

شبکه های مبتنی بر **encoder-decoder** به طور گسترده ای در مسائل **semantic segmentation** استفاده میشوند. موفقیت این شبکه ها را به ویژگی **skip connection** آنها نسبت میدهند که ویژگی های عمیق، معنادار و کلی **decoder** را با ویژگی های سطحی، سطح پایین و جزئی **encoder** ترکیب میکند.

با این وجود این ساختار **encoder-decoder** دو محدودیت دارد:

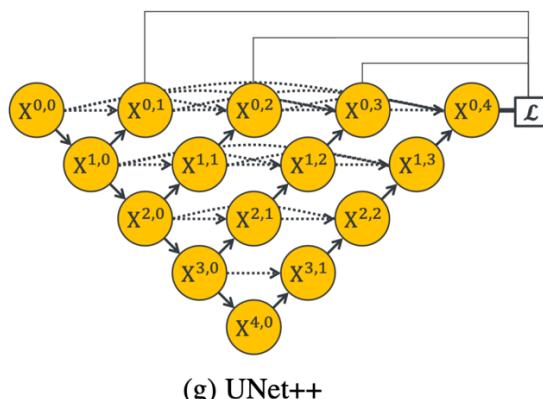
- ۱- عمق مناسب و بهینه این شبکه ها میتواند بسته به کاربرد های مختلف متفاوت باشد و به پیچیدگی آن مسئله و تعداد تصاویر لیبل دار موجود برای آموزش بستگی دارد. برای پیدا کردن عمق مناسب یک راه حل ساده این است که شبکه را برای عمق های مختلف به طور جداگانه آموزش دهیم و سپس در مرحله استنتاج نتایج آن ها را ترکیب کنیم که البته این کار بهینه نیست زیرا این شبکه ها به طور جداگانه آموزش میبینند و هنگام آموزش با هم همکاری نمیکنند.

۲- طراحی **skip connection** ها در این شبکه ها به طور غیر ضروری محدود کننده است زیرا الزام دارد تا فقط **feature map** های هم اندازه از **encoder** و **decoder** با هم ادغام شوند در صورتی که هیچ تضمینی وجود ندارد که این **feature map** ها برای ادغام مناسب هستند.

در این مقاله شبکه **unet++** معرفی میشود که سعی دارد بر دو محدودت بالا غلبه کند. شبکه ای است که **unet** های با عمق مختلف را در خود جای داده است در حالی که **decoder** های آن ها به صورت **dense** به هم متصل هستند و **skip connection** ها مجدداً طراحی شده اند و به این ترتیب مزیت های زیر ایجاد شده است:

۱- این شبکه مسئله انتخاب عمق مناسب را حل میکند زیرا در ساختار خود **unet** های با عمق مختلف را جای داده است، همه این **unet** ها تا حدودی **encoder** خود را با هم به اشتراک میگذارند، و **decoder** آن ها درهم تنیده است. در فرایند آموزش **unet++** هر کدام از **unet** ها به صورت همزمان آموزش میبینند و در عین حال هنگام آموزش با هم همکاری میکنند.

۲- معماری **unet++** خودش را به **skip connection** هایی که فقط اجازه میدهند فقط **feature map** های هم اندازه از **encoder** و **decoder** با هم ادغام شوند محدود نمیکند، بلکه **skip connection** های جدید اجازه میدهند **feature map** های با سایز متفاوت با هم ادغام شوند و به صورت دیکودر های کاملاً متصل در **resolution** یکسان پیاده سازی شده است. این ویژگی باعث شده است این شبکه در **semantic & instance segmentation** عملکرد بهتری نسبت به نسخه کلاسیک داشته باشد.



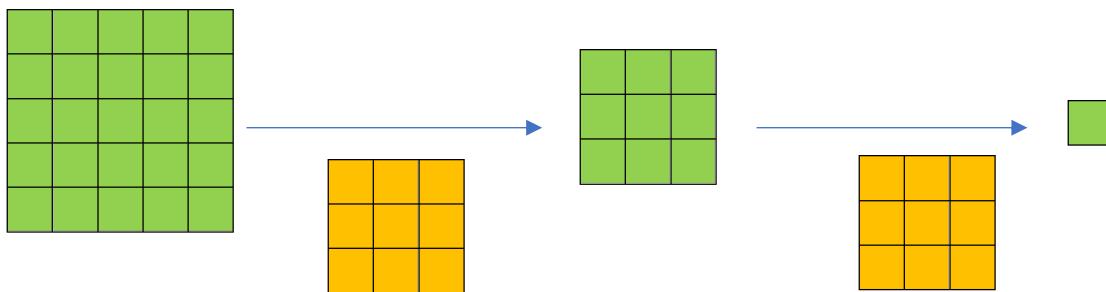
به نظر میرسد این شبکه بتواند هر دو مسئله semantic & instance segmentation را انجام دهد زیرا هم در متن مقاله به آن اشاره شده است و هم این دو مسئله به هم شباهت دارند؛ اما از آنجایی که پایه این شبکه unet است و در اصل برای semantic segmentation ارائه شده است، این شبکه نیز برای مسئله instance segmentation بهتر عمل میکند.

-۲

تعداد پارامتر ها در حالت اول:

$$(3 \times 3 + 1) + (3 \times 3 + 1) = 20$$

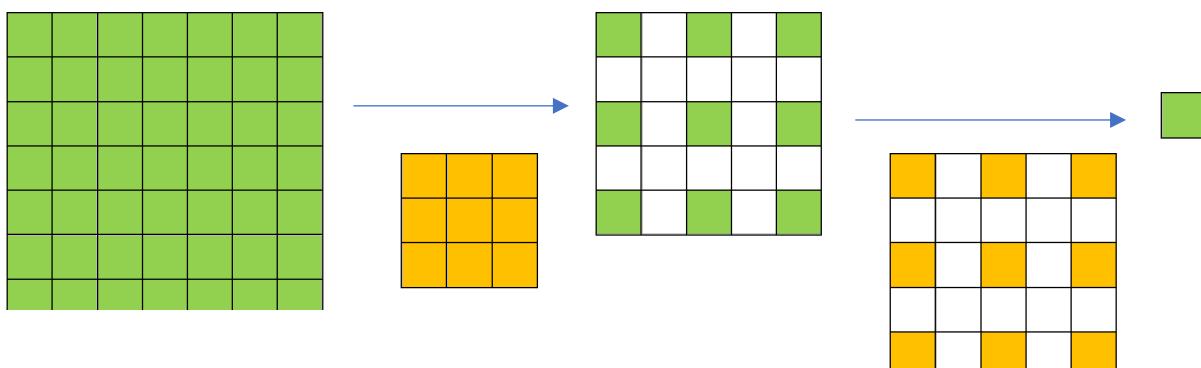
در این حالت ناحیه موثر یک پیکسل خروجی ابتدا یک ناحیه  $3 \times 3$  و سپس یک ناحیه  $5 \times 5$  است، در نتیجه ROFV نهایی یک ناحیه  $5 \times 5$  است.



