به نام خدا

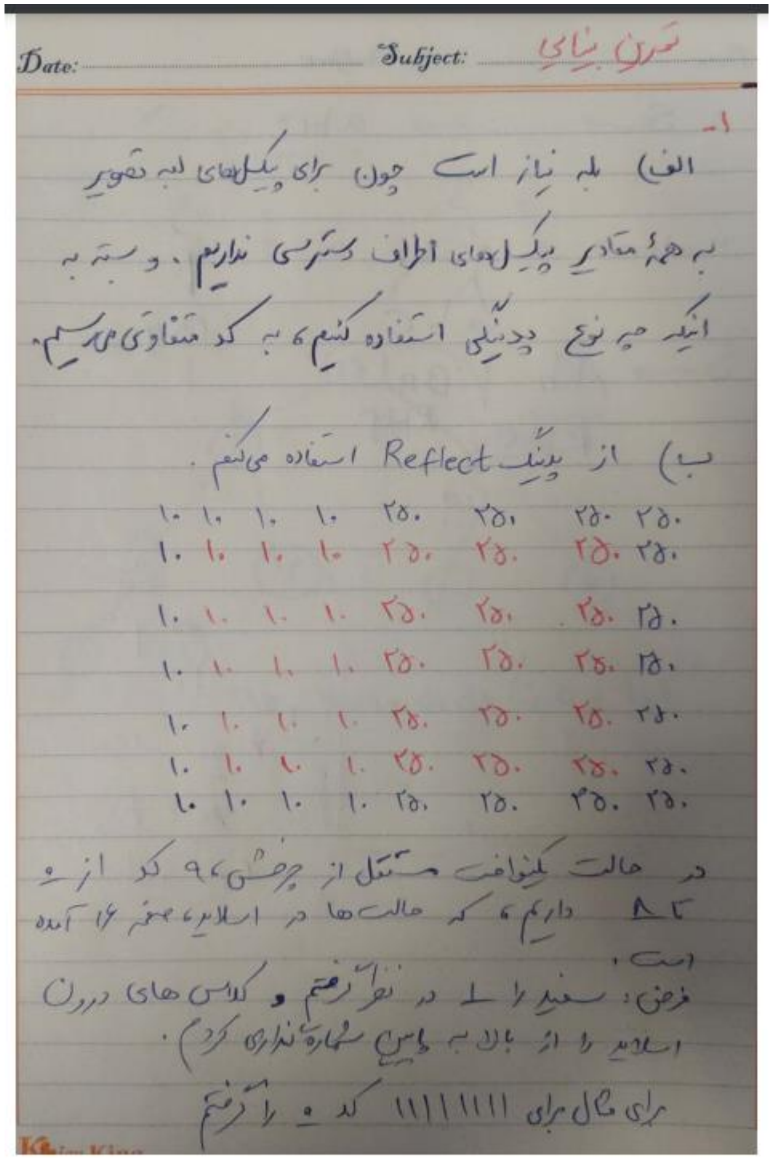
تمرین سری پنجم

درس بینایی کامپیوتر

سینا علی‌نژاد

شماره دانشجویی: ۹۹۵۲۱٤٦۹

سوال ۱-



Date: ...... Subject: تمرین بینایی

۱-

الف) بله نیاز است چون برای پیکسل‌های لبه تصویر به همه مقادیر پیکسل‌های اطراف دسترسی نداریم. و بسته به اینکه چه نوع پدینگی استفاده کنیم، به کد متفاوتی می‌رسیم.

