50.039 – Theory and Practice of Deep Learning

Alex

Week 04: Backpropagation

[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources.]

1 The infamous backpropagation

The learning goal of this lecture is to see

Key takeaways:

- Backpropagation is an algorithm to compute derivatives in a neural network (w.r.t. network parameters, and inputs)
- Backpropagation inside is chain rule applied top-down along the graph structure of the network
- the backprop comes with an efficient mechanism to reuse computed partial derivatives for computing new derivatives further down in the network. These two points are the key to know about chain rule.
- if you understand how to apply chain rule for computing a derivative, then backprop is easy
- you should be able to derive the derivative of a loss of a neural network as a sum-product of derivatives of single neurons with respect to their inputs and parameters
- in this lecture we try to keep the structure of chain rule separated from how the derivative of a neuron looks like
- the equations derived with backprop show one central challenge in training neural networks: vanishing and exploding gradients

1.1 Recap: example loss

Definition: cross-entropy loss for C-class classification

Suppose we have for every sample x_i a function $p(x_i) = (p_1(x_i), p_2(x_i), \dots p_C(x_i))$ which provides a probability $p_c(x_i) = P(Y = c|X = x_i)$.
Then the crossentropy loss for a sample (x_i, y_i) is given as

$$\begin{aligned} L(p(x_i), y_i) &= \Big\{ -\log(p_c(x_i)) \quad \text{if } y_{i,c} = +1 \\ &= \sum_{c=1}^{C} -y_{i,c} \log(p_c(x_i)) \end{aligned}$$

Thus, in a multiclass setting, for a sample $(x_i, y_i) = (x_i, (y_{i,1}, \dots, y_{i,C}))$, the cross-entropy loss is just the neg log of the predicted probability for the ground truth class: $-\log(p_c(x_i))$ if $y_{i,c} = 1$.

1.2 without backprop: discrete gradient

We have a prediction function $f_w(x)$ and a loss function $L(f_w(x_i), y_i)$ The objective to minimize will be

$$LL(w) = \sum_{(x_i,y_i) \in \text{minibatch}} L(f_w(x_i), y_i)$$

Suppose such that we can set values for w and x and observe the value of $R(w)$.

How to compute gradients? The Gradient is a vector of partial derivatives:

$$\nabla LL(w) = \big(\tfrac{\partial LL}{\partial w_1}(w), \tfrac{\partial LL}{\partial w_2}(w), \dots, \tfrac{\partial LL}{\partial w_D}(w)\big)$$

How to compute the partial derivative $\frac{\partial LL}{\partial w_d}(w)$ for one element w_d of the parameter vector w ? One can use perturbations.

1.3 A recap on gradients

Lets recapitulate the gradient here.

Consider two functions that can be concatenated:

$$\begin{aligned} f(a) &= f(a_1, \dots, a_R) \\ g(b) &= (g_1(b), \dots, g_R(b)) \\ g_i(b) &= g_i(b_1, \dots, b_l), i = 1, \dots, R \end{aligned}$$

First of all the gradient is the vector of all partial derivatives. Each partial derivative depends on the input point a: $\frac{\partial f}{\partial a_1}(a)$.

$$\nabla f(a) = (\frac{\partial f}{\partial a_1}(a), \ldots, \frac{\partial f}{\partial a_R}(a))$$

Secondly the gradient $\nabla f(a)$ as a vector defines a linear mapping $Df(a)[\cdot]$ into tangent space

$$h \mapsto Df(a)[h] = \nabla f(a) \cdot h = \sum_{r=1}^{R} \frac{\partial f}{\partial a_r}(a) h_r$$

This gives the value of the derivative of function f in input point $a = (a_1, \ldots, a_R)$, when looking into the direction of vector $h = (h_1, \ldots, h_R)$, that is a measure of local increase of function value of f when starting in a and going into direction h.
Thus the gradient $\nabla f(a)$ is a vector, which captures the local increase of function f in point a in all possible directions h by this mapping.

It should be further noted that inner products correspond to matrix multiplication of vectors

$$\begin{aligned} v \cdot w &= \sum_{r=1}^{R} v_r w_r \\ & v.size() = (R),\ w.size() = (R) \\ \leftrightarrow & v.size() = (1, R),\ w.size() = (R, 1) \\ = & Matmul(v, w) \text{ as reshaped} \end{aligned}$$

1.4 chain rule for simple concatenations

Lets recapitulate the chain rule here.