تعداد پارامتر ها در حالت دوم نیز مشابه حالت اول است زیرا با اینکه کانولوشن دوم از هم باز تر شده ولی باز هم در واقع یک کانولوشن  $3 \times 3$  است:

$$(3 \times 3 + 1) + (3 \times 3 + 1) = 20$$

در این حالت ناحیه موثر یک پیکسل خروجی ابتدا یک ناحیه  $5 \times 5$  است، اگرچه تمام پیکسل های این ناحیه  $5 \times 5$  در محاسبات دخیل نیستند اما به دلیل پراکندگی پیکسل های موثر، این کانولوشن ناحیه بیشتری را پوشش داده است، در مرحله بعد ناحیه موثر یک ناحیه  $7 \times 7$  است در نتیجه ROFV نهایی یک ناحیه  $7 \times 7$  است.



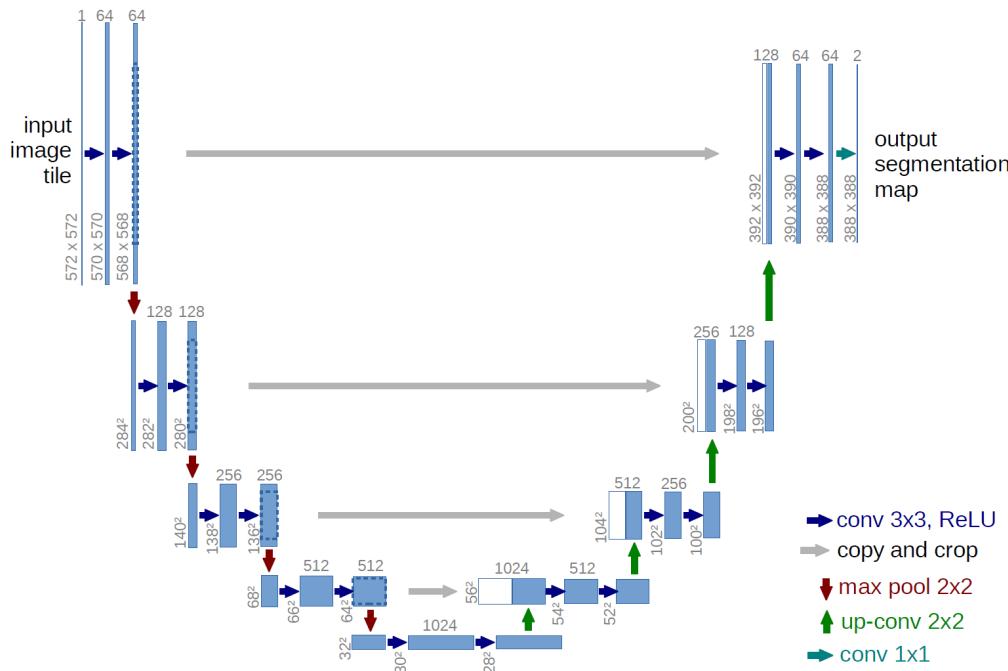
مشاهده میشود که با آن که تعداد پارامتر ها ثابت بودند، ناحیه موثر در حالت دوم افزایش یافته است.

ابتدا کتابخانه های لازم و دیتاست oxford\_iit\_pet را وارد میکنیم:

```
import tensorflow as tf
import tensorflow_datasets as tfds
import matplotlib.pyplot as plt
from IPython.display import clear_output
from keras.layers import *
from tensorflow import keras
import numpy as np
from tensorflow.keras import layers
from tensorflow.keras.models import Model
from tensorflow.keras.callbacks import ModelCheckpoint, ReduceLROnPlateau, EarlyStopping
import random
```

```
dataset, info = tfds.load('oxford_iit_pet:3.*.*', with_info=True)
```

حال باید ساختار شبکه unet خود را تعریف کنیم.  
بر اساس ساختار مطرح شده در نوت بوک لایه ها را تعریف میکنیم.



این ساختار سه قسمت encoder و decoder و bridge دارد.  
:Encoder

در هر مرحله از بخش encoder دو بار کانولوشن میگیریم و سپس با down sampling انجام میدهیم.  
نتایج هر مرحله قبل از down sampling را ذخیره میکنیم تا در مرحله decoder بتوانیم از آن ها استفاده کنیم، شبکه unet با ترکیب این نتایج که در واقع ویژگی های سطح پایین (سطحی) هستند با ویژگی های سطح بالایی (عمیق) که در مرحله decoding به دست می آید سعی میکند در عین یافتن ناحیه های معنادار، جزئیات تصویر اولیه را نیز حفظ کند.  
اگر این روش را به کار نگیرد جزئیات تصویر اولیه در فرایند encoding-decoding حذف میشوند.

:Decoder

در قسمت decoder ابتدا با استفاده از upsampling، transposed convolution انجام میدهیم، سپس نتایج ذخیره شده از مرحله encoder را با آن concatenate میکنیم و در آخر دو بار کانولوشن میگیریم.

:bridge

این قسمت صرفا پلی بین انکودر و دیکودر است و دو بار کانولوشن در آن انجام میشود

برای بهبود مازولاتی کد، برای هر قسمت یک بلاک تعریف کرده ایم (conv\_block, encoder\_block, decoder\_block)

در لایه آخر برای محاسبه خروجی یک کانولوشن  $1^*1$  میگیریم که کاهش بعد انجام دهد.

