SAP ID:

SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (DATA SCIENCE)

COURSE CODE: DJ19DSC501 **DATE: 21/10/2023**

COURSE NAME: Machine Learning - II **CLASS: AY 2023-24**

Jhanvi Parekh

60009210033

D11

LAB EXPERIMENT NO.5

AIM :

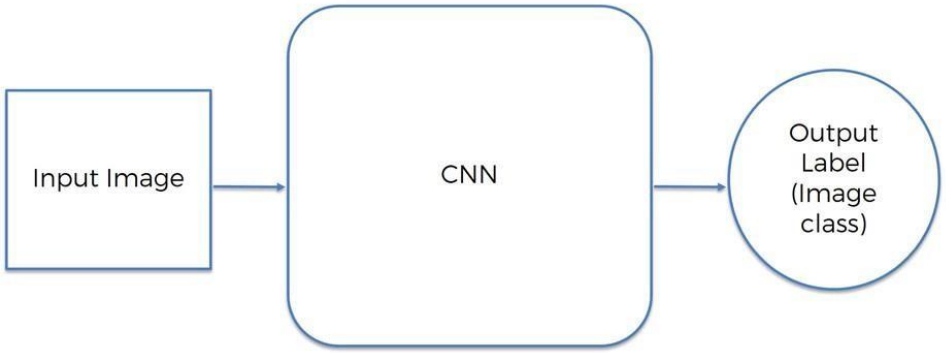
Building CNN models for image categorization.

THEORY:

A **Convolutional Neural Network (ConvNet/CNN)** is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets have the ability to learn these filters/characteristics. The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlap to cover the entire visual area.

CNN in image categorization base view:

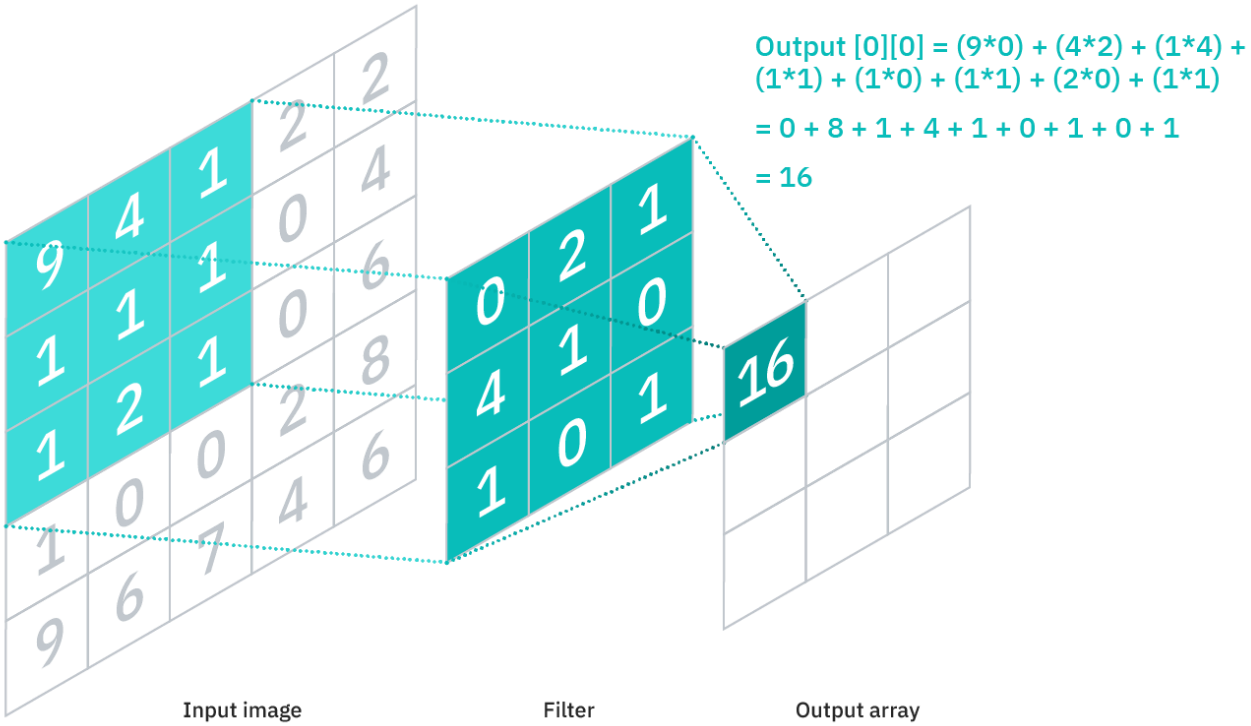
SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)



Major Steps in building a CNN:

1) Convolution: The convolutional layer is the core building block of a CNN, and it is where the majority of computation occurs. It requires a few components, which are input data, a filter, and a feature map. Let's assume that the input will be a color image, which is made up of a matrix of pixels in 3D. This means that the input will have three dimensions—a height, width, and depth—which correspond to RGB in an image. We also have a feature detector, also known as a kernel or a filter, which will move across the receptive fields of the image, checking if the feature is present. This process is known as a convolution. The feature detector is a two-dimensional (2-D) array of weights, which represents part of the image. While they can vary in size, the filter size is typically a 3x3 matrix; this also determines the size of the receptive field. The filter is then applied to an area of the image, and a dot product is calculated between the input pixels and the filter. This dot product is then fed into an output array. Afterwards, the filter shifts by a stride, repeating the process until the kernel has swept across the entire image. The final output from the series of dot products from the input and the filter is known as a feature map, activation map, or a convolved feature.

SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)



2) ReLU: After each convolution operation, a CNN applies a Rectified Linear Unit (ReLU) transformation to the feature map, introducing nonlinearity to the model.

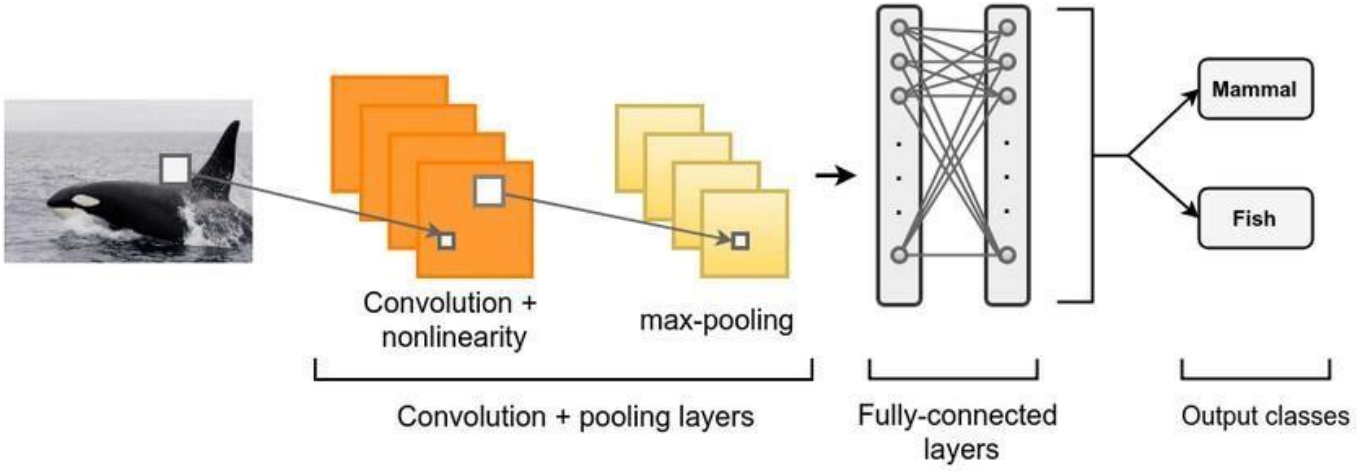
Another convolution layer can follow the initial convolution layer. When this happens, the structure of the CNN can become hierarchical as the later layers can see the pixels within the receptive fields of prior layers.

3) Pooling: Pooling layers, also known as downsampling, conducts dimensionality reduction, reducing the number of parameters in the input. Similar to the convolutional layer, the pooling operation sweeps a filter across the entire input, but the difference is that this filter does not have any weights. Instead, the kernel applies an aggregation function to the values within the receptive field, populating the output array. There are two main types of pooling:

- **Max pooling:** As the filter moves across the input, it selects the pixel with the maximum value to send to the output array. As an aside, this approach tends to be used more often compared to average pooling.
- **Average pooling:** As the filter moves across the input, it calculates the average value within the receptive field to send to the output array.

While a lot of information is lost in the pooling layer, it also has a number of benefits to the CNN. They help to reduce complexity, improve efficiency, and limit risk of overfitting.

