به نام خدا

تمرین سری ششم

درس مبانی بینایی کامپیوتر

سینا علی‌نژاد

شماره دانشجویی: ۹۹۵۲۱۴۶۹

سوال ۱-

الف)

این دو مفهوم در واقع به ماهیت ضرب زنجیره‌ای یا chain rule در استفاده از گرادیان برای آپدیت وزنها در آخر هر epoch برمی‌گردد. از آنجا که در backpropagation انتشار از لایه‌ی آخر به لایه های ابتدایی صورت میگیرد، اگر مقدار مشتق‌ها کوچک شود، ضرب این مقادیر کوچک در chain rule باعث کاهش مقدار نهایی به صورت exponential می‌شود و این باعث می‌شود یادگیری در لایه‌های اولیه و دورتر از لایه‌های آخر، بسیار کم شود و اگر در بدترین حالت در مسیر انتشار، مشتقی صفر شود، یادگیری در این لایه‌ها متوقف می‌شود. به این پدیده در شبکه‌های عصبی vanishing gradient problem گفته می‌شود.

در سمت دیگر، ممکن است مقادیر مشتق‌ها در chain rule اعداد بزرگی باشند و ضرب این مقادیر باعث افزایش مقدار نهایی به صورت exponential برای لایه‌های دورتر از لایه‌ی آخر میشود. اگر این مقدار خیلی بزرگ شود، overflow رخ داده و مقدار وزن‌ها در آن لایه‌ها nan (not a number) شده و در این ادامه یادگیری در مراحل بعد را کامل خراب می‌کند. به این پدیده در شبکه‌های عصبی exploding gradient problem گفته می‌شود. این مشکل باعث ناپایداری شبکه می‌شود زیرا وزن‌ها به صورت ناگهانی زیاد تغییر می‌کنند و این باعث میشود در نمودار گرادیان نتوانیم به صورت تدریجی به نقطه خوب و بهینه برسیم زیرا جابجایی در حدی است که کلا به سمت دیگری میرود که بی‌ربط است.

چند راه برای فهمیدن اینکه احتمالا مدل ما دچار exploding gradient problem شده:

۱- مقادیر loss در هر مرحله تغییرات زیاد و بزرگ داشته باشد.
۲- مقادیر loss مقدار nan داشته باشند.
۳- مدل ما روند یادگیری بسیار پایینی حتی روی داده‌های آموزشی داشته باشد.

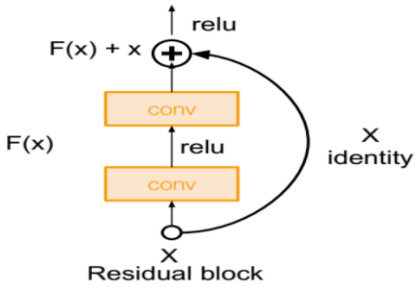
چند راه برای فهمیدن اینکه احتمالا مدل ما دچار vanishing gradient problem شده:

۱- پیشرفت مدل حتی روی داده آموزشی بسیار کند باشد.
۲- مقادیر وزن‌ها در لایه‌های آخر تغییرات نسبتا بزرگتری نسبت به لایه‌های ابتدایی داشته باشند.
۳- وزنهای مدل اعداد بسیار کوچکی شوند.
۴- حتی برخی وزنهای مدل صفر شوند.

ب)

مدل‌های قبل از ResNet برای افزایش تعداد لایه‌ها دچار مشکل vanishing gradient می‌شدند و در نتیجه صحت داده‌های هم آموزش و هم تست پایین میامد زیرا یادگیری بسیار کند بود و اون حجم از داده برای این تعداد لایه کافی نبود، در واقع مشکل overfitting نبود بلکه مدل ما حتی روی داده آموزش هم دقت پایینی بدست می‌آورد و این بخاطر مشکلی بود که در بهینه‌سازی آن وجود داشت. اما ResNet با ۱۵۲ لایه در شبکه خود، انقلابی در تعداد لایه‌ها ایجاد کرد.

ایده‌ی ResNet بسیار ساده بود، به طور خلاصه به هر لایه می‌گفت: "اگر نتوانستی چیزی یاد بگیری،حداقل از همان آموخته‌های لایه‌ی قبل استفاده کن"، و مفهومی به اسم residual block را ایجاد کرد و پس از هر چند لایه کانولوشنی، یک لینک مستقیم از لایه‌ی قبل از لایه‌های کانولوشنی به بعد آنها قرار داد.

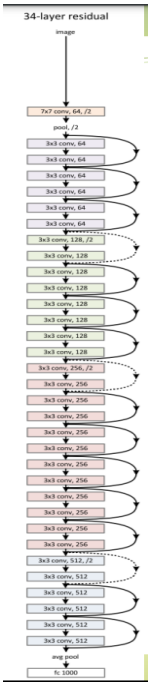


شکل بالا نمونه‌ای از یک residual block را نشان میدهد، در واقع مدل ما به جای اینکه H(x) یا نگاشت مطلوب را در انتهای بلوک یاد بگیرد، باقیمانده‌ی آن را یاد میگیرد، به همین دلیل به آن residual یا باقیمانده گفته می‌شود.

مدل ما $F(x) = H(x) - x$ را یاد می‌گیرد.

در واقع این کار باعث میشود مدلهای عمیقتر حداقل به خوبی مدلهای کم‌عمقتر بتوانند عمل کنند. این لینکی که مستقیما به آخر بلوک وصل میشود، تاثیر خود را در chain rule ایجاد میکند، زیرا اگر در مسیرهای دیگر مشتق کم شود یا صفر شود، در این مسیر مطمئن هستیم که این اتفاق نمی‌افتد و از آنجا که این مقادیر chain rule ها در مسیرهای مختلف با هم جمع می‌شوند، مقدار نهایی نیز هرگز صفر نخواهد شد.

شکل زیر یک شبکه با معماری ResNet را نشان می‌دهد:



ابتدا یک لایه کانولوشن ۷ در ۷ و سپس pooling و بعد از آن تعداد زیادی بلوک باقیمانده که هر بلوک شامل دو لایه کانولوشنی ۳ در ۳ و بعضیشان شامل یک pooling نیز هستند، تعداد فیلترها هم به مرور بیشتر و بیشتر می‌شود و در انتها یک global average pooling

برای کاهش مکان و در نتیجه کاهش تعداد پارامترهای لایه آخر که یک لایه کاملا متصل است، زده می‌شود. این مدل روی ImageNet اعمال شده پس ۱۰۰۰ نورون برای لایه کاملا متصل در نظر گرفته.

سوال۲-

الف)

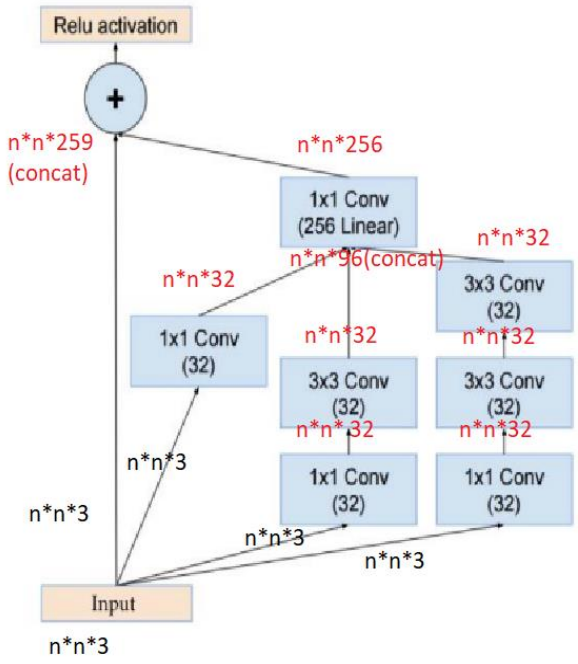
محاسبه تعداد پارامترهای قابل آموزش: در inception module که در GoogleNet ایده‌اش مطرح شد، padding ضروری است زیرا حاصل کانولوشن‌ها در مسیرهای مختلف با هم concat می‌شوند و برای این کار نیاز است دو بعد اول آنها اندازه‌های یکسانی داشته باشند. طبق این موضوع تعداد پارامترها را محاسبه میکنیم:

32(1*1*3(depth) + 1(for bias)) + 32(1*1*3(depth) + 1(for bias)) + 32(1*1*3(depth) + 1(for bias)) +

32(3*3*32(depth) + 1(for bias)) + 32(3*3*32(depth) + 1(for bias)) + 32(3*3*32(depth) + 1(for bias)) +

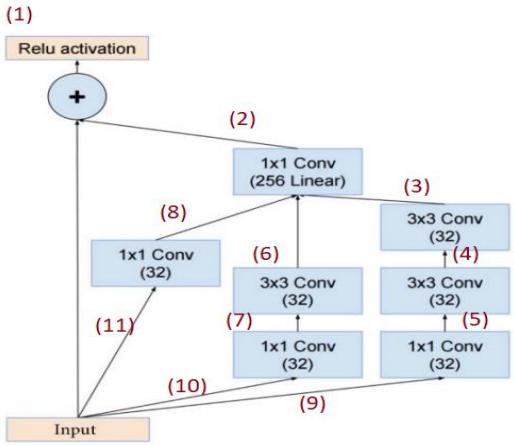
256(1 * 1 * 96(depth after concating) + 1(for bias)) = 52960

در تصویر زیر برای هر لینک، سایز ورودی یا خروجی را نوشتم و محاسبات تعداد پارامترها را طبق آن انجام دادم.