میتوانیم از وزن های pretrained شبکه resnet50 استفاده کنیم برای این منظور لایه هایی را از آن شبکه انتخاب میکنیم که خروجی با ابعاد یکسان با لایه متناظر در شبکه U-net داشته باشند و از وزن این لایه ها برای مقدار دهی اولیه لایه های قسمت bridge و encoder استفاده میکنیم. این کار باعث میشود شبکه از نقطه مناسب تری شروع به آموزش کند.

```
""" Pre-trained ResNet50 Model """
from tensorflow.keras.applications import ResNet50

""" Input """
input_shape = (128, 128, 3)

def conv_block(input, num_filters):
    x = Conv2D(num_filters, 3, padding="same")(input)
    x = BatchNormalization()(x)
    x = Activation("relu")(x)

    x = Conv2D(num_filters, 3, padding="same")(x)
    x = BatchNormalization()(x)
    x = Activation("relu")(x)

    return x

def encoder_block(input, num_filters):
    x = conv_block(input, num_filters)
    p = MaxPool2D((2, 2))(x)
    return x, p

def decoder_block(input, skip_features, num_filters):
    x = Conv2DTranspose(num_filters, (2, 2), strides=2, padding="same")(input)
    x = Concatenate()([x, skip_features])
    x = conv_block(x, num_filters)
    return x

def build_unet(input_shape, num_classes, pre_trained):
    inputs = Input(input_shape)
    resnet50 = ResNet50(include_top=False, weights="imagenet", input_tensor=inputs)
    if pre_trained:
        """ Encoder """
        s1 = resnet50.get_layer("input_1").output      ## (512 x 512)
        s2 = resnet50.get_layer("conv1_relu").output    ## (256 x 256)
        s3 = resnet50.get_layer("conv2_block3_out").output ## (128 x 128)
        s4 = resnet50.get_layer("conv3_block4_out").output ## (64 x 64)

        """ Bridge """
        b1 = resnet50.get_layer("conv4_block6_out").output ## (32 x 32)

    else:
        s1, p1 = encoder_block(inputs, 64)
        s2, p2 = encoder_block(p1, 128)
        s3, p3 = encoder_block(p2, 256)
        s4, p4 = encoder_block(p3, 512)

        """ Bridge """
        b1 = conv_block(p4, 1024)

    """ Decoder """
    d1 = decoder_block(b1, s4, 512)
    d2 = decoder_block(d1, s3, 256)
    d3 = decoder_block(d2, s2, 128)
    d4 = decoder_block(d3, s1, 64)

    outputs = Conv2D(num_classes, 1, padding="same", activation="softmax")(d4)

    if pre_trained:
        model = Model(inputs, outputs, name="Pretrained")
    else:
        model = Model(inputs, outputs, name="U-Net")
    return model
```

این دیتابیس ۳۷ کلاس مختلف دارد و هر کدام یک تصویر با ابعاد  $3 \times 128 \times 128$  است.

Learning rate را  $10^{-4}$  در نظر میگیریم و شبکه را run میکنیم.

تعداد گام های لازم براساس نسبت سایز دیتاست به سایز دسته های دیتا (batch size) به دست می آید.

```
""" Hyperparameters """
shape = (image_size, image_size, 3)
num_classes = 37
lr = 1e-4

train_steps = len(train_dataset)//BATCH_SIZE
valid_steps = VALIDATION_SIZE//BATCH_SIZE

callbacks = [
    ModelCheckpoint("model.h5", verbose=1, save_best_model=True),
    ReduceLROnPlateau(monitor="val_loss", patience=3, factor=0.1, verbose=1, min_lr=1e-6),
    EarlyStopping(monitor="val_loss", patience=5, verbose=1)
]

""" Model """
model = build_unet(shape, num_classes, pre_trained=False)
model.compile(optimizer=tf.keras.optimizers.Adam(lr), loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=[tf.keras.metrics.SparseCategoricalAccuracy(name="accuracy")])

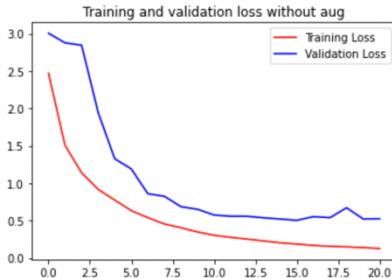
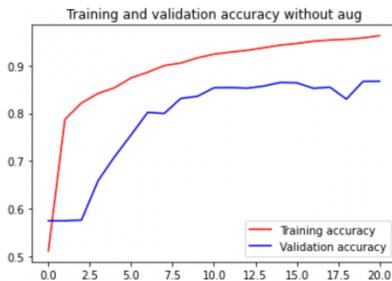
history = model.fit(train_batches,
                     steps_per_epoch=train_steps,
                     validation_data=validation_batches,
                     validation_steps=valid_steps,
                     epochs=NUM_EPOCHS,
                     callbacks=callbacks
)

57/57 [=====] - ETA: 0s - loss: 2.4699 - accuracy: 0.5122
Epoch 1: saving model to model.h5
57/57 [=====] - 93s 1s/step - loss: 2.4699 - accuracy: 0.5122 - val_loss: 3.0055 - val_accuracy: 0.5752 - lr: 1.0000e-04
Epoch 2/30
57/57 [=====] - ETA: 0s - loss: 1.5052 - accuracy: 0.7873
Epoch 2: saving model to model.h5
57/57 [=====] - 77s 1s/step - loss: 1.5052 - accuracy: 0.7873 - val_loss: 2.8767 - val_accuracy: 0.5752 - lr: 1.0000e-04
Epoch 3/30
57/57 [=====] - ETA: 0s - loss: 1.1393 - accuracy: 0.8218
Epoch 3: saving model to model.h5
57/57 [=====] - 69s 1s/step - loss: 1.1393 - accuracy: 0.8218 - val_loss: 2.8453 - val_accuracy: 0.5769 - lr: 1.0000e-04
Epoch 4/30
57/57 [=====] - ETA: 0s - loss: 0.9170 - accuracy: 0.8416
Epoch 4: saving model to model.h5
57/57 [=====] - 69s 1s/step - loss: 0.9170 - accuracy: 0.8416 - val_loss: 1.9445 - val_accuracy: 0.6591 - lr: 1.0000e-04
Epoch 5/30
57/57 [=====] - ETA: 0s - loss: 0.7742 - accuracy: 0.8537
.

.

.

Epoch 19: ReduceLROnPlateau reducing learning rate to 9.999999747378752e-06.
57/57 [=====] - 70s 1s/step - loss: 0.1480 - accuracy: 0.9550 - val_loss: 0.6713 - val_accuracy: 0.8299 - lr: 1.0000e-04
Epoch 20/30
57/57 [=====] - ETA: 0s - loss: 0.1391 - accuracy: 0.9579
Epoch 20: saving model to model.h5
57/57 [=====] - 69s 1s/step - loss: 0.1391 - accuracy: 0.9579 - val_loss: 0.5215 - val_accuracy: 0.8668 - lr: 1.0000e-05
Epoch 21/30
57/57 [=====] - ETA: 0s - loss: 0.1258 - accuracy: 0.9626
Epoch 21: saving model to model.h5
57/57 [=====] - 69s 1s/step - loss: 0.1258 - accuracy: 0.9626 - val_loss: 0.5245 - val_accuracy: 0.8671 - lr: 1.0000e-05
Epoch 21: early stopping
```