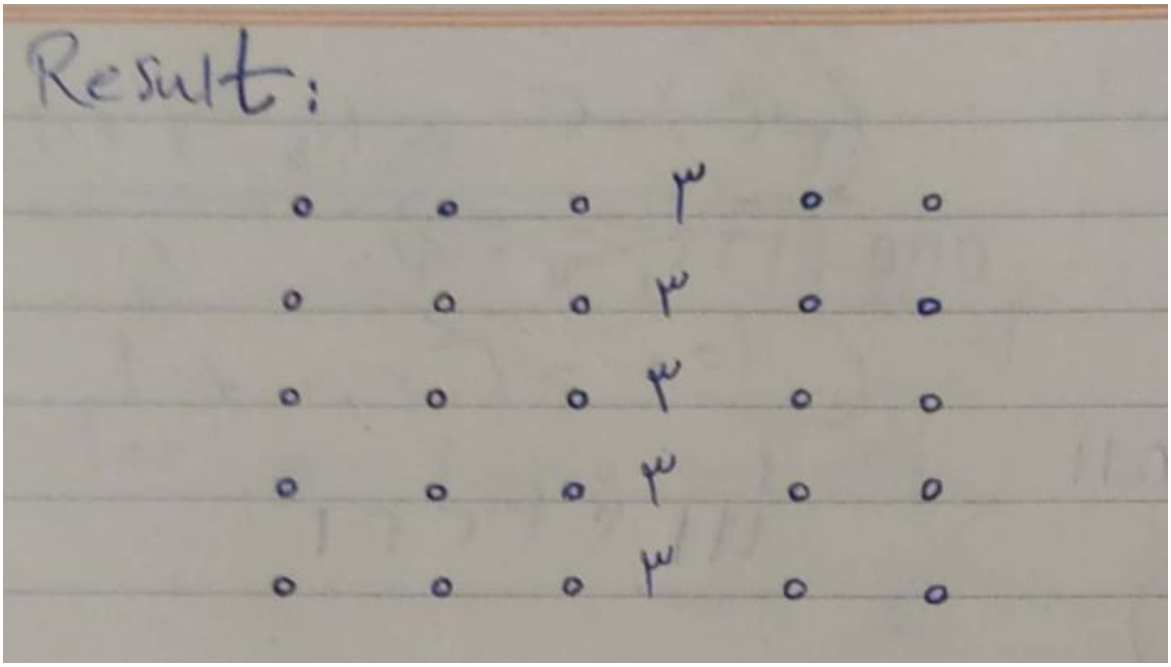
ب) از پدینگ Reflect استفاده می‌کنم.

۱۰	۱۰	۱۰	۱۰	۲۵۰	۲۵۰	۲۵۰	۲۵۰
۱۰	۱۰	۱۰	۱۰	۲۵۰	۲۵۰	۲۵۰	۲۵۰
۱۰	۱۰	۱۰	۱۰	۲۵۰	۲۵۰	۲۵۰	۲۵۰
۱۰	۱۰	۱۰	۱۰	۲۵۰	۲۵۰	۲۵۰	۲۵۰
۱۰	۱۰	۱۰	۱۰	۲۵۰	۲۵۰	۲۵۰	۲۵۰
۱۰	۱۰	۱۰	۱۰	۲۵۰	۲۵۰	۲۵۰	۲۵۰
۱۰	۱۰	۱۰	۱۰	۲۵۰	۲۵۰	۲۵۰	۲۵۰

در حالت یکنواخت مستقل از چرخش، ۹ کد از ۰ تا ۸ داریم، که حالت‌ها در اسلاید، صفحه ۱۶ آمده است.

فرض: سفید را ۱ در نظر گرفتم و کلاس‌های درون اسلاید را از بالا به پایین شماره‌گذاری کردم.

برای مثال برای ۱۱۱۱۱۱۱ کد ۰ را گرفتم



سوال ۲-

برای این مسئله من از دو ویژگی solidity , compactness استفاده کردم.

تابع محاسبه solidity :

```
hull = cv2.convexHull(contour, False)
area = cv2.contourArea(contour)
convex_area = cv2.contourArea(hull)
output = area / convex_area
return output
```

با استفاده از تابع cv2.convexHull ابتدا کانتور را ورودی دادم و کانتور مربوط به convex hull را بدست آوردم، سپس با استفاده از تابع cv2.contourArea مساحت مربوط به خود کانتور و convex hull را بدست آوردم. با تقسیم این دو مقدار به مقدار solidity میرسیم.

