1000-719bMSB Modeling of Complex Biological Systems

Deep Neural Network: Supervised Learning

Basic python and pandas

https://www.kaggle.com/lavanyashukla01/pandas-numpy-python-cheatsheet

https://www.utc.fr/~jlaforet/Suppl/python-cheatsheets.pdf

List comprehensions are a concise way to create new lists from existing ones.

```
list1 = list(range(0,10))
print(list1)
```

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

```
list1[0] # A vector in R starts with an index of 1. In Python, 0.
```

```
0
```

```
list1[2:5]
```

```
[2, 3, 4]
```

```
list2 = []
for i in list1:
    list2.append(i+1)

print(list2)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
list3 = [i+1 for i in list1]

print(list3)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Classification of MNIST using densely connected layers

We are going to use the Keras library to implement a neural network that can classify handwritten digits - in just a few lines of code.

First we load and inspect the data. The dataset is split into training and test data.

```
import numpy as np
import tensorflow.keras as keras
import matplotlib.pyplot as plt


import tensorflow as tf
print(tf.__version__)
tf.compat.v1.disable_eager_execution()
```

```
2.15.0
```

```
(train_images, train_labels), (test_images, test_labels) = keras.datasets.fashion_mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
29515/29515 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-images-idx3-ubyte.gz
26421880/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 [==============================] - 0s 0us/step
```

```
train_images.shape
```

```
(60000, 28, 28)
```

```
train_labels.shape
```

```
(60000,)
```

```
test_images.shape
```

```
(10000, 28, 28)
```

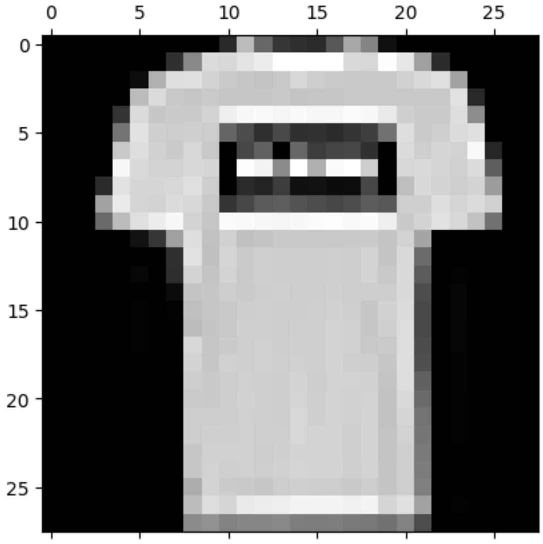
```
test_labels.shape
```

```
(10000,)
```

Let's plot one of the digits and the corresponding label.

```
print('Label of element 0:',train_labels[1])
plt.matshow(train_images[1], cmap='gray')
plt.show()
```

```
Label of element 0: 0
```



In this step we define the neural network. ReLu is an activation function defined as f(x) = max(0,x).

Softmax activation function is normalized such that the sum of all outputs is equal 1.

```
from tensorflow.keras import layers
from tensorflow.keras import models
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
model.add(layers.Dense(10, activation='softmax'))
```

With compile we tell the network which optimizer and loss function to use. Optimizer specifies the particular implementation of the gradient-descent, e.g. how it adapts the learning rate. 'Metrics' specifies the output during the training.

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

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 512)               401920

 dense_1 (Dense)             (None, 10)                5130

=================================================================
Total params: 407050 (1.55 MB)
Trainable params: 407050 (1.55 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

We are using a densely connected network, so we have to flatten the images.
Input values should be in the range (0,1) for fast convergence.

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

Convert the labels to a 'one-hot' coding.

```
from tensorflow.keras.utils import to_categorical
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

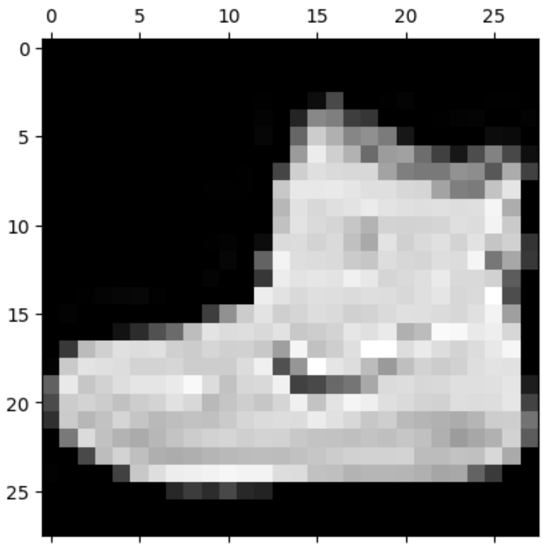
```
train_images.reshape((60000,28*28)).shape
```

```
(60000, 784)
```

```
train_labels[0]
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 1.], dtype=float32)
```

```
plt.matshow(train_images[0], cmap='gray')
plt.show()
```



