assignment_10_solution

April 26, 2023

Please fill in your name and that of your teammate.

You:

Teammate:

[1]:
```
%matplotlib inline
# %pdb on
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
sns.set(rc={'figure.figsize':(8,6)}, style="whitegrid")
```

1 Introduction

Welcome to the tenth lab. Neural networks are more a class of tools than a single tool, though the foundation you built last week should enable you to understand what is going on here without too much trouble.

There is relatively little coding this week, which is unfortunate: we are starting to touch topics that require more than a lab's worth of practice to achieve basic proficiency. Rather than overloading you of work, this week we focus a bit more on foundations and give you time to study; then we should hit more interesting and fun applications over the next lectures with Deep Learning and Reinforcement Learning.

1.0.1 How to pass the lab?

Below you find the exercise questions. Each question awarding points is numbered and states the number of points like this: **[0pt]**. To answer a question, fill the cell below with your answer (markdown for text, code for implementation). Incorrect or incomplete answers are in principle worth 0 points: to assign partial reward is only up to teacher discretion. Over-complete answers do not award extra points (though they are appreciated and will be kept under consideration). Save your work frequently! (`ctrl+s`)

You need at least 14 points (out of 21 available) to pass (66%).

2 1. Fundamentals

1.1 [1pt] Explain in English what is the distinctive feature of a residual network. Residual networks implement skip connections, which are (forward) connections between non-consecutive layers of the network, such as connecting the output of layer i with the input of layer $i + k$, with $k > 1$.

1.2 [2pt] Write the full equation of a network with structure [2, 4, 1] (same as last week), but this time add (i) biases on all neurons, and (ii) self-recurrent connections only on the hidden layer. How many weights does this network have?

- I would suggest starting from your answer from last week, fixing it based on the solution if you need to and have not already, then add what you need.
- To avoid changing the indices of the weights, you can simply call bias weights b rather than w, and recurrent connections r.
- The main thing to remember is: each line has the weights entering one destination neuron, and each column refers to one of the inputs to the layer.
- Then for a recurrent network, remember to pass the output of all neurons of the same layer (technically representing the previous-step activations, initialized as `0`s) as inputs to each neuron.

The network has three layers: - An input layer (no neurons!) with two elements (x_1, x_2) - One hidden layer composed of four neurons (n_1, n_2, n_3, n_4) - The output layer with only one neuron (n_5)

We will need to add biases and recurrencies this time: it could be helpful to describe the inputs/outputs for each layer together with the weight matrix. - X is the network input, same as before - Then come the recurrent connections: all the outputs of the neurons of the hidden layer, technically from the previous time step (initialize as zeros) - Finally the bias input, the constant 1 that will be multiplied by the bias weight - X_{hid} is the actual full input to the hidden (and first) layer: all three above - W_h is the weight matrix for ALL the connections entering the hidden layer in the columns, while the rows group the connections entering each neuron - The output can be written with n_i same as we did last time; the don't forget you will need the bias also for the output layer (but no recursion!) - You can call X_{out} the input to the output (and second) layer, and W_{out} its weight matrix. And do not underestimate the value of a quick sketch on a piece of paper! Or head to draw.io if you want a computer drawing that is easy, quick and professional looking.

Remember that the output can be interpreted as one scalar, but is in principle a vector with one element (because having only one output is a special case, normally you need a list of outputs here).

Linear Algebra version:

$$X = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad out_{hid} = \begin{pmatrix} n_1 \\ n_2 \\ n3 \\ n4 \end{pmatrix}$$

$$X_{hid} = \begin{pmatrix} X \\ out_{hid} \\ 1 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ n_1 \\ n_2 \\ n_3 \\ n_4 \\ 1 \end{pmatrix}$$

$$W_{hid} = \begin{pmatrix} w_1 & w_2 & r_1 & r_2 & r_3 & r_4 & b_1 \\ w_3 & w_4 & r_5 & r_6 & r_7 & r_8 & b_2 \\ w_5 & w_6 & r_9 & r_{10} & r_{11} & r_{12} & b_3 \\ w_7 & w_8 & r_{13} & r_{14} & r_{15} & r_{16} & b_4 \end{pmatrix}$$

$$X_{out} = \begin{pmatrix} out_{hid} \\ 1 \end{pmatrix} = \begin{pmatrix} n_1 \\ n_2 \\ n_3 \\ n_4 \\ 1 \end{pmatrix} \quad =_{init} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

$$W_{out} = \begin{pmatrix} w_9 & w_{10} & w_{11} & w_{12} & b_5 \end{pmatrix}$$

$$\text{act} = \sigma\left(W_{out}\, X_{out}\right)$$

This time the definition of X_{out} does not expand directly to the activated linear combination of the hidden layer's inputs and weights, as there are recurrencies and bias so I will leave it at this.

