lions@epfl

EE-556: MATHEMATICS OF DATA: FROM THEORY TO COMPUTATION
LABORATORY FOR INFORMATION AND INFERENCE SYSTEMS
FALL 2018

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

INSTRUCTOR
PROF. VOLKAN CEVHER

HEAD TA
YA-PING HSIEH

HOMEWORK EXERCISE-2 (FOR LECTURE 8)

This homework covers Lecture 8. We will study *autoencoders* for representation learning as well as the hand-written digit classification task. In order to train an autoencoder you will have to implement the backpropagation algorithm from scratch. You will also implement two versions of stochastic gradient descent, Adam, Adagrad, and RMSProp, and you will use them to train a neural network to solve the image classification task. In this second part you are allowed to leverage the automatic differentiation capabilities (i.e. backpropagation) from one of the most popular deep learning frameworks "PyTorch".

1 Autoencoder for Representation Learning

Autoencoder is a kind of neural network used to learn efficient data codings in an unsupervised manner, typically for the purpose of dimension reduction. As Figure 1 shows, it consists of an encoding network ϕ, which maps the input to the code, and a decoding network φ, which maps the code to the output. Given data $\{\mathbf{x}^{(i)}\}_{i=1,2,\dots,N}$, the loss of the autoencoder is the mean squared error between the input and output data:

$$L = \frac{1}{N}\sum_{i=1}^{N} L^{(i)} = \frac{1}{2N}\sum_{i=1}^{N} \|\mathbf{x}^{(i)} - \varphi(\phi(\mathbf{x}^{(i)}))\|_2^2. \tag{1}$$

We study a very simple case here: both the encoder ϕ and decoder φ are one-layer networks without bias terms, parameterized by $\mathbf{W}_1$ and $\mathbf{W}_2$ respectively:

$$\mathbf{z}^{(i)} = \phi(\mathbf{x}^{(i)}) = \sigma(\mathbf{W}_1\mathbf{x}^{(i)}), \tag{2}$$

$$\mathbf{x}'^{(i)} = \varphi(\mathbf{z}^{(i)}) = \mathbf{W}_2\mathbf{z}^{(i)}, \tag{3}$$

where σ is the elementwise activation function such as sigmoid and ReLU. We assume $\mathbf{x}^{(i)} \in \mathbb{R}^d$ and $\mathbf{z}^{(i)} \in \mathbb{R}^k$, so $\mathbf{W}_1 \in \mathbb{R}^{k\times d}$ and $\mathbf{W}_2 \in \mathbb{R}^{d\times k}$.

1. (15 points) Write down the loss function $L^{(i)}$ in terms of input $\mathbf{x}^{(i)}$ and parameters $\mathbf{W}_1$, $\mathbf{W}_2$. Calculate the gradient of $L^{(i)}$ with respect to $\mathbf{W}_1$ and $\mathbf{W}_2$.

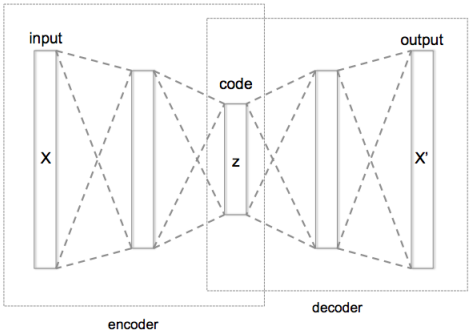

Figure 1: Structure of autoencoder

2. (15 points) Implement the training of autoencoder on MNIST in PyTorch using minibatch stochastic gradient descent. Write the update rule using standard mathematical operators (in *util.py* file). Attach the learning curve in the report (run *train.py* script with the hyper-parameters you chose). Any auto gradient calculation in PyTorch is strictly FORBIDDEN here.

   Hint: In minibatch training, the input is a batch of data, so the batch size should be taken in the account when calculating the gradient.

