To get access to this week's code use the following link: https://classroom.github.com/a/EcZpQQJa

---

General constraints for submissions: Please adhere to these rules to make our and your life easier! We will deduct points if you fail to do so.

- Your code should work with *Python 3.8*.
- You should only fill out the *TODO-gaps* and not change anything else in the code.
- Add comments to your code, to help us understand your solution.
- Your code should adhere to the PEP8 style guide. We allow line lengths of up to 120.
- While working on the exercise, push all commits to the `dev` branch (details in assignment 1). Only push your final results to the `main` branch, where they will be automatically tested in the cloud. If you push to `main` more than 3 times per exercise, we will deduct points.
- All provided unit tests have to pass: In your *GitHub* repository navigate to *Actions* → your last commit → Autograding → education/autograding to see which tests have passed. The points in autograding only show the number of tests passed and have nothing to do with the points you get for the exercise.
- `for` loops can be slow in Python, use vectorized `numpy` operations wherever possible (see assignment 1 for an example).
- Submit *a single PDF* named `submission.pdf`. Include your matriculation numbers on the top of the sheet. Add the answers and solution paths to all non-coding questions in the exercise. Do not leave answers to any questions as comments in the code. You can use Latex with the student template (provided in exercise 1 / ILIAS) or do it by hand.
- Please help us to improve the exercises by filling out and submitting the `feedback.md` file.
- We do not tolerate plagiarism. If you copy from other teams or the internet, you will get 0 points. Further action will be taken against repeat offenders!
- Passing the exercises ($\geq 50\%$ in total) is a requirement for passing the course.

How to run the exercise and tests

- See the `setup.pdf` in exercise 1 / ILIAS for installation details.
- We always assume you run commands in the *root folder* of the exercise repository.
- If you use miniconda, do not forget to activate your environment with `conda activate mydlenv`
- Install the required packages with `pip install -r requirements.txt`
- Python files in the *root folder* of the repository contain the scripts to run the code.
- Python files in the `tests/` folder of the repository contain the tests that will be used to check your solution.
- Test everything at once with `python -m pytest`
- Run a single test with `python -m tests.test_something` (replace `something` with the test's name).
- To check your solution for the correct code style, run `pycodestyle --max-line-length 120 .`
- The scripts `runtests.sh` (Linux/Mac) or `runtests.bat` (Windows) can be used to run all the tests described above. If you are on Linux, you need execution rights to run `runtests.sh`.

---

1. A few pointers before you begin

Throughout this course, you'll be working with Numpy arrays, so we figured it's important that everyone gets a good understanding of how dot products and broadcasting work in Numpy. The following are some useful links to help you with the topic:

1. Broadcasting in Numpy (the illustrations of broadcasting concepts in this page are particularly helpful)

2. Documentation of numpy.matmul

It's worth noting that the exact same Tensor Math will also happen in PyTorch, which you will be using in later exercises.

Also, tensor-sensor is a very useful python package to visualize tensor multiplications. You can use `tsensor.explain()` for visualizing a working operation and `tsensor.clarify()` for debugging errors. Here's an example of how it can help you figure out where your tensor operations are going wrong:

NumPy

```
[3]: import numpy as np

n = 200          # number of instances
d = 764          # number of instance features
n_neurons = 100 # how many neurons in this layer?

W = np.random.rand(d,n_neurons)
b = np.random.rand(n_neurons,1)
X = np.random.rand(n,d)
Y = W @ X.T + b
```

```
---------------------------------------------------------------------------
ValueError                                Traceback
(most recent call last)
<ipython-input-3-1cab907988b7> in <module>
      8 b = np.random.rand(n_neurons,1)
      9 X = np.random.rand(n,d)
---> 10 Y = W @ X.T + b

ValueError: matmul: Input operand 1 has a mismatch in
its core dimension 0, with gufunc signature (n?,k),
(k,m?)->(n?,m?) (size 764 is different from 100)
```

NumPy + TensorSensor

```
[4]: import tsensor
W = np.random.rand(d,n_neurons)
b = np.random.rand(n_neurons,1)
X = np.random.rand(n,d)
with tsensor.clarify():
    Y = W @ X.T + b
```

```
Y =  W  @ X.T + b
     100    200
764
764
```

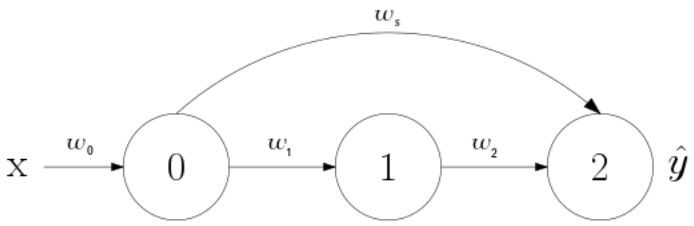
```
---------------------------------------------------------------------------
ValueError                                Traceback
(most recent call last)
<ipython-input-4-ead242a749fb> in <module>
      4 X = np.random.rand(n,d)
      5 with tsensor.clarify():
----> 6     Y = W @ X.T + b

ValueError: matmul: Input operand 1 has a mismatch in
its core dimension 0, with gufunc signature (n?,k),
(k,m?)->(n?,m?) (size 764 is different from 100)
Cause: @ on tensor operand W w/shape (764, 100) and o
perand X.T w/shape (764, 200)
```

See this link for details on installing tensor-sensor.

2. Pen and Paper Tasks

Perform a forward and backward pass to calculate the gradients for the weights w_0, w_1, w_2, w_s in the following MLP. Each node represents one unit with a weight w_i, $i \in \{0, 1, 2\}$ connecting it to the previous node. The connection from unit 0 to unit 2 is called a **skip connection**, which means unit 2 receives input from two sources and thus has an additional weight w_s. The weighted inputs are added before the nonlinearity is applied. You can refer to this blog post for a more intuitive understanding of backpropagation. In case of multiple outgoing connections from a neuron (for eg, forward and skip connections), the final gradient can be computed by accumulating the incoming gradients via these multiple connections as a direct consequence of the chain rule.



We assume that we want to solve a regression task. We use an L1-loss $L(\hat{y}, y) = |y - \hat{y}|$ The nonlinearities for the first two units are rectified linear functions/units (ReLU): $g_0(z) = g_1(z) = \begin{cases} 0, z < 0 \\ z, else \end{cases}$.

We do not use a nonlinearity for the second unit: $g_2(z_2) = z_2$ and there are **no biases**.

Note: We use the notation of the Deep Learning book here, i.e. $z = Wx + b$. If you attended the Machine Learning course, you might be used to the different notation used in the Bishop Book, where z denotes the value after applying the activation function. Here, z is the value before applying the activation function. Your tasks:

1) [2 points] Perform the backpropagation algorithm for the above network.

   First perform the forward pass, followed by the backward pass to obtain gradients for the weights w_0, w_1, w_2, w_s. Please assign equations/values to variables step by step similar to the lecture, and reuse

them later when possible to make your solution more readable.(see slide 23, Lecture 3)

2) [1 point] Explain what difference the skip connection makes when propagating back the error.

3) [2 points] Calculate the gradients for the given datapoint and the given initial weights (calculating the gradients requires to calculate a forward pass first). Also calculate the weights and the loss after one gradient descent step.

$$(x_1, y_1) = (1, -3)$$
$$w_0 = w_1 = w_2 = w_s = 0.5$$
$$LearningRate = 1$$

Hint: Gradient descent is the simplest method to train/optimize your neural network. Given some learning rate α and the gradient $\frac{\partial L}{\partial w}$ for a weight w, we update the weight like this: $w = w - \alpha \frac{\partial L}{\partial w}$. More details on optimization will follow in week 4.

3. Coding Tasks

The coding tasks build on the previous exercise. We complete implementing a small feedforward neural network and then use backprop to obtain the gradients and update the weights of the network for the XOR dataset as last time. We provide code for structure and utility and you have to fill in the TODO-gaps.

In the previous exercise, you used the Parameter and Module classes and implemented their forward passes. This time you will implement their backward passes.

Note: An additional utility module is used to check the correctness of your implementation by approximating **backward** with finite difference approximations

Nonlinearities: Implement the backward passes for the Modules below. The backward() function of a Module receives a grad argument from the Module after it in the network and using the chain rule calculates the gradient to be passed to Modules before it. This way the gradient information flows backward through backpropagation. You may refer class `Sigmoid` in file `lib/activations.py` to get an idea of what it looks like.

First, we will need a function that can set our gradients to zero so we can start updating them. Your tasks:

1) [1 point] **Todo:** Implement a function to reset the gradients of parameters. Fill the gaps in function `zero_grad` in file `lib/gradient_utilities.py`. Run `python -m tests.test_zero_grad` to test your implementation.

2) [1 point] **Todo:** Implement the backward pass for the ReLU module (class `Relu` in file `lib/activations.py`). Run `python -m tests.test_gradient_relu` to test your implementation.

3) [2 points] **Todo:** Implement the backward pass for the Linear layer (class `Linear` in file `lib/network.py`). Remember that the Linear layer holds objects of the Parameter class and you would need to update their gradients during the backward pass.

   Run `python -m tests.test_gradient_linear` to test your implementation.

4) [2 points] **Todo:** Implement the backward pass for Cross Entropy module (class `CrossEntropyLoss` in file `lib/losses.py`).

   This module calculates the softmax on the input and then the loss, so you will need to calculate the

gradient based on the combined expression.

Note: We decided this derivation is too difficult. You can find the original question and the solution in file `celoss_derivation.pdf`. Use that solution to implement the backward pass.

Run `python -m tests.test_gradient_celoss` to test your implementation.

5) [1 point] **Todo:** Implement backward pass for Sequential module (class `Sequential` in file `lib/network.py`).

Gradient Check: Gradient checking is a useful utility to check whether the gradients obtained through finite differences and backward pass are matching. We have implemented the gradient checking in the Module class for you. As all classes we defined up to here inherit from Module, we can run check_gradients to check if you have implemented their backward passes correctly. Your task:

6) [1 point] **Todo:** Instantiate a Sequential network and perform gradient check on it (complete `check_gradients()` in file `lib/gradient_utilities.py`).

Experiments: We use the XOR dataset from last time and perform backpropagation on the one hidden layer model from last time, except that we now use two output units to perform binary classification as multi-class classification. First, perform one step of backprop and use the gradient you obtain to update the weights of the network. See how the parameters and their gradients change in the network. Then perform mutiple steps and see how the loss on the dataset evolves. Your tasks:

7) [1 point] **Todo:** Perform backward pass to obtain gradients and then use them to update the parameters of the network. Fill the gaps in function `backward_pass` in file `lib/experiments.py` and run `run_xor_experiment.py`.

   Run `python -m tests.test_backward_pass` to test your implementation. Note that the test will pass only if the implementations of `Linear` and `CrossEntropyLoss` modules are correct

4. Questions on experiments

Answer the following questions based on the experiments above:

1) [½ point] What do you observe with regards to the loss after 1 step of updating parameters?

2) [½ point] What do you observe after multiple steps of updating parameters? Does the loss always decrease? Explain why it may not.

3) [½ point] Run the experiment multiple times. Do you always end up with the correct final predictions? Explain why or why not?

4) [½ point] What is the role of the variable lr above? When would you set it to a relatively larger value and when would you set it to a relatively smaller value? Explain.

5. [1 bonus point] Code Style

On each exercise sheet, we will also be using `pycodestyle` to adhere to a common Python standard. `pycodestyle` checks your Python code against some common style conventions in PEP 8 and reports any deviations with specific error codes.

Your code will be automatically evaluated on submission (on push to `main`). You can run `pycodestyle`

`--max-line-length=120 .` to manually evaluate your code before submission.

One bonus point will be awarded if there are no code style errors detected in your code by the `pycodestyle --max-line-length=120 .` command.

6. [1 bonus point] **Feedback**

Todo: Please give us feedback by filling out the `feedback.md` file.

- Major Problems?
- Helpful?
- Duration (hours)? For this, please follow the instructions in the `feedback.md` file.
- Other feedback?

This assignment is due on 13.11.2024 23:59 CET. Submit your solution for the tasks by uploading (`git push`) the PDF, txt file(s) and your code to your group's repository. The PDF has to include the name of the submitter(s). Teams of at most 3 students are allowed.