Programming assignment 4: Software for neural network training

In this assignment, we will take a deep look at the fundamental mechanics of a neural network software framework. To do this, we are going to implement a small software framework that imitates PyTorch, one of the most important machine learning libraries.

You should structure your solution as a **notebook**. You have two choices: either you work on your own machine using a Jupyter notebook. (You can work with Jupyter directly or use the notebook support of popular IDEs such as PyCharm or VS Code.) Alternatively, you use the Colab service, which works in a similar fashion. (Here is a document about Colab we prepared for another course. You can ignore the GPU-related parts for now.) In case you use Colab, you can submit your solution either by downloading the notebook (`.ipynb`) or by providing a link to the notebook. (In that case, make sure you allow access to the notebook.) In addition, for the convenience of the grader, please also submit a **pdf printout** of the same notebook.

When you work with the notebook, you will enter your solution in the code cells. It is more or less a pure programming assignment and the documentation requirements are minimal: we expect the technical solutions to be self-explanatory.

This is a **group assignment** and your work should be carried out in a group of 2 or 3 people. If you don't have a group partner, please ask around in the Discussion tool in Canvas. Please include **the names of the group members** in your submission.

Submit your solution through the Canvas website.

Deadline: February 23

Didactic purpose of this assignment:

- getting to know the most important ideas in a neural network library (PyTorch);
- understanding the mechanics of training neural networks on a fundamental level.

That we are able to do our computation.
Remember that we need to store our computations
And then move to the parameter transfer
Task 4: He will create extra reading material, on how task 4 should be done.
Task 5: Extend your framework for a classification model with hidden layers.
Fully optional: Imporving the PyTorch Code.

This section is completely new and so we can ask for clarifications.

References

- Slides about Neural network training.
- Notebook about PyTorch fundamentals.
- Notebook about classification with PyTorch.

In addition, you may look at the chapter about neural networks in the Lindholm et al. book. However, their explanation of backpropagation is not aligned with the PyTorch-like approach we will be using in this assignment.

In order to make it easier to solve Task 4 in particular, we have prepared the following document that spells out the backpropagation algorithm in a way that relates to this assignment.

This is a new assignment and if there is anything that needs more explanation, feel free to write to Richard over email or ask in the discussion forum.

Preliminaries

If you prefer to run the code locally on your machine, download this package. It includes a notebook that contains the skeleton of a solution. In addition, the package includes a couple of CSV files with data for the two experiments. If you run the notebook locally, you will need to make sure that PyTorch is installed. (This page shows the installation commands via pip or conda.)

Alternatively, if you prefer to use Colab rather than running your code locally, you can make a copy of this notebook. There are cells in this notebok that download the CSV file to the local file system of your Colab virtual machine.

Task 1: A small linear regression example in PyTorch

Our first goal is to train a linear regression model using PyTorch. The purpose of this is to serve as a reference point for what you implement later.

In this task, we will use a small synthetic dataset. The notebook includes a cell that loads this dataset.

Next, there is a notebook cell that includes a skeleton of the training loop based on PyTorch.

Your work: Implement the missing pieces to complete the training loop and execute this code.

Hint: Use the matrix multiplication operator `@` in the forward computation. This will make it easier to be compatible with what you do in Tasks 2-4. Do not use PyTorch utilities such as `nn.Linear`: we should express our computations as tensor operations.

✓ **Sanity check.** If your solution is correct, you should have a mean squared error of about 0.009 after 10 epochs. If your MSE does not decrease, there must be an error!

Task 2: Implementing tensor arithmetics for forward computations

We will now gradually build our own PyTorch-like framework. Our first target will be to implement the basic functionality of tensors, so that we can carry out the forward computations.

Take a look at the class `Tensor`. As in PyTorch, a `Tensor` serves two main purposes: 1) it stores the numerical values that we have computed, and 2) it keeps track of the computations that were used to compute these values. Right now, we focus on the first of these purposes.

✓ **Your work:** Complete the missing code. You should implement all the arithmetic operations so that we can carry out the forward part of the training loop in Task 1. (The backwards step computations will be addressed in Tasks 3-4.) As a starting point, the addition operation has already been implemented: the `__add__` method inside the `Tensor` and then the helper function `addition` below.

Explanation: The methods `__add__`, `__sub__`, `__matmul__`, and `__pow__` are special methods that define what happens when we use the arithmetic operators `+`, `-`, `@`, and `**` respectively. This is an example of *operator overloading* in Python. The reason why we have them is that we want to be able to write simple formulas involving tensors such as `x + y`, `x @ w`, etc.

