assignment_09_solution

April 21, 2023

Please fill in your name and that of your teammate.

You:

Teammate:

[1]:
```
%matplotlib inline
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 ninth lab. Time to start with the famed neural networks. Everything should be fine until you hit the backpropagation algorithm. There are dozens of versions and implementations online, none are simple or straightforward, but in the lecture I tried an explanation that keeps the complexity to a minimum. It may be confusing to find which part does what and how to implement it, so I added a few tips that I hope will limit your debugging time. Enjoy!

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] Describe a real (human) neuron. Use the words "dendrite", "axon", "synapses" and "spike". A neuron is a specialized type of cell of the nervous system. Its peculiarity is in forming large networks of similar cells, connected by touch (synapses), and communicating with each other using electric signals. It does so by (i) receiving electric impulses

(spikes) from other neurons through its dendrites; (ii) if the total potential received over a certain amount of time is above a threshold, then (iii) a spike is sent through its axon (a sort of elongated output connection) to all cells it is connected to.

1.2 [1pt] Describe the logistic function (in English). Include/utilize the concept of "saturation". The logistic function σ is a sigmoid function (meaning shaped like an S) typically employed to model the non-linearity of neurons. It is defined over the whole $\mathbb{R}$ in input, but its output is bounded in the open interval $(0, 1)$. Its inflection point is at point $(0, 0.5)$, meaning that depending the sign of the input it will output a value above or below 0.5. It is close to linear near its inflection point, but then curves nonlinearly to respect its boundary. This implies that the output for two near, high-valued inputs will be basically indistinguishable: both $\sigma(10^{10})$ and $\sigma(10^{10} + 1)$ will return the same value of 1, with indistinguishable variations.

1.3 [1pt] Explain the relationship between the human brain, neural networks, and perceptrons (in English). Neural networks were initially derived to describe the human brain, as networks of artificial neurons. These artificial neurons were mathematical models of extreme simplifications on the structure of human neurons. The resulting model however is typically better understood as a generic function approximator with a formula (and corresponding behavior) more similar to an extended Perceptron. This can be found both in its use of the parametrization (linear combination of weights and inputs) and in its main learning algorithm Backpropagation (iteratively update the weights in the direction of diminishing errors).

1.4 [2pt] Write the full equation for a network with structure [2, 4, 1], specifying the value of the final activation `act`. How many weights does this network have?

- This means two inputs, one hidden layer of four neurons, and one neuron in the output layer.
- You can provide either the linear algebra equation (still, define any vector or matrix you use) or the fully-expanded version (as `act` in the slides).
- Feel free to ignore the bias connection for now.
- To define multiple equations on individual lines: use the double dollar sign environment, then \\ to go to a new line (see source of this cell: `$$ eq_1=0 \\ eq_2=1 $$` =>

$$eq_1 = 0 eq_2 = 1$$

- To draw nice matrices: wrap the elements in a `pmatrix` environment, then use `&` to tabulate to the next column and \\ to go to the next row (see source of this cell):

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

The network has three layers: an input layer with two elements (x_1, x_2), one hidden layer composed of four neurons (n_1, n_2, n_3, n_4), while the output layer has only one neuron (n_5). 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:

$$X = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

$$W_1 = \begin{pmatrix} w_1 & w_2 \\ w_3 & w_4 \\ w_5 & w_6 \\ w_7 & w_8 \end{pmatrix}$$

$$W_2 = \begin{pmatrix} w_9 & w_{10} & w_{11} & w_{12} \end{pmatrix}$$

$$\text{act} = \sigma\left(W_2\, \sigma\left(W_1\, X\right)\right)$$

Expanded definition using neuron names:

$$X = \{x_1, x_2\}$$

$$n_1 = \sigma\left(w_1 x_1 + w_2 x_2\right) \quad n_2 = \sigma\left(w_3 x_1 + w_4 x_2\right) \quad n_3 = \sigma\left(w_5 x_1 + w_6 x_2\right) \quad n_4 = \sigma\left(w_7 x_1 + w_8 x_2\right)$$

$$n_5 = \sigma\left(w_9 n_1 + w_{10} n_2 + w_{11} n_3 + w_{12} n_4\right)$$

$$act = \begin{pmatrix} n_5 \end{pmatrix}$$

Fully expanded equation:

$$act = \Big[\sigma(w_9 \sigma\left(w_1 x_1 + w_2 x_2\right) + w_{10} \sigma\left(w_3 x_1 + w_4 x_2\right) + w_{11} \sigma\left(w_5 x_1 + w_6 x_2\right) + w_{12} \sigma\left(w_7 x_1 + w_8 x_2\right))\Big]$$

This network has a total of 12 weights because there are no biases, as it was not required in the question. A corresponding network with biases would have one more weight for each of the 5 neurons, totaling 17 weights.

1.5 [2pt] The network parametrization only defines an *upper bound* for the network's functional complexity: explain what does this mean (in English). Then also answer: would an overly large network work for a simple problem? Would an overly small network work for a complex problem? A neural network with fixed structure and activation defines a subspace of all possible functions. These range in complexity from the simplest functions (constant) to an upper bound defined by the network structure and activation. In principle, the larger the structure (and the more complex the activation), the higher this upper bound can go. Even the most complex, largest of network though, can be used to define constant functions. By simply setting the weight matrix of the output layer to zeros, we force the network output to correspond to a vector of constant numbers that do not depend on the inputs, but only on the value of the activation function on zero.

For example, using the logistic function and a network of arbitrary size with 3 output neurons, setting the last weight matrix to zeros yield the following constant function:

$$f(x) = \begin{pmatrix} 0.5 \\ 0.5 \\ 0.5 \end{pmatrix}$$

This function does not depend on x and actually ignores any computation depending on the other weight matrices.

- An overly large network should in principle work for a simpler problem. Beware though as larger structure means larger function subspace, which may be harder or take longer to explore (higher sensitivity to initialization).
- An overly small network would in principle "work" for a complex problem, in the sense that if the number of inputs and outputs match then the function can be applied. At the same time, small networks are inherently incapable of approximating higher complexity functions to arbitrary precision, so in that sense the answer is "no".

1.6 [1pt] Explain the implications of the Universal Approximation theorem (in English). The fundamental implication of the Universal Approximation theorem is that any function can in principle be approximated by a neural network of sufficient size.

This is a double-edged sword though: - The higher the complexity of the network, the higher the size of the function space they map to, which can make the target function actually harder to find. - At the same time, the regular structure of the network implies multiple equivalent solutions are always available (e.g. think of a network graph and its upside-down mirrored equivalent), making the search space highly multimodal. - The theorem proof utilizes an arbitrarily large hidden layer, while function complexity grows faster with multiple layers using function composition. This is the reason why deep networks work better than (equivalent-size) broad networks. Which led us to focus on deep networks, which are not directly considered in the Universal Approximation theorem. The result is that most research nowadays blindly employs larger and larger networks, trading off the raised complexity upper bound with a plethora of problems such as sensitivity from initialization conditions, extremely low sample-efficiency (needing humongous amount of labeled training data), and huge performance requirements.

3 2. Multilayer feed-forward neural networks

As is customary, in this section you get to code neural networks by hand. There are many possible implementation; we will focus on a toy version without parallelism or GPUs, but employing Numpy and linear algebra.

2.1 [2pt] Fix the implementation of a simple neural network below (missing parts are denoted by ?). Print the activation for a network with logistic transfer, structure [4,5,4,3], random weights, and input [2,1,2,1].

- If this is the first Python class you see, worry not: it's just a container to keep code and data together
- For now remember only to place `self` as first argument of all methods, and to call class methods as `self.method_name()`

- We will **not** implement the bias at this round: start thinking about what would you change to add it though, because we will next week.
- Remember that a weight matrix corresponds to one layer of neurons: the rows correspond to the incoming connections to each neuron (so as many rows as neurons), the columns to incoming inputs that are connected to each of the neurons (so one column per input)
- So, if the structure was for example [2,3,1], then the first matrix would be between a layer of size 2 (inputs) and a layer of size 3: 3 neurons with 2 inputs each, so a weight matrix of size (3,2). The second weight matrix would have size (1,3). Adapt these numbers to the question's structure of course
- When testing, remember to pass numpy arrays as inputs! Python lists will not work (think: why?).
- The method `__init__()` is one of the pre-determined methods for object oriented programming in Python. It is called automatically every time you make an object of the class.
  - Technically: when instantiating an object, first the constructor allocates the memory, then this method initializes the values and performs any process to get the object in a ready-to-compute state.

[2]:
```
class FeedForwardNeuralNetwork:
    def __init__(self, struct):
        # The structure of the network is a list of (inputs and) layers sizes
        self.struct = struct
        self.nlayers = len(self.struct)
        # You can deconstruct the list using our old acquaintance the splat
        self.nins, *self.nhids, self.nouts = self.struct
        # IMPORTANT: state is a list of layer outputs, filled by the forward␣
↪pass
        # IMPORTANT: state[0] contains the inputs, state[1] will contain the␣
↪output
        # of the first layer of neurons, corresponding to struct[1]
        self.state = [np.empty(lsize, dtype='float64') for lsize in self.struct]
        # We also need to compute the size of the weight matrices
        # Remember: a row describes the input connections to one neuron
        self.wsizes = [[self.struct[i+1], self.struct[i]] for i in␣
↪range(len(self.struct)-1)]
        # IMPORTANT: weights is a list of weight matrices per each layer of␣
↪neurons
        # IMPORTANT: weights[0] will hold the weights entering the first layer,␣
↪which
        # corresponds to the neurons of struct[1], and will be used to compute␣
↪state[1]
        self.weights = [np.random.normal(size=ws) for ws in self.wsizes]
        # Finally our activation function, the humble (logistic) sigmoid
        self.sigma = lambda x: 1/(1+np.exp(-x)) # we could have used a `def`␣
↪here
```

```
    # To activate a layer: activate the linear combination of weights and␣
↪inputs
    def act_layer(self, nlay):
        return self.sigma(self.weights[nlay].dot(self.state[nlay]))

    # To activate a network, activate each layer in turn, saving the activation␣
↪in
    # the network state, finally return the output of the last layer
    def act_net(self, inp):
        assert len(inp) == self.nins, f"got input `{inp}`, expected np.array of␣
↪length `{self.nins}`"
        self.state[0] = inp
        for nlay in range(self.nlayers-1):
            self.state[nlay+1] = self.act_layer(nlay) # different indexes!
        return self.state[-1]
```

[3]:
```
struct = [4,5,4,3]
inputs = [2,1,2,1]
net = FeedForwardNeuralNetwork(struct)
print("activation:", net.act_net(np.array(inputs)))
```

```
activation: [0.80612908 0.49873031 0.2606843 ]
```

2.2 [5pt] Fix the implementation of Backpropagation below (missing parts are denoted by ?). Then (i) instantiate a new network with the same structure as before, (ii) activate it on dataset provided and compute the Risk based on Mean Squared Error, (iii) run the Backpropagation algorithm for 10000 epochs, (iv) compute the Risk again (should be decreased).

- We *subclass* our implementation from the question above:
  - This means creating a new class, but as if we **resume** implementing from where we left of with the class above
  - Think of it as copy+paste of *all methods* defined in the parent class (above), then everything we add (new specialized functionality) is written "below" the old code
  - We can overwrite previous definitions: "overloading". If our subclass below implements `def act_layer()` for example, this code will be used in place of the parent class' one
  - Don't worry too much for now, it's very intuitive, only know that some of your function definitions are in the class above instead of this one, nothing more
- Keep it simple: use the implementation seen in the slides, based on logistic function and MSE loss.
- You need two update functions: updating the output layer and the hidden layers uses different equations.
- If you follow this implementation, you don't need a vectorized version of the activation function. But if you want it to work with numpy broadcast you should learn about `np.vectorize(act_net, signature='(n)->(m)')`.
- Careful with the size of inputs and outputs you generate for testing, remember that you need to match the structure of the network (number of inputs).

- The `online` algorithm will save you from one layer of linear algebra implementation: simply loop for the number of epochs, then loop for each point (pairing input/label with `zip`).
- Backprop cheatsheet: 1. forward pass, 2. backward on output layer, 3. loop of backward on each hidden layer back-to-front.

[4]:
```
class FeedForwardNeuralNetworkWithBackprop(FeedForwardNeuralNetwork):

    # This definition _overloads_ ("overwrites") the definition in the parent␣
↪class
    def __init__(self, struct):
        super().__init__(struct) # calls the `__init__` of the _parent_class_␣
↪above!

        # But then we add one new action to the initialization:
        # vectorizing the method allows to "activate" on a whole dataset at␣
↪once!
        self.act_net_v = np.vectorize(self.act_net, signature='(n)->(m)')


    def backprop(self, x, y, nepochs=10000, lrate=0.1):
        for nepoch in range(nepochs):
            # show progress, particulary important on slow methods
            if nepoch%100==0: print('.', end='', flush=True)
            # loop for each point in the dataset
            for xi, yi in zip(x, y):

                ## Forward pass (we will take the activations from `state`)
                self.act_net(xi)

                ## Backward pass on output layer (easy)
                last_act = self.state[-1]
                output_layer_input = self.state[-2] # different from above!
                d_act = (1-last_act) * last_act
                delta = (last_act-yi) * d_act
                de_dw =  np.outer(delta, output_layer_input)
                # `+= -1 * ...` => explicitly: incremental change, negative␣
↪direction
                self.weights[-1] += -1 * lrate * de_dw

                ## Backward pass on hidden layers
                # Navigate the [weight matrices] of [hidden layers], [backwards]
                indices_of_hidlay_wmat_bwards = range(len(self.weights)-1)[::-1]
                for idx in indices_of_hidlay_wmat_bwards:
                    # Consider the outputs of two layers at a time: current and␣
↪previous
                    # Look at self.struct:
                    # - we have a weight matrix between each pair of sizes
```

```
                    # - we have an activation for each size but for inputs
                    # So the indices are skewed! If we loop on the index (idx) of
                    # the weight matrix that we are currently updating, the inputs to
                    # the corresponding layer will be in `state[idx]`, the outputs in
                    # `state[idx+1]`! Be careful and check the shapes!
                    curr_act = self.state[idx+1]
                    prev_act = self.state[idx]
                    d_act = (1-curr_act) * curr_act
                    # `delta` currently contains the delta we calculated for idx+1!
                    delta_idx_plus_one = delta
                    delta = np.dot(delta_idx_plus_one, self.weights[idx+1]) * d_act
                    # delta = self.weights[idx+1].T.dot(delta) * (1-last_act) * last_act
                    de_dw = np.outer(delta, prev_act)
                    self.weights[idx] += -1 * lrate * de_dw

        print() # newline after all the points
```

[5]:
```
# Use this test to ensure your code works correctly
# I strongly recommend you read and understand every line here now

# Instantiate networks (random weights!)
struct = [4,5,4,3] # input size 4, labels size 3
network = FeedForwardNeuralNetworkWithBackprop(struct)

# Prepare toy data as usual
npoints = 30 # dataset size
x = np.random.uniform(size=[npoints, network.nins])
y = np.random.uniform(size=[npoints, network.nouts])
mean_square_error = lambda x,y: ((y-x)**2).mean()

# Check errors of current network, before training
predictions_pre_train = network.act_net_v(x)
error_pre_train = mean_square_error(predictions_pre_train, y)
print("Error pre-train: ", error_pre_train)

# Train with Backpropagation!
network.backprop(x, y)

# Check errors again: have them gone down?
predictions_post_train = network.act_net_v(x)
```

```
error_post_train = mean_square_error(predictions_post_train, y)
print("Error post-train:", error_post_train)
```

```
Error pre-train:  0.13657835122821005
…
…
Error post-train: 0.037355804028937054
```

2.3 [3pt] Train a network to classify the Iris dataset using your implementation. Do not alter the code provided: only fix the missing parts, denoted by ?.

- We have three classes, we need to train 3 neurons on 4 inputs. Let's try without hidden layers.
- To match the labels to the output, encode the (discrete) class using one-hot encoding. I suggest you use `pd.get_dummies()`. (Note: for prediction we would typically use `np.argmax()` on the network output)
- Remember to drop the `species` column from the dataframe, and merge the dummy variables with one-hot encoding using `pd.merge()`. You want to align the rows/indices, using `left_index=True` and `right_index=True`.
- We are back to SL so you need both the `x` (for the forward pass) and the `y` (for the backprop). Then you are ready for the split.
- Your implementation of backprop may have problems with dataframes, in which case convert its inputs using `to_numpy()`.
- Writing a NN class simplifies greatly the introduction of the backprop code. However you are going to use the code only twice, so copy+paste is also acceptable. Just remember if not that you will need to define a new `struct` here, and that all the variables depending on it (and methods that use those outside-defined variables) should be redefined too. You are in for some nasty bugs if not, try killing the Jupyter kernel often and running only what you need.
- Feel free to experiment with learning rates. You can start with 0.1, but you could go as low as 10^{-5}.

[6]:
```
# First the data preparation as usual

iris = sns.load_dataset('iris')
# The feature 'species' is discrete with string values. We want a binary␣
 ↪encoding.
# We can create 3 columns with the names of the 3 species, and each row then␣
 ↪will
# have 2 `0` values and one `1`. Use `pd.get_dummies`!
species_enc = pd.get_dummies(iris['species'])
y_cols = species_enc.columns # save the names of the label columns
iris_feats = iris.drop('species', axis=1) # `drop` the original 'species' column
# Note: `iris_feats` now only contains the numeric (input) features
x_cols = iris_feats.columns # save the names of the input columns
# `merge` the features and encoding, set `left_index` and `right_index` to True
df = pd.merge(iris_feats, species_enc, left_index=True, right_index=True) # add␣
 ↪ints
```

```
train, test = train_test_split(df, test_size=0.2) # 80-20 split
x_train = train.loc[:, x_cols] # all rows from train, input columns
y_train = train.loc[:, y_cols] # all rows from train, label colum(s)
x_test = test.loc[:, x_cols]   # same with the...
y_test = test.loc[:, y_cols]   # ...test set here
```

[7]:
```
net = FeedForwardNeuralNetworkWithBackprop([4,3])
# Remember: for performance estimation you should not use the training set.
# Technically here is where you want to use the validation set, but we will
# use the test set since we have it handy and it's still consistent.
# Even in this case to show the error before training.
preds = net.act_net_v(x_test)
print("# Pre-train mean error:")
print(mean_square_error(preds, y_test))
# Then, super important, the training process only accesses the training set!
net.backprop(x_train.to_numpy(), y_train.to_numpy()) # because we use `zip`
# Now back to using the test set to showcase the new performance
preds = net.act_net_v(x_test)
print("# Post-train mean error:")
print(mean_square_error(preds, y_test))
```

```
# Pre-train mean error:
setosa        0.699947
versicolor    0.412497
virginica     0.265704
dtype: float64
…
…
# Post-train mean error:
setosa        0.000003
versicolor    0.102407
virginica     0.076005
dtype: float64
```

Think: what would you do to improve the results? If you think you can do better, COPY + paste below this message any cell you want to modify (so the solution above still gets you the points safely), and give it your best to minimize those errors! Also: try to copy the least amount of code you really need, it's a good practice.

4 3. First taste of Keras

I selected to use Keras here simply because it will be more easily available for everyone using Colab. You should be aware of Pytorch as a solid alternative. The trade off is typically between something easier to use for a quick prototype (e.g. Pytorch) vs. something that scales to bigger and more complex; Keras is founded on Tensorflow, which means more complexity to use it but access to more powerful tools (and Google support), though for a course with only local installations (and for quick sketches) I would have rather recommended Pytorch. Feel free to use either – be flexible

with your tools!
But I strongly advise you not to use one if you already have experience with it (if you already used Pytorch, use Keras here). Think about the difference between putting in your CV that you already have experience with one library vs. with none. Now think about writing that you have experience with both.

IF YOU USE PIPENV: you need to install the right package. I choose **to not distribute** an updated version of the Pipfile, so you have a chance to use the `pipenv install` command (keras is packaged within `tensorflow`), and to allow you to install whichever library works for your particular system. Since all solutions will be with Keras, doing this homework with Pytorch instead will allow you to see both versions. You will notice that there is not much difference for toy problems like this anyway.
Some resources: - https://keras.io/getting_started/ - https://www.tensorflow.org/install/pip - https://keras.io/examples/

3.1 [2pt] Train a network to classify the Iris dataset using Keras, and print the trained model accuracy.

- Use a sequential model with only one dense layer for simplicity
- Remember to correspond the number of neurons in the model('s output layer) to the number of classes
- The first layer explicitly needs the `input_dim=` parameter
- Explicitly use a sigmoid activation
- After finishing constructing the model, you need to `compile` it using an optimizer, a loss and a (list of) metric(s). Use stochastic gradient descent, mean squared error, and accuracy, respectively.
- The next step is the training, as usual the method is called `fit`. Pass a `validation_split` and it will take care of the split itself, plus it will allow visualizing the performance of the model on the test set at each epoch.
- You also want to pass `epochs` and `batch_size`. Values of `1000` and `5` work well, but feel free to experiment.
- Finally to print the model accuracy you will need to call `evaluate`, which will pick the `accuracy` measure from when you compiled the model.
- Use fewer epochs to test the code faster, 10-100 should work fine.
- You may not get lucky at every trial and thus need multiple runs. Yes this is actually accepted as common practice, as backpropagation provides no exploration capabilities and easily converges into the nearest local optimum.

[10]:
```
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(3, input_dim=4, activation='sigmoid'))
model.compile(optimizer='sgd',
              loss='mean_squared_error',
              metrics=['accuracy'])
history = model.fit(x_train, y_train, validation_split=0.1,
```

```
                    epochs=500, batch_size=5, verbose=1)
_, accuracy = model.evaluate(x_test, y_test)
print(f"\nAccuracy: {round(accuracy,4)}")
```

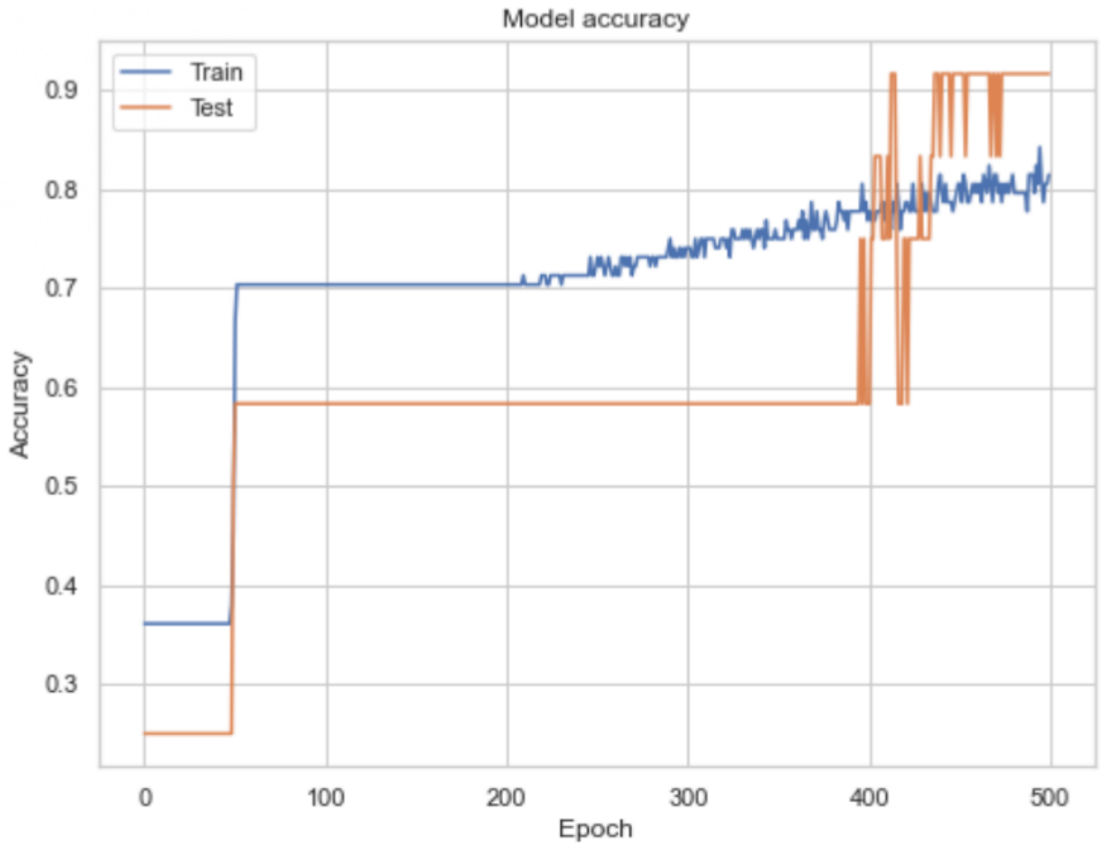
```
Epoch 1/500
22/22 [==============================] - 0s 5ms/step - loss: 0.3507 - accuracy:
0.3611 - val_loss: 0.4233 - val_accuracy: 0.2500
Epoch 2/500
22/22 [==============================] - 0s 2ms/step - loss: 0.3452 - accuracy:
0.3611 - val_loss: 0.4188 - val_accuracy: 0.2500
Epoch 3/500
22/22 [==============================] - 0s 2ms/step - loss: 0.3414 - accuracy:
0.3611 - val_loss: 0.4159 - val_accuracy: 0.2500
Epoch 4/500
22/22 [==============================] - 0s 2ms/step - loss: 0.3391 - accuracy:
0.3611 - val_loss: 0.4142 - val_accuracy: 0.2500
Epoch 5/500
22/22 [==============================] - 0s 2ms/step - loss: 0.3376 - accuracy:
0.3611 - val_loss: 0.4129 - val_accuracy: 0.2500
Epoch 6/500
22/22 [==============================] - 0s 2ms/step - loss: 0.3365 - accuracy:
0.3611 - val_loss: 0.4120 - val_accuracy: 0.2500
Epoch 7/500
22/22 [==============================] - 0s 2ms/step - loss: 0.3356 - accuracy:
0.3611 - val_loss: 0.4112 - val_accuracy: 0.2500
Epoch 8/500
22/22 [==============================] - 0s 2ms/step - loss: 0.3348 - accuracy:
0.3611 - val_loss: 0.4106 - val_accuracy: 0.2500
Epoch 9/500
22/22 [==============================] - 0s 2ms/step - loss: 0.3342 - accuracy:
0.3611 - val_loss: 0.4099 - val_accuracy: 0.2500
Epoch 10/500
22/22 [==============================] - 0s 1ms/step - loss: 0.3335 - accuracy:
0.3611 - val_loss: 0.4093 - val_accuracy: 0.2500
Epoch 11/500
22/22 [==============================] - 0s 2ms/step - loss: 0.3329 - accuracy:
0.3611 - val_loss: 0.4087 - val_accuracy: 0.2500
Epoch 12/500
22/22 [==============================] - 0s 2ms/step - loss: 0.3323 - accuracy:
0.3611 - val_loss: 0.4082 - val_accuracy: 0.2500
Epoch 13/500
22/22 [==============================] - 0s 2ms/step - loss: 0.3317 - accuracy:
0.3611 - val_loss: 0.4076 - val_accuracy: 0.2500
Epoch 14/500
22/22 [==============================] - 0s 1ms/step - loss: 0.3312 - accuracy:
0.3611 - val_loss: 0.4071 - val_accuracy: 0.2500
Epoch 15/500
```

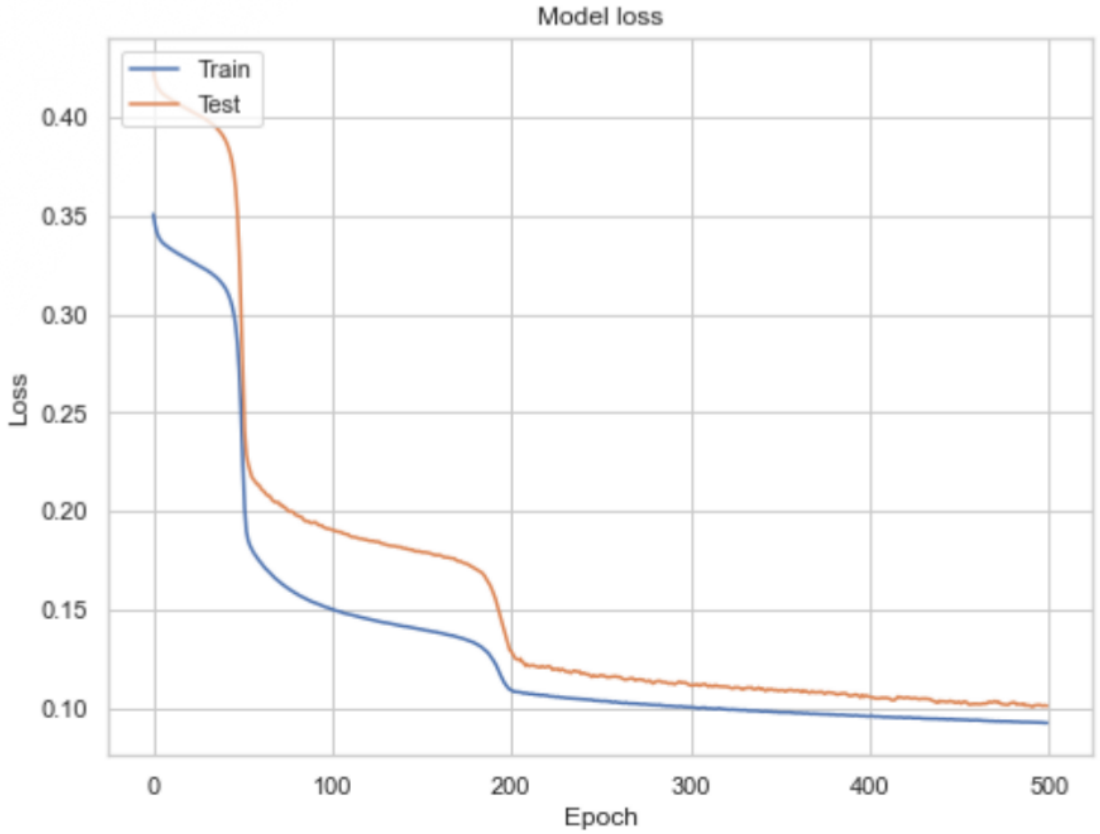
```
22/22 [==============================] - 0s 2ms/step - loss: 0.0931 - accuracy:
0.8426 - val_loss: 0.1020 - val_accuracy: 0.9167
Epoch 496/500
22/22 [==============================] - 0s 2ms/step - loss: 0.0931 - accuracy:
0.8056 - val_loss: 0.1020 - val_accuracy: 0.9167
Epoch 497/500
22/22 [==============================] - 0s 2ms/step - loss: 0.0931 - accuracy:
0.7870 - val_loss: 0.1016 - val_accuracy: 0.9167
Epoch 498/500
22/22 [==============================] - 0s 2ms/step - loss: 0.0930 - accuracy:
0.8056 - val_loss: 0.1015 - val_accuracy: 0.9167
Epoch 499/500
22/22 [==============================] - 0s 2ms/step - loss: 0.0928 - accuracy:
0.8056 - val_loss: 0.1018 - val_accuracy: 0.9167
Epoch 500/500
22/22 [==============================] - 0s 2ms/step - loss: 0.0929 - accuracy:
0.8148 - val_loss: 0.1015 - val_accuracy: 0.9167
1/1 [==============================] - 0s 16ms/step - loss: 0.1131 - accuracy:
0.6667

Accuracy: 0.6667
```

3.2 [1pt] Visualize the model's accuracy and loss over time.

- You can find a practical example [here].
- You should expect the accuracy to grow, the loss to decrease, and train and test performance to be related but different.
- Also with only 3 classes and few data points it's perfectly normal for the accuracy lines to look "discretized" (like a step function).

[11]:
```python
# 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
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0
100
200
300
400
500
Epoch



Note: it may take several hundreds of epochs to actually reach full convergence. Give it a try if you have the time. I will show an example in the solution.

5 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] Re-write from scratch the code for FeedForwardNeuralNetwork and FeedForwardNeuralNetworkWithBackprop. Cheat as much as you need by looking at the code provided, but of course the more you do by yourself, the better your training. Feel free to write a first version with naked methods, without the class, then add the class structure in a second step (incapsulation).

BONUS [ZERO pt] This exercise should already blur the line between "classification" and "regression". Go all the way and learn to predict the value of one of the four continuous features of the Iris dataset based on the other three.

BONUS [ZERO pt] A classic example is the XOR problem: write a neural network that maps two binary inputs to one binary output, and learn the 2D XOR logical operation. If you draw the four points, you will see they are not linearly separable. However you can write a neural network with one hidden layer of two neurons that solves the problem. Initialize the network with random weights, then execute the backpropagation algorithm by hand on paper until you solve it. This is a great exercise if you're stuck with the implementation of backprop and you cannot figure out what went wrong, as it forces you to get the dimensions right. Using 3 hidden neurons is a bit simpler and should require less iterations.

5.0.1 Final considerations

- The most important take-home message here is to distinguish between the *model* and the *learning*. You will find most people use "neural network" to refer to both together, which limits the understanding of either part in isolation and therefore each part's limitations and applicability. Be flexible.
- Spoiler alert: Deep Learning, all the way to ChatGPT, is just neural networks and back-propagation, with some fancy tricks (which either you saw before or will see in the next weeks):
  - Deep networks have many layers
  - Convolution
  - Transformers, a network architecture for time-series memory-aware processing
  - A sophisticated word embedding to enable textual input, check out for example BERT