

-۱

برای حل مسائل دو رویکرد وجود دارد، رویکرد توصیفی و رویکرد پیش بینی.  
در رویکرد توصیفی مدلی ریاضی جامع از مسئله ساخته میشود که مسئله مورد نظر، پارامتر های آن و روابط بین آن ها را توصیف میکند.

در مقابل روش پیش بینی قرار دارد که در آن یک مدل پیش بینی کننده تعریف میشود و سعی میشود خطای بین پیش بینی و داده واقعی کاهش یابد.

در ماشین لرنینگ از رویکرد دوم استفاده میشود به این صورت که داده های زیادی از مسئله به مدل عرضه میشود و سعی میشود مدل به صورتی اصلاح شود که خطای پیش بینی آن در مقایسه با جواب واقعی کاهش یابد. این روش سعی میکند پارامتر های مسئله و روابط بین آن ها را خودش یاد بگیرد.

یادگیری عمیق زیر مجموعه یادگیری ماشین است، این روش بر عملکرد واحد های محاسباتی کوچکی به نام نورون متکی است، نورون های مصنوعی از نورون های مغزی الهام گرفته شده اند، در مغز انسان تعداد بسیار زیادی از این نورون ها به یکدیگر متصل هستند، هر کدام محاسبات کوچکی انجام می دهند و با یکدیگر ارتباط برقرار میکنند تا در نهایت تصمیم گیری کنند، شبکه ها عصبی مصنوعی نیز عملکرد مشابهی البته در سطح پایین تر دارند.

یادگیری عمیق بر اساس ارتباط تعداد زیادی لایه های یک شبکه عصبی کار میکند و سعی میکند بدون نیاز به ناظر مدل مسئله را درک کند و براساس آن پیش بینی انجام دهد. با پیشرفت هایی که در زمینه سخت افزار روی داده است استفاده از شبکه های یادگیری عمیق مرسم تر شده است زیرا توانایی زیادی در مدل کردن مسائل پیچیده دارد.

(ب)

هر کدام از روش های کلاسیک فقط در حوزه ای که برایش طراحی شده اند کارایی دارند. اما شبکه های عصبی قابلیت این را دارند که با یک ساختار و پیکربندی برای مسائل مختلف آموزش داده شوند و خروجی های مطلوب آن مسئله را تولید کنند.

در روش های کلاسیک نیاز است که یک متخصص ویژگی ها را از داده ها استخراج کند و از میان آن ها ویژگی های مهم تر را برای مسئله انتخاب کند اما در شبکه های عصبی، خود شبکه ویژگی های مورد نیاز را از داده ها استخراج میکنند و آن ها را یادمیگیرند البته در بعضی مواقع دخالت انسان هم لازم است.

روش های سنتی را بعضا میتوان حتی روی یک میکروکنترلر پیاده سازی کرد اما روش های یادگیری عمیق نیاز سخت افزاری بالاتری دارند و دلیل مرسم شدن آن ها در سال های اخیر، رشد چشمگیر در حوزه سخت افزار است.

برای آموزش روش های یادگیری عمیق نیاز بالایی به جمع آوری داده است، این در حالی است که روش های کلاسیک نیاز زیادی به داده های آموزشی ندارند.

در روش های مبتنی بر شبکه های عصبی ریسک overfitting وجود دارد، در این حالت شبکه داده های آموزشی را بیش از حد یادمیگیرد و قابلیت تامیم خود را از دست میدهد؛ در حالی که در روش های کلاسیک به دلیل وابسته نبودن به داده های آموزشی، این مشکل وجود ندارد.

در روش های یادگیری عمیق، عملکرد شبکه مانند یک جعبه سیاه است و از دید کاربر مخفی است، اما در روش های کلاسیک عملکرد مدل قابل درک است و کاربر میتواند در صورت نیاز آن را اصلاح کند تا به خروجی مطلوب برسد.

برخی از مسائل پیچیده با روش های مبتنی بر مشتق (مانند شبکه های عصبی) قابل حل نیستند و برای حل آن ها باید از روش های کلاسیک استفاده کرد، برای مثال میتوان به video processing, scene understanding, motion capture, video stabilization, robotics, ... اشاره کرد.

ج) علاوه بر مشکلاتی که برای آموزش شبکه های عصبی و نیاز میرم آن ها به داده و محاسبات زیاد وجود دارد، یادگیری عمیق در حوزه بینایی ماشین با مشکلات دیگری نیز مواجه است از جمله مشکل و چالش در شناسایی الگوهای بصری (برای مثال تشخیص این که چند شی در یک تصویر متفاوت هستند یا یکسان) و یا ناتوان بودن در استخراج همه روابط بین داده ها (برای مثال در مدل سازی سه بعدی لازم است اطلاعات اولیه ای درباره روشنایی، صافی و تاری را به مدل بدهیم).

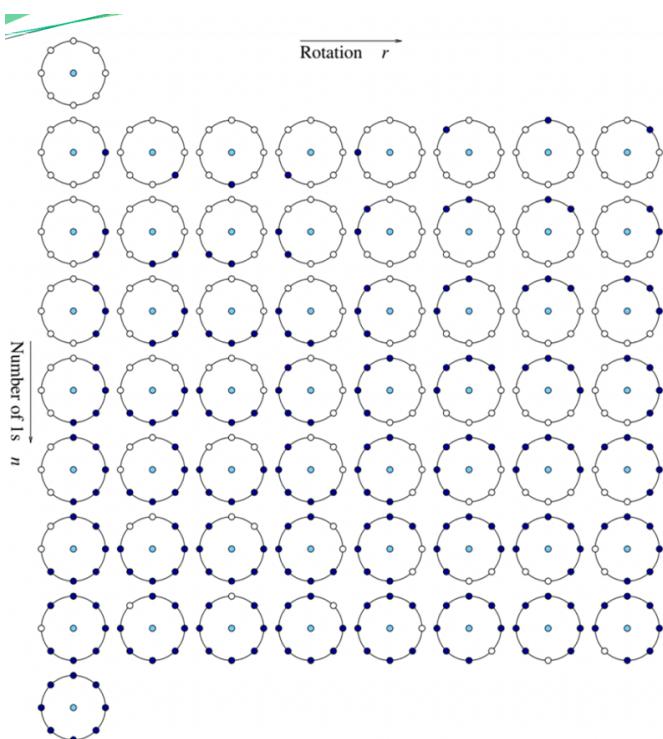
-۲  
الف)

خیر زیرا در صورتی که پد بزنیم درواقع داریم الگوهای جدید به تصویر اضافه میکنیم.

(ب)

کد LBP برای هر پیکسل از مقایسه مقدار آن پیکسل با مقدار پیکسل های همسایه به دست می آید، به این صورت که پیکسل های همسایه نسبت به پیکسل مرکزی اگر مقدار مساوی یا بیشتر داشتند با ۱ و اگر کوچکتر از آن بودند با ۰ نمایش داده می شوند.  
به این ترتیب ۲۵۶ کد مختلف خواهیم داشت. اما بسیاری از آن ها اهمیت کمی دارند، کد هایی که بیش از ۲ تغییر از ۰ به ۱ یا از ۱ به ۰ داشته باشند کد های non-uniform هستند که تعداد آن ها ۱۹۸ عدد است و ۵۸ عدد کد uniform داریم که تغییر از ۰ به ۱ یا برعکس در آن ها کمتر مساوی ۲ عدد است.  
در بسیاری از کاربردها چرخش الگوهای ما مهم نیست و میتوان الگوهایی که از چرخش هم به دست می آیند را با یک کد نمایش دهیم.

به این ترتیب ۹ کد برای الگوهای یکنواخت مستقل از چرخش داریم.  
هر ردیف را با یک کد از ۰ تا ۸ نشان می دهیم.



