INF265
Project 1:
Backpropagation and Gradient Descent

Deadline: February 25th, 23.59
Deliver here:
`https://mitt.uib.no/courses/33548/assignments/56010`

Projects are a compulsory part of the course. This project contributes a total of 15% of the final grade. **The project is done in pairs. However, you are allowed to do it alone if you cannot find a pair**. If you do the work with a pair, add a paragraph to your report explaining the division of labor (Note that both students will get the same grade regardless of the division of labor).

Sections 2 and 3 are independent of each other, their respective tasks as well.

Code of conduct: You are allowed to copy-paste code from the solutions of previous weekly exercises. However, it is not allowed to copy-paste from online sources nor from other students' projects. To some extent, discussions on parts of the project with other pairs/students are tolerated. If you do so, indicate with whom and on which parts of the project you have collaborated. Sections 2.3 and 3.2 provide some hints, if you need additional assistance, teaching assistants and group leaders are available to help you.

Grading: Grading will be based on the following qualities:

- Correctness (your answers/code are correct and clear)
- Clarity of code (documentation, naming of variables, logical formatting)
- Reporting (thoroughness and clarity of the report)

Deliverables: You should deliver 3 files:

- `backpropagation.ipynb` addressing section 2. Cells should already be run and output visible.
- `gradient_descent.ipynb` addressing section 3. Cells should already be run and output visible.
- A PDF report, addressing section 4. Note that exporting your notebooks as a PDF is not what is expected here.

If you need to provide additional files, include a `README.txt` that briefly explains the purpose of these additional files.

Late submission policy: All late submissions will get a deduction of 2 points. In addition, there is a 2-point deduction for every starting 12-hour period. That is, a project submitted at 00.01 on February 26th will get a 4-point deduction and a project submitted at 12.01 on the same day will get a 6-point deduction (and so on). All projects submitted on February 28th or later are automatically failed. (Executive summary: Submit your project on time.) There will be no possibility to resubmit failed projects so start working early.

1 Introduction

Objectives of this project

In this project, you will implement the gradient descent (Eq.3) as a part of the neural network training process in two steps:

- Implementation of the backpropagation algorithm to compute $\nabla L(\theta)$.
- Manual weight update inside the training loop.

Objectives include **a)** getting a better understanding of the training process, (hyper-) parameters involved and PyTorch corresponding methods **b)** learning how to set up a basic machine learning pipeline, in particular model selection and model evaluation **c)** learning how to carry out reproducible experiments and how to interpret your results. **Neural network training seen as an optimization problem**

A neural network is trained by iteratively updating its weights such that the output of training samples get *closer* and *closer* to their expected output. This is a general optimization problem that consists in minimizing a loss function L which describes how far the outputs are from the expected results. If the loss function is the mean squared error, we have:

$$\begin{aligned} L(\theta) &= \frac{1}{m}\sum_{s=1}^{m} ||\mathbf{y_s} - \hat{\mathbf{y}}_\mathbf{s}||_2^2 = \frac{1}{m}\sum_{s=1}^{m}\sum_{i=1}^{n}(y_{i,s} - \hat{y}_{i,s})^2 \\ &= \frac{1}{m}\sum_{s=1}^{m}\sum_{i=1}^{n} e_i(y_{i,s}, \hat{y}_{i,s}) \qquad \text{with } e_i(y_{i,s}, \hat{y}_{i,s}) = (y_{i,s} - \hat{y}_{i,s})^2 \end{aligned}$$

with:

- m: total number of samples in the dataset (or in the batch)
- θ: all the parameters to be optimized ($\in \mathrm{R}^q$)
- $\mathbf{y}$: expected result ($\in \mathrm{R}^{n\times m}$)
- $\hat{\mathbf{y}}$: predicted result ($\in \mathrm{R}^{n\times m}$)
- $L : \mathrm{R}^q \to \mathrm{R}$: loss function

From now on, we will limit our scope to $m = 1$ (mini-batch). The formula is then simply:

$$L(\theta) = ||\mathbf{y} - \hat{\mathbf{y}}||_2^2 = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{1}$$

$$= \sum_{i=1}^{n} e_i(y_i, \hat{y}_i) \qquad \text{with } e_i(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2 \tag{2}$$

Gradient descent

This loss function (Eq.1) can be minimized using gradient descent, a general optimization algorithm in which parameters are iteratively updated as follows:

$$\theta_t = \theta_{t-1} - \alpha \nabla L(\theta_{t-1}) \tag{3}$$

where:

- α is often called *step* in optimization and *learning rate* in machine learning
- $\nabla L(\theta)$ is the gradient of the loss function. If $\theta = \left[w_1 \cdots w_q\right]^T$, then $\nabla L(\theta) = \left[\frac{\partial L}{\partial w_1}(\theta) \cdots \frac{\partial L}{\partial w_q}(\theta)\right]^T$

2 Backpropagation

In machine learning, the backpropagation algorithm is used to compute $\frac{\partial L}{\partial w_i}$ for all weights w_i of a neural network from the output layer to the input layer (hence the name *backpropagation*). Following Andrew's notations (which can be found in the fourth tutorial as well):

$$\frac{\partial L}{\partial w_{i,j}^{[l]}} = \delta_i^{[l]} \times a_j^{[l-1]} \quad \forall l \in [1..L] \qquad \text{with } \delta_i^{[l]} \textit{ local gradient}\text{: } \delta_i^{[l]} = \frac{\partial L}{\partial z_i^{[l]}} = \frac{\partial L}{\partial a_i^{[l]}} \times \frac{\partial a_i^{[l]}}{\partial z_i^{[l]}} \tag{4}$$

For the output layer `L`, since $a_i^{[L]} = \hat{y}_i$, we have:

$$\delta_i^{[L]} = \frac{\partial L}{\partial \hat{y}_i} \times \frac{\partial \hat{y}_i}{\partial z_i^{[L]}} = e_i'(\hat{y}_i) \times f_i'^{[L]}(z_i^{[L]}) \qquad \text{with } e_i'(y_i, \hat{y}_i) = -2 \times (y_i - \hat{y}_i) \tag{5}$$

For hidden layers `l` (general case):

$$\delta_i^{[l]} = \frac{\partial L}{\partial a_i^{[l]}} \times \frac{\partial a_i^{[l]}}{\partial z_i^{[l]}} = \Big(\sum_{k=1}^{n^{[l+1]}} \frac{\partial L}{\partial a_k^{[l+1]}} \frac{\partial a_k^{[l+1]}}{\partial a_i^{[l]}} \Big) \times \frac{\partial a_i^{[l]}}{\partial z_i^{[l]}} = \Big(\sum_{k=1}^{n^{[l+1]}} \delta_k^{[l+1]} w_{k,i}^{[l+1]} \Big) \times f_i'^{[l]}(z_i^{[l]}) \tag{6}$$

Note that for biases $b_j^{[l]}$, we simply have: $\frac{\partial L}{\partial b_j^{[l]}} = \delta_j^{[l]} \times 1$

2.1 Tasks

Using equations (4), (5) and (6), write a function `backpropagation(model, y_true, y_pred)` that computes:

- $\frac{\partial L}{\partial w_{i,j}^{[l]}}$ and store them in `model.dL_dw[l][i,j]` for $l \in [1..L]$
- $\frac{\partial L}{\partial b_j^{[l]}}$ and store them in `model.dL_db[l][j]` for $l \in [1..L]$

A vectorized implementation of these equations is appreciated.

2.2 Provided code

In order to have access to activation values, activation functions and their derivatives, an implementation of a MLP is provided in `backpropagation.ipynb`, in the `MyNet` class. In this section, `model` refers to an instance of this `MyNet` class. You are highly encouraged to read this class carefully before writing your `backpropagation` function. Write your `backpropagation` function in the same file, `backpropagation.ipynb`.

Once your implementation is complete, you can test it by running and checking the output of the last 2 cells of the notebook. The test functions used in these cells are defined in `tests_backpropagation.py`. You do not have to (nor need to) read the content of this file. The test procedure includes a comparison with PyTorch autograd's computations as well as a comparison with gradient values computed using the finite differences method (gradient checking method).

2.3 Hints

- To prevent PyTorch from computing any unwanted gradients, wrap all your computations inside a "`with torch.no_grad():`" context.
- Remember that in PyTorch, the first dimension is always the batch size and that our scope here is limited to batches of size one. Some tensors will then naturally have an extra dimension. For instance `model.a[l]`, `model.z[l]` have shape `(1, n(l))` and `y_true`, `y_pred` have shape `(1, 2)`.

- Gradients should have the same shape as their corresponding parameter. In particular, weights at layer `l` have shape `(n(l+1), n(l))` while biases have shape `(n(l+1))`.
- Test your `backpropagation` function by running the last cells of `backpropagation.ipynb`.

3 Gradient Descent

In the training process, the objective is to iteratively update weights θ such that the loss L gets lower, $L(\theta_{next}) < L(\theta_{curr})$.

$\nabla L(\theta)$ represents the direction in which the function L rises most quickly from a given θ. Therefore, to find θ_{next} from θ_{curr}, we should follow the direction in which L decreases most quickly, that is to say $-\nabla L(\theta_{curr})$. We do not know for how long L decreases in that direction, so θ_{next} should be taken close to θ_{curr}, at a α distance, with α small enough, hence the gradient descent equation (3).

In this section, you will implement a basic machine learning pipeline that updates weights manually following equation 3. This pipeline includes **a)** data loading and preprocessing, **b)** definition of a neural network, **c)** implementation of the training process **d)** training of different model instances **e)** model selection and **f)** model evaluation.

Unlike section 2, it is now allowed to use PyTorch's autograd to compute $\nabla L(\theta)$.

3.1 Tasks

1. Load and preprocess the CIFAR-10 dataset. Split it into 3 datasets: *training*, *validation* and *test*. Take a subset of these datasets by keeping only 2 labels: *bird* and *plane*.
2. Write a `MyMLP` class that implements a MLP in PyTorch (so only fully connected layers) such that:
   (a) The input dimension is `3072` (= `32*32*3`) and the output dimension is `2` (for the 2 classes).
   (b) The hidden layers have respectively `512`, `128` and `32` hidden units.
   (c) All activation functions are `ReLU`. The last layer has no activation function since the cross-entropy loss already includes a softmax activation function.
3. Write a `train(n_epochs, optimizer, model, loss_fn, train_loader)` function that trains `model` for `n_epochs` epochs given an optimizer `optimizer`, a loss function `loss_fn` and a dataloader `train_loader`.
4. Write a similar function `train_manual_update` that has no `optimizer` parameter, but a learning rate `lr` parameter instead and that manually updates each trainable parameter of `model` using equation (3). Do not forget to zero out all gradients after each iteration.
5. Train 2 instances of `MyMLP`, one using `train` and the other using `train_manual_update` (use the same parameter values for both models). Compare their respective training losses. To get exactly the same results with both functions, see section 3.3.
6. Modify `train_manual_update` by adding a L2 regularization term in your manual parameter update. Add an additional `weight_decay` parameter to `train_manual_update`. Compare again `train` and `train_manual_update` results with $0 <$ `weight_decay` < 1.
7. Modify `train_manual_update` by adding a momentum term in your parameter update. Add an additional `momentum` parameter to `train_manual_update`. Check again the correctness of the new update rule by comparing it to `train` function (with $0 <$ `momentum` < 1).
8. Train different instances (at least 4) of the `MyMLP` model with different learning rate, momentum and weight decay values . You can choose the same values as in the `gradient_descent_output.txt` file.

9. Select the best model among those trained in the previous question based on their accuracy.
10. Evaluate the best model.

3.2 Hints

- Wrap your computations inside a "`with torch.no_grad():`" context.
- Remember that trainable parameters can be accessed using "`for p in model.parameters()`" or "`for name, p in model.named_parameters()`".
- Remember that parameter values can then be accessed using "`p.data`" and their gradients using "`p.grad`".
- Gradient descent rules with L2-regularization and momentum can be found in the documentation of `torch.optim.SGD`.

3.3 Getting the same results with `train` and `train_manual_update`

To get exactly the same results with `train` and `train_manual_update`, do the following:

- Write `torch.manual_seed(123)` (or any other seed) at the beginning of your notebook.
- Write `torch.set_default_dtype(torch.double)` at the beginning of your notebook to alleviate precision errors.
- Change `imgs.to(device=device)` to `imgs.to(device=device, dtype=torch.double)` in your training functions and when computing accuracies in order to convert your images to the right datatype.
- Set `shuffle` to `False` when creating dataloaders.
- Add a "`torch.manual_seed(seed)`" line with a fixed `seed` value right above each of "`model = MyMLP()`" lines in order to get exactly the same weight initialization for all your models.

4 Report

The report should consist of two parts (in the same pdf):

1. An explanation of your approach and design choices to help us understand how your particular implementation works.
2. Answers to the following questions:
   (a) Which PyTorch method(s) correspond to the tasks described in section 2?
   (b) Cite a method used to check whether the computed gradient of a function seems correct. Briefly explain how you would use this method to check your computed gradients in section 2.
   (c) Which PyTorch method(s) correspond to the tasks described in section 3, question 4.?
   (d) Briefly explain the purpose of adding momentum to the gradient descent algorithm.
   (e) Briefly explain the purpose of adding regularization to the gradient descent algorithm.
   (f) Report the different parameters used in section 3, question 8., the selected parameters in question 9. as well as the evaluation of your selected model.
   (g) Comment your results. In case you do not get expected results, try to give potential reasons that would explain why your code does not work and/or your results differ.