4) Fully Connected Layer: The name of the full-connected layer aptly describes itself. As mentioned earlier, the pixel values of the input image are not directly connected to the output layer in partially connected layers. However, in the fully-connected layer, each node in the output layer connects directly to a node in the previous layer. This layer performs the task of classification based on the features extracted through the previous layers and their different filters. While convolutional and pooling layers tend to use ReLu functions, FC layers usually leverage a softmax activation function to classify inputs appropriately, producing a probability from 0 to 1. Output layer can have a single neuron if problem is a binary classification problem or it can have more than 1 output neurons if problem is a multi-class classification problem.



Tasks to be performed:

1. **Choose appropriate Image Categorization dataset**
   (Image Categorization dataset from Kaggle, UCI Machine Learning Repository, Data.Gov or any other online resource/website.)
   OR
   INPUT DATA: https://www.kaggle.com/datasets/paultimothymooney/blood-cells
2. **Build an multiclass image categorization CNN network which correctly classifies different categories of images in the dataset.**
3. **Split original dataset to train and test set**
4. **Generate the accuracy of the built model.**

SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

5. **Perform hyperparameter tuning to increase the accuracy of the CNN.**
6. **Apply transfer learning to implement different pre-trained models like VGG16, VGG19, DenseNet, ResNet, InceptionNet etc. And compare their prediction performance over different evaluation parameters.**

LINK : https://colab.research.google.com/drive/1shY6n0g-jz0YLVwPekx8w66dRAwCEbMm?usp=sharing

```
import keras
import cv2
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D
import matplotlib.pyplot as plt


(x_train, y_train), (x_test, y_test) = mnist.load_data()
print(x_train.shape)
print(y_train.shape)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
(60000, 28, 28)
(60000,)
```

```
x_train = x_train.reshape(60000,28,28)
x_test = x_test.reshape(x_test.shape[0],28,28)


input_shape=(28,28,1)
```

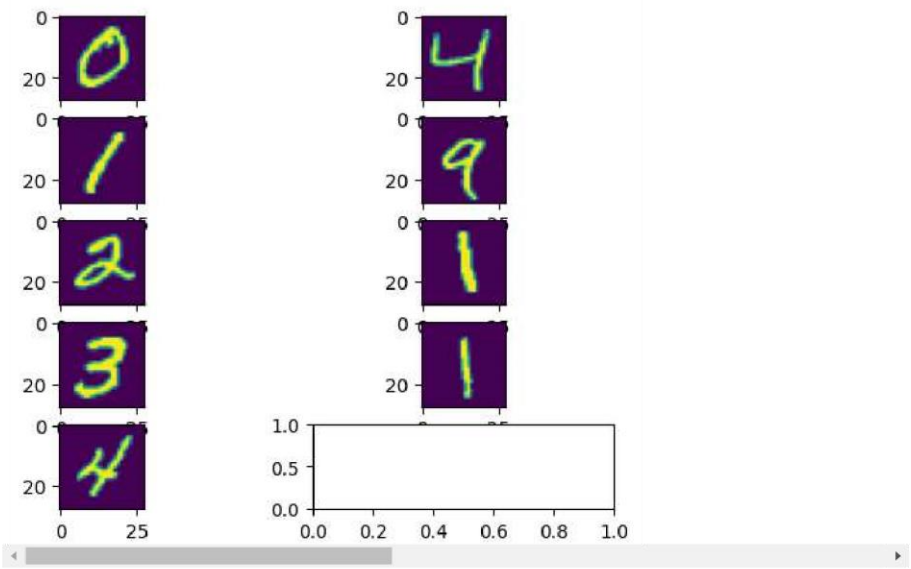
+ Code + Text

```
y_train = keras.utils.to_categorical(y_train,10)
print(y_train.shape)
```

```
(60000, 10)
```

```
y_test = keras.utils.to_categorical(y_test,10)


x_train = x_train.astype('float32')
x_test = x_test.astype('float32')


for i in range(10):
  plt.imshow(x_train[i])
  plt.subplot(5,2,i+1)
```

```
<ipython-input-14-2a1f7e8a1497>:3: MatplotlibDeprecationWarning: Auto-removal of overlap
  plt.subplot(5,2,i+1)
```




```
#from keras.optimizers import Adam
#optimizer = Adam(learning_rate=0.001)
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
```

```
model.add(Dense(10, activation='softmax'))
model.compile(loss=keras.losses.categorical_crossentropy, optimizer=optimizer, metrics=['accuracy'])
model.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_13 (Conv2D)          (None, 26, 26, 32)        320

 max_pooling2d_12 (MaxPooli  (None, 13, 13, 32)        0
 ng2D)

 conv2d_14 (Conv2D)          (None, 11, 11, 32)        9248

 max_pooling2d_13 (MaxPooli  (None, 5, 5, 32)          0
 ng2D)

 dropout_12 (Dropout)        (None, 5, 5, 32)          0

 flatten_6 (Flatten)         (None, 800)               0

 dense_12 (Dense)            (None, 256)               205056

 dropout_13 (Dropout)        (None, 256)               0

 dense_13 (Dense)            (None, 10)                2570

=================================================================
Total params: 217194 (848.41 KB)
Trainable params: 217194 (848.41 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

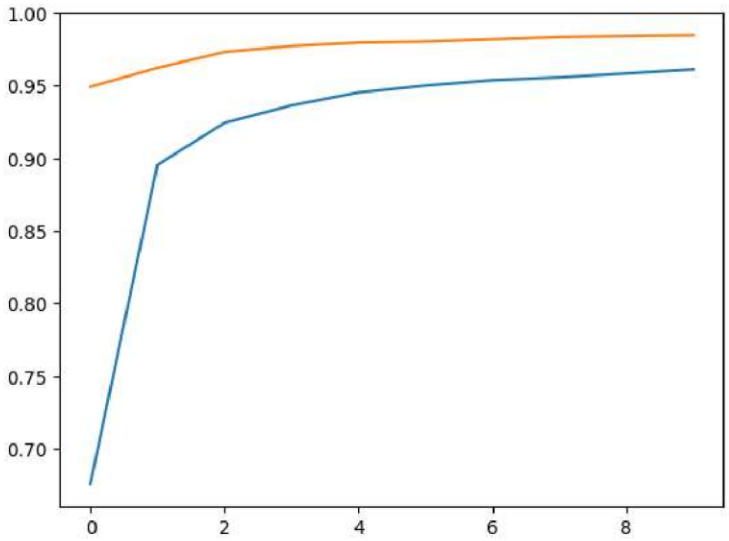
```
from keras.utils import plot_model
from keras.optimizers import Adadelta, Adagrad, Adam, RMSprop, SGD
batch_size = 128
epochs = 10
optimizers = ['Adadelta', 'Adagrad', 'Adam', 'RMSprop', 'SGD']
for optimizer_name in optimizers:
    model = Sequential()
    model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(32, kernel_size=(3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))
    model.add(Flatten())
    model.add(Dense(256, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(10, activation='softmax'))

    # Compile the model with the current optimizer
    if optimizer_name == 'Adadelta':
        optimizer = Adadelta()
    elif optimizer_name == 'Adagrad':
        optimizer = Adagrad()
    elif optimizer_name == 'Adam':
        optimizer = Adam()
    elif optimizer_name == 'RMSprop':
        optimizer = RMSprop()
    elif optimizer_name == 'SGD':
        optimizer = SGD()

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

    # Visualize and save the model
    plot_model(model, to_file=f"{optimizer_name}_mnist_model.jpg", show_shapes=True)

    # Train the model
    hist = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, verbose=1, validation_data=(x_test, y_test))
```

```
Epoch 1/10
469/469 [==============================] - 24s 49ms/step - loss: 66.6308 - accuracy: 0.0944 - val_loss: 21.1252 - val_accuracy: 0.0966
Epoch 2/10
469/469 [==============================] - 25s 54ms/step - loss: 52.9381 - accuracy: 0.1161 - val_loss: 14.1818 - val_accuracy: 0.1787
Epoch 3/10
469/469 [==============================] - 32s 68ms/step - loss: 43.6745 - accuracy: 0.1379 - val_loss: 10.1683 - val_accuracy: 0.2872
Epoch 4/10
469/469 [==============================] - 31s 67ms/step - loss: 35.9457 - accuracy: 0.1623 - val_loss: 7.5930 - val_accuracy: 0.3956
```