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

```
Train on 60000 samples
Epoch 1/5
60000/60000 [==============================] - 5s 88us/sample - loss: 0.0317 - accuracy: 0.7789
Epoch 2/5
60000/60000 [==============================] - 5s 90us/sample - loss: 0.0208 - accuracy: 0.8576
Epoch 3/5
60000/60000 [==============================] - 5s 91us/sample - loss: 0.0185 - accuracy: 0.8734
Epoch 4/5
60000/60000 [==============================] - 4s 75us/sample - loss: 0.0171 - accuracy: 0.8834
Epoch 5/5
60000/60000 [==============================] - 6s 96us/sample - loss: 0.0163 - accuracy: 0.8896
<keras.src.callbacks.History at 0x7e715b9af790>
```

Let's check the performance on the test set. If the accuracy is less than the training accuracy, then we might be overfitting!

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

```
/usr/local/lib/python3.10/dist-packages/keras/src/engine/training_v1.py:2335: UserWarning: `Model.state_updates` will be removed in
  updates = self.state_updates
test_acc: 0.8743
```

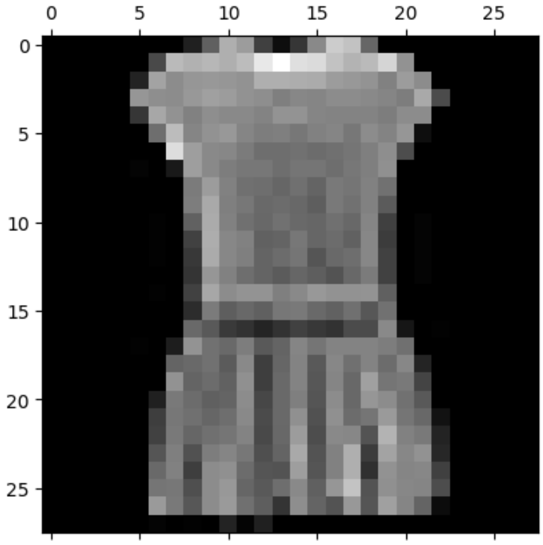
We can also find the predictions for a selection of input images.

```
predictions = model.predict(train_images_flat[:10])
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/engine/training_v1.py:2359: UserWarning: `Model.state_updates` will be removed in
  updates=self.state_updates,
```

```
img_num = 3
print(predictions[img_num])
print(train_labels[img_num])
plt.matshow(train_images[img_num], cmap='gray')
plt.show()
```

```
[2.2515472e-02 2.1882821e-03 6.1906874e-04 9.0620458e-01 3.2007782e-05
 9.6665724e-05 6.8153732e-02 5.2839969e-06 1.8312993e-04 1.8089752e-06]
[0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
```




```
train_labels
```

```
array([[0., 0., 0., ..., 0., 0., 1.],
       [1., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
```

Classification of MNIST using convolutional layers

We have build a classifier for handwritten images only using densely connected layers. Let's see if we can do better using convolutional layers!

First define the convolutional layers.

```
model2 = models.Sequential()
model2.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model2.add(layers.MaxPooling2D((2, 2)))
model2.add(layers.Conv2D(128, (3, 3), activation='relu'))
model2.add(layers.MaxPooling2D((2, 2)))
model2.add(layers.Conv2D(128, (3, 3), activation='relu'))
```

```
model2.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2  (None, 13, 13, 32)        0
 D)

 conv2d_1 (Conv2D)           (None, 11, 11, 128)       36992

 max_pooling2d_1 (MaxPoolin  (None, 5, 5, 128)         0
```

```
 g2D)

 conv2d_2 (Conv2D)           (None, 3, 3, 128)         147584

=================================================================
Total params: 184896 (722.25 KB)
Trainable params: 184896 (722.25 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Now add a classifier on top of the convnet.

```
model2.add(layers.Flatten())
model2.add(layers.Dense(64, activation='relu'))
model2.add(layers.Dense(10, activation='softmax'))
```

```
model2.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2  (None, 13, 13, 32)        0
 D)

 conv2d_1 (Conv2D)           (None, 11, 11, 128)       36992

 max_pooling2d_1 (MaxPoolin  (None, 5, 5, 128)         0
 g2D)

 conv2d_2 (Conv2D)           (None, 3, 3, 128)         147584

 flatten (Flatten)           (None, 1152)              0

 dense_2 (Dense)             (None, 64)                73792

 dense_3 (Dense)             (None, 10)                650

=================================================================
Total params: 259338 (1013.04 KB)
Trainable params: 259338 (1013.04 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
train_images_conv = train_images.reshape((60000, 28, 28, 1))
train_images_conv = train_images_conv.astype('float32') / 255
test_images_conv = test_images.reshape((10000, 28, 28, 1))
test_images_conv = test_images_conv.astype('float32') / 255
```

```
model2.compile(optimizer='rmsprop',
loss='categorical_crossentropy',
metrics=['accuracy'])
```

```
model2.fit(train_images_conv, train_labels, epochs=15, batch_size=64)
```

```
Train on 60000 samples
Epoch 1/15
60000/60000 [==============================] - 97s 2ms/sample - loss: 0.4981 - accuracy: 0.8155
Epoch 2/15
60000/60000 [==============================] - 96s 2ms/sample - loss: 0.3030 - accuracy: 0.8895
Epoch 3/15
60000/60000 [==============================] - 98s 2ms/sample - loss: 0.2562 - accuracy: 0.9065
Epoch 4/15
60000/60000 [==============================] - 94s 2ms/sample - loss: 0.2261 - accuracy: 0.9158
Epoch 5/15
60000/60000 [==============================] - 94s 2ms/sample - loss: 0.2026 - accuracy: 0.9264
Epoch 6/15
60000/60000 [==============================] - 93s 2ms/sample - loss: 0.1827 - accuracy: 0.9324
Epoch 7/15
60000/60000 [==============================] - 94s 2ms/sample - loss: 0.1644 - accuracy: 0.9401
Epoch 8/15
60000/60000 [==============================] - 93s 2ms/sample - loss: 0.1488 - accuracy: 0.9452
Epoch 9/15
60000/60000 [==============================] - 93s 2ms/sample - loss: 0.1367 - accuracy: 0.9505
Epoch 10/15
60000/60000 [==============================] - 93s 2ms/sample - loss: 0.1254 - accuracy: 0.9544
Epoch 11/15
60000/60000 [==============================] - 91s 2ms/sample - loss: 0.1159 - accuracy: 0.9584
Epoch 12/15
60000/60000 [==============================] - 93s 2ms/sample - loss: 0.1084 - accuracy: 0.9606
Epoch 13/15
```

```
60000/60000 [==============================] - 95s 2ms/sample - loss: 0.1017 - accuracy: 0.9635
Epoch 14/15
60000/60000 [==============================] - 96s 2ms/sample - loss: 0.0960 - accuracy: 0.9652
Epoch 15/15
60000/60000 [==============================] - 98s 2ms/sample - loss: 0.0915 - accuracy: 0.9665
<keras.src.callbacks.History at 0x7e714e76be50>
```

```
test_loss, test_acc = model2.evaluate(test_images_conv, test_labels)
print(test_acc)
```

```
0.9113
```

Introducing Fashion MNIST (Homework dataset)

The MNIST dataset is not too demanding, let's try something a little more difficult - Fashion MNIST.

LINK TO IMAGE

Check out labels on GitHub

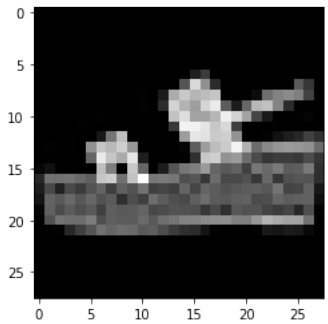
```
(train_imgs_fash, train_labels_fash), (test_imgs_fash, test_labels_fash) = keras.datasets.fashion_mnist.load_data()
```

```
train_imgs_fash.shape
```

```
(60000, 28, 28)
```

```
plt.imshow(train_imgs_fash[12],  cmap=plt.get_cmap('gray'))
```

```
<matplotlib.image.AxesImage at 0x7fd36038a850>
```



HOMEWORK 1

Build a classifier for fashion MNIST.

1. Use exactly the same architectures (both densely connected layers and from convolutional layers) as the above MNIST e.g., replace the dataset. Save the Jupyter Notebook in its original format and output a PDF file after training, testing, and validation. Make sure to write down how do they perform (training accuracny, testing accuracy).
2. Improve the architecture. Experiment with different numbers of layers, size of layers, number of filters, size of filters. You are required to make those adjustment to get the highest accuracy. Watch out for overfitting -- we want the highest testing accuracy! Please provide a PDF file of the result, the best test accuracy and the architecture (different numbers of layers, size of layers, number of filters, size of filters)

Visualizing Filter Response

We use gradient descent in input space to display the visual pattern each filter is maximally responsive to.
To this end we take a VGG19 convnet pretrained on the ImageNet dataset.

Very Deep Convolutional Networks for Large-Scale Image Recognition Karen Simonyan, Andrew Zisserman

DL Architecture

```
from tensorflow.keras.applications import VGG19
from tensorflow.keras import backend as K
import numpy as np

import matplotlib.pyplot as plt

#Load pretrained model
```

```
#we omit the densely connected layers of the network
model = VGG19(weights='imagenet', include_top=False)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg19/vgg19_weights_tf_dim_ordering_tf_kernels_no
80134624/80134624 [==============================] - 1s 0us/step
```

```
model.summary()
```

```
Model: "vgg19"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_1 (InputLayer)        [(None, None, None, 3)]   0         
                                                                 
 block1_conv1 (Conv2D)       (None, None, None, 64)    1792      
                                                                 
 block1_conv2 (Conv2D)       (None, None, None, 64)    36928     
                                                                 
 block1_pool (MaxPooling2D)  (None, None, None, 64)    0         
                                                                 
 block2_conv1 (Conv2D)       (None, None, None, 128)   73856     
                                                                 
 block2_conv2 (Conv2D)       (None, None, None, 128)   147584    
                                                                 
 block2_pool (MaxPooling2D)  (None, None, None, 128)   0         
                                                                 
 block3_conv1 (Conv2D)       (None, None, None, 256)   295168    
                                                                 
 block3_conv2 (Conv2D)       (None, None, None, 256)   590080    
                                                                 
 block3_conv3 (Conv2D)       (None, None, None, 256)   590080    
                                                                 
 block3_conv4 (Conv2D)       (None, None, None, 256)   590080    
                                                                 
 block3_pool (MaxPooling2D)  (None, None, None, 256)   0         
                                                                 
 block4_conv1 (Conv2D)       (None, None, None, 512)   1180160   
                                                                 
 block4_conv2 (Conv2D)       (None, None, None, 512)   2359808   
                                                                 
 block4_conv3 (Conv2D)       (None, None, None, 512)   2359808   
                                                                 
 block4_conv4 (Conv2D)       (None, None, None, 512)   2359808   
                                                                 
 block4_pool (MaxPooling2D)  (None, None, None, 512)   0         
                                                                 
 block5_conv1 (Conv2D)       (None, None, None, 512)   2359808   
                                                                 
 block5_conv2 (Conv2D)       (None, None, None, 512)   2359808   
                                                                 
 block5_conv3 (Conv2D)       (None, None, None, 512)   2359808   
                                                                 
 block5_conv4 (Conv2D)       (None, None, None, 512)   2359808   
                                                                 
 block5_pool (MaxPooling2D)  (None, None, None, 512)   0         
                                                                 
=================================================================
Total params: 20024384 (76.39 MB)
Trainable params: 20024384 (76.39 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

```
#Specify filter you want to visualize and get its output
layer_name = 'block5_conv3'
filter_index = 3
layer_output = model.get_layer(layer_name).output

#Loss is the averaged activation of the chosen filter
loss = K.mean(layer_output[:, :, :, filter_index])


#Gradients of loss with respect to the input
#upgrading to 2.x: tf.gradients is no longer supported
#requiring tf.compat.v1.disable_eager_execution()
grads = K.gradients(loss, model.input)[0]

#A trick is to normalize the gradients by their L2 norm
#This ensures that the magnitude of the gradients is always in the same range
#and leads to a smooth descent process
grads /= (K.sqrt(K.mean(K.square(grads))) + 1e-5)


#The tensors defined so far (loss, grads) were symbolic
#To obtain values we need to feed an input via K.function
```

```
iterate = K.function([model.input], [loss, grads])
loss_value, grads_value = iterate([np.zeros((1, 150, 150, 3))])


print(grads)
print(grads_value)
```

```
Tensor("truediv:0", shape=(None, None, None, 3), dtype=float32)
[[[[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.]]]]
```

```
#Implement the actual gradient descent
#Initial input is a grey image with some noise

input_img_data = np.random.random((1, 150, 150, 3)) * 20 + 128.
step = 1.
for i in range(40):
    loss_value, grads_value = iterate([input_img_data])
    input_img_data += grads_value * step


print(grads_value)
```

```
[[[[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.]]]]
```

```
#Postprocess to turn into displayable image

def deprocess_image(x):
    x -= x.mean()
    x /= (x.std() + 1e-5)
    x *= 0.1

    x += 0.5
    x = np.clip(x, 0, 1)

    x *= 255
    x = np.clip(x, 0, 255).astype('uint8')
    return x


plt.imshow(deprocess_image(input_img_data[0]))
```

```
<matplotlib.image.AxesImage at 0x7e712c7fb820>
```