TP 3 - Apprentissage Statistique Appliqué

Nokri Amale, Rahis Erwan, Vuillemot Bertrand

2020 - 2021

1 Part 1

1.1 Gradient descent solving

Question : Solving the problem $\min_{w\in\mathbb{R}^5}(1-x^\top w)^2$ with $x=(1,\dots,1)^\top\in\mathbb{R}^5$ analytically.
First we compute the gradient of the loss function $L(w)$:

$$\nabla L(w)=\left[\frac{\partial L(w)}{\partial w_1},\frac{\partial L(w)}{\partial w_2},\frac{\partial L(w)}{\partial w_3},\frac{\partial L(w)}{\partial w_4},\frac{\partial L(w)}{\partial w_5}\right]$$

We can rewrite the loss function as :

$$L(w)=\left(1-\sum_{i=1}^{5}x_iw_i\right)^2$$

We notice that we have a unique analytical expression for the partial derivatives of the loss function considering that $\forall i : x_i = 1$:

$$\forall i: \quad \frac{\partial L(w)}{\partial w_i}=-2\times\left(1-\sum_{j=1}^{5}w_j\right)$$

We want the gradient equal to zero as it means we are at the minimum of the function. We want the mean of the gradient elements equal to zero. As all the elements are equal to one another, we compute :

$$\begin{aligned}&\frac{\partial L(w)}{\partial w_i}=0\\ \Leftrightarrow\ &-2\times\left(1-\sum_{j=1}^{5}w_j\right)=0\\ &\Leftrightarrow\sum_{j=1}^{5}w_j=1\Rightarrow w_i=\frac{1}{5}=0.2\end{aligned}$$

The second derivative $\frac{\partial^2 L(w)}{\partial w_i^2}$ is positive so we know it is a convex function. We have the same result as with the gradient descent algorithm.

Question : What is the learning rate that we need to set to ensure convergence ?
The learning rate is introduced when using a gradient descent algorithm. The algorithm is based on a step-by-step exploration (descent) of the gradient of the optimized function. It is as follows:

Input : w_0 starting point, η learning rate
Loop : For $i = 1, \ldots, \max_{iter}$:
$w_i \leftarrow w_{i-1} - \eta \nabla L(w_{i-1})$
End

The learning rate η is the rate at which the gradient is explored. It shouldn't be too small otherwise the algorithm will take too long to converge. On the other side it shouldn't be too large otherwise the minimum value could be missed and the optimal solution never found. A learning rate of 0.01 will ensure a fast convergence but will not ensure that it is the global minimum of the function if it is not convex. We could also use a dynamic learning rate that decreases the more we approach the solution.

Question : Explain the connection of loss.backward() and the backpropagation for feedforward neural nets.
The backward method, that comes from the autograd library of PyTorch, compute the sum of the gradient of the given tensor. It is, in this case, applied on the loss tensor that computes the loss function. Il is connected to three tensors, x, y, and w. The last one, w is the tensor used for the gradient computation because when we created it we set the parameter `requires_grad` as true. It is useful for the back-propagation method in which we compute gradient of the loss function.

1.2 Multi layer perceptron

Question: Run the above block several times. Is it plotting the same number all the time? If not, why?

When we run the block several times, the plotting number is not the same all the time since the code block is an iteration over the data-set that stops every time it loops with the instruction `break`.

2 Problem 1 : Logistic regression via pytorch

2.1 Mathematical description of the logistic regression

Logistic regression is used to describe data and to explain the relationship between one dependent binary variable and one or more nominal, ordinal, interval or ratio-level independent variables. Here, we have a multinomial logistic regression.
In order to map predicted values to probabilities, we use the Sigmoid function. The function

maps any real value into another value between 0 and 1.

$x = \begin{pmatrix} x_1 \\ \dots \\ x_d \end{pmatrix}$ and $\mathcal{Y} = \{0, 1\}$

The prediction is : $y = g(x) = \dfrac{exp(b + w^T x)}{1 + exp(b + w^T x)} \in [0, 1], w \in \mathbb{R}^d, b \in \mathbb{R}$

We can write the logistic fonction $\sigma : \mathbb{R} \longrightarrow \mathbb{R}$:

$$\sigma(u) = \frac{e^u}{1 + e^u}$$

With the multinominial fonction, we can write :

$$\sigma(z) = \frac{1}{\sum_{k=1}^{K} e^{z_k}} \begin{pmatrix} e^{z_1} \\ \dots \\ e^{z_k} \end{pmatrix} \in \mathbb{R}^K$$

This function is softmax function, which is used in multinomial logistic regression and is often used as the last activation function of a NN to normalize the output of a network to a probability distribution over predicted output classes :

$$g(x) = \sigma(w_1^T x + b_1, \dots, w_K^T x + b_k)$$

2.2 Mathematical description of the optimization algorithm usec

We use the Stochastic Gradient Descent (SGD) optimizer. To do this, the sample is partitioned into batches B_1 to B_M such as :

$$card(B_m) \simeq [\frac{n}{M}] m = 1, \cdots, M$$
$$B_1 \cup \cdots \cup B_M = \{1, \cdots, n\}$$

For every iteration, we replace $\nabla L_n(\Theta^{(t)})$ by :

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

We have, $\forall \theta$:

$$\mathbb{E}[\nabla L^m(\theta)] = \nabla L_n(\theta)$$

2.3 High level idea of how to implement logisitic regression with pytorch

We initialized our model with this linear layer: torch.nn.Linear(input size, output size). For the activation function, we used `torch.nn.Softmax()`. The softmax function is used as the activation function in the output layer of neural network models that predict a multinomial

probability distribution.
Next, we used SGD optimizer for the update of hyperparameters. model.parameters() provided the learnable parameters to the optimizer and $lr = 0.01$ defines the learning rates for the parameter updates.
Then, we used the same architecture than the TP.

