import numpy as np

from itertools import product

from skimage import draw, transform

def circle(size, val=None, r\_shrink=0):

circle = np.zeros([size[0]+1, size[1]+1])

rr, cc = draw.circle\_perimeter(

size[0]//2, size[1]//2,

radius=size[0]//2 - r\_shrink,

shape=[size[0]+1, size[1]+1],

)

if val **is** None:

circle[rr, cc] = np.random.uniform(size=circle.shape)[rr, cc]

else:

circle[rr, cc] = val

circle = transform.resize(circle, size, order=0)

return circle

def show\_kernel(kernel, label=True, digits=None, text\_size=28):

*# Format kernel*

kernel = np.array(kernel)

if digits **is** **not** None:

kernel = kernel.round(digits)

*# Plot kernel*

cmap = plt.get\_cmap('Blues\_r')

plt.imshow(kernel, cmap=cmap)

rows, cols = kernel.shape

thresh = (kernel.max()+kernel.min())/2

*# Optionally, add value labels*

if label:

for i, j **in** product(range(rows), range(cols)):

val = kernel[i, j]

color = cmap(0) if val > thresh else cmap(255)

plt.text(j, i, val,

color=color, size=text\_size,

horizontalalignment='center', verticalalignment='center')

plt.xticks([])

plt.yticks([])

def show\_extraction(image,

kernel,

conv\_stride=1,

conv\_padding='valid',

activation='relu',

pool\_size=2,

pool\_stride=2,

pool\_padding='same',

figsize=(10, 10),

subplot\_shape=(2, 2),

ops=['Input', 'Filter', 'Detect', 'Condense'],

gamma=1.0):

*# Create Layers*

model = tf.keras.Sequential([

tf.keras.layers.Conv2D(

filters=1,

kernel\_size=kernel.shape,

strides=conv\_stride,

padding=conv\_padding,

use\_bias=False,

input\_shape=image.shape,

),

tf.keras.layers.Activation(activation),

tf.keras.layers.MaxPool2D(

pool\_size=pool\_size,

strides=pool\_stride,

padding=pool\_padding,

),

])

layer\_filter, layer\_detect, layer\_condense = model.layers

kernel = tf.reshape(kernel, [\*kernel.shape, 1, 1])

layer\_filter.set\_weights([kernel])

*# Format for TF*

image = tf.expand\_dims(image, axis=0)

image = tf.image.convert\_image\_dtype(image, dtype=tf.float32)

*# Extract Feature*

image\_filter = layer\_filter(image)

image\_detect = layer\_detect(image\_filter)

image\_condense = layer\_condense(image\_detect)

images = {}

if 'Input' **in** ops:

images.update({'Input': (image, 1.0)})

if 'Filter' **in** ops:

images.update({'Filter': (image\_filter, 1.0)})

if 'Detect' **in** ops:

images.update({'Detect': (image\_detect, gamma)})

if 'Condense' **in** ops:

images.update({'Condense': (image\_condense, gamma)})

*# Plot*

plt.figure(figsize=figsize)

for i, title **in** enumerate(ops):

image, gamma = images[title]

plt.subplot(\*subplot\_shape, i+1)

plt.imshow(tf.image.adjust\_gamma(tf.squeeze(image), gamma))

plt.axis('off')

