Neural Networks on MNIST

In [1]:
```python
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from sklearn.metrics import classification_report
%matplotlib inline
```

In [2]:
```python
(X_train, y_train), (X_test, y_test) = keras.datasets.mnist.load_data()
```

In [3]:
```python
len(X_train)
```

Out[3]:
```
60000
```

In [4]:
```python
len(X_test)
```

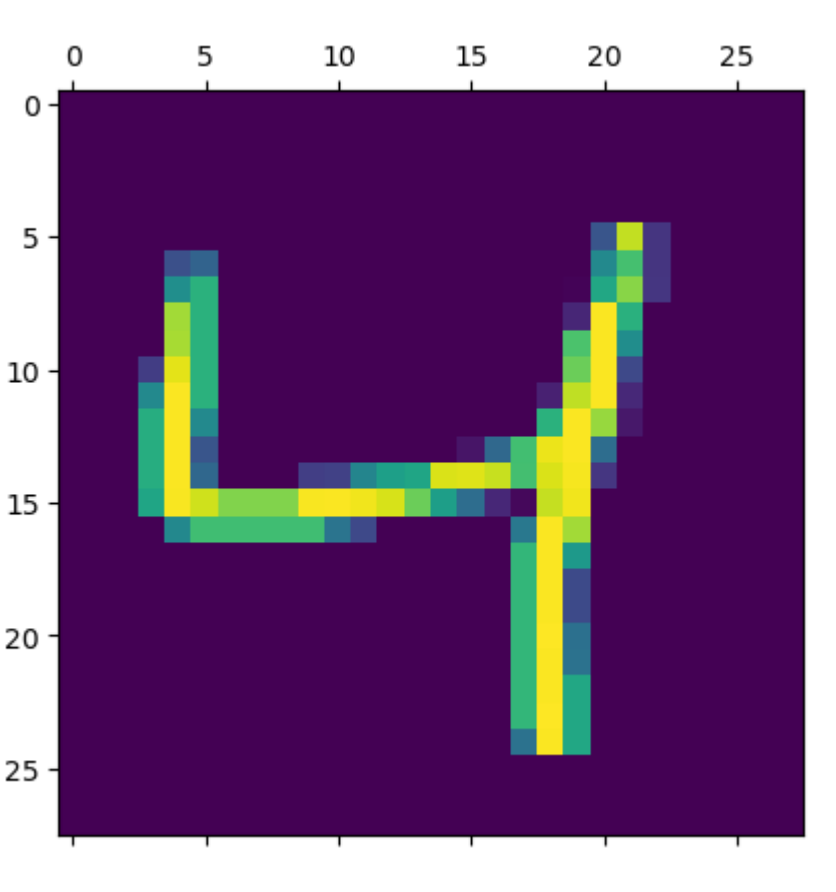
Out[4]:
```
10000
```

In [5]:
```python
X_train[0].shape
```

Out[5]:
```
(28, 28)
```

In [6]:
```python
plt.matshow(X_train[2])
```

Out[6]:
```
<matplotlib.image.AxesImage at 0x1e6f575aaf0>
```



In [7]:
```python
y_train[2]
```

Out[7]:
```
4
```

In [8]:
```python
X_train[0]
```

Out[8]:
```
array([[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   3,
         18,  18,  18, 126, 136, 175,  26, 166, 255, 247, 127,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,  30,  36,  94, 154, 170,
        253, 253, 253, 253, 253, 225, 172, 253, 242, 195,  64,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,  49, 238, 253, 253, 253, 253,
        253, 253, 253, 253, 251,  93,  82,  82,  56,  39,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,  18, 219, 253, 253, 253, 253,
        253, 198, 182, 247, 241,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,  80, 156, 107, 253, 253,
        205,  11,   0,  43, 154,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,  14,   1, 154, 253,
         90,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, 139, 253,
        190,   2,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  11, 190,
        253,  70,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  35,
        241, 225, 160, 108,   1,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         81, 240, 253, 253, 119,  25,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,  45, 186, 253, 253, 150,  27,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,  16,  93, 252, 253, 187,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0, 249, 253, 249,  64,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,  46, 130, 183, 253, 253, 207,   2,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  39,
        148, 229, 253, 253, 253, 250, 182,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,  24, 114, 221,
        253, 253, 253, 253, 201,  78,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,  23,  66, 213, 253, 253,
        253, 253, 198,  81,   2,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,  18, 171, 219, 253, 253, 253, 253,
        195,  80,   9,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,  55, 172, 226, 253, 253, 253, 253, 244, 133,
         11,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0, 136, 253, 253, 253, 212, 135, 132,  16,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0]], dtype=uint8)
```

In [9]:
```python
X_train = X_train/255
X_test = X_test/255
```

In [ ]:

In [10]:
```python
X_train_flatten = X_train.reshape(len(X_train), 28*28)
X_test_flatten = X_test.reshape(len(X_test), 28*28)
```

In [11]:
```python
X_train_flatten[0].shape
```

Out[11]:
```
(784,)
```

In [12]:
```python
model = keras.Sequential(
    [keras.layers.Dense(100, input_shape=(784, ), activation='relu'),
     keras.layers.Dense(10, activation='sigmoid')]
)
```

In [13]:
```python
from keras.losses import sparse_categorical_crossentropy

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

In [14]:
```python
model.fit(X_train_flatten, y_train, epochs=10)
```

```
Epoch 1/10
1875/1875 [==============================] - 8s 3ms/step - loss: 0.2698 - accuracy: 0.9245
Epoch 2/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.1206 - accuracy: 0.9639
Epoch 3/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0852 - accuracy: 0.9740
Epoch 4/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0644 - accuracy: 0.9808
Epoch 5/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0524 - accuracy: 0.9841
Epoch 6/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0432 - accuracy: 0.9863
Epoch 7/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0346 - accuracy: 0.9891
Epoch 8/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0283 - accuracy: 0.9912
Epoch 9/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0242 - accuracy: 0.9927
Epoch 10/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0201 - accuracy: 0.9938
```

Out[14]:
```
<keras.callbacks.History at 0x1e6f5108760>
```

In [15]:
```python
model.evaluate(X_test_flatten, y_test)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.0779 - accuracy: 0.9786
```

Out[15]:
```
[0.07791927456855774, 0.978600025177002]
```

In [16]:
```python
y_pred = model.predict(X_test_flatten)
```

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

In [17]:
```python
y_pred_labels = [np.argmax(i) for i in y_pred]
```

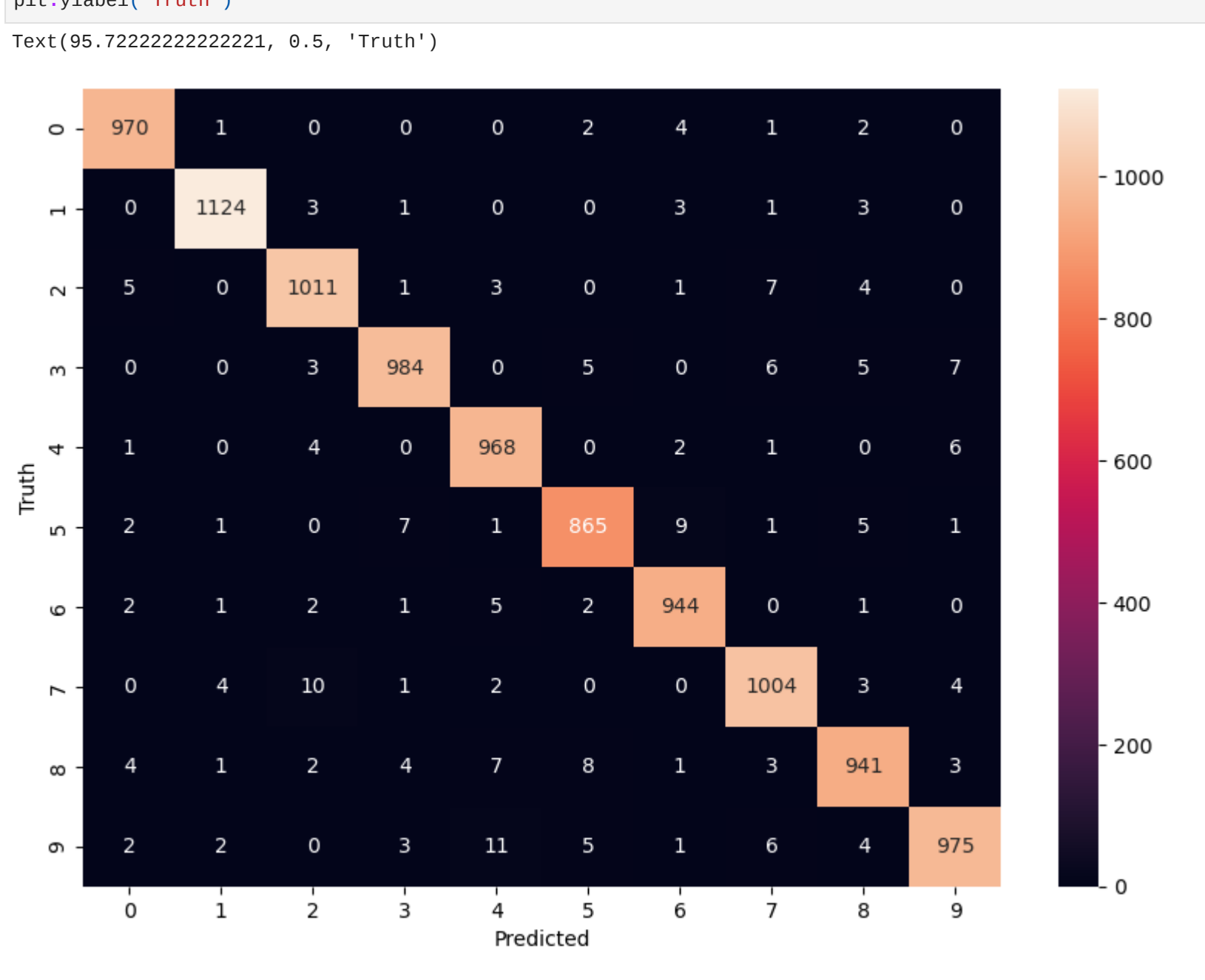
In [18]:
```python
cm = tf.math.confusion_matrix(labels=y_test, predictions=y_pred_labels)
```

In [19]:
```python
cm
```

Out[19]:
```
<tf.Tensor: shape=(10, 10), dtype=int32, numpy=
array([[ 970,    1,    0,    0,    0,    2,    4,    1,    2,    0],
       [   0, 1124,    3,    1,    0,    0,    3,    1,    3,    0],
       [   5,    0, 1011,    1,    3,    0,    1,    7,    4,    0],
       [   0,    0,    3,  984,    0,    5,    0,    6,    5,    7],
       [   1,    0,    4,    0,  968,    0,    2,    1,    0,    6],
       [   2,    1,    0,    7,    1,  865,    9,    1,    5,    1],
       [   2,    1,    2,    1,    5,    2,  944,    0,    1,    0],
       [   0,    4,   10,    1,    2,    0,    0, 1004,    3,    4],
       [   4,    1,    2,    4,    7,    8,    1,    3,  941,    3],
       [   2,    2,    0,    3,   11,    5,    1,    6,    4,  975]])>
```

In [20]:
```python
plt.figure(figsize=(10, 7))
sns.heatmap(cm, annot=True, fmt='d')
plt.xlabel('Predicted')
plt.ylabel('Truth')
```

Out[20]:
```
Text(95.72222222222221, 0.5, 'Truth')
```



In [21]:
```python
print(classification_report(y_test, y_pred_labels))
```

