IFT 6390B Fundamentals of Machine Learning
Ioannis Mitliagkas

Homework 3 - Practical Part

- This homework must be done and submitted to Gradescope in teams of 2. You are welcome to discuss with students outside of your group but the solution submitted by a group must be its own. Note that we will use Gradescope's plagiarism detection feature. All suspected cases of plagiarism will be recorded and shared with university officials for further handling.

- The practical part should be coded in python (the only external libraries you can use are numpy, matplotlib and PyTorch) and all code will be submitted as a python file to Gradescope. To enable automated code grading you should work off of the template file given in this homework folder. Do not modify the name of the file or any of the function signatures of the template file or the code grading will not work for you. You may, of course, add new functions and any regular python imports.

- Any graphing, charts, derivations, or other explanations should be submitted to Gradescope as a practical report and **separately from the homework's theoretical part**. You are, of course, encouraged to draw inspiration from what was done in lab sessions.

1 Training Neural Networks with modern autodiff [25 points]

This part consists in the implementation of a neural network trainer for univariate regression. It will be based on PyTorch and will include a diverse set of functionalities. You will need to implement most of the required functions inside the `Trainer` class provided in the solution template.

You will explore various Multi-Layer Perceptrons (MLPs) and Convolutional Neural Networks (CNNs). You will train your neural network with minibatch stochastic gradient descent, using the mean-squared-error loss and the Adam optimizer.

The `Trainer` class is initialized with the following parameters:

- `network_type`: A string from the set {'mlp', 'cnn'} denoting the type of network.
- `net_config`: a data structure defining the architecture of the neural network. It should be a `NetworkConfiguration` named tuple, as defined in the solution file. The different fields of the tuple are tuples of integers, specifying the hyperparameters for the different hidden layers of the network, namely: `n_channels`, `kernel_sizes`, `strides`, and `dense_hiddens`.
  - If `network_type='cnn'`, a CNN should be created. The first 3 fields specify the topology of the convolutional layers, while `dense_hiddens` specifies the number of neurons for the hidden fully-connected layers for the last part of the network. For instance, calling `NetworkConfiguration(n_channels=(16,32), kernel_sizes=(4,5),`

`strides=(1,2), dense_hiddens=(256, 256))` will instantiate a network configuration object which corresponds to a CNN with two convolutional layers: the first layer will have 16 channels, a 4×4 kernel, and a 1×1 stride; the second layer will have 32 channels, a 5×5 kernel, and a 2×2 stride. The last attribute implies two fully-connected hidden layers with 256 neurons each in the final part of the network.

  - If `network_type='mlp'`, only the attribute `dense_hiddens` is used, and a network with an input layer, two hidden layers with 256 neurons and an output layer should be built.

- `lr`: the learning rate used to train the neural network with the Adam optimizer.
- `batch_size`: the batch size for minibatch Adam.
- `activation_name`: a string describing the activation function. It can be `"relu"`, `"sigmoid"`, or `"tanh"`. We remind you of three activation functions:
  - $\text{RELU}(x) = \max(0, x)$
  - $\sigma(x) = \frac{1}{1+e^{-x}}$
  - $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

The `__init__` function is provided for you. In addition to loading the dataset and storing the parameters in class variables, this function initializes a dictionary of training logs, that contains information about the losses and metrics on the training and test sets during training, automatically populated during training.

1. [1 points] This question asks you to complete the `Trainer.create_activation_function` method, which should accept a string from the set {`"relu"`, `"tanh"`, `"sigmoid"`} as an argument and return a `torch.nn.Module` object implementing that specific activation function.

2. [5 points] This question is about writing a constructor function for an MLP. You will have to complete the `Trainer.create_mlp` function, which has as arguments the input dimension for the first layer, a `NetworkConfiguration` object defining the structure of the network, and a callable activation function in the form of a `torch.nn.Module` object. The provided activation function should be applied after each layer, apart from the last layer, which should have no activation. The function should return a `torch.nn.Module` object with the desired architecture; please note that the network should expect a non-flattened image tensor as input, and flatten it as part of its internal processing. The model should have one output unit, which is the prediction for regression. **We will expect the fully-connected layers to be instances of `torch.nn.Linear`. We also suggest you to use a `torch.nn.Sequential` container. As an example, an `hidden_sizes` of length 3 implies an MLP of 4 `Linear` layers in total; 3 with the prescripted activation function, and one for the output with no activation.**

3. [6 points] Similarly to the previous one, this question is about writing a constructor function for a CNN, which will be used to build a network in case it will be the selected architecture for training. You will have to complete the `Trainer.create_cnn` function, which has as arguments the number of channels of the input image, a `NetworkConfiguration` object, and an activation function in the form of a `torch.nn.Module`. The activation function should be applied after each convolutional and fully-connected layer, apart from the last one,

which should have no activation. After the activation function of each convolutional layer, a `torch.nn.MaxPool2d(kernel_size=2)` layer should be applied; the last convolutional layer should have no max pooling and being instead followed by a `torch.nn.AdaptiveMaxPool2d((4, 4))` and a `torch.nn.Flatten()`, right before the fully-connected layers, to ensure that the network is agnostic (up to a certain degree) to the input size. The function should return a `torch.nn.Module` object with the desired architecture. The model should have one output unit, which is the prediction for regression. **Again, we will expect both convolutional and fully-connected layers to be instances of `torch.nn.Conv2d` and `torch.nn.Linear`. We also suggest you to use a `torch.nn.Sequential` container.**

4. [4 points] Complete the `Trainer.compute_loss_and_mae` function that takes as input a tensor of samples and a tensor of labels. The function should return the Mean Squared Error loss for the current network, as well as the Mean Absolute Error on that batch. Beware that the returned loss and Mean Absolute Error value should be a `torch.Tensor` keeping track of the computations that led to it, in order for automatic differentiation to be able to compute gradients with the backpropagation algorithm.

5. [6 points] For this question, you should use the previous functions you implemented.

   Complete the `Trainer.training_step` function. This function implements a single training step, taking a batch of inputs and labels as inputs. It will be then used inside the `train_loop` function to have a complete training procedure. Your function should compute the gradient of the loss function of your current network and update it using the optimizer. Your function must also return the loss and the MAE for the batch taken as input

6. [3 points] Complete the `Trainer.evaluate` function. Your function should return the average loss and Mean Absolute Error on a given set of data. You can use the functions that you have already implemented.

2 Experimenting on the CIFAR-10 [20 points]

In this part of the homework, you are going to do some experimentation, also leveraging your trainer implementation, on the well-known CIFAR-10 dataset. The dataset can be loaded using the provided `Trainer.load_dataset` method.

The CIFAR-10 dataset consists of 60 000 32x32 colour images in 10 classes, with 6000 images per class. There are 50 000 training images and 10 000 test images. Each image is represented by a $3 \times 32 \times 32$ tensor, where the first dimension represents the three RGB channels composing the image. The variables `self.train, self.test` initialized in the constructor of the `Trainer` are PyTorch `DataLoader` type objects. These objects are iterables which yield a tuple `(images, labels)` which contain a batch of images with its corresponding labels.

The answers to the questions of this section, along with the required figures, should be written in a report you will submit to Gradescope as the practical part of the homework (seperately from the theoretical part).

1. [10 points] Train an MLP with 2 hidden layers of size 128 on the CIFAR-10 dataset, for 50 epochs, and a batch size of 128. Use the ReLU activation function. For reproducibility purposes, **please do not change the seed which is already set in the solution file**.

When not specified, leave the default value for the arguments of the `Trainer`. Include in your report the figure and answer for the following question:

- Generate a figure with the test Mean Absolute Error w.r.t. increasing epochs for values of learning rates in the set $\{0.01, 1 \times 10^{-4}, 1 \times 10^{-8}\}$. What are the effects of learning rates that are very small or very large?

2. [5 points] Train a CNN on the CIFAR-10 dataset for 50 epochs with a batch size of 128. Use the ReLU activation function. Use three hidden convolutional layers with a number of filters of $(16, 32, 45)$, kernels of size 3, and a stride of 1. Use one final fully-connected hidden layers with 128 neurons. For reproducibility purposes, **please do not change the seed which is already set in the solution file**.

   When not specified, leave the default value for the arguments of the `Trainer`. Include in your report the figure and answer for the following question:

   - Generate a figure with the test Mean Absolute Error w.r.t. increasing epochs for the above CNN.

3. [5 points] We will now experiment with the meaning of "depth" and "width" in convolutional neural networks. As seen previously, sparse interactions are one of the features of convolution applied to neural networks: this means that, thanks to the locality of kernels, only parts of the sample features interact with each other. By constrast, in MLPs, matrix multiplication allows a dense interaction between the inputs and the parameters. We are now going to break this property in two antipodic ways.

   (a) Imagine a **deep CNN** with about the same number of parameters as the original one, 4 layers with a number of filters of $(8, 16, 32, 64)$, unitary strides and zero padding. This time, a `kernel_size` of 1 in all the convolutional layers, and a final MLP as one-hidden-layer MLP of 128 hidden neurons. What is the relationship between the type of convolutional layers used by this network and fully-connected layers?

   (b) Imagine a **shallow CNN** with only one convolutional layer, with a `kernel_size` equal to the size of the input image (32×32) and a number of channels in this layer such that the number of parameters is approximately the same as the one of the previous networks. Consider the other hyperparameters the same as the ones in the previous CNN (unitary stride, zero padding, one 128-neurons final hidden layers). What is the relationship between the type of convolutional layers used by this network and fully-connected layers?

   (c) Now train the **deep CNN** and **shallow CNN** as defined above for 50 epochs with a batch size of 128 and learning rate of 0.001. Include in your report a figure comparing the validation and training MAE of these three types of CNNs (including the original CNN of the previous question). How does their general performance and generalization capabilities compare? And what is your hypothesis about their different behaviors?