```
Epoch 5/10
469/469 [==============================] - 22s 48ms/step - loss: 29.8841 - accuracy: 0.1897 - val_loss: 5.7701 - val_accuracy: 0.4716
Epoch 6/10
469/469 [==============================] - 22s 47ms/step - loss: 24.8090 - accuracy: 0.2201 - val_loss: 4.4938 - val_accuracy: 0.5330
Epoch 7/10
469/469 [==============================] - 23s 50ms/step - loss: 20.8205 - accuracy: 0.2457 - val_loss: 3.5246 - val_accuracy: 0.5869
Epoch 8/10
469/469 [==============================] - 23s 49ms/step - loss: 17.6769 - accuracy: 0.2763 - val_loss: 2.8603 - val_accuracy: 0.6375
Epoch 9/10
469/469 [==============================] - 25s 52ms/step - loss: 15.0296 - accuracy: 0.3032 - val_loss: 2.4116 - val_accuracy: 0.6690
Epoch 10/10
469/469 [==============================] - 29s 62ms/step - loss: 13.0525 - accuracy: 0.3299 - val_loss: 2.0450 - val_accuracy: 0.6964
Epoch 1/10
469/469 [==============================] - 27s 54ms/step - loss: 4.4907 - accuracy: 0.5621 - val_loss: 0.4189 - val_accuracy: 0.8735
Epoch 2/10
469/469 [==============================] - 22s 47ms/step - loss: 1.2628 - accuracy: 0.6910 - val_loss: 0.3323 - val_accuracy: 0.9040
Epoch 3/10
469/469 [==============================] - 22s 46ms/step - loss: 0.9532 - accuracy: 0.7447 - val_loss: 0.2797 - val_accuracy: 0.9180
Epoch 4/10
469/469 [==============================] - 23s 48ms/step - loss: 0.7956 - accuracy: 0.7799 - val_loss: 0.2474 - val_accuracy: 0.9290
Epoch 5/10
469/469 [==============================] - 23s 49ms/step - loss: 0.6909 - accuracy: 0.8041 - val_loss: 0.2216 - val_accuracy: 0.9363
Epoch 6/10
469/469 [==============================] - 21s 45ms/step - loss: 0.6215 - accuracy: 0.8218 - val_loss: 0.2045 - val_accuracy: 0.9394
Epoch 7/10
469/469 [==============================] - 23s 49ms/step - loss: 0.5692 - accuracy: 0.8351 - val_loss: 0.1879 - val_accuracy: 0.9443
Epoch 8/10
469/469 [==============================] - 24s 51ms/step - loss: 0.5303 - accuracy: 0.8463 - val_loss: 0.1772 - val_accuracy: 0.9474
Epoch 9/10
469/469 [==============================] - 32s 69ms/step - loss: 0.4952 - accuracy: 0.8562 - val_loss: 0.1673 - val_accuracy: 0.9500
Epoch 10/10
469/469 [==============================] - 29s 61ms/step - loss: 0.4681 - accuracy: 0.8634 - val_loss: 0.1603 - val_accuracy: 0.9525
Epoch 1/10
469/469 [==============================] - 23s 47ms/step - loss: 1.2270 - accuracy: 0.8429 - val_loss: 0.0897 - val_accuracy: 0.9699
Epoch 2/10
469/469 [==============================] - 23s 49ms/step - loss: 0.1822 - accuracy: 0.9460 - val_loss: 0.0629 - val_accuracy: 0.9801
Epoch 3/10
469/469 [==============================] - 22s 46ms/step - loss: 0.1354 - accuracy: 0.9595 - val_loss: 0.0544 - val_accuracy: 0.9837
Epoch 4/10
469/469 [==============================] - 23s 49ms/step - loss: 0.1109 - accuracy: 0.9667 - val_loss: 0.0423 - val_accuracy: 0.9859
Epoch 5/10
469/469 [==============================] - 21s 45ms/step - loss: 0.0993 - accuracy: 0.9710 - val_loss: 0.0423 - val_accuracy: 0.9863
Epoch 6/10
469/469 [==============================] - 22s 48ms/step - loss: 0.0861 - accuracy: 0.9732 - val_loss: 0.0379 - val_accuracy: 0.9877
Epoch 7/10
469/469 [==============================] - 22s 47ms/step - loss: 0.0803 - accuracy: 0.9759 - val_loss: 0.0378 - val_accuracy: 0.9891
Epoch 8/10
469/469 [==============================] - 22s 46ms/step - loss: 0.0772 - accuracy: 0.9766 - val_loss: 0.0367 - val_accuracy: 0.9878
Epoch 9/10
```

```
y1 = hist.history['accuracy']
y2 = hist.history['val_accuracy']
plt.plot(y1)
plt.plot(y2)
```

```
[<matplotlib.lines.Line2D at 0x7950ba5f9540>]
```



```
model.save("mnist.h5")
from tensorflow.keras.models import load_model
import numpy as np
```

```
import numpy as np
img = x_test[4]
#.reshape(len(x_test[0]),28)
#img = cv2.resize(img,(28,28))
img = np.reshape(img,(1,28,28))
img = img.astype('float32')
img = img/255
pred = model.predict(img)
answer = np.argmax(pred)
print(answer)
plt.matshow(x_test[4])
```

```
1/1 [==============================] - 0s 25ms/step
1
<matplotlib.image.AxesImage at 0x7950b8684a60>
```