2.4 Report classification accuracy on test data.

The classification accuracy on test data is : 0.898.

3 Elements of CNN

3.1 Understanding the convolutional layer

Problem : Implement the convolutional layer method by hand

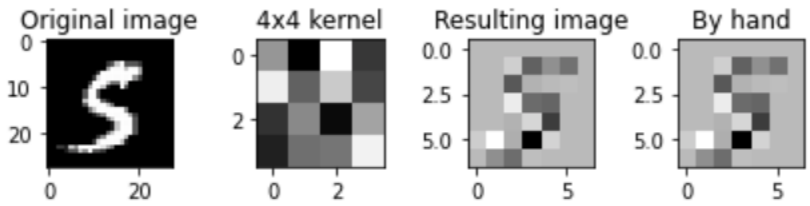

Figure 1: Convolution plot

In this case we apply a 4×4 (K) kernel on the 28×28 (N) input picture (greyscale). The stride, or step size is $(4, 4)$ (S). It means that we go through the input picture with a 4×4 rolling windows that computes the weighted sum of the pixel values. The resulting feature map is a $\frac{N-K}{S} = \frac{28-4}{4} + 1 = 7$ by 7 output picture. We do this hand convolution as follows :

```
np_image = image[0][0].data.numpy()
image_conv = np.zeros((7,7))
for i in range(7):
    for j in range(7):
        image_conv[i, j]=np.sum(np_image[(4*i):(4*i+4), (4*j):(4*j+4)] *
    weight)
```

4 Problem 2 : Dropout

Question: Modify the code for ConvNet and insert Dropout layer (wherever you want).
The dropout allows us to shut down parts of the neural networks to prevent from over-fitting. A fully connected layer occupies most of the parameters so it risks to favour co-dependency between neurons during the train phase. This would result in poor performances on held-out

test data.
It's a regularization approach which allows us to reduce interdependency (also called co-adaptation) between neurons with a view to increasing robustness of the model. This approach looks alike the Random Forest with iterated random features and data selections. But here, we will be randomly deactivating p*node on each layer and at each iteration.

More precisely :

- During the training phase, for each hidden layer, for each training sample, for each iteration, we will ignore nodes (by "zeroing" them) with a probability p using samples from a Bernoulli distribution.
- During the testing phase, we will use (1-p) activations to take into account the missing nodes (hence activations) during the training phase.

The method dropout in Pytorch takes two arguments, the first one is p the probability of node dropout which is set by default to 0.5 (following the paper by Hinton et Al. in 2012). The second one is inplace set by default to False. We decide to use the `Dropout2d` function that works on the level of the channels. It will randomly set to zero entire channels (entire feature maps). We put it just after the convolutional layer. Indeed, the convolution layer has extended the channel size of the output from 1 to 8.

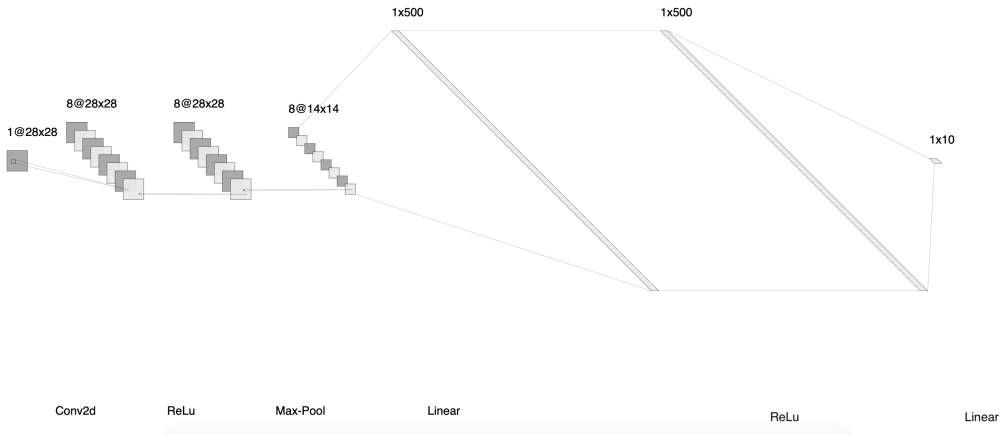

Figure 2: Neural Network plot

As the figure 2 suggests, we use a convolutionnal layer, followed by the dropout, a ReLu activation and a Max-Pool to diminish the dimension of the data. Then the output is flattened in a 1×500 vector and passed through a ReLu activation before the last linear transformation that takes the 1×500 vector and diminishes it into a 1×10 vector for the 10 different possible outputs.

References

[1] 100 - Logistic Regression with IRIS and pytorch — ensae_teaching_dl.

[2] Dropout — PyTorch 1.7.0 documentation.

[3] Hyperparameter tuning with Ray Tune — PyTorch Tutorials 1.7.1 documentation.

[4] Amar Budhiraja. Learning Less to Learn Better — Dropout in (Deep) Machine learning, March 2018.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. Adaptive computation and machine learning. The MIT Press, Cambridge, Massachusetts, 2016.

[6] Andrea Eunbee Jang. PyTorch: Linear and Logistic Regression Models, February 2019.

[7] Grégoire Montavon, editor. *Neural networks: tricks of the trade*. Number 7700 in Lecture notes in computer science. Springer, Heidelberg, 2. ed edition, 2012. OCLC: 828098376.

[8] Shubhendu Trivedi and Risi Kondor. Lecture 7 Convolutional Neural Networks - CMSC 35246: Deep Learning. page 97.