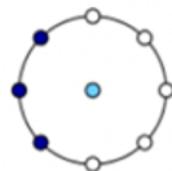
۱۰	۱۰	۱۰	۲۵۰	۲۵۰	۲۵۰
۱۰	۱۰	۱۰	۲۵۰	۲۵۰	۲۵۰
۱۰	۱۰	۱۰	۲۵۰	۲۵۰	۲۵۰
۱۰	۱۰	۱۰	۲۵۰	۲۵۰	۲۵۰
۱۰	۱۰	۱۰	۲۵۰	۲۵۰	۲۵۰

برای نمونه یکی از خانه ها را تحلیل میکنیم:

		less	equal	equal	
		less	۲۵۰	equal	
		less	equal	equal	

		.	۱	۱	
		.	۲۵۰	۱	
		.	۱	۱	

که الگوی شماره ۳ است:



حال بر این اساس اگر ماتریس مد نظر را کد گذاری کنیم به صورت زیر خواهد بود:

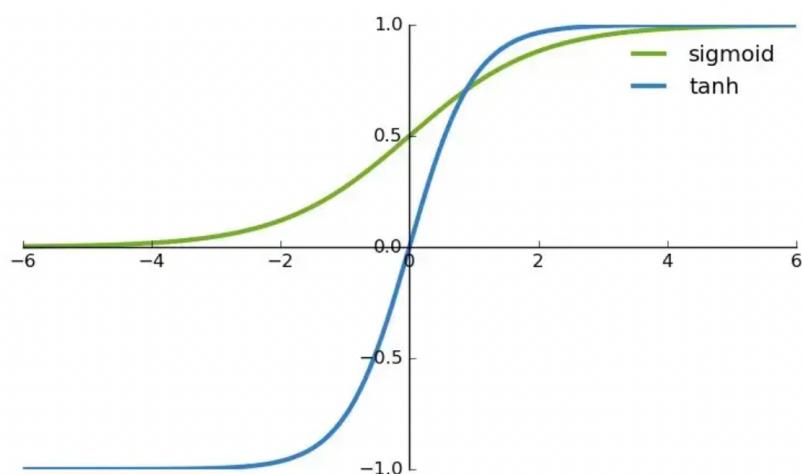
.	۵	۳	.
.	۵	۳	.
.	۵	۳	.

طبیعی است به دلیل پد ندادن، ابعاد ماتریس کاهش یابد.

عملکرد نورون های مغز انسان به این صورت است که هر نورون به چندین نورون دیگر متصل است و از آن ها سیگنال ورودی دریافت میکند، اگر مجموع این سیگنال های ورودی از حدی بیشتر شود، نورون به آن واکنش نشان می دهد و بر اساس تابعی که آن را تابع فعال ساز می نامیم یک سیگنال خروجی تولید میکند.

تابع فعال ساز می توانند خطی یا غیر خطی باشند، توابع غیر خطی برای ما اولویت دارند زیرا روابط بین داده ها در دنیای واقعی اغلب غیر خطی است و اگر از توابع خطی استفاده کنیم شبکه عصبی ما توانایی مدل کردن روابط پیچیده را ندارد و عملکرد آن شبیه یک پرسپترون خطی ساده است.

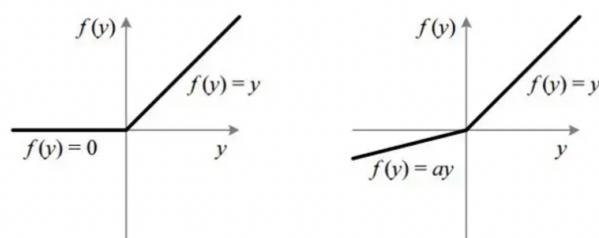
تابع sigmoid جزو اولین توابع فعال سازی است که مرسوم شد. خروجی آن بین  $-1$  و  $1$  است، مشکلات آن این است که به دلیل اینکه همه مقادیر آن مثبت است، همگرای سخت میشود به همین خاطر تابع  $\tanh$  پیشنهاد شد که بسیار شبیه به sigmoid است ولی مقادیر آن بین  $-1$  و  $1$  قرار دارد و این خاصیت به بهینه سازی کمک میکند.



