نمونه معرفی شده فعلی براساس ترتیب الگوریتم KNN-MST است. برای آموزش مدل پیشگو، یک رشته زمینه (context string) از هر یک از پیکسل های تصاویر دیتا استخراج میکنیم، این رشته یک آرایه از ویژگی های مربوط به پیش بینی است.

برای نمونه یکی از استراتژی های مورد استفاده DAB/DABC است که از پیکسل های تصویر جاری و دو تصویر قبلی طبق الگوی بیان شده استفاده میکند.



در رابطه با تصاویر RGB استفاده از دو رویکرد ممکن است، حالت تکی و حالت سه گانه، در حالت سه گانه یک مدل پیشگویی برای هر کدام از کanal ها آموزش داده می شود اما در حالت تکی برای هر سه کanal یک مدل آموزش داده میشود. به دلیل رابطه های متقابلی که بین سه کanal قرار دارد مدل تکی بهتر عمل میکند.

برای پیش گویی از یک رویکرد دو مرحله ای استفاده می شود، ابتدا برای هر پیکسل در دیتاست یک context string استخراج میشود، سپس از این داده ها برای آموزش مدل روی همه داده ها استفاده میشود.

آموزش را آنقدر ادامه میدهیم تا overfit شود زیرا نیز به تعمیم روی داده های جدید نداریم.

بعد از اینکه آموزش تمام شد، مدل را روی تک تک پیکسل ها اعمال میکنیم و خط را ذخیره میکنیم، برای کد کردن خط از الگوریتم هافمن کمک میگیریم تا حجم اطلاعات بهینه شود.

Algorithm 1 Predictive Dataset Compression

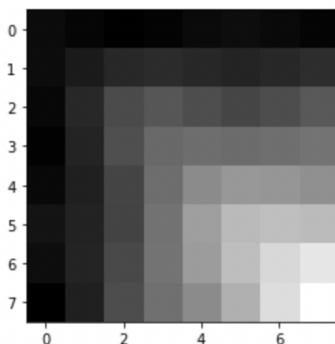
```

Input: a dataset  $\mathcal{D} = \{x_i\}_{i=1}^n$ 
 $(x_{i,j})_{j=1}^n = \text{KNN-MST-REORDER}(x^n)$ 
 $\{\{\text{context}_a, a\}_{a \in x_t}\}_{t=1}^n = \text{EXTRACTCONTEXT}(\mathcal{D})$ 
 $h = \text{TRAINPREDICTIVEMODEL}(\{\{\text{context}_a, a\}_{a \in x_t}\}_{t=1}^n)$ 
error_string = [ ]
for  $x_t$  in  $\mathcal{D}$  do
    for  $a$  in  $x_t$  do
        error_string +=  $h(\text{context}_a) - a$ 
    end for
end for
HUFFMANENCODE(metadata, f.params, error_string)

```

```
▶ #### in this cell, show us the image in grayscale ####
----start
plt.imshow(image, cmap='gray')
----end
```

☞ <matplotlib.image.AxesImage at 0x7f5a760d0690>



تصویر اصلی را نشان میدهیم.

```
#### Define JPEG compression Function #####
def jpeg_compression(image,z):
    ----start
    subtracted_image = image - (2***(image.shape[0]-1))
    dct_transformed = scipy.fftpack.dct(scipy.fftpack.dct(subtracted_image,norm='ortho',axis=0),norm='ortho',axis=1)
    divide_with_z = dct_transformed/z
    rounded = (np.rint(divide_with_z)).astype(int)
    ----end
    return rounded
```

ابتدا مقدار 2^{k-1} را از آرایه های ماتریس تصویر کم میکنیم، این کار به این جهت انجام می شود که مقدار اولین المان در تبدیل DCT که بیانگر تقریب کلی تصویر است کاهش یابد و برای کد کردن آن به تعداد بیت کمتری نیاز باشد.

سپس تبدیل DCT دو بعدی میگیریم، برای این کار ابتدا در یک راستا و سپس در راستای دیگر (عمود بر راستای قبلی) تبدیل DCT میگیریم.

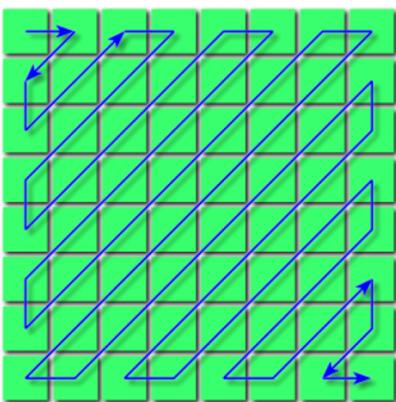
ماتریس حاصل از تبدیل را بر ماتریس z تقسیم می کنیم، مقادیر ماتریس z براساس میزان اهمیت هر کدام از آرایه های ماتریس حوزه فرکانس تصویر انتخاب شده است، به این صورت که به مقادیر کم اهمیت تر مقدار بیشتری اختصاص داده تا پس از تقسیم و گرد کردن با احتمال بیشتری صفر شوند.

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

در آخر المان های ماتریس حاصله را گرد میکنیم.

در این مرحله میخواهیم ماتریس حاصل از مرحله قبل را به صورت آرایه ذخیره کنیم، ترتیب ذخیره سازی المان ها مهم است و تاثیر بسزایی در مقدار فشرده سازی دارد زیرا میتوان مقادیر صفر پایانی آرایه را ذخیره نکرد و از این طریق طول آرایه را کاهش داد.

بهینه ترین ترتیب ذخیره سازی، ترتیب زیکزکی است، در این روش اولویت ذخیره سازی را به فرکانس های پایین تر میدهیم زیرا در اکثر تصاویر این فرکانس ها، فرکانس های غالب هستند و احتمال صفر بودن آن ها کم است، از آن طرف فرکانس های بالاتر که با احتمال بیشتری صفر هستند در آخر ماتریس قرار میگیرند و ما به خواسته خود که قرار گیری المان های صفر در آخر ماتریس و عدم ذخیره آن هاست دست میباییم.



در بدترین حالت طول آرایه ما به تعداد المان های ماتریس است، پس یک حلقه for با همین تعداد تکرار (تعداد المان های ماتریس) تعریف میکنیم.

در هر مرحله باید مختصات آن اولویت را در ماتریس زیگزاک پیداکنیم و المان متناظر با آن مختصات را از ماتریس حاصل از مرحله قبلی برداریم و به آرایه اضافه کنیم.

برای پیدا کردن مختصات متناظر با هر اولویت از روش زیر استفاده میکنیم:

در هر مرحله از را به عنوان اندیس برای درج در آرایه استفاده میکنیم پس مقدار α همان اولویت درج است. ترم $i == zigzag_a$ ماتریس به ما میدهد که در مختصاتی که شرط ارضاء میشود $true$ و در سایر جایگاه ها $false$ است، با ضرب کردن ۱ در این ماتریس، مقدار $true$ به ۱ و مقادیر $false$ به صفر تبدیل میشوند، حال کافی است اندیس مقدار ماکزیمم در آن ماتریس که همان یک است را بیابیم.

پس از کامل شدن آرایه، از آخر آرایه رو به عقب حرکت میکنیم و به ازای هر المانی که صفر بود یک واحد از طول آرایه میکاهیم تا زمانی که به یک المان غیر صفر برسیم. در صورتی که به المان غیر صفر رسیدیم حلقه را ترک میکنیم و عبارت EOB (End Of Block) را به نشانه پایان آرایه درج میکنیم؛ در هنگام بازیابی هنگامی که به این عبارت رسیدیم به تعداد پیکسل های باقی مانده، مقدار صفر درج میکنیم.

در این مثال ماتریس و آرایه حاصل از آن به شرح زیر است:

```
[[-27 -17 -4 -1 0 0 0 0]
 [-15 11 0 0 0 0 0 0]
 [-3 0 2 0 0 0 0 0]
 [-1 0 0 1 0 0 0 0]
 [-1 0 0 0 0 0 0 0]
 [ 0 0 0 0 0 0 0 0]
 [ 0 0 0 0 0 0 0 0]
 [ 0 0 0 0 0 0 0 0]]
```

این وکتور تبدیل به کد ۰ و ۱ میشود و روی حافظه ذخیره میشود.

بازیابی تصویر

```
####define a vec_2_mat function#####
def vector_to_matrix(vector1):
    #----start
    mat = np.zeros_like(image)
    zigzag_a = np.array(zigzag)
    for i in range(len(vector1)-1):
        index = np.unravel_index(np.argmax(1*(zigzag_a==i),axis=None),zigzag_a.shape)
        mat[index] = vector1[i]
    #----end
    return mat
```

برای بازیابی تصویر ابتدا باید وکتور آن را به ماتریس تبدیل کرد، برای این منظور ابتدا ماتریسی با درایه های صفر و ابعاد در نظر گرفته شده برای بلوک های فشرده سازی، میسازیم.

با توجه به اینکه اندیس‌های وکتور، متناظر با اولویت‌های بیان شده در ماتریس زیکز-اک است، حلقه‌ای تعریف می‌کنیم و در هر مرحله اجرای آن، مختصات متناظر با اندیس خانه جاری وکتور را به کمک ماتریس زیکز-اک و روش بیان شده در مرحله قفل مباییم و مقدار آن خانه را در ماتریس اولیه ای که ساختیم به روزرسانی می‌کنیم.

پس از رسیدن به انتکاهی و کتور، مقدار خانه های باقیمانده ماتریس اولیه صفر باقی میماند که مطلوب ما نیز همین است.

```
#### Define JPEG Decompression Function #####
def jpeg_decompress(T_hat, Z):
    #----start
    # Please set norm='ortho' in Discrete Cosine Transform Function in fftpack
    # you can use scipy.fftpack for dct and idct
    T_dot = T_hat * Z
    idct_transformed = scipy.fftpack.idct(scipy.fftpack.idct(T_dot,norm='ortho',axis=0),norm='ortho',axis=1)
    T = idct_transformed + (2***(image.shape[0]-1))
    #----end

    return T
```

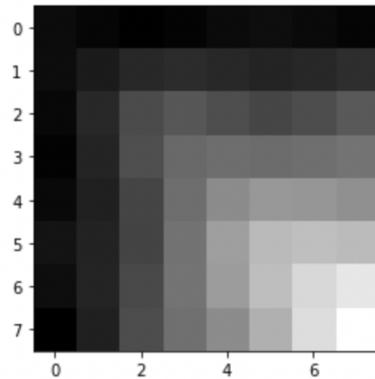
حال ماتریس خروجی از مرحله قبل را در ماتریس Z ضرب میکنیم، و از ماتریس حاصل تبدیل DCT معکوس میگیریم.

در نهایت مقدار 2^{k-1} را که در مراحل اولیه از ماتریس تصویر کم کرده بودیم، به آن اضافه میکنیم.

میتوانیم تصویر اولیه و ماتریس آن را با نتیجه بازیابی پس از فشرده سازی آن مقایسه کنیم:

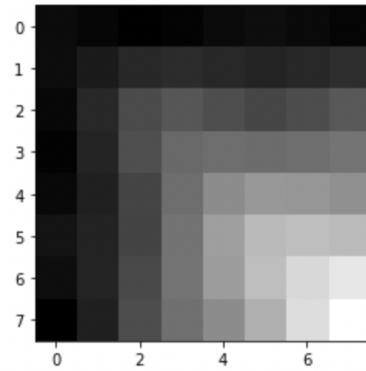
تصویر اولیه:

```
# here we got an image with size of 8*8 #
image = np.array([[ 33.90241005, 30.54775832, 27.97959216, 29.41620856,
   33.24129806, 35.02424581, 32.94586615, 29.93754577],
[ 34.69138758, 42.05585559, 50.00845073, 52.39975944,
 49.87770263, 47.92905119, 49.74401583, 52.76852369],
[ 31.77462629, 49.6598556 , 69.28626278, 75.85118649,
 70.55222279, 66.14976936, 70.26596303, 77.1838152 ],
[ 28.6068495 , 47.44677111, 71.24679329, 85.75392986,
 88.52192546, 87.37539338, 89.03778505, 92.22403802],
[ 32.34196256, 44.83473787, 65.85785317, 88.43194903,
105.17498572, 112.09541293, 110.98089772, 107.82000439],
[ 38.13783369, 46.89900981, 65.4683231 , 91.29834031,
116.1083363 , 130.97097637, 134.03007616, 131.82387514],
[ 35.16966279, 47.82418321, 68.69472555, 92.07094076,
114.10402987, 133.14028493, 147.87319677, 156.18775704],
[ 26.92228824, 45.6096369 , 70.37352354, 90.01813956,
105.70408236, 125.69502973, 150.94870287, 169.98238288]])
```



تصویر بازیابی شده پس از فشرده سازی:

```
[[ 33.90241005 30.54775832 27.97959216 29.41620856 33.24129806
 35.02424581 32.94586615 29.93754577]
[ 34.69138758 42.05585559 50.00845073 52.39975944 49.87770263
 47.92905119 49.74401583 52.76852369]
[ 31.77462629 49.6598556 , 69.28626278 75.85118649 70.55222279
 66.14976936 70.26596303 77.1838152 ]
[ 28.6068495 , 47.44677111, 71.24679329, 85.75392986 88.52192546
 87.37539338 89.03778505 92.22403802]
[ 32.34196256, 44.83473787, 65.85785317, 88.43194903 105.17498572
 112.09541293 110.98089772 107.82000439]
[ 38.13783369, 46.89900981, 65.4683231 , 91.29834031 116.1083363
 130.97097637 134.03007616 131.82387514]
[ 35.16966279, 47.82418321, 68.69472555, 92.07094076 114.10402987
 133.14028493 147.87319677 156.18775704]
[ 26.92228824, 45.6096369 , 70.37352354, 90.01813956 105.70408236
 125.69502973 150.94870287 169.98238288]]
```



مشاهده میشود که تصویر بازیابی شده بسیار به تصویر اولیه شباهت دارد.

همچنین میتوانیم از معیار های MSE, PSNR, SSIM نیز برای مقایسه استفاده کنیم:

```
''' in this cell,caculate 3 following metrics using our original image(first cell of part A) and
our final image in part B(last cell of part B)''
#----start
mse = skimage.metrics.mean_squared_error(img,image)
psnr = skimage.metrics.peak_signal_noise_ratio(image,img,data_range=(np.max(image)-np.min(image)))
ssim = skimage.metrics.structural_similarity(img,image)

print("mse={}".format(mse))
print("psnr={}".format(psnr))
print("ssim={}".format(ssim))

####print them all####

#----end

mse=9.054369180945465e-18
psnr=213.54178817671354
ssim=1.0000000000000007
```

که نتایج خوبی را نشان میدهد.



Z*1

C=10



Z*8

C=50



Z*32

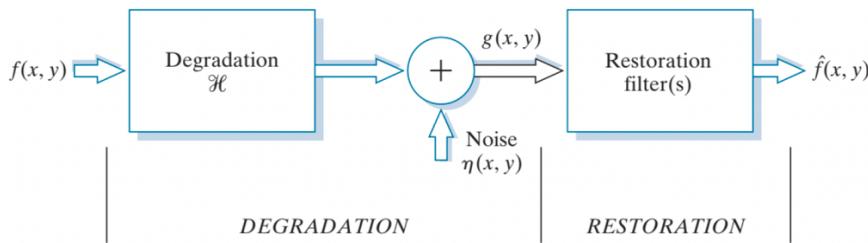
C=150

هرچه مقادیر موجود در ماتریس Z را بزرگ تر کنیم، در هر بلوک پس از تقسیم مقدار پیکسل ها به Z و گرد کردن نتیجه، تعداد صفر بیشتری خواهیم داشت. زیرا در تقسیم مخرج کسر بزرگتر و نتیجه تقسیم به صفر نزدیک تر میشود.

در نتیجه میتوانیم به فشرده سازی بیشتری دست یابیم اما در مقابل کیفیت کمتر میشود.

مقدار RMSE از چپ به راست افزایش میباید زیرا اختلاف تصاویر نسبت به تصویر اصلی بیشتر میشود.

فرض می کنیم مدل تخریب شامل یک تابع خطی مستقل از مکان (LSI) و نویز جمع شونده است.



$$g(x, y) = (h * f)(x, y) + \eta(x, y)$$

برای رسیدن به تصویر اولیه پیش از تخریب، کافی است پس از حذف نویز، کانولوشن معکوس بگیریم اما دو مشکل اساسی داریم:

- 1 - تابع نویز را نداریم
- 2 - محاسبه کانولوشن معکوس کار راحتی نیست

با تحلیل ناحیه هایی از تصویر که رنگ ثابت دارند می توانیم مدل نویز را تخمین بزنیم و از این طریق تا حدودی از مشکل بکاهیم.

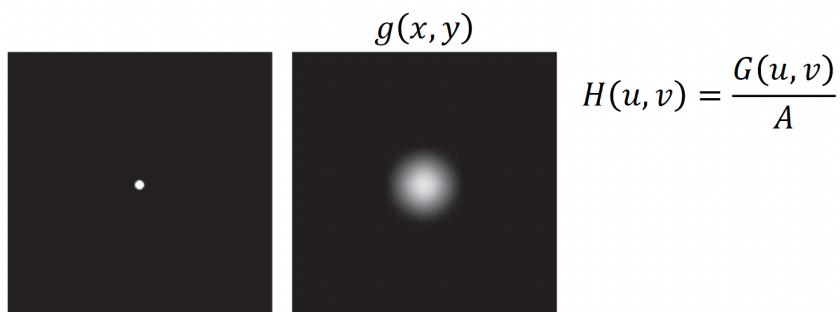
اگر معادله بالا را به فضای فرکانس ببریم، کانولوشن تبدیل به ضرب میشود و برای معکوس گرفتن کافی است تقسیم کنیم.

$$G(u, v) = H(u, v)F(u, v) + N(u, v)$$

برای تخمین تابع تبدیل میتوانیم از پاسخ ضربه استفاده کنیم، به این صورت که از نقطه ای روشن در زمینه تاریک تصویر برداری کنیم و نتیجه را با تصویر اصلی تخریب نشده (که میدانیم چیست) مقایسه کنیم.

در این صورت پاسخ ضربه همان H (تابع تخریب) خواهد بود، زیرا هر تابعی که با ضربه کانولو شود، نتیجه خودش میشود:

$$h * \delta = h$$



البته شدت نور و پارامتر های آن نقطه در پاسخ تاثیر دارند به همین دلیل، تابع تخریب را ضریبی از پاسخ ضربه در نظر میگیریم.

حال که تابع تخریب را داریم، میتوانیم با تقسیم معادله به آن (در حوزه فرکانس) اثر تخریب را از بین ببریم.

$$\hat{F}(u, v) = \frac{G(u, v)}{H(u, v)}$$

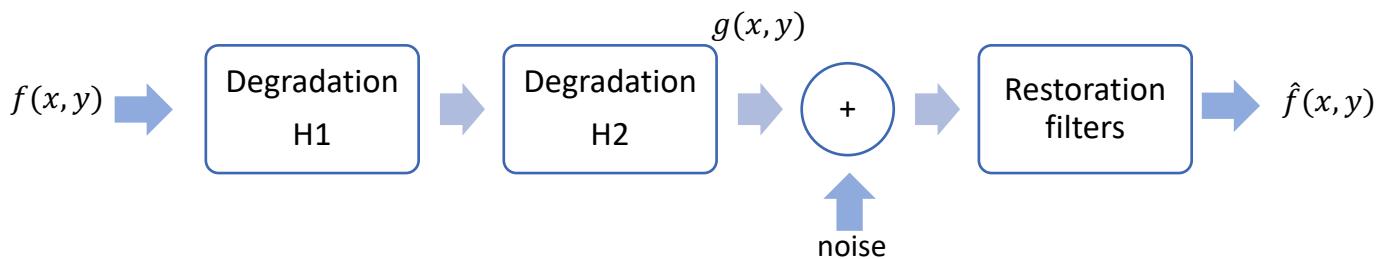
$$\hat{F}(u, v) = F(u, v) + \frac{N(u, v)}{H(u, v)}$$

با اینکه اثر تخریب از بین میرود ولی معمولاً نویز تصویر تقویت میشود به قدری که تنها چیزی که میبینیم نویز است؛ زیرا اگر $H(u, v)$ در فرکانس های بالا مقدار کمی داشته باشد، با توجه به قرار داشتن در مخرج، در آن فرکانس ها به شدت نویز را تقویت میکند.

یک راه کار برای رفع این مشکل استفاده از فیلتر پایین گذار است.

$$\hat{F}(u, v) = F(u, v)H_{BLPF}(u, v) + \frac{H_{BLPF}(u, v)}{H(u, v)} N(u, v)$$

فیلتر پایین گذار فرکانس های بالا که مؤلفه های نویز اغلب در آنجا هستند را حذف میکند و با اینکه بخشی از سیگنال اصلی را نیز تخریب میکند ولی اثر تضعیف نویز آن مطلوب است.



به این تصویر دوتابع تخریب و یک نویز اعمال شده

$$g(x, y) = (h_1 * (h_2 * f))(x, y) + \eta(x, y)$$

برای رسیدن به تصویر اولیه پیش از تخریب، کافی است پس از حذف نویز، کانولوشن معکوس بگیریم اما دو مشکل اساسی داریم:

- 1-تابع نویز را نداریم
- 2-محاسبه کانولوشن معکوس کار راحتی نیست

با تحلیل ناحیه هایی از تصویر که رنگ ثابت دارند می توانیم مدل نویز را تخمین بزنیم.

اگر معادله بالا را به فضای فرکانس ببریم، کانولوشن تبدیل به ضرب میشود و برای معکوس گرفتن کافی است تقسیم کنیم.

$$G(u, v) = H_1(u, v)H_2(u, v)F(u, v) + N(u, v)$$

$$H_1(u, v) = e^{-k(u^2+v^2)^{\frac{5}{6}}}$$

$$H_2(u, v) = \frac{T}{\pi(ua+vb)} \sin[\pi(ua+vb)] e^{-j\pi(ua+vb)}$$

اگر تابع نویز را هم داشتیم ابتدا آن را از بین میردیم و سپس با تقسیم معادله به توابع H_1 و H_2 تصویر بازیابی میشود. ولی حال که آن را نداریم از روش دیگری پیش میرویم.

معادله را بر H_1 و H_2 تقسیم میکنیم:

$$\hat{F}(u, v) = \frac{G(u, v)}{H(u, v)}$$

$$\hat{F}(u, v) = F(u, v) + \frac{N(u, v)}{H_1(u, v)H_2(u, v)}$$

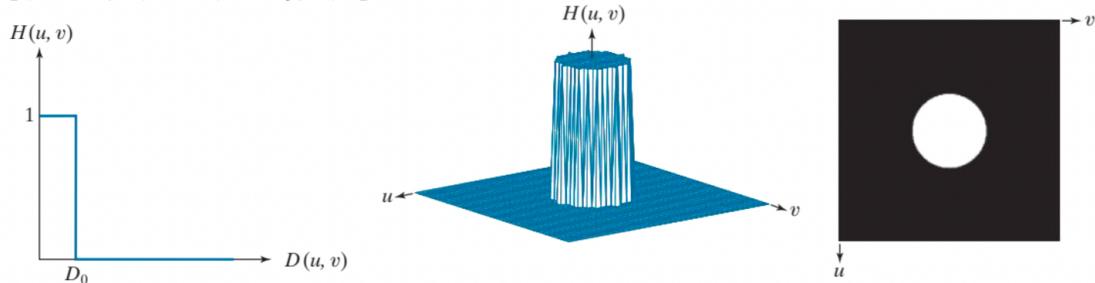
با توجه به اینکه تابع H_1 در مقادیر بالای u و v ، مقدار بسیار کمی دارد، با توجه به قرار داشتن در مخرج باعث میشود در فرکانس های بالا نویز به شدت تقویت شود. و تصویر خروجی به شدت نویزی باشد.

برای رفع این مشکل، از فیلتر پایین گذر فرکانس های بالا که مؤلفه های نویز اغلب در آنجا هستند را حذف میکند و با اینکه بخشی از سیگنال اصلی را نیز تخریب میکند ولی اثر تضعیف نویز آن مطلوب است.

معادله این فیلتر به شرح زیر است:

$$H(u, v) = \begin{cases} 1, & \text{if } D(u, v) \leq D_0 \\ 0, & \text{if } D(u, v) > D_0 \end{cases}$$

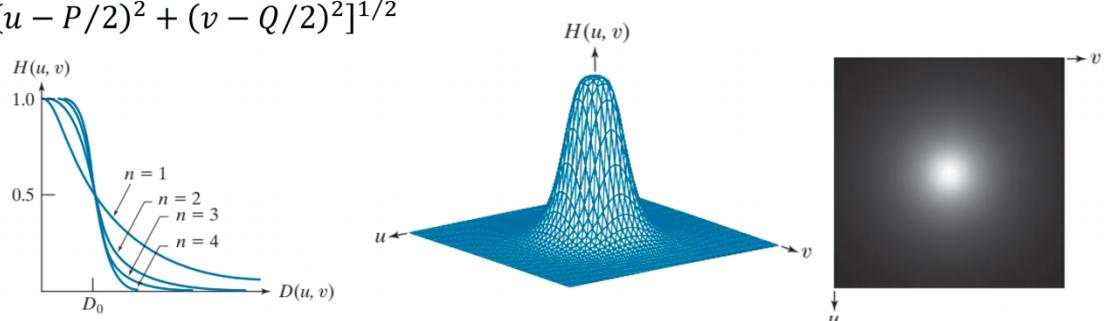
$$D(u, v) = [(u - P/2)^2 + (v - Q/2)^2]^{1/2}$$



و یا میتوانیم از فیلتر Butterworth استفاده کنیم که دارای معادله زیر است:

$$H(u, v) = \frac{1}{1 + [D(u, v)/D_0]^{2n}}$$

$$D(u, v) = [(u - P/2)^2 + (v - Q/2)^2]^{1/2}$$



پس از انجام مراحل بالا، حال میتوانیم با گرفتن تبدیل فوریه معکوس به فضای مکان بازگردیم و تصویر بازیابی شده را داشته باشیم.