Backpropagation in Neural Network

Victor BUSA victor.busa@gmail.com

April 12, 2017

Well, again, when I started to follow the Stanford class as a self-taught person. One thing really bother me about the computation of the gradient during backpropagation. It is actually the most important part in the first 5 lessons and yet all the examples from Stanford class are on 1-D functions. You can see those examples via this link: http://cs231n.github.io/optimization-2/ Well the advice that they give is that on higher dimension it "works quite the same". It is actually not false but I think we style need to do the math to see it. So, in this paper I will compute the gradient on higher dimension of the **ReLu**, the **bias** and the **weight matrix** in a fully connected network.

Forward pass Before dealing with the backward pass and the computation of the gradient in higher dimension, let's compute the forward pass first, and then we will backpropagate the gradient. Using the notation of the assignment, we have:

$$
\begin{aligned}
y_1 &= XW_1 + b_1 \\
h_1 &= max(0, y_1) \\
y_2 &= h_1 W_2 + b_2 \\
L_i &= \left(\frac{e^{y_{2_{y_i}}}}{\sum_j e^{y_{2_j}}} \right) \\
L &= \frac{\sum_{samples} L_i}{N}
\end{aligned}
$$

where N=number of training sample

In python code we can compute the forward pass using the following code:

```
y1 = X.dot(W1) + b1 #(N,H) + (H)
h1 = np.maximum(0, y1)
y2 = h1.dot(W2) + b2
scores = y2

# correspond to e^y2 in maths
exp_scores = np.exp(scores)

# correspond to e^y2/sum(e^(y2)[j]) in maths
probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

# correspond to -log[(e^y2/sum(e^(y2)[j]))[yi]] = Li in maths
correct_logprobs = -np.log(probs[range(N), y])

# correspond to L
loss = np.sum(correct_logprobs) / N
```

And using the notation of the assignment, we have:

- $X \in \mathbb{R}^{N \times D}$, $W_1 \in \mathbb{R}^{D \times H}$, $b_1 \in \mathbb{R}^H$
- $h_1 \in \mathbb{R}^{N \times H}$, $W_2 \in \mathbb{R}^{H \times C}$
- $y_2 \in \mathbb{R}^{N \times C}$, $b_2 \in \mathbb{R}^C$

Backpropagation pass

gradient of the Softmax We already saw (previous paper) that the gradient of the softmax is given by:

$$\frac{\partial L_i}{\partial f_k} = (p_k \ - \ 1(k \ = \ y_i))$$

where:

$$p_{y_i} = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}}$$

So actually the gradient of the loss with respect to y_2 is just the matrix *probs* (see python code of the forward pass) in which we substract 1 only in the y_i^{th} column. And we need to do this for each row of the matrix probs (because each row correspond to a sample). So in python we can write:

```
dy2 = probs
dy2[range(N),y] -= 1
dy2 /= N
```

Note : We divide by N because the total loss is averaged over the N samples (see forward pass code).

Gradient of the fully connected network (weight matrix) Then now we want to compute $\frac{dL}{dW_2}$. To do so, we will use the chain rule. Note that to avoid complex notation, I rewrite W_2 as being W and w_{ij} being the coefficient of W (b_2 is replace by b, y_2 by y and h_1 by h). As L is a scalar we can compute $\frac{\partial L}{\partial w_{ij}}$ directly. To do so, we will use the chain rule in higher dimension. Let's recall first that with our simplified notation, we have:

$$y = hW + b$$

where:

- h is a (N, H) matrix, W is a (H, C) matrix
- b is a (C, 1) column vector

$$\frac{dL}{dw_{ij}} = \sum_{p,q} \frac{dL}{dy_{pq}} \frac{dy_{pq}}{dw_{ij}} \tag{1}$$

We already know all the $\frac{dL}{dy_{pq}}$ (this is the term of the $\frac{\partial L}{\partial y_2}$ we computed in the previous paragraph). So we only need to focus on computing $\frac{dy_{pq}}{dw_{ij}}$:

$$\begin{aligned} \frac{dy_{pq}}{dw_{ij}} &= \frac{d}{dw_{ij}} \left(\sum_{u=1}^{H} h_{pu} w_{uq} + b_q \right) \\ &= 1\{q = j\} h_{pi} \end{aligned} \tag{2}$$

So finally replacing (2) in (1) we have:

$$\frac{dL}{dw_{ij}} = \sum_{p,q} \frac{dL}{dy_{pq}} 1\{q = j\} h_{pi} = \sum_{p} \frac{dL}{dy_{pj}} h_{pi} = \sum_{p} h_{p,i} \frac{dL}{dy_{pj}} = \sum_{p} h^{\intercal}_{i,p} \left(\frac{dL}{dy} \right)_{p,j} \tag{3}$$

We used the fact the h_{pi} and $\frac{dL}{dy_{pj}}$ are scalars and $\times$ is a commutative operation for scalars. Finally we see that:

$$\left(\frac{\partial L}{\partial W} \right)_{i,j} = \sum_{p} h^{\intercal}_{i,p} \left(\frac{\partial L}{\partial y} \right)_{p,j}$$

So we recognize the product of two matrix: $h^{\intercal}$ and $\frac{\partial L}{\partial y}$ Using the assignment notations we have:

$$\frac{\partial L}{\partial W_2} = h_1^{\intercal} \frac{\partial L}{\partial y_2}$$

In python we denote dx as being $\frac{\partial L}{\partial x}$, so we can write:

```
dW2 = h1.T.dot(dy2)
```

Gradient of the fully connected network (bias) Let's define $Y = Wx + b$ with x being a column vector. With the notation of the assignment we have:

$$\begin{bmatrix} w_{11} & w_{12} & \dots & w_{1C} \\ w_{21} & w_{22} & \dots & w_{2C} \\ \vdots & \ddots & \ddots & \vdots \\ w_{H1} & w_{H2} & \dots & w_{HC} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_C \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_H \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_H \end{bmatrix} \tag{4}$$

We already computed $\frac{\partial L}{\partial y_2}$ (gradient of the softmax), so according to the chain rule we want to compute:

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial b}$$

Note that here $y_2 = y$ to simplify the notations.

also according to (4), we saw that $\forall i \neq j$:

$$\frac{dy_i}{db_j} = \frac{d}{db_j}\left(\sum_{k=1}^{C} w_{ik}x_k + b_i\right) = 0$$

also if $i = j$:

$$\frac{dy_i}{db_i} = \frac{d}{db_i}\left(\sum_{k=1}^{C} w_{ik}x_k + b_i\right) = 1$$

hence we have that $\frac{\partial y_2}{\partial b}$ is the identity matrix (1 on the diagonal and 0 elsewhere). Noting this matrix I_{HC}, we have:

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y_2} I_{HC} = \frac{\partial L}{\partial y_2}$$

Now, if we are dealing with X as being a matrix we can simply noticed that the gradient is the sum of all local gradient in y_i (see Figure 1). So we have:

$$\frac{\partial L}{\partial b} = \sum_{i=1}^{n} \frac{\partial L}{\partial y_i}\frac{\partial y_i}{\partial b} = \sum_{i=1}^{n} \frac{\partial L}{\partial y_i}$$

In python, we can achieve the gradient with the following code:

```
db2 = np.sum(dy2, axis=0)
```

Gradient of ReLu I won't enter into to much details as we understand how it works now. We use the chain rule and the local gradient. Here again I will focus on computing the gradient of x, x being a vector. In reality X is actually a matrix as we use mini-batch and vectorized implementation to speed up the computation. For the local gradient we have:

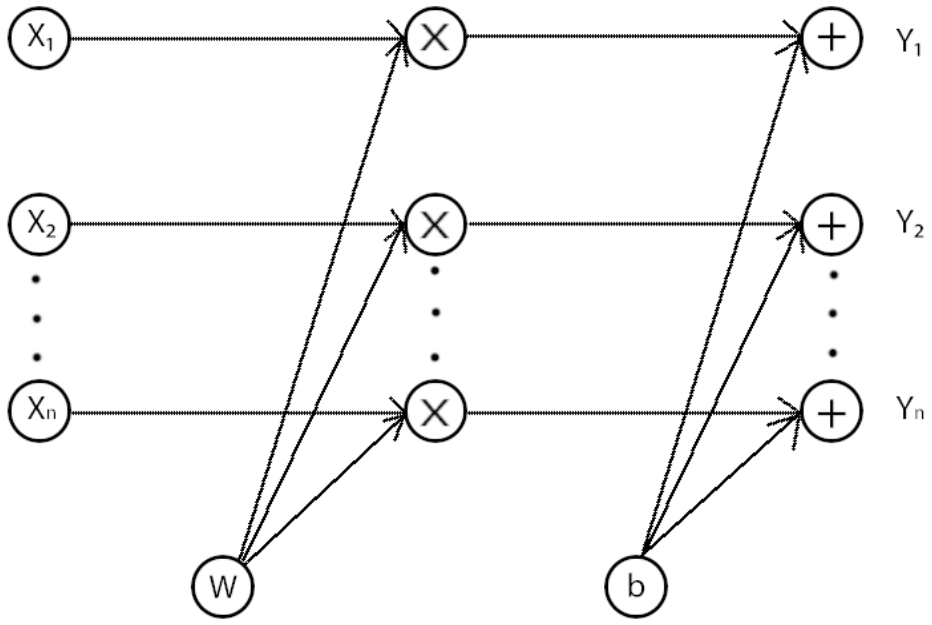

Figure 1: Gradient of the bias. We see that b receives n incoming gradients. So we have to add all those incoming gradients to get the gradient w.r.t the bias

$$\frac{\partial}{\partial x}\left(ReLu(x)\right) = \frac{\partial}{\partial x} max(0,x) = \begin{bmatrix} \frac{\partial}{\partial x_1} max(0,x_1) & \frac{\partial}{\partial x_2} max(0,x_1) & \dots & \frac{\partial}{\partial x_H} max(0,x_1) \\ \frac{\partial}{\partial x_1} max(0,x_2) & \frac{\partial}{\partial x_2} max(0,x_2) & \dots & \frac{\partial}{\partial x_H} max(0,x_2) \\ \vdots & \ddots & \ddots & \vdots \\ \frac{\partial}{\partial x_1} max(0,x_H) & \frac{\partial}{\partial x_2} max(0,x_H) & \dots & \frac{\partial}{\partial x_H} max(0,x_H) \end{bmatrix}$$

$$= \begin{bmatrix} 1(x_1 > 0) & 0 & \dots & 0 \\ 0 & 1(x_2 > 0) & \dots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 1(x_H > 0) \end{bmatrix}$$

and then using chain rule we have what we want:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial}{\partial x}\left(ReLu(x)\right) = \frac{\partial L}{\partial y}\begin{bmatrix} 1(x_1 > 0) & 0 & \dots & 0 \\ 0 & 1(x_2 > 0) & \dots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 1(x_H > 0) \end{bmatrix}$$

In python ($dy1$ being $\frac{\partial L}{\partial x}$ and $dh1$ being $\frac{\partial L}{\partial y}$), we can write:

```
dy1 = dh1 * (y1 >= 0)
```

Note : I let the reader compute the gradient of the Relu if x is a matrix. It isn't difficult. We just need to use the chain rule in higher dimension (like I did for the computation of the Gradient w.r.t the weight matrix). I preferred to use x as a vector to be able to visualize the Jacobian of the Relu.

Conclusion In this paper we tried to understand quite precisely how to compute the gradient in higher dimension. We hence gain a better understanding of what's happening behind the python code and are ready to compute the gradient of other activations functions or other kind of layers (not necessarily fully connected for example).