```
              precision    recall  f1-score   support

           0       0.98      0.99      0.99       980
           1       0.99      0.99      0.99      1135
           2       0.98      0.98      0.98      1032
           3       0.98      0.97      0.98      1010
           4       0.97      0.99      0.98       982
           5       0.98      0.97      0.97       892
           6       0.98      0.99      0.98       958
           7       0.97      0.98      0.98      1028
           8       0.97      0.97      0.97       974
           9       0.98      0.97      0.97      1009

    accuracy                           0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

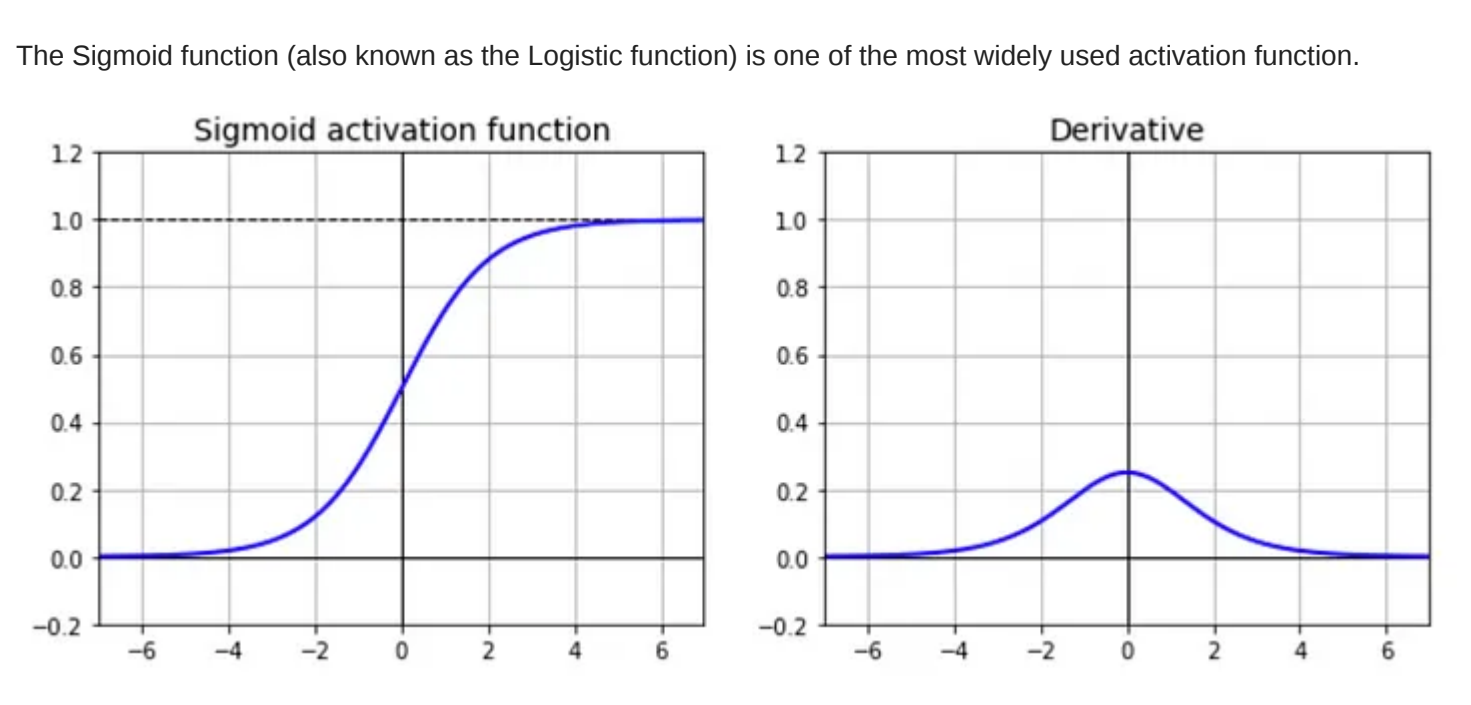
In [ ]:

Activation Function

Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

The Sigmoid function (also known as the Logistic function) is one of the most widely used activation function.



The output of the function is centered at 0.5 with a range from 0 to 1.

The main problems with the Sigmoid function are:

Vanishing gradient: looking at the function plot, you can see that when inputs become small or large, the function saturates at 0 or 1, with a derivative extremely close to 0. Thus it has almost no gradient to propagate back through the network, so there is almost nothing left for lower layers.
Computationally expensive: the function has an exponential operation.
The output is not zero centered

In [25]:
```python
model = keras.Sequential(
[
    keras.layers.Dense(100, input_shape=(784,), activation='sigmoid'),
    keras.layers.Dense(10, activation='sigmoid')
])
```

In [26]:
```python
model.compile(optimizer='adam',
             loss=sparse_categorical_crossentropy,
             metrics=['accuracy'])
```

In [28]:
```python
model.fit(X_train_flatten, y_train, epochs=5)
```

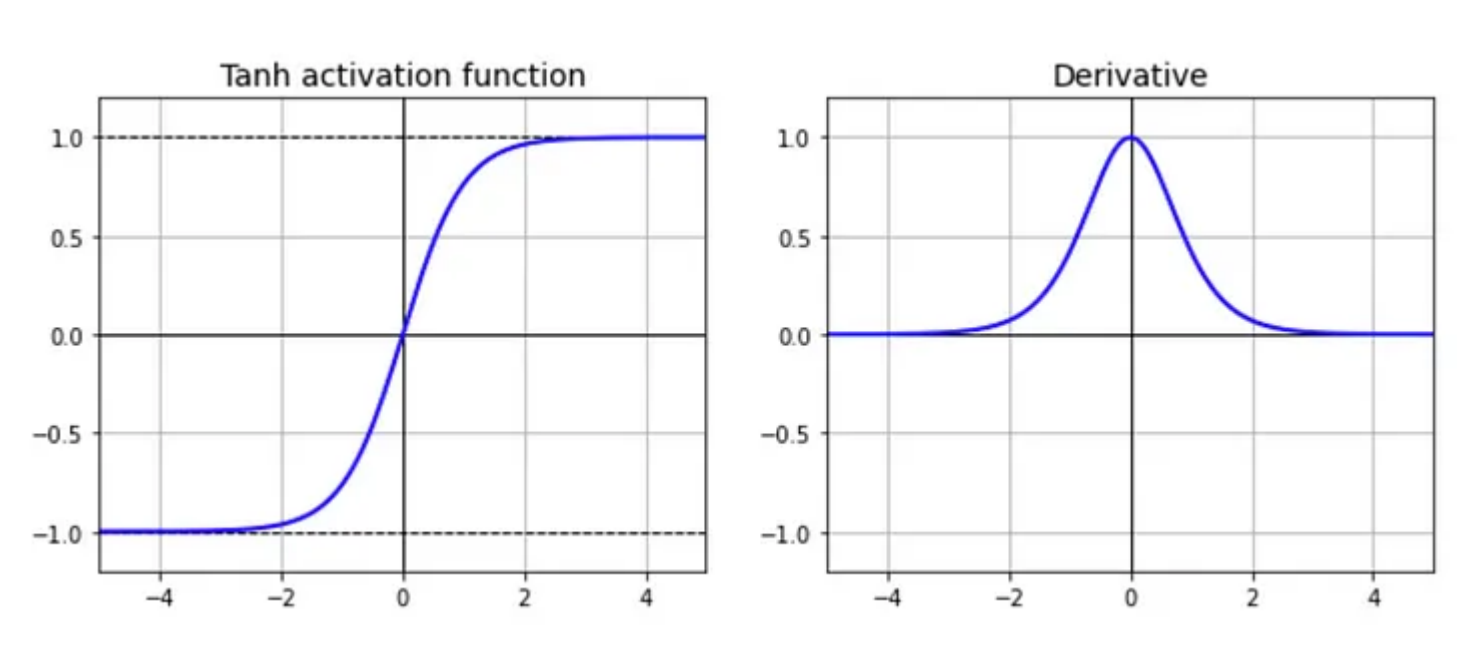
```
Epoch 1/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0356 - accuracy: 0.9911
Epoch 2/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0306 - accuracy: 0.9928
Epoch 3/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0260 - accuracy: 0.9942
Epoch 4/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0223 - accuracy: 0.9954
Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0191 - accuracy: 0.9965
```

Out[28]:
```
<keras.callbacks.History at 0x1e6f545f9d0>
```

Hyperbolic Tangent (Tanh)

Another very popular and widely used activation function is the Hyperbolic Tangent, also known as Tanh.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



The difference is that the output of Tanh is zero centered with a range from -1 to 1 (instead of 0 to 1 in the case of the Sigmoid function)

Since Tanh has characteristics similar to Sigmoid, it also faces the following two problems:

Vanishing gradient: looking at the function plot, you can see that when inputs become small or large, the function saturates at -1 or 1, with a derivative extremely close to 0. Thus it has almost no gradient to propagate back through the network, so there is almost nothing left for lower layers.
Computationally expensive: the function has an exponential operation.