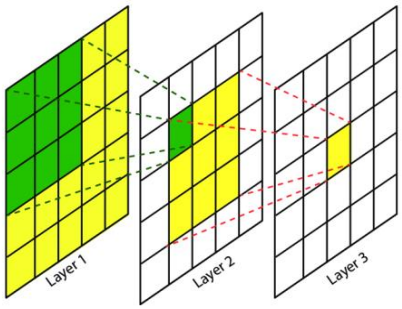


برای محاسبه میدان تاثیر از بالاترین نورون ها شروع میکنیم، طبق شکل زیر از شماره (1) آغاز میکنیم، خروجی در این قسمت برابر است با چسباندن(concat) دو ورودی، پس بعضی از پیکسلها مستقیما از Input می‌آیند و برخی از مسیر دیگر. آنهایی که از Input می‌آیند، میدان تاثیر ۱ دارند، زیرا هر پیکسل مستقیما از همان ورودی اولیه می‌آید و مربوط به یک بخش از ورودی نمیشود. اما آنهایی که از مسیر دیگر می‌آیند را باید بررسی کنیم. در قسمت (2) هر پیکسل از کانولوشن ۱*۱ آمده است پس در قسمتهای (6),(3) و (8) به یک بخش ۳۲*۱*۱

وابسته بوده است. از اینجا به بعد، سه مسیر متفاوت داریم و برای هر کدام جداگانه میدان تاثیر را در هر قسمت شماره گذاری شده محاسبه میکنیم. در قسمت (3) هر پیکسل به یک محدوده ۳*۳ در قسمت (4) وابسته است، پس تا این قسمت، میدان تاثیر برابر است با ۳۲*۳*۳ .



حال هر پیکسل در قسمت (4) دوباره به یک محدوده‌ی ۳*۳ در قسمت (5) وابسته است و حالتی مثل عکس زیر در اسلایدها داریم:



که نشان میدهد، چگونه دو کانولوشن ۳*۳ متوالی، میدان تاثیری برابر با یک محدوده ۵*۵ ایجاد می‌کند. پس طبق همین، تا قسمت (5) میدان تاثیری برابر با ۳۲*۵*۵ داریم. در نهایت در قسمت (9) میدان تاثیر برابر است با ۳*۵*۵ ، یعنی پیکسل هایی که از مسیر 9-5-4-3-2 به خروجی رفته‌اند، در ورودی اولیه، دارای میدان تاثیر ۵*۵*۳ هستند که ۳ تعداد کانالها است و اگر میدان تاثیر را صرفا از لحاظ مکانی در نظر بگیریم، برابر است با ۵*۵ و کاری به تعداد کانال یا عمق نخواهیم داشت.

با استدلال مشابه برای باقی قسمت‌ها داریم:

قسمت (7) : ۳*۳

قسمت (10) : ۳*۳

قسمت (11) : ۱*۱

پس اگر میدان تاثیر را برابر با ماکزیمم این مقادیر بگیریم، به طور ماکزیمم، یک پیکسل در خروجی به یک محدوده‌ی ۵*۵ در ورودی وابسته است.

ب) ابتدا تعداد پارامتر را بررسی میکنیم:

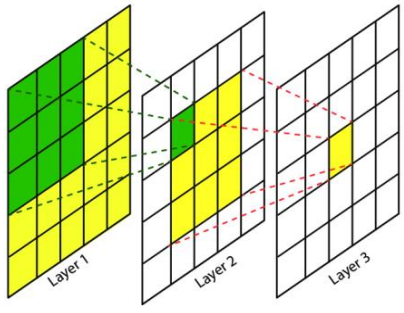
برای حالت A هر فیلتر ۳ * ۳ است اما اگر بخواهیم دقیقتر بگوییم، هر فیلتر ۳*۳*۳ است زیرا عکس ورودی ۳ کاناله است پس تعداد خانه‌های این فیلتر و در نتیجه تعداد وزنهای مربوط به هر فیلتر برابر است با ۳*۳*۳ = ۲۷ و البته یکی هم برای Bias پس میشود ۲۸ تا وزن. تعداد فیلترها برابر است با ۱۶ تا پس برای لایه اول تعداد پارامترها برابر است با: ۱۶*۲۸ = ۴۴۸ ، خروجی این لایه از آنجا که پدینگ نداریم یا همان پدینگ ولید داریم، num_of_filters * (n-2) * (n-2) خواهد بود که num_of_filters تعداد فیلترها در لایه اول است. پس تعداد وزنها برای هر فیلتر برای لایه دوم برابر است با ۱۶*۳*۳ که برابر است با و یکی هم برای bias که میشود ۱۴۵ تا. تعداد فیلترها در این لایه برابربا ۳۲ است پس کلا در این لایه به تعداد ۱۴۵*۳۲ که برابراست با ۴۶۴۰ پارامتر داریم. مجموع پارامترهای دو لایه میشود برابر با : ۵۰۸۸

برای حالت B دیگر وزنها را به اشتراک نمیگذاریم، برای مثال اگر تصویر را به پنجره‌های ۳*۳ تقسیم کنیم و حالت padding = valid را در نظر بگیریم، به تعداد (n-2)*(n-2) از این پنجره‌ها خواهیم داشت و برای هر پنجره از یک فیلتر جدا استفاده میکنیم و تعداد پارامترهای هر فیلتر برابراست با ۳*۳*۳ که سومین ۳ برای تعداد کانالها است و یکی هم برای bias در نظر میگیریم پس تعداد کل پارامترها برابر است با ۲۸ پس برای کل پنجره‌ها تعداد پارامتر برابراست با 28*(n-2)*(n-2) ، تعداد کل unitها برابر است با ۱۶ پس برای لایه اول تعداد پارامترها برابر است با ۱۶*۲۸*(n-2)*(n-2) ، برای لایه‌ی دوم هم به همین شکل عمل کرده و برای هر فیلتر تعداد پارامترها برابر است با : ۳*۳*۱۶ که ۱۶ تعداد ویژگی‌هایی است که از لایه قبل آمده است و یکی هم برای bias در نظر می‌گیریم که میشود ۱۴۵ تا پارامتر، همچنین تعداد پنجره‌ها برای این لایه برابر است با (n-4)*(n-4) به دلیل padding=valid پس تعداد کل پارامترها برای این لایه برابر است با: (n-4)*(n-4)*145*32 . تعداد کل پارامترها برای حالت B برابر است با مجموع تعداد پارامترها برای این دو لایه.

بررسی مقدار میدان تاثیر:

برای حالت A دو لایه داریم که تو هردوشون فیلترهای ۳*۳ داریم، پس پیکسلهای خروجی لایه آخر میدان تاثیری برابر با یک محدوده‌ی ۵*۵ در ورودی اولیه دارند. (مطابق تصویر زیر) البته بخواهیم دقیقتر بگوییم ۵*۵*۳ چون مقادیر هر سه کانال موثر هستند و در واقع فیلترهای لایه اول ما ابعاد ۳*۳*۳ دارند.

منظور از میدان تاثیر هم یعنی اینکه مقادیر نهایی که در خروجی می‌آیند، از چه محدوده‌ای از مقادیر ورودی اولیه تاثیر گرفته‌اند.



در حالت B هم از این لحاظ فرقی ندارد، باز هم فیلترها ۳*۳ هستند و در دو لایه این اتفاق افتاده است پس مقدار میدان تاثیر ۵*۵ است.

سوال۳-

الف) برای این بخش ابتدا یک پیش پردازش انجام دادم تا از آن در همه قسمت های بعدی سوال استفاده کنم.

```
num_classes = 10
y_train = keras.utils.to_categorical(y_train, num_classes)
y_val = keras.utils.to_categorical(y_val, num_classes)
x_train = (x_train).astype(np.float32) / 255
x_val = (x_val).astype(np.float32) / 255
```

و شبکه رو با لایه های زیر و به صورت functional تعریفش کردم:

```
inputs = keras.Input(shape=(x_train[0].shape))
layer1 = Conv2D(128, kernel_size=(3,3), activation='relu')(inputs)
layer2 = Conv2D(128, kernel_size=(3,3), activation='relu')(layer1)
layer3 = Conv2D(128, kernel_size=(3,3), activation='relu')(layer2)
layer4 = Conv2D(128, kernel_size=(3,3), activation='relu')(layer3)
layer5 = MaxPool2D()(layer4)
layer6 = Flatten()(layer5)
outputs = Dense(num_classes, activation='softmax')(layer6)
model = keras.Model(inputs=inputs, outputs=outputs)
```

چهار لایه کانولوشنی، یکی لایه max pooling و در نهایت لایه flatten و کاملا متصل برای کلاس بندی. برای تابع فعالسازی از relu برای لایه های hidden و از softmax برای لایه خروجی نهایی برای کلاس بندی استفاده کردم.

برای تابع ضرر از cross-entropy و برای بهینه ساز از adam استفاده کردم.

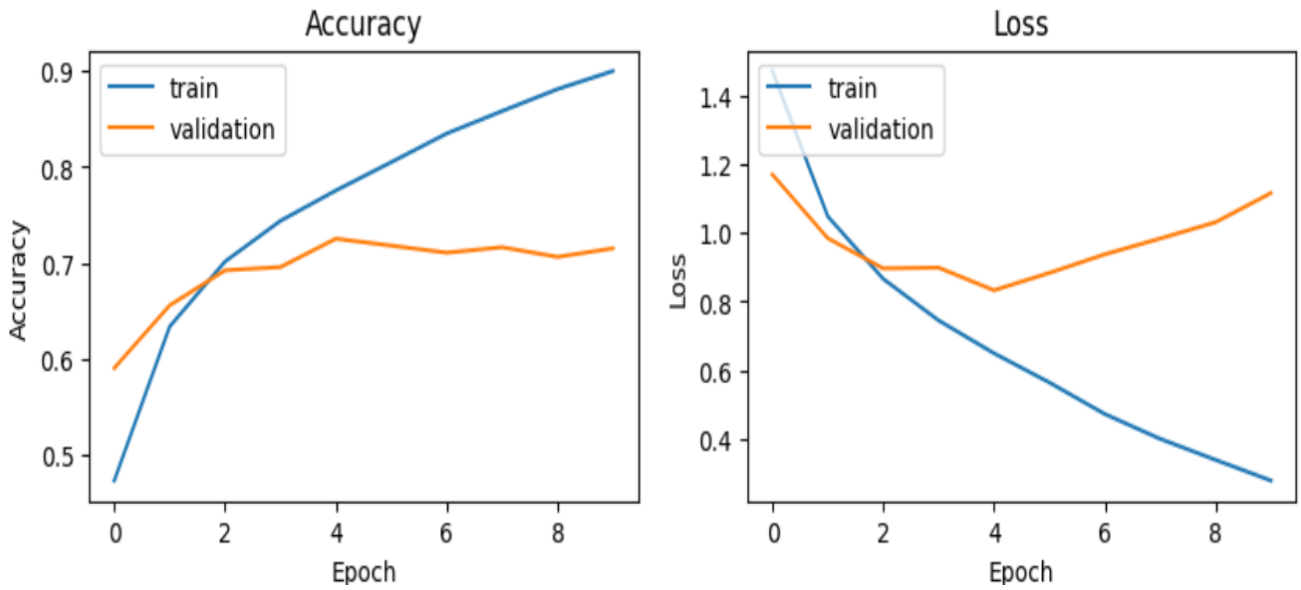
```
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

در نهایت به دقت زیر رسیدم:

```
history = model.fit(x_train, y_train, batch_size=100, epochs=10, validation_data=(x_val,y_val), shuffle=True)
################################################
```

```
Epoch 1/10
500/500 [==============================] - 19s 34ms/step - loss: 1.4735 - accuracy: 0.4727 - val_loss: 1.1714 - val_accuracy: 0.5902
Epoch 2/10
500/500 [==============================] - 17s 34ms/step - loss: 1.0492 - accuracy: 0.6335 - val_loss: 0.9852 - val_accuracy: 0.6556
Epoch 3/10
500/500 [==============================] - 17s 35ms/step - loss: 0.8667 - accuracy: 0.7014 - val_loss: 0.8972 - val_accuracy: 0.6922
Epoch 4/10
500/500 [==============================] - 17s 34ms/step - loss: 0.7453 - accuracy: 0.7439 - val_loss: 0.8995 - val_accuracy: 0.6956
Epoch 5/10
500/500 [==============================] - 17s 34ms/step - loss: 0.6494 - accuracy: 0.7755 - val_loss: 0.8332 - val_accuracy: 0.7252
Epoch 6/10
500/500 [==============================] - 17s 34ms/step - loss: 0.5640 - accuracy: 0.8051 - val_loss: 0.8841 - val_accuracy: 0.7180
Epoch 7/10
500/500 [==============================] - 17s 34ms/step - loss: 0.4720 - accuracy: 0.8350 - val_loss: 0.9383 - val_accuracy: 0.7106
Epoch 8/10
500/500 [==============================] - 17s 34ms/step - loss: 0.3994 - accuracy: 0.8584 - val_loss: 0.9844 - val_accuracy: 0.7163
Epoch 9/10
500/500 [==============================] - 17s 34ms/step - loss: 0.3380 - accuracy: 0.8813 - val_loss: 1.0322 - val_accuracy: 0.7062
Epoch 10/10
500/500 [==============================] - 17s 34ms/step - loss: 0.2776 - accuracy: 0.9002 - val_loss: 1.1173 - val_accuracy: 0.7151
```

نمودار accuracy و loss به صورت زیر شد:

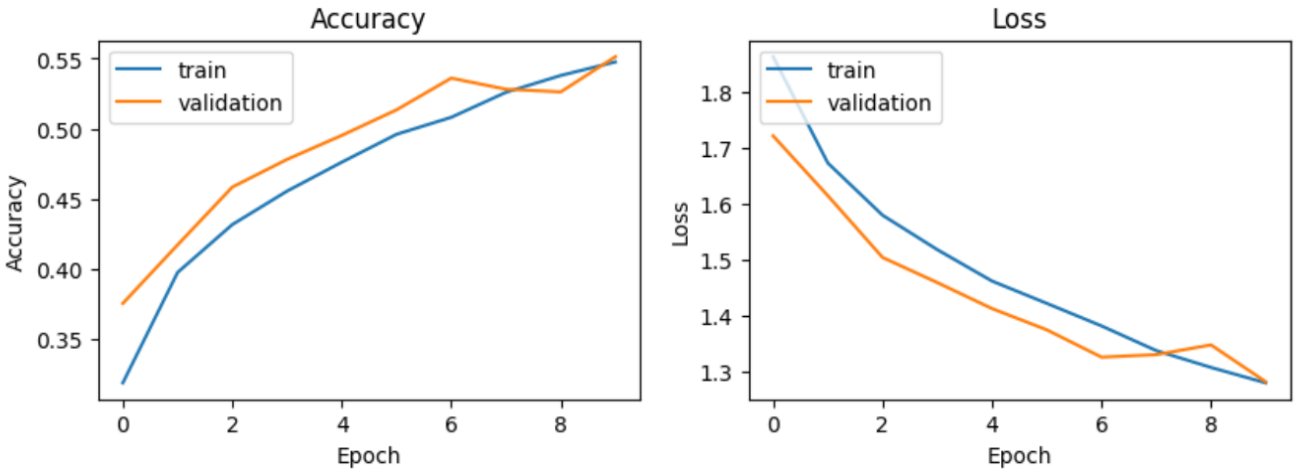


همانطور که از نمودارها مشخص است، مدل ما دچار overfitting شده است، زیرا نمودار loss مربوط به داده آموزش بهتر و بهتر میشود ولی برای validation از یک جایی به بعد در حال بدتر شدن است. همچنین اختلاف accuracy برای داده آموزش و تست در آخر زیاد است.

ب) گاهی اوقات، پیچیدگی مدل باعث overfitting میشود، یکی از راههایی که به کمک آن میتوان مسئله را برای مدل سختتر کرد، تنوع داده است. یعنی به جای اینکه مدل خود را ساده کنیم، مدل را پیچیده نگه داشته ولی مسئله را سختتر میکنیم، data augmentation و dropout دو روش برای سختتر کردن مسئله می‌باشند. برای data augmentation میتوان از کلاس imageDataGenerator در keras استفاده کرد که به ما امکانات متنوعی برای ایجاد تنوع در داده‌ی تصویری می‌دهد.

```
datagen = ImageDataGenerator(rotation_range=90, horizontal_flip=True, vertical_flip=True)
batch_size = 100
history = model.fit(datagen.flow(x_train, y_train, batch_size=batch_size),
                    epochs=10, # one forward/backward pass of training data
                    steps_per_epoch=x_train.shape[0]//batch_size, # number of images comprising of one epoch
                    validation_data=(x_val, y_val), # data for validation
                    validation_steps=x_val.shape[0]//batch_size)
```

در کد بالا یک شی از کلاس ImageDataGenerator ساختم و چندتا از کارهایی که هنگام دادن عکس آموزشی به مدل میتواند انجام دهد را به آن دادم. برای مثال rotation_range = 90 یعنی اینکه هنگام دادن عکس میتوانی تا محدوده‌ی ۹۰ درجه عکس را چرخانده و حاصل را به مدل برای آموزش بدهی. طرز کار آن بدین صورت است که به صورت رندوم عمل میکند، مثلا یک تاس می‌اندازد و تصمیم میگیرد که این تصویر را بچرخاند، یا انتقال دهد یا نورش را کم کند یا ... یا خود عکس را بدون تغییر بدهد. پارامتر horizontal_flip و vertical_flip هم همانطور که از نامشان پیداست، امکان آینه شدن عکس را فراهم میکنند. برای شبکه، از همان شبکه قسمت الف استفاده کردم و فقط data augmentation را اضافه کردم و خروجی به صورت زیر بود:



تحلیل نمودار بالا: وقتی از روش افزودن داده استفاده کردیم، دقت مدل روی داده آموزش پایین آمد و در آخرین epoch به حدود ۵۷ درصد رسید زیرا با افزودن داده، داریم کار را برای مدل سختتر میکنیم و مجبور است الگوهای پیچیده‌تری یاد بگیرد و این باعث پایین آمدن دقت هم روی داده آموزش و هم روی داده تست میشود. تغییر دیگری که نسبت به حالت الف دارد، این است که دیگر overfitting نداریم و میبینیم که نمودار loss برای داده تست و آموزشی با یکدیگر کاهش پیدا میکنند، همچنین accuracy نهایی برای این داده تست و آموزش اختلاف زیادی ندارند.

د) حال میخواهیم از روش انتقال یادگیری استفاده کنیم. در این روش از وزنهایی که یک مدل موفق روی یک مسئله دیگر بدست آورده است، به عنوان وزن اولیه برای مدل خود استفاده میکنیم.

```
x_train = keras.applications.resnet50.preprocess_input(x_train)
x_val = keras.applications.resnet50.preprocess_input(x_val)
```

ابتدا پیش پردازشی که resnet50 دارد، را روی داده خود انجام می‌دهیم.(کد بالا)

حال یک base_model از ResNet50 تعریف میکنیم:

```
base_model = ResNet50(include_top=False, weights='imagenet')
```

پارامتر include_top میگوید که آیا لایه‌های آخر که برای کلاس بندی است و از نوع کاملا متصل است را نیز میخواهید یا نه؟ از آنجا که تعداد کلاسهای مسئله ما با مسئله imagenet متفاوت است پس این پارامتر را برابر با False قرار می‌دهیم. پارامتر weights هم میگوید که برای وزنهای اولیه از چه مقادیری استفاده کنم، مقدار imagenet یعنی از وزنهایی که این مدل روی مسئله imagenet بدست آورده است، به عنوان وزن اولیه استفاده کن.

حال باید به کمک این base_model شبکه خود را بسازیم. به یک لایه resize نیز نیاز داریم:

```
inputs = keras.Input(shape=(x_train[0].shape))
base_model = ResNet50(include_top=False, weights='imagenet')
resize_layer = tf.keras.layers.Resizing(
    224, 224, interpolation="bicubic", crop_to_aspect_ratio=False
)
# add custom head
x = resize_layer(inputs)
x = base_model(x)
x = Flatten()(x)
# x = Dense(256, activation='relu')(x)
x = Dense(10, activation='softmax')(x)

# define new model
model = keras.Model(inputs=inputs, outputs=x)
```

اگر یک summary از مدل بگیریم، به شکل زیر خواهد بود:

```
Model: "model_7"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_14 (InputLayer)       [(None, 32, 32, 3)]       0

 resizing_6 (Resizing)       (None, 224, 224, 3)       0

 resnet50 (Functional)       (None, None, None, 2048)  23587712

 flatten_7 (Flatten)         (None, 100352)            0

 dense_14 (Dense)            (None, 10)                1003530

=================================================================
Total params: 24,591,242
Trainable params: 1,003,530
Non-trainable params: 23,587,712
```

میتوان پارامتر expand_nested را در تابع summary را ست کرد تا لایه های درون resnet50 را هم نشان دهد.

برای freeze کردن لایه‌هایی که به ResNet50 مربوط میشود، میتوان یک حلقه روی لایه‌های base_model زده و پارامتر trainable آنها را برابر با False قرار داد.

```
# freeze base model layers
for layer in base_model.layers:
    layer.trainable = False
```

برای تابع ضرر و بهینه ساز از همان موارد الف و ب استفاده کردم:

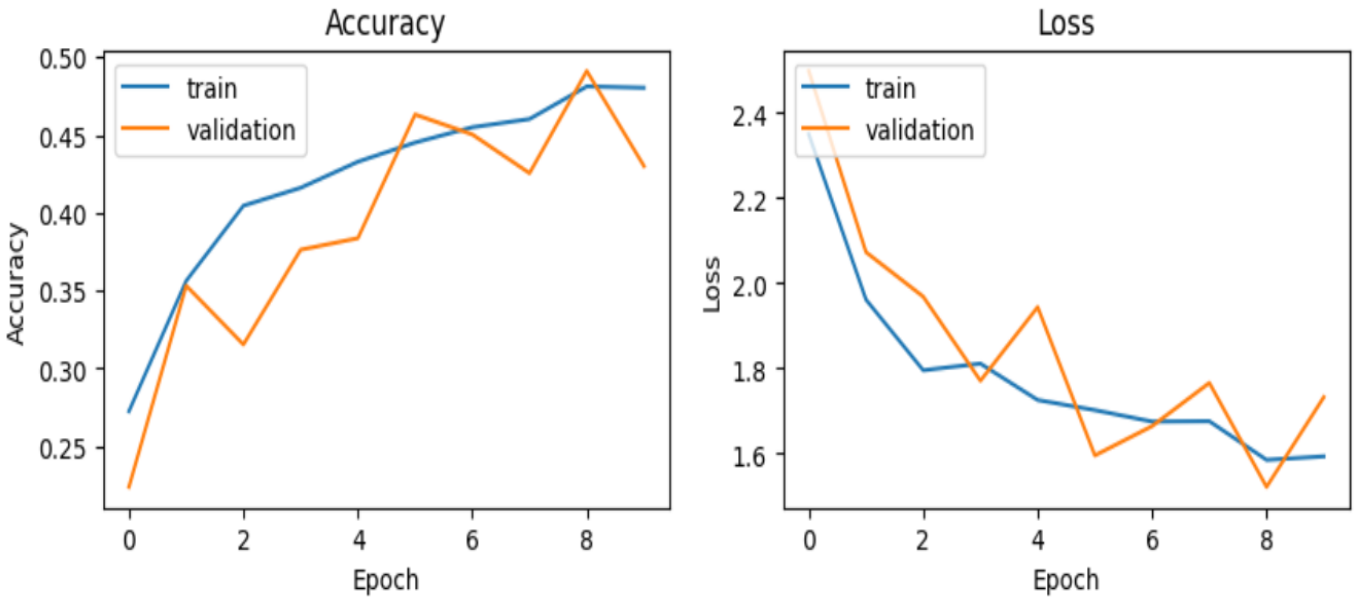
```
model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])
```

و نتیجه نهایی به صورت زیر بود:

```
history = model.fit(x_train,y_train, epochs=10, batch_size=64, validation_data=(x_val,y_val), shuffle=True)
```

```
Epoch 1/10
782/782 [==============================] - 205s 258ms/step - loss: 2.3502 - accuracy: 0.2726 - val_loss: 2.4989 - val_accuracy: 0.2239
Epoch 2/10
782/782 [==============================] - 205s 262ms/step - loss: 1.9617 - accuracy: 0.3563 - val_loss: 2.0733 - val_accuracy: 0.3532
Epoch 3/10
782/782 [==============================] - 205s 262ms/step - loss: 1.7959 - accuracy: 0.4046 - val_loss: 1.9688 - val_accuracy: 0.3154
Epoch 4/10
782/782 [==============================] - 205s 262ms/step - loss: 1.8115 - accuracy: 0.4161 - val_loss: 1.7713 - val_accuracy: 0.3763
Epoch 5/10
782/782 [==============================] - 196s 250ms/step - loss: 1.7260 - accuracy: 0.4328 - val_loss: 1.9443 - val_accuracy: 0.3837
Epoch 6/10
782/782 [==============================] - 204s 261ms/step - loss: 1.7022 - accuracy: 0.4452 - val_loss: 1.5958 - val_accuracy: 0.4632
Epoch 7/10
782/782 [==============================] - 204s 262ms/step - loss: 1.6753 - accuracy: 0.4550 - val_loss: 1.6647 - val_accuracy: 0.4504
Epoch 8/10
782/782 [==============================] - 205s 262ms/step - loss: 1.6762 - accuracy: 0.4603 - val_loss: 1.7662 - val_accuracy: 0.4255
Epoch 9/10
782/782 [==============================] - 205s 262ms/step - loss: 1.5854 - accuracy: 0.4812 - val_loss: 1.5220 - val_accuracy: 0.4912
Epoch 10/10
782/782 [==============================] - 205s 262ms/step - loss: 1.5936 - accuracy: 0.4804 - val_loss: 1.7327 - val_accuracy: 0.4301
```

و اگر نمودار loss و accuracy را برای داده آموزشی و تست رسم کنیم، داریم:



تحلیل نمودار: وقتی از وزنهای اولیه یک مدل بر روی مسئله‌ای دیگر، برای مسئله‌ی خودمان استفاده میکنیم، انتظار داریم روی دیتاست کوچک هم بتواند نتیجه‌ی خوبی بدهد و زودتر و بهتر به نقطه بهینه برسد. در نمودار بالا میبینیم که دقت نهایی روی داده تست و آموزش نسبت که حالتی که

Transfer learning نداشتیم، پایین‌تر است و حدسم اینه که دلیلش بخاطر trainable بودن تعداد کمی از لایه‌ها است. در واقع تنها لایه آخر که کاملا متصل است را قابل آموزش گذاشتیم و این باعث میشود مدل ما ظرفیت یادگیری‌اش کم شود و دقت نهایی کمتر شود. از لحاظ overfit شدن ظاهرا نمودار loss برای داده آموزش و تست در یک نگاه کلی روند نزولی داشته‌اند و accuracy هم برای هر دوی آنها نزدیک به هم است. برای حل مشکل دقت پایین میتوان لایه‌های بیشتری را قابل آموزش گذاشت یا به اصطلاح لایه‌های بیشتری را fine-tune کرد.

ه)

برای این حالت کافیست تا لایه‌ای که در صورت سوال آمده است را غیر قابل آموزش کنیم.

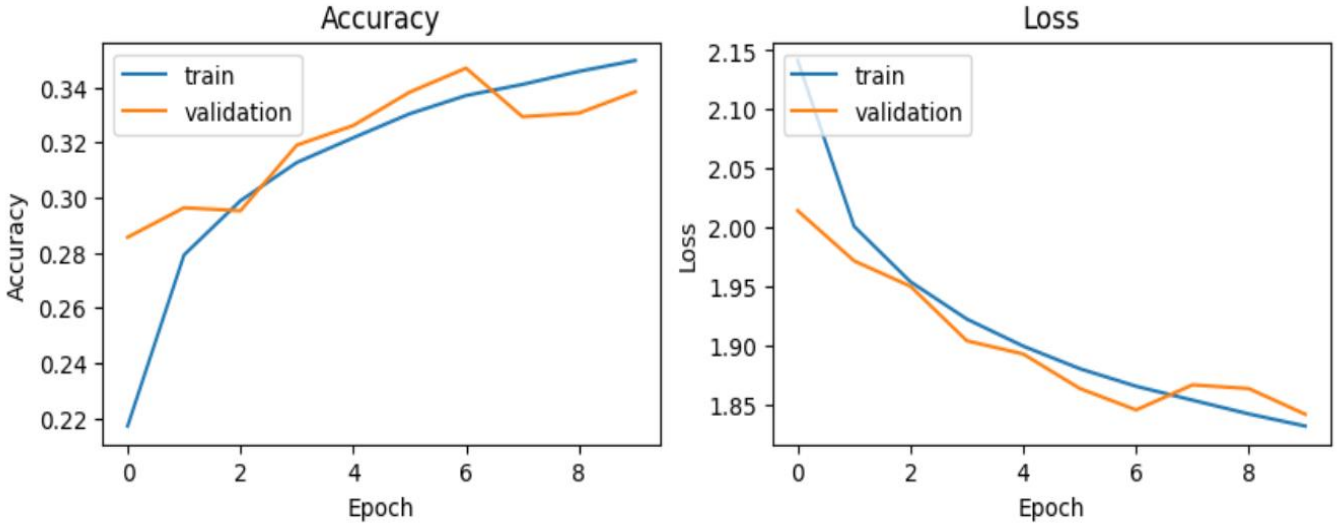
همه چیز همانند حالت د است به غیر از اینکه تعداد کمتری از لایه ها را فریز میکنیم. لایه‌ی ذکر شده در صورت سوال را در بین لایه‌ها جستجو کردم، ۸۰ امین لایه بود پس با کد زیر، آن لایه ها را فریز کردم.

```python
# freeze base model first three blocks
i = 0
for layer in base_model.layers:
  if(i < 81):
    layer.trainable = False
  else:
    break
```

و نتیجه به صورت زیر بود:

```
Epoch 1/10
782/782 [==============================] - 18s 18ms/step - loss: 2.1414 - accuracy: 0.2172 - val_loss: 2.0142 - val_accuracy: 0.2856
Epoch 2/10
782/782 [==============================] - 13s 17ms/step - loss: 2.0009 - accuracy: 0.2792 - val_loss: 1.9716 - val_accuracy: 0.2963
Epoch 3/10
782/782 [==============================] - 13s 16ms/step - loss: 1.9539 - accuracy: 0.2988 - val_loss: 1.9501 - val_accuracy: 0.2952
Epoch 4/10
782/782 [==============================] - 13s 17ms/step - loss: 1.9223 - accuracy: 0.3127 - val_loss: 1.9039 - val_accuracy: 0.3190
Epoch 5/10
782/782 [==============================] - 13s 17ms/step - loss: 1.8993 - accuracy: 0.3217 - val_loss: 1.8926 - val_accuracy: 0.3261
Epoch 6/10
782/782 [==============================] - 13s 17ms/step - loss: 1.8802 - accuracy: 0.3304 - val_loss: 1.8635 - val_accuracy: 0.3382
Epoch 7/10
782/782 [==============================] - 13s 16ms/step - loss: 1.8653 - accuracy: 0.3370 - val_loss: 1.8453 - val_accuracy: 0.3469
Epoch 8/10
782/782 [==============================] - 13s 17ms/step - loss: 1.8535 - accuracy: 0.3410 - val_loss: 1.8665 - val_accuracy: 0.3293
Epoch 9/10
782/782 [==============================] - 12s 16ms/step - loss: 1.8418 - accuracy: 0.3457 - val_loss: 1.8633 - val_accuracy: 0.3306
Epoch 10/10
782/782 [==============================] - 12s 16ms/step - loss: 1.8316 - accuracy: 0.3497 - val_loss: 1.8415 - val_accuracy: 0.3383
```

و نمودار آن نیز به شکل زیر در آمد:



تحلیل نمودار: در ابتدا مقدار loss زیاد است و این احتمالا بدین خاطر است این وزنها برای مسئله دیگری بودند و مدل ما کم کم باید این وزنها را به وزنهای مناسب برای مسئله ما نزدیک کرده تا به دقت بهتری برسد. میبینیم که باز هم overfit نشده است زیرا نمودار loss برای داده تست و آموزش همگام در حال کاهش است و نمودار accuracy برای آنها تفاوت فاحشی ندارد. اگر یادگیری را برای تعداد epoch بیشتر قرار میدادیم، قطعا دقت بهتر و بهتر میشد. اگر ده epoch دیگر آموزش را ادامه دهیم، داریم:

```
Epoch 1/10
782/782 [==============================] - 13s 16ms/step - loss: 1.8242 - accuracy: 0.3521 - val_loss: 1.8332 - val_accuracy: 0.3500
Epoch 2/10
782/782 [==============================] - 13s 16ms/step - loss: 1.8183 - accuracy: 0.3556 - val_loss: 1.8154 - val_accuracy: 0.3488
Epoch 3/10
782/782 [==============================] - 12s 15ms/step - loss: 1.8080 - accuracy: 0.3624 - val_loss: 1.7984 - val_accuracy: 0.3657
Epoch 4/10
782/782 [==============================] - 12s 16ms/step - loss: 1.8020 - accuracy: 0.3636 - val_loss: 1.7971 - val_accuracy: 0.3605
Epoch 5/10
782/782 [==============================] - 13s 16ms/step - loss: 1.7948 - accuracy: 0.3666 - val_loss: 1.8136 - val_accuracy: 0.3582
Epoch 6/10
782/782 [==============================] - 12s 15ms/step - loss: 1.7912 - accuracy: 0.3678 - val_loss: 1.7898 - val_accuracy: 0.3686
Epoch 7/10
782/782 [==============================] - 12s 16ms/step - loss: 1.7873 - accuracy: 0.3714 - val_loss: 1.8138 - val_accuracy: 0.3485
Epoch 8/10
782/782 [==============================] - 12s 16ms/step - loss: 1.7804 - accuracy: 0.3698 - val_loss: 1.7874 - val_accuracy: 0.3736
Epoch 9/10
782/782 [==============================] - 13s 17ms/step - loss: 1.7759 - accuracy: 0.3708 - val_loss: 1.8220 - val_accuracy: 0.3354
Epoch 10/10
782/782 [==============================] - 12s 15ms/step - loss: 1.7728 - accuracy: 0.3762 - val_loss: 1.7701 - val_accuracy: 0.3784
```

که دقت تا ۳۷ درصد میرسد. چیزی که مشخص است این است که باز هم داریم تعداد لایه های زیادی را فریز میکنیم و این باعث پایین آمدن ظرفیت یادگیری شبکه میشود و نتیجتا به دقت پایینی میرسیم.

هرچند با کم کردن لایه‌های قابل آموزش ممکن است به مشکل دیگری که همان overfitting باشد بربخوریم زیرا به مدل اجازه داده‌ایم الگوهای پیچیده‌تری را بیاموزد. پس کاملا یک tradeoff است.

سوال۴-

الف) مفهوم stride بدین صورت است که هنگام محاسبه کانولوشن در یک لایه کانولوشنی، مقدار پرش بعد از هر محاسبه برای یک پیکسل را مشخص میکند، برای مثال stride=1 یعنی پس از محاسبه کانولوشن برای یک پیکسل، یک گام به جلو برو و این مقدار را برای پیکسل جدید هم حساب کن، یا stride=2 یعنی پس از محاسبه کانولوشن برای یک پیکسل، دو گام به جلو برو و برای پیکسل جدید، این مقدار را محاسبه کن که در این حالت سایز خروجی یا feature map پس از کانوالو، نصف حالت قبل میشود.

اما pooling در شبکه های عصبی مفهوم متفاوتی دارد، در pooling، نقشه ویژگی را به چند بخش یا region تقسیم میکنیم و برای هر region بسته به نوع pooling یک عدد را قرار میدهیم، برای مثال در max pooling مقدار ماکزیمم آن بخش و در average pooling مقدار میانگین آن بخش را قرار میدهیم. البته میتوان در کنار pooling مقدار stride را نیز مشخص کرد ولی به طور پیش‌فرض مقدار آن برابر است با سایز regionها در pooling. برای مثال اگر بخواهیم در قسمت های ۲ در ۲ pooling بزنیم، در keras مقدار stride نیز (2,2) خواهد بود مگر اینکه آن را تغییر دهیم.

کاربرد Stride در شبکه های عصبی، کاهش ابعاد خروجی یک لایه و در نتیجه کاهش تعداد پارامترها و کاهش محاسبات است.

کاهش تعداد پارامترها به overfit نشدن مدل کمک میکند و سرعت training را نیز بالا میبرد.

اما در pooling همچنان باید کل خروجی محاسبه شود و سپس pooling انجام شود، پس از لحاظ سرعت در این مرحله حتی زمان بیشتری میگیرد زیرا یک مرحله محاسبات بیشتر دارد، اما باعث کاهش ابعاد مکانی خروجی میشود و این باعث میشود در لایه های بعد از این لایه، دوباره سرعت بهتر شود و تعداد پارامترها نیز کمتر شود تا به مشکل overfitting کمک کند.

در stride ما بخشی از اطلاعات مکانی را داریم از دست می‌دهیم و اگر بیش از حد شود، باعث میشود مدل نتواند به خوبی یاد بگیرد. در واقع pooling برای این است که این از دست دادن اطلاعات را با روشهایی مثل ماکزیمم گرفتن نامحسوس‌تر کند.

برای محاسبه ابعاد خروجی بر حسب ابعاد ورودی در حالتی که stride داریم، از فرمول زیر که در اسلاید درس است، استفاده میشود.

$$W_2 = (W_1 - F + 2P)/S + 1$$
$$H_2 = (H_1 - F + 2P)/S + 1$$
$$D_2 = K$$

که F سایز فیلترها را مشخص میکند، S مقدار stride و P مقدار padding را اگر padding داشته باشیم. K هم تعداد فیلترها را نشان میدهد. ورودی یک حجم با ابعاد W1 * H1 * D1 و خروجی یک حجم با ابعاد W2 * H2 * D2 .

ب)

۱- پیشنهاد من برای لایه‌های میانی استفاده از Relu است که از لحاظ محاسباتی بسیار ساده است و در عمل هم ثابت شده که عملکرد خوبی داشته است و مشکل vanishing gradient را نیز برعکس sigmoid ندارد. البته برای این که به مشکل مشتق صفر برای مقادیر منفی برخورد نکنیم، میتوانیم از Leaky Relu استفاده کنیم ولی در آن صورت مزیت محاسبات ساده تر را کمی از دست میدهیم. برای لایه‌ی آخر به نظرم بهتر است از Softmax استفاده کنیم، زیرا خروجی ها را به اعداد احتمال تبدیل میکند که جمعشان برابر با ۱ شود و میتوان پس از آن prediction را انجام داد.

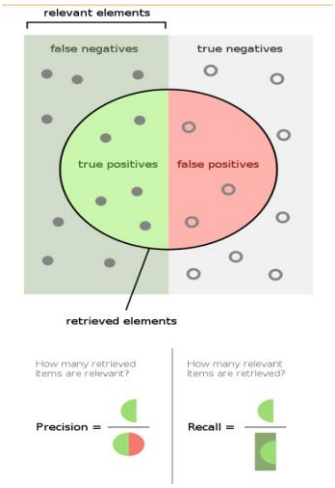
۲- برای تابع ضرر، cross entropy میتواند گزینه خوبی برای همه مسائل classification باشد، زیرا probability distribution واقعی را با پیش‌بینی شده مقایسه میکند و فرمول آن به صورت زیر است:

Intuitively Understanding the Cross Entropy

$$H(P^*|P) = -\sum_i \underbrace{P^*(i)}_{\text{TRUE CLASS DISTRIBUTION}} \log \underbrace{P(i)}_{\text{PREDICTED CLASS DISTRIBUTION}}$$

پیش‌بینی هایی که از مقدار درست دورتر باشند، بیشتر روی این مقدار تاثیر گذاشته و موجب جریمه بیشتر میشوند و این کمک میکند مدل ما کم کم به سمت پیش بینی های درست پیش برود.

-۳



معیار Recall برابر است با درصدی از محصولات معیوب که مدل ما معیوب بودن آن را تشخیص داده است. پس اگر بخواهیم کمترین محصول معیوب به دست مشتری برسد، میتوان از این معیار صحت استفاده کرد، اگر این مقدار یک باشد، یعنی همه محصولات معیوب توسط مدل ما شناسایی شده و هیچ محصول معیوبی بدست مشتری نرسیده است. اما مشکلی که این معیار دارد، در نظر نگرفتن false positive (positive here means being defective) ها است، یعنی محصولاتی که مدل ما معیوب تشخیص داده ولی در واقع معیوب نبودند. حتی اگر مدل ما همه محصولات را معیوب تشخیص دهد(که اینطور نیست) باز هم معیار Recall یک میشود. پس در برخی شرایط باید معیار صحت را عوض کرد و برای مثال از precision استفاده کرد، این معیار برابر است با تعداد محصولات معیوبی که مدل ما درست تشخیص داده تقسیم بر تعداد کل محصولاتی که مدل ما معیوب تشخیص داده است. اگر این مقدار برابر با یک باشد، یعنی همه محصولاتی که توسط مدل، معیوب شناخته شده‌اند، واقعا معیوب بودند، هرچند ممکن است محصولات معیوبی باشند که توسط مدل سالم تشخیص داده شده اند ولی آن ها در این معیار تاثیری ندارند.

پس اگر از recall استفاده کنیم، میتوان کمترین محصول معیوب را بدست مشتری رساند ولی اگر از precision استفاده کنیم، میتوان بیشترین محصول سالم را بدست مشتری رساند هرچند محصول معیوب هم بدستشان میرسد. به طور خلاصه، یک trade-off است و باید در شرایط مختلف تصمیم گیری صورت گیرد.

ج)

۱- شبکه های عصبی کانولوشن (CNN) در طبقه بندی موضوع متن موفق نخواهند بود. دلیل این امر این است که CNN ها برای کار بر روی داده های ساختار یافته از لحاظ مکانی، مانند تصاویر، و نه بر روی داده های متوالی مانند متن طراحی شده اند. درون یک عکس، ما با یک مجموعه از پیکسل ها مواجه هستیم که ارتباط مکانی دارند، برای مثال لبه جایی هست که پیکسلهای مجاور تفاوت مقدار زیادی دارند ولی در text اینطور نیست، زیرا در text کلمه‌ها معنای خاص خودشان را دارند و ممکن است در متنهای مختلف معنای مختلف داشته باشند. برای وظایف طبقه‌بندی متن، شبکه‌های عصبی بازگشتی (RNN) و انواع آن‌ها، مانند شبکه‌های حافظه کوتاه‌مدت (LSTM) و واحدهای بازگشتی دردار (GRU) مؤثرتر نشان داده شده‌اند.

۲- از CNN می توان برای شناسایی گوینده از روی صدا استفاده کرد. این به این دلیل است که صدا شکلی از داده های موجی است و می تواند به عنوان یک سیگنال ۱ بعدی در نظر گرفته شود. CNN ها را می توان برای کار با سیگنال های ۱ بعدی وفق داد و با موفقیت برای تشخیص گفتار و وظایف شناسایی سخنران استفاده شده‌اند.

۳- CNN ها همچنین می توانند برای تجزیه و تحلیل داده های مشتری برای پیش بینی رفتار بعدی هر مشتری استفاده شوند. این به این دلیل است که داده‌های مشتری را می‌توان به صورت داده‌های ساختاریافته، مانند جدول، نشان داد و CNNها را می‌توان برای استخراج ویژگی‌های مرتبط از این داده‌ها استفاده کرد. با این حال، انواع دیگر شبکه های عصبی، مانند شبکه های عصبی بازگشتی (RNN) و شبکه های حافظه کوتاه مدت (LSTM) نیز ممکن است برای این کار موثر باشند، بسته به ویژگی های خاص داده ها و ماهیت کار پیش بینی. مثلا در تصویر جدولی که در صورت سوال قرار داده شده است، میتوان فیلترهایی با ابعاد ۱ * (تعداد اطلاعات درباره اون مشتری یا همون تعداد ستونها) رو اعمال کرد و برای هر مشتری به صورت جداگانه ویژگیهایی را استخراج کرد و طبق آن ویژگی‌ها رفتارهای بعدی مشتری رو پیش‌بینی کرد.

د)

۱- overfitting: CNN ها می توانند مستعد overfitting باشند، که زمانی رخ می دهد که مدل یاد می گیرد که داده های آموزشی را خیلی خوب طبقه بندی کند و نتواند به داده های جدید و نادیده تعمیم دهد. اگر مدل خیلی پیچیده باشه یا داده های آموزشی کافی وجود نداشته باشن، ممکنه این اتفاق بیفته. تکنیک‌هایی مانند dropout , data augmentation، می‌توانند برای جلوگیری از آن مورد استفاده قرار بگیرند.

۲- نیاز به تعداد زیاد داده: در واقع مشابه مورد ۱ است، اگر داده کم باشد، به مشکل overfitting برخورد میکنیم یا اینکه اصلا مدل روی داده آموزشی هم دقت پایین می‌دهد.

۳- انعطاف پذیری محدود: CNN ها برای کار بر روی داده های ساختار یافته از لحاظ مکانی، مانند تصاویر، طراحی شده اند و ممکن است برای انواع دیگر داده ها، مانند داده های متوالی(متن) یا نموداری مناسب نباشند. انواع مختلف شبکه های عصبی، مانند شبکه های عصبی تکراری (RNN) یا شبکه های کانولوشن گراف (GCNs)، ممکن است برای این نوع داده ها مناسب تر باشند.

۴- تفسیرپذیری پایین: CNN ها اغلب به عنوان مدل های "جعبه سیاه" نامیده می شوند زیرا تفسیر آنها ممکن است دشوار باشد. درک اینکه مدل چگونه پیش‌بینی‌های خود را انجام می‌دهد، می‌تواند چالش‌برانگیز باشد، که می‌تواند در کاربردهایی که تفسیرپذیری مهم است، مانند مراقبت‌های بهداشتی یا مالی، مشکل‌ساز باشد. این موارد در حوزه explainable AI گنجانده می‌شود.

۵- این مشکل از نظر خودم توی CNN ها است و اون اینه که تعداد HyperParameter هاش زیاده و پیدا کردن اون مقادیر درستی که مدل به بهترین نحو باهاش کار کنه سخت میشه و زمانبر.

۶- در حالی که شبکه‌های عصبی کانولوشنال (CNN) در طیف گسترده‌ای از وظایف مرتبط با تصویر موفق بوده‌اند، کاربردهای خاصی وجود دارند که ممکن است بهترین انتخاب نباشند. یکی از این کاربردها پردازش زبان است، مانند پردازش زبان طبیعی (NLP) یا تشخیص گفتار. دلیل این امر این است که CNN ها برای کار بر روی داده های ساختار یافته مکانی، مانند تصاویر، طراحی شده اند، در حالی که داده های زبان متوالی هستند و ساختار مکانی یکسانی ندارند.

- https://aspiringyouths.com/advantages-disadvantages/convolutional-neural-network-cnn

(also by the help of chatGpt)

سوال۵-

مرحله اول در انجام این تسک، جمع‌آوری دیتاست آن و شناخت این دیتا است. دیتاست این مسئله قبلا جمع‌آوری و در اختیار ما قرار گرفته است.

دیتاستی که در اختیار ما قرار گرفته است، پوشه‌ای شامل ۵ دایرکتوری درون آن است که هر دایرکتوری تعدادی تصویر ورودی و تعدادی برچسب برای آن تصاویر ورودی که خودشان نیز تصویر هستند، وجود دارد. برای یکپارچه کردن همه‌ی عکسهای ورودی و همه تصاویر برچسب در کنار هم از کد زیر استفاده میکنیم:

```
data_dir = '/content/ss_dataset'
img_size = 256
!rm -rf ./train
!rm -rf ./train_masks
image_root = '/content/train'
label_root = '/content/train_masks'

if not os.path.isdir(image_root):
    os.mkdir(image_root)
if not os.path.isdir(label_root):
    os.mkdir(label_root)

images = list()
labels = list()
```

کل دیتاست درون پوشه content/ss_dataset قرار دارد. دو پوشه در دایرکتوری content به اسمهای train , train_masks میسازیم.

```
for (dirpath, dirname, filenames) in os.walk(data_dir):
  for filename in filenames:
    img_name = filename.split('.')[0]
    if 'label' in img_name:
      labels.append(dirpath + f'/{filename}')
    else:
      images.append(dirpath + f'/{filename}')
```

با استفاده از این کد، یکبار همه فایلهای درون پوشه ss_dataset را بررسی میکنیم، اگر فایل مربوط به برچسب بود، مسیر آن را در لیست labels و در غیر اینصورت در لیست images میریزیم.

حال باید این عکسها و برچسب‌هایشان را در پوشه های جداگانه بریزیم:

```
for img_path in images:
  file_name = img_path.split('/')[-1].split('.')[0]
  img = Image.open(img_path)
  img = img.resize((256, 256))
  dir_name = img_path.split('/')[-2]
  img.save(image_root + '/' + dir_name + '_' + file_name + '.png', 'png')

for label_path in labels:
  file_name = label_path.split('/')[-1].split('.')[0].replace('_label', '')
  img = Image.open(label_path)
  img = img.resize((256, 256))
  dir_name = label_path.split('/')[-2]
  img.save(label_root + '/' + dir_name + '_' + file_name + '.png', 'png')
```

در اینجا ابتدا روی عکسهای ورودی لوپ میزنیم و یک آبجکت Image از کتابخونه PIL می‌سازیم. سپس آن را ریسایز میکنیم و در نهایت یک فایل از آن ساخته و در پوشه train و با فرمت png ذخیره میکنیم. اینجا چون درون پوشه‌هایی که درون پوشه‌ی ss_dataset بودند، فایل با اسم یکسان وجود داشت، پس کاری که کردیم، اضافه کردن اسم آن دایرکتوری به قبل از اسم فایل عکس بود. برای برچسب‌ها هم همین کار را کردیم.

```
def dataframe_creation(image_path, name):
    """
    This function is storing path of the images in a dataframe beside of each image id (in image name).
    Walk though the image_path and read the dirpathes and images name in each dir.
    Then append each image full path in a list.
    Extract the image name without the full path and extension and append it to the ids list.
    Please make sure each full path in first list is correspond to the id in second list at the same index.
    Arguments:
      image_path: root directory full path
      name: name for column of full pathes in dataframe
    return:
      df: a df contains ids and full path of each image id. call the ids column, id and pathes column, name.
          Please set the ids column to be index in this df.

    """
    img_pathes = []
    img_names = []
    for (dirpath, dirname, filenames) in os.walk(image_path):
      for file_name in filenames:
        img_pathes.append(dirpath + '/' + file_name)
        img_names.append(file_name.split('.')[0])

    df = {'id': img_names, name: img_pathes}
    df = pd.DataFrame(df)
    df.set_index('id')

    return df
```

تابع بالا، مسیر پوشه‌ای که دیتا یا برچسبهای دیتا درون آن قرار دارد را گرفته و یک dataframe برای کار کردن راحتتر با دیتاست برمیگرداند. با استفاده از این تابع، به شکل زیر دو تا دیتافریم، یکی برای عکسهای ورودی و یکی برای برچسبها میسازیم. همچنین درون همان train_df و تحت کلید mask_path ، برچسب مربوط به هر تصویر را نیز ذخیره میکنیم تا همه چیزهایی که میخواهیم را یکجا داشته باشیم.

```
train_df = dataframe_creation(image_root, 'image_path')
mask_df = dataframe_creation(label_root, 'mask_path')
train_df['mask_path'] = mask_df['mask_path']
```

۵ مورد اول این دیتافریم به صورت زیر میباشد:

	id	image_path	mask_path
0	0_140	/content/train/0_140.png	/content/train_masks/0_140.png
1	0_99	/content/train/0_99.png	/content/train_masks/0_99.png
2	0_316	/content/train/0_316.png	/content/train_masks/0_316.png
3	0_542	/content/train/0_542.png	/content/train_masks/0_542.png
4	3_104	/content/train/3_104.png	/content/train_masks/3_104.png

تابع data_augmentation:

```
def data_augmentation(img, mask_img):
    """
    A function for data augmentation.
    We wanna just do some flips.
    Just make a random number, if it was greater than 0.5 do a lef_right flip
    hint:
      https://www.tensorflow.org/api_docs/python/tf/random/uniform
      https://www.tensorflow.org/api_docs/python/tf/image/flip_left_right
    Arguments:
      img: image tensor
      mask_img: mask image tensor
    return:
      img, mask_img
    """
    img_fliped = img
    mask_fliped = mask_img
    if np.random.randint(0, 1) > 0.5:
      img_fliped = tf.image.flip_left_right(img)
      mask_fliped = tf.image.flip_left_right(mask_img)
    return img_fliped, mask_fliped
```

این تابع، یک تصویر را ورودی گرفته و با احتمال ۵۰ درصد، هم ورودی و هم برچسب آن را flip میکند، flip کردن برچسب هم بسیار مهم است در این مسئله. این میشود افزودن داده در این مسئله.

```python
def preprocessing(path, mask_path):
    '''
    Do the usual preprocessing steps for image processing algorithms
    Read image tensors. decode them, resize to img_size, cast them fo float dtype and normalize between 0-1
    Hint:
      https://www.tensorflow.org/api_docs/python/tf/io/read_file
      https://www.tensorflow.org/api_docs/python/tf/io/decode_jpeg
      https://www.tensorflow.org/api_docs/python/tf/image/resize
      https://www.tensorflow.org/api_docs/python/tf/cast
    Set channels in decoding to 3
    Arguments:
      path: image path
      mask_path: mask image path
    return:
      pre_processed image and mask image tensors
    '''
    img = tf.io.read_file(path)
    img = tf.io.decode_jpeg(img, channels=3)
    img = tf.image.resize(img, img_size)
    img = tf.cast(img, tf.float32)/255.0

    mask_img = tf.io.read_file(mask_path)
    mask_img = tf.io.decode_jpeg(mask_img, channels=3)
    mask_img = tf.image.resize(mask_img, img_size)
    mask_img = mask_img[:, :, 0]
    mask_img = tf.math.sign(mask_img)

    return img, mask_img
```

تابع preprocessing را هم طبق توضیحات پیاده‌سازی میکنیم، ابتدا عکس را از مسیر داده شده با استفاده از کتابخانه tensorflow خوانده و آن را به فرمت jpeg دیکود میکنیم و پس از ریسایز کردن، آن را به اعداد بین صفر و یک نرمالایز میکنیم. برای برچسب هم همان کار انجام میدهیم به غیر از اینکه در برچسب هر پیکسل یا متعلق به کلاس پنل خورشیدی هست یا نیست پس مقدار باینری ۰ و ۱ میتوان به آن نسبت داد و برای این کار کافی بود تنها از یک بعد تصویر استفاده کنیم و نیازی به استفاده از هر سه کانال نبود. همچنین برای تشخیص اینکه این مقدار بزرگتر از ۱۲۸ است کافی بود بیت آخر یا بیت علامت استفاده شود. اگر مقدارش یک بود، یک بذار و در غیر اینصورت صفر.

```python
def create_dataset(df, train = False):
    '''
    A function for applying preprocessing and augmentation steps.
    Augment data just in train mode.
    First make a Dataset of tensors to reach high speed and ability.
    Then apply needed steps.
    For creating dataset, please use tensorflow-tf-data-dataset-from_tensor_slices to get
      a dataset of images and correspondig masks path. use values of image_path and mask_path columns of your dataframe
    Then use map function of created ds and call above functions respectively.
    use tf.data.AUTOTUNE in map function
    Hint:
      https://www.geeksforgeeks.org/tensorflow-tf-data-dataset-from_tensor_slices/
      https://www.tensorflow.org/api_docs/python/tf/data/Dataset
    Arguments:
      df: dataframe of images with masks path.
      train: boolean for switching between train and inference mode.
    return:
      dataset
    '''
    ds = tf.data.Dataset.from_tensor_slices((df['image_path'].values, df['mask_path'].values))
    ds = ds.map(preprocessing, tf.data.AUTOTUNE)
    if train:
      ds = ds.map(data_augmentation, tf.data.AUTOTUNE)

    return ds
```

برای ساخت دیتاست از تابع from_tensor_slices در tensorflow استفاده میکنیم، مقادیر ورودی و برچسبها را به عنوان ورودی به آن میدهیم. روی همه داده ها، اعم از داده تست و آموزشی پیش‌پردازش‌ها را انجام میدهیم، اما افزودن داده فقط برای داده‌ی آموزشی انجام میشود نه داده‌ی تست.

```
train_df, valid_df = train_test_split(train_df, test_size=0.2)
train = create_dataset(train_df, True)
valid = create_dataset(valid_df)
```

با استفاده از تابع train_test_split که از کتابخانه sklearn آوردیم، بخشی از داده‌ی آموزشی را برای validation استفاده میکنیم و سپس دیتاست مربوط به هر کدام را میسازیم.

```
TRAIN_LENGTH = len(train_df)
BATCH_SIZE = 24
BUFFER_SIZE = 1000
```

مقادیر سایز هر batch و سایز بافر را ست میکنیم.

```
train_dataset = train.cache().shuffle(BUFFER_SIZE).batch(BATCH_SIZE).repeat()
train_dataset = train_dataset.prefetch(buffer_size=tf.data.AUTOTUNE)
valid_dataset = valid.batch(BATCH_SIZE)
```

با استفاده از این کد، دیتاست را به batch های مختلف که داده‌ها در آن shuffle شده‌اند، تبدیل میکنیم. برای داده validation عملیات shuffle لازم نیست.

یک نمونه از ورودی و برچسب آن:

مرحله جمع‌آوری دیتاست و مرتب کردن آن به پایان رسید، حال نوبت پیاده‌سازی یک مدل encoder-decoder که برای این مسئله مرسوم است، می‌باشد.

ما در اینجا میخواهیم، مدلی مشابه U-Net را پیاده کنیم ولی برای قسمت encoding از شبکه MobileNetV2 به عنوان backbone استفاده میکنیم و قسمت decoding را خودمان مینویسیم.

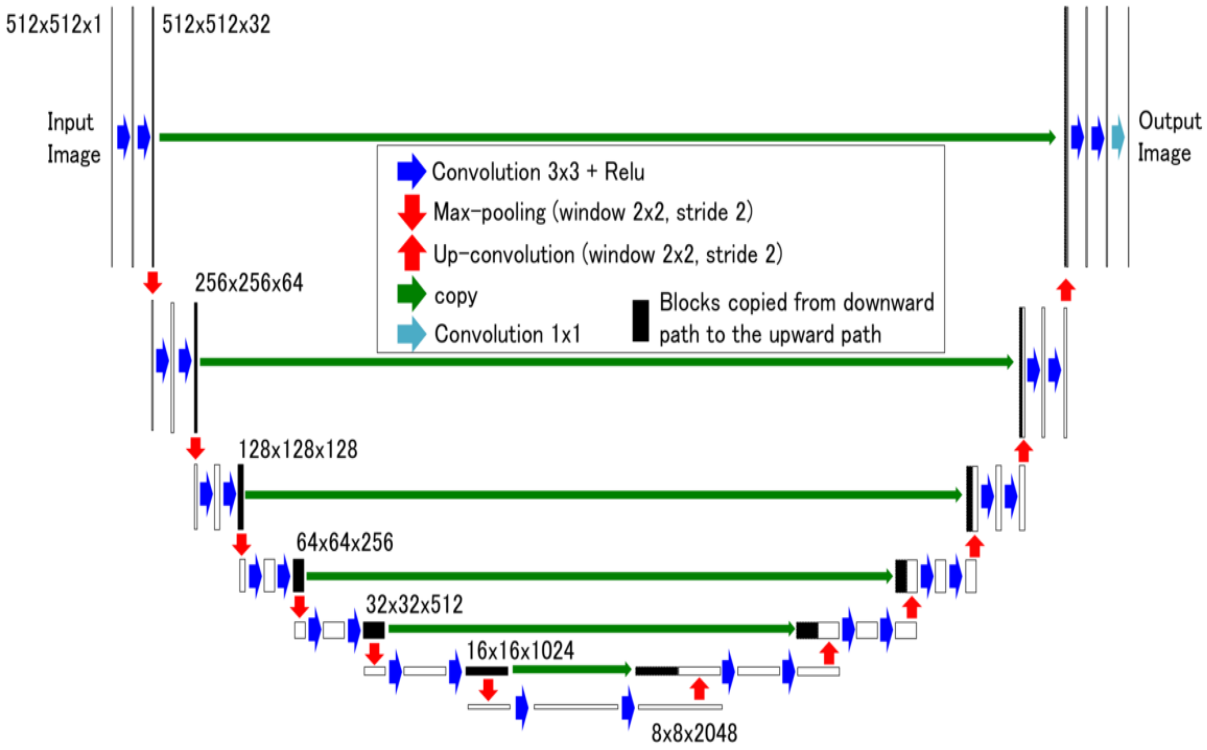
```
img_size = (256, 256)
backbone = tf.keras.applications.MobileNetV2(img_size + (3,), include_top=False, weights='imagenet')
layer_names = ['block_1_expand_relu', 'block_3_expand_relu', 'block_6_expand_relu',
               'block_13_expand_relu', 'block_16_project']
output_layers = [backbone.get_layer(name).output for name in layer_names]
backbone = tf.keras.Model(inputs=backbone.input, outputs=output_layers)
backbone.trainable = False
backbone.summary(expand_nested=True)
```

همانطور که میبینید، در اینجا از وزنهای اولیه این مدل، روی مسئله imagenet نیز استفاده میکنیم، زیرا دیتاست این مسئله بزرگ نیست و استفاده از transfer learning ضروری است. کل این قسمت را freeze میکنیم، همچنین تعداد outputهای آن را در چندین مرحله قرار میدهیم تا بتوانیم از آن output بعدا برای decoding استفاده کنیم.

تابع upsampling:

```
initializer = tf.random_normal_initializer(0., 0.02)

################################################
result = tf.keras.Sequential()
result.add(tf.keras.layers.Conv2DTranspose(filters=filters, kernel_size=size, strides=(2, 2),
                                           padding='same', kernel_initializer=initializer, use_bias=False))
result.add(tf.keras.layers.BatchNormalization())
################################################
```

در این تابع از لایه‌های Conv2DTranspose و BatchNormalization و توضیحات تابع برای decode کردن استفاده میکنیم.



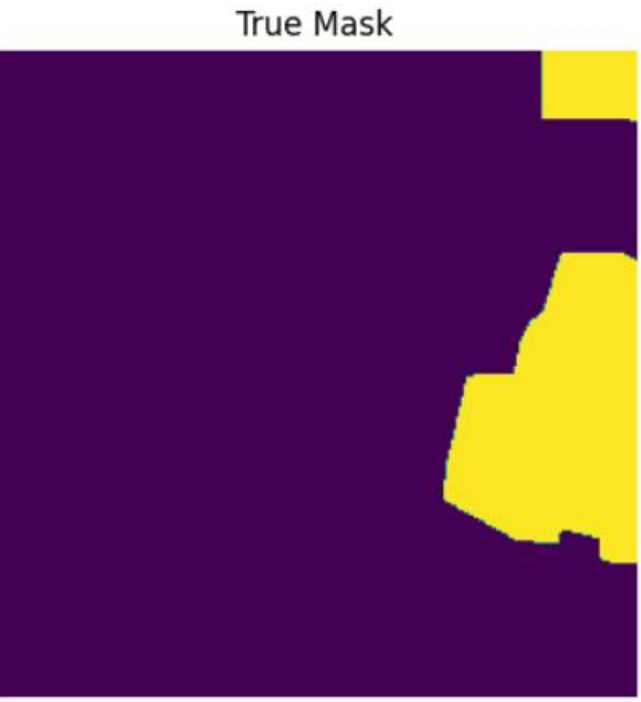
بلاک های decoder را داخل up_stack قرار دادهایم. برای اینکه خروجی های backbone را به بلاک مدنظر آن در up_stack نظیر کنیم، باید خروجی ها را reverse کنیم .حال در decoder خروجی لایه قبلی را با خروجی لایه نظیر آن در encoder با هم concatenateمیکنیم. در آخر چون نیاز داریم که با استفاده از upsampling از map feature به دست آمده mask را بسازیم، از لایه Conv2dTranspose استفاده میکنیم.

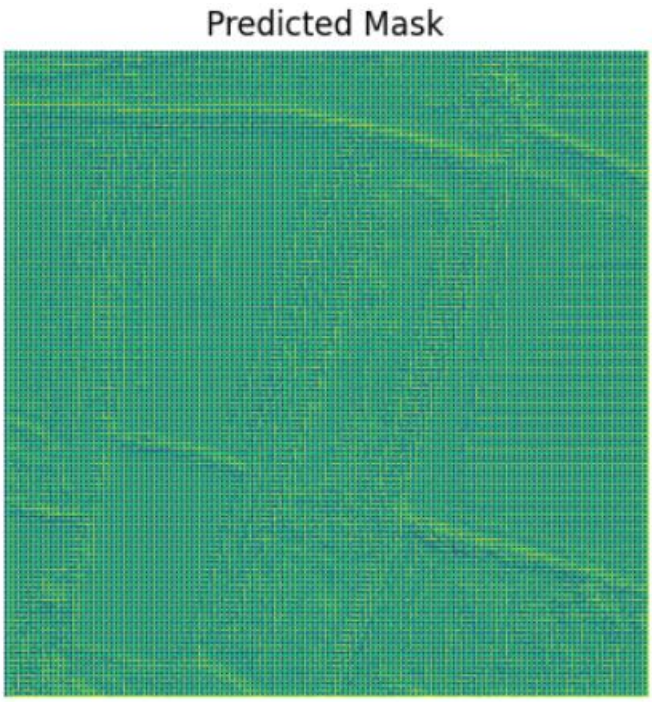
```
up_stack = [
    upsample(512, 3),  # 4x4 -> 8x8
    upsample(256, 3),  # 8x8 -> 16x16
    upsample(128, 3),  # 16x16 -> 32x32
    upsample(64, 3),   # 32x32 -> 64x64
]
```

برای کامپایل کردن مدل از بهینه‌ساز adam و تابع ضرر dice_loss که بالاتر تعریف کردیم، استفاده میکنیم.

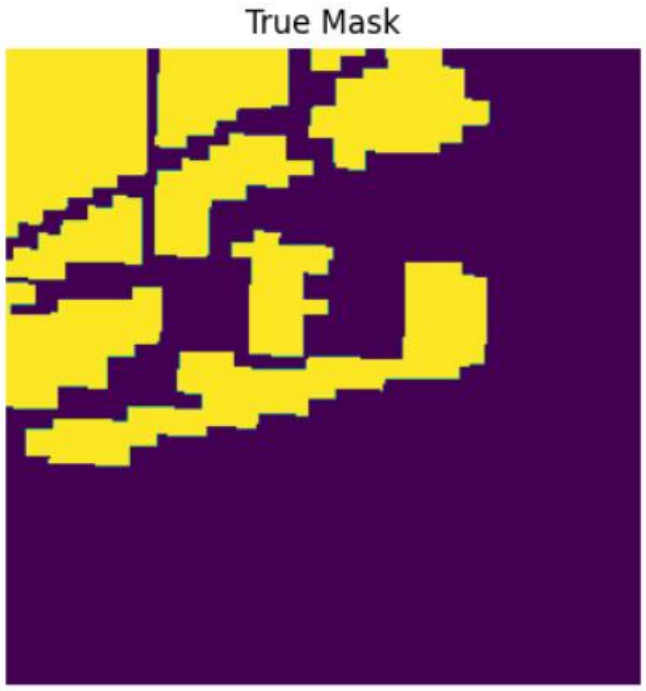
```
model = unet_model(1)
model.compile(optimizer='adam',
              loss=dice_loss,
              metrics=['binary_accuracy', dice_coef])

tf.keras.utils.plot_model(model, show_shapes=True)
```

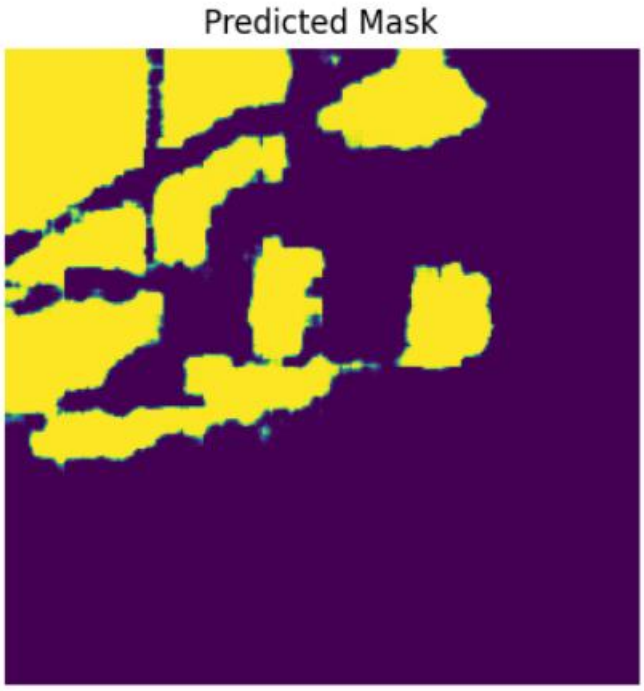
قبل از آموزش مدل:





بعد از آموزش مدل:





میبینیم که عملکر مدل پس از آموزش تا حد خوبی بهتر شد.