تابع محاسبه compactness :

```
area = cv2.contourArea(contour)
perimeter = cv2.arcLength(contour, True)
output = (4 * math.pi * area) / (perimeter**2)
return output
```

با استفاده از توابع contourArea و arcLength مساحت و محیط کانتور را بدست آوردم و با استفاده از فرمول فشردگی که در اسلایدها بود، این مقدار را برای کانتور برگشت میدهم.

تابع محاسبه eccentricity :

```
ellipse = cv2.fitEllipse(contour)
major_axis_length = max(ellipse[1])
minor_axis_length = min(ellipse[1])
output = minor_axis_length / major_axis_length
return output
```

ابتدا با استفاده از تابع fitEllipse یک بیضی به دور کانتور کشیده و قطر بزرگ و کوچک آن را گرفته و بر هم تقسیم میکنیم تا به مقدار کشیدگی کانتور دست یابیم.

البته از این تابع استفاده‌ای نشد و با همان دو ویژگی قبلی، دسته های مختلف اشکال بدست آمد.

تابع distance_criteria :

```
output = abs(LA.norm(x) - LA.norm(y))
```

برای این تابع از تابع آماده norm در numpy استفاده کردم. فرمول محاسبه آن به شرح زیر است:

$$||A||_F = [\sum_{i,j} abs(a_{i,j})^2]^{1/2}$$

The nuclear norm is the sum of the singular values.

ایجاد آرایه features برای پاس دادن به تابع grouping:

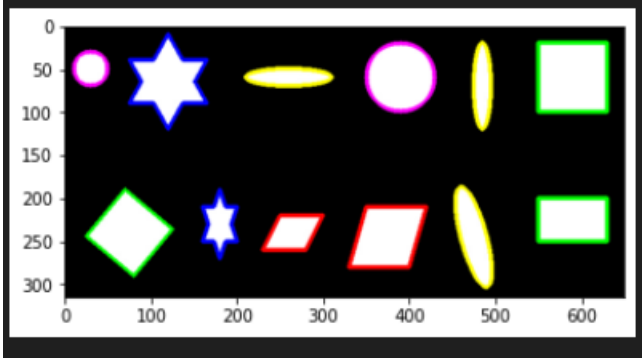
```
descriptors = [solidity, compactness]
features = np.zeros((len(contours), len(descriptors)))
for idx, contour in enumerate(contours):
    for des_idx, descriptor in enumerate(descriptors):
        features[idx][des_idx] = descriptor(contour)
```

توابع استخراج ویژگی را در لیست descriptors قرار دادم، بدین شکل اگر تابع ویژگی جدیدی اضافه شود، تنها کافیست در این لیست اضافه شود و نیاز به تغییر در جای دیگری نیست.

خروجی نهایی:

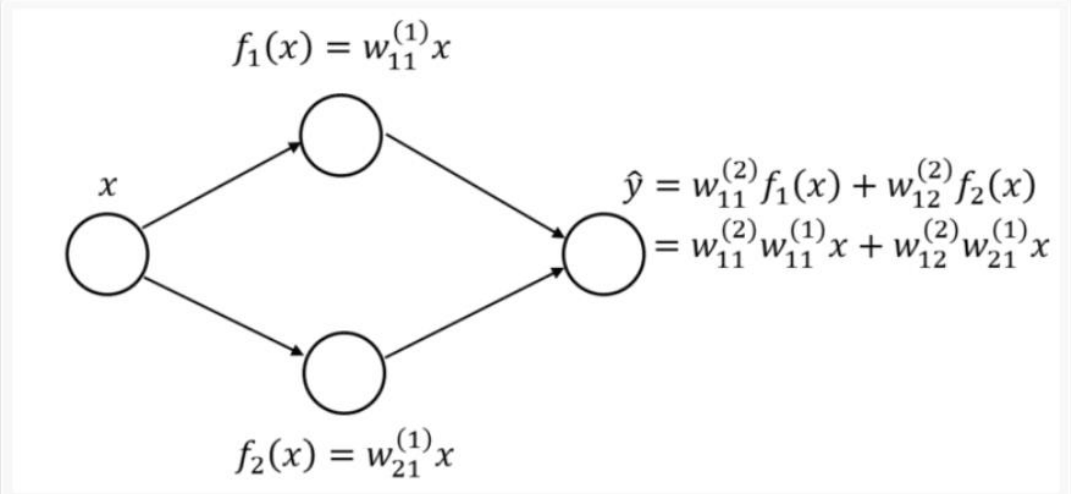
```
# You should replace your features array with None
colors = [(255,0,0),(0,255,0),(0,0,255),(255,255,0),(255,0,255)]
groups = grouping(features,0.042)
painting(groups, contours, colors)
```

✓ 0.3s



سوال ۳-

توابع فعالساز، خاصیت غیرخطی را به شبکه ما اضافه میکند و این باعث میشود بتواند الگوهای پیچیده‌تری را برای جداسازی کلاسهای مختلف پیدا کند. همچنین هدف ما در شبکه‌های عصبی استفاده از چندین لایه پشت سر هم است که بتوانند از ویژگی‌های سطح پایین به ویژگی‌های سطح بالا برسند، در حالیکه پشت هم قرار دادن چند لایه که توابع فعالساز غیرخطی ندارند، مشابه استفاده از تنها یک لایه خطی است و زیاد کردن تعداد لایه‌ها ما را به هدفمان نمی‌رساند.



Single hidden layer neural network with linear layers

همانطور که در این تصویر مشخص است، لایه آخر را میتوان به صورت مستقیم و با ضرایب نشان داده شده از لایه اول بدست آورد و نیازی به لایه میانی نبود.

پس برای اینکه شبکه توابع پیچیده‌تری را نشان دهد، به توابع فعال‌سازی غیرخطی نیاز داریم.

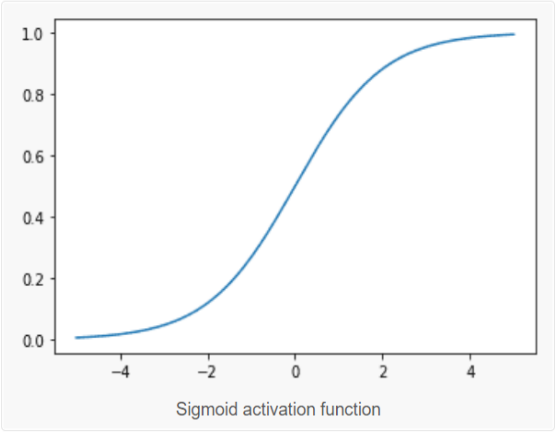
توابع فعالسازی:

Sigmoid Function:

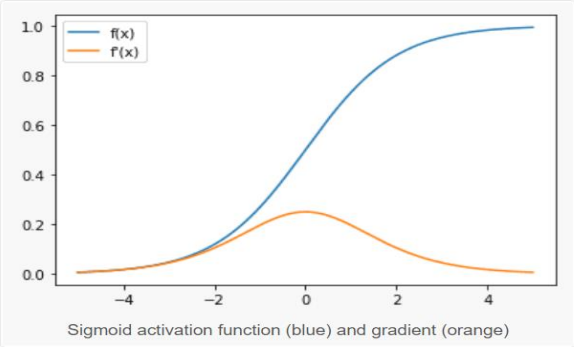
یکی از دلایلی که این تابع، یکی از توابع معروف فعالسازی است، این میباشد که مقادیر بین صفر و یک تولید میکند که میتواند نقش مقدار احتمال را بازی کند که این ویژگی مخصوصا برای مسائل دسته‌بندی پرکاربرد است.
فرمول آن به شکل زیر است:

$$\sigma = \frac{1}{1+e^{-1}}.$$

و نمودار آن به شکل زیر است:

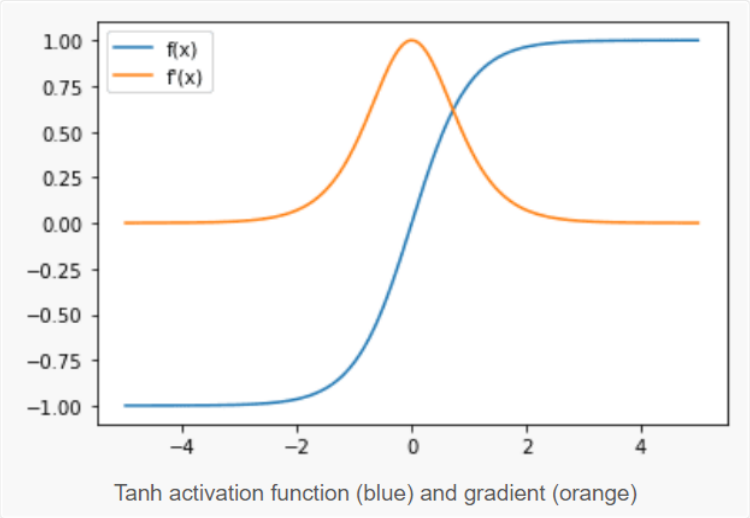


البته یکی از مشکلهای این تابع، تشدید vanishing gradient problem است، مشکلی که در شبکه‌های عمیق با لایه‌های زیاد ایجاد میشود و باعث میشود روند یادگیری لایه‌های ابتدایی بسیار کند شود و به نوعی یادگیری اتفاق نیفتد. علت اینکه تابع sigmoid این مشکل را تشدید میکند، این است که مشتق این تابع برای xهای بسیار کوچک و بسیار بزرگ به صفر میل میکند و برای باقی مقادیر حداکثر تا ۰/۲۵ میرود.



Hyperbolic Tangent Function:

این تابع محدوده مقادیر بیشتری را میتواند خروجی دهد، به طور مشخص مقادیر بین 1- و 1 . همچنین ماکزیمم مقدار مشتق این تابع نیز از تابع sigmoid بیشتر است.



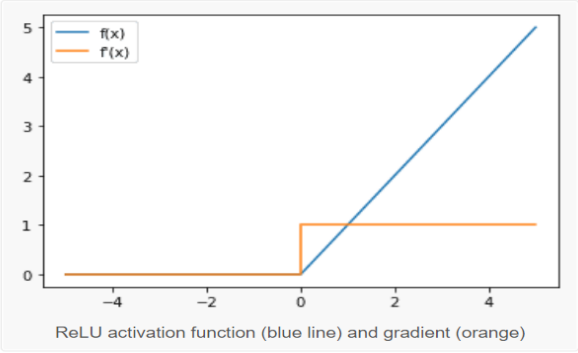
Tanh activation function (blue) and gradient (orange)

بیشتر بودن مقدار ماکزیمم گرادیان این تابع نسبت به تابع sigmoid باعث می‌شود شبکه عصبی ما کمتر مستعد vanishing gradient problem شود.

Rectified Linear Unit (ReLU):

این تابع فعالسازی که اخیرا محبوب شده است، محاسبه ساده‌ای دارد و همین محاسبه‌ی ساده باعث میشود performance شبکه بالا برود. این تابع مقادیر منفی را به صفر و مقادیر مثبت را به خودشان map میکند. فرمول آن به صورت زیر است:

$$\max(0, x)$$



ReLU activation function (blue line) and gradient (orange)

همانطور که در نمودار مشخص است، مقدار گرادیان این تابع برای مقادیر منفی، صفر و برای مقادیر مثبت، یک است که این vanishing gradient problem را حل میکند اما در عین حال باعث مشکل دیگری به اسم the dead neuron بشود و که در آن یک نورون به طور مداوم غیرفعال می‌شود و این مشکل زمانی ممکن است رخ دهد که ورودی ما منفی و در نتیجه گرادیان آن صفر باشد. در این حالت نورون هرگز نمیتواند یادگیری داشته باشد چون یکی از termها در chain rule مقدار صفر دارد و در نتیجه وزنهای این نورون آپدیت نمی‌شوند.

