Artificial Intelligence

Playing with the Fashion-MNIST dataset, using Python

Fashion MNIST - Introduction

- 60K images, 10 classes
  - T-shirt/top
  - Trouser
  - Pullover
  - Dress
  - Coat
  - Sandal
  - Shirt
  - Sneaker
  - Bag
  - Ankle boot
- Each image is 28*28 = 784 pixels

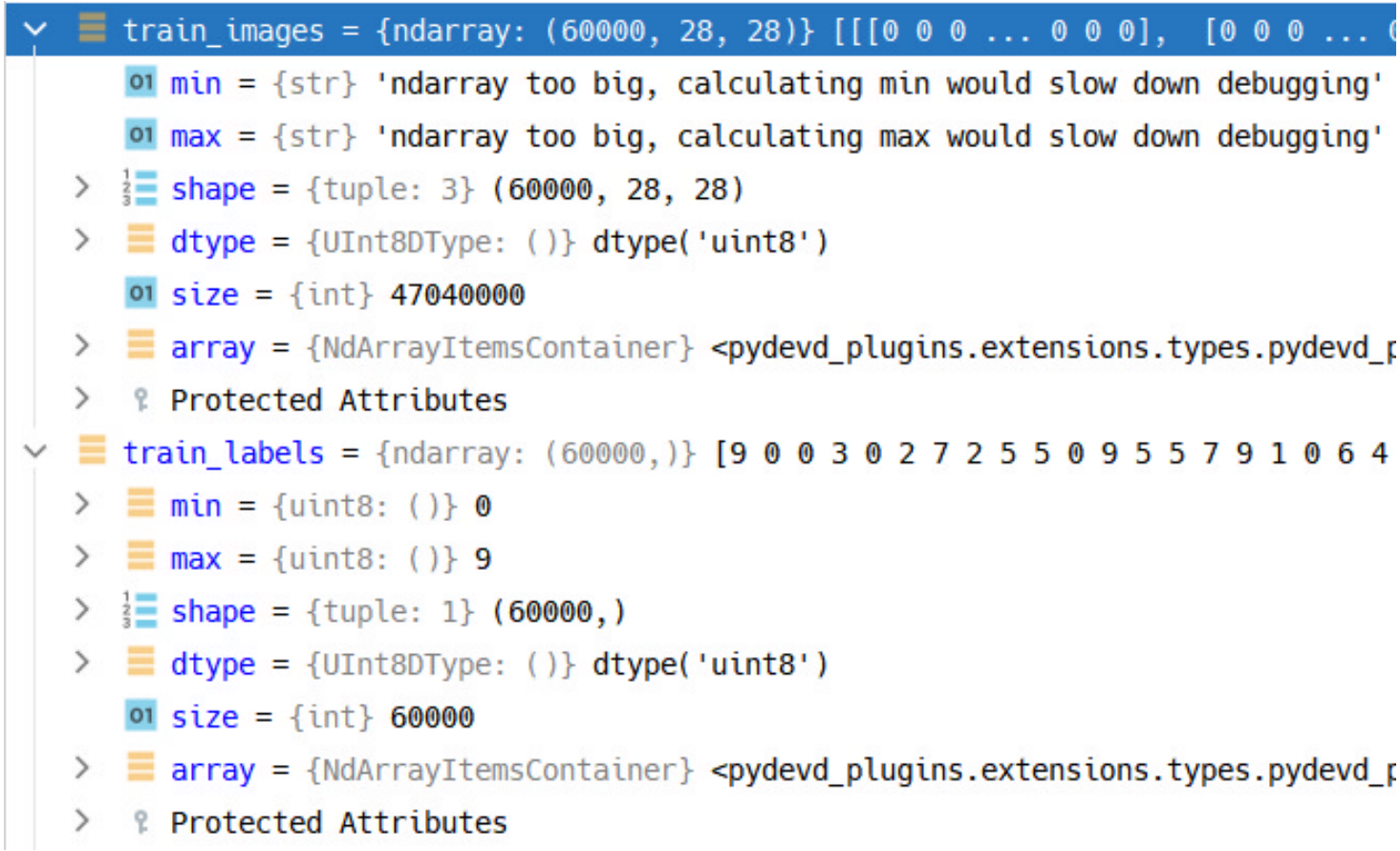
Fashion-MNIST structure - #1

- Loading the module

  fashion_mnist = tf.keras.datasets.fashion_mnist

- Building train and test tuples

  (train_images, train_labels), (test_images, test_labels) = fashion_mnist.load_data()



Fashion-MNIST structure - #2

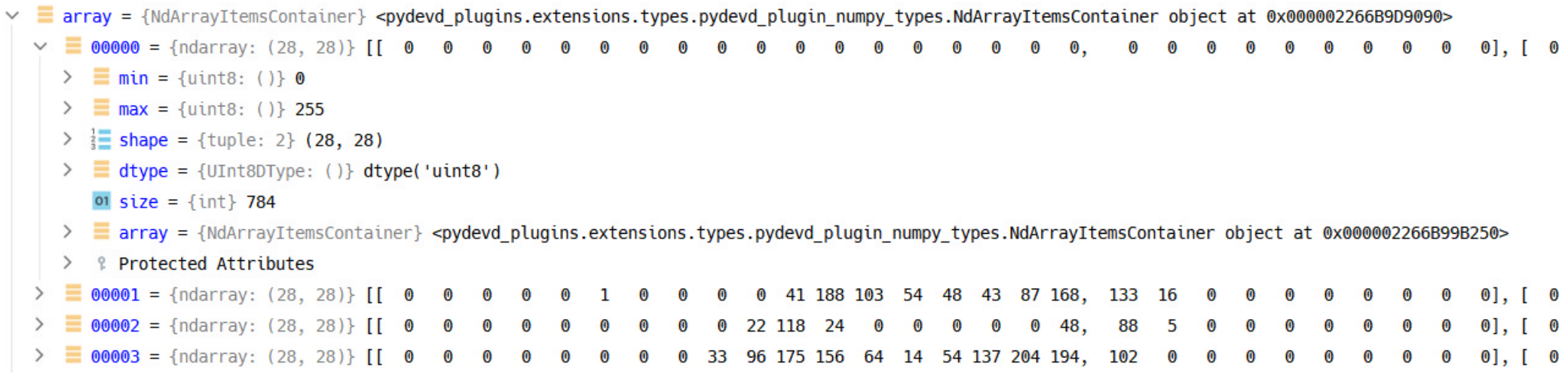
- train_images is a numpy.ndarray, indexed from zero

  type(train_images)
- It has 60000 elements

  len(train_images)
- Each element is another numpy.ndarray, shaped 28x28

  type(train_images[0]) ; train_images[0].shape
- So, each element is (again) an array of arrays, each a row of 28 ints

  len(train_images[0][0])
- test_images has the same structure, but with 10000 elements



Fashion-MNIST structure - #3

- train_labels and test_labels are also numpy.ndarrays
  - *Parallel* to train_images and test_images, respectively
- But they are 1D shaped: just integers, 0 to 9
- The value at train_labels' index i
  - is the classification of sample train_images[i]
- The value at test_labels' index i
  - is the classification of sample test_images[i]

```
train_labels = {ndarray: (60000,)} [9 0 0 3 0 2 7 2 5 5 0 9 5 5 7 9 1 0 6 4 3 1 4 8 4 3 0 2 4 4 5 3 6 6 0 8 5, 2 1 6
  min = {uint8: ()} 0
  max = {uint8: ()} 9
  shape = {tuple: 1} (60000,)
  dtype = {UInt8DType: ()} dtype('uint8')
test_labels = {ndarray: (10000,)} [9 2 1 1 6 1 4 6 5 7 4 5 7 3 4 1 2 4 8 0 2 5 7 9 1 4 6 0 9 3 8 8 3 3 8 0 7,
  min = {uint8: ()} 0
  max = {uint8: ()} 9
  shape = {tuple: 1} (10000,)
  dtype = {UInt8DType: ()} dtype('uint8')
  size = {int} 10000
```

Fashion-MNIST code - pre-processing with normalization

- # Normalize the pixel values of the train and test images
- train_images = train_images / 255.0 # Brodcasting ; test_images = test_images / 255.0

```
train_images = {ndarray: (60000, 28, 28)} [[[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
  min = {str} 'ndarray too big, calculating min would slow down debugging'
  max = {str} 'ndarray too big, calculating max would slow down debugging'
> shape = {tuple: 3} (60000, 28, 28)
> dtype = {UInt8DType: ()} dtype('uint8')
  size = {int} 47040000
v array = {NdArrayItemsContainer} <pydevd_plugins.extensions.types.pydevd_plugin_numpy_types.NdArrayItemsContainer object at 0x00002266B860790>
  > 00000 = {ndarray: (28, 28)} [[  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
  > 00001 = {ndarray: (28, 28)} [[  0   0   0   0   0   1   0   0   0   0  41 188 103  54  48  43  87 168,  133  16   0   0   0   0   0   0   0   0   0], [  0
  > 00002 = {ndarray: (28, 28)} [[  0   0   0   0   0   0   0   0   0  22 118  24   0   0   0   0   0  48,   88   5   0   0   0   0   0   0   0   0   0], [  0
  > 00003 = {ndarray: (28, 28)} [[  0   0   0   0   0   0   0   0  33  96 175 156  64  14  54 137 204 194,  102   0   0   0   0   0   0   0   0   0   0], [  0
  > 00004 = {ndarray: (28, 28)} [[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  26,   0   0   0   0   0   0   0   0   0   0], [  0
  > 00005 = {ndarray: (28, 28)} [[  0   0   0   0   1   0   0   0   0  22  88 188 172 132 125 141 199 143,   9   0   0   0   1   0   0   0   0   0], [  0
  > 00006 = {ndarray: (28, 28)} [[  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
  > 00007 = {ndarray: (28, 28)} [[  0   0   0   0   0   1   1   0   0   0   0  63  28   0   0   0  33  85,   0   0   0   0   0   0   0   0   0   0   0], [  0
  > 00008 = {ndarray: (28, 28)} [[  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
  min = {str} 'ndarray too big, calculating min would slow down debugging'
  max = {str} 'ndarray too big, calculating max would slow down debugging'
> shape = {tuple: 3} (60000, 28, 28)
> dtype = {Float64DType: ()} dtype('float64')
  size = {int} 47040000
v array = {NdArrayItemsContainer} <pydevd_plugins.extensions.types.pydevd_plugin_numpy_types.NdArrayItemsContainer object at 0x00002266B9D8550>
  > 00000 = {ndarray: (28, 28)} [[0.         0.         0.         0.         0.         0.,  0.         0.         0.         0.         0.         0.,  0
  > 00001 = {ndarray: (28, 28)} [[0.         0.         0.         0.         0.         0.00392157,  0.         0.         0.         0.         0.1607843
  > 00002 = {ndarray: (28, 28)} [[0.         0.         0.         0.         0.         0.,  0.         0.         0.         0.08627451 0.4627451  0.0941
  > 00003 = {ndarray: (28, 28)} [[0.         0.         0.         0.         0.         0.,  0.         0.         0.12941176 0.37647059 0.68627451 0.6117
  > 00004 = {ndarray: (28, 28)} [[0.         0.         0.         0.         0.         0.,  0.         0.         0.         0.         0.         0.,  0
  > 00005 = {ndarray: (28, 28)} [[0.         0.         0.         0.         0.00392157 0.,  0.         0.         0.         0.08627451 0.34509804 0.7372
  > 00006 = {ndarray: (28, 28)} [[0.         0.         0.         0.         0.         0.,  0.         0.         0.         0.         0.         0.,  0
  > 00007 = {ndarray: (28, 28)} [[0.         0.         0.         0.         0.         0.00392157,  0.00392157 0.         0.         0.         0.
  > 00008 = {ndarray: (28, 28)} [[0.         0.         0.         0.         0.         0.,  0.         0.         0.         0.         0.         0.,  0
```

Fashion-MNIST code - pre-processing with normalization

- Normalization is a preprocessing step:
  - Divide each value by the max possible value (in this case 2^8=256, ranging 0 .. 255)
  - By dividing the pixel values of the images in the Fashion MNIST dataset by 255.0, values are scaled down to a range of 0 to 1
- Improves Gradient Descent Efficiency: Neural networks use gradient descent as an optimization algorithm to minimize the loss function. Normalizing the input data to a range between 0 and 1 can make the training process faster and more stable by ensuring that the gradient descent algorithm doesn't have to deal with values that are too large.
- Ensures Consistent Input Scale: When the features (in this case, pixel values) are on the same scale, it helps the model learn more effectively. Without normalization, the disparity in the scale of the input features could bias the model and impede the learning process.
- Reduces Computational Complexity: Working with smaller, normalized values can also reduce the numerical computation burden, making operations faster and less resource-intensive.
- Improves Convergence: Normalizing inputs can lead to a better convergence of the model during training, as it ensures that all inputs (pixel values) contribute equally to the learning process.

Fashion-MNIST : ANN + APP

- At repo: https://github.com/amsm/am_fashion_mnist
- There is an ANN (Artificial Neural Network) for Fashion-MNIST based classification
  - 3 layers only
  - for creating models trained on the Fashion-MNIST dataset
  - and saving them to a .h5 model file format
    - or, instead, to a folder in "TensorFlow SavedModel Format"
- Also, contains a Web app for uploading a file and see its classification
  - that can reuse the previously-trained model
    - Loading it from file or from folder
  - that accepts image uploads
  - and responds a probabilistic classification for the file
    - one of the 10 Fashion-MNIST classes
  - it will perform horribly bad with everyday pictures
    - because no effort has been done to enrich Fashion-MNIST, or even scale-up the existing data
  - it will perform great with pics from Fashion-MNIST itself

References

- https://github.com/zalandoresearch/fashion-mnist
- https://github.com/amsm/am_fashion_mnist