Exercise set 1

Exercises made by Oliver Gurney-Champion. Please contact us via Canvas, or e-mail directly to:

Oliver: o.j.gurney-champion@amsterdamumc.nl
Matthan: m.w.a.caan@amsterdamumc.nl
Daisy: daisyvandenberg@amsterdamumc.nl
Daan: d.kuppens@amsterdamumc.nl

Deadline: 23-4-2023 23:59.

These are a large set of challenging exercises, for which you will get 3 weeks to complete. I would strongly advise you to stick to the following schedule, which will ensure you have sufficient knowledge to answer the questions when completing them, and finalize all questions in time.

Week 1: Complete exercises A and B. Try to work on these before the practical on Friday, so you can prepare any questions.

Week 2: Complete exercise C.

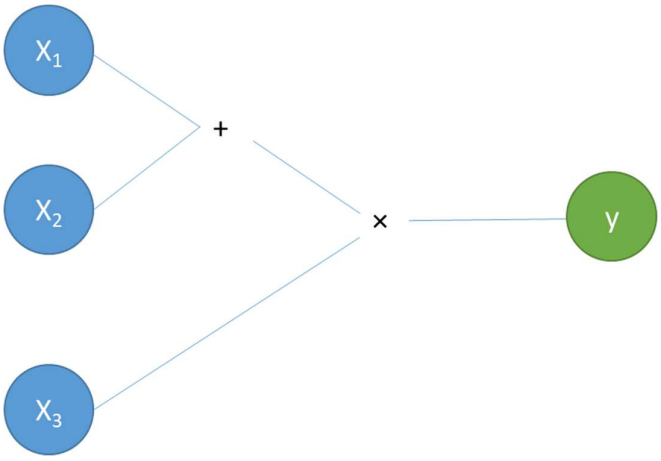
Week 3: Complete exercise D. I would advise you to complete the PyTorch tutorial before working on this.

Note that these networks will be lite and can run on your local computer/laptop. There is no need as yet to run this on Google Collab, although we highly encourage you to make sure Google Collab works for you (for exercise set 2 and 3).

0: general questions about Deep Learning

A. Feed forward loop calculation by pen **(10 %)**

Emily designed a network, which ended up in the following form:

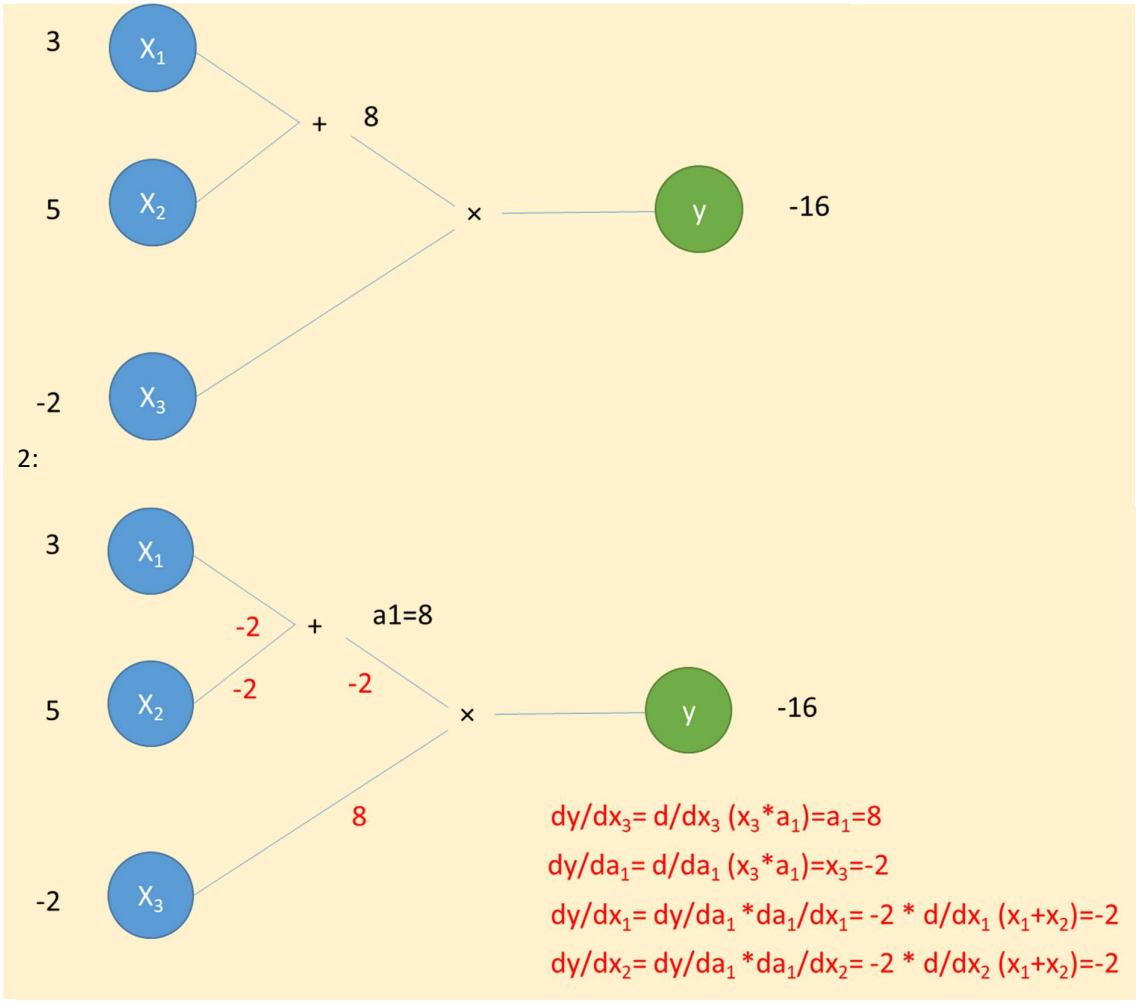


Example questions

1. Please calculate the forward pass for input x=[3, 5, -2]
2. Please calculate dy/dx1, dy/dx2 and dy/dx3 using backpropagation.

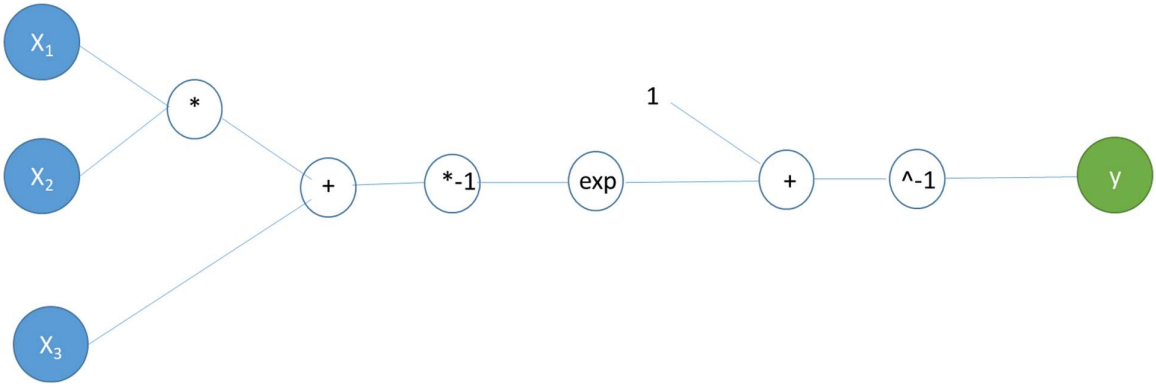
Answers:

1:



Now do it self

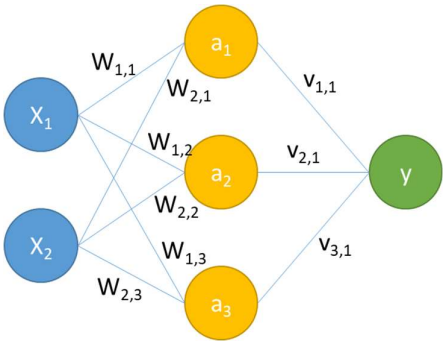
3. Calculate the forward and backward pass through this network using backpropagation using x=[2, -3, 8]. E.g. what is y, dy/dx1, dy/dx2 and dy/dx3?



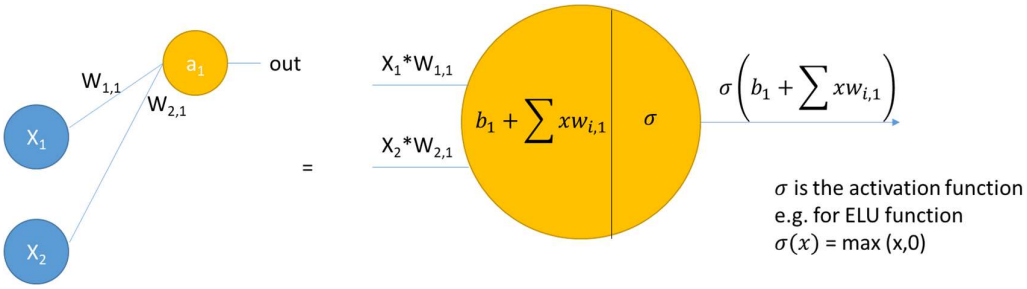
4. Instead of visualizing every mathematical step, one can also write y down in function form. Please give y(x1, x2, x3) and verify the derivatives from (3).

B: Neural network **(10 %)**

In Neural networks, we typically have layers of artificial neurons that are connected to each other. A simplified version is given bellow:



Where W is the weight matrix of the hidden state and V is the weight matrix of the output. At each node, signal is passed on in the following fashion



which, for a full layer of perceptron, can be written in matrix form as:

$$out = \sigma(w^T x + b)$$

1. Given weight vectors

$$W = \begin{bmatrix} 2 & -1 & 3 \\ 1 & 2 & 3 \end{bmatrix}$$

$$v = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix}$$

Bias vector

$$b = \begin{bmatrix} -2 \\ 3 \\ 4 \end{bmatrix}$$

and input x=[-2, 3], calculate the forward and backward pass. Assume no bias term on the output.

2. In neural networks, we typically need to know how x depends on some predefined loss function, L, to update W such that it performs better next round. Given a Loss function

$$L = (y - y_{true})^2$$

Calculate $dL/dw_{2,2}$ for a y_{true} = 35.

Exercise B2 just illustrated how one can update the network weights when using ground truth reference parameters. In practice, all weights will be updated using a computer, as we will do in our next exercise.

3: **Multilayer perceptron (MLP) (40 %)**

During the class, a brief introduction was given to quantitative imaging. In this exercise, you will program your first neural network that will help estimate quantitative MRI parameters from quantitative data. In particular, we will be looking at the intra-voxel incoherent motion (IVIM) model for diffusion-weighted MRI:

$$S(b) = S_0 \times \left((1-f) \times e^{-b \times D} + f \times e^{-b \times D^*}\right)$$

With *S* the measured signal, S_0 the baseline signal at *S(b=0)*, *f* the perfusion fraction, *D* the diffusion coefficient and *D** the pseudo diffusion coefficient. For more information on what the model means exactly and how it is used clinically, I would suggest reading Introduction to IVIM MRI | Radiology Key. But for the purpose of this exercise, it is just a model.

Normally, *f*, *D* and *D** (named Dp in the code) are obtained by fitting *S(b)* using least-squares fitting. But these approaches are known to be prone to noise in the data and often produce poor estimates.

Therefore, you will write a neural network that predicts *f* from a given *S(b)*. There are great tools available that take care of forward pass and back propagation, such as PyTorch, Karas and Tensorflow. However, for the purpose of this exercise, you are not allowed to make use of these tools. Instead, you will program the separate units of a neural networks yourselves using Numpy.

At https://github.com/oliverchampion/AI_for_medical_imaging_course you will find the Python assignment. As you can see, we have already provided a data-generator, some plotting tools to plot the training progress, and a unit-testing facility.

You will have to write the code for all spots indicating

```
########################
# PUT YOUR CODE HERE  #
########################

########################
# END OF YOUR CODE    #
########################
```

We advise you to work in the following order

1: Program the different neural network modules in “modules.py”. Use “do_unittest.py” to test whether the forward and backward passes are correct.

2: Combine all modules into a multilayer perceptron (MLP) in “MLP_numpy.py”. Use ELU activation functions after each fully connected linear module, except for the last module, after which a Tanh module should be used (constraining parameters between [-1, 1]). The number of layers and number of nodes per layer should be adjustable.

3: Train the neural network using your code in train_mlp_numpy. Use the mini-batch stochastic gradient descent algorithm (Differences Between Gradient, Stochastic and Mini Batch Gradient

Descent | Baeldung on Computer Science). Note, this is a small network and should typically take in the order of seconds per epoch.

4: Monitor training and test the performance using the "plot_results" script provided. Note the script is extensive and has a lot of functions you will not use at first. Initially, you will only have to provide: D_ref, f_ref, Dp_ref, f_pred, data, b_values and loss_train. As a sanity check, a network with 2 layers (64, 32), combined with a lr of 0.001, 30 epochs, and a batch size of 128 should work reasonably well and result in a systematic error of around 0.9% and a random error of around 11%.

5: Test the performance at different learning rates, network depth and layer widths. Which combination worked best? Why do certain combinations work, while others do not?

6: Hand in your Python code, a short overview of your findings in 4 and 5 (graphs are encouraged), and a discussion on these findings. Max 1 a4 of text, and max 1 a4 of figures.

Rules:

- Not allowed to use Pytorch/Keras/Tensorflow oid. Instead, use Numpy and program modules yourself
- Make only use of matrix multiplications, no for loops. Points will be deduced when for-loops are used where matrix multiplications were possible (exceptions: you can use 2 loops to loop over the epochs and training batches in *train_mlp_numpy.py* [but each n x m batch should go through the network in 1 go!]. You can use a loop over the MLP layers in *MLP_numpy.py)*.

Bonus suggestions (for higher than 8,0):

- Extend the network such that it also predicts D and D*. Note that the optimal learning-rate may differ per parameter, as the L2-loss can be very different.
    - You can experiment between having a single network predicting D, f and D*, or having separate networks per IVIM parameter.
- Add parameter constraints to the fit parameters
- Use the validation data (data_val) as an independent dataset (not seen during training)
- Training to the identical order of data each epoch may introduce some bias. Shuffling the data each epoch may help the training converge faster. Implement shuffling of the training data.

4: Program network in PyTorch. **(30%)**

Most of what you have done in the previous exercise, can be taken care of with (among other programs) PyTorch. I recommend you only start working on this exercise after watching the PyTorch tutorial from week 3.

1. Build the same MLP network as in exercise 3 using PyTorch. Note that you no longer need "Modules.py". Usefull pytorch commands are:
   nn.ModuleList() → to make a list of modules
   nn.Linear → generate a lineair layer
   nn.ELU → generate a ELU layer
   nn.Sigmoid → generate a sigmoid layer
   utils.data.random_split → split data and labels
   utils.data.DataLoader → an object containing the data
   optim.### → optim contains optimizers like adam and SGD

2. Train the MLP network. Make sure it trains on both devices by using the ".to(device)" command. If your computer does not have CUDA/GPU than this can only be tested in Colabnet/Surf.
3. Play around with learning rate. Also, besieged SGD, try using the ADAM optimizer. What setting combinations work well?
4. Add a batch normalization layer to the network. What does batch normalization do? Does this help?
5. Hand in your Python code, a short overview of your findings in 3 and 4 (graphs are encouraged), and a discussion on these findings. Max 1 a4 of text, and max 1 a4 of figures.

Bonus suggestions (for higher than 8,0):

- Extend the network such that it also predicts D and D*. Note that the optimal learning-rate may differ per parameter, as the L2-loss can be very different.
    - You can experiment between having a single network predicting D, f and D*, or having separate networks per IVIM parameter.
- Be creative and explore additional functionalities to the network, such as drop-out, or additional activation functions. Do they improve te performance?

5: Real-world-data and physics-informed loss **(10%)**

So far, you have been working with simulated data (taken care of by Data_loader.py). For such data, we know the ground truth values. However, in vivo, we have no way of knowing the ground truth. How will our network perform?

1. Use the network, as trained in (4) and apply it to real-world data which is provided by running "test_in_vivo.py". Note that alongside your plot (the first), also a conventional least squares fit is provided as a reference. How does your approach compare? Why do you think your particular approach would look better/worse?
2. Ideally, you would train your network on real-world data. However, in this particular case, it is hard to get gold standard references. Luckily, we can use our understanding of physics, and of how stuff "should behave" to work our way around this. You will redesign your network loss, such that it can train on data without any gold standard references! Instead of placing the L2 loss on f_pred v.s. f_ref, you will:
    a. Let the network predict 3 outputs, D, f and D*
    b. Forward the D_pred, f_pred and D*_pred through the IVIM formula to get S_pred. Note that you will need to implement the IVIM formula into PyTorch to allow for backpropagation.
    c. Define your loss as the L2_loss of S_pred v.s. S_ref.
3. Optimize the network's training using the real data provided ("test_in_vivo.py"; datatrain). Evaluate the network on the same data as in 1 (data, valid_id, bvalues = dl.load_real_data(eval=True)). How does it perform?

Bonus:

- For those that performed the bonus exercise in 3 or 4; how does optimizing the network in 5 compare to those in 3 and 4. Especially, was it easy to train a single network to predict all 3 parameters in 3, 4 and 5?