آموزش شبکه با نتایج زیر پایان یافت:

Loss= 0.1258      accuracy=0.9626

(ب)

حال با استفاده از **data augmentation** دیتاست خود را بزرگتر میکنیم و مجددا شبکه را **run** میکنیم.  
برای این منظور تابع **augment** به صورت زیر تعریف میشود:

```
def augment(input_image, input_mask):
    # your code here #
    # write suitable Augmentation transforms

    input_image = tf.cast(input_image, tf.float32)
    input_mask = tf.cast(input_mask, tf.float32)
    # zoom in a bit
    if tf.random.uniform() > 0.5:
        # use original image to preserve high resolution
        input_image = tf.image.central_crop(input_image, 0.75)
        input_mask = tf.image.central_crop(input_mask, 0.75)
    # resize
    input_image = tf.image.resize(input_image, (image_size, image_size))
    input_mask = tf.image.resize(input_mask, (image_size, image_size))

    # random brightness adjustment illumination
    input_image = tf.image.random_brightness(input_image, 0.3)
    # random contrast adjustment
    input_image = tf.image.random_contrast(input_image, 0.2, 0.5)

    # flipping random horizontal or vertical
    if tf.random.uniform() > 0.5:
        input_image = tf.image.flip_left_right(input_image)
        input_mask = tf.image.flip_left_right(input_mask)
    if tf.random.uniform() > 0.5:
        input_image = tf.image.flip_up_down(input_image)
        input_mask = tf.image.flip_up_down(input_mask)

    # rotation in 90°s
    k = random.randint(1, 3)
    input_image = tf.image.rot90(input_image,k)
    input_mask = tf.image.rot90(input_mask,k)

    return input_image, input_mask
```

این تابع تصویر و ماسک متناظر با آن را با روش های مختلف از جمله چرخش، بزرگنمایی، تغییر شدت روشنایی، تغییر کنترast و یا برگرداندن (**flip**) تغییر میدهد.

```

def load_image_train_aug(datapoint):
    input_image = datapoint["image"]
    input_mask = datapoint["segmentation_mask"]
    input_image, input_mask = resize(input_image, input_mask)
    input_image, input_mask = augment(input_image, input_mask)
    input_image, input_mask = normalize(input_image, input_mask)

    return input_image, input_mask

train_dataset = dataset["train"].map(load_image_train, num_parallel_calls=tf.data.AUTOTUNE)
train_dataset_aug = dataset["train"].map(load_image_train_aug, num_parallel_calls=tf.data.AUTOTUNE)

test_dataset = dataset["test"].map(load_image_test, num_parallel_calls=tf.data.AUTOTUNE)

```

در این قسمت ابتدا تابع `augment` را روی تصاویر اعمال میکنیم.

```

BATCH_SIZE = 64
BUFFER_SIZE = 1000
VALIDATION_SIZE = 3000
TEST_SIZE = 669

train_batches_aug = train_dataset_aug.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE).repeat()
train_batches_aug = train_batches_aug.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)

train_batches = train_dataset.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE).repeat()
train_batches = train_batches.prefetch(buffer_size=tf.data.experimental.AUTOTUNE)

validation_batches = test_dataset.take(VALIDATION_SIZE).batch(BATCH_SIZE)
test_batches = test_dataset.skip(VALIDATION_SIZE).take(TEST_SIZE).batch(BATCH_SIZE)

```

این بخش ابتدا تصاویر را به صورت بخش های ۱۰۰۰ تایی به صورت رندوم انتخاب میکند (`shuffle`) و سپس آن ها را به دسته های ۶۴ تایی تقسیم میکند (`batch`), هنگام آموزش تصاویر به صورت دسته ای به شبکه داده میشوند و به ازای هر دسته خطای محاسبه و وزن ها به روزرسانی میشود.

برای نمونه میتوانیم یکی از تصاویر را مشاهده کنیم:

```

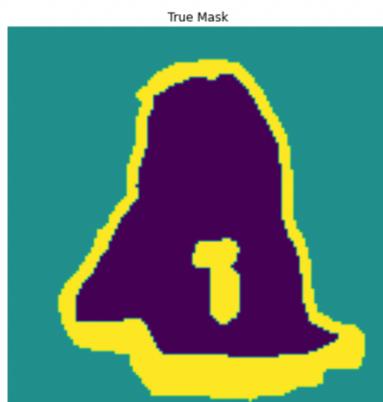
def display(display_list):
    plt.figure(figsize=(15, 15))

    title = ["Input Image", "True Mask", "Predicted Mask"]

    for i in range(len(display_list)):
        plt.subplot(1, len(display_list), i+1)
        plt.title(title[i])
        plt.imshow(tf.keras.utils.array_to_img(display_list[i]))
        plt.axis("off")
    plt.show()

sample_batch = next(iter(train_batches_aug))
random_index = np.random.choice(sample_batch[0].shape[0])
sample_image, sample_mask = sample_batch[0][random_index], sample_batch[1][random_index]
display([sample_image, sample_mask])

```



```

"""
Hyperparameters """
shape = (image_size, image_size, 3)
num_classes = 37
lr = 1e-4

train_steps = len(train_dataset_aug)//BATCH_SIZE
valid_steps = VALIDATION_SIZE//BATCH_SIZE

callbacks = [
    ModelCheckpoint("model_aug.h5", verbose=1, save_best_model=True),
    ReduceLROnPlateau(monitor="val_loss", patience=3, factor=0.1, verbose=1, min_lr=1e-6),
    EarlyStopping(monitor="val_loss", patience=5, verbose=1)
]

"""
Model """
model_aug = build_unet(shape, num_classes, pre_trained=False)
model_aug.compile(optimizer=tf.keras.optimizers.Adam(lr), loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  metrics=[tf.keras.metrics.SparseCategoricalAccuracy(name="accuracy")])

history_aug = model_aug.fit(train_batches_aug,
                            steps_per_epoch=train_steps,
                            validation_data=validation_batches,
                            validation_steps=valid_steps,
                            epochs=NUM_EPOCHS,
                            callbacks=callbacks
)