Expanded definition using neuron names:

$$X_{hid} = \begin{pmatrix} x_1 \\ x_2 \\ n_1 \\ n_2 \\ n_3 \\ n_4 \\ 1 \end{pmatrix}$$

$$n_1 = \sigma\left(w_1x_1+w_2x_2+r_1n_1+r_2n_2+r_3n_3+r_4n_4+b_11\right) \quad n_2 = \sigma\left(w_3x_1+w_4x_2+r_5n_1+r_6n_2+r_7n_3+r_8n_4+b_21\right)$$

$$n_3 = \sigma\left(w_5x_1+w_6x_2+r_9n_1+r_{10}n_2+r_{11}n_3+r_{12}n_4+b_31\right) \quad n_4 = \sigma\left(w_7x_1+w_8x_2+r_{13}n_1+r_{14}n_2+r_{15}n_3+r_{16}n_4+b_41\right)$$

$$n_5 = \sigma\left(w_9n_1 + w_{10}n_2 + w_{11}n_3 + w_{12}n_4 + b_51\right)$$

$$act = (n_5)$$

Fully expanded equation:

No seriously why would you even considering doing this. No. You need to use the neuron names at the very least, else you will be expanding the equation of the hidden neurons twice. Don't do that. Also always remember that here we are ignoring the t: recurrent networks process streams/series sequentially, and as such the activation at time t (output) depends on the activation at time $t-1$ (which is the recurrent input).

This network has a total of 12 feed-forward weights, 16 recurrent connections (fully connected from the 4 neurons to themselves), and 5 biases (one per neuron), for a total of 33 weights.

1.3 [2pt] A neural network has only one layer of two convolutional neurons with identity activation. Below you will find respective kernels W_1 and W_2 and input X. Activate the network on the input by hand showing all calculation. Assume no padding and state explicitly the expected output size. It's easier to understand what you need to explain about your calculations if you actually start doing them :) just mark what you actually input in the calculator, what the calculator returns, and what calculation you are confident skipping.

$$W_1 = \begin{pmatrix} -1 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & -1 \end{pmatrix} \quad , \quad W_2 = \begin{pmatrix} -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \end{pmatrix} X = \begin{pmatrix} 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 \\ 1 & 1 & 1 & 1 & 1 \\ 3 & 3 & 3 & 3 & 3 \\ 2 & 2 & 2 & 2 & 2 \end{pmatrix}$$

Here are the calculations I actually made for f_1:

- I choose no padding for less calculations: with a 3x3 mask on a 5x5 input, the output will be 3x3.
- Top left of the output: multiply the mask W_1 with the top left of the input.
- The mask negates the top and bottom of the window, while doubling the center: $3 \times -1 \times 2 + 3 \times -1 \times 1 + 3 \times 2 \times 3 = -6 - 3 + 18 = 9$.
- Top center and top right are exactly the same: 9
- On the second row we get slightly different input. First and third window row have the same numbers so I just double the first element here, then add the center line: $2 \times 3 \times -1 \times 3 + 3 \times 2 \times 1 = -18 + 6 = -12$.
- This repeats 3 times for the numbers in the center output row
- Finally the last row is the same as the first but swapping top and bottom window rows. Since our mask is symmetric horizontally, the numbers will be the same as the first output row.

$$f_1 = \begin{pmatrix} 9 & 9 & 9 \\ -12 & -12 & -12 \\ 9 & 9 & 9 \end{pmatrix}$$

Now f_2:

- Again no padding, aim at a 3x3 output.

- This time the input and mask are not repeating, but that's no big deal, just a few more operations.
- The three values on each output row remain always the same, since the input is repeated along the rows.
- The value for the first row is: $-1\times(2+3+1)+2\times(2+3+1)+-1\times(2+3+1) = -6+12-6 = 0$
- On the second row we have: $-1\times(3+1+3)+2\times(3+1+3)+-1\times(3+1+3) = -7+14-7 = 0$
- Third row is of course symmetric with the first: $-1\times(1+2+3)+2\times(1+2+3)+-1\times(1+2+3) = -6+12-6 = 0$

$$f_2 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

1.4 [2pt] Look at the activations of the two neurons from 1.3 and discuss why they are so different. Explain in particular the regularities both in the inputs and in the kernels. Then go one step further and explain, to the best of your understanding, which types of features are detected by the two kernels.

- This is another open question: as long as you do not write anything wrong, while showing competence and intuition, you will get the points.
- Hint: focus on thinking about the *patterns* that you can see by eye both in the data matrix and in the kernels. Try to go for an intuitive answer.

Feature f_1 shows a distinct horizontal pattern, which means it will highlight (resonate with) an horizontal pattern in the data. Particularly, it looks for **high values on the center row**, and lower values on the rows above and below. As such it will have a higher output (feature recognition) when the input is

$$\begin{pmatrix} 2 & 2 & 2 \\ 3 & 3 & 3 \\ 1 & 1 & 1 \end{pmatrix}$$

than when it is

$$\begin{pmatrix} 3 & 3 & 3 \\ 1 & 1 & 1 \\ 3 & 3 & 3 \end{pmatrix}$$

(which is higher on first and last row and lower in the center.

Then f_2 is the same mask as f_1 but rotated by 90 degrees, so it does horizontal pattern recognition. There is however no vertical pattern in the data, yielding a constant low response. Note that the exact value of zero comes from the symmetry of the mask, and is just a special case.

1.5 [1pt] Activate a $3x3$ max pooling layer on the outputs of the two convolutions from your answer to question 1.3. Assume no padding. This is trivial thanks to the choice of no padding: a 3×3 feature space is just enough for a single output, computed by taking the `max`:

$$p_1 = (9) \quad , \quad p_2 = (0)$$

Mind, those are still 1×1 matrices, not scalars, this output size is just a special case.

To be completely correct though, you would need to recognize that the previous question mention the two neurons being in a *layer*. So the output of the convolution is not two matrices, but **a $3 \times 3 \times 2$ tensor**, a cube if you will made of two slices f_1 and f_2. The max pooling of this is a 1x1x2 tensor, with the above p_1 and p_2 as "slices".

