Computer Vision
Dr. Mohammadi
Fall 2022
Hoorieh Sabzevari - 98412004
HW11

۱. الف) معایب استفاده از لایه‌ی کاملا متصل این است که اولا به صورت *locally* ویژگی استخراج نمی‌کند، در صورتی که ویژگی‌های یک تصویر وابسته به مختصات آن است و در پردازش تصویر مختصات هم برای ما اهمیت دارد. همچنین تعداد پارامترهای لایه‌ی کاملا متصل هم بسیار بالاست و شبکه را حجیم و کند می‌کند. شبکه‌های کانولوشنی علاوه بر اینکه ویژگی‌های محلی را برای ما استخراج می‌کنند، تعداد پارامترهای کمتری دارند لذا سرعت بالاتری خواهند داشت. لایه‌های کاملا متصل معمولاً در لایه‌های بالای شبکه قرار می‌گیرند، در نقطه ای که ورودی (توسط لایه های کانولوشنی قبلی) به نمایش فشرده‌ای از ویژگی‌ها کاهش یافته است. لذا می‌توانیم (تقریبا) لایه‌های *conv* را به‌عنوان شکستن ورودی (مثلاً یک تصویر) به ویژگی‌های مشترک و لایه‌های FC را به‌عنوان ترکیب کردن این ویژگی‌ها در نظر بگیریم. (لینک کمکی) (۱۰ دقیقه)

ب) اگر *padding* نداشته باشیم، ابعاد خروجی به اندازه‌ی یکی کمتر از سایز فیلتر کاهش پیدا خواهد کرد و عمق آن نیز به اندازه‌ی تعداد فیلترها است، پس $16 \times 12 \times 12$ خواهد شد، لذا باید دو سطر از بالا و پایین و دو ستون از چپ و راست *padding* اضافه کنیم تا ابعاد خروجی با ابعاد ورودی یکسان باشد.*(padding=4)*

تعداد پارامترهای آن هم به صورت زیر محاسبه خواهد شد:

$$\big((shape\ of\ width\ of\ filter\ *\ shape\ of\ height\ filter\ *\ number\ of\ filters\ in\ the\ previous\ layer + 1) * number\ of\ filters\big) = (5 \times 5 \times 5 + 1) \times 16 = 2016$$

که ۱۶ تعداد فیلترها و ۱ برای *bias* اضافه شده است. (۱۰ دقیقه)

پ) طبق توضیحات بخش قبل ابعاد خروجی به صورت $3 \times 28 \times 28$ خواهد بود. اندازه‌ی گام ۱ هم یعنی تمام پیکسل‌ها را حساب کنیم و تاثیری در ابعاد نخواهد داشت.

برای قسمت دوم، پس از یک لایه‌ی کانولوشنی ابعاد تصویر خروجی به صورت $9 \times 30 \times 30$ خواهد بود، چون به اندازه‌ی یکی کمتر از ابعاد فیلتر یعنی ۲ از ابعاد ورودی کم می‌شود و عمق آن هم تعداد فیلترهای اعمال شده است. پس از اعمال دوباره‌ی این لایه خروجی به صورت $9 \times 28 \times 28$ می‌شود. (۱۰ دقیقه)

ت) ***Pooling*** عملیاتی است که برای کاهش مقیاس تصویر استفاده می‌شود و در صورت عدم استفاده و جایگزینی آن با *Convolution* برای استخراج مهمترین ویژگی‌ها هزینه محاسباتی بالایی را می‌طلبد. ما از *Max Pooling* برای استخراج حداکثر مقدار از نقشه ویژگی با توجه به اندازه فیلتر و گام‌ها استفاده می‌کنیم. *Max Pooling* فقط پیکسل هایی با حداکثر مقدار را ذخیره می‌کند. این مقادیر در نقشه ویژگی، اهمیت یک ویژگی و مکان آن را نشان می‌دهد. بنابراین، گرفتن تنها مقدار حداکثر به معنای استخراج مهم‌ترین ویژگی در یک منطقه است.

سه دلیل برای استفاده از *Max Pooling*:

۱. کاهش مقیاس تصویر با استخراج مهم‌ترین ویژگی در نتیجه کمتر شدن تعداد پارامترها

۲. حذف تغییر ناپذیری مانند تغییر موقعیت، چرخشی و مقیاس

۳. کاهش *overfitting*

تغییرناپذیری در تصاویر مهم است اگر ما به وجود یک ویژگی به جای اینکه دقیقاً در کجاست اهمیت دهیم. در زمینه‌های دیگر، حفظ مکان یک ویژگی مهم‌تر است.