```

پارامتر های شبکه را تنظیم میکنیم و شبکه را run میکنیم.

```

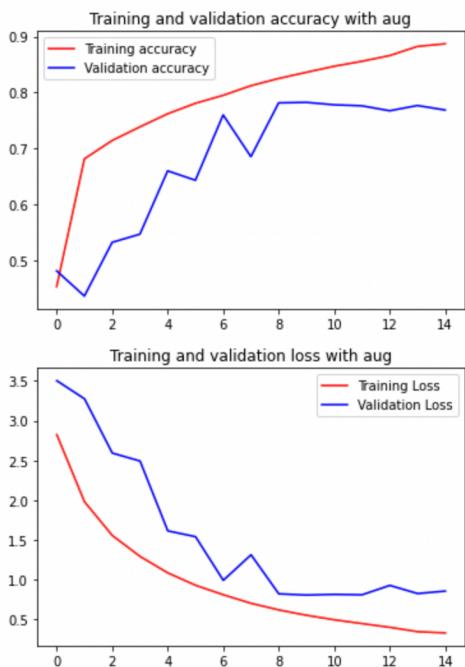
Epoch 1/30
57/57 [=====] - ETA: 0s - loss: 2.8239 - accuracy: 0.4538
Epoch 1: saving model to model_aug.h5
57/57 [=====] - 77s 1s/step - loss: 2.8239 - accuracy: 0.4538 - val_loss: 3.5029 - val_accuracy: 0.4820 - lr: 1.0000e-04
Epoch 2/30
57/57 [=====] - ETA: 0s - loss: 1.9793 - accuracy: 0.6818
Epoch 2: saving model to model_aug.h5
57/57 [=====] - 78s 1s/step - loss: 1.9793 - accuracy: 0.6818 - val_loss: 3.2746 - val_accuracy: 0.4370 - lr: 1.0000e-04
Epoch 3/30
57/57 [=====] - ETA: 0s - loss: 1.5563 - accuracy: 0.7145
Epoch 3: saving model to model_aug.h5
57/57 [=====] - 70s 1s/step - loss: 1.5563 - accuracy: 0.7145 - val_loss: 2.5926 - val_accuracy: 0.5328 - lr: 1.0000e-04
Epoch 4/30
57/57 [=====] - ETA: 0s - loss: 1.2930 - accuracy: 0.7389
Epoch 4: saving model to model_aug.h5
57/57 [=====] - 70s 1s/step - loss: 1.2930 - accuracy: 0.7389 - val_loss: 2.4927 - val_accuracy: 0.5474 - lr: 1.0000e-04
Epoch 5/30
57/57 [=====] - ETA: 0s - loss: 1.0860 - accuracy: 0.7622
.

.

.

Epoch 13: ReduceLROnPlateau reducing learning rate to 9.999999747378752e-06.
57/57 [=====] - 70s 1s/step - loss: 0.4017 - accuracy: 0.8660 - val_loss: 0.9284 - val_accuracy: 0.7676 - lr: 1.0000e-04
Epoch 14/30
57/57 [=====] - ETA: 0s - loss: 0.3468 - accuracy: 0.8823
Epoch 14: saving model to model_aug.h5
57/57 [=====] - 70s 1s/step - loss: 0.3468 - accuracy: 0.8823 - val_loss: 0.8262 - val_accuracy: 0.7769 - lr: 1.0000e-05
Epoch 15/30
57/57 [=====] - ETA: 0s - loss: 0.3306 - accuracy: 0.8871
Epoch 15: saving model to model_aug.h5
57/57 [=====] - 71s 1s/step - loss: 0.3306 - accuracy: 0.8871 - val_loss: 0.8572 - val_accuracy: 0.7689 - lr: 1.0000e-05
Epoch 15: early stopping

```



## آموزش شبکه با نتایج زیر پایان یافت:

Loss= 0.3306 accuracy=0.8871

با اینکه انتظار میرفت شبکه نتایج بهتری نسبت به حالت بدون augmentation ارائه دهد ولی اینگونه نشد!!!

ج)

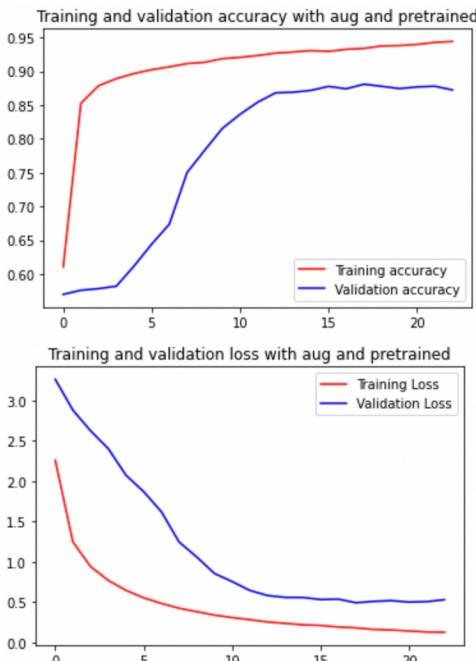
حال شبکه را با استفاده از وزن های `res50` شده شبکه `pretrain` مقدار دهی اولیه میکنیم و `run` میکنیم. جزئیات پیاده سازی این بخش (`pretrain`) در قسمت اول (ساختار شبکه) شرح داده شد.

```

57/57 [=====] - ETA: 0s - loss: 2.2570 - accuracy: 0.6102
Epoch 1: saving model to model_aug_pretrained.h5
57/57 [=====] - 86s 1s/step - loss: 2.2570 - accuracy: 0.6102 - val_loss: 3.2611 - val_accuracy: 0.5697 - lr: 1.0000e-04
Epoch 2/30
57/57 [=====] - ETA: 0s - loss: 1.2451 - accuracy: 0.8525
Epoch 2: saving model to model_aug_pretrained.h5
57/57 [=====] - 66s 1s/step - loss: 1.2451 - accuracy: 0.8525 - val_loss: 2.8785 - val_accuracy: 0.5758 - lr: 1.0000e-04
Epoch 3/30
57/57 [=====] - ETA: 0s - loss: 0.9410 - accuracy: 0.8787
Epoch 3: saving model to model_aug_pretrained.h5
57/57 [=====] - 59s 1s/step - loss: 0.9410 - accuracy: 0.8787 - val_loss: 2.6223 - val_accuracy: 0.5782 - lr: 1.0000e-04
Epoch 4/30
57/57 [=====] - ETA: 0s - loss: 0.7689 - accuracy: 0.8890
Epoch 4: saving model to model_aug_pretrained.h5
57/57 [=====] - 59s 1s/step - loss: 0.7689 - accuracy: 0.8890 - val_loss: 2.4016 - val_accuracy: 0.5818 - lr: 1.0000e-04
.
.
.

Epoch 21: ReduceLROnPlateau reducing learning rate to 9.99999747378752e-06.
57/57 [=====] - 59s 1s/step - loss: 0.1426 - accuracy: 0.9398 - val_loss: 0.5004 - val_accuracy: 0.8768 - lr: 1.0000e-04
Epoch 22/30
57/57 [=====] - ETA: 0s - loss: 0.1307 - accuracy: 0.9427
Epoch 22: saving model to model_aug_pretrained.h5
57/57 [=====] - 59s 1s/step - loss: 0.1307 - accuracy: 0.9427 - val_loss: 0.5061 - val_accuracy: 0.8779 - lr: 1.0000e-05
Epoch 23/30
57/57 [=====] - ETA: 0s - loss: 0.1268 - accuracy: 0.9442
Epoch 23: saving model to model_aug_pretrained.h5
57/57 [=====] - 59s 1s/step - loss: 0.1268 - accuracy: 0.9442 - val_loss: 0.5300 - val_accuracy: 0.8724 - lr: 1.0000e-05
Epoch 23: early stopping

