TP3 - Advanced Statistical Learning
NN and CNN with pytorch

ENSAE 2021/2022

Mastère Spécialisé - Data Science

Reda TAWFIKI - Jeffrey VERDIERE

Contents

1 Introduction

1.1 Question 1

Let's solve analytically the problem $\min_{w\in\mathbb{R}^5} L = (1 - x^\top w)^2$ with $w = (a_0, a_1, a_2, a_3, a_4)^T$ and $x = (1, 1, 1, 1, 1)^T$ We can rewrite our minimization problem as follow with Lagrage's computation optimization:

$$\min_{w\in\mathbb{R}^5} L = [1 - (a_0 + a_1 + a_2 + a_3 + a_4)]^2$$

Thus we need to compute the gradient and equalize it to 0 (no need to check if the hessian matrix is positive-definite since the squared loss function is obviously convex):

Therefore we can compute the gradient and equalize to to 0 and we will obtain the global minimum of the surface since it is a squared loss is convex. Moreover, this problem is the same the

$$\nabla L = \begin{pmatrix} \frac{\partial L}{\partial a_0} \\ \vdots \\ \frac{\partial L}{\partial a_4} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \Leftrightarrow 2 \times (-1) \times [1 - (a_0 + a_1 + a_2 + a_3 + a_4)] = 0 \Leftrightarrow a_0 + a_1 + a_2 + a_3 + a_4 = 1$$

As it said in the jupyter notebook, it is a simple least square problem on single observation (x, y) that we can compute easily with the normal equation $\theta = (X^\top * X)^- 1X^\top * Y)$.
So $a_0 = a_1 = a_2 = a_3 = a_4 = \frac{1}{5}$ is a solution of the equation above which is equal to the result of the Gradient Descent algorithm.

1.2 Question 2

The learning rate is a tuning parameter in an optimization algorithm that determines the step size at each iteration while moving toward a minimum of a loss function. Since it influences to what extent newly acquired information overrides old information, it kind of represents the speed at which a machine learning model "learns". The learning rate `lr` is the rate at which the gradient is explored. The greater the learning rate is the faster the model will learn. But one should be pay attention as a value too great might lead the minimum value to be missed and the optimal solution never found. On the other hand, the smaller the learning rate is, the more likely the model is to find a global minimum. However, that makes the algorithm considerably slower. The range of values to consider for the learning rate is smaller than 1 and greater than 10^{-6}. A traditional default value for the learning rate is 0.1 or 0.01, and here we choose 0.01 as this value ensures a fast convergence. In the case of the batch gradient descent or the stochastic gradient descent, θ involves the learning rate with the following iteration: $\theta_j := \theta_j - \alpha(L(x_i) - y_i)x_{ij}$
We denote i the index of the raw and j the index of the column.

1.3 Question 3

The `backward()` function computes the gradient of current tensor with respect to graph leaves. Here, as we call `loss.backward()` for the variable *loss* that involved x in its calculations, then $x.grad$ will hold $\frac{\partial loss}{\partial x}$ for every tensor with the value $True$ for the parameter `requires_grad`. These then are accumulated into $w.grad$ (this is why we are using `.grad.zero()` afterwards). This way, the function performs the calculations needed for each step of the backpropagation for feed-forward neural nets.

2 Multi layer perceptron

2.1 Question

`torch.utils.data.DataLoader()` combines a dataset and a sampler, and provides an iterable over the given dataset. The parameter `shuffle` is set to `True` so we will get a new order of exploration at each pass.

Shuffling the order in which examples are fed to the classifier is helpful so that batches between epochs do not look alike. Doing so will eventually make our model more robust. Moreover, the `break` statement in the for loop terminates the loop containing it so the loop displays the first pair $(input, target)$ that there is in the DataLoader and then, it breaks. That's why it's not plotting the same number all the time.

3 Problem 1: Logistic regression via pytorch

3.1 Mathematical description of logistic regression

Given a feature vector X and a discrete response Y taking values in the set $\mathcal{C}$, classification is to build a function $C(X) \in \mathcal{C}$ that takes X as input and predicts its value for Y. We are more interested in estimating the probabilities that X belongs to each category in C i.e. $p(X \in c) = \mathbb{P}(Y = c|X)$. In the binary logistic regression, the outcome has 2 possible modalities while in multinomial logistic regression it has more. The simplest case is when Y is binary, known as a Bernoulli variable, which is the simplest non-trivial random variable.

$$Y \sim \text{Ber}(p) \Leftrightarrow Y = \begin{cases} 1, & \text{with probability } p, \\ 0, & \text{with probability } 1-p, \end{cases}$$

or, equivalently, if $\mathbb{P}[Y = y] = p^y(1-p)^{1-y}, \quad y = 0, 1$. Recall that a binomial variable with size n and probability p, $\text{Bi}(n, p)$, was obtained by adding n independent $\text{Ber}(p)$.
Assume then that Y is a Bernoulli variable and that X are predictors associated to them, the purpose in logistic regression is to estimate

$$p(x) = \mathbb{P}[Y = 1|X = x] = \mathbb{E}[Y|X = x],$$

i.e. how the probability of $Y = 1$ changes according to particular values x, of the random variables X.
The goal is to encapsulate the value of $z = \beta_0 + \beta_1 x$, in $\mathbb{R}$, and map it to $[0, 1]$. In logistic regression, we use the logistic function,

$$p(X) = \frac{e^{\beta_0+\beta_1 X}}{1 + e^{\beta_0+\beta_1 X}}$$

No matter what values β_0, β_1 or X take, $p(X)$ will have values between 0 and 1. The logistic function always produces an S-shaped curve and the logistic distribution function is

$$F(z) = logistic(z) = \frac{exp(z)}{1 + exp(z)} = \frac{1}{1 + exp(-z)}$$

The odds take values between 0 and ∞ and we have :

$$\frac{p(X)}{1-p(X)} = e^{\beta_0+\beta_1 X} \quad and \quad \log\left(\frac{p(X)}{1-p(X)}\right) = \beta_0 + \beta_1 X$$

where $\log\left(\frac{p(X)}{1-p(X)}\right)$ is called the log-odds or logit that is here linear in X.

We estimate β_0 and β_1 using the Maximum Likelihood Estimation method (MLE) which is the probability of the data based on the model. It gives the probability of the observed zeros and ones in the data. The estimates $\widehat{\beta}_0$ and $\widehat{\beta}_1$ are chosen to maximize this likelihood function :

$$l(\beta_0, \beta_1) = \prod_{i=1}^{n} p(x_i)^{Y_i}(1 - p(x_i))^{1-Y_i}.$$

We can generalize the simple logistic regression equation with $X = (X_1, \ldots, X_p)$, p predictors:

$$\log\left(\frac{p(X)}{1-p(X)}\right) = \beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p$$

Just as in the simple logistic regression we use the maximum likelihood method to estimate $\beta_0, \beta_1, \ldots, \beta_p$ and we have :

$$p(X) = \frac{e^{\beta_0+\beta_1 X_1+\ldots+\beta_p X_p}}{1 + e^{\beta_0+\beta_1 X_1+\ldots+\beta_p X_p}}$$

3.2 Mathematical description of optimization algorithm that we use

Stochastic gradient descent (SGD) performs a parameter update for each training example $x^{(i)}$ and label $y^{(i)}$:

$$\theta_k = \theta_{k-1} - \eta \nabla L(\theta_{k-1})$$

To accelerate the calculations of $\widehat{\theta}_n$ by gradient descent when n is very large, the stochastic gradient descent is used. To do this, the sample is partitioned into small batches $B_1, ..., B_M$ such as :

$$card(B_m) \simeq \left[\frac{n}{M}\right], \quad m = 1, ..., M \quad and \quad B_1 \cup B_2 \cup ... \cup B_M = \{1, ...n\}$$