(لینک کمکی)

Max Pooling پیکسل های روشن تر را از تصویر انتخاب می‌کند. زمانی مفید است که پس‌زمینه تصویر تاریک است و ما فقط به دنبال پیکسل‌های روشن‌تر تصویر هستیم. به عنوان مثال: در مجموعه داده *MNIST*، ارقام به رنگ سفید و پس‌زمینه سیاه نشان داده می‌شوند.به طور مشابه، *Min Pooling* در زمانی که پس‌زمینه‌ی تصویر، روشن است، استفاده می‌شود زیرا پیکسل‌های تیره‌تر را انتخاب می‌کند.

در نوع سوم یعنی *Average Pooling* خلاصه ویژگی‌های یک منطقه با مقدار متوسط آن منطقه نشان داده می‌شود. علاوه بر کاهش ابعاد تصویر، لبه‌های تیز یک عکس را صاف می‌کند و زمانی استفاده می‌شود که چنین لبه‌هایی مهم نباشند. از این رو ممکن است هنگام استفاده از این روش، ویژگی‌های واضح شناسایی نشود. ویژگی‌ها را راحت‌تر از *Max Pooling* استخراج می‌کند، در حالی که*Max Pooling* ویژگی‌های برجسته‌تری مانند لبه‌ها را استخراج می‌کند. (لینک کمکی)

در *Global Pooling* هر کانال در نقشه‌ی ویژگی فقط به یک مقدار کاهش می‌یابد. مقدار به نوع *Global Pooling* بستگی دارد که می‌تواند هر یک از انواع توضیح داده‌شده قبلی باشد و تقریباً مانند اعمال فیلتری از ابعاد دقیق نقشه‌ی ویژگی است.

Global Average Pooling اغلب برای جایگزینی لایه‌های *Flatten* در یک طبقه‌بندی استفاده می‌شوند. در عوض، مدل با یک لایه کانولوشن خاتمه می‌یابد که به تعداد کلاس‌های هدف نقشه‌های ویژگی تولید می‌کند و برای هر یک از آن‌ها *Global Average Pooling* را اعمال می‌کند تا هر نقشه ویژگی را به یک مقدار تبدیل کند. از آنجایی که نقشه‌های ویژگی می‌توانند عناصر خاصی را در داده‌های ورودی تشخیص دهند، نقشه‌های لایه نهایی به طور موثری یاد می‌گیرند که حضور یک کلاس خاص را در ورودی تشخیص دهند. به طور خلاصه همان کار لایه‌ی *Flatten* را انجام می‌دهد با پارامترهای بسیار کمتر اما اطلاعات مکان را از دست می‌دهد. یعنی امتیاز وجود یک کلاس در تصویر را معین می‌کند اما این که دقیقا کجای تصویر است را خیر. علاوه بر این، این رویکرد ممکن است عملکرد مدل را بهبود دهد و به دلیل این واقعیت که هیچ پارامتری برای آن وجود ندارد *overfitting* را کاهش دهد. (لینک کمکی) (۵۰ دقیقه)

ث) شبکه‌ی ***VGG*** برای کاهش تعداد پارامترها در لایه‌های *CONV* و بهبود زمان *train* به وجود آمد. انواع مختلفی از ***VGGNet*** (***VGG16***، ***VGG19*** و غیره) وجود دارد که تنها در تعداد کل لایه‌های شبکه متفاوت است. *VGG16* در مجموع ۱۳۸ میلیون پارامتر دارد که نسبت به *AlexNet* بسیار کاهش یافته است. ایده‌ای که منجر به بهبود شد این است که تمامی کرنل های *conv* در اندازه 3×3 و کرنل‌های *max pooling* در اندازه‌ی 2×2 با گام دو هستند.

