Gradient of the SVM Hinge loss

Victor BUSA victor.busa@gmail.com

April 9, 2017

When I started to follow CS231n course from Stanford as a self-taught person, I was a bit irritated that they weren't more explanations about how we are supposed to compute the gradient of the hinge loss. Actually, in the lecture course (http://cs231n.github.io/optimization-1/) we can see a formula for the gradient of the SVM loss. Although the formula seems understandable, I still thinks we might need to get our hands dirty by doing the math. Indeed, what does $\nabla_{w_{yi}} Li$ means ? Let's dive into how we can compute the gradient of the SVM loss function

Loss Function In this part, I will quickly define the problem according to the data found in assignment1 of CS231n. Let's define our Loss function by:

$$L_i = \sum_{j \neq y_i} [\ max(0, x_i w_j - x_i w_{y_i} + \Delta)\ ]$$

Where:

- w_j are the column vectors. So for example $w_j^\intercal = [w_{j1},\ w_{j2},\ \ldots, w_{jD}]$
- $X \in \mathbb{R}^{N \times D}$ where each x_i are a single example we want to classify. $x_i = [x_{i1},\ x_{i2},\ \ldots,\ x_{iD}]$
- hence i iterates over all N examples
- j iterates over all C classes.
- y_i is the index of the correct class of x_i
- Δ is the margin parameter. In the assignment $\Delta = 1$
- also, notice that $x_i w_j$ is a scalar

Analytic gradient We want to compute $\forall i, j \in [1, N] \times [1, C]\ \nabla_{w_j} L_i$. As we know $w_j \in \mathbb{R}^{D \times 1}$, so we can write:

$$\nabla_{w_j} Li = \begin{bmatrix} \frac{dLi}{dw_{j1}} \\ \frac{dLi}{dw_{j2}} \\ \vdots \\ \frac{dLi}{dw_{jD}} \end{bmatrix}$$

Hence, let's find the derivative of $\frac{dLi}{dw_{kj}}$ with $k \in [1,\ C]$. To compute this derivative I will write L_i without $\sum$ so it will be easier to visualize:

$$
\boxed{
\begin{aligned}
L_i = &\max(0, x_{i1}w_{11} + x_{i2}w_{12} + \ \dots \ + x_{ij}w_{1j} + \ \dots \ + x_{iD}w_{1D} - x_{i1}w_{y_i1} - x_{i2}w_{y_i2} + \ \dots \ - x_{y_iD}w_{1D}) + \\
&\max(0, x_{i1}w_{21} + x_{i2}w_{22} + \ \dots \ + x_{ij}w_{2j} + \ \dots \ + x_{iD}w_{2D} - x_{i1}w_{y_i1} - x_{i2}w_{y_i2} + \ \dots \ - x_{y_iD}w_{1D}) + \\
&\vdots \\
&\max(0, x_{i1}w_{k1} + x_{i2}w_{k2} + \ \dots \ + x_{ij}w_{kj} + \ \dots \ + x_{iD}w_{CD} - x_{i1}w_{y_i1} - x_{i2}w_{y_i2} + \ \dots \ - x_{y_iD}w_{1D}) + \\
&\vdots \\
&\max(0, x_{i1}w_{C1} + x_{i2}w_{C2} + \ \dots \ + x_{ij}w_{Cj} + \ \dots \ + x_{iD}w_{CD} - x_{i1}w_{y_i1} - x_{i2}w_{y_i2} + \ \dots \ - x_{y_iD}w_{1D}) +
\end{aligned}
}
$$

So, now that we can see things quite easily, we see that:

$$\forall k \in [1,\ C] \backslash \{y_i\},\ \forall j \in [1,\ D]\ \frac{dLi}{dw_{kj}} = 1(x_i w_k - x_i w_{y_i} + \Delta > 0) x_{ij}$$

Using the definition of $\nabla_{w_j} Li$, we now have:

$$\nabla_{w_j} Li = \begin{bmatrix} \frac{dLi}{dw_{j1}} \\ \frac{dLi}{dw_{j2}} \\ \vdots \\ \frac{dLi}{dw_{jD}} \end{bmatrix} = \begin{bmatrix} 1(x_i w_j - x_i w_{y_i} + \Delta > 0) x_{i1} \\ 1(x_i w_j - x_i w_{y_i} + \Delta > 0) x_{i2} \\ \vdots \\ 1(x_i w_j - x_i w_{y_i} + \Delta > 0) x_{iD} \end{bmatrix} = 1(x_i w_j - x_i w_{y_i} + \Delta > 0) \begin{bmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{iD} \end{bmatrix}$$

Now, what happen when $y_i = k$? Using the form of L_i in the box, we see that w_{y_ij} intervenes in all lines. Hence we have that:

$$y_i = k,\ \forall j \in [1,\ D]\ \frac{dLi}{dw_{y_ij}} = -\sum_{k \neq y_i} 1(x_i w_k - x_i w_{y_i} + \Delta > 0) x_{ij}$$

leading to:

$$\nabla_{w_{y_i}} Li = \begin{bmatrix} \frac{dLi}{dw_{y_i1}} \\ \frac{dLi}{dw_{y_i2}} \\ \vdots \\ \frac{dLi}{dw_{y_iD}} \end{bmatrix} = \begin{bmatrix} -\sum\limits_{k \neq y_i} 1(x_i w_k - x_i w_{y_i} + \Delta > 0) x_{i1} \\ -\sum\limits_{k \neq y_i} 1(x_i w_k - x_i w_{y_i} + \Delta > 0) x_{i2} \\ \vdots \\ -\sum\limits_{k \neq y_i} 1(x_i w_k - x_i w_{y_i} + \Delta > 0) x_{iD} \end{bmatrix} = -\sum_{k \neq y_i} 1(x_i w_k - x_i w_{y_i} + \Delta > 0) \begin{bmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{iD} \end{bmatrix}$$

Vectorized implementation Now that we understand how we got the gradient of the hinge loss function. We will compute the gradient using Numpy and a vectorized implementation (the unvectorized implementation is quite straightforward). I won't put the Python code here, I will just use image and pseudo code to present the result. The Python implementation can be found in the linear_svm.py file.

Forward pass Firstly we will focus on the implementation of the forward pass. In other words, we will derive a formula to compute the loss with a vectorized implementation. For a better understanding, I created a picture:

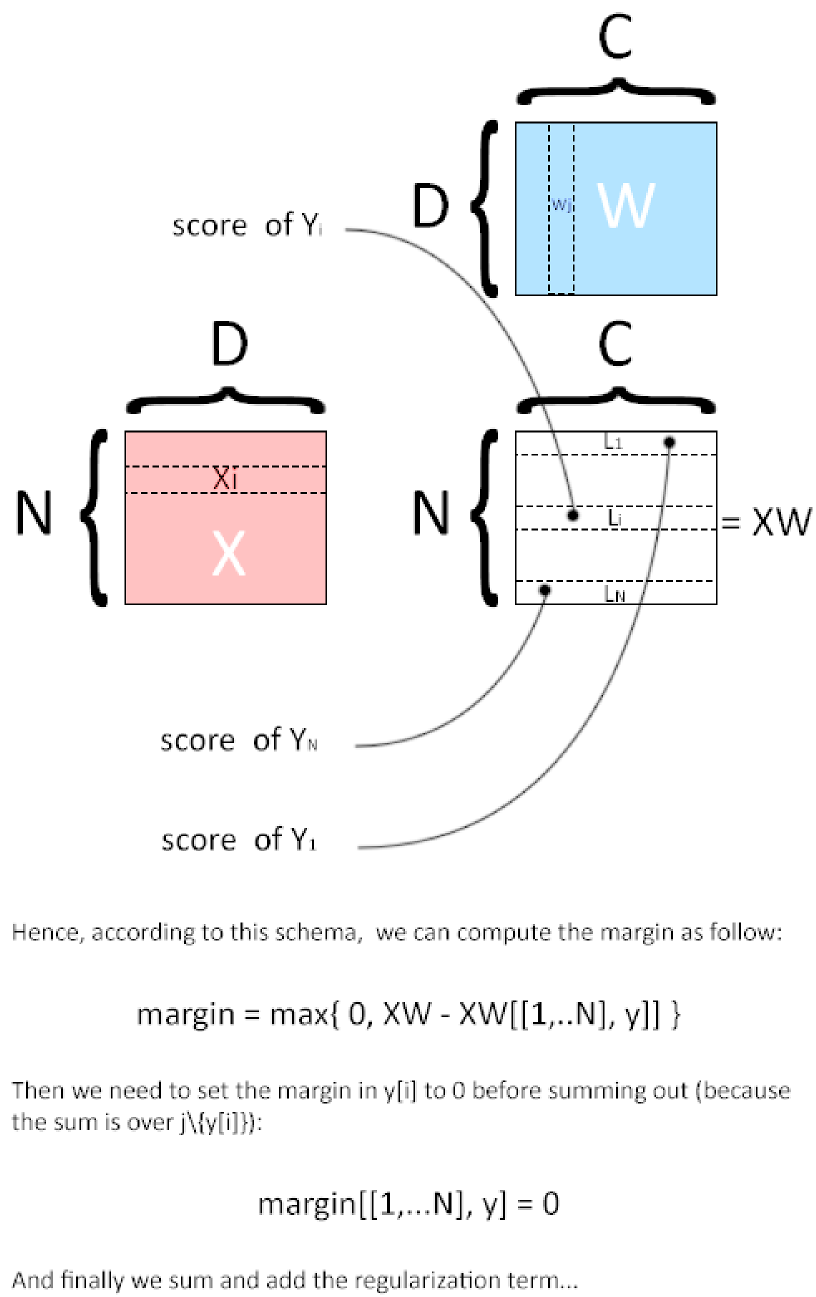

Figure 1: Hinge loss - vectorized implementation

Backward pass Now that we understand how to implement the forward pass, we will deal with a slightly more difficult challenge. How to compute the backward pass, that is to say, how to compute $\nabla_w L$ with a vectorized implementation.

Firstly, we will rewrite our $\Delta_{w_j} L_i$ to have a better understanding of what the matrix should look like:

$$\nabla_{w_j} L_i = \begin{bmatrix} \frac{dL_i}{dw_1} & \frac{dL_i}{dw_2} & \dots & \frac{dL_i}{dw_C} \end{bmatrix} = \begin{bmatrix} \frac{dL_i}{dw_{11}} & \frac{dL_i}{dw_{21}} & \dots & \frac{dL_i}{dw_{y_i1}} & \dots & \frac{dL_i}{dw_{C1}} \\ \frac{dL_i}{dw_{12}} & \frac{dL_i}{dw_{22}} & \dots & \frac{dL_i}{dw_{y_i2}} & \dots & \frac{dL_i}{dw_{C2}} \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \frac{dL_i}{dw_{1j}} & \frac{dL_i}{dw_{2j}} & \dots & \frac{dL_i}{dw_{y_ij}} & \dots & \frac{dL_i}{dw_{Cj}} \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \frac{dL_i}{dw_{1D}} & \frac{dL_i}{dw_{2D}} & \dots & \frac{dL_i}{dw_{y_iD}} & \dots & \frac{dL_i}{dw_{CD}} \end{bmatrix}$$

$$= \begin{bmatrix} 1(x_iw_1 - x_iw_{y_i} + \Delta > 0)x_{i1} & \dots & -\sum\limits_{j \neq y_i} 1(x_iw_j - x_iw_{y_i} + \Delta > 0)x_{i1} & \dots & 1(x_iw_C - x_iw_{y_i} + \Delta > 0)x_{i1} \\ 1(x_iw_1 - x_iw_{y_i} + \Delta > 0)x_{i2} & \dots & -\sum\limits_{j \neq y_i} 1(x_iw_j - x_iw_{y_i} + \Delta > 0)x_{i2} & \dots & 1(x_iw_C - x_iw_{y_i} + \Delta > 0)x_{i2} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 1(x_iw_1 - x_iw_{y_i} + \Delta > 0)x_{ij} & \dots & -\sum\limits_{j \neq y_i} 1(x_iw_j - x_iw_{y_i} + \Delta > 0)x_{ij} & \dots & 1(x_iw_C - x_iw_{y_i} + \Delta > 0)x_{ij} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 1(x_iw_1 - x_iw_{y_i} + \Delta > 0)x_{iD} & \dots & -\sum\limits_{j \neq y_i} 1(x_iw_j - x_iw_{y_i} + \Delta > 0)x_{iD} & \dots & 1(x_iw_C - x_iw_{y_i} + \Delta > 0)x_{iD} \end{bmatrix}$$

Now that we see the shape of the matrix is is easy to implement the unvectorized formula. We just need to:

- construct a matrix of zeros having shape (D,C) (same shape as W)
- assign x_i to each column of this matrix if $j \neq y_i$ and $(x_iw_1 - x_iw_{y_i} + \Delta > 0)$
- assign $-\sum\limits_{j \neq y_i} 1(x_iw_j - x_iw_{y_i} + \Delta > 0)x_i$ to the y_i column

Now, the vectorized implementation is slightly harder to compute but fortunately we've already done the job. Actually we computed in the forward pass (see Forward pass) a matrix having on each of his element (besides $j = y_i$ where it is 0):

$$(x_iw_j - x_iw_{y_i} + \Delta > 0)$$

So this matrix (let's call it the **margin matrix**) looks like what we want except that:

1. We want to construct a matrix that has the same shape as the margin matrix and that has 1 when the quantity of each cell of the margin matrix is positive and a zero otherwise
2. We want to construct a matrix that have on each cell of its $j = y_i$ column the negative sum of the indicator function of all the columns (except column y_i) of margin matrix
3. We need to multiply this newly created matrix by X (because we see x_{ij} is present in each cell of $\nabla_{w_j} L_i$)

So now, it is relatively straightforward:

1. We create a matrix of the same size of the margin matrix. Let's call it **mask**. Then we need to have 1 on each cell of the **mask** matrix when the quantity on the corresponding cell of the **margin matrix** is positive. In python we can do this using:
   $mask[margin > 0] = 1$

2. Now, we need to change the content of each cell of **mask matrix** when we are on the y_ith column. And we need to put in each row of this y_ith column the negative value of the sum of all the value in the other rows. Hence in python we can do that by creating a vector containing the sum of the column:

   $$np\_sup\_zero = np.sum(mask, axis = 1)$$

   and then we replace the y_ith column vector of the **mask matrix** by this new vector by doing:

   $$mask[np.arange(num\_train), y] = -np\_sup\_zero$$

3. finally we need to multiply by X so the final matrix is of shape (D,C) the same shape as W. We know mask's dimension is (N,C) and X's dimension is (N, D) so we need to return $X^{\intercal}W$

Don't forget to divide by the number of training samples and to add the regularization term.

Conclusion Finally we saw how to compute a big matrix gradient and how matrix visualization can quickly help us elaborate techniques to implement vectorization.