ایراد دیگری که میتوان به توابع بالا گرفت این است که پیاده سازی آن ها به لحاظ سخت افزاری مشکل است.

تابع Relu تقریبا جای توابع قبلی را گرفت زیرا نسبت به توابع قبلی خیلی راحت تر بهینه میشود، دلیل آن این است که در توابع sigmoid و tanh در جاهای زیادی مشتق صفر است و روش های مبتنی بر مشتق، دچار مشکل میشوند و نمیتوانند نقطه بهینه را به راحتی پیدا کنند، اما مشتق تابع Relu در ناحیه مثبت همیشه ۱ است.

البته چون در ناحیه منفی همه مقادیر را صفر میکند مشکلاتی ایجاد می شود که با نسخه های Leaky Relu این مشکلات کمتر میشود.



ReLU vs Leaky ReLU

مزیت فرمول های خطی این است که خیلی راحت بهینه میشوند اما عیب آن ها این است که توانایی یادگیری زیادی ندارند و نمی توانند توابع پیچیده را یاد بگیرند؛ پس از توابع غیر خطی استفاده می کنیم اما اگر تا حد امکان به توابع خطی نزدیک تر باشند، پاسخ بهینه را راحت تر پیدا میکنند.

-۴

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Input, Dense, Flatten, Softmax
```

کتابخانه های لازم را وارد میکنیم.

```
(x_train_digit, y_train_digit), (x_test_digit, y_test_digit) = tf.keras.datasets.mnist.load_data()
```

از مجموعه **mnist**، داده های آموزشی و تست را وارد میکنیم.

```
class_names_digit = ['0', '1', '2', '3', '4', '5',
                     '6', '7', '8', '9']
```

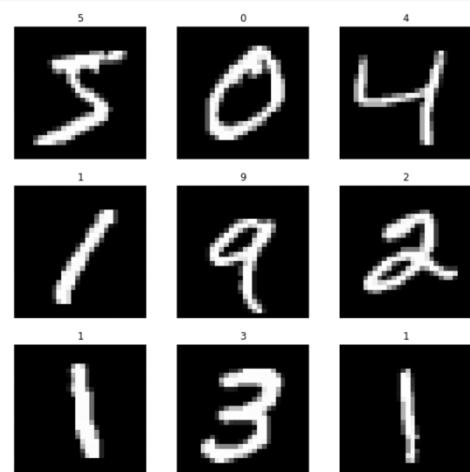
در این مجموعه داده، ده کلاس عددی از ۰ تا ۹ داریم، این مجموعه را نام گذاری میکنیم.

```
print(f'x_train : {x_train_digit.shape}')
print(f'y_train : {y_train_digit.shape}')
print(f'x_test : {x_test_digit.shape}')
print(f'y_test : {y_test_digit.shape}')
```

```
x_train : (60000, 28, 28)
y_train : (60000,)
x_test : (10000, 28, 28)
y_test : (10000,)
```

حال میتوانیم ابعاد این داده ها را مشاهده کنیم، همانگونه که مشخص است تعداد ۶۰۰۰ عدد داده آموزش به صورت عکس های با سایز ۲۸ در ۲۸ و لیبل های مربوط به آنها و تعداد ۱۰۰۰۰ داده تست داریم.

```
plt.figure(figsize=(10, 10))
for i in range(9):
    ax = plt.subplot(3, 3, i + 1)
    plt.imshow(x_train_digit[i].astype("uint8"), cmap='gray')
    plt.title(class_names_digit[int(y_train_digit[i])])
    plt.axis("off")
```



میتوانیم تعدادی از این داده ها را مشاهده کنیم.

```
x_train_digit = x_train_digit/ 255.0
x_test_digit = x_test_digit / 255.0
```

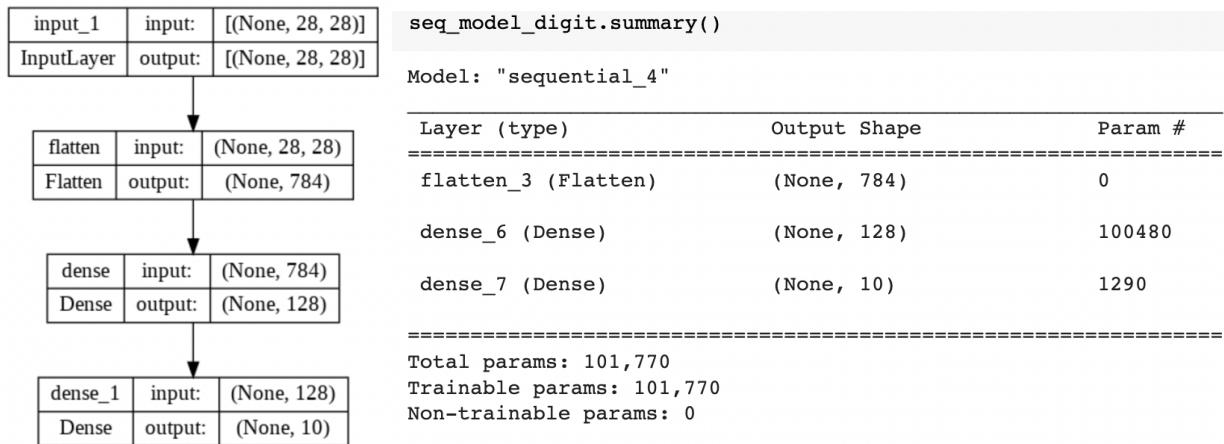
برای عملکرد بهتر شبکه عصبی ابتدا داده ها را نرمال میکنیم.

رویکرد ترتیبی (sequential)  
در این رویکرد، لایه ها به ترتیب پشت سر هم تعریف میشوند

```
seq_model_digit = Sequential()
seq_model_digit.add(Input(shape= (28,28)))
seq_model_digit.add(Flatten())
seq_model_digit.add(Dense(units=128, activation='relu'))
seq_model_digit.add(Dense(units=10))
```

در اینجا ابتدا لایه ورودی تعریف شده که تصاویر با سایز ۲۸ در ۲۸ را دریافت میکند سپس در لایه بعد (flatten) این ورودی های دو بعدی به صورت آرایه های یک بعدی با سایز ۷۸۴ در می آیند. لایه بعد یک لایه سراسر متصل (fully connected) است که ۱۲۸ نورون دارد، تابع فعال ساز این لایه ReLU در نظر گرفته شده است.

در نهایت لایه آخر نیز یک لایه fully connected است که به تعداد کلاس های مسئله است.



```
seq_model_digit.compile(optimizer='adam',
                        loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                        metrics=[ 'accuracy'])
```

در این مرحله روش بهینه سازی مورد استفاده در مدل، تابع محاسبه خطا، و معیار های ارزیابی را برای مدل تعریف میکنیم.

: این تابع میزان دقیق بودن مدل در طول آموزش را تعیین میکند، هدف شبکه کمینه کردن این تابع است تا مدل به وضعیت درست هدایت شود.

: روش به روز رسانی مدل در هر مرحله از آموزش بر اساس مقدار خطا را مشخص میکند. Optimizer

: معیار ارزیابی مدل است که در هر مرحله میتوان عملکرد مدل را با آن سنجید. Metrics

```

seq_history_digit = seq_model_digit.fit(x_train_digit, y_train_digit, epochs=10)