Lets go back to: Consider two functions that can be concatenated:

$$\begin{aligned} f(a) &= f(a_1, \ldots, a_R) \\ g(b) &= (g_1(b), \ldots, g_R(b)) \\ g_i(b) &= g_i(b_1, \ldots, b_l), i = 1, \ldots, R \end{aligned}$$

In order to compute the concat $f \circ g(b) = f(g(b))$, g must have as many output components, as f has inputs, here R.

The chain rule states in the sense of partial derivatives:

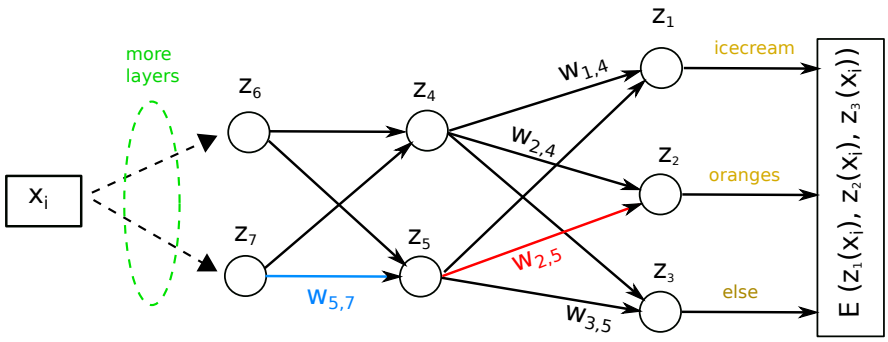
$$\frac{\partial(f \circ g)}{\partial b_k}(b) = \nabla f(g(b)) \cdot \frac{\partial(g_1, \ldots, g_R)}{\partial b_k}(b), \quad \cdot \text{ is inner product of vecs}$$
$$= \sum_{r=1}^{R} \frac{\partial f}{\partial a_r}(g(b)) \frac{\partial g_r}{\partial b_k}(b)$$

You can write it in a shorter form, but then you need to remember what function values to plug in for every derivative:

$$\frac{\partial(f \circ g)}{\partial b_k} = \nabla f \cdot \frac{\partial(g_1, \ldots, g_R)}{\partial b_k}$$
$$= \sum_{r=1}^{R} \frac{\partial f}{\partial a_r} \frac{\partial g_r}{\partial b_k}$$

1.5 Backprop

have now a loss function for the three neuron outputs z_1, z_2, z_3. How to compute the derivatives?
First observation: you can compute the derivative for every sample separately, as gradients and sums commute:



Key insight: the graph tells you

- which function receives input from other functions, e.g. z_4 takes inputs from z_6 and z_7
- which function provides input to other functions: z_5 provides inputs to z_1, z_2, z_3

Chain-rule derivation for neural network topologies.

Backpropagation

1. Start at the top by finding the set of neurons z_l such that the loss function directly depends on their inputs $E = E(z_l)$
2. the graph structure of the neural net tells you which neurons z_k give input to z_l
3. Then use for walking downwards (against directions of the forward computation flow):

$$\frac{dE}{dz_k} = \sum_{l\text{: s.t. } k \text{ gives input to } l} \frac{dE}{dz_l}\frac{\partial z_l}{\partial z_k}$$

4. next: repeat steps 2. and 3. for the z_k until you reach the bottom / or until you have covered all paths backwards from E to your weight of interest w_k

5 . Finish at the bottom by $\frac{dE}{dw_r} = \frac{dE}{dz_r}\frac{\partial z_r}{\partial w_r}$

This is the way, how computers compute backpropagation. reason: usually they are interested in getting derivatives for all the weights. One can see that

$$\frac{dE}{dz_k} = \sum_{l\text{: s.t. } k \text{ gives input to } l} \frac{dE}{dz_l}\frac{\partial z_l}{\partial z_k}$$

comes from applying chainrule to:

$$E = E(z_l \text{ s.t. } k \text{ gives input to } l) = E(z_{l_1}(z_k), \ldots, z_{l_R}(z_k))$$

If you want the derivative for only one weight, then the following version is easier to use (one does not need to track all paths backwards, there is no need to go to neurons which never receive inputs from the neuron associated with w_k):

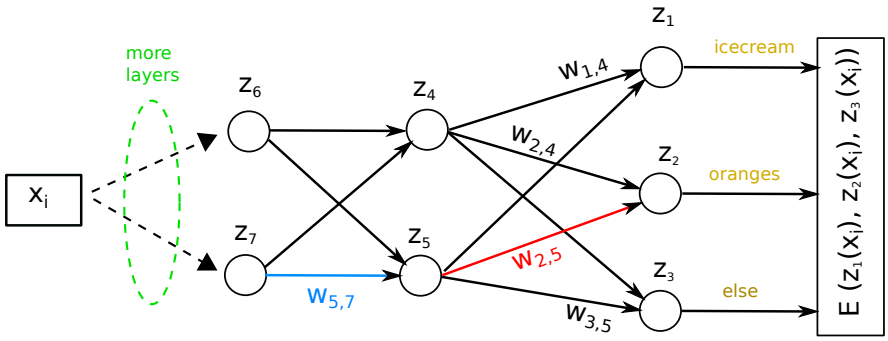
Backpropagation easier for computing by hand

1. Start at the bottom to factorize out terms $\frac{dE}{dw_k} = \frac{dE}{\partial z_k}\frac{dz_k}{\partial w_k}$
2. the graph structure of the neural net tells you which neurons z_l receive input from z_k
3. Then use for walking upwards (along directions of the forward computation flow):

$$\frac{dE}{dz_k} = \sum_{l:l \text{ receives input from } k} \frac{dE}{dz_l}\frac{\partial z_l}{\partial z_k}$$

4. next: repeat steps 2. and 3. for the z_l until you reach the top

Lets go through this in examples!



$\frac{\partial E}{\partial w_{2,5}} = ???$

- example: compute gradient with respect to parameters w_{25} of neuron z_2

$$\frac{\partial E}{\partial w_{2,5}} = \frac{\partial E}{\partial z_2}\frac{\partial z_2}{\partial w_{2,5}}$$

- important: both terms $\frac{\partial E}{\partial z_2}$ and $\frac{\partial z_2}{\partial w_{2,5}}$ easily computable in terms of NN structure

$$z_2(x_i) = g(w_{2,4}z_4(x_i) + w_{2,5}z_5(x_i))$$
$$\frac{\partial z_2}{\partial w_{2,5}} = g'(w_{2,4}z_4(x_i) + w_{2,5}z_5(x_i)) \cdot z_5(x_i)$$

- E is defined directly as a function of z_2 and z_1, z_3

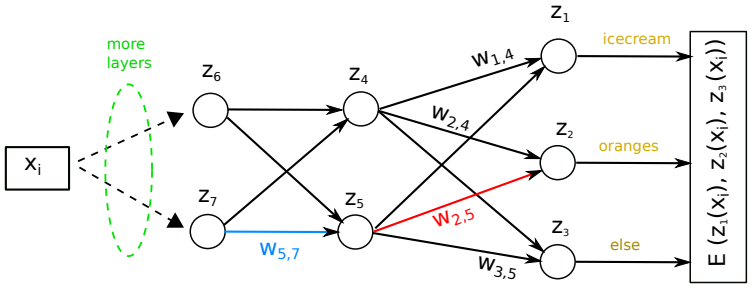
$$\frac{\partial E}{\partial z_2} = -1 \sum_{i \in data} \left(y_{2,i} - \frac{e^{z_2(x_i)}}{\sum_{c'=1}^{3} e^{z_{c'}(x_i)}} \right)$$

How about $\frac{\partial E}{\partial w_{5,7}} = ???$

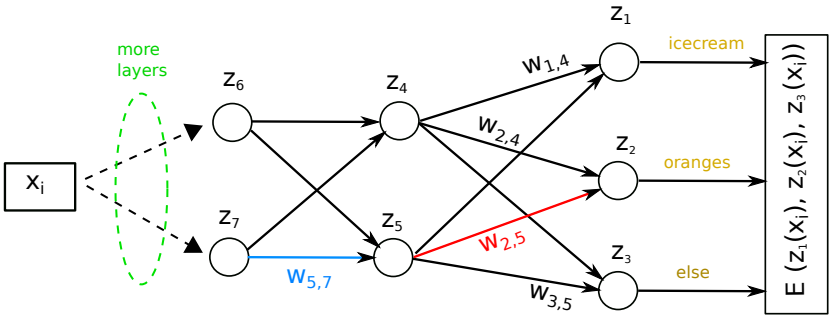
- same approach - using chain rule

$$\frac{\partial E}{\partial w_{5,7}} = \frac{\partial E}{\partial z_5} \frac{\partial z_5}{\partial w_{5,7}}$$

- $\frac{\partial E}{\partial z_5}$ not directly computable (z_5 is no input to E)



- idea: apply top-down chainrule to compute $\frac{\partial E}{\partial z_5}$ – **we know all the functions that depend on z_5 as input:** z_1, z_2, z_3. Use this knowledge:



$$\frac{\partial E}{\partial z_5} = \frac{\partial E}{\partial z_1}\frac{\partial z_1}{\partial z_5} + \frac{\partial E}{\partial z_2}\frac{\partial z_2}{\partial z_5} + \frac{\partial E}{\partial z_3}\frac{\partial z_3}{\partial z_5}$$

- know already how to compute $\frac{\partial E}{\partial z_1}, \frac{\partial E}{\partial z_2}, \frac{\partial E}{\partial z_3}$ – can be done directly by differentiating E.
- open is only $\frac{\partial z_1}{\partial z_5}, \frac{\partial z_2}{\partial z_5}, \frac{\partial z_3}{\partial z_5}$

$$z_2(x_i) = g(w_{2,4} z_4(x_i) + w_{2,5} z_5(x_i))$$
$$\frac{\partial z_2}{\partial z_5} = g'(w_{2,4} z_4(x_i) + w_{2,5} z_5(x_i)) \cdot w_{2,5}$$
very similar to
$$\frac{\partial z_2}{\partial w_{2,5}} = g'(w_{2,4} z_4(x_i) + w_{2,5} z_5(x_i)) \cdot z_5$$

Observation

- $\frac{\partial E}{\partial w_{7,5}} = \frac{\partial E}{\partial z_5} \frac{\partial z_5}{\partial w_{7,5}}$
- $\frac{\partial E}{\partial z_5}$ can be computed by top-down chain rule

$$\frac{\partial E}{\partial z_5} = \frac{\partial E}{\partial z_1} \frac{\partial z_1}{\partial z_5} + \frac{\partial E}{\partial z_2} \frac{\partial z_2}{\partial z_5} + \frac{\partial E}{\partial z_3} \frac{\partial z_3}{\partial z_5}$$

- $\frac{\partial z_2}{\partial z_5}$ is easy to compute by NN structure, similar to $\frac{\partial z_2}{\partial w_{25}}$
- (final result:)

$$\frac{\partial E}{\partial w_{5,7}} = \frac{\partial E}{\partial z_1} \frac{\partial z_1}{\partial z_5} \frac{\partial z_5}{\partial w_{5,7}} + \frac{\partial E}{\partial z_2} \frac{\partial z_2}{\partial z_5} \frac{\partial z_5}{\partial w_{5,7}} + \frac{\partial E}{\partial z_3} \frac{\partial z_3}{\partial z_5} \frac{\partial z_5}{\partial w_{5,7}}$$

- have shown: $\frac{\partial z_i}{\partial w_{i,j}}, \frac{\partial z_i}{\partial z_j}$ are simple terms
- can use chainrule to get every $\frac{\partial E}{\partial z_i}$
- there is a pattern in these computations – formalize it

1.6 In class exercise

Chain-rule derivation for neural network topologies.

...

Start at the bottom to factorize out terms $\frac{dE}{dw_k} = \frac{dE}{dz_k} \frac{\partial z_k}{\partial w_k}$

Then use for walking upwards (along directions of the forward computation flow):

$$\frac{dE}{dz_k} = \sum_{l:\ l \text{ receives input from } k} \frac{dE}{dz_l} \frac{\partial z_l}{\partial z_k}$$

1.7 Backpropagation as algorithm for layered NNs

...

for *layerindex* $l = N \ldots 1$ (decreasing order)

- if $l == N$: compute $\frac{dE}{dz_i^{(l)}}$ directly
  (error as a function of NN outputs)
- else: use precomputed $\frac{dE}{dz_j^{(>l)}}$, $z_j^{(>l)}$ are those neurons in layers $l+1, l+2, l+3, ...$ which receive directly inputs from $z_i^{(l)}$
  compute and store $\frac{dE}{dz_i^{(l)}} = \sum_{z_j^{(>l)}} \frac{dE}{dz_j^{(>l)}} \frac{\partial z_j^{(>l)}}{\partial z_i^{(l)}}$
- compute partial derivative for variable $w_{hi}^{(l)}$ as $\frac{dE}{dw_{hi}^{(l)}} = \frac{dE}{dz_i^{(l)}} \frac{\partial z_i^{(l)}}{\partial w_{hi}^{(l)}}$

end for

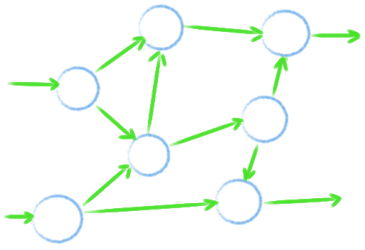
Now: assemble partial derivatives for variables w_{ij} into one huuuge gradient vector ... and descend in w-space!

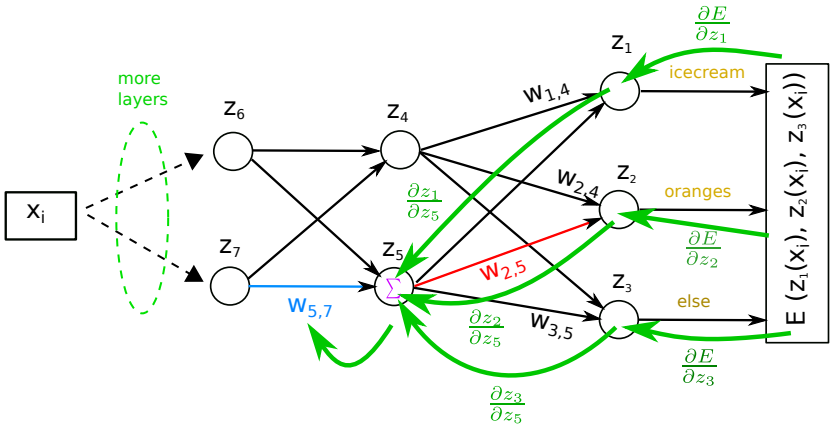
1.8 another Backprop Takeaway

- backprop: chainrule PLUS we can reuse terms:
- reuse $\frac{\partial z^{(>l)}}{\partial z^{(l)}}$ for all weights in levels below l which compute inputs to neuron $z^{(l)}$
- recycling of $\frac{\partial z^{(>l)}}{\partial z^{(l)}}$ is the key to fast computation of gradient

Backpropagation in general

- does work without layer assumptions.
- replace *neurons of next higher layer* $(l+1)$ by
  neurons such that the current neuron is input to them

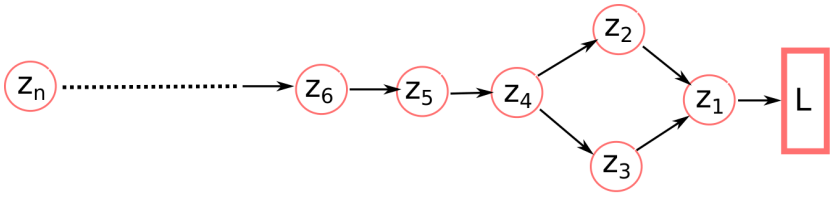




- "flow" pattern

1.9 The problem of gradient flow

Here I like to show that for classical neural networks the size of the gradient can become very small for layers close to the input.
Consider the following NN:



$$\begin{aligned}
\frac{\partial L}{\partial z_4} &= \frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial z_2}\frac{\partial z_2}{\partial z_4} &&+\frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial z_3}\frac{\partial z_3}{\partial z_4}\\
\frac{\partial L}{\partial z_5} &= \frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial z_2}\frac{\partial z_2}{\partial z_4}\frac{\partial z_4}{\partial z_5} &&+\frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial z_3}\frac{\partial z_3}{\partial z_4}\frac{\partial z_4}{\partial z_5}\\
\frac{\partial L}{\partial z_6} &= \frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial z_2}\frac{\partial z_2}{\partial z_4}\frac{\partial z_4}{\partial z_5}\frac{\partial z_5}{\partial z_6} &&+\frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial z_3}\frac{\partial z_3}{\partial z_4}\frac{\partial z_4}{\partial z_5}\frac{\partial z_5}{\partial z_6}\\
\frac{\partial L}{\partial z_n} &= \frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial z_2}\frac{\partial z_2}{\partial z_4}\frac{\partial z_4}{\partial z_5}\frac{\partial z_5}{\partial z_6}\cdot\ldots\cdot\frac{\partial z_{n-1}}{\partial z_n} &&+\frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial z_3}\frac{\partial z_3}{\partial z_4}\frac{\partial z_4}{\partial z_5}\frac{\partial z_5}{\partial z_6}\cdot\ldots\cdot\frac{\partial z_{n-1}}{\partial z_n}
\end{aligned}$$

The partial derivatives are sums of chains of products. For neurons close to the input these chains are longer.

Now lets consider a classical neural network neuron with a sigmoid activation

$$\begin{aligned}
z_1 &= tanh(w_{12}z_2 + b)\\
\frac{\partial z_1}{\partial z_2} &= tanh'(w_{12}z_2 + b)w_{12} = (1 - tanh^2(w_{12}z_2 + b))w_{12}
\end{aligned}$$

The point is: $(1 - tanh^2(w_{12}z_2 + b))$ is 1 at zero, otherwise quickly dropping to zero. Weights are usually initialized to be random values close to zero. Most of the time, such a derivative will be smaller than 1 in absolute value.

Multiplying long chains of values in $(-1, +1)$ quickly drops to zero:

$$0.5^2 = 0.25,\ 0.5^4 = 0.0625,\ 0.5^{10} \approx 0.001,\ , 0.5^{20} \approx \frac{1}{1000000}$$
$$0.1^5 = 0.00001, 0.1^{10} = 10^{-10}$$

In theory also gradient explosion may occur, if weights are set very large.

The implication? Gradient update close to the input, far from the output will be very slow.

one challenge in deep learning

In total this raises three problems:

- the gradients may become very small. Small gradients imply very low weight updates, slow learning
- the sizes of gradients will highly vary between neurons in a NN. So update speeds will vary across the network
- if used sigmoids as activations, a saturated sigmoid will result in a gradient close to zero, thus killing gradients along the whole chain downwards(see ReLU, leaky ReLU as alternatives)

One key challenge in deep learning is to maintain gradient flow so as to be able to update weights quickly, and at approximately the same speeds across the whole network

1.10 Additional material

The chain rule states in the sense of partial derivatives:

$$\frac{\partial(f \circ g)}{\partial b_k}(b) = \nabla f(g(b)) \cdot \frac{\partial(g_1, \ldots, g_R)}{\partial b_k}(b),\ \cdot \text{ is inner product of vecs}$$
$$= \sum_{r=1}^{R} \frac{\partial f}{\partial a_r}(g(b)) \frac{\partial g_r}{\partial b_k}(b)$$

Written for the whole gradient instead of the partial derivative $\frac{\partial}{\partial b_k}(b)$ we get:

$$\nabla(f \circ g)(b) = \nabla f(g(b)) \cdot \nabla g(b)$$

however note that $\nabla g(b)$ is really a matrix of shape (R, l)

which is in the sense of linear mappings in the tangent space:

$$\nabla(f \circ g)(b) \cdot h = \nabla f(g(b)) \cdot \nabla g(b) \cdot h$$

Does that makes sense from a matrix multiplication view?

$$\begin{aligned}
\nabla f(g(b)).size() &= (R) \sim (1, R) \\
\nabla g(b).size() &= (R, l) \\
\nabla f(g(b)) \cdot \nabla g(b).size() &= (1, R) \cdot (R, l) = (1, l) \\
h.size() &= (l) \sim (l, 1)
\end{aligned}$$

So one can multiply $\nabla f(g(b)) \cdot \nabla g(b)$, and obtain a term of $size() = (l) \sim (1, l)$, which in turn can be maxtrix multiplied with a vector h of size (l), as this can be reshaped into $(l, 1)$

This extends into any chain of functions:

$$\begin{aligned}
\nabla(f \circ g \circ h) &= \nabla f \cdot \nabla(g \circ h) = \nabla f \cdot \nabla g \cdot \nabla h \\
&= \nabla f(g(h(b))) \cdot \nabla g(h(b)) \cdot \nabla h(b)
\end{aligned}$$

The order how you resolve it (starting from up or from down), does not matter

$$\begin{aligned}
\nabla(f \circ g \circ h \circ e) &= \nabla(f \circ g) \cdot \nabla(h \circ e) && = \nabla f \cdot \nabla g \cdot \nabla h \cdot \nabla e \\
&= \nabla(f \circ g \circ h) \cdot \nabla e && = \nabla f \cdot \nabla g \cdot \nabla h \cdot \nabla e
\end{aligned}$$

The fact that you can resolve it in any order will help.
just note, that some of these terms are truly matrix-shaped, that is not $(1, R)$ but (S, R) in size.