با جایگزین کردن فیلترهای بزرگ با اندازه هسته (به ترتیب ۱۱ و ۵ در لایه‌ی اول و دوم) با چندین فیلتر به اندازه هسته 3×3 بهبود را نسبت به *AlexNet* انجام می‌دهد. با یک *receptive field*

معین (تعداد پیکسل‌های موثر تصویر ورودی که خروجی به آن‌ها بستگی دارد)، فیلترهایی با اندازه کوچک‌تر، بهتر از فیلترهایی با اندازه‌ی بزرگ‌تر است، زیرا چندین لایه‌ی غیرخطی عمق شبکه را افزایش می‌دهد که آن را قادر می‌سازد ویژگی های پیچیده تر را با هزینه کمتر یاد بگیرد.

(لینک کمکی)

ResNet مشابه *GoogLeNet*، از یک *Global Average Pooling* و به دنبال آن لایه طبقه‌بندی استفاده می‌کند. از طریق تغییرات ذکر شده، *ResNet* با عمق شبکه به بزرگی ۱۵۲ آموخته شد. دقت بهتری نسبت به *VGGNet* و *GoogLeNet* دارد در حالی که از نظر محاسباتی کارآمدتر از *VGGNet* است. *ResNet-152* به ۹۵.۵۱ دقت *top 5* دست می‌یابد.

معماری شبیه به *VGGNet* است که بیشتر از فیلترهای 3×3 تشکیل شده است. از *VGGNet*، اتصال میانبر برای تشکیل یک شبکه باقی‌مانده وارد می‌شود.

هنگامی که تعداد لایه‌های شبکه را زیاد می‌کنیم، ممکن است محوشدگی گرادیان رخ دهد و گرادیان به لایه‌های اول نرسد. این مورد باعث می‌شود که شبکه حداقل به خوبی مدل‌های کم‌عمق‌تر نباشد، ایده‌ی این شبکه این است که وزن‌های لایه‌های مدل‌های کم‌عمق را به لایه‌های نخست شبکه کپی کنیم و لایه‌های باقیمانده را جوری تنظیم کنیم که اگر گرادیان نرسید حداقل نگاشت همانی باشند که گرادیان صفر نشود. لذا یک اتصال از ورودی به خروجی ایجاد کرده و باهم جمع می‌کند تا اگر شبکه نتوانست چیز خوبی را یاد بگیرد، لااقل ورودی را در خروجی داشته باشیم. این شبکه از تعدادی بلوک باقیمانده تشکیل شده و هر کدام دارای دو لایه‌ی کانولوشنی 3×3 هستند و به طور دوره‌ای تعداد فیلترها دو برابر شده و رزولوشن مکانی نصف می‌شود.

VGG-16 تقریباً ۱۳۸ میلیون پارامتر دارد و *ResNet* دارای ۲۵.۵ میلیون پارامتر و به همین دلیل سریع‌تر است و حافظه‌ی کمتری مصرف می‌کند. همچنین در دو لایه‌ی اول *ResNet* موفق می‌شود ارتفاع و عرض تصویر را تا ۴ برابر کاهش دهد. از *VGG19* متوجه می‌شویم که دو لایه اول پیچیدگی را در تصویر کامل اعمال می‌کنند که بسیار پرهزینه است. اگر محاسبه کنیم متوجه خواهیم شد که لایه اول *170M FLOP* انجام می دهد، اما خروجی $64 \times 224 \times 224$ را از تصویر $3 \times 224 \times 224$ تولید می‌کند. این لایه به تنهایی به اندازه‌ی کل ***ResNet-34*** *FLOP* دارد.

(لینک کمکی) (۶۰ دقیقه)

۲. مدل *fully-connected* خود را به صورت زیر تعریف می‌کنیم که حدودا ۴۰۴ هزار پارامتر دارد.

```
# Fully connected model
fc_model = keras.Sequential()
fc_model.add(keras.layers.Input(shape=x_train[0].shape))
# Write your code here
# Add Flatten layer and few Dense layers
fc_model.add(Flatten())
fc_model.add(Dense(128, activation='relu'))
fc_model.add(Dense(64, activation='relu'))
fc_model.add(Dense(32, activation='relu'))
fc_model.add(Dense(10, activation='softmax'))
fc_model.summary()
```

```
Total params: 404,010
Trainable params: 404,010
Non-trainable params: 0
```