Epoch 1/10
1875/1875 [=====] - 13s 7ms/step - loss: 0.2593 - accuracy: 0.9264
Epoch 2/10
1875/1875 [=====] - 11s 6ms/step - loss: 0.1136 - accuracy: 0.9671
Epoch 3/10
1875/1875 [=====] - 6s 3ms/step - loss: 0.0773 - accuracy: 0.9768
Epoch 4/10
1875/1875 [=====] - 6s 3ms/step - loss: 0.0577 - accuracy: 0.9828
Epoch 5/10
1875/1875 [=====] - 5s 3ms/step - loss: 0.0457 - accuracy: 0.9855
Epoch 6/10
1875/1875 [=====] - 5s 3ms/step - loss: 0.0351 - accuracy: 0.9893
Epoch 7/10
1875/1875 [=====] - 6s 3ms/step - loss: 0.0276 - accuracy: 0.9912
Epoch 8/10
1875/1875 [=====] - 5s 3ms/step - loss: 0.0231 - accuracy: 0.9924
Epoch 9/10
1875/1875 [=====] - 5s 3ms/step - loss: 0.0188 - accuracy: 0.9942
Epoch 10/10
1875/1875 [=====] - 5s 3ms/step - loss: 0.0147 - accuracy: 0.9955

```

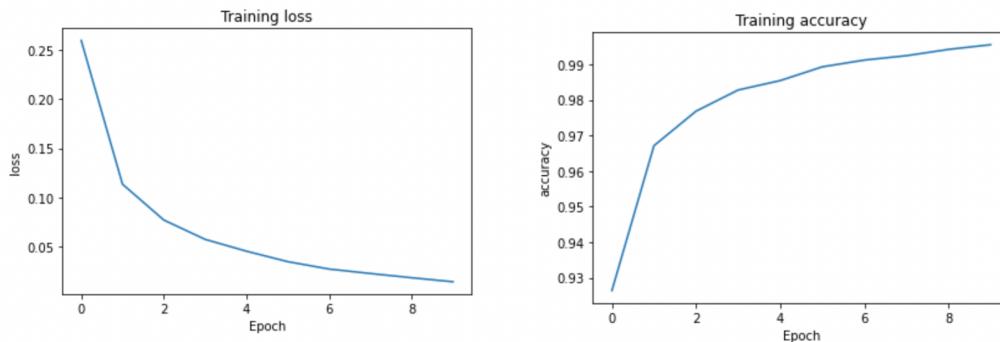
حال شبکه را با ۱۰ بار تکرار، آموزش میدهیم.

```

plt.plot(seq_history_digit.history['loss'])
plt.title('Training loss')
plt.xlabel('Epoch')
plt.ylabel('loss')
plt.show()

plt.plot(seq_history_digit.history['accuracy'])
plt.title('Training accuracy')
plt.xlabel('Epoch')
plt.ylabel('accuracy')
plt.show()

```



با رسم نمودار خطأ و دقت نسبت به تعداد تکرار آموزش، مشاهده میکنیم که در هر مرحله آموزش، روی داده های آموزشی، خطأ کاهش و دقت افزایش یافته است و شبکه توانسته روی داده های آموزشی به دقت ۹۹.۵٪ برسد اما این کافی نیست، ممکن است شبکه overfit شده باشد و باید دقت شبکه روی داده های تست که شبکه قبل آن ها را ندیده هم سنجیده شود تا قابلیت تعمیم شبکه نیز بررسی شود.

```

test_loss_digit, test_acc_digit = seq_model_digit.evaluate(x_test_digit, y_test_digit, verbose=2)

print('\nTest accuracy:', test_acc_digit)

313/313 - 1s - loss: 0.0847 - accuracy: 0.9784 - 610ms/epoch - 2ms/step

```

Test accuracy: 0.9783999919891357

با عرضه داده ها تست به شبکه مشاهده میکنیم که دقت شبکه روی داده های تست هم مناسب است و توانسته به دقت ۹۷.۸٪ برسد.

```

probability_model_digit = tf.keras.Sequential([seq_model_digit, Softmax()])

```

برای بررسی احتمال هر کلاس خروجی شبکه را به تابع softmax میدهیم. این تابع  $k$  عدد حقیقی را دریافت میکند و آن ها را تبدیل به توزیع احتمال  $k$  پیشامد میکند خروجی این تابع مقادیر بین ۰ و ۱ دارند که نشان دهنده احتمال آن کلاس است.

