Practical-1: Pen and paper exercises

Instructions

- Answer the questions in the following exercises thoroughly. However, this does not necessarily mean long answers. Be precise and compact.
- We highly recommend that your solutions are written in LaTeX and must have a easy-to-follow format (e.g. sectionings/subsectionings etc.).
- This exercise belongs to Practical-1, therefore, please include your solution in your main submission file for which the instructions are given separately. (Do not submit individually.)
- For *this* assignment, use the name below: `paper_assignment_1_solved.pdf`
- The bonus points are only valid for those submissions with a grade lower than 40.

Exercise-1 (15 *pts*)

In Figure 1, you are given a 2-layer feedforward neural network. Its variables are given as follows:

- $s_1 = W_1 \cdot x_{in}$
- $z_1 = f_1(s_1) = f_1(W_1 \cdot x_{in})$
- $s_2 = W_2 \cdot z_1$
- $z_2 = f_2(s_2) = f_2(W_2 \cdot f_1(W_1 \cdot x_{in}))$
- $s_{out} = W_{out} \cdot z_2$
- $z_{out} = f_3(s_{out}) = f_3(W_{out} \cdot f_2(W_2 \cdot f_1(W_1 \cdot x_{in}))) = y_{out}$

where f_i, $i \in \{1, 2, 3\}$, denotes any differentiable function (e.g. sigmoid, tanh, relu, etc.) Using these compute (generalized) derivatives of the loss function, $\mathcal{L} = 0.5 \cdot (y_{out} - y_{gt})^2$, with respect to the weights, $\frac{\partial \mathcal{L}}{\partial W_1}, \frac{\partial \mathcal{L}}{\partial W_2}, \frac{\partial \mathcal{L}}{\partial W_{out}}$.

Hint: The final solution for $\frac{\partial \mathcal{L}}{\partial W_1}$ in the form: $(y_{out} - y_{gt}) \cdot f_3^{'}(s_{out}) \cdot z_2$. Show

typo?

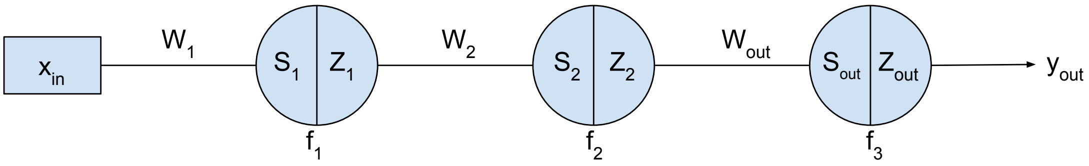


Figure 1: A network with 3 weights

your work to get to this result and also solve for the latter gradients with the given form in mind. (*Note:* In this exercise, the weights are some real numbers not matrices.)

Prelude

This exercise has shown that a neural network is nothing but a *composite function* in the form of $f_N \circ f_{N-1} \circ \dots \circ f_1(x) = f_N(f_{N-1}(\dots(f_1(x))))$. From calculus, we are familiar with how to take derivatives (in machine learning, these are usually referred to as *gradients*) of such functions. Namely, we call it *chain rule*.

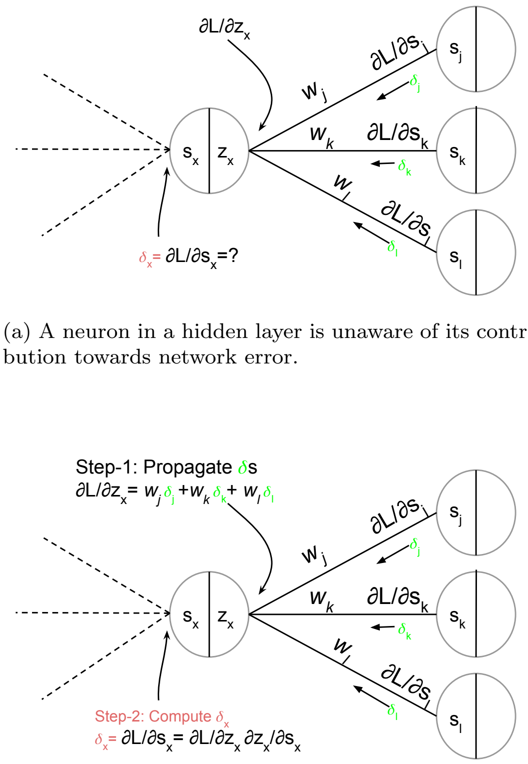
By the end of the Exercise-1 you realized that computing these derivatives this way becomes intractable to solve (at least by hand) as we increase the network dimensions in both width and depth. However, you have seen a pattern emerging in your solutions. In order to have an efficient update mechanism, we can exploit this recursive pattern.

What this recursive pattern tells us is that updating the parameters of a network boils down to determining the rates at which the error, or loss, varies with small perturbations to unit inputs (i.e. s_j). Mathematically, this is equivalent to computing $\frac{\partial \mathcal{L}}{\partial s_j}$. For an output unit this is straightforward. On the other hand, for units in inner layers, we do not have an immediate error signal. We just know how much influence (good or bad) a unit has over the overall error by looking at the weighted edges that branches out from it. It turns out that we can propagate (distribute) the error towards the hidden units by following the edges from top to bottom. The procedure (error propagation) is summarized in Figure 2. We can further formulate it using chain rule as follows:

$$
\begin{aligned}
\delta_N &= \frac{\partial \mathcal{L}}{\partial s_N} \\
\delta_{N-1} &= \frac{\partial \mathcal{L}}{\partial s_{N-1}} = \delta_N \cdot W_N \cdot \frac{\partial f(s_{N-1})}{\partial s_{N-1}} \\
\delta_{N-2} &= \frac{\partial \mathcal{L}}{\partial s_{N-2}} = \delta_{N-1} \cdot W_{N-1} \cdot \frac{\partial f(s_{N-2})}{\partial s_{N-2}} \\
&\vdots \\
\delta_1 &= \frac{\partial \mathcal{L}}{\partial s_1} = \delta_2 \cdot W_2 \cdot \frac{\partial f(s_1)}{\partial s_1}
\end{aligned}
$$

Once the backward pass computes the δ_js, we can determine ΔW_ks

(i) Write down $\Delta W_k = \frac{\partial \mathcal{L}}{\partial W_k}$ in terms of δ_k. **in the previous setup?**



(a) A neuron in a hidden layer is unaware of its contribution towards network error.

(b) The errors from consecutive layer is propped to the neuron in the hidden layer.

Figure 2: Propagation of error from one layer to another

Exercise-2 (15 *pts*)

In this exercise you are going to execute forward and backward modes of a neural network on paper for 3 iterations. We illustrate our model in Figure 3. This network comprises of a hidden layer with 3 ReLU units and a squared-error loss as in the first exercise. (*Note*: Please use *tanh* as activation function in the output unit.)

$$ReLU(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

(*Hint:* Notice that this function is *non-differentiable* around 0. Therefore, when computing your gradients, you will have to compute one gradient if the input is larger than 0, another if it is smaller than 0, and ignore when the input is 0 and thus non-differentiable.)

Step-1: Initialize the weights to these random values as shown here.

$$\begin{bmatrix} W_{11} & W_{21} & W_{31} \\ W_{12} & W_{22} & W_{32} \end{bmatrix} = \begin{bmatrix} 0.60 & 0.70 & 0.00 \\ 0.01 & 0.43 & 0.88 \end{bmatrix}$$

and

$$\begin{bmatrix} w_{11} \\ w_{12} \\ w_{13} \end{bmatrix} = \begin{bmatrix} 0.02 \\ 0.03 \\ 0.09 \end{bmatrix}$$

The data is provided in the following matrix where samples are stored in rows.

$$\begin{bmatrix} x_1^T \\ x_2^T \\ x_3^T \\ x_4^T \end{bmatrix} = \begin{bmatrix} 0.75 & 0.80 \\ 0.20 & 0.05 \\ -0.75 & 0.80 \\ 0.20 & -0.05 \end{bmatrix}$$

and their corresponding labels are $y = [1, 1, -1, -1]^T$.

Step-2: Exploit the recursive patterns you discovered in the first exercise to update parameters of this network.

Pseudo-code:

1. Forward the input and record (i) input to the unit, s_j, and (ii) output of the unit, z_j, for each unit j in all layers.
2. Compute the loss and record it.

   Use $\mathcal{L} = 0.5 \cdot (y_{out} - y)^2$
3. Compute the error signal, $\delta_{out} = \frac{\partial \mathcal{L}}{\partial s_{out}}$ at the output.
4. Propagate δ_{out} backwards to compute $\delta_j = \frac{\partial \mathcal{L}}{\partial s_j}$ at hidden units.

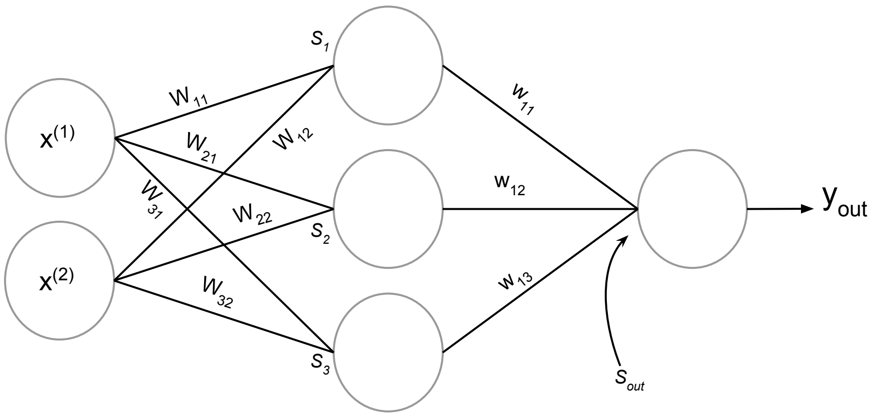


Figure 3: A network with one hidden layer

5. Compute gradients (i.e. $\Delta W, \Delta w$) using these and update weights using gradient descent.

   Hint: $W^{(t+1)} = W^{(t)} - \alpha \cdot \Delta W$

6. Repeat 3 times.

Exercise-3 (10 *pts*)

Although a *softmax* layer is usually coupled with a *cross-entropy* loss, this is not necessary and you can use a different loss function. In this exercise, you are going to couple softmax layer with multiclass *hinge* loss and derive the gradient solution with respect to the inputs (to softmax).

Let us set up our problem now. Assume we want to classify points, $x_i \in \mathbb{R}^{d\times 1}$, sampled from a distribution with k (where $k > 2$) classes. Hence, our network architecture boils down to a k-way softmax layer before the loss. Let us denote the inputs to this layer with o_j where $j \in \{1, \ldots, k\}$. Let us also denote the output of softmax layer with p_j where $j \in \{1, \ldots, k\}$. Softmax layer applies the following transformation elementwise:

$$\bar{P} = \begin{bmatrix} \exp(o_1) \\ \vdots \\ \exp(o_k) \end{bmatrix} \cdot \left(\sum_{j=1}^{k} \exp(o_j) \right)^{-1} = \begin{bmatrix} p_1 \\ \vdots \\ p_k \end{bmatrix}, \text{ where } p_i = \frac{\exp(o_i)}{\sum\limits_{j} \exp(o_j)}$$

(*Hint:* Notice that p_i depends on all other dimensions $j = \{1, ..., k\}$. Hence the derivative of the softmax layer, i.e. $\frac{\partial \bar{P}}{\partial \bar{O}}$, implies a Jacobian matrix of $k \times k$.) Then, our multiclass hinge loss takes $\bar{P}$ as argument to compute the following:

$$\mathcal{L}^{(i)}_{hinge} = \sum_{j \neq y_i} \max(0, p_j - p_{y_i} + margin)$$

where $\mathcal{L}_{hinge}^{(i)}$ is the loss resulting from ith point in our training set and y_i is the groundtruth value for it.

(i) Explain, in (your own) words, what this particular loss function is designed to achieve.

(ii) Derive the gradient for $\frac{\partial \mathcal{L}_{hinge}}{do_j}$.

(Bonus) Exercise-4 (5 *pts*)

As you have been told deep learning 'breakthrough' was mainly thanks to developments in hardware technology (e.g. graphics processing units (GPU)). However, even the compute power offered by the most advanced GPUs are finite. For that very reason, the network architectures one can think of are still bounded by hardware limits (e.g. memory, number of core on your GPU etc.). Should you happen to have designed a very complicated model with billions of parameters you are likely to encounter memory problems during training. What would you do in such a case where you either cannot afford a more expensive, say a device with twice the memory, compute device or there exist no better hardware solution than what you have now?

Consider the backpropagation procedure carefully. Can you identify parts which you think may be redundant in terms of memory efficiency? If yes, how would you reengineer the part(s) so that your model can be fit into your hardware? List your solutions and explain them compactly. (*Tip:* Consider the memory-speed tradeoff while coming up with your solution. That is, you can compromise from speed to reduce memory requirements.)