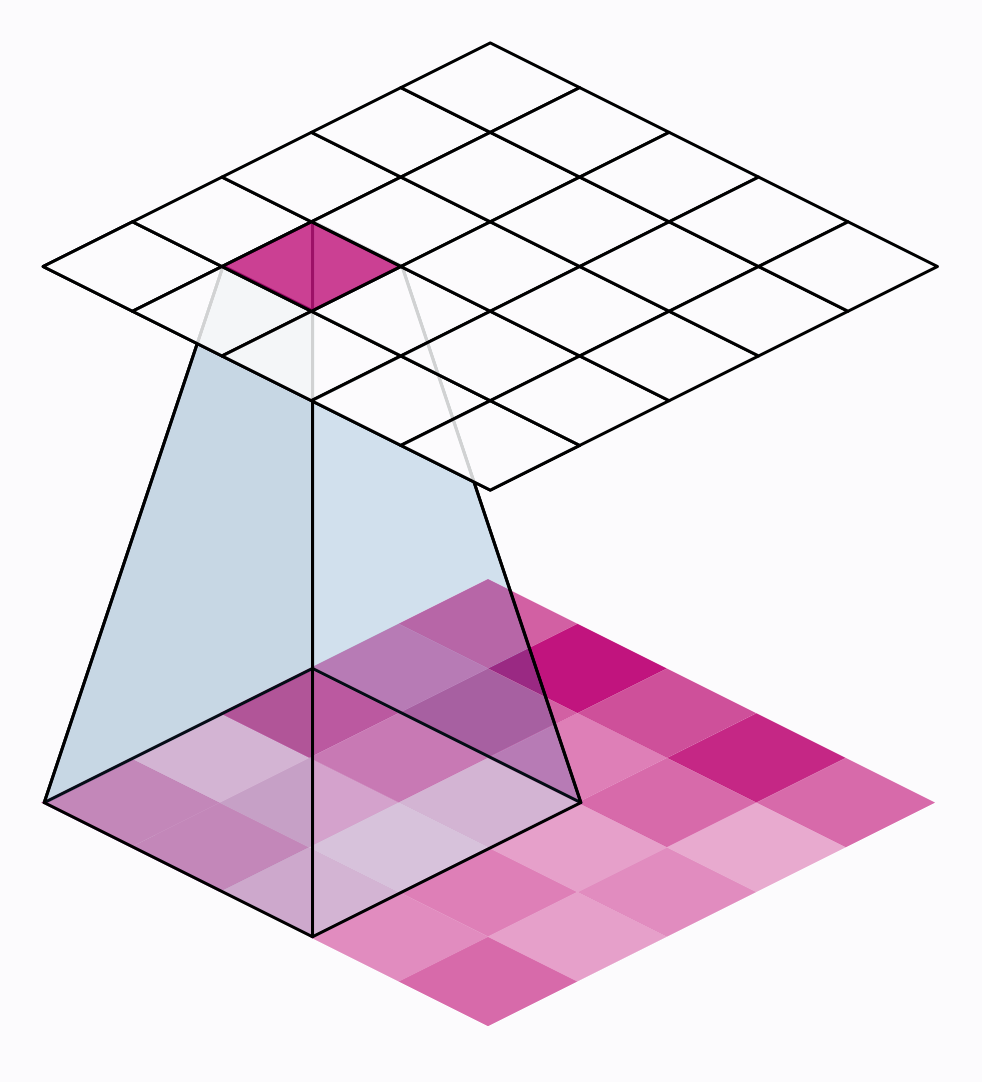
plt.title(title)

Introduction[¶](https://www.kaggle.com/code/ryanholbrook/the-sliding-window" \l "Introduction" \t "_self)

Önceki iki derste, bir görüntüden öznitelik çıkarmayı gerçekleştiren üç işlemi öğrendik:

* Bir **evrişim** katmanıyla **filtrelemek**
* **ReLU** aktivasyonuyla **tespit etmek**
* Bir **maksimum havuzlama** katmanıyla **yoğunlaştırmak**

Evrişim ve havuzlama işlemleri ortak bir özelliğe sahiptir: Her ikisi de **kayan bir pencere** üzerinden gerçekleştirilir. Evrişimde, bu "pencere" çekirdeğin boyutları, yani **kernel\_size** parametresi tarafından verilir. Havuzlamada ise bu, **pool\_size** tarafından verilen havuzlama penceresidir.



Hem evrişim hem de havuzlama katmanlarını etkileyen iki ek parametre vardır: pencerenin **adımları** (strides) ve görüntü kenarlarında **dolgu** (padding) kullanılıp kullanılmayacağı. **strides** parametresi, pencerenin her adımda ne kadar hareket etmesi gerektiğini söylerken, **padding** parametresi girdi görüntüsünün kenarlarındaki pikselleri nasıl ele alacağımızı açıklar.

Bu iki parametre ile, iki katmanı tanımlama şöyle olur:

from tensorflow import keras

from tensorflow.keras import layers

model = keras.Sequential([

layers.Conv2D(filters=64,

kernel\_size=3,

strides=1,

padding='same',

activation='relu'),

layers.MaxPool2D(pool\_size=2,

strides=1,

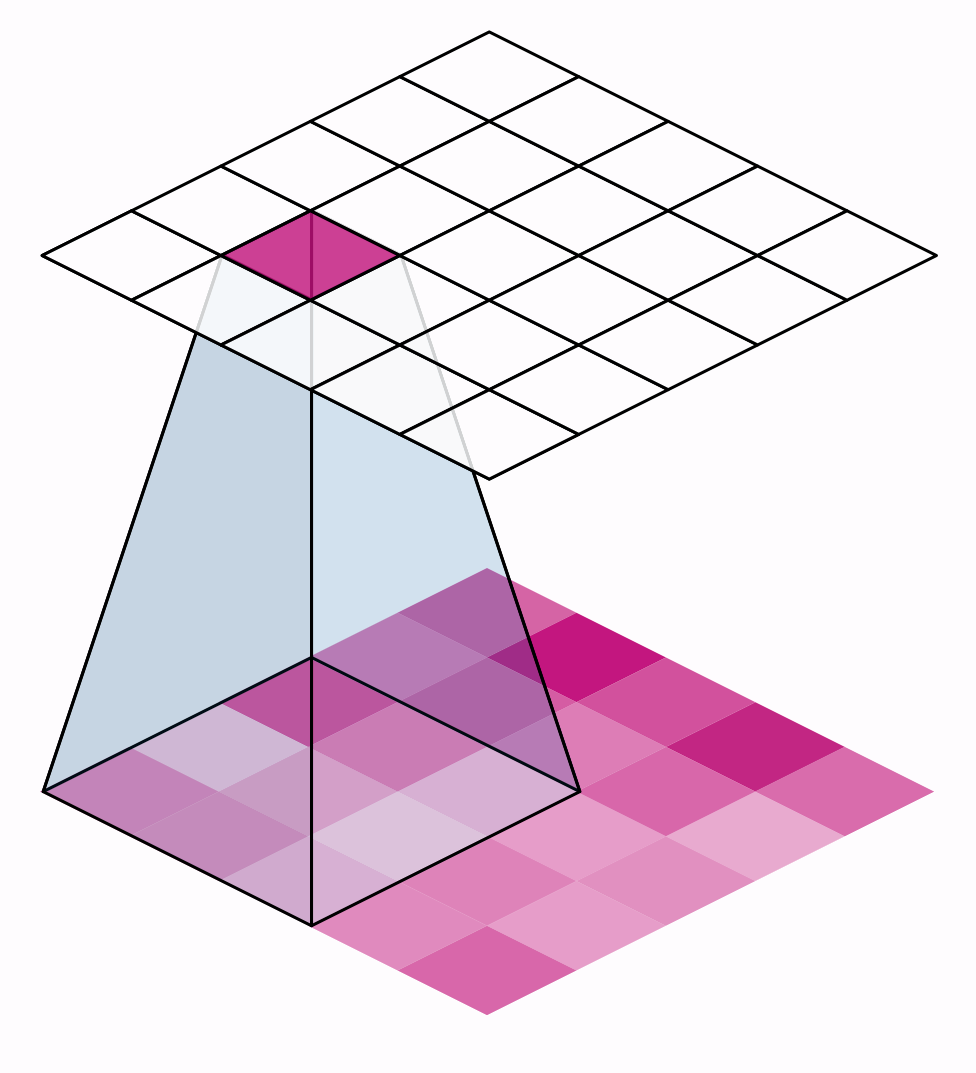
padding='same')

*# More layers follow*

])

# Stride[¶](https://www.kaggle.com/code/ryanholbrook/the-sliding-window" \l "Stride" \t "_self)

Pencerenin her adımda hareket ettiği mesafeye **adım** (stride) denir. Bu adımı görüntünün her iki boyutu için de belirtmemiz gerekir: biri sağdan sola hareket için, diğeri yukarıdan aşağıya hareket için. Bu animasyon, her adımda 2 piksel hareket anlamına gelen **strides=(2, 2)** değerini göstermektedir.



Adımın (stride) etkisi nedir? Herhangi bir yöndeki adım 1'den büyük olduğunda, kayan pencere her adımda girdi görüntüsündeki bazı pikselleri atlar.

Sınıflandırma için yüksek kaliteli öznitelikler istediğimiz için, evrişimsel katmanlar çoğunlukla **strides=(1, 1)** değerine sahip olacaktır. Adımı artırmak, özetimizde potansiyel olarak değerli bilgileri kaçırmamız anlamına gelir. Ancak, maksimum havuzlama katmanları neredeyse her zaman **(2, 2)** veya **(3, 3)** gibi 1'den büyük adım değerlerine sahip olur, ancak pencerenin kendisinden daha büyük olmaz.

Son olarak, **strides** değeri her iki yönde de aynı sayı olduğunda, yalnızca o sayıyı belirtmeniz yeterlidir; örneğin, **strides=(2, 2)** yerine, parametre ayarı için **strides=2** kullanabilirsiniz.