همچنین مدل کانولوشنی خود را به صورت زیر تعریف می‌کنیم که تقریباً نصف مدل قبلی یعنی حدودا ۱۹۰ هزار پارامتر دارد.

```
Conv_model = Sequential()
Conv_model.add(keras.layers.Input(shape=x_train[0].shape))
# write your code here
# add few Conv layers and Flatten layer
# you can use pool layers after Conv
Conv_model.add(Conv2D(32, (3,3), activation='relu'))
Conv_model.add(MaxPooling2D((2, 2)))
Conv_model.add(Conv2D(128, (3,3), activation='relu'))
Conv_model.add(Conv2D(64, (3,3), activation='relu'))
Conv_model.add(Flatten())
Conv_model.add(Dense(10, activation='softmax'))
Conv_model.summary()
```

```
Total params: 189,130
Trainable params: 189,130
Non-trainable params: 0
```

الف)

میزان خطا و دقت روی مدل کاملا متصل:

```
Loss and Accuracy on Test set :
313/313 [==============================] - 1s 3ms/step - loss: 1.5680 - accuracy: 0.4409
```

میزان خطا و دقت روی مدل کانولوشنی:

```
Loss and Accuracy on Test set :
313/313 [==============================] - 1s 3ms/step - loss: 0.9947 - accuracy: 0.6625
```

هیچ رابطه‌ای بین دو معیار *loss* و *accuracy* وجود ندارد.

ضرر را می‌توان به عنوان فاصله بین مقادیر واقعی مسئله و مقادیر پیش‌بینی‌شده توسط مدل در نظر گرفت. *loss* هر چه بیشتر باشد، *error* هایی که در داده‌ها انجام شده، بزرگ‌تر است. دقت را می‌توان به عنوان تعداد خطاهایی که در داده‌ها انجام داده‌ایم، در نظر بگیریم. دقت کم و ضرر زیاد یعنی در بسیاری از داده ها خطاهای بزرگی انجام داده‌ایم. دقت کم و ضرر کم یعنی خطاهای کمی در بسیاری از داده‌ها داشته‌ایم. دقت عالی با ضرر کم یعنی در تعداد کمی از داده‌ها خطای کمی داشته‌ایم. (بهترین حالت) (لینک کمکی)

ب) مدت زمان هر ایپاک برای مدل اول: ۵ ثانیه

```
Fully Connected Model
Epoch 1/5
1563/1563 [==============================] - 5s 3ms/step - loss: 1.6095 - accuracy: 0.4221
Epoch 2/5
1563/1563 [==============================] - 5s 3ms/step - loss: 1.5928 - accuracy: 0.4263
Epoch 3/5
1563/1563 [==============================] - 5s 3ms/step - loss: 1.5791 - accuracy: 0.4324
Epoch 4/5
1563/1563 [==============================] - 5s 3ms/step - loss: 1.5655 - accuracy: 0.4386
Epoch 5/5
1563/1563 [==============================] - 5s 3ms/step - loss: 1.5573 - accuracy: 0.4401
```

مدت زمان هر ایپاک برای مدل دوم: ۷ ثانیه

```
Convolutional Model
Epoch 1/5
1563/1563 [==============================] - 7s 4ms/step - loss: 1.4966 - accuracy: 0.4602
Epoch 2/5
1563/1563 [==============================] - 6s 4ms/step - loss: 1.1616 - accuracy: 0.5896
Epoch 3/5
1563/1563 [==============================] - 7s 5ms/step - loss: 1.0167 - accuracy: 0.6445
Epoch 4/5
1563/1563 [==============================] - 7s 4ms/step - loss: 0.9214 - accuracy: 0.6773
Epoch 5/5
1563/1563 [==============================] - 7s 5ms/step - loss: 0.8395 - accuracy: 0.7071
```

پ) نه لزوما، زیرا در این سوال تعداد پارامترهای مدل کانولوشنی نصف مدل کاملامتصل بود اما هر ایپاک آن زمان بیشتری طول کشید، لذا ارتباط مستقیمی وجود ندارد و بیشتر به نوع معماری شبکه وابسته است.

(۱.۵ ساعت)

۳. الف) برای تکمیل این بخش از تابع آماده‌ی *copyMakeBorder()* استفاده کردیم، به این صورت که پارامتر ورودی اول خود تصویر، چهار پارامتر بعدی به ترتیب فاصله از بالا، پایین، چپ و راست برای اضافه کردن *padding* و پارامترهای بعدی نوع *padding* است که ما اینجا از BORDER_CONSTANT و رنگ مشکی که همان معادل *zero-padding* است، استفاده کردیم.