At each iteration of the gradient descent, the calculation of $\nabla L_n(\theta^{(t)})$ is replaced by the average over the batch m :

$$\nabla L^{(m)}(\theta^{(t)}) = \frac{1}{|B_m|} \sum_{i \in B_m} \nabla_\theta l(Y_i, g(X_i, \theta^{(t)}))$$

The batch is changed at each iteration. When M iterations have been performed and therefore all the data has been used, it is said that a training epoch has been made. We can then choose a new batch partition and start a new epoch.

Algorithm: INPUT: $\theta^{(0)} \in \Theta$, $h > 0$, number of epochs n_e, M
ITERATIONS:
For $i = 1, ..., n_e$:
  Randomly choose $B_1, ...B_M$
  For $m = 1, ..., M$:
    $\theta = \theta - h \nabla L^{(m)}(\theta)$
  End
End

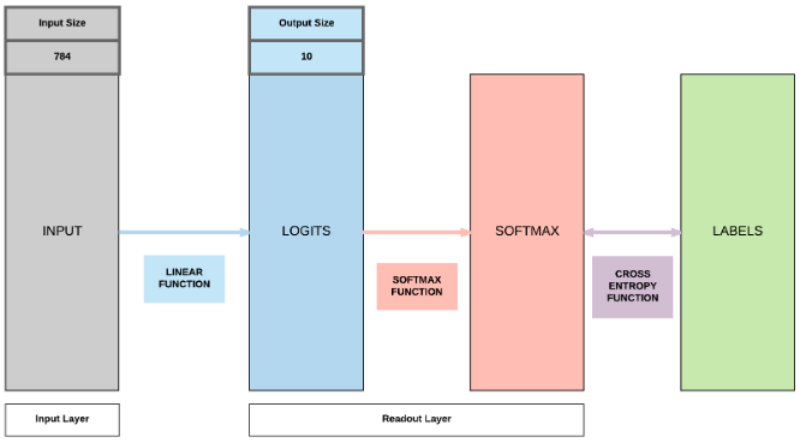
The above algorithm is the Mini-Batch version of the gradient descent. The idea behind this algorithm is that $\forall \theta$ fixed, we have :

$$\mathbb{E}[\nabla L^{(m)}(\theta)] = \nabla L_n(\theta) \qquad and \qquad \mathbb{E}[(\nabla_j L^{(m)}(\theta) - \nabla_j L_n(\theta))^2] \leq \frac{C}{|B_m|} \simeq \frac{CM}{n}$$

So $\nabla L^{(m)}$ does not necessarily give the steepest descent direction but it is often a descent direction anyway.

3.3 High level idea of how to implement logistic regression with pytorch

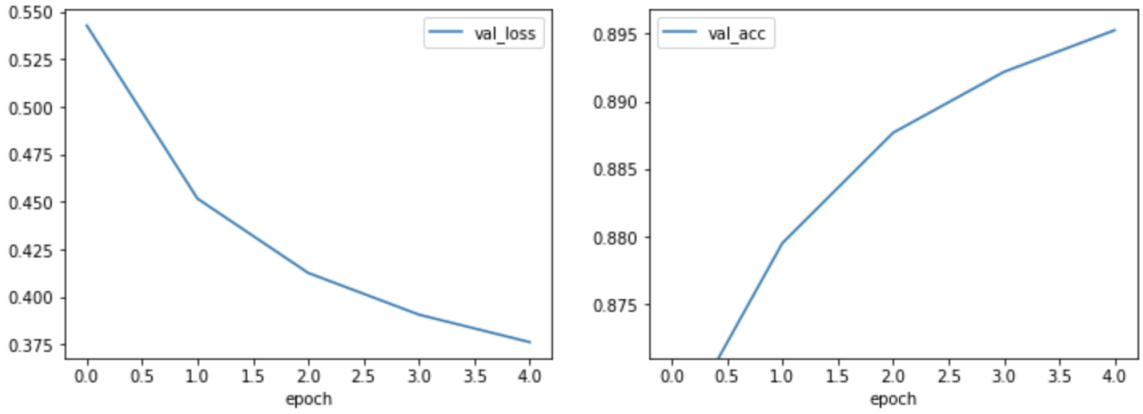
Logistic Regression calculates the logits i.e. scores in order to predict the target classes. We want probabilities between 0 and 1, so we wrap the right side of the linear regression equation into the logistic function. The calculated logits function will pass through the softmax function, also known as the normalized exponential function, that calculates the probabilities. The high probability target class will be the predicted target class.



(1) First, we're going to **load the dataset** using `torchvision.datasets`, a library which has almost all the popular datasets used in Machine Learning. We use `torchvision.transforms` module, that contains various methods to transform objects into others, to transform from images to PyTorch tensors. Then, we use the `DataLoader` class to **make our dataset iterable** in order to input our input data into the model. It will load the data into memory in batches by specifing the batch_size. The shuffle is set to True for training data to load different images each time at each epoch. This will make model more robust and avoid over/underfitting. (2) Here, we **create the Model Class** that is a class that defines the architecture of Logistic Regression. It's a subclass of `torch.nn.Module`. We initialize the model and define the forward pass. In the `forward()` we use the `.reshape` method because the shape of our input data is $1 \times 28 \times 28$ and we need to reshape or flatten them to size 784. We can apply softmax to our current linear output in order to convert our Linear Output into Softmax Probabilities. (3) We **instantiate the Model Class**. We initialized some hyper parameters and the Logistic Regression Model. Recall that MNIST is composed of images of size 28×28 , hence the dimension of the input is 784. We have 10 classes, so the dimension of the output is 10. (4) Then, we **instantiate the Loss Class**. We decided to use the cross-entropy to compute the loss during training and validation. We need Cross-entropy loss to calculate our loss before we backpropagate and update our parameters. It performs logarithmic softmax on classification model to fix the problem of slow learning rate. It turns out that we can solve this problem of slow learning rate by replacing the quadratic cost with a different cost function, known as the cross-entropy. We can consider it as loss or cost function because of its Non-Negative Nature and also the cross-entropy is positive and tends toward zero as the neuron gets better at computing the desired output for all training input. (5) Now, we **instantiate the Optimizer Class** to create an optimizer that will be the learning algorithm to use. Here, we decided to use the Stochastic Gradient Descent explained in the previous part. (6) After all of that, we **train our model** using the previous function we implemented. We are iterating to the number of epochs and call the train function that we created to makes one passe over the train data and updates weights. We also call for each epoch the validation function created to makes one passe over validation data and provides validation statistics. The training is performing the following tasks: (a) Reset all gradients to 0. (b) Make a forward pass. (c) Calculate the loss. (d) Perform backpropagation. (e) Update all weights. (5) Finally, we **test our model** on the test dataset and we display the results.

3.4 Report classification accuracy on test data.

```
Test accuracy: 0.9074 | Test loss: 0.34793874781131745
```



As we see, the classification accuracy on test data is almost 91%. By plotting the evolution of the accuracy and the loss in function of the number of epoch, we can see that the accuracy is increasing while the loss is decreasing.

4 Elements of CNN: nn.Conv2d and MaxPool2d

4.1 Problem

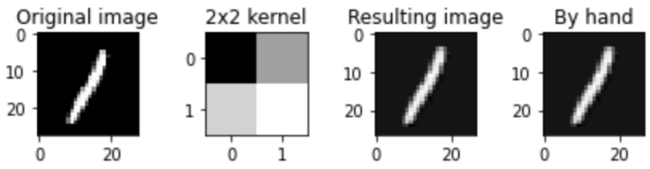
Kernel, padding and strided convolutions affect the size of the output. Assuming that (1) the input shape is $n_h \times n_w$, (2) the convolution kernel shape is $k_h \times k_w$, (3) the stride for the height/the width are s_h/s_w and (4) we add p_h rows and p_w columns of padding, the output shape will be

$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor$$

Here we have and 28×28 input, a 4×4 kernel, a 4×4 stride and a padding of 0 so the shape of our convolved image will be 7×7. Moreover, because we have a padding of 0, the application of the kernel for each 4×4 window of the original image will result in the following convolved image :

$$C[i,j] = \sum_{a=i\times s_h}^{i\times s_h + k_h} \left(\sum_{b=j\times s_w}^{j\times s_w + k_w} (O[a,b] \times weight) \right)$$

with O the original image, C the resulting image, $i \in 1, \ldots, 7$, $j \in 1, \ldots, 7$, $s_h \times s_w$ the stride shape, $k_h \times k_w$ the kernel shape and $weight$ the learnable weights of shape $k_h \times k_w$. The result is the following:



5 Problem 2: Dropout

5.1 High level description of the dropout

Gal and Gharamani[2015] showed that Monte Carlo Dropout simulation during training enable to approximate the varational inference thanks to q_θ a determined probability distribution. Gal and Gharamani[2016] explain that MonteCarlo Dropout is same as approximate $q_\theta(\omega)$ through a Bernoulli distribution $Z_i \hookrightarrow B(p_i)$ with p_i the probability of dropout on the NN weight ω_i. Each weight is therefore computed with the following method: $\omega_i = m_i * Z_i$. Learning to infer the parameter m_i is done thanks to the grandient descent explained before in the text. The dropout parameter (p_i) can be defined by the user. By the model structure of MonteCarlo Dropout, Gal and Gharamani[2016] showed that Bernoulli distribution has a on/off effect on the weight of NN such as on the following figure with p_i probability.

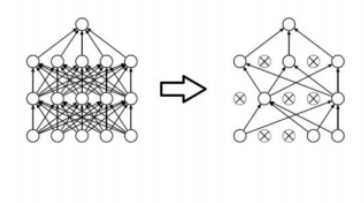


Figure 1: Impact du MonteCarlo Dropout sur un réseau de neurones

MonteCarlo Dropout has a deep impact on the neural network structure. Indeed, the model turn off some weights of the NN with a p_i probability and therefore to generate a new structure a each step of the training wich is different from the last one. The same author explain that if we maintain during the test phase, we can infer the epistemic uncertainty of our model.

5.2 High level description of your architecture

We created the `ConvNet2` model that has the following architecture:

- The first layer is firstly composed of a 2 dimensional convolutional layer (`nn.Conv2d(1, 8, kernel_size=5, stride=[1, 1], padding=2)`). We will then use the the `nn.Dropout2d(0.5)` model which turns off 50% of our neuron weights. Then we use a relu activation function on the remaining ones and a pooling layer (`nn.MaxPool2d(kernel_size=2, stride=2)`)). So the convolutional layer takes as an input a black & white 28×28 pixels image and increases its size to 32×32 pixels. Then it applies a 5×5 kernel that slides by 1 step in both (x, y) directions and the padding argument finally gives us a 28×28 output which, after the dropout, goes into the relu activation function and then in the pooling layer that applies to each of the 8 channels a 2×2 kernel that slides by 2 steps in both (x, y) directions meaning that the output will still have 8 channels but the images will be of size 14×14 pixels.
- Then we will flatten our features and put them into `nn.Linear(14 * 14 * 8, 500)`, where the input size is precisely the output size of our previous layer, and outputs a tensor of size 500. Then, we apply the relu activation function. Finally this output pass by a fully connected linear layer `nn.Linear(500, 10)` to match the dimension of 10 classes.