3. (10 points) Change the dimension of code k as well as the activation function. Plot the corresponding reconstructed images $\varphi(\phi(\mathbf{x}^{(i)}))$ and report your discoveries (run *reconstruct.py* script with hyper-parameters you choose).

   Principle component analysis (PCA) is another popular method used for dimension reduction. Assume data $\{\mathbf{x}^{(i)}\}_{i=1,2,...,N}$ are normalized, i.e., the expectation of each feature is 0. The procedure of PCA is as follows:

   A Calculate covariance matrix: $\Sigma = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{x}^{(i)})(\mathbf{x}^{(i)})^T$.

   B Eigenvector decomposition: $\Sigma = \mathbf{U}\mathbf{D}\mathbf{U}^T$ where $\mathbf{D}$ is a diagonal matrix and $\mathbf{U}$ is an orthogonal matrix.

   C Pick top k eigenvectors from $\mathbf{U}$ to construct a projection matrix: $\mathbf{P} = (\mathbf{u}^{(1)}, \mathbf{u}^{(2)}, ..., \mathbf{u}^{(k)})$.

   D Calculate low-dimension projection: $\mathbf{z}^{(i)} = \mathbf{P}^T\mathbf{x}^{(i)}$.

4. (10 points) Show how you can set the activation function σ such that the resulting autoencoder resembles PCA. Is there any additional constraints on weight $\mathbf{W}_1$ and $\mathbf{W}_2$?

Guides of the code

- Any builtin optimizers in PyTorch and functions such as *backward* are NOT allowed in this execrise. The derivative of activation function is provided as *dactivation* function.
- In PyTorch, the graph is dynamic, and the update rule is in the *train* function with real *data batch*. You can simply treat PyTorch tensor just like ones in Numpy here.
- In *train.py*, the hyper-parameters you can modify include the number of training epochs, hidden dimension k, activation function σ, model directory to save, and the learning rate.
- In *reconstruct.py*, the hyper-parameters you can modify include the hidden dimension k, activation function σ, model directory to restore, and the number of images to display. The hidden dimension k and activation function σ should be consistent with the model restored.

2 Optimizers of Neural Network

The goal of this exercise is to implement different optimizers for a handwritten digit classifier using the well-known MNIST dataset. This dataset has 60000 training images, and 10000 test images. Each image is of size 28×28 pixels, and shows a digit from 0 to 9.

(a) General guideline: complete the codes marked with *TODO* in the *optimizers.py*.

(b) The implementation of mini-batch SGD and SGD with HB momentum are provided as examples.

Vanilla Minibatch SGD
Input: learning rate γ
1. initialize θ_0 **2. For** t = 0, 1,... ,N-1: obtain the minibatch gradient $\hat{\mathbf{g}}_t$ update $\theta_{t+1} \leftarrow \theta_t - \gamma\hat{\mathbf{g}}_t$

Minibatch SGD with Momentum
Input: learning rate γ, momentum ρ
1. initialize θ_0, $\mathbf{m_0} \leftarrow \mathbf{0}$ **2. For** t = 0, 1,... ,N-1: obtain the minibatch gradient $\hat{\mathbf{g}}_t$ update $\mathbf{m}_{t+1} \leftarrow \rho\mathbf{m}_t + \hat{\mathbf{g}}_t$ update $\theta_{t+1} \leftarrow \theta_t - \gamma\mathbf{m}_{t+1}$

(c) Implement of following optimizers:

(a) (10 points) Implement AdaGrad method

AdaGrad
Input: global learning rate γ, damping coefficient δ
1. initialize θ_0, $\mathbf{r} \leftarrow \mathbf{0}$ **2. For** t = 0, 1,... ,N-1: obtain the minibatch gradient $\hat{\mathbf{g}}_t$ update $\mathbf{r} \leftarrow \mathbf{r} + \hat{\mathbf{g}}_t \odot \hat{\mathbf{g}}_t$ update $\theta_{t+1} \leftarrow \theta_t - \frac{\gamma}{\delta + \sqrt{\mathbf{r}}}\hat{\mathbf{g}}_t$

where $\odot$ is the element-wise multiplication between two matrices.