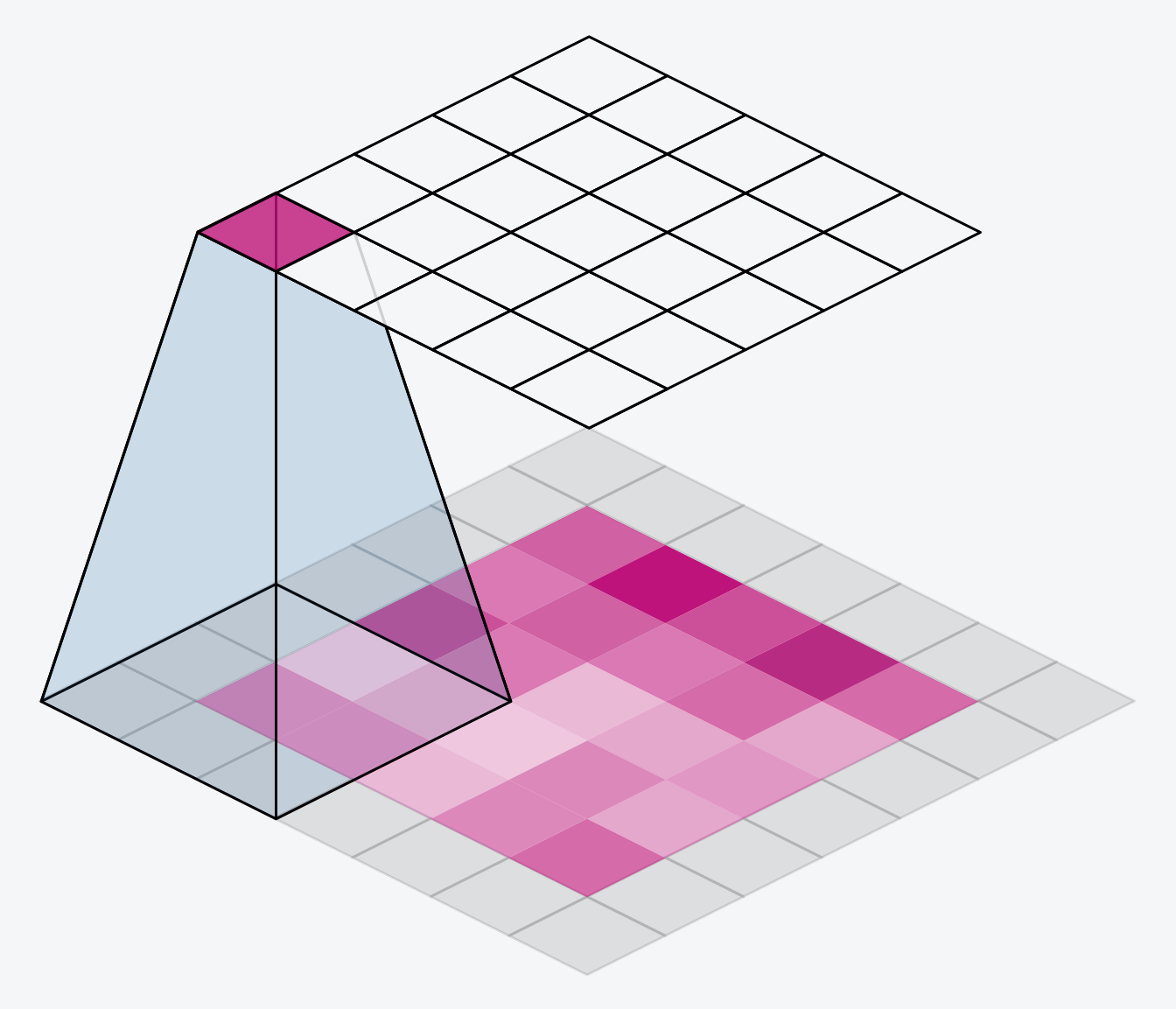
# Padding[¶](https://www.kaggle.com/code/ryanholbrook/the-sliding-window" \l "Padding" \t "_self)

Kayan pencere hesaplaması yapılırken, girdinin sınırlarında ne yapılacağı sorusu ortaya çıkar. Girdi görüntüsünün tamamen içinde kalmak, pencerenin, girdideki diğer tüm piksellerde olduğu gibi, bu sınır piksellerinin üzerine tam olarak oturmayacağı anlamına gelir. Tüm piksellere tam olarak aynı şekilde davranmadığımız için bir sorun olabilir mi?

Evrişimin bu sınır değerleriyle ne yapacağı, **dolgu** (padding) parametresiyle belirlenir. TensorFlow'da iki seçeneğiniz vardır: **padding='same'** veya **padding='valid'**. Her birinin kendi avantajları ve dezavantajları vardır.