In [30]:
```python
model = keras.Sequential([
    keras.layers.Dense(100, input_shape=(784, ), activation='tanh'),
    keras.layers.Dense(10, activation='tanh')
])
```

In [31]:
```python
model.compile(optimizer='adam',
             loss=sparse_categorical_crossentropy,
             metrics=['accuracy'])
```

In [32]:
```python
model.fit(X_train_flatten, y_train, epochs=5)
```

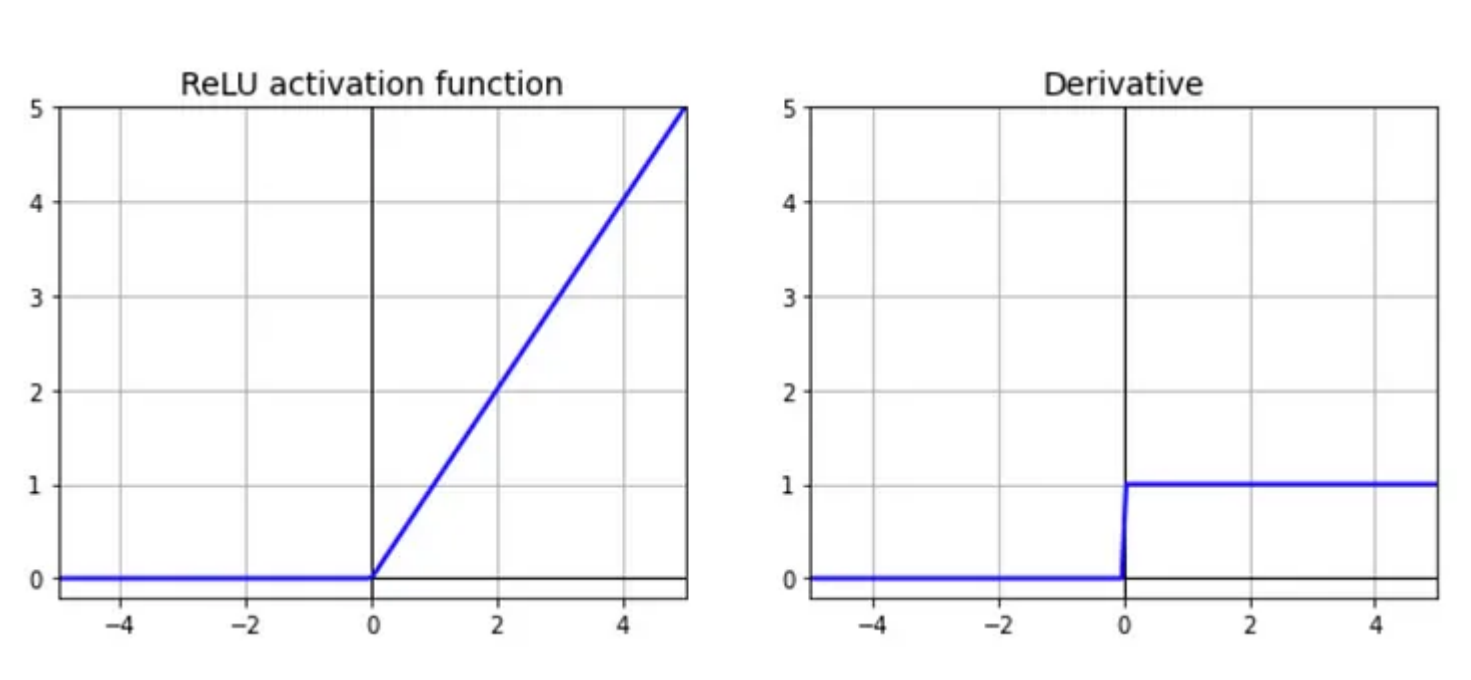
```
Epoch 1/5
1875/1875 [==============================] - 7s 3ms/step - loss: 2.3306 - accuracy: 0.3097
Epoch 2/5
1875/1875 [==============================] - 6s 3ms/step - loss: 2.2787 - accuracy: 0.2274
Epoch 3/5
1875/1875 [==============================] - 6s 3ms/step - loss: 2.3025 - accuracy: 0.1268
Epoch 4/5
1875/1875 [==============================] - 6s 3ms/step - loss: 2.3025 - accuracy: 0.1266
Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step - loss: 2.3025 - accuracy: 0.1266
```

Out[32]:
```
<keras.callbacks.History at 0x1e6f5478670>
```

Rectified Linear Unit (ReLU)

The Rectified Linear Unit (ReLU) is the most commonly used activation function in deep learning. The function returns 0 if the input is negative, but for any positive input, it returns that value back.

$$\begin{cases} 0 & \text{if } x \le 0 \\ x & \text{if } x > 0 \end{cases}$$



ReLU works great in most applications, but it is not perfect. It suffers from a problem known as the dying ReLU.