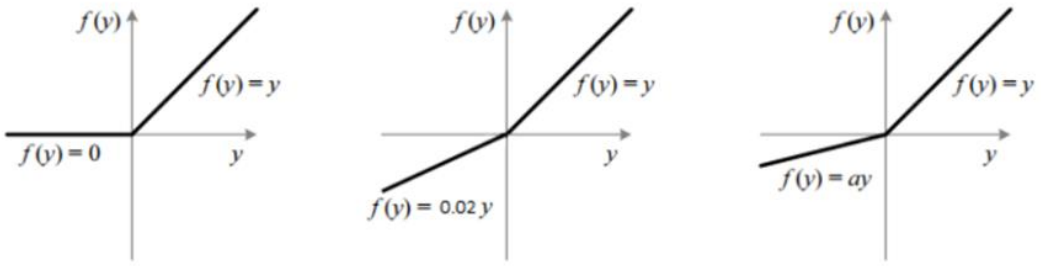
همانطور که در هرسه‌ی این توابع مشخص است، مقادیر صعودی هستند و این برای اینکه الگوریتم gradient descent به درستی عمل کند، لازم است.

- مقایسه‌ی بین این توابع را در توضیح هرکدام آوردم.

Parametric ReLU (PReLU):

مشکلی که تابع ReLU داشت، این بود که مقادیر منفی را به طور کلی به صفر تبدیل میکرد و این باعث مشکلی مثل dead neuron میشد، برای حل این مشکل آمدند Leaky ReLU را پیشنهاد دادند که در آن مقادیر منفی در یک عدد بسیار کوچک مثل ۰/۰۲ ضرب میشوند و نمودار این تابع در مقادیر منفی شیب بسیار کمی دارد، اما درعمل بهبود زیادی دیده نشد. سپس آمدند، این شیب را در قسمت منفی نمودار، به یک پارامتر قابل یادگیری تبدیل کردند و به PReLU رسیدند. در PReLU هر لایه میتواند شیب متفاوتی را براساس چیزی که یاد گرفته، اعمال کند. به طور خلاصه، اگر شیب نمودار برای مقادیر منفی را a_i بگیریم، حالات زیر را داریم:(این قسمت از یک مقاله در اینترنت برداشته شده است)

- if a_i=0, f becomes ReLU
- if a_i>0, f becomes leaky ReLU
- if a_i is a learnable parameter, f becomes PReLU



(Left) ReLU, (Middle) LeakyReLU and (Last) PReLU

سوال ۴-

برای لود کردن دیتا از دستور زیر استفاده میکنیم.

```
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
```

برای اینکه شبکه عملکرد بهتری داشته باشد، باید یک سری preprocess بر روی داده انجام شود:

```
x_train = x_train.reshape(60000, 784).astype("float32") / 255
x_test = x_test.reshape(10000, 784).astype("float32") / 255

y_train = to_categorical(y_train, num_classes=10)
y_test = to_categorical(y_test, num_classes=10)
```

در اینجا ابتدا داده آموزشی را reshape کردم زیرا ورودی لایه Dense که میخواهیم استفاده کنیم، باید دارای یک ستون باشد. همچنین مقادیر ۰ تا ۲۵۵ را بین ۰ تا ۱ آوردم و تایپ مقادیر هم به تبع باید float شود. همچنین از تابع to_categorical برای تبدیل فرمت label به one hot استفاده کردم که بتوان از تابع categorical_crossentropy برای loss استفاده کرد.

ایجاد یک شبکه عصبی با استفاده از functional api در keras به ما قابلیت flexibility بیشتری میدهد. در این نوع ساختار، هر لایه را میتوان به صورت مستقل تعریف کرد و آن را به عنوان ورودی به یک لایه دیگر داد(جزئیات فواید functional api در سوال ۵ توضیح داده شده است.)

```
inputs = keras.Input(shape=(None, 784))
layer1 = Dense(64, activation='relu')(inputs)
layer2 = Dense(64, activation='relu')(layer1)
outputs = Dense(10, activation='softmax')(layer2)
model = keras.Model(inputs=inputs, outputs=outputs)
```

همانطور که در تصویر مشخص است، ابتدا یک لایه Input با سایز ۷۸۴ که همان ۲۸ * ۲۸ است تعریف کردیم، و این لایه را به عنوان ورودی به یک لایه Dense دادیم. سپس layer1 را که یک لایه Dense است را به عنوان ورودی به یک لایه Dense دیگر دادیم. در نهایت این لایه را نیز به عنوان ورودی به لایه آخر که outputs باشد، دادیم. حال برای ساخت مدل لازم است از keras.Model استفاده کرده، ورودی و خروجی را مشخص کنیم. لایه‌های hidden به لایه outputs وصل هستند.

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

برای کامپایل هم از بهینه ساز adam و برای تابع ضرر از categorical_crossentropy استفاده کردم.

```
model.fit(x_train, y_train, batch_size=64, epochs=30, validation_split=0.2)
```

در اینجا گفته‌ایم تا ۳۰ epoch به آموزش ادامه دهد و ۲۰ درصد داده ها را برای validation استفاده کند. سایز هر batch که در مموری لود میشود را هم ۶۴ گرفتیم.

خروجی بعد از ۳۰ epoch:

```
Epoch 29/30
750/750 [==============================] - 2s 3ms/step - loss: 0.0094 - accuracy: 0.9969 - val_loss: 0.1611 - val_accuracy: 0.9728
```

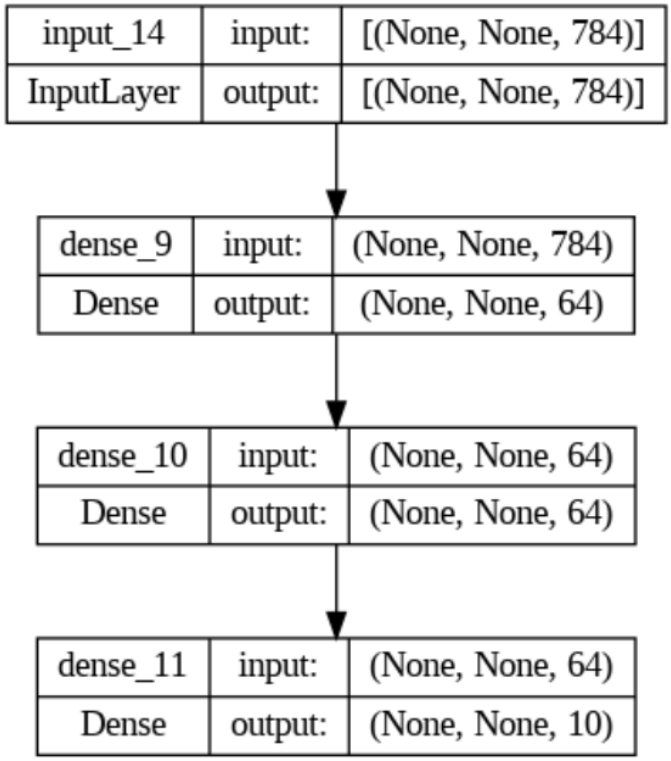
همچنین اگر با ورودی تست ارزیابی کنیم، داریم:

```
test_loss, test_acc = model.evaluate(x_test,  y_test, verbose=2)

print('\nTest accuracy:', test_acc)
```

```
313/313 - 1s - loss: 0.1555 - accuracy: 0.9694 - 600ms/epoch - 2ms/step

Test accuracy: 0.9693999886512756
```

دقت ۹۶ درصد دارد.

```
from keras.utils.vis_utils import plot_model
plot_model(model, to_file='model_plot.png', show_shapes=True, show_layer_names=True)
```



با استفاده از تابع plot_model دیاگرام شبکه را رسم کردم.

برای حالت Sequential از آنجا که در نوتبوک یک نمونه انجام شده و توضیحات داده شده است، دیگر توضیح اضافه‌ای نمیدهم.

برای حالت Sequential بعد از ۳۰ epoch به دقت زیر میرسیم(اینکه از functional استفاده کنیم یا sequential ، در دقت نهایی تاثیری ندارد)

```
750/750 [==============================] - 3s 4ms/step - loss: 0.0039 - accuracy: 0.9986 - val_loss: 0.1712 - val_accuracy: 0.9735
<keras.callbacks.History at 0x7f848cb478b0>
```

ارزیابی داده تست هم به صورت زیر خروجی میدهد:

```
313/313 - 1s - loss: 0.1579 - accuracy: 0.9731 - 609ms/epoch - 2ms/step

Test accuracy: 0.9731000065803528
```

سوال ۵-

خیر این امکان وجود ندارد.

تفاوت اصلی بین این دو API این است که Functional API اجازه می دهد تا معماری های شبکه پیچیده تر و انعطاف پذیرتری داشته باشد، در حالی که API Sequential فقط پشته خطی لایه ها را پشتیبانی می کند.

Functional API به شما امکان می دهد مدل هایی با ورودی یا خروجی های متعدد، لایه های مشترک و الگوهای اتصال لایه پیچیده تر ایجاد کنید. این نوع معماری ها را نمی توان به راحتی با استفاده از Sequential API پیاده سازی کرد.

به عنوان مثال، شبکه ای با چندین ورودی را در نظر بگیرید، که در آن هر ورودی قبل از ترکیب شدن در لایه بعدی، جداگانه پردازش می شود. این نوع معماری را می توان به راحتی با استفاده از Functional API پیاده سازی کرد، اما برای پیاده سازی با استفاده از Sequential API به راه حل های پیچیده تری نیاز دارد.

سوال ۶-

الف) تصویر حاصل ۱ در ۱ در تعداد فیلترها خواهد بود که اینجا فقط یک فیلتر گفته شده است.

ب) خروجی باز هم ۱ در ۱ خواهد بود. (1,1) => (3,3) => (5,5) => (7,7)

ج) از نظر عمیق یا سطحی تر بودن ویژگی‌ها، در حالت دوم که در چند مرحله کانوالو انجام میشود، به ویژگی‌های عمیقتری میرسیم و با استفاده از این ویژگیهای عمیق‌تر بهتر میتوان مسئله دسته‌بندی را انجام داد، فرض کنید میخواهیم کلاس یک حیوان را مشخص کنیم، صرفا با چند ویژگی سطحی مثل لبه‌ها و گرادیان نمیتوان به طور دقیق مشخص کرد که این حیوان جزو چه کلاسی است اما مثلا اگر ویژگی‌هایی مثل داشتن خرطوم مشخص میکند که این حیوان فیل است یا مثالهایی مانند این.

از لحاظ خطی‌تر بودن یا غیرخطی‌تر بودن، حالت دوم غیرخطی‌تر است، زیرا در سه مرحله و در سه لایه کانوالو انجام شده پس حداکثر ۳ بار تابع غیرخطی روی آن زده شده است. در حالت اول حداکثر ۱ بار تابع غیرخطی اعمال شده است، پس حالت اول میتواند الگوهای پیچیده‌تری استخراج کند و موثرتر است.

تعداد پارامترها: در مورد الف تعداد پارامترها ۱۴۸ تاست. ۴۹ تا برای خانه‌های کرنل و از آنجا که تعداد کانالها سه تاست پس در واقع ۱۴۷ = ۴۹ * ۳ تاست و یکی هم برای bias نهایی که کلا میشود ۱۴۸ پارامتر قابل آموزش.

برای مورد ب تعداد پارامترها برابر است با ۴۸ تا. ۲۸ تا برای کانوالو لایه اول، ۱۰ تا برای لایه دوم و ۱۰ تا هم برای لایه سوم.