(لینک کمکی)

```
def resize_img(img, desired_size = 224):
    # write your code here
    old_size = img.shape[:2]
    ratio = float(desired_size)/max(old_size)
    new_size = tuple([int(x*ratio) for x in old_size])
    image = cv2.resize(img, (new_size[1], new_size[0]))
    delta_w = desired_size - new_size[1]
    delta_h = desired_size - new_size[0]
    top, bottom = delta_h//2, delta_h-(delta_h//2)
    left, right = delta_w//2, delta_w-(delta_w//2)
    color = [0, 0, 0]
    new_im = cv2.copyMakeBorder(image, top, bottom, left, right, cv2.BORDER_CONSTANT, value=color)
    return new_im
```

ب) برای این بخش، شبکه‌ی *ResNet* را به عنوان بیس مدل لود کرده و تنها لایه‌ی آخر آن را فریز کردیم، زیرا تعداد کلاس‌های ما متفاوت است. از وزن‌های رندوم استفاده کردیم و در انتها یک لایه‌ی *Flatten* و *Dense* با تعداد نورون ۲۴ (تعداد کلاس‌ها) و تابع فعالسازی *softmax* استفاده کردیم.

```
resnet = tf.keras.models.Sequential()
# Write your code here
base_model = tf.keras.applications.ResNet50(weights= None, include_top=False, input_shape= (224, 224, 3))
resnet.add(base_model)
resnet.add(Flatten())
resnet.add(Dense(24, activation='softmax'))
resnet.summary()
```

دقت روی داده‌های آموزشی به صورت زیر شد:

```
Epoch 1/3
65/65 [==============================] - 69s 987ms/step - loss: 15.6880 - acc: 0.2032
Epoch 2/3
65/65 [==============================] - 65s 999ms/step - loss: 3.2720 - acc: 0.5839
Epoch 3/3
65/65 [==============================] - 65s 992ms/step - loss: 2.2978 - acc: 0.8848
```

سپس شبکه‌ی *ResNet* را دوباره لود کرده و وزن‌های شبکه را فریز می‌کنیم. لایه‌های دلخواه خود را نیز اضافه می‌کنیم، اما این بار از وزن‌های یادگرفته شده از *imagenet* استفاده می‌کنیم. در اینجا ما سه لایه‌ی *Dence* و یک لایه‌ی *Flatten* اضافه کردیم.

```
fine_tune_resnet = tf.keras.models.Sequential()
# write your code here
baseModel = ResNet50(weights="imagenet", include_top=False, input_shape=(224, 224, 3))
fine_tune_resnet.add(baseModel)
baseModel.trainable = False
fine_tune_resnet.add(Flatten())
fine_tune_resnet.add(Dense(256, activation="relu"))
fine_tune_resnet.add(Dense(128, activation='relu'))
fine_tune_resnet.add(Dense(24, activation="softmax"))
fine_tune_resnet.summary()
```

دقت روی داده‌های آموزشی به صورت زیر شد:

```
Epoch 1/3
65/65 [==============================] - 98s 552ms/step - loss: 3.3244 - acc: 0.3550
Epoch 2/3
65/65 [==============================] - 34s 527ms/step - loss: 0.7859 - acc: 0.8013
Epoch 3/3
65/65 [==============================] - 34s 527ms/step - loss: 0.4130 - acc: 0.8918
```

در انتها هر دو مدل را تست کردیم و نتایج به شرح زیر است:

```
resnet.evaluate(test_generator)
```

```
33/33 [==============================] - 18s 508ms/step - loss: 8.2649 - acc: 0.1026
[8.264884948730469, 0.10257234424352646]
```

```
fine_tune_resnet.evaluate(test_generator)
```

```
33/33 [==============================] - 18s 515ms/step - loss: 0.3040 - acc: 0.9322
[0.30396291613578796, 0.9321543574333191]
```

همانطور که واضح است مدل دوم بسیار خوب عمل کرده و دقت ۹۳٪ دارد در حالی که مدل اول بسیار بد عمل کرده و فقط ۱۰٪ دقت داشته و همچنین *overfit* شده است. دلیل افزایش دقت اضافه کردن لایه‌های مناسب و استفاده از وزن‌های شبکه‌های از پیش آموخته‌شده و به طور کلی انجام عملیات تنظیم دقیق می‌باشد.

(لینک کمکی) (۲.۵ ساعت)