Sanity check. Create some tensors and make sure that you can compute the arithmetic operations that you used in the linear regression examples above. There is also a cell that carries out some basic tests. If you see an error when executing this cell, you have some problem in your implementation.

Task 3: Building the computational graph

We will now build the data structure that keeps track of how a `Tensor` was computed: the computational graph.

Take a look at the cell corresponding to Task 3. This cell defines a class called `Node` that will serve as the superclass of all computational graph nodes. There is also a node type called `AdditionNode`: this type of node will be used when we have carried out an addition of two tensors. As you can see, it stores references to the two tensors (`left` and `right`) that were used in this computation.

The function `addition` implements tensor addition, and you implemented similar functions for the other tensor arithmetic operations. In the provided code in `addition`, there is a missing piece. You will just create a node here (an instance of the class `AdditionNode`) and use this as `grad_fn`.

Your work: Implement the missing part in `addition` and then computational graph nodes for the other arithmetic operations as well.

Sanity check. The notebook contains a test case that creates three tensors, carries out a computation, and checks that you have built a reasonable graph.

Task 4: Implementing the backwards computations

Now that we have set up the computational graph, we are ready to do the heavy lifting and implement the backpropagation algorithm.

Backpropagation on a user-defined computational graph becomes a recursive algorithm. The recursion is initiated when the user calls `.backward()` on some tensor, typically the tensor holding the output of the loss function in a training loop. The recursive algorithm then walks backward over the computational graph, going from a `Tensor` to its computational graph node, and then from the node to the other `Tensors` it refers to. Along the way, the algorithm computes the gradient with respect to the current `Tensor`. Eventually, we reach leaf nodes (end points of the computational graph) that hold trainable parameters or input data, where the recursion terminates.

Your work: Implement the recursive algorithm as described above. You will have to work at two places: 1) in the `backward` method of `Tensor`, and 2) in the `backward` methods of all the node classes (`AdditionNode` and the others you defined in Task 3). Each call to `backward` except the initial one will take an input called `grad_output` that represents the *gradient with respect to this tensor*. At

the leaf nodes where `requires_grad` is set to `True`, make sure to store the gradient as an attribute `.grad`.

Hint. The additional reading material linked above will be useful here.

Sanity check. The notebook includes another test case. Here, we carry out some computations, then call `.backward()`, and check that you have stored correct gradients at the right places.

Task 5: Optimizers to update the model parameters

The final piece is to use the computed gradients to update the model.

A code cell in the notebook defines a general class `Optimizer` corresponding to the PyTorch equivalent.

Your work: Implement your own equivalent of `torch.optim.SGD`. Then create a training loop that looks exactly like what you did in Task 1 (except PyTorch-specific prefixes such as `torch.`, `torch.optim.`).

Sanity check. If your implementation is correct, your result should be ***identical*** to what you got with PyTorch in Task 1.

Task 6: Classifying raisins (only required for a higher grade)

This task can be skipped if you are content with a minimal pass.

Our final task is to build a model that classifies raisins. Each individual raisin is described using 7 numerical features and will be classified as one of two types. The dataset was originally collected by a team in Turkey and is described in this paper. The notebook contains a cell that loads this dataset from a csv file and uses scikit-learn to carry out basic preprocessing.

Your work: Your job now is to implement a classifier that solves this task using your own framework. This model should be implemented as a feedforward neural network for binary classification. It should have one hidden layer with tanh activation, and be trained by minimizing the binary cross-entropy loss. Extend your framework so that you can implement this classifier. You will need to implement new functions and computational graph nodes to deal with the new requirements. Finally, evaluate the trained classifier on the test set.

Sanity check. The accuracy on the test set will vary depending on your implementation. We believe that a reasonable implementation should have an accuracy above 0.8.

Fully optional tasks: Improving the PyTorch code

These tasks are completely optional and will not affect your score. They are just tasks for self-study to give some context to what we have done here.

The PyTorch code above is very much a bare-bones solution designed to make the fundamentals of tensors and gradient computations very clear. However, in

practical solutions, we typically use more high-level utilities to make the code more concise and efficient.

For an effective use of PyTorch in real applications, the following issues are worth looking into:

- Using minibatches instead of single instances. Using a DataLoader to split the dataset into minibatches.
- A more object-oriented coding style. Using `nn.Module` instead of raw parameter tensors.
- Better optimizers such as Adam.
- Using the GPU.
- Dropout regularization.
- Speeding up computations at test time by disabling the gradient computations.
