INF265
Project 1:
Backpropagation and Gradient Descent

Deadline: February 20th, 23.59
Deliver here:
`https://mitt.uib.no/courses/57496/assignments/112644`

Projects are a compulsory part of the course. This project contributes a total of 10% of the final grade. **Projects have to be done in pairs. If you have good reasons not to do it in pairs, contact Pekka by email before February 6th.** 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).

Section 2 is independent of sections 3 and 4, and their respective tasks as well.

Code of conduct: Use of AI is allowed. However, the use must be documented; the guidelines from the faculty can be found here: `https://www.uib.no/en/nt/180737/examples-how-you-can-describe-use-ai--faculty-science-and-technology`. Remember that the goal of this project is to help you to learn. Thus, use AI responsibly. If you use AI to minimise your own effort, then you are likely to learn very little.

You should understand all code that you (or your group) submits. Students may be invited to explain their work orally and failure to demonstrate understanding satisfactorily may lead to point deduction.

Discussions on parts of the project with other pairs/students are allowed. If you do so, indicate with whom and on which parts of the project you have collaborated. Sections 2.3, 3.2 and 4.2 provide some hints, and 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 4 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.
- `ml_pipeline.ipynb` addressing section 4. Cells should already be run and output visible.
- A PDF report, addressing section 5. 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 1 point. In addition, there is a 1-point deduction for every starting 12-hour period. That is, a project submitted at 00.01 on February 21st will get a 2-point deduction and a project submitted at 12.01 on the same day will get a 3-point deduction (and so on). Submissions after February 22nd, 23:59, will not be accepted. (Executive summary: Submit your project on time.) There will be no possibility to re-take 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 of 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 $\theta = \begin{bmatrix} w_1 \cdots w_q \end{bmatrix}^T$, then $\nabla L(\theta) = \begin{bmatrix} \frac{\partial L}{\partial w_1}(\theta) \cdots \frac{\partial L}{\partial w_q}(\theta) \end{bmatrix}^T$

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 Ng's notation (which can be found in the fourth tutorial as well):

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

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 then simply have:

$$\frac{\partial L}{\partial b_j^{[l]}} = \delta_j^{[l]} \times 1 \tag{7}$$

2.1 Tasks

Using equations (4), (5), (6) and (7), 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 will be favored.

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 training process that updates the model weights manually following equation 3. This process should include **a)** data loading, data analysis and preprocessing, **b)** definition of a neural network and **c)** implementation of a training loop.

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

3.1 Tasks

1. Load, analyse 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 a `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` hyperparameter instead and that manually updates each trainable parameter of the `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` hyperparameter 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` hyperparameter to `train_manual_update`. Check again the correctness of the new update rule by comparing it to `train` function (with $0 <$ `momentum` < 1).

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(265)` (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 Machine Learning Pipeline

In this section, you will implement a complete machine learning pipeline, experiment with extending the model architecture and evaluate and analyze the results of your training process. This pipeline should include **a)** data loading, data analysis and preprocessing, **b)** definition of a neural network, **c)** implementation of the training process, **d)** experimentation with different model architectures, **e)** hyperparameter tuning and model selection, **f)** model evaluation.

Here you are allowed to train the model by using the Pytorch SGD optimizer to perform the gradient decent and update the model weights.

4.1 Tasks

1. Load and analyse the CIFAR-2 subset created earlier. As before it should be split it into 3 datasets: *training*, *validation* and *test*, only have 2 labels: *bird* and *plane* and the data should be preprocessed.

2. Extend the `train` function from before such that it also takes in validation data and calculates the validation loss along with the training loss for each epoch.

3. Using the MyMLP model class and the train function, train at least four different versions of the MyMLP model with different architectures to analyze how architectural choices affect model behavior.You should experiment with:

   (a) Increasing the size of a hidden layer

   (b) Increasing model depth by adding one or more hidden layers

   (c) Adding a dropout layer for regularization

   For each model, perform model selection through hyperparameter tuning. Try selecting different values for hyperparameters such as the learning rate, momentum, weight decay, dropout rate, and number of training epochs. Be careful not to explore too many hyperparameter combinations, as this can lead to excessive training times during the model selection process.

4. Analyze the behavior of the different model architectures by providing figures that show the training and validation loss for each of the model variations.

5. Select the best model among those trained in the previous step based on their validation accuracy.

6. Evaluate the best model and analyze its performance.

4.2 Hints

- Remember to have the model in training mode while updating the weights and setting it into evaluation mode when calculating the validation loss by using "`model.train()`" and "`model.eval()`" Wrap your validation computations inside a "`with torch.no_grad():`" context.
- When performing model selection it can be good to observe both the training and validation loss curves to see if the model has started to overfit. Neural networks are data hungry models that can often easily overfit, so fewer training epochs might sometimes give a better result.
- When analyzing the performance of the best model it can be useful to create a confusion matrix to see how well the model classifies the individual classes and to observe if one class gets more misclassified than the other.

5 Report

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

1. An explanation of your approach and design choices to help us understand how your particular implementation works for both the backpropagation and gradient decent algorithms.
2. For the machine learning pipeline, report:

   (a) The different model architectures selected for experimentation.

   (b) The different hyperparameters used for the model selection process.

   (c) Analysis of the results of the model selection process for the different architectures.

   (d) An evaluation of your selected best model.

3. Discussion on 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.

For a clear overview of what your report should include, please consult the project checklist.