```



آموزش شبکه با نتایج زیر پایان یافت:

Loss= 0.1268      accuracy=0.9442

همانگونه که انتظار میرفت با استفاده از این روش نتایج خوبی حاصل شد. دلیل آن این است که مقدار اولیه وزن های شبکه را از نقطه خوبی شروع کردیم و از وزن های شبکه ای که قبلاً آموزش داده شده بود استفاده کردیم.

حال باید تعدادی تصویر تست را به شبکه ارائه کنیم و پاسخ شبکه را بسنجیم، برای این منظور تابع predict را تعریف میکنیم که به صورت زیر است:

```

predictor = load_model('model_aug_pretrained.h5')
predicts = predictor.predict(test_batches)

def display(display_list):
    plt.figure(figsize=(15, 15))

    title = ["Input Image", "True Mask", "Predicted Mask"]

    for i in range(len(display_list)):
        plt.subplot(1, len(display_list), i+1)
        plt.title(title[i])
        plt.imshow(tf.keras.utils.array_to_img(display_list[i]))
        plt.axis("off")
    plt.show()

sample_batch = next(iter(test_batches))
random_index = np.random.choice(sample_batch[0].shape[0])
print(random_index)
sample_image, sample_mask = sample_batch[0][random_index], sample_batch[1][random_index]
display([sample_image, sample_mask, np.expand_dims(np.argmax(predicts[random_index], axis=-1), axis=-1)])

predictor.evaluate(test_batches)

```

11/11 [=====] - 15s 604ms/step

14

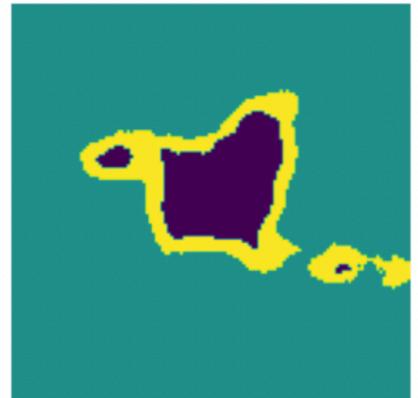
Input Image



True Mask



Predicted Mask



11/11 [=====] - 13s 351ms/step - loss: 0.5252 - accuracy: 0.8741

[0.5252105593681335, 0.8741411566734314]

11/11 [=====] - 12s 352ms/step

35

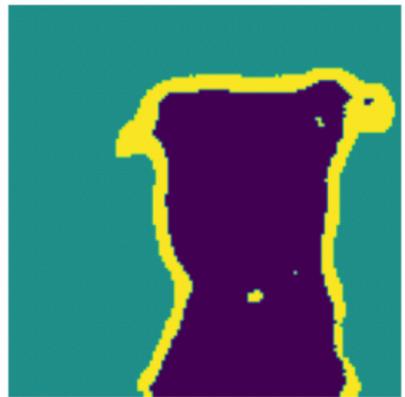
Input Image



True Mask



Predicted Mask

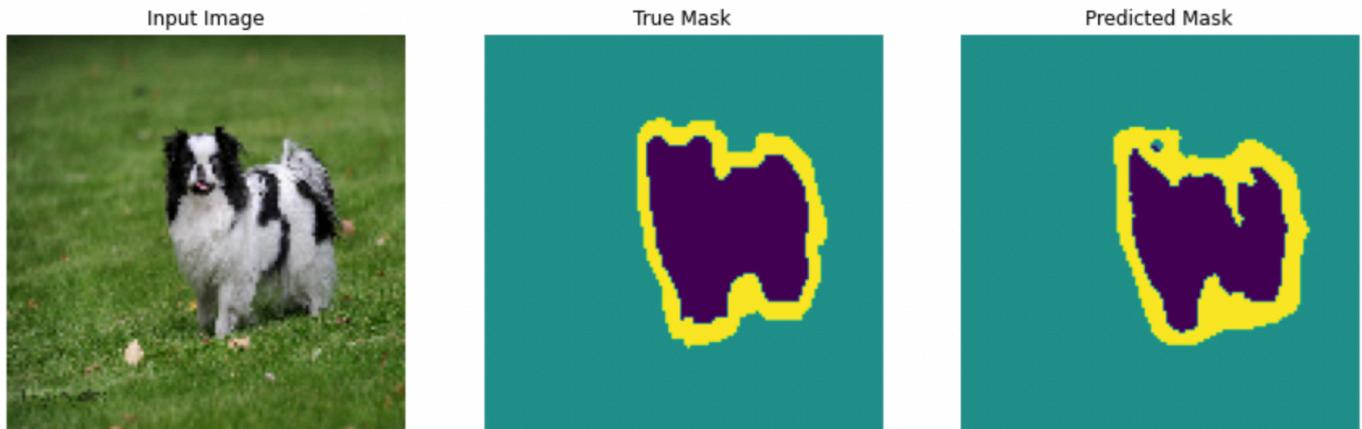


11/11 [=====] - 13s 353ms/step - loss: 0.5252 - accuracy: 0.8741

[0.5252105593681335, 0.8741411566734314]

```
11/11 [=====] - 14s 349ms/step
```

47



```
11/11 [=====] - 13s 366ms/step - loss: 0.5252 - accuracy: 0.8741  
[0.5252105593681335, 0.8741411566734314]
```

که نتایج قابل قبولی است.

## References:

<https://stackoverflow.com/questions/65475057/keras-data-augmentation-pipeline-for-image-segmentation-dataset-image-and-mask>

<https://github.com/nikhilroxtomar/Semantic-Segmentation-Architecture/blob/main/TensorFlow/unet.py>

[https://github.com/nikhilroxtomar/Semantic-Segmentation-Architecture/blob/main/TensorFlow/resnet50\\_unet.py](https://github.com/nikhilroxtomar/Semantic-Segmentation-Architecture/blob/main/TensorFlow/resnet50_unet.py)

## Transposed convolution

برخی منابع آن را **deconvolution** نامیده اند در صورتی که اشتباه است، عملگر **عمل عکس کانولوشن** را انجام میدهد به این صورت که اگر تصویر نتیجه یک کانولوشن را به آن بدهیم، تصویر اولیه را به ما باز برمیگرداند در صورتی که **Transposed convolution** تصویر اولیه را به ما نمیدهد و فقط از این نظر با **deconvolution** شباهت دارد که خروجی آن با تصویر اولیه هم اندازه است ولی از لحاظ محاسبات با **deconvolution** متفاوت است.

طريقه عملکرد آن به اين صورت است که ابتدا بر روی ورودی يك سري پدينگ خاص اعمال ميکند سپس تعدادی عملگر کانولوشن عادي روی آن اعمال ميکند، نتيجه يك تصویر با ابعاد تصویر اولیه است.  
این عملگر درواقع ترکیب **upsampling** و **convolution** است و میتواند به عنوان **upsampling** هم استفاده شود.

### image references:

A guide to convolution arithmetic for deep learning, Vincent Dumoulin and Francesco Visin

MILA, Université de Montréal ,AIRLab, Politecnico di Milano

January 12, 2018

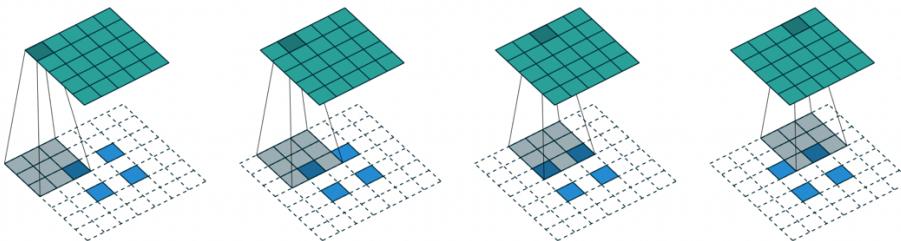


Figure 4.5: The transpose of convolving a  $3 \times 3$  kernel over a  $5 \times 5$  input using  $2 \times 2$  strides (i.e.,  $i = 5$ ,  $k = 3$ ,  $s = 2$  and  $p = 0$ ). It is equivalent to convolving a  $3 \times 3$  kernel over a  $2 \times 2$  input (with 1 zero inserted between inputs) padded with a  $2 \times 2$  border of zeros using unit strides (i.e.,  $i' = 2$ ,  $\tilde{i}' = 3$ ,  $k' = k$ ,  $s' = 1$  and  $p' = 2$ ).

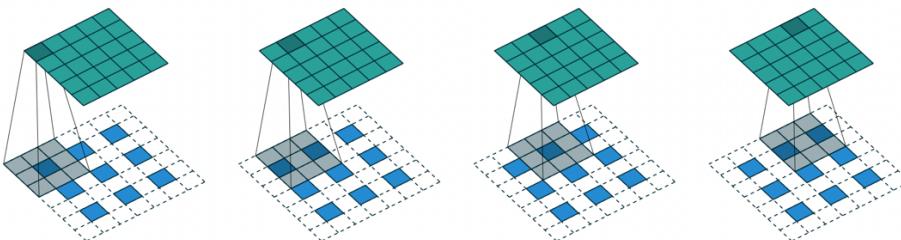


Figure 4.6: The transpose of convolving a  $3 \times 3$  kernel over a  $5 \times 5$  input padded with a  $1 \times 1$  border of zeros using  $2 \times 2$  strides (i.e.,  $i = 5$ ,  $k = 3$ ,  $s = 2$  and  $p = 1$ ). It is equivalent to convolving a  $3 \times 3$  kernel over a  $3 \times 3$  input (with 1 zero inserted between inputs) padded with a  $1 \times 1$  border of zeros using unit strides (i.e.,  $i' = 3$ ,  $\tilde{i}' = 5$ ,  $k' = k$ ,  $s' = 1$  and  $p' = 1$ ).