(b) (10 points) Implement RMSProp

RMSProp
Input: global learning rate γ, damping coefficient δ, decaying parameter τ
1. initialize θ_0, $\mathbf{r} \leftarrow \mathbf{0}$ **2. For** t = 0, 1,... ,N-1: obtain the minibatch gradient $\hat{\mathbf{g}}_t$ update $\mathbf{r} \leftarrow \tau\mathbf{r} + (1-\tau)\hat{\mathbf{g}}_t \odot \hat{\mathbf{g}}_t$ update $\theta_{t+1} \leftarrow \theta_t - \frac{\gamma}{\delta + \sqrt{\mathbf{r}}}\hat{\mathbf{g}}_t$

(c) (10 points) Implement Adam method

Adam
Input: global learning rate γ, damping coefficient δ, first order decaying paramter β_1, second order decaying parameter β_2
1. initialize θ_0, $\mathbf{m}_1 \leftarrow \mathbf{0}$, $\mathbf{m}_2 \leftarrow \mathbf{0}$ **2. For** t = 0, 1,... ,N-1: obtain the minibatch gradient $\hat{\mathbf{g}}_t$ update $\mathbf{m}_1 \leftarrow \beta_1\mathbf{m}_1 + (1-\beta_1)\hat{\mathbf{g}}_t$ update $\mathbf{m}_2 \leftarrow \beta_2\mathbf{m}_2 + (1-\beta_2)\hat{\mathbf{g}}_t \odot \hat{\mathbf{g}}_t$ correct bias $\hat{\mathbf{m}}_1 \leftarrow \frac{\mathbf{m}_1}{1-\beta_1^{t+1}}$ $\hat{\mathbf{m}}_2 \leftarrow \frac{\mathbf{m}_2}{1-\beta_2^{t+1}}$ update $\theta_{t+1} \leftarrow \theta_t - \gamma\frac{\hat{\mathbf{m}}_1}{\delta + \sqrt{\hat{\mathbf{m}}_2}}$

(d) (20 points) Set the learning rate to 0.5, 10^{-2}, and 10^{-5}. Run the neural network for 15 epochs, repeat it for 3 times and compare the obtained average accuracies. Plot the obtained training loss for different optimizers. State how the performance of the adaptive learning-rate methods versus SGD and SGD with momentum methods is compared.

3 Guidelines for the preparation and the submission of the homework

Work on your own. Do not copy or distribute your code to other students in the class. Do not reuse any other code related to this homework. Here are few warnings and suggestions for you to prepare and submit your homework.

- This homework is due at 09:00AM, 19 November, 2018
- Submit your work before the due date. Late submissions are not allowed and you will get 0 point from this homework if you submit it after the deadline.
- Questions of 0 points are for self study. You do not need to answer them in the report.
- Your final report should include detailed answers and it needs to be submitted in PDF format.
- The PDF file can be a scan or a photo. Make sure that it is eligible.
- The results of your simulations should be presented in the final report with clear explanation and comparison evaluation.
- We provide Pytorch scripts that you can use to implement the algorithms, but you can implement them from scratch using any other convenient programming tool (in this case, you should also write the codes to time your algorithm and to evaluate their efficiency by plotting necessary graphs).
- Even if you use the Pytorch scripts that we provide, you are responsible for the entire code you submit. Apart from completing the missing parts in the scripts, you might need to change some written parts and parameters as well, depending on your implementation.
- The code should be well-documented and should work properly. Make sure that your code runs without errors. If the code you submit does not run, you will not be able to get any credits from the related exercises.
- Compress your code and your final report into a single ZIP file, name it as `ee556_2018_hw2_NameSurname.zip`, and submit it through the moodle page of the course.