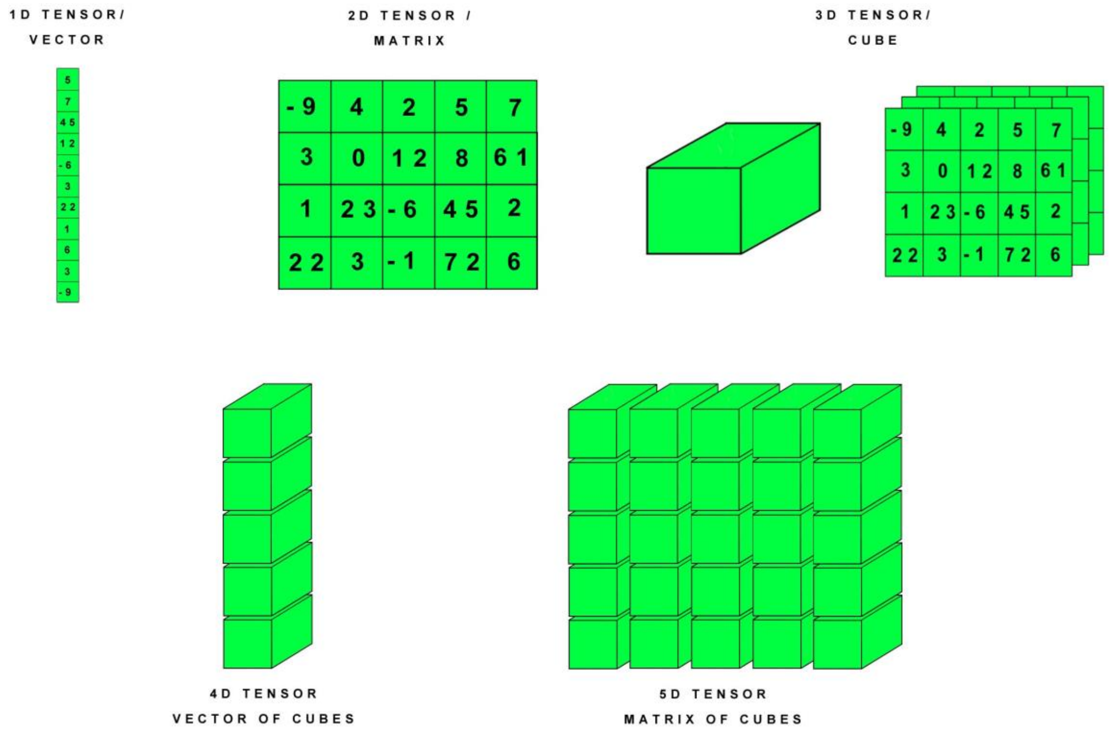
- **Tensors**
- **Training a model in Keras**
- **Character recognition example -MINST**

Common Data Stored in Tensors

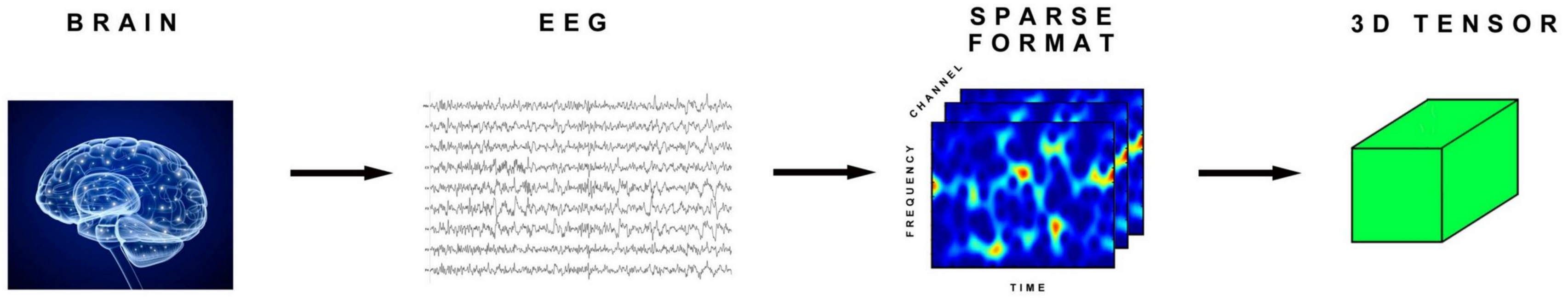
- Like vectors and matrices, tensors are represented with N dimensional arrays in Python.
- Some of widespread datasets :
- 3D = Time series
- 4D = Images
- 5D = Videos



https://www.experfy.com/blog/ai-ml/learning-ai-if-you-suck-at-math-part4-tensors-illustrated-with-cats/

Time Series Data

- **Medical Scans**
  We can encode an electroencephalogram EEG signal from the brain as a 3D tensor, because it can be encapsulated as 3 parameters:
- (time, frequency, channel)
- The transformation would look like this:



- If we had EEG scan of more than one patient, we would have a 4D tensor like this:
- (sample_size, time, frequency, channel)

Kaynak:https://hackernoon.com/learning-ai-if-you-suck-at-math-p4-tensors-illustrated-with-cats-27f0002c9b32

Text Data

- We can also store a tweet dataset in a 3D tensor.
- For example, Tweet is 140 characters. Twitter uses the UTF-8 standard, which allows for millions of character types, but we are only concerned with the first 128 characters, as they are the same as basic ASCII.
- A single tweet can be encapsulated as a vector (140,128) of the 2D shape.
- If we use 1 million tweets, we will store it as a 3D shape tensor:
- **(number_of_tweets, tweet, character)**
- **(1000000,140,128)**

Image dataset

- TensorFlow usually stores image data as follows:

  (sample_size, height, width, color_depth)

- There are 60,000 images in the MNIST dataset. It is 28 pixels wide x 28 pixels high. They have 1 color depth representing the gray scale.

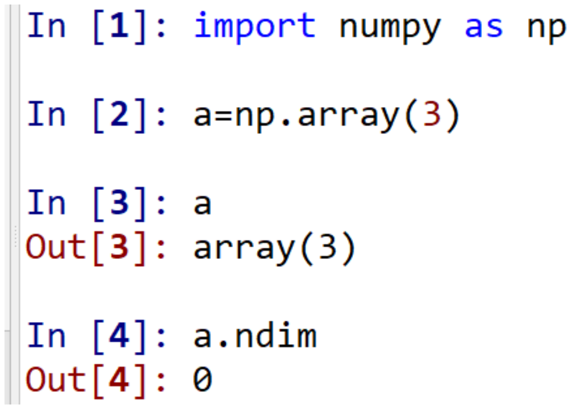
  (60000,28,28,1)

Video dataset

- A 5D tensor can store video data. In TensorFlow, video data is stored like this:
- **(sample_size, frames, width, height, color_depth)**
- A five-minute video (60 seconds x 5 = 300 seconds), 1920 pixels x 1080 pixels would store a 4D tensor that looks like this in 25 sampled color frames per second (300 seconds x 15 = 4500 frames):
- **(4500,1920,1080,3)**
- If there were 10 videos, we would have a shape tensor of 5D:
- **(10,4500,1920,1080,3)**
- Keras allows us to store 32-bit or 64-bit as floating point numbers:
- If we store the 5D tensor above with float32
- 10 x 4500 x 1920 x 1080 x 3 x 32
- = 8.957.952.000.000 byte
- = 1.0184 Terabyte

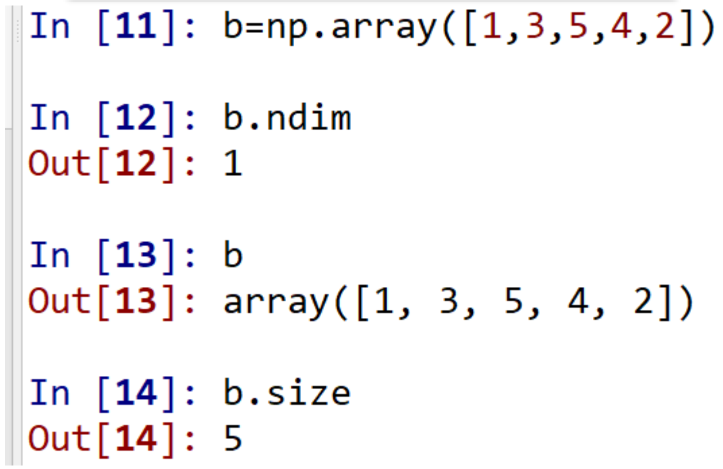
Tensor examples

0D Tensor (Scaler)



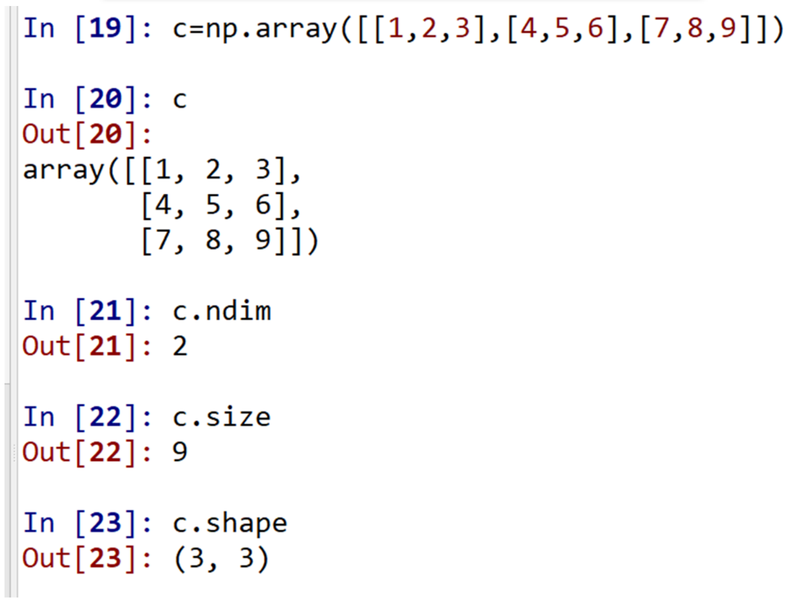
```
In [1]: import numpy as np

In [2]: a=np.array(3)

In [3]: a
Out[3]: array(3)

In [4]: a.ndim
Out[4]: 0
```

1D Tensor (Array)



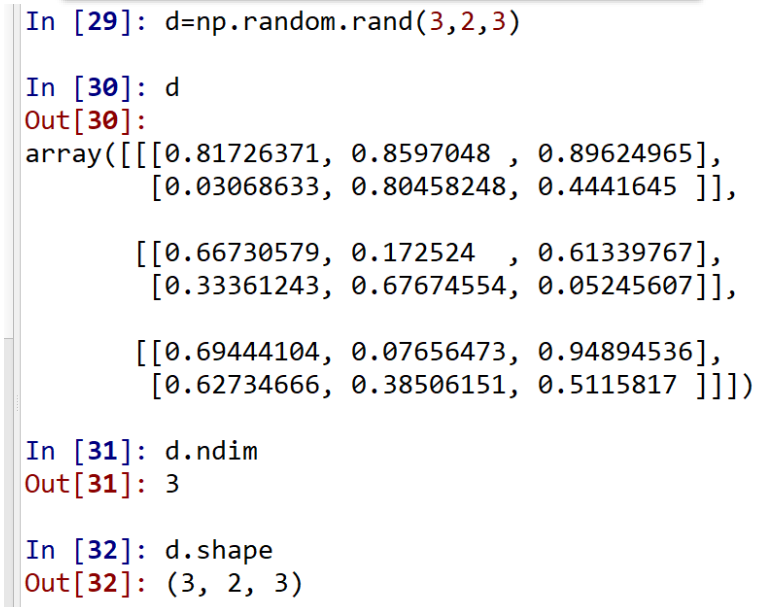
```
In [11]: b=np.array([1,3,5,4,2])

In [12]: b.ndim
Out[12]: 1

In [13]: b
Out[13]: array([1, 3, 5, 4, 2])

In [14]: b.size
Out[14]: 5
```

2D Tensor (matrix)



```
In [19]: c=np.array([[1,2,3],[4,5,6],[7,8,9]])

In [20]: c
Out[20]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [21]: c.ndim
Out[21]: 2

In [22]: c.size
Out[22]: 9

In [23]: c.shape
Out[23]: (3, 3)
```

3D Tensor



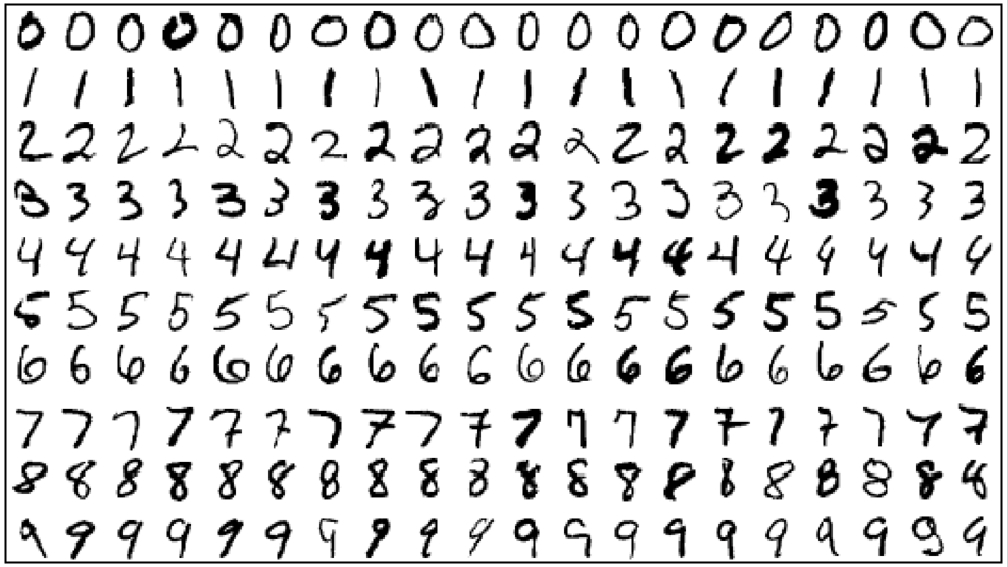
```
In [29]: d=np.random.rand(3,2,3)

In [30]: d
Out[30]:
array([[[0.81726371, 0.8597048 , 0.89624965],
        [0.03068633, 0.80458248, 0.4441645 ]],

       [[0.66730579, 0.172524  , 0.61339767],
        [0.33361243, 0.67674554, 0.05245607]],

       [[0.69444104, 0.07656473, 0.94894536],
        [0.62734666, 0.38506151, 0.5115817 ]]])

In [31]: d.ndim
Out[31]: 3

In [32]: d.shape
Out[32]: (3, 2, 3)
```

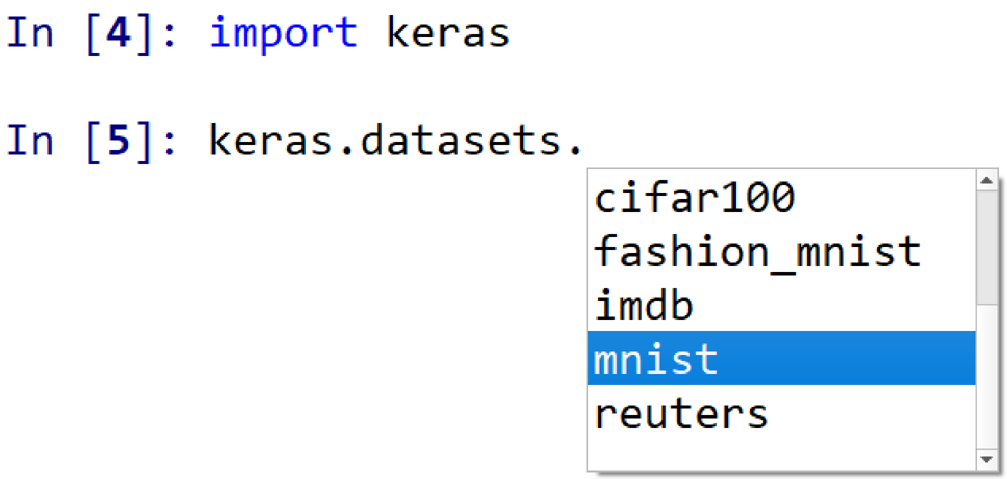
Training a model in Keras

- **Load dataset:** Training set may need to be converted to fit the input of the model
- **Network model:**
  - Sequential or Functional network structures
  - The number of inputs and outputs, the number of layers, activation functions.
- **Compile the network:** Optimization functions, loss functions, metrics
- **Training:** epochs, bacth size, learning rate

MNIST Example: Loading data set

- MNIST dataset:





```
In [4]: import keras

In [5]: keras.datasets.
```

Loading data set

```
from keras.datasets import mnist

(train_images, train_labels),
(test_images, test_labels) =
                  mnist.load_data()
```

- The Mnist data set is divided into two parts as training and testing:
- Training set : 60.000 images
- Testing set : 10.000 images

Preprocessing

- The image in the form of a 28x28 matrix is transformed into an array of 784 elements
- Also, pixel values ranging from 0-255 are scaled between 0-1.

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

Defining a network model

- After the source from which the models are defined is imported, the network model is defined.

```
from keras import models
model = models.Sequential()
```

Adding layer:

- First layer 28 * 28 (image size) = 784 entries,
- It consists of 256 artificial neurons with relu (rectified linear unit) activation function.

```
model.add( layers.Dense(256,
            activation='relu',
           input_shape=(28 * 28,)))
```

Adding output layer

- Layers like the previous one can be added and the number of neurons can be determined empirically.
- For the last layer, the number of neurons is determined according to the number of outputs
- Each of the outputs corresponds to a digit.

```
model.add(layers.Dense(10,
                      activation='softmax'))
```

Ağı derlemek-optimizasyon fonksiyonları

- `optimizer:` optimization algorithm determines how to update network parameters by looking at change in loss function
- https://keras.io/optimizers/
  - **SGD**
  - **Adagrad**
  - **RMSprop**:
  - **Adadelta**
  - **Adam**
  - **Adamax**
  - **Nadam**

Optimization methods: https://arxiv.org/pdf/1609.04747.pdf

Compiling the network

- `optimizer`: the optimization algorithm determines how the network parameters are updated by looking at the change in the loss function
- `Loss`: is a function to measure the performance of the network
- `metrics`: to be observed during the training and testing phases. Here accuracy will be observed.

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

Compiling the network - loss function

categorical_crossentropy -cce

If your targets are one-hot encoded, use categorical_crossentropy.

Examples of one-hot encodings:

[1,0,0]
[0,1,0]
[0,0,1]

sparse_categorical_crossentropy-scce

But if your targets are integers, use sparse_categorical_crossentropy.

Examples of integer encodings:

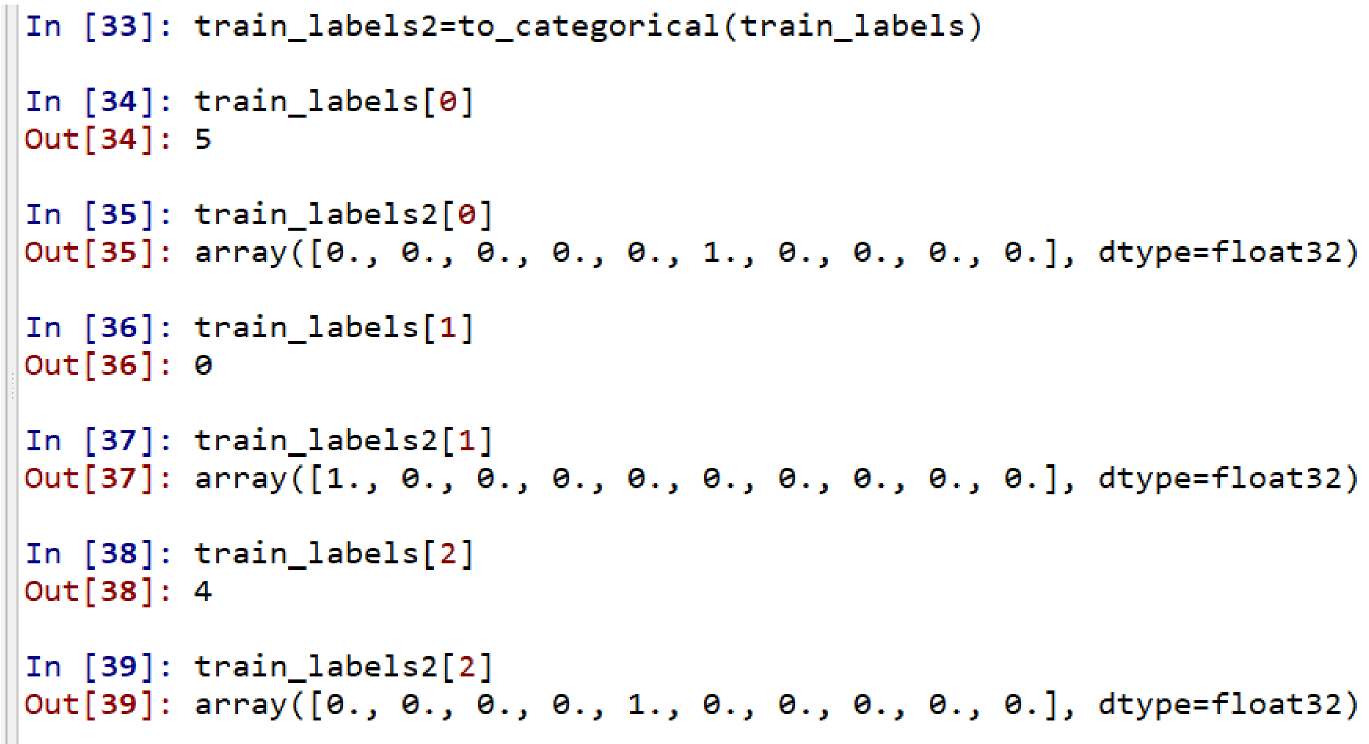
1
2
3

- While cce use a lot of space when compared to scce it gives the information about the probability of other classes.

Compiling the network - loss function

- `to_categorical` is used with categorical_crossentropy. Converts a class vector of integers to binary class matrix.

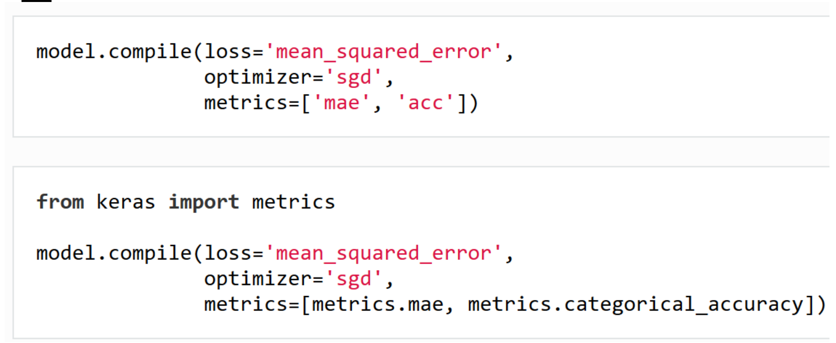
```
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```



```
In [33]: train_labels2=to_categorical(train_labels)

In [34]: train_labels[0]
Out[34]: 5

In [35]: train_labels2[0]
Out[35]: array([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.], dtype=float32)

In [36]: train_labels[1]
Out[36]: 0

In [37]: train_labels2[1]
Out[37]: array([1., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)

In [38]: train_labels[2]
Out[38]: 4

In [39]: train_labels2[2]
Out[39]: array([0., 0., 0., 0., 1., 0., 0., 0., 0., 0.], dtype=float32)
```

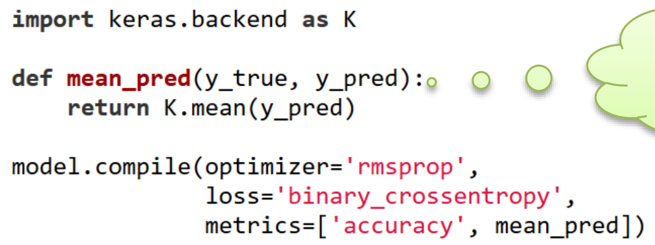
Compiling the network - metrics

- `metrics`: A metric is a function that is used to judge the performance of your model.
- Metric functions are similar to loss functions, except that the results from evaluating a metric are not used when training the model. Note that you may use any loss function as a metric.
  - **binary_accuracy**
  - **categorical_accuracy**
  - **sparse_categorical_accuracy**
  - **top_k_categorical_accuracy**
  - **sparse_top_k_categorical_accuracy**
  -



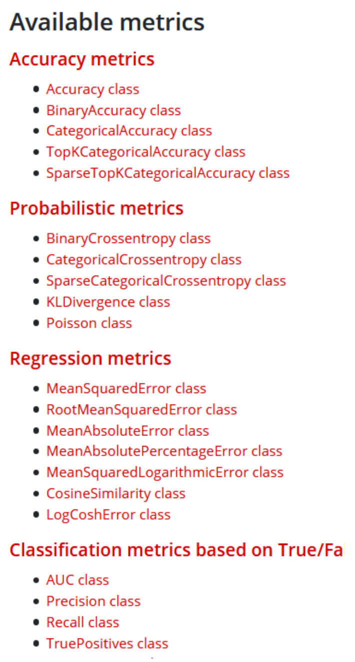
```
model.compile(loss='mean_squared_error',
              optimizer='sgd',
              metrics=['mae', 'acc'])
```

```
from keras import metrics

model.compile(loss='mean_squared_error',
              optimizer='sgd',
              metrics=[metrics.mae, metrics.categorical_accuracy])
```



```
import keras.backend as K

def mean_pred(y_true, y_pred):
    return K.mean(y_pred)

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy', mean_pred])
```

Custom metric definition



Available metrics

Accuracy metrics

- Accuracy class
- BinaryAccuracy class
- CategoricalAccuracy class
- TopKCategoricalAccuracy class
- SparseTopKCategoricalAccuracy class

Probabilistic metrics

- BinaryCrossentropy class
- CategoricalCrossentropy class
- SparseCategoricalCrossentropy class
- KLDivergence class
- Poisson class

Regression metrics

- MeanSquaredError class
- RootMeanSquaredError class
- MeanAbsoluteError class
- MeanAbsolutePercentageError class
- MeanSquaredLogarithmicError class
- CosineSimilarity class
- LogCoshError class

Classification metrics based on True/Fa

- AUC class
- Precision class
- Recall class
- TruePositives class

https://keras.io/api/metrics/

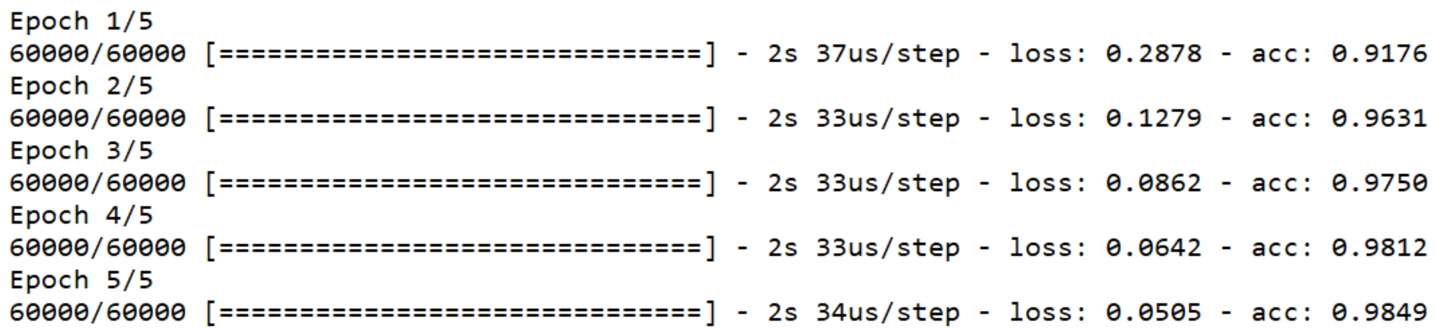
Training the netwok

- **fit**: Trains the model for a fixed number of epochs (iterations on a dataset).
- **epochs** is number of epochs to train the model. An epoch is an iteration over the entire x and y data provided
- **batch_size** is the number of samples per gradient update. If unspecified, batch_size will default to 32.
- **if batch_size =1 then stockastic gd**
- **if batch_size=dataset_size then batch gd**
- **if batch_size=dataset_size/n then minibatch gd**

```
model.fit(train_images, train_labels,
epochs=5,batch_size=128))
```

Evaluation of the network

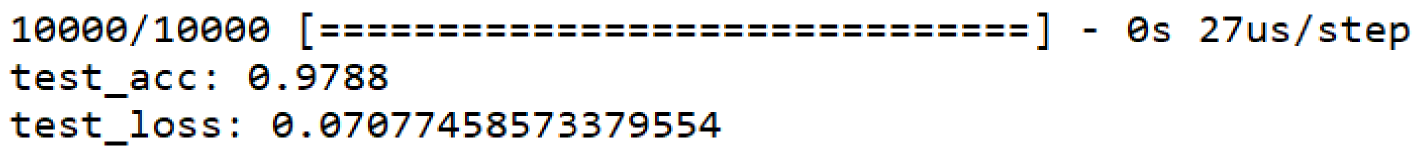
Training output:



```
Epoch 1/5
60000/60000 [==============================] - 2s 37us/step - loss: 0.2878 - acc: 0.9176
Epoch 2/5
60000/60000 [==============================] - 2s 33us/step - loss: 0.1279 - acc: 0.9631
Epoch 3/5
60000/60000 [==============================] - 2s 33us/step - loss: 0.0862 - acc: 0.9750
Epoch 4/5
60000/60000 [==============================] - 2s 33us/step - loss: 0.0642 - acc: 0.9812
Epoch 5/5
60000/60000 [==============================] - 2s 34us/step - loss: 0.0505 - acc: 0.9849
```

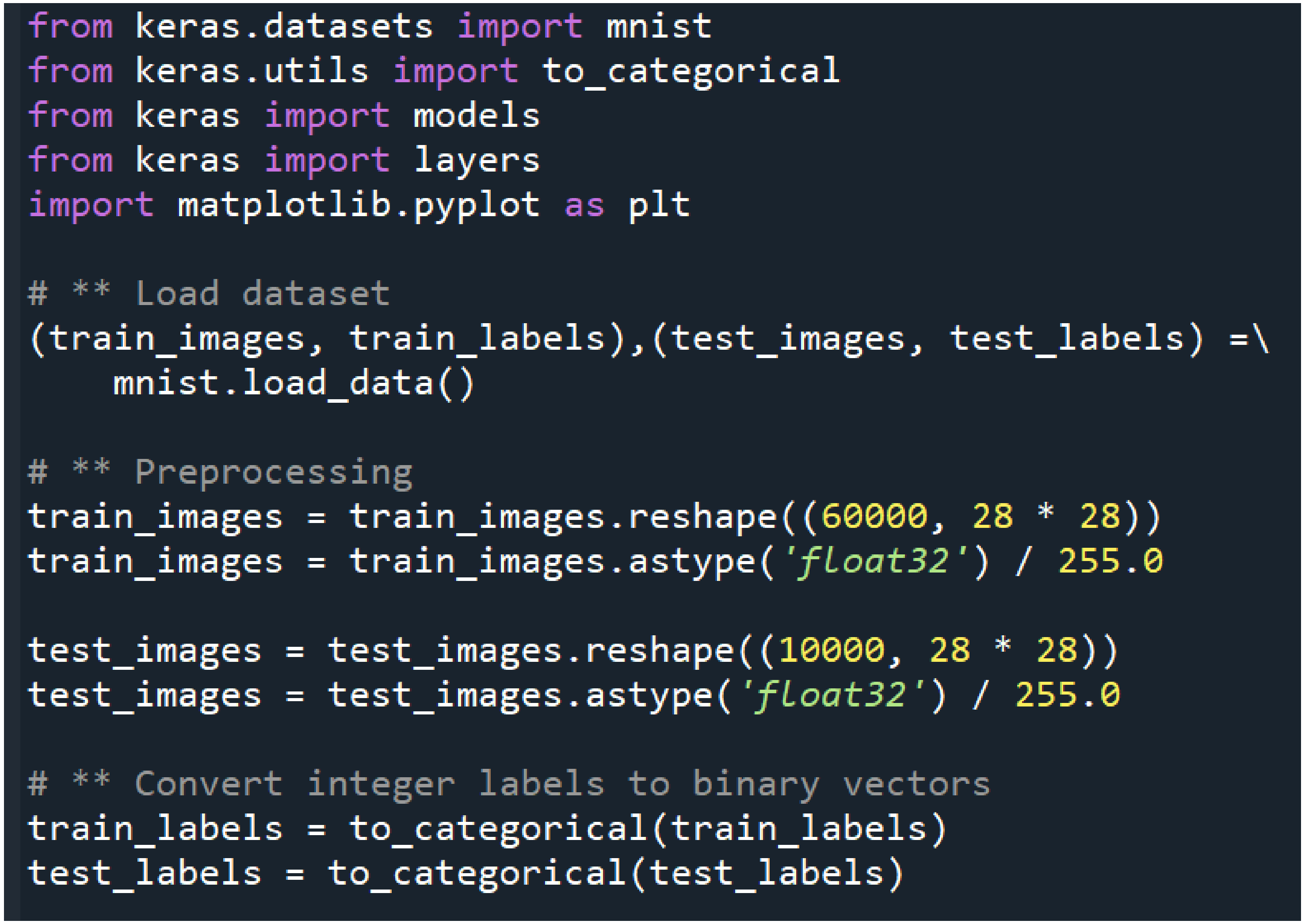
- evaluate () function and test images are used to examine performance other than training images.
- Here, according to the test results with 10.000 images, 97.88% success was achieved.
- The achievements obtained by testing are generally below the success achieved by training. If the test performance is too low compared to the training performance, the training set samples selected may be insufficient.

```
test_loss, test_acc = model.evaluate(test_images, test_labels)
print('test_acc:', test_acc)
print('test_loss:',test_loss)
```



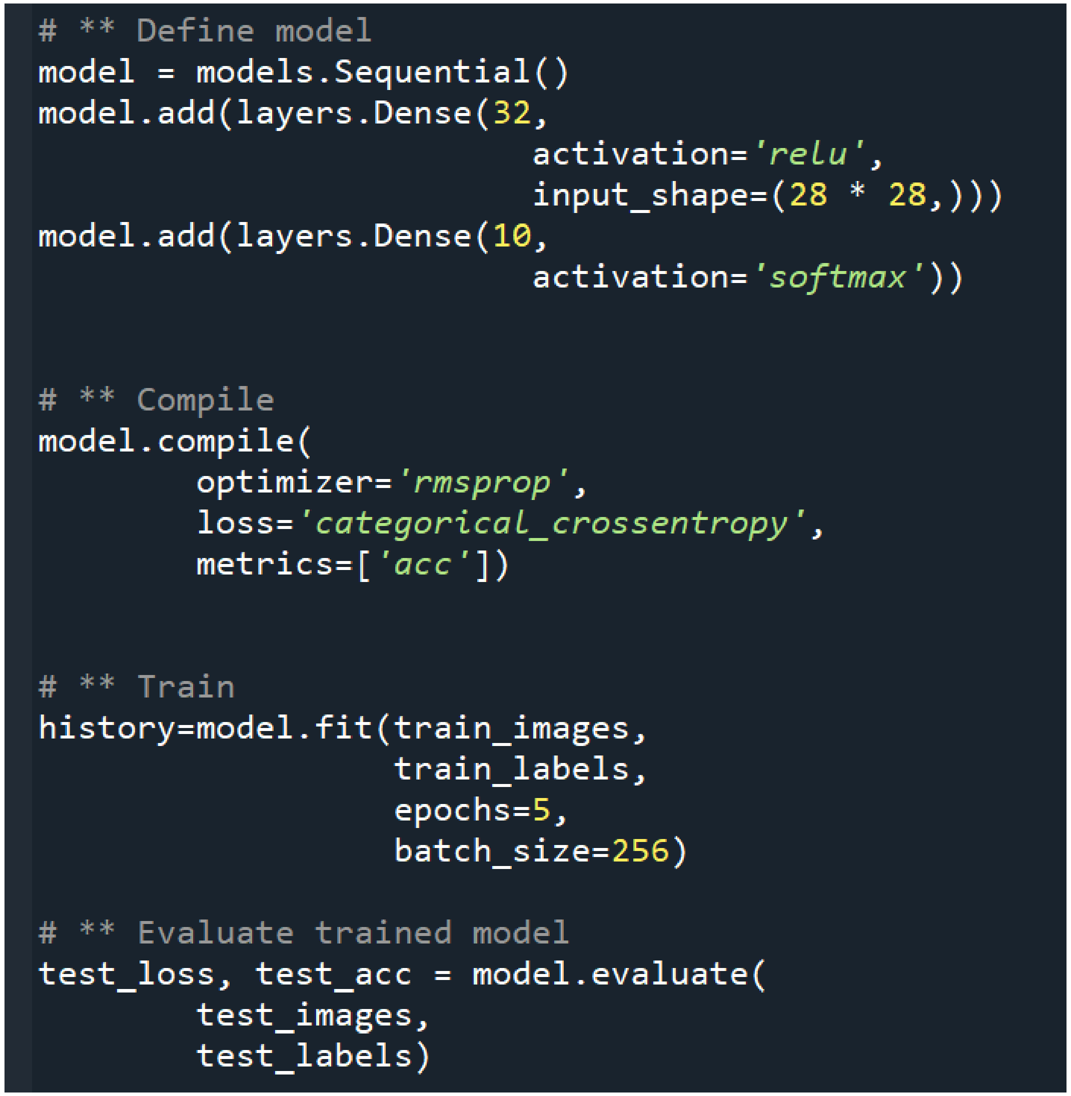
```
10000/10000 [==============================] - 0s 27us/step
test_acc: 0.9788
test_loss: 0.07077458573379554
```

Python code for training and testing

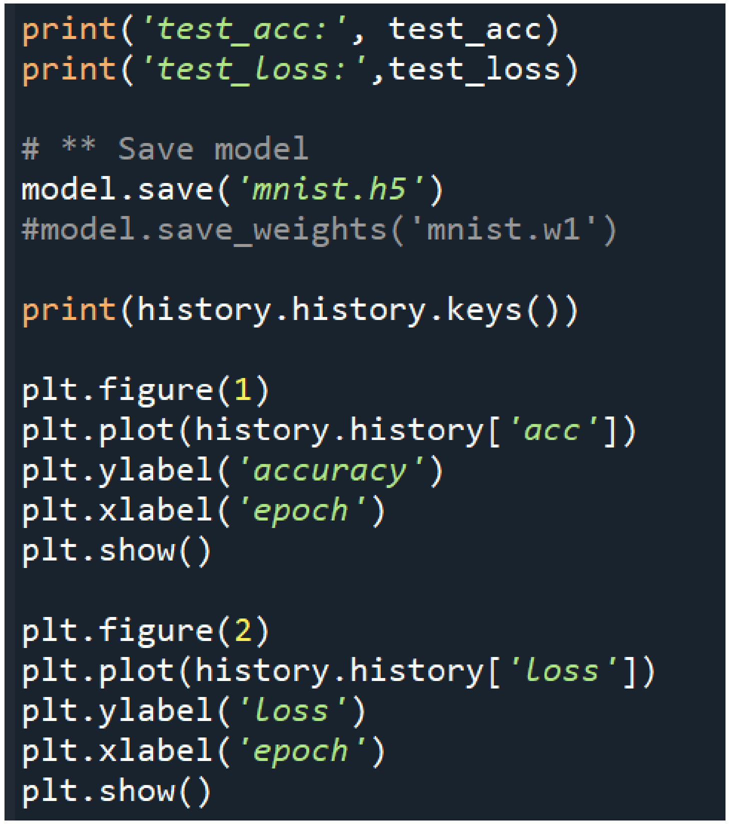


```python
from keras.datasets import mnist
from keras.utils import to_categorical
from keras import models
from keras import layers
import matplotlib.pyplot as plt

# ** Load dataset
(train_images, train_labels),(test_images, test_labels) =\
    mnist.load_data()

# ** Preprocessing
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255.0

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255.0

# ** Convert integer labels to binary vectors
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

Python code for training and testing



```python
# ** Define model
model = models.Sequential()
model.add(layers.Dense(32,
                       activation='relu',
                       input_shape=(28 * 28,)))
model.add(layers.Dense(10,
                       activation='softmax'))


# ** Compile
model.compile(
        optimizer='rmsprop',
        loss='categorical_crossentropy',
        metrics=['acc'])


# ** Train
history=model.fit(train_images,
                  train_labels,
                  epochs=5,
                  batch_size=256)

# ** Evaluate trained model
test_loss, test_acc = model.evaluate(
        test_images,
        test_labels)
```



```python
print('test_acc:', test_acc)
print('test_loss:',test_loss)

# ** Save model
model.save('mnist.h5')
#model.save_weights('mnist.w1')

print(history.history.keys())

plt.figure(1)
plt.plot(history.history['acc'])
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.show()

plt.figure(2)
plt.plot(history.history['loss'])
plt.ylabel('loss')
plt.xlabel('epoch')
plt.show()
```

Saving the trained network

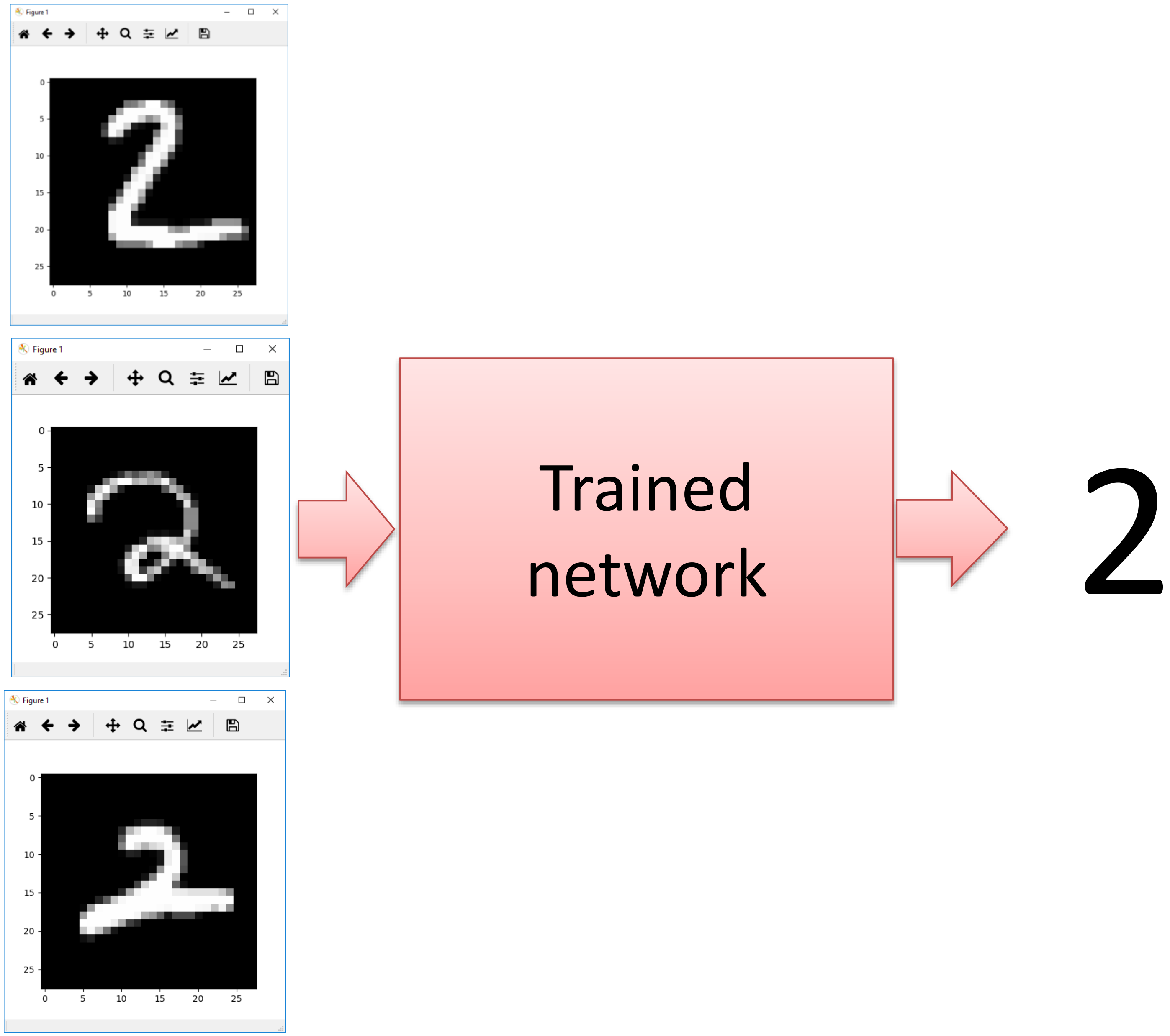
- Saving the model

```
model.save(‘file_name’)
```

- Saving the weights

```
model.save_weights(‘file_name’)
```

Using the trained network



Using the trained network

```
In [73]: from keras import models

In [74]: model=models.load_model('mnist.model1')

In [75]: (train_images, train_labels),(test_images, test_labels) = mnist.load_data()

In [76]: test_goruntu=test_images[0].reshape(1,28*28).astype('float32')/255

In [77]: y=model.predict(test_goruntu)

In [78]: y
Out[78]:
array([[3.7268353e-05, 9.8041175e-09, 1.0974355e-04, 5.2160835e-03,
        1.2340735e-07, 1.1374642e-05, 1.0773977e-09, 9.9419445e-01,
        3.5760764e-05, 3.9512845e-04]], dtype=float32)

In [79]: from numpy import argmax

In [80]: rakam=argmax(y)

In [81]: rakam
Out[81]: 7

In [82]: import matplotlib.pyplot as grafik

In [83]: grafik.imshow(test_images[0])
```

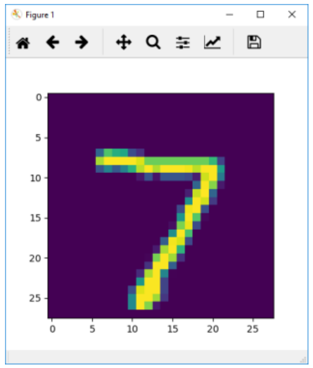


Figure 1
0
5
10
15
20
25
0
5
10
15
20
25