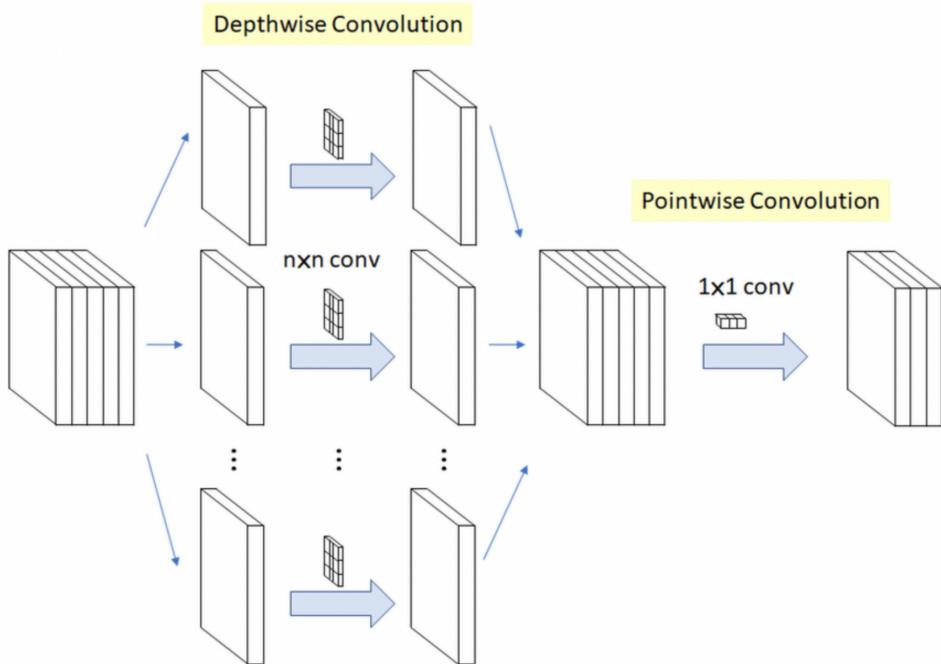
## Separable Convolutions

در اين روش کرنل کانولوشن را به حاصل ضرب چند کرنل با بعد کمتر ميشكنيم، برای مثال میتوان کرنل دو بعدی  $k$  را بتوان به صورت  $k = k_1 \cdot k_2$  به صورت ضرب داخلی دو کرنل يك بعدی  $k_1$  و  $k_2$  شکست و حاصل را با گرفتن دو بار کانولوشن يك بعدی به دست آورد،

به اين ترتيب اگر برای مثال کرنل  $k = 3 \times 3$  باشد به جای محاسبه با 9 پارامتر میتوان با 6 پارامتر محاسبات را انجام داد.  
این نوع **spatial separable convolution** نامیده ميشود که در دیپ لرنینگ کاربرد زیادی ندارد.

<https://www.analyticsvidhya.com/blog/2021/11/an-introduction-to-separable-convolutions/>

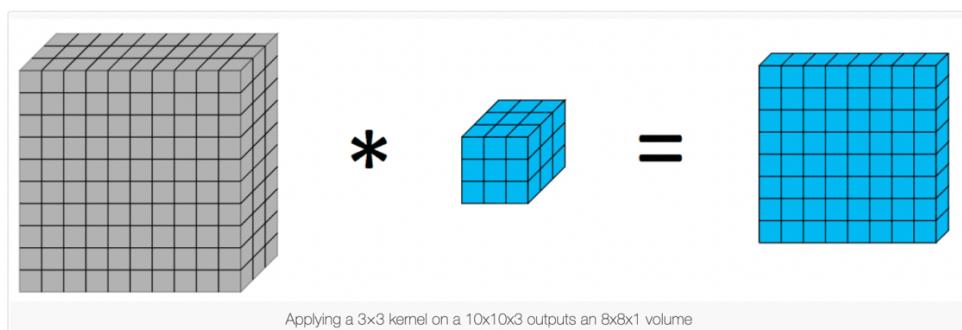
صورت دیگر آن که کاربرد زیادی در دیپ لرنینگ دارد depthwise separable convolution نامیده میشود که به صورت زیر عمل میکند:



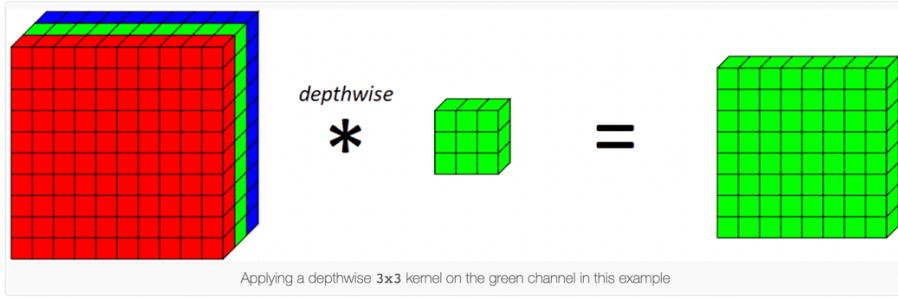
این روش ابتدا از هر کدام از کانال های ورودی جداگانه کانولوشن دو بعدی میگیرد سپس نتایج را کنار هم قرار میدهد و اینبار به صورت عمقی کانولوشن میگیرد.  
میتوان با یک مثال دیگر این روش را واضح تر توضیح داد

<https://machinelearningmastery.com/using-depthwise-separable-convolutions-in-tensorflow/>

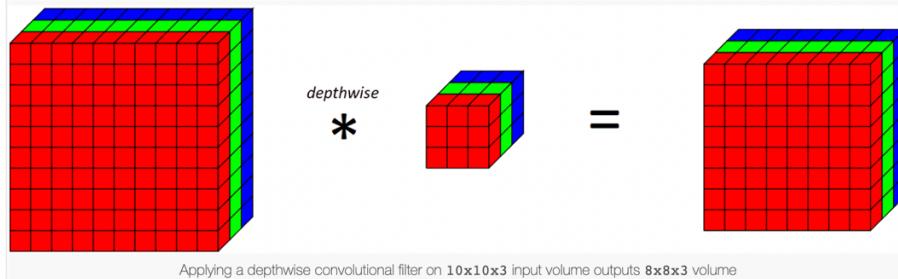
فرض کنید یک تصویر با سه کانال سبز، آبی و قرمز داریم و میخواهیم با یک کرنل  $3 \times 10 \times 3$  کانولوشن بگیریم.  
اگر به صورت یکجا و با یک کرنل سه بعدی کانولوشن بگیریم تعداد پارامتر های کانولوشن  $10 \times 10 \times 3 = 300$  پارامتر میشود



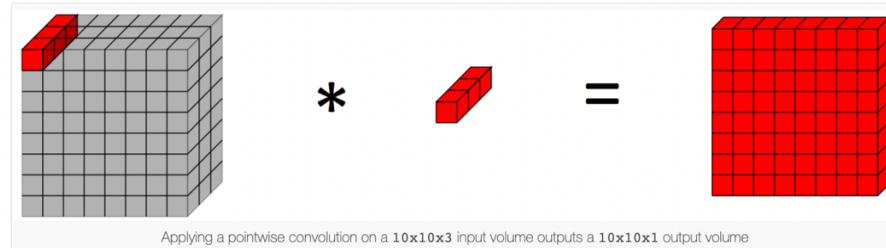
اما اگر آن را به دو مرحله بشکنیم یعنی ابتدا روی هر کانال به صورت جداگانه کانولوشن دو بعدی بگیریم:



سپس نتایج را کنار هم قرار دهیم:



و در آخر به صورت عمقی کانولوشن بگیریم:



تعداد پارامتر ها به  $3 \times (3 \times 3) + 3 = 30$  پارامتر کاهش می یابد.