```

y_pred_digit = probability_model_digit.predict(x_test_digit)

313/313 [=====] - 1s 2ms/step

y_pred_digit[0]

array([2.1778571e-10, 2.6707208e-12, 5.0457707e-08, 4.4457931e-07,
       4.3673741e-16, 1.5116056e-09, 3.7145399e-16, 9.9999565e-01,
       1.3118400e-07, 3.6345753e-06], dtype=float32)

plt.imshow(x_test_digit[0], cmap='gray')
label = np.argmax(y_pred_digit[0])
print(f'label : {label} and class : {class_names_digit[label]}')

label : 7 and class : 7


```

یک داده تست را به شبکه میدهیم و شبکه ۱۰ مقدار به ما میدهد که هر کدام احتمال کلاس متناظرش است. با پیدا کردن اندیس المانی که بیشترین مقدار را دارد میتوان کلاس داده را مشخص کرد. همانگونه که مشاهده میشود در اینجا شبکه درست طبقه بندی کرده است.

### رویکرد تابعی (functional approach)

در این روش هر لایه مانند یک تابع تعریف میشود که ورودی و خروجی دارد، با توجه به این خاصیت این روش انعطاف بیشتری نسبت به روش ترتیبی دارد و امکانات بیشتری در اختیار ما میگذارد.

```

inputs = tf.keras.Input(shape= (28,28))
flatten_input = tf.keras.layers.Flatten()(inputs)
x = tf.keras.layers.Dense(128, activation="relu")(flatten_input)
outputs = tf.keras.layers.Dense(10)(x)
func_model_digid = tf.keras.Model(inputs=inputs, outputs=outputs, name="func_model_digid")

```

همانگونه که مشاهده می شود همان لایه های قبلی را این بار به صورت تابع تعریف کردیم، خروجی هر لایه را به لایه بعدی دادیم. و در آخر با شناساندن ورودی ها و خروجی ها مدل شبکه را تعریف کردیم.

```
func_model_digid.summary()
Model: "func_model_digid"
-----  

Layer (type)      Output Shape       Param #
-----  

input_1 (InputLayer) [(None, 28, 28)]    0  

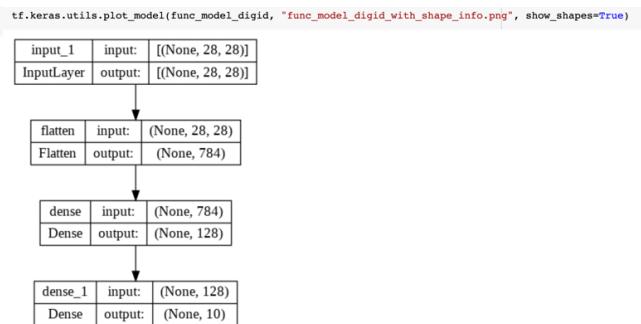
flatten (Flatten)  (None, 784)        0  

dense (Dense)     (None, 128)        100480  

dense_1 (Dense)   (None, 10)         1290  

-----  

Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
```



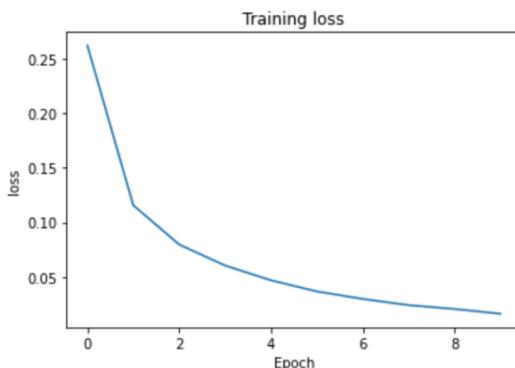
نحوه آموزش و تست شبکه مشابه قبل است.

```
func_model_digid.compile(optimizer='adam',
                         loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                         metrics=['accuracy'])
```

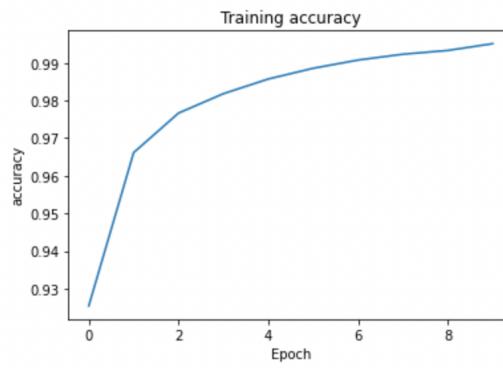
```
func_history_digit = func_model_digid.fit(x_train_digit, y_train_digit, epochs=10)
```

```
Epoch 1/10
1875/1875 [=====] - 6s 3ms/step - loss: 0.2620 - accuracy: 0.9255
Epoch 2/10
1875/1875 [=====] - 6s 3ms/step - loss: 0.1156 - accuracy: 0.9662
Epoch 3/10
1875/1875 [=====] - 7s 4ms/step - loss: 0.0796 - accuracy: 0.9767
Epoch 4/10
1875/1875 [=====] - 5s 3ms/step - loss: 0.0603 - accuracy: 0.9818
Epoch 5/10
1875/1875 [=====] - 5s 3ms/step - loss: 0.0468 - accuracy: 0.9857
Epoch 6/10
1875/1875 [=====] - 6s 3ms/step - loss: 0.0366 - accuracy: 0.9886
Epoch 7/10
1875/1875 [=====] - 5s 3ms/step - loss: 0.0297 - accuracy: 0.9908
Epoch 8/10
1875/1875 [=====] - 6s 3ms/step - loss: 0.0240 - accuracy: 0.9923
Epoch 9/10
1875/1875 [=====] - 5s 3ms/step - loss: 0.0205 - accuracy: 0.9933
Epoch 10/10
1875/1875 [=====] - 5s 3ms/step - loss: 0.0162 - accuracy: 0.9951
```

```
plt.plot(func_history_digit.history['loss'])
plt.title('Training loss')
plt.xlabel('Epoch')
plt.ylabel('loss')
plt.show()
```



```
plt.plot(func_history_digit.history['accuracy'])
plt.title('Training accuracy')
plt.xlabel('Epoch')
plt.ylabel('accuracy')
plt.show()
```



```
test_loss_digit, test_acc_digit = func_model_digid.evaluate(x_test_digit, y_test_digit, verbose=2)

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

```
313/313 - 1s - loss: 0.0887 - accuracy: 0.9783 - 593ms/epoch - 2ms/step
```

```
Test accuracy: 0.9782999753952026
```

```
probability_model_digit = tf.keras.Sequential([func_model_digid, Softmax()])
```

```

y_pred_digit = probability_model_digit.predict(x_test_digit)

313/313 [=====] - 1s 2ms/step

y_pred_digit[0]

array([5.7783400e-11, 1.6171436e-12, 3.0522371e-09, 2.9169671e-06,
       3.3115021e-14, 3.1228553e-09, 8.2140501e-16, 9.9999648e-01,
       1.4785075e-08, 5.4437083e-07], dtype=float32)

plt.imshow(x_test_digit[0], cmap='gray')
label = np.argmax(y_pred_digit[0])
print(f'label : {label} and class : {class_names_digit[label]}')

label : 7 and class : 7


```

-۵

در حالت **sequential**، لایه ها پشت سر هم قرار میگیرند و خروجی لایه های قبل ورودی لایه های بعدی میشود و روش دیگری برای ارتباط لایه ها وجود ندارد اما در روش **functional** دست ما باز تر است و میتوانیم به هر طریق که خواستیم لایه ها را به هم مرتبط کنیم. برای مثال شبکه های **ResNet** و **GoogleNet** قابل پیاده سازی نیستند.