1.6 [1pt] Explain in one sentence what is an autoencoder. Why do autoencoders have an hourglass shape? Could you design an autoencoder with a different shape? Autoencoders are neural networks that reconstruct their input after compressing and decompressing it. The central layer is smaller to enforce a compact encoding of all the information necessary to reconstruct the original input. The hourglass shape comes from the layer sizes slowly decreasing and then increasing again, to support the process.

In principle though an autoencoder can have any architecture or shape, as long as it reconstructs its inputs, it just means it cannot be used for dimensionality reduction or encoding.

I will accept also the less correct answer that no other shape is possible, but only if motivated clearly by the fact that autoencoders only *compress* their input and not expand. I accept it because most people would agree with you :) but still the goal of this course is not to limit your knowledge to what has already been done, to the names that have been assigned, but to **think** out of the box and find new examples and applications. What about using an autoencoder with a broad layer for nonlinear input pre-processing, same as kernels do for SVMs? *(we are actually currently working on this)*

3 2. Recurrent Neural Networks

2.1 [3pt] Below is last week's implementation of a neural network augmented into a fully-connected RNN with bias connections. Fix it by writing the missing code as marked by `?`.

- Unless otherwise stated, a RNN has fully-connected self-recurrent connections on each layer.
- You should know exactly which connections to add if you answered the RNN question in the fundamentals.
- For the bias: remember that you need all elements in `state` to be longer by one element: put actual `1`s in these last positions at initialization, then never touch them again.
- Recurrencies: you need to make space in the input to each layer for its own output. I typically sort them as [input, recurrencies, bias], but order is not important: consistency is. Make sure all your sizes are correct.
- When calculating the size of an input now you need to use `struct` twice: once for the size of the layer entering (something like `struct[nlay]`) and once for the size of the output that goes back as input in the recursion (hence `struct[nlay+1]`). HINT: to make the pairs for each layer execute and understand the following: `zip(struct, struct[1:])`
- When you activate a layer remember to copy the activation to both (i) its output and (ii) its input, at the correct indices.

- It's easier to compute the size of each input beforehand, then use it to make the `state` list. Then for each weight matrix you can take the number of rows from the structure (as before), and the number of inputs from the `state` sizes. Don't worry about duplicating the activation in the layer's input, it's actually faster because you have a ready numpy array rather than composing at activation.
- Remember to initialize the recurrent output to `0`. Simplest way is to initialize `state` using `np.zeros()` instead of `np.empty()`. Then set the last value of each `state` element to `1` for the bias.
- I used myself `import IPython; IPython.embed()` heavily to get this to work. You can also call (once!) `%pdb` to drop in the debugger on error. Keep calm and check the dimensions.
- The layer activation function should change because you are saving the activation to two locations (and at specific indices, not direct substitution like before), but the network activation function should change just marginally (add indices to insert input in state)
- To *convolve* with stride 1 and no padding a window of size two on a 1D list (take a pair at a time, advance by one) in Python you can use `zip(lst, lst[1:])`.
- This question only refers to neural networks, not learning algorithm, so leave backpropagation out and do not worry about unrolling the network.
- Again activate it on a simple input to verify everything is works. The input should be exactly the same as last week's (as the network architecture).
- Think: how many weights do you expect to have? Remember that you have 3 matrices (for the 3 layers of neurons), in each the number of rows is unchanged (because you have one per each neuron) but the inputs now are not only connections from the previous layer, you also have recursion (take the output of this layer as its own input) and bias (constant 1 appended to the inputs).

[2]:
```
class RecurrentNeuralNetwork:
    def __init__(self, struct):
        # These are basic, copy+paste from FFNN
        self.struct = struct
        self.nlayers = len(self.struct)
        self.nins, *self.nhids, self.nouts = self.struct
        self.sigma = lambda x: 1/(1+np.exp(-x))

        # Each `state` is an input for next layer: it now includes rec and bias
        state_sizes = [inp+rec+1 for inp, rec in zip(self.struct, self.struct[1:
↪])]
        # Notice the `zip` above ends when the second list reaches the end (1␣
↪shorter)
        # Last `state` is only the output of the last layer (no rec/b)
        state_sizes += [self.struct[-1]]
        # We can now build the state of the network
        self.state = [np.empty(size, dtype='float64') for size in state_sizes]

        # We will need to access inputs and recurrencies by index: the␣
↪following helps
        self.inp_idxs = [range(0, nins) for nins in self.struct]
```

```
        self.rec_idxs = [range(orig, orig+nneur) for orig, nneur in zip(self.
↪struct, self.struct[1:])]
        # Finally, fix the bias input in the last position of all input `state`s
        # Just set and forget, and no need for indices because we won't access␣
↪it again
        for s in self.state: s[-1] = 1

        # The `state` sizes now correspond to the row lengths (ncols) for the␣
↪weight matrices
        self.wsizes = [[nrows, ncols] for nrows, ncols in zip(self.struct[1:],␣
↪state_sizes)]
        # Finally: weight initialization. Bad practice to hardcode this, but ok␣
↪here
        self.weights = [np.random.normal(size=ws) for ws in self.wsizes]

    # The layer activation is unchanged: sigma(W.dot(X)) -- only W and X␣
↪(=state) differ
    def act_layer(self, nlay):
        return self.sigma(self.weights[nlay].dot(self.state[nlay]))

    # The network activation only writes the act twice this time: output &␣
↪rec-input
    def act_net(self, inp):
        assert len(inp) == self.nins, f"got input `{inp}`, expected np.array of␣
↪length `{self.nins}`"
        self.state[0][self.inp_idxs[0]] = inp
        for nlay in range(self.nlayers-1):
            act = self.act_layer(nlay)
            # This time the layer activation goes in two places:
            # - In the input indices of the output of this layer / input to next
            self.state[nlay+1][self.inp_idxs[nlay+1]] = act
            # - In the recurrent indices of the input to this layer
            self.state[nlay][self.rec_idxs[nlay]] = act
        return self.state[-1]
```

[3]:
```
struct = [4,5,4,3]
inputs = [3,2,4,3]
net = RecurrentNeuralNetwork(struct)
# We expect the activation to change this time upon multiple calls on the same␣
↪input
# This is because the `state` of the RNN is maintained in the recurrent␣
↪connections
print("activation 1:", net.act_net(np.array(inputs)))
print("activation 2:", net.act_net(np.array(inputs)))
print("activation 3:", net.act_net(np.array(inputs)))
```

```
activation 1: [0. 0. 0.]
```

```
activation 2: [0.68170409 0.80287028 0.40699344]
activation 3: [0.78289463 0.83589859 0.73550227]
```

```
/tmp/ipykernel_120189/1927083085.py:7: RuntimeWarning: overflow encountered in
exp
  self.sigma = lambda x: 1/(1+np.exp(-x))
```

4 3. Convolutional Networks

1.1 [2pt] Write a Python function for 2D convolution, then run it on a randomly generated input matrix and show the output.

- You need to write a function that takes a 2D input and a function to convolve as parameters, then convolves the function over the inputs to produce the output.
- The window size is not specified: you can use a 3x3 to keep it simple since you saw that in the examples.
- The function to convolve is not specified: no need for it to be a neural network, something as simple as `sum` or a quick lambda calling the numpy `x.sum()` would work perfectly well. Remember that the important part is that it should take a high-dimensional (3x3?) input and output only one value.
- The size of the input matrix is not specified: with a 3x3 mask we could go as small as 5x5 with no padding and that would still show that the convolution works.
- Then remember: the neural networks are just other functions than `sum`, but behave exactly the same way. Convolving a neural network only allows you to learn the function rather than hardcoding it, but the convolution process is independent.
- Answer the question until the end: show that you know how to create a matrix of random numbers, and of the right size.

[4]:
```
def convolve(fn, inp):
    nrows, ncols = inp.shape
    assert nrows>=3 and ncols >=3, f"input too small! {inp}"
    outp = np.empty([nrows-2, ncols-2]) # the 3x3 mask drops 2 in each dimension
    for r in range(nrows-2):
        for c in range(ncols-2):
            outp[r,c] = fn(inp[r:r+2, c:c+2])
    return outp

feat = convolve(lambda x: x.sum(), np.random.uniform(size=[5,5]))
print(feat)
```

```
[[1.8815096  1.59487879 1.33690188]
 [2.22172509 2.06492432 1.66707057]
 [2.7011297  2.06212249 1.93019955]]
```

5 4. Handwritten digit recognition with Keras

As mentioned at the beginning, we need to cross a gap in exercise complexity. On one hand you are ready to understand the inner work of a DL library like Keras, on the other asking you for such

a task on top of today's lecture is too much even for this course :) So let's leave the creative part for next weeks, where we will see some more advanced applications anyway, and focus today on what we learned and on a new skill: how to justify your code.

Below is a tutorial from the Keras website on convolutional networks [source]. It uses the MNIST dataset, a standard dataset for handwritten character recognition. Keras offers you a backend to automatically download the dataset, similarly to what we did so far with Seaborn and Iris.

The code should work as is (did you `pipenv install tensorflow keras`?), but take a while to run. Read it line by line, really study and understand it, feel free to change it so that it runs in few seconds if you wish to play with it; then answer the questions below.

NOTE: you don't need to run the code to answer any of the questions below. If you used PyTorch at the last lecture, this is your chance to try out Keras. If you cannot (looking at you M1 users), you can use Colab for this assignment, or replace the Keras tutorial with a PyTorch equivalent such as this. This only affects 4.3, while 4.1 and 4.2 should be fine.

[5]:
```
"""
Title: Simple MNIST convnet
Author: [fchollet](https://twitter.com/fchollet)
Date created: 2015/06/19
Last modified: 2020/04/21
Description: A simple convnet that achieves ~99% test accuracy on MNIST.
SOURCE: https://github.com/keras-team/keras-io/blob/master/examples/vision/
 ↪mnist_convnet.py
"""

"""
## Setup
"""

import numpy as np
from tensorflow import keras
from tensorflow.keras import layers

"""
## Prepare the data
"""

# Model / data parameters
num_classes = 10
input_shape = (28, 28, 1)

# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Scale images to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
```

```
# Make sure images have shape (28, 28, 1)
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
print("x_train shape:", x_train.shape)
print(x_train.shape[0], "train samples")
print(x_test.shape[0], "test samples")


# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

"""
## Build the model
"""

model = keras.Sequential(
    [
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation="softmax"),
    ]
)

model.summary()

"""
## Train the model
"""

batch_size = 128
epochs = 15

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

history = model.fit(
    x_train, y_train,
    batch_size=batch_size,
    epochs=epochs,
```

```
    validation_split=0.1)

"""
## Evaluate the trained model
"""

score = model.evaluate(x_test, y_test, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

```
2023-04-26 16:25:18.983306: I tensorflow/core/util/util.cc:169] oneDNN custom
operations are on. You may see slightly different numerical results due to
floating-point round-off errors from different computation orders. To turn them
off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2023-04-26 16:25:18.986645: W
tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load
dynamic library 'libcudart.so.11.0'; dlerror: libcudart.so.11.0: cannot open
shared object file: No such file or directory; LD_LIBRARY_PATH:
:/home/giuse/.mujoco/mujoco200/bin:/home/giuse/.mujoco/mjpro150/bin
2023-04-26 16:25:18.986655: I tensorflow/stream_executor/cuda/cudart_stub.cc:29]
Ignore above cudart dlerror if you do not have a GPU set up on your machine.
```

```
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
Model: "sequential"
```

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

 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 flatten (Flatten)           (None, 1600)              0

 dropout (Dropout)           (None, 1600)              0

 dense (Dense)               (None, 10)                16010

=================================================================
Total params: 34,826
Trainable params: 34,826
```

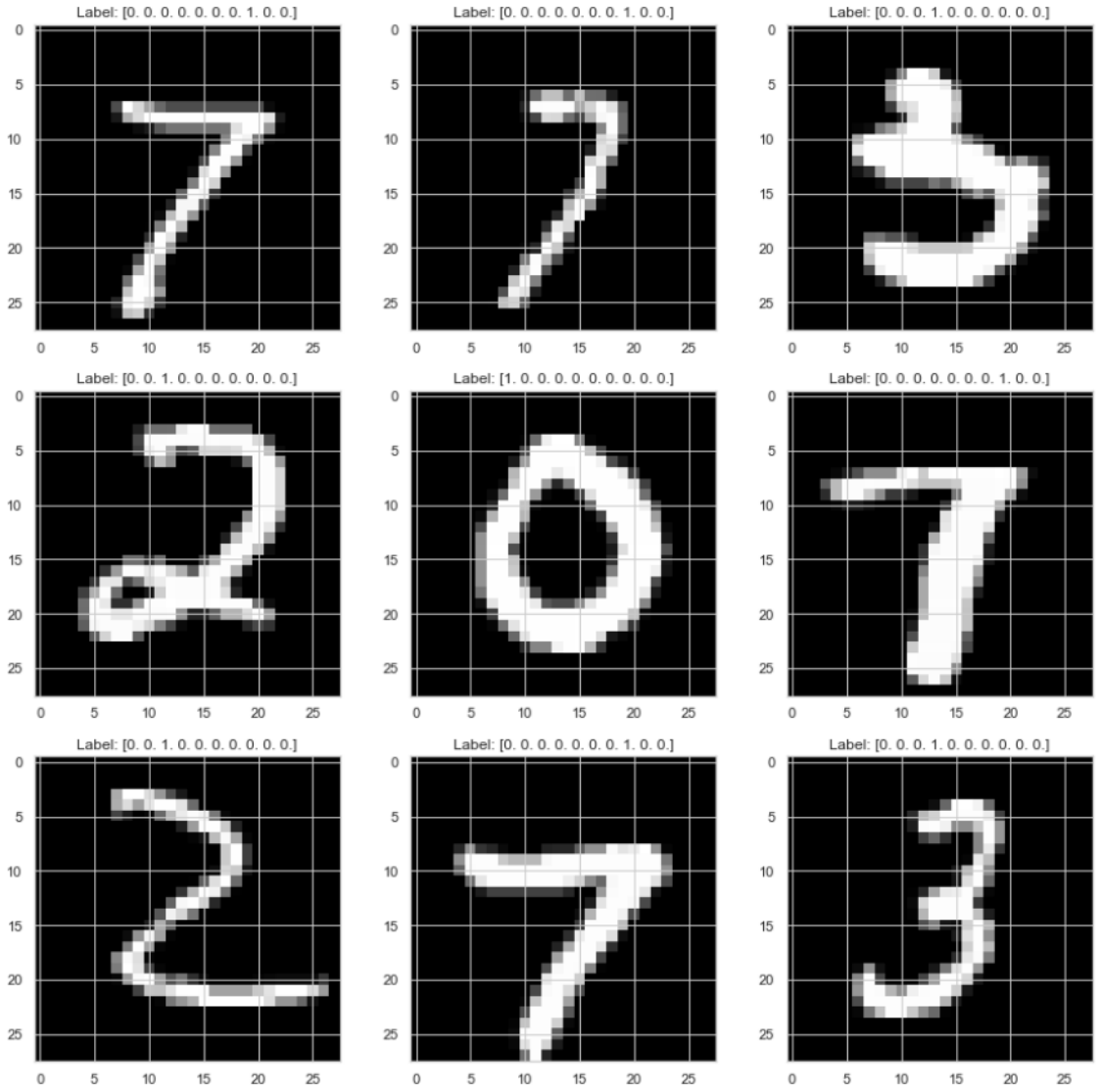
```
Non-trainable params: 0
_________________________________________________________________
2023-04-26 16:25:20.167923: W
tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load
dynamic library 'libcuda.so.1'; dlerror: libcuda.so.1: cannot open shared object
file: No such file or directory; LD_LIBRARY_PATH:
:/home/giuse/.mujoco/mujoco200/bin:/home/giuse/.mujoco/mjpro150/bin
2023-04-26 16:25:20.167945: W
tensorflow/stream_executor/cuda/cuda_driver.cc:269] failed call to cuInit:
UNKNOWN ERROR (303)
2023-04-26 16:25:20.167976: I
tensorflow/stream_executor/cuda/cuda_diagnostics.cc:156] kernel driver does not
appear to be running on this host (leaf): /proc/driver/nvidia/version does not
exist
2023-04-26 16:25:20.168192: I tensorflow/core/platform/cpu_feature_guard.cc:193]
This TensorFlow binary is optimized with oneAPI Deep Neural Network Library
(oneDNN) to use the following CPU instructions in performance-critical
operations:  AVX2 AVX512F AVX512_VNNI FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.

Epoch 1/15
422/422 [==============================] - 8s 19ms/step - loss: 0.3720 -
accuracy: 0.8857 - val_loss: 0.0822 - val_accuracy: 0.9775
Epoch 2/15
422/422 [==============================] - 8s 19ms/step - loss: 0.1110 -
accuracy: 0.9656 - val_loss: 0.0570 - val_accuracy: 0.9847
Epoch 3/15
422/422 [==============================] - 8s 19ms/step - loss: 0.0833 -
accuracy: 0.9746 - val_loss: 0.0451 - val_accuracy: 0.9900
Epoch 4/15
422/422 [==============================] - 8s 19ms/step - loss: 0.0695 -
accuracy: 0.9786 - val_loss: 0.0385 - val_accuracy: 0.9903
Epoch 5/15
422/422 [==============================] - 8s 19ms/step - loss: 0.0604 -
accuracy: 0.9811 - val_loss: 0.0365 - val_accuracy: 0.9902
Epoch 6/15
422/422 [==============================] - 8s 19ms/step - loss: 0.0554 -
accuracy: 0.9823 - val_loss: 0.0383 - val_accuracy: 0.9898
Epoch 7/15
422/422 [==============================] - 8s 19ms/step - loss: 0.0506 -
accuracy: 0.9839 - val_loss: 0.0330 - val_accuracy: 0.9917
Epoch 8/15
422/422 [==============================] - 8s 19ms/step - loss: 0.0470 -
accuracy: 0.9854 - val_loss: 0.0329 - val_accuracy: 0.9923
Epoch 9/15
422/422 [==============================] - 8s 19ms/step - loss: 0.0442 -
accuracy: 0.9861 - val_loss: 0.0296 - val_accuracy: 0.9918
```

```
Epoch 10/15
422/422 [==============================] - 8s 19ms/step - loss: 0.0437 -
accuracy: 0.9862 - val_loss: 0.0304 - val_accuracy: 0.9915
Epoch 11/15
422/422 [==============================] - 8s 20ms/step - loss: 0.0394 -
accuracy: 0.9873 - val_loss: 0.0320 - val_accuracy: 0.9912
Epoch 12/15
422/422 [==============================] - 8s 20ms/step - loss: 0.0373 -
accuracy: 0.9875 - val_loss: 0.0311 - val_accuracy: 0.9918
Epoch 13/15
422/422 [==============================] - 9s 20ms/step - loss: 0.0360 -
accuracy: 0.9886 - val_loss: 0.0296 - val_accuracy: 0.9928
Epoch 14/15
422/422 [==============================] - 8s 20ms/step - loss: 0.0334 -
accuracy: 0.9889 - val_loss: 0.0313 - val_accuracy: 0.9917
Epoch 15/15
422/422 [==============================] - 9s 21ms/step - loss: 0.0329 -
accuracy: 0.9892 - val_loss: 0.0296 - val_accuracy: 0.9918
Test loss: 0.024067629128694534
Test accuracy: 0.9916999936103821
```

4.1 [1pt] Data plotting: plot the first 9 images in MNIST using a 6x6 subplot. Let's first see what the MNIST looks like. I showed how to use subplots in a recent solution – do you remember where it was? You'll need the ability to search quickly for what you need to complete the exam in time. Try timing how long it takes you to answer this question (no seriously challenge your teammate on who solves this the quickest and feel free to brag about it in your solution below). - Add the label on the title to see how the numbers are represented: do you see the connection to last week's `species`? - After you obtain the axis from `plt.subplots()`, you can print a 2D image using `ax[?,?].imshow()`

[6]:
```
nrows, ncols = [3,3]
fig, ax = plt.subplots(nrows, ncols, figsize = (15,15))
for r in range(nrows):
    for c in range(ncols):
        idx = np.random.randint(len(x_train))
        ax[r, c].imshow(x_train[idx], cmap='gray')
        ax[r, c].set_title(f"Label: {y_train[idx]}")
```



4.2 [4pt] Explain the following lines in the Keras MNIST Tutorial code (in English): 27, 48, 51, 52, 57, 67/77, 68/78, 71

- To answer this question you need to show complete competence, as if you wrote this code yourself and you were asked to explain your choices at an oral exam.
- For each of the lines mentioned, check the code provided and explain it thoroughly
- For each variable, explain its meaning, its use, and the choice of value assigned
- For each function call, explain what it does, the meaning of all parameters, and the choices of all values.
- Reading the code like "assign 12 to variable `epochs`" will not constitute an acceptable answer.
- Reading the code like "creates a new Sequential" is also not acceptable: check the documentation for `Sequential`, understand what the call does, and present your findings.

- **27:** `(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()`

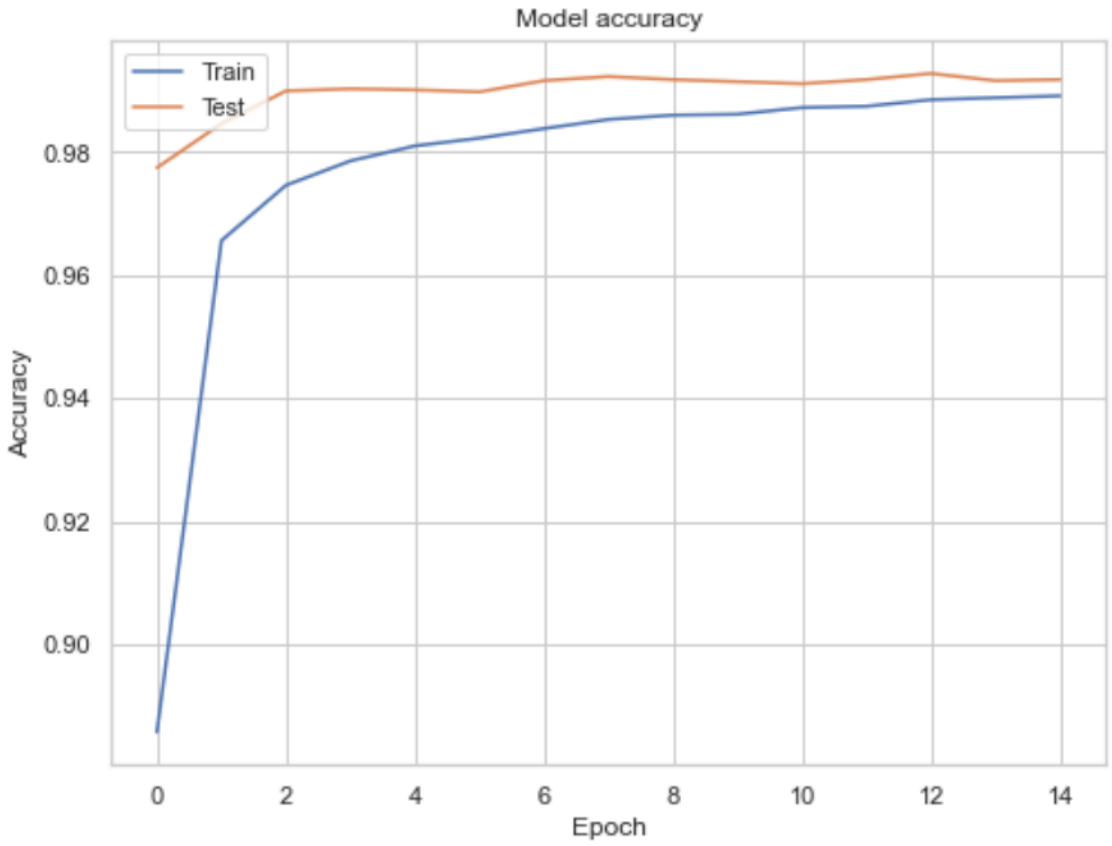
The function `load_data()` returns the MNIST dataset already split in train and test sets. Because you *always* need to remember to do that, right? ;)

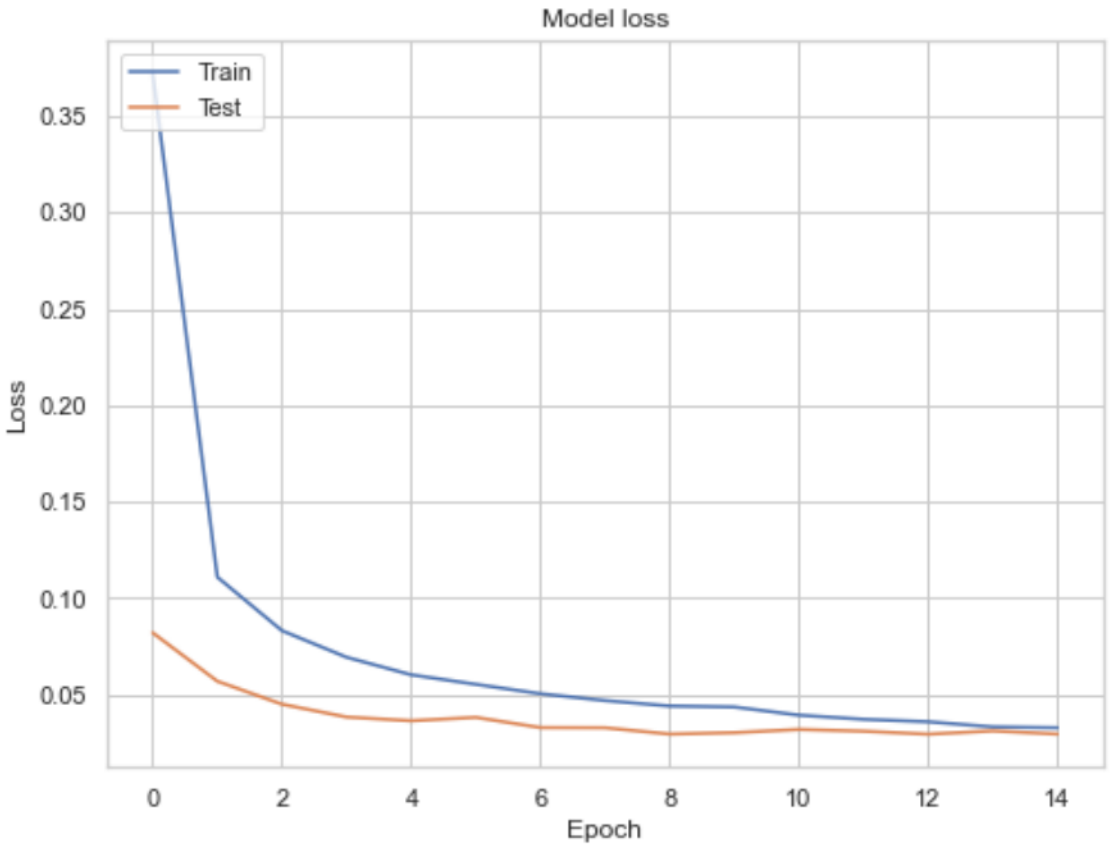
- **48:** `model = keras.Sequential()`
  InstantiateS a new Keras model using class Sequential. This creates a neural network that activates sequentially, without skips and with fully connected layers. Its argument is the list of layers in their right order.
- **51:** `layers.Conv2D(32, kernel_size=(3, 3), activation="relu")`
  Adds a 2D convolutional layer to the sequential model. It is composed of 64 neurons, and the kernel (for each neuron) is 3x3. It also uses a `relu` as activation function, which we will see next week: a Rectified Linear Unit, an activation function that is zero for negative inputs and identity (f(x)=x) for positive inputs. It uses default values for stride (1,1) and padding ('valid', meaning no extra padding).
- **52:** `layers.MaxPooling2D(pool_size=(2, 2))`
  Adds a 2D Max-Pooling layer, which will process the output of the convolution described above. Its 2x2 window size means that it will return the maximum for each 2x2 elements in the convolved feature.
- **57:** `layers.Dense(num_classes, activation="softmax")`
  The output layer of the network is a fully-connected layer with one neuron for each class and softmax activation. This means that the output vector from the network activation will have 10 numbers that sum up to 1.0, each corresponding to the confidence of our model that the class correspondent to that neuron is the correct one.
- **67/77:** `batch_size = 128 # argument to model.fit()`
  Sets the batch size to 128, which is the number of MNIST images that will be run through a forward pass then aggregated to compute one error for the backpropagation algorithm.
- **68/78:** `epochs = 15 # argument to model.fit()`
  Sets the number of backpropagation passes to 15, which should strike as surprisingly low for 99% accuracy, even with a batch size so large.
- **71:** `loss="categorical_crossentropy" # argument to model.compile()`
  Selects categorical cross-entropy as the loss function for our training. This is a logical choice as we have a multiclass classification problem, with each digit being its own class.

4.3 [2pt] Run Keras MNIST code, tweaking it as needed if it takes too long on your machine. Plot the model's accuracy and loss over time.

- This is almost for free since you did the same visualization last week, but you need to get the code to run first.
- Also you may want to make sure your changes include setting 'accuracy' and the `history` variable, or at the end of the run you could end up with still nothing to show :)

[7]:
```
# Plot training & validation accuracy values
for idx, metric in enumerate(['accuracy', 'loss']):
    plt.figure(idx) # generates distinct plots
    plt.plot(history.history[metric])
    plt.plot(history.history['val_' + metric])
    plt.title('Model ' + metric)
    plt.ylabel(metric.capitalize())
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Test'], loc='upper left')
```


Model accuracy
Train
Test
Accuracy
0.98
0.96
0.94
0.92
0.90
0
2
4
6
8
10
12
14
Epoch



6 At the end of the exercise

Bonus question with no points! Answering this will have no influence on your scoring, not at the assignment and not towards the exam score – really feel free to ignore it with no consequence. But solving it will reward you with skills that will make the next lectures easier, give you real applications, and will be good practice towards the exam.

The solution for this questions will not be included in the regular lab solutions pdf, but you are welcome to open a discussion on the Moodle: we will support your addressing it, and you may meet other students that choose to solve this, and find a teammate for the next assignment that is willing to do things for fun and not only for score :)

BONUS [ZERO pt] Edit the Keras MNIST code to use a simple RNN, then cheat by passing all images of a class in a sequence (careful with batch size). Reset the network between classes. RNNs will recognize that you expect a constant output per each sequence, decide which output with the first few images, then just saturate the right neurons using the recurrent connections to generate a constant output regardless of the input. You can verify this by then testing the network on a sequence of elements from a constant class, followed by one (or more) elements from another class: they will likely be misclassified. All intelligent learning picks up on shortcuts whenever

available, here is a famous example (check the full paper): husky vs. wolf. Notice that getting a "simple" RNN in Keras is not straightforward, and we will see LSTMs next week.

6.0.1 Final considerations

- At the end of this lecture + exercise you should *own* neural networks. It does not mean that you know everything about them, but you know enough to understand any resource on the topic, and actually understand how these things work better than most people who just use Keras/Pytorch on a daily basis (unfortunately).
- Get the fixed Keras MNIST code to work on GPUs/TPUs using Colab to get a first feel of the speed boost from specialized hardware that we will present next week.