Basic MLP with manually-derived Backprop

Teo Asinari

October 14, 2023

1 Introduction

Goal: To design, train and use a simple 3-layer MLP for binary classification of size-2 vectors.

Design: of the form

$$[(layer\_size, Activation)...]$$

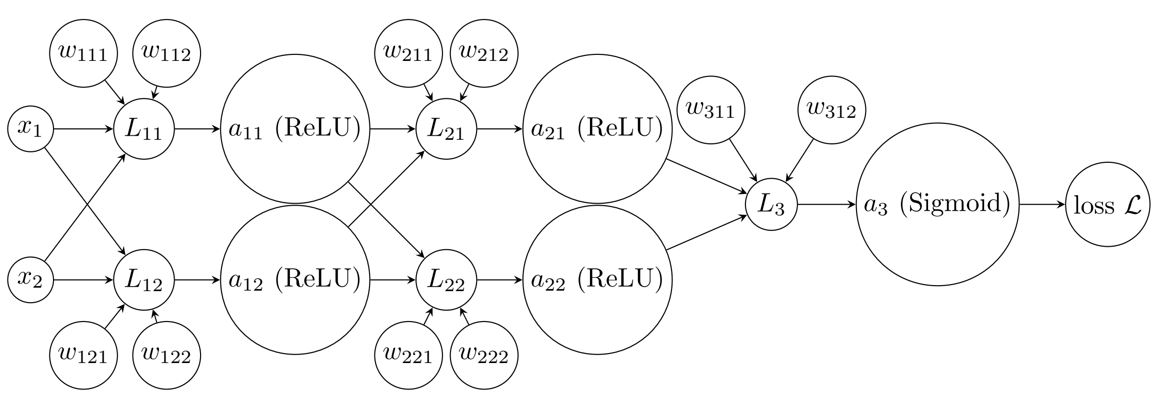
: [(2, ReLU), (2, ReLU), (1, Sigmoid)]

1.1 Diagrams

1.1.1 Vectorized Diagram (Equiv to Roger Grosse' 'Computational Graph')

$$\begin{array}{ccccccccccccccc} & \boldsymbol{w_1} & & & & \boldsymbol{w_2} & & & & \boldsymbol{w_3} & & & & \boldsymbol{t} & \\ & \searrow & & & & \searrow & & & & \searrow & & & & \searrow & \\ \boldsymbol{x} & \longrightarrow & \boldsymbol{L_1} & \rightarrow & \boldsymbol{a_1} & \rightarrow & \boldsymbol{L_2} & \rightarrow & \boldsymbol{a_2} & \rightarrow & \boldsymbol{L_3} & \rightarrow & \boldsymbol{a_3} & \longrightarrow & \mathcal{L} \end{array}$$

1.1.2 Expanded Diagram (Equiv. to Roger Grosse' 'Network Architecture')



1.2 Definitions

1.2.1 Remark on weight notation

$w_{i,j,k}$ is to say the weight at the i-th layer, j-th neuron, k-th weight. Hence w_{111} is the first weight of the first neuron in the first layer, etc.

1.2.2 Remark on layer notation

This is a sub-case of the weight notation. I.e., L_{ij} is the scalar value of the j-th neuron at the i-th layer, etc.

1.2.3 Neuron firing calculation

This is just a straightforward dot-product. We have:

$$L_{ij} = \boldsymbol{w}_{ij}\boldsymbol{x}_i$$

Where $\boldsymbol{x}_i$ in this case is referring to a more general notion of 'layer input', not necessarily just the first input to the network as in the diagrams above.

1.3 BackPropagation Derivation

Notation for derivative of loss w.r.t. to a function I will be using the following: $\overline{f} = \frac{\partial \mathcal{L}}{\partial f}$. This notation was introduced by Roger Grosse from the University of Toronto.

Pa(x) and Ch(x) these refer to the sets of parent and child vertices of a vertex in a graph.

General Approach Let's label the computational graph nodes as $v_1, ..., v_N$ with some topological ordering. Then, our general goal for backprop is to compute $\overline{v_i}$ for $i \in 1, ...N$. With these, we can trivially calculate the weight updates. We compute a forward pass of the network, then set $v_N = 1$, then, for $i = N-1, ..., 1$, we have:

$$\overline{v_i} = \sum_{j \in \mathrm{Ch}(v_i)} \overline{v_j} \frac{\partial v_j}{\partial v_i} \quad \text{(The Backprop Rule)} \tag{1}$$

1.3.1 Applying the backprop rule

Loss and final activation Then, going backwards through the 'computational graph', starting at the end:

$$\overline{\mathcal{L}} = 1 \tag{2}$$

$$\overline{a_3} = \overline{\mathcal{L}} \frac{\partial \mathcal{L}}{\partial a_3}$$

$$\overline{a_3} = (1)\frac{\partial \mathcal{L}}{\partial a_3}$$

$$\overline{a_3} = \frac{\partial \mathcal{L}}{\partial a_3}$$

$$\overline{a_3} = \frac{\partial}{\partial a_3}\frac{1}{2}(a_3 - t)^2$$

$$\overline{a_3} = (a_3 - t)\frac{\partial}{\partial a_3}(a_3 - t)$$

$$\overline{a_3} = (a_3 - t)(1)$$

$$\overline{a_3} = (a_3 - t) \tag{3}$$

Final layer ***N.B.*** I use σ to denote the sigmoid function here, not an activation function.

$$\overline{L_3} = \overline{a_3}\frac{\partial a_3}{\partial L_3}$$

$$\overline{L_3} = \overline{a_3}\frac{\partial}{\partial L_3}\sigma(L_3)$$

$$\overline{L_3} = \overline{a_3}\ \sigma(L_3)(1 - \sigma(L_3)) \tag{4}$$

Final layer weights

$$\overline{w_{31i}} = \overline{L_3}\frac{\partial}{\partial w_{31i}}L_3$$

$$\overline{w_{31i}} = \overline{L_3}\frac{\partial}{\partial w_{31i}}\sum_j w_{31j}a_{2j}$$

$$\overline{w_{31i}} = \overline{L_3}a_{2i} \tag{5}$$

Second layer activation

$$\overline{a_{2i}} = \overline{L_3}\frac{\partial}{\partial a_{2i}}L_3$$

$$\overline{a_{2i}} = \overline{L_3}\frac{\partial}{\partial a_{2i}}\sum_j w_{31j}a_{2j}$$

$$\overline{a_{2i}} = \overline{L_3}w_{31i} \tag{6}$$

Second layer

$$\overline{L_{2i}} = \overline{a_{2i}} \frac{\partial}{\partial L_{2i}} a_{2i}$$

$$\overline{L_{2i}} = \overline{a_{2i}} \frac{\partial}{\partial L_{2i}} \mathrm{ReLU}(L_{2i})$$

Note that $d/dx(\mathrm{ReLU}(x))$ is the heaviside step function $\theta(x)$:

$$\begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{aligned} \overline{L_{2i}} &= \overline{a_{2i}} \quad \theta(L_{2i}) \cdot (1) \\ \overline{L_{2i}} &= \overline{a_{2i}} \quad \theta(L_{2i}) \end{aligned} \tag{7}$$

Second layer weights

$$\overline{w_{2ij}} = \overline{L_{2i}} \frac{\partial}{\partial w_{2ij}} L_{2i}$$

$$\overline{w_{2ij}} = \overline{L_{2i}} \frac{\partial}{\partial w_{2ij}} \sum_k w_{2ik} a_{1k}$$

$$\overline{w_{2i}} = \overline{L_{2i}} a_{1j} \tag{8}$$

Input layer activation

$$\overline{a_{1i}} = \overline{\boldsymbol{L}_2} \frac{\partial}{\partial a_{1i}} \boldsymbol{L}_2$$

$$\overline{a_{1i}} = \overline{\boldsymbol{L}_2} \frac{\partial}{\partial a_{1i}} \sum_j w_{2j} a_{1j}$$

$$\overline{\boldsymbol{a}_{1i}} = \overline{\boldsymbol{L}_2} w_{2i} \tag{9}$$

Input layer

$$\overline{\boldsymbol{L}_1} = \overline{\boldsymbol{a}_1} \frac{\partial}{\partial \boldsymbol{L}_1} \boldsymbol{a}_1$$

$$\overline{\boldsymbol{L}_1} = \overline{\boldsymbol{a}_1} \frac{\partial}{\partial \boldsymbol{L}_1} \mathrm{ReLU}(\boldsymbol{L}_1)$$

$$\begin{aligned} \overline{\boldsymbol{L}_1} &= \overline{\boldsymbol{a}_1} \; \theta(\boldsymbol{L}_1) \cdot (1) \\ \overline{\boldsymbol{L}_1} &= \overline{\boldsymbol{a}_1} \; \theta(\boldsymbol{L}_1) \end{aligned} \tag{10}$$

Input layer weights

$$\overline{w_{1i}} = \overline{\boldsymbol{L}_1} \frac{\partial}{\partial w_{1i}} \boldsymbol{L}_1$$

$$\overline{w_{1i}} = \overline{\boldsymbol{L}_1} \frac{\partial}{\partial w_{1i}} \sum_j w_{1j} \boldsymbol{x}_{1j}$$

$$\overline{w_{1i}} = \overline{\boldsymbol{L}_1} x_{1i} \tag{11}$$

N otes to self: Grosse follows a per-element approach first, then somehow transformed those results into a vectorized form involving (in some cases) re-arranged multiplications and matrix transpose. I am somehwat confused/overwhelmed by this.

1.4 Misc. Remarks

1.4.1 Rounding: Training vs Inference

Since we aim to train a binary classifier, the round() would be necessary for the correct output range. However since round() is not differentiable, we omit it during training, calculating fractional losses instead. We only include round() during inference.