Dying ReLU

During training, some neurons effectively die, meaning they stop outputting anything other than 0. In some cases, you may find that half of your network's neurons are dead, especially if you used a large learning rate. A neuron dies when its weights get tweaked in such a way that the weighted sum of its inputs are negative for all instances in the training set. When this happens, it just keeps outputting 0s, and gradient descent does not affect it anymore since the gradient of the ReLU function is 0 when its input is negative.

In [52]:
```python
model = keras.Sequential([
    keras.layers.Dense(100, input_shape=(784,), activation='relu'),
    keras.layers.Dense(10, activation='relu')
])
```

In [35]:
```python
model.compile(optimizer='adam',
             loss=sparse_categorical_crossentropy,
             metrics=['accuracy'])
```

In [36]:
```python
model.fit(X_train_flatten, y_train, epochs=5)
```

```
Epoch 1/5
1875/1875 [==============================] - 7s 3ms/step - loss: 2.3309 - accuracy: 0.0996
Epoch 2/5
1875/1875 [==============================] - 6s 3ms/step - loss: 2.3026 - accuracy: 0.0987
Epoch 3/5
1875/1875 [==============================] - 6s 3ms/step - loss: 2.3026 - accuracy: 0.0987
Epoch 4/5
1875/1875 [==============================] - 6s 3ms/step - loss: 2.3026 - accuracy: 0.0987
Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step - loss: 2.3026 - accuracy: 0.0987
```

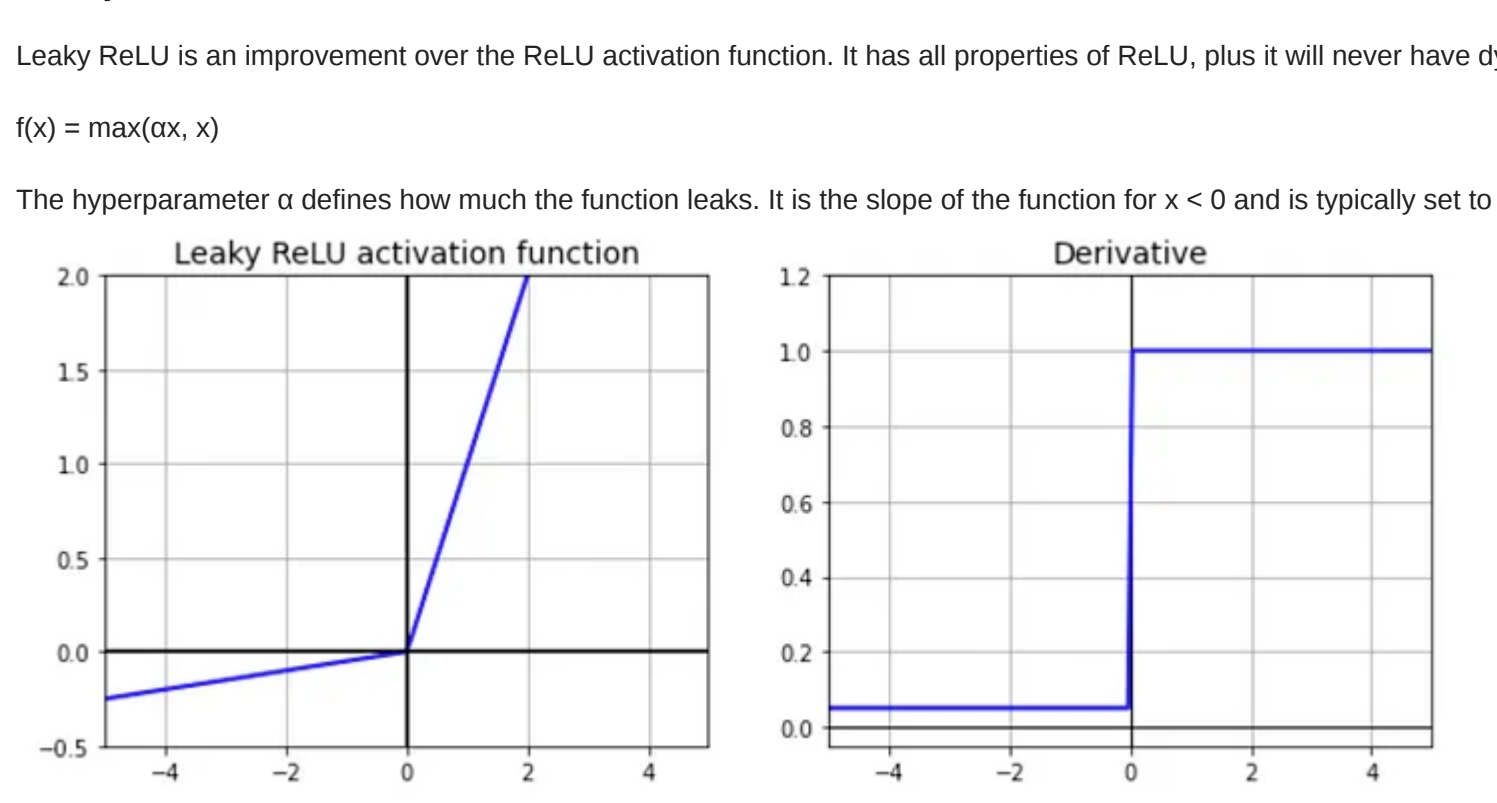
Out[36]:
```
<keras.callbacks.History at 0x1e698817f40>
```

Leaky ReLU

Leaky ReLU is an improvement over the ReLU activation function. It has all properties of ReLU, plus it will never have dying ReLU problem. Leaky ReLU is defined as:

f(x) = max(αx, x)

The hyperparameter α defines how much the function leaks. It is the slope of the function for x < 0 and is typically set to 0.01. The small slope ensures that Leaky ReLU never dies.



In [37]:
```python
from tensorflow.keras.layers import LeakyReLU
leaky_relu = LeakyReLU(alpha=0.01)
```

In [38]:
```python
model = keras.Sequential([
    keras.layers.Dense(100, input_shape=(784,), activation=leaky_relu),
    keras.layers.Dense(10, activation=leaky_relu)
])
```

In [39]:
```python
model.compile(optimizer='adam',
             loss=sparse_categorical_crossentropy,
             metrics=['accuracy'])
```

In [40]:
```python
model.fit(X_train_flatten, y_train, epochs=5)
```

```
Epoch 1/5
1875/1875 [==============================] - 7s 4ms/step - loss: 2.2475 - accuracy: 0.5133
Epoch 2/5
1875/1875 [==============================] - 8s 4ms/step - loss: 2.2971 - accuracy: 0.4893
Epoch 3/5
1875/1875 [==============================] - 7s 3ms/step - loss: 2.2964 - accuracy: 0.4888
Epoch 4/5
1875/1875 [==============================] - 7s 4ms/step - loss: 2.3025 - accuracy: 0.4834
Epoch 5/5
1875/1875 [==============================] - 7s 3ms/step - loss: 2.3025 - accuracy: 0.4834
```

Out[40]:
```
<keras.callbacks.History at 0x1e6f520ba30>
```

Parametric leaky ReLU (PReLU)

Parametric leaky ReLU (PReLU) is a variation of Leaky ReLU, where α is authorized to be learned during training (instead of being a hyperparameter, it becomes a parameter that can be modified by backpropagation like any other parameters). This was reported to strongly outperform ReLU on large image datasets, but on smaller datasets it runs the risk of overfitting the training set

In [41]:
```python
from tensorflow.keras.layers import PReLU
para_relu = PReLU()
```

In [42]:
```python
model = keras.Sequential([
    keras.layers.Dense(100, input_shape=(784,), activation=para_relu),
    keras.layers.Dense(10, activation=leaky_relu)
])
```

In [43]:
```python
model.compile(optimizer='adam',
             loss=sparse_categorical_crossentropy,
             metrics=['accuracy'])
```

In [44]:
```python
model.fit(X_train_flatten, y_train, epochs=5)
```

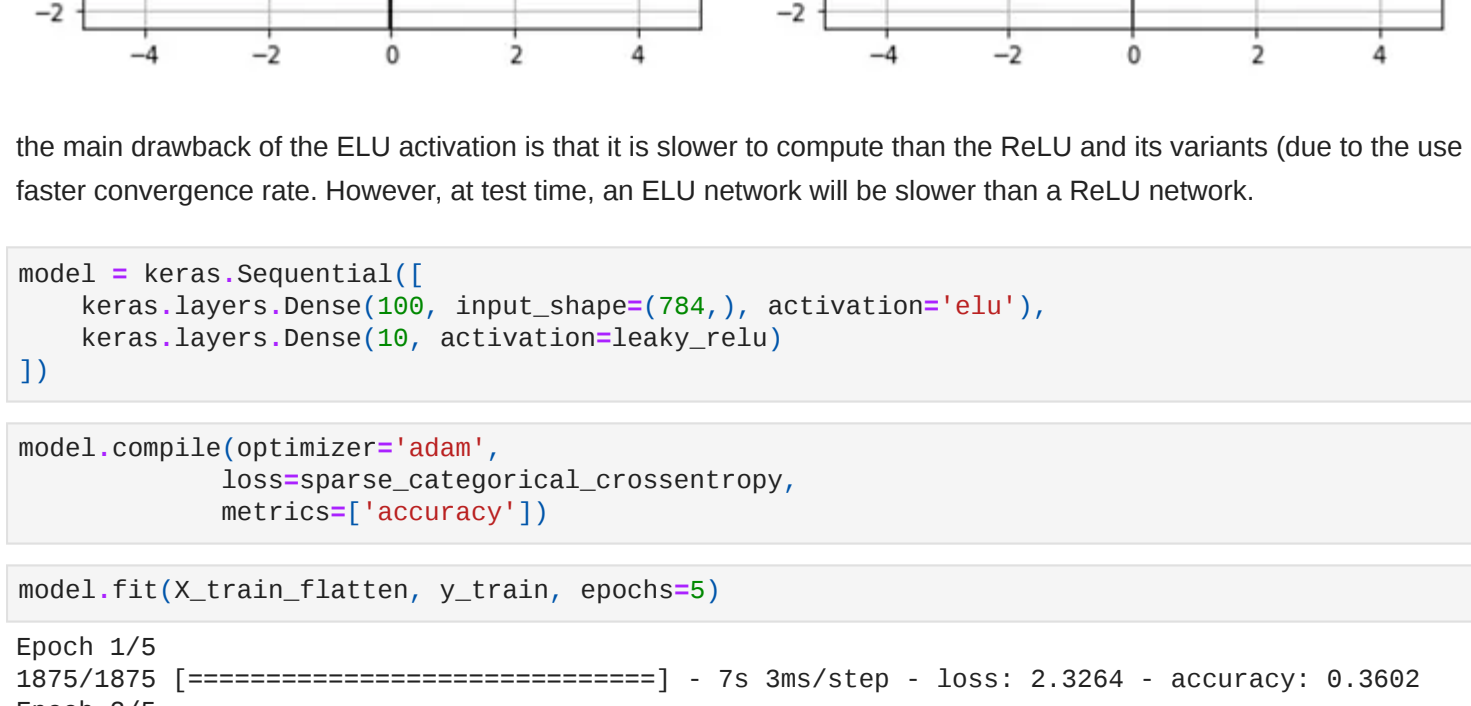
```
Epoch 1/5
1875/1875 [==============================] - 8s 4ms/step - loss: 2.3357 - accuracy: 0.3374
Epoch 2/5
1875/1875 [==============================] - 6s 3ms/step - loss: 2.3026 - accuracy: 0.2455
Epoch 3/5
1875/1875 [==============================] - 6s 3ms/step - loss: 2.3026 - accuracy: 0.2455
Epoch 4/5
1875/1875 [==============================] - 7s 4ms/step - loss: 2.3026 - accuracy: 0.2455
Epoch 5/5
1875/1875 [==============================] - 7s 4ms/step - loss: 2.3026 - accuracy: 0.2455
```

