1 General structure

The goal of the NeuronLearning project is to demonstrate that a a neural network can be taught without comparison to a final result.

The basic structure is expressed in the Network class. It has a hierarchical build:

- axons: this is the most basic block
- layer: this provides the geometry for axons.
- connection: establishes connections between layers
- update: provides an update function for each layer

Once we have a network that works effectively enough, we may add some other "skins" to it that specifies the way we can use it.

2 Backpropagation

When we want to do supervised learning, then the result of the last layer should be compared to an expected output. The result of the comparison is quantified by some loss function y. The goal of the learning is to minimize the loss over the training set. For that we have to know, in which direction should we change the parameters of the network in order to improve the loss. For that we have to know the derivative of the loss with respect to the parameters.

The loss L is formally a function of the input layer (x_0) and the parameters of the network (w), i.e. $L(w, x_0)$. We would need

$$\frac{\partial L}{\partial w_a}. \tag{1}$$

In general the derivatives can be computed one-by-one. But in a neural network we have a hierarchical structure. Let us denote the value of the ith axon by x_i. If x is changed, then also y changes, and we may want to compute

$$z_i = \frac{\partial L}{\partial x_i}, \tag{2}$$

Once we know these values, then the quantity of interest can be computed as (here and later on we suppress the notation of any sums)

$$\frac{\partial L}{\partial w_a} = \frac{\partial L}{\partial x_i}\frac{\partial x_i}{\partial w_a} = z_i \frac{\partial x_i}{\partial w_a}. \tag{3}$$

Usually a parameter is local, so there is only a single z_i that contributes in the above expression.

In a hierarchical structure y does not depend *directly* on x_i, only through some other x_j which are higher in the hierarchy. If we want to quantify this statement, we compose the index i from two parts $i = (n, a)$ where n is the layer level and a is the position within the layer. An axon x_j is at higher hierarchy than x_i if its layer level is larger, i.e. $j_1 > i_1$. But this composition may remain hidden in the notation. We simply write

$$\frac{\partial y(x_i)}{\partial x_i} = \frac{\partial y(x_j(x_i))}{\partial x_i} = \frac{\partial y}{\partial x_j}\frac{\partial x_j}{\partial x_i}, \tag{4}$$

which means

$$z_i = z_j \frac{\partial x_j}{\partial x_i}, \qquad \text{where } j_1 > i_1. \tag{5}$$

The derivative $\partial x_j / \partial x_i$ can be explicitly computed by knowing the connection between the axons. Then the above equation tells up that we can compute z_i, if we know all derivatives z_j at *higher* levels. This is the method of *backpropagation*.

2.1 Affine plus nonlinear layers

For example if the rule of getting from one layer to another is

$$x_j = f_j\left(M_{ji}x_i + b_j\right), \tag{6}$$

then

$$\frac{\partial y}{\partial x_i} = \frac{\partial y}{\partial x_j}\frac{\partial x_j}{\partial x_i} = \frac{\partial y}{\partial x_j} f_j(M_{ji}x_i + b_j)M_{ji}. \tag{7}$$

So we find

$$z_i = z_j f_j'(M_{ji}x_i + b_j)M_{ji}. \tag{8}$$

Moreover

$$\begin{aligned}\frac{\partial y}{\partial M_{ji}} &= \frac{\partial y}{\partial x_j}\frac{\partial x_j}{\partial M_{ji}} = z_j f_j'(M_{ji}x_i + b_j)x_i \\ \frac{\partial y}{\partial b_j} &= \frac{\partial y}{\partial x_j}\frac{\partial x_j}{\partial b_j} = z_j f_j'(M_{ji}x_i + b_j).\end{aligned} \tag{9}$$

Technically it is possible to compute $f_j'(M_{ji}x_i + b_j)$ at the same time when $x_j = f_j(M_{ji}x_i + b_j)$ is computed. But if we want apply backpropagation as a skin, we should have a function that determines f' from the value of x_j, which is already stored in the axon variable. It is possible if f is invertible:

$$f_j'(M_{ji}x_i + b_j) = f_j'(f_j^{-1}(x_j)) \stackrel{!}{=} df_j(x_j). \tag{10}$$

To have some examples:

$$\begin{aligned}&f(x) = \Theta(x)x \quad \Rightarrow \quad df = \Theta(f > 0) \qquad (\texttt{ReLU}) \\ &f(x) = \alpha\tanh(\beta x) \quad \Rightarrow \quad f'(x) = \alpha\beta\left(1 - \frac{f^2}{\alpha^2}\right), \\ &f(x) = \alpha\exp(\beta x) \quad \Rightarrow \quad f'(x) = \beta x, \\ &f(x) = x|_{\text{largest}} \quad \Rightarrow \quad f'(x) = 1|_{\text{largest}} \qquad (\texttt{maxpool})\ .\end{aligned} \tag{11}$$

In case of maxpool there is no parameters for the connection strength. In order to be treated in a uniform way, however, we require that the update funtion should set all of the $M_{ji} = 0$, except for the largest x, where it is 1. It is then ensured that the derivative propagates back only for the largest value.

In this way we have the recursions for $j_1 > i_1$:

$$\begin{aligned}&\frac{\partial y}{\partial b_j} = z_j df_j(x_j), \quad \frac{\partial y}{\partial M_{ji}} = z_j df_j(x_j)x_i \\ &z_i = z_i + z_j df_j(x_j)M_{ji}, \quad \text{starting from } z_i = 0.\end{aligned} \tag{12}$$

Then we can maintain the effectivity of the code as well as the modularity.

2.2 Loss functions and derivatives

The last layer is special, since it is directly connected to the loss function. In this case there are no parameters, backpropagation only propagates the derivative to its input sites. The loss is always a result of the comparison of the actual output x and the expected output $\bar{x}$.

Sometimes the loss is used on normalized output. This means that we compute $L(x) = \ell(\frac{x}{X})$, where $X = \sum_i x_i$. Its derivative reads

$$\frac{\partial L}{\partial x_i} = \frac{1}{X}\left(\partial_i\ell - C\right), \qquad \text{where} \quad C = \frac{1}{X}\sum_j x_j\partial_j\ell, \tag{13}$$

C is a constant. The last term ensures that $\sum_i x_i\partial_i L = 0$.

The most simple function is the p-norm loss function

$$\ell_p(x) = \frac{1}{p}\sum_i |x_i - \bar{x}_i|^p, \qquad \frac{\partial\ell_p}{\partial x_i} = \operatorname{sgn}(x_i - \bar{x}_i)|x_i - \bar{x}_i|^{p-1}. \tag{14}$$

Another, frequently used function is the Kullback-Leibler divergence:

$$\ell_{KL}(x) = \sum_i x_i \log \frac{x_i}{\bar{x}_i}, \qquad \frac{\partial \ell_{KL}}{\partial x_i} = \log \frac{x_i}{\bar{x}_i} + 1. \tag{15}$$

With normalizing output ($X = \sum_i x_i$)

$$L_{KL}(x) = \frac{1}{X} \sum_i x_i \log \frac{x_i}{X\bar{x}_i}, \qquad \frac{\partial L_{KL}}{\partial x_i} = \frac{1}{X} \left(\log \frac{x_i}{X\bar{x}_i} - L_{KL} \right) \tag{16}$$

The KL loss function has the following properties, for simplicity for two variables. Then, assuming that their sum is one

$$L_{KL} = x \log(\frac{x}{a}) + (1-x) \log(\frac{1-x}{1-a}) \tag{17}$$

Its first and second derivative reads

$$L'_{KL} = \log \frac{x}{a} - \log \frac{1-x}{1-a}, \qquad L''_{KL} = \frac{1}{x} + \frac{1}{1-x} > 0. \tag{18}$$

Thus L_{KL} has a minimum when $x = a$, there its value is zero, the second derivative is positive, so it is in fact a minimum.

3 Learning methods

Once we know the derivatives, we may start to find the minimum of the loss function (if it is the goal). There are different methods to find the minimum of a multivariate function.

3.1 Steepest descent

Let us assume that we want to find the minimum of $f(x)$ where $x \in \mathbf{R}^n$. Let us assume that we are at the point x_n where the function takes the value $f_n = f(x_n)$. Now try to proceed to $x_{n+1} = x_n + dx$ where $f_{n+1} < f_n$. We write

$$f_{n+1} = f(x_n + dx) = f_n + dx_i \partial_i f_n + \dots < f_n, \tag{19}$$

if we can ensure that $dx_i \partial_i f_n < 0$. This can be done by choosing

$$dx_i = -\alpha \partial_i f_n, \tag{20}$$

because then

$$dx_i \partial_i f_n = -\alpha |\partial f_n|^2 < 0. \tag{21}$$

Therefore is we follow the negative gradient we always decrease the function value (assuming that we are in the regime where the linear approximation is good).

3.2 Conjugate gradient method

Te drawback of the steepest descent method is that, if the function has a long narrow valley, then it makes a zigzag motion. To cure this behavior we do not follow the gradient directly, but we add a direction that is orthogonal to that.

Let us assume that we are close enough to the minimum that a quadratic approximation is satisfactory:

$$f(x) = f_0 - f_i^{(1)} x_i + \frac{1}{2} f_{ij}^{(2)} x_i x_j + \dots. \tag{22}$$

If this function possesses a minimum, then the matrix $f^{(2)}$ is positive definite. At the minimum, denoted by x^*, we have

$$0 = -\partial f(x^*) = f^{(1)} - f^{(2)} x^*. \tag{23}$$

We will construct a method that runs through $x_0 \to x_1 \to \dots$ which finally converges to x^*. The basic idea is to realize that we compute the gradient at each step, so we have an access to an n dimensional subset after the nth iteration (provided the gradients are linearly independent).

Denote the negative gradient after the nth step by

$$r_n = f^{(1)} - f^{(2)} x_n. \tag{24}$$

We prepare from $\{r_0, \ldots, r_n\}$ vectors a new set $\{p_0, \ldots, p_n\}$ that is $f^{(2)}$ orthogonal, i.e.

$$p_k f^{(2)} p_\ell \sim \delta_{k\ell}. \tag{25}$$

We start from $p_0 = r_0$ and build the appropriate set recursively. Let us assume that for $k \leq n$ the p_k vectors are $f^{(2)}$ orthogonal. Then given the vector r_{n+1} we should create p_{n+1} that is $f^{(2)}$ orthogonal to the previous ones:

$$p_{n+1} = r_{n+1} - \sum_{\ell=0}^{n} \frac{p_\ell (p_\ell f^{(2)} r_{n+1})}{p_\ell f^{(2)} p_\ell}. \tag{26}$$

It is easy to see that $p_k f^{(2)} p_{n+1} = 0$ for $k \leq n$.

Now we can construct the x series. We start from some starting x_0 value, compute r_0, and determine x_1 that is at the minimum of the one dimensional subset signed out by r_0. Continuing in the similar way, in the $n$the step we have the gradient vectors $r_0, \ldots r_n$, or the reorganized $p_0, \ldots, p_n$ vectors, and we want to compute x_{n+1}, where the gradient is orthogonal to the subset spanned by the p vectors.

Although in principle x_{n+1} depends on all p vectors, we try the following Ansatz:

$$x_{n+1} = x_n + \alpha_n p_n. \tag{27}$$

The negative gradient at this point is

$$r_{n+1} = f^{(1)} - f^{(2)} x_{n+1} = r_n - \alpha_n f^{(2)} p_n. \tag{28}$$

We want that this vector is orthogonal to all basis vectors $p_0, \ldots, p_n$, provided that $r_n p_k = 0$ for $k < n$. For $k < n$ this automatically true, because

$$k < n: \qquad p_k r_{n+1} = p_k r_n - \alpha_n p_k f^{(2)} p_n = 0 \tag{29}$$

by our hypothesis and the $f^{(2)}$ orthogonality of the p vectors. For $k = n$ we have

$$p_n r_{n+1} = p_n r_n - \alpha_n p_n f^{(2)} p_n \stackrel{!}{=} 0 \quad \Rightarrow \quad \alpha_n = \frac{p_n r_n}{p_n f^{(2)} p_n}. \tag{30}$$

Now we have $p_k r_{n+1} = 0$, meaning that for the linear combinations of p_k we also have zero result. This means that $r_k r_{n+1} = 0$ for $k \leq n$. The r vectors, therefore, form an orthogonal set.

This leads to other simplifications. Reorganizing (28) we find $f^{(2)} p_k = (r_k - r_{k+1})/\alpha_k$ we have for $k < n$

$$k < n: \qquad p_k f^{(2)} r_n = \frac{(r_k - r_{k+1}) r_n}{\alpha_k} = -\frac{r_n^2}{\alpha_{n-1}} \delta_{k,n-1}. \tag{31}$$

Therefore in the expression of (26) only the last term remains:

$$p_n = r_n + \frac{r_n^2}{\alpha_{n-1} p_{n-1} f^{(2)} p_{n-1}} p_{n-1} = r_n + \frac{r_n^2}{p_n r_n} p_{n-1}. \tag{32}$$

From this equation we also see that $p_n r_n = r_n^2$, so we have finally

$$p_n = r_n + \frac{r_n^2}{r_{n-1}^2} p_{n-1}. \tag{33}$$

This means that we do not have to store all the p_k vectors, it is enough to keep only the last one! This observation makes the conjugate gradient method so effective.

Thus we have the algorithm: start at $n = 0$ from $x = x_0$ arbitrary point, and $p_0 = r_0 = f^{(1)} - f^{(2)} x_0$, then repeat

$$\begin{aligned} &\alpha_n = \frac{r_n^2}{p_n f^{(2)} p_n} \quad \Rightarrow \quad x_{n+1} = x_n + \alpha_n p_n, \qquad r_{n+1} = f^{(1)} - f^{(2)} x_{n+1} = r_n - \alpha_n f^{(2)} p_n \\ &\beta_n = \frac{r_{n+1}^2}{r_n^2} \quad \Rightarrow \quad p_{n+1} = r_{n+1} + \beta_n p_n. \end{aligned} \tag{34}$$

To generalize this method to nonlinear systems we may observe that x_{n+1} is the minimum of the function in the direction p_n. In fact if we restric ourselves to $x = x_n + \alpha p_n$, then

$$0 = \frac{\partial}{\partial\alpha} f(x) = \frac{\partial}{\partial\alpha}\left[\frac{1}{2}(x_n+\alpha p_n) f^{(2)}(x_n+\alpha p_n) - f^{(1)}(x_n+\alpha p_n)\right] = p_n(f^{(2)}x_n - f^{(1)}) + \alpha p_n f^{(2)} p_n, \tag{35}$$

thus $\alpha = p_n r_n/(p_n f^{(2)} p_n)$ as before.

Algorithm for nonlinear systems:

1. start from x_0 and $r_0 = p_0 = -\nabla f(x_0)$, the repeat steps 2-3 until convergence
2. choose $\alpha_* = \arg\min_\alpha f(x_n + \alpha p_n)$ and $x_{n+1} = x_n + \alpha_* p_n$
3. update $r_{n+1} = -\nabla f(x_{n+1})$ and $p_{n+1} = r_{n+1} + \beta_n p_n$.

The coefficient β_n can be choosen as above, but slight modifications make the algorithm work better in a way that they fall back to the steepest descent method in case of numerical instability:

$$\beta_n^{FR} = \frac{r_{n+1}^2}{r_n^2}, \qquad \beta_n^{PR} = \frac{r_{n+1}(r_{n+1}-r_n)}{r_n^2}, \qquad \beta_n^{HS} = -\frac{r_{n+1}(r_{n+1}-r_n)}{p_n(r_{n+1}-r_n)}, \qquad \beta_n^{DY} = -\frac{r_{n+1}r_{n+1}}{p_n(r_{n+1}-r_n)}. \tag{36}$$

(FR: Fletcher-Reeves, PR: Polak-Ribierère, HS: Hestenes-Stiefel, DY: Dai-Yuan, cf. wikipedia "Nonlinear_conjugate_gradient_method").

The one dimensional minimization mentioned at step 2 above means an update $x' = x + \varepsilon p$ that

$$f(x+\varepsilon p) = f(x) + \varepsilon p \nabla f(x) < f(x) \quad \Rightarrow \quad \varepsilon \sim -p\nabla f(x), \tag{37}$$

like in the steepest descent method, but here we should consider direction p instead of direction ∇f.

The numerical implementation is the simplest in the FR or DY case. In this latter case we store p_{n-1} and $z_{n-1} = p_{n-1} r_{n-1}$. We obtain as input x_n and r_n.

1. update $p_n = r_n + \dfrac{r_n^2}{z_{n-1} - r_n p_{n-1}} p_{n-1}$. We may decide also $p_n = p_{n-1}$. If $n = 0$ then $p_0 = r_0$.
2. update $x_{n+1} = x_n + \alpha\,\mathrm{sgn}(p_n r_n) p_n$, where α here is the learning rate.

3.3 ADAM method

As we have seen, it is not always the best method to go towards the negative gradient computed in the present place, earlier gradient values carry a lot of information about the geometry of the surface that is worth to take into account. In the conjugate gradient method we use an average of the present gradient and the earlier ones.

Another observation that is an important factor when we train neural networks is that we usually compute gradients on a subset of all the inputs (batches). Therefore the value of the gradient somewhat depends on the actual batch, so it is a stochastic variable. Its actual value therefore does not reflect the global geometry, the local effects can be rather strong.

Another important effect is that the magnitude of the gradient depends strongly on the way we collect them. There may be situations when the gradient is not sensible, it is too small to give a good result. This will not modify the overall direction of the collected gradiants, but it magnitude may be too small (or eventually too large). Therefore we should use a normalized gradient to make learning effective.

One of the best method that adopts these thoughts is ADAM, where the name stands for ADAptive Moment method (arXiv: 1412.6980). It uses an exponentially averaged gradient and exponentially averaged gradient (pointlike) square:

$$\begin{aligned} g_n &= \beta_1 g_{n-1} + (1-\beta_1)\nabla f_n \\ v_n &= \beta_2 v_{n-1} + (1-\beta_2)\nabla f_n^2, \end{aligned} \tag{38}$$

starting at $g_0 = v_0 = 0$. Resoving the recursion, after the nth step

$$g_n = (1-\beta_1)\sum_{m=1}^{n} \beta_1^{n-m} \nabla f_m, \tag{39}$$

so the influence of the past gradients decays exponentially. If the gradient is constant, then

$$g_n = (1 - \beta_1^n)\nabla f. \tag{40}$$

In order to have an unbiased average, we have to divide the result by $(1 - \beta_1^n)$; the same situation concerns v_n. So we have the corrected values

$$\hat{g}_n = \frac{g_n}{1 - \beta_1^n}, \qquad \hat{v}_n = \frac{v_n}{1 - \beta_2^n}. \tag{41}$$

Finally the learning step is

$$x_{n+1} = x_n - \alpha \frac{\hat{g}_n}{\sqrt{\hat{v}_n} + \varepsilon}, \tag{42}$$

where ε is a regularization constant.

The proposed values are

$$\beta_1 = 0.9, \qquad \beta_2 = 0.999, \qquad \alpha = 0.001, \qquad \varepsilon = 10^{-8}. \tag{43}$$

This means that, especially in the beginning, without the correction term we would seriously misidentify the average gradient value.