In [ ]:

Import the relevant packages

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

# TensorFLow includes a data provider for MNIST that we'll use.
# It comes with the tensorflow-datasets module, therefore, if you haven't please install the package using
# pip install tensorflow-datasets
# or
# conda install tensorflow-datasets

import tensorflow_datasets as tfds

# these datasets will be stored in C:\Users\*USERNAME*\tensorflow_datasets\...
# the first time you download a dataset, it is stored in the respective folder
# every other time, it is automatically loading the copy on your computer
```

Data

In [2]:
```
# remember the comment from above
# these datasets will be stored in C:\Users\*USERNAME*\tensorflow_datasets\...
# the first time you download a dataset, it is stored in the respective folder
# every other time, it is automatically loading the copy on your computer

# tfds.load actually loads a dataset (or downloads and then loads if that's the first time you use it)
# in our case, we are interesteed in the MNIST; the name of the dataset is the only mandatory argument
# there are other arguments we can specify, which we can find useful
# mnist_dataset = tfds.load(name='mnist', as_supervised=True)
mnist_dataset, mnist_info = tfds.load(name='mnist', with_info=True, as_supervised=True)
# with_info=True will also provide us with a tuple containing information about the version, features, number of samples
# we will use this information a bit below and we will store it in mnist_info

# as_supervised=True will load the dataset in a 2-tuple structure (input, target)
# alternatively, as_supervised=False, would return a dictionary
# obviously we prefer to have our inputs and targets separated

# once we have loaded the dataset, we can easily extract the training and testing dataset with the built references
mnist_train, mnist_test = mnist_dataset['train'], mnist_dataset['test']

# by default, TF has training and testing datasets, but no validation sets
# thus we must split it on our own

# we start by defining the number of validation samples as a % of the train samples
# this is also where we make use of mnist_info (we don't have to count the observations)
num_validation_samples = 0.1 * mnist_info.splits['train'].num_examples
# let's cast this number to an integer, as a float may cause an error along the way
num_validation_samples = tf.cast(num_validation_samples, tf.int64)

# let's also store the number of test samples in a dedicated variable (instead of using the mnist_info one)
num_test_samples = mnist_info.splits['test'].num_examples
# once more, we'd prefer an integer (rather than the default float)
num_test_samples = tf.cast(num_test_samples, tf.int64)


# normally, we would like to scale our data in some way to make the result more numerically stable
# in this case we will simply prefer to have inputs between 0 and 1
# let's define a function called: scale, that will take an MNIST image and its label
def scale(image, label):
    # we make sure the value is a float
    image = tf.cast(image, tf.float32)
    # since the possible values for the inputs are 0 to 255 (256 different shades of grey)
    # if we divide each element by 255, we would get the desired result -> all elements will be between 0 and 1
    image /= 255.

    return image, label


# the method .map() allows us to apply a custom transformation to a given dataset
# we have already decided that we will get the validation data from mnist_train, so
scaled_train_and_validation_data = mnist_train.map(scale)

# finally, we scale and batch the test data
# we scale it so it has the same magnitude as the train and validation
# there is no need to shuffle it, because we won't be training on the test data
# there would be a single batch, equal to the size of the test data
test_data = mnist_test.map(scale)


# let's also shuffle the data

BUFFER_SIZE = 10000
# this BUFFER_SIZE parameter is here for cases when we're dealing with enormous datasets
# then we can't shuffle the whole dataset in one go because we can't fit it all in memory
# so instead TF only stores BUFFER_SIZE samples in memory at a time and shuffles them
# if BUFFER_SIZE=1 => no shuffling will actually happen
# if BUFFER_SIZE >= num samples => shuffling is uniform
# BUFFER_SIZE in between - a computational optimization to approximate uniform shuffling

# luckily for us, there is a shuffle method readily available and we just need to specify the buffer size
shuffled_train_and_validation_data = scaled_train_and_validation_data.shuffle(BUFFER_SIZE)

# once we have scaled and shuffled the data, we can proceed to actually extracting the train and validation
# our validation data would be equal to 10% of the training set, which we've already calculated
# we use the .take() method to take that many samples
# finally, we create a batch with a batch size equal to the total number of validation samples
validation_data = shuffled_train_and_validation_data.take(num_validation_samples)

# similarly, the train_data is everything else, so we skip as many samples as there are in the validation dataset
train_data = shuffled_train_and_validation_data.skip(num_validation_samples)

# determine the batch size
BATCH_SIZE = 100

# we can also take advantage of the occasion to batch the train data
# this would be very helpful when we train, as we would be able to iterate over the different batches
train_data = train_data.batch(BATCH_SIZE)

validation_data = validation_data.batch(num_validation_samples)

# batch the test data
test_data = test_data.batch(num_test_samples)


# takes next batch (it is the only batch)
# because as_supervized=True, we've got a 2-tuple structure
validation_inputs, validation_targets = next(iter(validation_data))
```

Outline the model

In [3]:
```
input_size = 784
output_size = 10
# Use same hidden layer size for both hidden layers. Not a necessity.
hidden_layer_size = 50

# define how the model will look like
model = tf.keras.Sequential([

    # the first layer (the input layer)
    # each observation is 28x28x1 pixels, therefore it is a tensor of rank 3
    # since we don't know CNNs yet, we don't know how to feed such input into our net, so we must flatten the images
    # there is a convenient method 'Flatten' that simply takes our 28x28x1 tensor and orders it into a (None,)
    # or (28x28x1,) = (784,) vector
    # this allows us to actually create a feed forward neural network
    tf.keras.layers.Flatten(input_shape=(28, 28, 1)), # input layer

    # tf.keras.layers.Dense is basically implementing: output = activation(dot(input, weight) + bias)
    # it takes several arguments, but the most important ones for us are the hidden_layer_size and the activation function
    tf.keras.layers.Dense(hidden_layer_size, activation='relu'), # 1st hidden layer
    tf.keras.layers.Dense(hidden_layer_size, activation='relu'), # 2nd hidden layer

    # the final layer is no different, we just make sure to activate it with softmax
    tf.keras.layers.Dense(output_size, activation='softmax') # output layer
])
```

```
C:\Users\fady\anaconda3\Lib\site-packages\keras\src\layers\reshaping\flatten.py:37: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the f
irst layer in the model instead.
  super().__init__(**kwargs)
```

Choose the optimizer and the loss function

In [4]:
```
# we define the optimizer we'd like to use,
# the loss function,
# and the metrics we are interested in obtaining at each iteration
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

Model Training

In [5]:
```
# determine the maximum number of epochs
NUM_EPOCHS = 5

# we fit the model, specifying the
# training data
# the total number of epochs
# and the validation data we just created ourselves in the format: (inputs,targets)
model.fit(train_data, epochs=NUM_EPOCHS, validation_data=(validation_inputs, validation_targets), verbose =2)
```

```
Epoch 1/5
540/540 - 3s - 5ms/step - accuracy: 0.8837 - loss: 0.4103 - val_accuracy: 0.9378 - val_loss: 0.2198
Epoch 2/5
540/540 - 2s - 3ms/step - accuracy: 0.9464 - loss: 0.1842 - val_accuracy: 0.9520 - val_loss: 0.1620
Epoch 3/5
540/540 - 2s - 3ms/step - accuracy: 0.9591 - loss: 0.1386 - val_accuracy: 0.9637 - val_loss: 0.1302
Epoch 4/5
540/540 - 2s - 3ms/step - accuracy: 0.9667 - loss: 0.1143 - val_accuracy: 0.9658 - val_loss: 0.1176
Epoch 5/5
540/540 - 2s - 3ms/step - accuracy: 0.9716 - loss: 0.0948 - val_accuracy: 0.9743 - val_loss: 0.0938
```

Out[5]:
```
<keras.src.callbacks.history.History at 0x219eebe1a00>
```

Test the model

In [6]:
```
test_loss, test_accuracy = model.evaluate(test_data)

# We can apply some nice formatting if we want to
print('Test loss: {0:.2f}. Test accuracy: {1:.2f}%'.format(test_loss, test_accuracy*100.))
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 218ms/step - accuracy: 0.9691 - loss: 0.1035
Test loss: 0.10. Test accuracy: 96.91%
```

In [ ]:

In [ ]: