IFT 6390 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 [35 points]

This part consists in the implementation of a neural network trainer for multi-class classfication. 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. In the next section, you will put this trainer at work.

You will explore various Multi-Layer Perceptrons (MLPs) and Convolutional Neural Networks (CNNs) but the last layer will be always equipped with a **softmax** non-linearity,.

You will train your neural network with minibatch stochastic gradient descent, using the cross entropy 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`, `padding` and `dense_hiddens`.

  - If `network_type='cnn'`, a CNN should be created. The first 4 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), paddings=(1,0), 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, a 1×1 stride and a 1×1 padding; the second layer will have 32 channels, a 5×5 kernel, a 2×2 stride and a 0×0 padding. 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.

- `n_classes`: the number of classes in the classification problem (also the number of neurons of the output layer).
- `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:
  - $\text{RELU}(x) = \max(0, x)$
  - $\sigma(x) = \frac{1}{1+e^{-x}}$
  - $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- `normalization`: a boolean set to true if the data should be normalized.

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 accuracies on the training and validation sets during training, automatically populated during training.

1. [3 points] Automatic differentiation frameworks such as Pytorch are useful for training neural networks, but the tools they provide are more general than that. They can compute the gradient of (almost) any composable function. To showcase this, complete the `gradient_norm` function, which takes a function (we assume is made out of pytorch primitives) and a list of tensors as inputs, and returns the Euclidean norm of the gradient of the output of the function w.r.t. the list of inputs. **Note that the function will take the unpacked tensor list as an input, as in `function(*tensor_list)`.**
2. [2 points] To touch a little bit more the capabilities of automatic differentiation, complete now the `jacobian_norm` function which, similarly to the previous one, takes a function as an argument, as well as a (single) tensor, and should output the Frobenius norm of the Jacobian matrix of that function evaluated at the input tensor provided as a second argument.
3. [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.

4. [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, the number of classes, 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 a softmax 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. **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 in output with softmax activation.**

5. [5 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, the number of classes 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 a softmax 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. **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.**

6. [2 points] To answer this question, you have to complete the `Trainer.normalize` method. This function should take the training, validation and test sets as arguments and return their normalized version. The normalization should happen by subtracting from the X matrices the feature-wise mean of the training samples and diving them by the feature-wise standard deviation of the training samples.

7. [2 points] Write the `Trainer.one_hot` function. This function should transform a PyTorch tensor `y` of size $batch\_size$ containing values from 0 to $n\_classes - 1$ into a matrix of size $batch\_size \times n\_classes$ containing the one-hot encodings of the true labels `y`.

   For example, if the number of classes is 5, then `one_hot(torch.tensor([1, 0, 3, 4]))` should return the following tensor:

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

8. [4 points] Complete the `Trainer.compute_loss_and_accuracy` function that takes as input a matrix of samples and a matrix of one-hot labels. The function should return the cross-entropy loss for the current network, as well as the accuracy on that batch. Beware that

the returned loss 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.

To avoid numerical errors, you should pre-process the probabilities in output from the network by clipping very small values (i.e. close to 0) to the fixed value `Trainer.epsilon` and very large values (i.e. close to 1) to $1-$ `Trainer.epsilon`.

9. [2 points] In this question, you are asked to complete the `Trainer.compute_gradient_norm` function, which should take a `torch.nn.Module` as an input and output the Euclidean norm of the flattened gradient vector of its parameters. The gradient can be found in specific buffers that are filled during backpropagation. This is a useful quantity when debugging the training of neural networks.

10. [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 into 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. In addition, the method should return the norm of the gradient of the loss function with respect to the network's parameter evaluated at that particular batch.

11. [3 points] Complete the `Trainer.evaluate` function. Your function should return the average loss and accuracy on a given set of data. You can use the functions that you have already implemented.

2 Experimenting on the FashionMNIST dataset [20 points]

In this part of the homework, you are going to do some experimentation, also leveraging your trainer implementation, on the FashionMNIST dataset.

The dataset will be automatically downloaded using the `Trainer.load_dataset` method that we already wrote, using the `torchvision` module. It will be downloaded into the directory specified through the `datapath` argument (`./data` by default).

The training set of the FashionMNIST dataset consists of 60000 28×28 black and white images from 10 classes of garments (e.g., T-shirts, trousers). We are going to use a test set of 8000 images and validation set of 2000 images. Each image is represented by a $1 \times 28 \times 28$ tensor, where the first dimension represents the single channel composing the image. The variables `self.train, self.valid, self.test` initialized in the constructor of the `Trainer` will contain the dataset. Each of these three variables is a tuple. For example, `self.valid[0]` is a 4-dimensional PyTorch tensor of size $2000 \times 1 \times 28 \times 28$, and `self.valid[1]` is a 1-dimensional PyTorch tensor of size 2000 containing the labels of the 2000 validation set images, from 0 to 9. You can double check whether the tensors have the right sizes.

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 Theoretic part).

1. [5 points] As a warmup before doing any training, this question will let you explore some properties of CNNs. Convolution in neural networks induces powerful inductive biases via sparse interactions, weight sharing and equivariant representations.

   We will now test the third feature. Equivariance for a function $f : X \to Y$ to a transformation g means that $f(g(x)) = g(f(x))$, i.e., applying that transformation to the input of the function is equivalent to applying it to the output. Convolution is equivariant to some types of transformations only.

   You can find an incomplete `test_equivariance` method in the `Trainer` object. It already implements two functions which shift and rotate an image respectively, as well as a very small convolution-only neural network. Use this function to help you generate the following figures, to be included in your report.

   - A figure showing the original image, which is the first image `self.train[0][0]` of the training set;
   - A figure showing the output features of the small CNN `model` when given that first image as an input;
   - A figure showing the image of the pixel-by-pixel absolute difference between (1) the shifted output features given the unshifted input, and (2) the unshifted output of the CNN given the shifted input. Shifts should be performed by using the provided `shift` transformation;
   - A similar figure, but this time with rotation. In other words, show the pixel-by-pixel absolute difference between the rotated output of the CNN given the original image and the unprocessed output of the CNN given the rotated image. Use the provided `rotation` transformation, without any further processing.

   Write which are your observations on the equivariance properties of convolutional layers;

   (a) To which types of transformations are convolutional layers equivariant?

   (b) Why should equivariance to some transformations be more useful than equivariance to other transormations for learning? Does it depend only on the training algorithm or on the data distribution as well?

2. [3 points] You are now asked to train an MLP with 2 hidden layers, of size 256 and 256 respectively on the FashionMNIST 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 following figures, and answer related questions:

   - Set `normalization=False` and generate a figure with the validation accuracy w.r.t. increasing epochs for values of learning rates in the set $\{0.01, 1 \times 10^{-4}, 1 \times 10^{-8}\}$;
   - Now set `normalization=True` and generate the figure for the same learning rates.
     (a) What is the effect of normalization?
     (b) Does it help convergence? Why / why not?
     (c) What are the effects of learning rates that are very small or very large?

   For verification purposes, the best combination of learning rate and normalization should be able to achieve accuracies above 80%.

3. [5 points] In the next three questions, you will be asked asked to experiment with depth in MLPs and CNNs and to comprare their behavior. When asked to create networks with approximately the same number of parameters, you should ensure that the number of parameters differs at most of 10000.

   (a) How many learnable parameters (scalars) does the previous neural network have?

   (b) Now train a neural network with approximately the same number of parameters as the one that you trained, but hidden layers with 64 neurons only. Hint: they are going to be many layers!

      i. Plot train and validation accuracies in the same plot as the previous MLP trained with default hyperparameters. Put this figure in your report;

      ii. Now plot the evolution of the gradient norm (which should be already logged) during training for the two architectures and put this figure in your report;

      iii. Does the behavior of the gradient norm explain the performance?

4. [2 points] Now train a CNN with approximately the same number of parameters, with the following constraints. Use three hidden convolutional layers with a number of filters of $(16, 32, 45)$, a stride of 1 and a padding of 0 at each layer. Use two final fully-connected hidden layers with 256 neurons each. Find the appropriate kernel size to maintain approximately the same number of parameters.

   (a) Put in the report a figure showing the performance of this CNN and the original (non-deep) MLP;

   (b) Which one does perform better? What can you say about generalization capabilities of the two architectures?

5. [5 points] Let us 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 still a final MLP as two-hidden-layer MLP of 256 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 (28×28) 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, two 256-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 256 and put in your report a figure comparing the validation and training accuracies 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?