Out[44]:
```
<keras.callbacks.History at 0x1e6989bd670>
```

Exponential Linear Unit (ELU)

Exponential Linear Unit (ELU) is a variation of ReLU with a better output for z < 0.

$$\begin{cases} \alpha\left(e^x - 1\right) & \text{if } x \le 0 \\ x & \text{if } x > 0 \end{cases}$$



the main drawback of the ELU activation is that it is slower to compute than the ReLU and its variants (due to the use of the exponential function), but during training this is compensated by the faster convergence rate. However, at test time, an ELU network will be slower than a ReLU network.

In [45]:
```python
model = keras.Sequential([
    keras.layers.Dense(100, input_shape=(784,), activation='elu'),
    keras.layers.Dense(10, activation=leaky_relu)
])
```

In [46]:
```python
model.compile(optimizer='adam',
             loss=sparse_categorical_crossentropy,
             metrics=['accuracy'])
```

In [47]:
```python
model.fit(X_train_flatten, y_train, epochs=5)
```

```
Epoch 1/5
1875/1875 [==============================] - 7s 3ms/step - loss: 2.3264 - accuracy: 0.3602
Epoch 2/5
1875/1875 [==============================] - 6s 3ms/step - loss: 2.2979 - accuracy: 0.3786
Epoch 3/5
1875/1875 [==============================] - 6s 3ms/step - loss: 2.2943 - accuracy: 0.3842
Epoch 4/5
1875/1875 [==============================] - 6s 3ms/step - loss: 2.2902 - accuracy: 0.3792
Epoch 5/5
1875/1875 [==============================] - 7s 4ms/step - loss: 2.3024 - accuracy: 0.3548
```

Out[47]:
```
<keras.callbacks.History at 0x1e69afc3d00>
```

Scaled Exponential Linear Unit (SELU)

Exponential Linear Unit (SELU) activation function is another variation of ReLU. if you build a neural network composed exclusively of a stack of dense layers, and if all hidden layers use the SELU activation function, then the network will self-normalize (the output of each layer will tend to preserve mean 0 and standard deviation 1 during training, which resolves the vanishing/exploding gradients problem). This activation function often outperforms other activation functions very significantly.

To use SELU with Keras and TensorFlow 2, just set activation='relu' and kernel_initializer='lecun_normal'

In [49]:
```python
model = keras.Sequential([
    keras.layers.Dense(100, input_shape=(784,), activation='relu', kernel_initializer='lecun_normal'),
    keras.layers.Dense(10, activation=leaky_relu)
])
```

In [50]:
```python
model.compile(optimizer='adam',
             loss=sparse_categorical_crossentropy,
             metrics=['accuracy'])
```

In [51]:
```python
model.fit(X_train_flatten, y_train, epochs=5)
```

```
Epoch 1/5
1875/1875 [==============================] - 7s 3ms/step - loss: 2.3477 - accuracy: 0.3004
Epoch 2/5
1875/1875 [==============================] - 7s 4ms/step - loss: 2.3021 - accuracy: 0.2736
Epoch 3/5
1875/1875 [==============================] - 7s 4ms/step - loss: 2.2992 - accuracy: 0.2296
Epoch 4/5
1875/1875 [==============================] - 8s 4ms/step - loss: 2.3025 - accuracy: 0.2353
Epoch 5/5
1875/1875 [==============================] - 8s 4ms/step - loss: 2.3026 - accuracy: 0.2402
```

Out[51]:
```
<keras.callbacks.History at 0x1e6f5118340>
```

In [ ]: