Week 2: Attitude estimation with neural networks

Artificial Intelligence for Aerospace Engineering

AE-2224II - Q3 2022/3

Guido de Croon

February 23, 2023

This document contains the instructions for the lab session of week 2. Please read it page by page, working on the exercises as you go. The topic of this lab session is attitude estimation, a vital capability of any flying vehicle or animal. Attitude is defined as the orientation of the body frame with respect to the inertial frame (see Fig. 1 for an illustration of the roll angle of a flapping wing drone). Attitude estimation and control is especially relevant to unstable flyers like quadcopter drones. If attitude estimation is off, this will lead to quick accelerations and - when close to obstacles or the ground - a quick crash. Almost all drones use *inertial* sensors for attitude estimation. An Inertial Measurement Unit (IMU) typically contains accelerometers, gyros, and magnetometers. Especially the first two sensors play a big role in attitude estimation.

Figure 1: Illustration of attitude estimation on a flapping wing drone, from [1].

Aim: You will set up a neural network that has to estimate the attitude of a drone from the measurements of its IMU.

Dataset: The dataset consists of IMU readings from a quadcopter, from the scientific article:

[1] De Croon, G.C.H.E., Dupeyroux, J.J., De Wagter, C., Chatterjee, A., Olejnik, D.A., and Ruffier, F. (2022). Accommodating unobservability to control flight attitude with optic flow. Nature, 610(7932), 485-490.

You will find the files necessary to complete this instruction in directory `week2/attitude-estimation` of your docker container. From now on, all the paths will be relative to that directory.

Part 1: Setting up a multi-layer perceptron for learning regression

First, you will set up a multi-layer perceptron (MLP) for learning regression. Open the file `MLP_regression_attitude.py`, and go through the start of the file. It currently sets variable `sensor_data` to `False`, so that you will first be working with artificial data (a sine wave). This will help you to validate that your implementation of the MLP learning is correct. Code the following steps in the file `MLP_regression_attitude.py` (you can use the code from the lecture as a guide):

1. Create a variable `model` that represents an MLP with two hidden layers, each with 30 hidden neurons that have sigmoid activation functions. The MLP should have one output with a linear activation function (so no need to define the function). **Hint:** You can use `torch.nn.Sequential` to create a model with multiple layers in one go.

2. Create a variable `loss_fn` that represents the mean squared error loss function (do not write it yourself, but use torch's MSE loss function). No need to define the reduction variable, as the default value is `mean`.

3. Create a variable `optimizer` that represents the Adam optimizer. This optimizer is a variant of stochastic gradient descent (SGD) that will be explained in the lecture on deep neural networks. You can create the Adam optimizer by using `torch.optim.Adam`. Set the learning rate to a value that you think is appropriate.

4. Uncomment and set the variable `n_epochs` to an appropriate value. As a reminder, the number of epochs represents how often you want to present the entire training data set to the neural network training process.

5. Make a `for` loop that iterates over the number of epochs. In each iteration, you should: run the model on the inputs `X`, calculate the loss with the loss function, set the gradients to zero, calculate the gradients with `loss.backward()`, and update the parameters with the optimizer using the `step` method.

At the end of the code, the results are plotted, i.e., the neural network predictions are plotted together with the data points. By setting all relevant parameters, i.e., the number of epochs and learning rate, a good fit can be reached with a Root Mean Squared Error (RMSE) of smaller than 0.1. The fit should look like the one in Fig. 2.

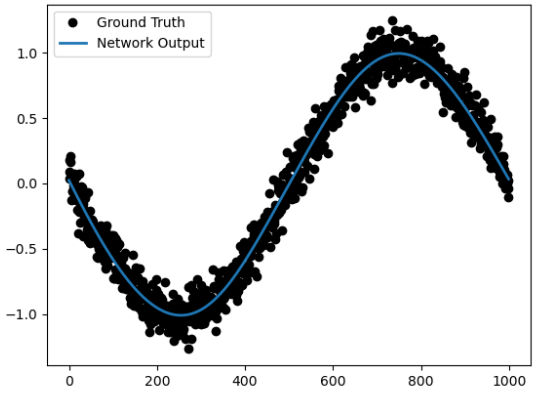

Figure 2: Fit of a sine wave with an MLP.

Part 2: Predicting attitude from IMU data with an MLP

Now, you will use the same code to predict the attitude of a drone from its IMU data. For this, change the variable `sensor_data` to `True`, and run the code. Does it work? After answering this question, go to the next page.

Part 3: Investigating the data

Although the error looks low, the plot reveals that the neural network is not able to learn the attitude from the IMU data. In order to find out why, investigate the data. Perform two tests:

1. Plot histograms of all the IMU readings and of the outputs. **Are the distributions of the IMU readings and the outputs of similar magnitude?** *Hint:* You can use the function `plt.hist` to plot histograms (Don't forget: `from matplotlib import pyplot as plt`). You can create subplots with the function `plt.subplot(n_rows, n_columns, current_index)` command.

2. Plot the IMU readings over time. **What do you observe?** *Hint:* You can use the function `plt.plot` to plot the data.

Part 4: Normalizing and smoothing the data

Since the inputs and outputs have very different magnitudes, it is a good idea to normalize the data. A typical procedure is to consider each variable as approximately normally distributed. Then first the mean can then be subtracted, and subsequently the data can be divided by the standard deviation. This is called standardization. **Can you think of a reason why this is not a good idea in the current case?**

Instead, we will use some domain knowledge here. We divide the sensor measurements (inputs) by 1024. Moreover, we will scale the attitude (outputs) from radians to degrees, by multiplying the outputs by $180/\pi$. Moreover, since the ground-truth roll attitude angle may have a bias, we subtract its mean. As a result, the outputs should have zero mean, which corresponds to our knowledge that the drone was hovering during the flight. We will also smoothen the measurements before feeding it to the network. One way of doing this is to use a moving average filter, another is to use a filter like the Savitzky-Golay filter. The latter can be implemented by including `from scipy import signal` and `signal.savgol_filter` in the code.

Retrain the network. **Does it lead to better results? How can you know if the result is good?**

In order to answer that last question, we will again use some domain knowledge. In the plots at the end, include an attitude estimated with the help of the accelerometers:

```
multiplication_factor = 180 / np.pi
y_domain = -np.arctan2(X[:,1], -X[:,2]) * multiplication_factor
y_domain = signal.savgol_filter(y_domain, smooth_window, 1)
```

Figure 3 shows the results with our code. The results show that the very crude arctangent attitude estimate is quite far off from the real. The neural network estimates look closer to the real attitude than the accelerometer estimate. The network in the figure had a mean squared error (MSE) of 3.59.

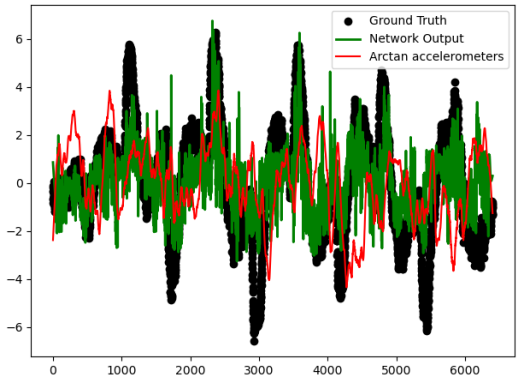

Figure 3: Attitude estimates with an ANN versus a smoothed estimate based on the arctangent.

Do you think it will help to include the gyros? You can try this out by setting `include_gyros` to `True`.

Surprisingly, the gyros do help to reduce the error (The results in Fig. 4 have an MSE of 1.69). The suprise lies in the fact that gyros capture rotation *rates*, i.e., how the attitude angle varies over time. Hence, one would expect that adding the gyros to the inputs does not help very much at this stage. Namely, an MLP is a *feedforward* neural network that maps inputs directly to outputs. It cannot integrate the inputs,

so it cannot integrate the angular changes over time. However, evidence in this case suggests otherwise. This may be due to *overfitting*, i.e., the network may be exploiting patterns that are only present in this small data set. In fact, the gyros may improve the performance, because the data is gathered on a drone that was successfully hovering based on a successfully working traditional attitude estimator. Hence, when the angle is positive, the drone will try to go to hover again by making the roll rate negative, and viceversa. This pattern may be exploited by the neural network.

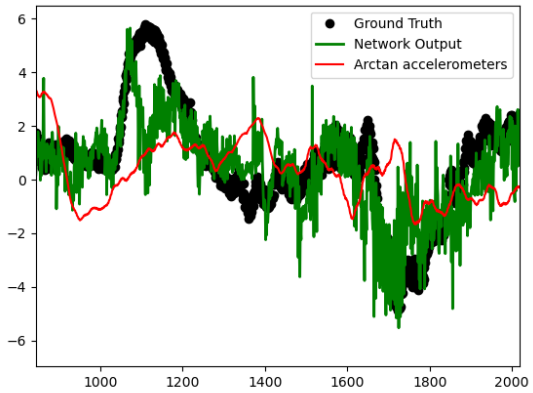

Figure 4: Attitude estimates with an ANN that has both accelerometers and gyros versus a smoothed estimate based on the arctangent.

What you should have learned until now (**learning objectives**):

1. How to set up and train an MLP for regression tasks.
2. How to inspect and normalize data. Large absolute magnitude inputs or outputs can be problematic for neural networks. Moreover, large relative magnitude differences between inputs can also be problematic. Remember that weights are initialized close to zero, and that a local search in parameter space is performed.
3. How to validate results with the help of domain knowledge. Although AI allows for automatic learning and good results, domain knowledge remains important. It allows to compare results from the network with traditional techniques and provides a means of validation of the results. For example, at this point, it is suspect that the gyros improve the results. Moreover, domain knowledge can provide guidelines as to how the neural network should be structured. The optional part below is an example of this, as we will introduce memory into the neural network.

Copy your code for Parts 1 and 4 in weblab and submit it. A typical result at this point in the figures in this section. An MSE lower than 4.0 will be accepted for the assignment. If you want and have time to go beyond the current results, you can do the optional part below.

OPTIONAL - Part 5: Predicting the attitude with a Recurrent Neural Network

The sensory inputs are time series, and the attitude is a function of the past. Hence, a recurrent neural network (RNN) is a better choice than an MLP. Recurrent neural networks have neurons that receive past neural activations as inputs. Therefore, they feature a memory of the past. In the current case, this is particularly important for using the gyros, which have to be integrated over time.

In order to use an RNN, we need to change the code. PyTorch has standard modules for recurrent neural networks, such as `nn.RNN`. However, they cannot simply be used in a `nn.Sequential` model. The reason for this is that the `nn.RNN` module returns two things, i.e., the activations and the outputs of the (recurrent) hidden layer. Moreover, the activations have to be re-used for calculating the layer's activations and outputs at the next time step. Hence, all samples in the batch have to be passed to the hidden layer one by one instead of at once. So, instead of using a `Sequential` module, we will use the `nn.RNN` module as a layer in a `nn.Module` class.

Please carefully inspect the class defined below. The `__init__` function initializes an RNN model, with its activation functions and the state of the hidden neurons `self.hidden_activations`. The `forward` function performs a forward pass through the network. The `reset_hiddens` function resets the hidden activations to zero. You can either copy the code below, or import the `RNN.py` file as module and then create the model with `model = RNN.RNN(n_features, n_hidden_neurons, N)`, where N is the batch size (sequence length).

```
# Set up a class for the recurrent neural network:
class RNN(torch.nn.Module):

    def __init__(self, n_inputs, n_hiddens, batch_size,n_outputs=1):
        super(RNN, self).__init__()
        self.n_hiddens = n_hiddens
        self.layer1 = torch.nn.Linear(n_inputs, n_hiddens)
        self.act1 = torch.nn.Sigmoid()
        self.layer2 = torch.nn.RNN(n_hiddens, n_hiddens)
        self.layer3 = torch.nn.Linear(n_hiddens, n_outputs)

        self.batch_size = batch_size
        self.hidden_activations = self.reset_hiddens()

    def forward(self, input, show_hiddens=False):

        # Reset the hidden activations:
        self.reset_hiddens()

        # How does this work?
        layer1_act = self.layer1(input)
        layer1_out = self.act1(layer1_act)
        layer2_out, self.hidden_activations = self.layer2(layer1_out, self.hidden_activations)
        if(show_hiddens):
            plt.figure()
            plt.plot(self.hidden_activations.view(self.batch_size, self.n_hiddens).detach().numpy())
            plt.show()
        output = self.layer3(layer2_out)

        return output

    def reset_hiddens(self):
        # reset the hidden activations to zero
        self.hidden_activations = torch.zeros(1, self.batch_size, self.n_hiddens)
        return
```

```
    def set_batch_size(self, batch_size):
        self.batch_size = batch_size
        self.reset_hiddens()
        return
```

Importantly, the batch that we pass to the network is a *sequence*. The `nn.RNN` module works such that it outputs the activations and the outputs of the hidden layer for each time step. In this lab, we will only pass one sequence to the network, but the RNN module can also be used to pass multiple sequences. Therefore, it expects a *3-dimensional* tensor as input with as indices the sequence index, the batch index (time step), and the sensory input index (e.g., accelerometer or gyro). This means that we have to reshape our inputs:

```
X = X.reshape(1, X.shape[0], n_features)
Y = Y.reshape(1, Y.shape[0], 1)
```

Moreover, before plotting, we need to reshape the outputs of the network with the help of the `view` function. This function reshapes a tensor without changing the number of elements. Passing -1 as argument means that the `view` function should find out itself what the correct dimension is for retaining all elements.

```
y_pred = y_pred.view(N, -1)
```

Train the RNN. **Can you now obtain a good performance on the data set?**

It is possible to get the result shown below in Figure 5, with an MSE of 1.89. This error is higher than that of the ANN in the previous section, and it can be questioned if the memory is actually used by the network.

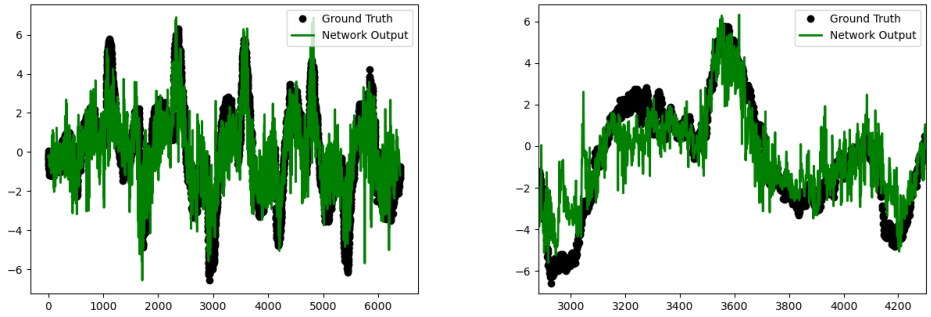

Figure 5: Fit of the roll attitude angle with an RNN.

OPTIONAL - Part 6: Advanced questions

For the fast and furious or the really passionate, here are some advanced questions:

1. **Did the network actually find a real solution to attitude estimation, or did it find a way to 'cheat' on this dataset? Can you devise a strategy to test whether the trained neural network generalizes beyond the training set? Is there a difference between the ANN and RNN for generalization?**

2. **The ground truth data is not perfect, as it is based on a complementary filter. Can you compare the performance of your neural network with a complementary filter of your own making?**