**padding='valid'** ayarlandığında, evrişim penceresi tamamen girdinin içinde kalır. Dezavantajı, çıktının küçülmesidir (piksel kaybetmesidir) ve daha büyük çekirdekler için daha da küçülür. Bu durum, özellikle girdiler küçük boyutlu olduğunda, ağın içerebileceği katman sayısını sınırlar.

Alternatif olarak, **padding='same'** kullanılır. Buradaki hile, girdinin kenarlarına, çıktının boyutunu girdinin boyutuyla **aynı** yapmaya yetecek kadar 0 eklemektir. Ancak bu, kenarlardaki piksellerin etkisini azaltma etkisine sahip olabilir. Aşağıdaki animasyon, **'same'** dolgusu ile bir kayan pencereyi göstermektedir.



İncelediğimiz VGG modeli, tüm evrişimsel katmanları için aynı dolguyu kullanır. Çoğu modern evrişim ağı, ikisinin bir kombinasyonunu kullanır. (Ayarlanması gereken bir parametre daha!)

# Example - Exploring Sliding Windows[¶](https://www.kaggle.com/code/ryanholbrook/the-sliding-window#Example---Exploring-Sliding-Windows)

Kayan pencere parametrelerinin etkisini daha iyi anlamak için, düşük çözünürlüklü bir görüntüde özellik çıkarımını gözlemlemek ve böylece pikselleri tek tek görmek faydalı olabilir. Basit bir daireye bakalım.

Bir sonraki gizli hücre bizim için bir görüntü ve çekirdek oluşturacak.

import tensorflow as tf

import matplotlib.pyplot as plt

plt.rc('figure', autolayout=True)

plt.rc('axes', labelweight='bold', labelsize='large',

titleweight='bold', titlesize=18, titlepad=10)

plt.rc('image', cmap='magma')

image = circle([64, 64], val=1.0, r\_shrink=3)

image = tf.reshape(image, [\*image.shape, 1])

*# Bottom sobel*

kernel = tf.constant(

[[-1, -2, -1],

[0, 0, 0],

[1, 2, 1]],

)

show\_kernel(kernel)

VGG mimarisi oldukça basittir. 1'lik adımlarla evrişim ve **2x2'lik pencerelerle** ve 2'lik adımlarla maksimum havuzlama kullanır. visiontools yardımcı betiğine, tüm adımları bize gösterecek bir fonksiyon ekledik.

show\_extraction(

image, kernel,

*# Window parameters*

conv\_stride=1,

pool\_size=2,

pool\_stride=2,

subplot\_shape=(1, 4),

figsize=(14, 6),

)

Ve bu oldukça iyi çalışıyor! Çekirdek, yatay çizgileri algılayacak şekilde tasarlanmıştı ve ortaya çıkan özellik haritasında, girdinin daha yatay kısımlarının en yüksek aktivasyona sahip olduğunu görebiliyoruz.

Konvolüsyonun adım sayısını 3 olarak değiştirirsek ne olur?

show\_extraction(

image, kernel,

*# Window parameters*

conv\_stride=3,

pool\_size=2,

pool\_stride=2,

subplot\_shape=(1, 4),

figsize=(14, 6),

)

Bu, çıkarılan özelliğin kalitesini düşürüyor gibi görünüyor. Girdi dairemiz, sadece 1 piksel genişliğinde olduğu için oldukça "ince detaylı". 3 adımlı bir evrişim (convolution), ondan iyi bir özellik haritası üretmek için fazla kaba kalıyor.

Bazen, bir model ilk katmanında daha büyük adımlı bir evrişim kullanır. Bu genellikle daha büyük bir çekirdekle de birleştirilir. Örneğin, ResNet50 modeli, ilk katmanında 2 adımlı **7x7'lik çekirdekler** kullanır. Bu, girdiden çok fazla bilgi kaybetmeden büyük ölçekli özelliklerin üretimini hızlandırıyor gibi görünüyor.

# Conclusion[¶](https://www.kaggle.com/code/ryanholbrook/the-sliding-window" \l "Conclusion" \t "_self)

Bu derste, hem evrişim hem de havuzlama için ortak olan karakteristik bir hesaplamaya baktık: **kayan pencere** ve bu katmanlardaki davranışını etkileyen parametreler. Pencere tabanlı bu hesaplama stili, evrişimli ağların karakteristik özelliklerinin çoğunu oluşturur ve işleyişlerinin önemli bir parçasıdır.