Report TP3 Apprentissage Statistique Appliqué

HEDFI Houeida

KONDO Godson Leopold Junior

December 22, 2020

1. Analytical problem solving

$$\min_{w \in R^5} (1 - x^\top w)^2 \text{ with } x = (1, \ldots, 1)^\top \in R^5$$

with $x = (1, \ldots, 1)^\top \in R^5$, we will get :

$$\min_{w_i \in R} (1 - w_1 - w_2 - w_3 - w_4 - w_5)^2$$

Which lays to $w_1 - w_2 - w_3 - w_4 - w_5 = 1$. The solution of the problem is all matrix $w \in R^5$ where $\sum_{i=1}^{5} w_i = 1$. The result of the gradient is $w = (0.2, 0.2, 0.2, 0.2, 0.2)$. We can easily noticed that, the result of the gradient is a particular case of the analytical solution where all w_i are equal.

2. Learning rate

Using the theory of numerical optimization, we can see that there is no specific learning rate to ensure the fastest convergence. It depends on the problem we are dealing with and our function to minimize. To have the best learning rate and to avoid to take too much time to converge or get an undesirable local minimum (too small learning rate) or jump over the minimum, we can tune it. The learning rate can also be fixed during the optimization process. The learning rate ranges often between 0.0 and 1.0.

(A value of 0.01 is typically used for multi-layers neural networks.)

3. loss.backward() and backpropagation

The "forward pass" refers to calculation process, values of the output layers from the inputs data. It's traversing through all neurons from first to last layer.

A loss function is calculated from the output values.

And then "backward pass" refers to process of counting changes in weights (de facto learning), using gradient descent algorithm (or similar). Computation is made from last layer, backward to the first layer.

Backpropagation repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. In other words, backpropagation aims to minimize the cost function by adjusting network's weights and biases. The level of adjustment is determined by the gradients of the cost function with respect to those parameters.

loss.backward() compute, from the loss function, the backward pass when we are doing an interation in an epoch. A backward pass is just one backpropagation (for feedforward neural nets) during an iteration.

4. Image plotting with break

The code is not plotting the same number all the time. The plots vary every time we run this cell. We get after each run a plot of an image coming from different batches.
We write break. This break command ensures that we plot only one number. Because our data is a loaded data, it is stored in memory. When we run the block for the n-th time, it is the number in the n-th position in the data that is plotted.

Problem 1: Logistic regression via pytorch

1. Mathematical description of logistic regression

Multinomial Logistic regression is model used to estimate the probabilities for the K categories of a qualitative dependent variable Y, using a set of explanatory variables X. It is parametrized by a weight matrix W and a bias vector b. The probabilities are obtained using the Softmax function. The probability that the element i (image in our case) belongs to the categoty k ($k = 1....K$) is written as follows:

$$\Pr(Y_{ik}) = \Pr(Y_i = k|x_i; W_1, ..., W_K, b_1, ..., b_K) = \frac{\exp(W_k^T x_i + b_k)}{\sum\limits_{k=1}^{K} \exp(W_k^T x_i + b_k)}$$

The prediction is then done by taking the argmax of the vector whose i'th element is P($Y_i = k|x$),$Y_{pred} = argmax\ \Pr(Y_i = k|x, W, b)$.
The optimal model parameters are obtained through the minimization of a loss function. In the case of multi-class logistic regression, it is very common to use the negative log-likelihood as the loss. Let us first start by defining the likelihood $\mathcal{L}$ and loss ℓ:

$$\mathcal{L}(W, b, \mathcal{D}) = \sum_{i=0}^{|\mathcal{D}|} \log(\Pr(Y = y^{(i)}|x^{(i)}, W, b))$$
$$\ell(W, b, \mathcal{D}) = -\mathcal{L}(W, b, \mathcal{D})$$

2. Mathematical description of optimization algorithm that we use

The loss function (the cross entropy loss specified above) is minimized using gradient descent method with mini-batches. The gradient descent proceeds as:

$$W_j = W_{j-1} - \eta \nabla \ell(W_{j-1})$$

$$b_j = b_{j-1} - \eta \nabla \ell(b_{j-1})$$

The gradient is calculated for every batch.

3. High level idea of how to implement logistic regression with pytorch

First, we create a logistic regression model class. Then, we instantiate the cross entropy loss function and we define the optimization algorithm. We iterate over the mini batches of our data; train, validation and test.

First of all we need to do the preprocessing DataSet (in our case with transforms.ToTensor()).
Second, we split data into mini-batches. In our case the mini batch size is equal to 32.
After that, **third**, we create a logistic regression model Class and we initialize it with specifying the input size as well as the output size.
Fourth, we define the Loss function. In our case, the Cross Entropy Loss function was the one defined.
Fifth, we define the optimizer class and the gradient descent procedure.
Sixth, we train the model in which we will iterate over the mini batches of our data.

- We initialize the parameters for the first forward stage.
- Get output given inputs.
- Apply softmax function.
- Calculate the cross entropy loss.
- Calculate gradients with respect to parameters.
- Update parameters using gradients.
- Clear gradient previously stored.
- Repeat the same procedure for all batches.
- Repeat for two epochs.

Seventh, we test the Model.

4. Classification accuracy on test data

The classification accuracy on test data is equal to 0.7237.

Problem 2: Dropout

1. High level description of the dropout

In deep learning, there are various ways to avoid the overfitting and the underfitting. One way is to regularize the networks. In *Pytorch*, one can penalize the norms of the parameter vector using **torch.norm** or use dedicated layers such as **torch.nn.Dropout**.
Dropout (Srivastava, Hinton, Krizhevsky, Sutskever, Salakhutdinov, 2014) consists in setting to 0 the activation of a certain fraction of the units in a layer. Doing this allows to avoid overfitting. We decide to use our dropout between our two layers. We choose 0.5 as the fraction of the units layers which activation are set to 0.

2. High level description of your architecture

We use the simple **ConvNet architecture** as the one used in the notebook.
We set two layers and a dropout layer. The first layer is a **Conv2D** which will not "*change*" the dimensions of our input. Indeed, our parameters are set to give at the output the same dimensions. The general formula for squared images and squared kernels is :

$$S_{out} = \frac{S_{in} - S_{kernel} + 2S_{padding}}{S_{stride}} + 1$$

And in our case it is :

$$S_{out} = \frac{28 - 5 + 4}{1} + 1 = 28$$

After that, first, the output will go to the **ReLu** activation function define as : $f(x) = x.\mathbb{1}_{x>0}$. The **ReLu** doesn't affect the size. Secondly, it will go eventually into the pooling layer. The pooling layer will be applied to each single channel and the output will have 8 channels and the images will be halfed in both (x, y) directions. Thus, the output will have 8 channels with 14×14 images.
Then we use our dropout to select randomly 50% of the units of the second layer which activation are set to 0.
After all that, we enter the second layer and we use a **nn.Linear** then we use **ReLu** and then

another **nn.Linear(500, 10)** to match the dimension of 10 classes.

The results of our model are better than before. We got 0.9824 on test accuracy.