Chapter 04 – Model Training | Fitting and Backpropagation

Author: Gianmarco Scarano

gianmarcoscarano@gmail.com

1. Gradient Descent

We know, in literature, that training a ML model means learning the network parameters. This can be done through an initial set of parameters (random) and by iterating the following two steps:

- Compute derivatives (gradients) of the loss w.r.t parameters
  - $$\mathrm{L}[\phi] = \sum_{i=1}^{I} \ell_i \qquad \frac{\partial L}{\partial \phi} = \begin{bmatrix} \frac{\partial L}{\partial \phi_0} \\ \frac{\partial L}{\partial \phi_1} \\ \vdots \\ \frac{\partial L}{\partial \phi_N} \end{bmatrix}$$
- Adjust parameters (weights) based on the gradients in order to decrease loss and hopefully reaching a minimum.
  - $$\phi \longleftarrow \phi - \boxed{\alpha} \frac{\partial L}{\partial \phi},$$
  - The parameter α is called: **Learning Rate**.

Gradient Descent, then, is a technique to adjust these parameters in order to decrease loss.

As for the *Linear Regression example*, we can say that our model $y = f[x, \phi]$ maps a scalar input x to a scalar output y and has parameters $\phi = [\phi_0, \phi_1]^T$, which represent the y-intercept and the slope: $y = f[x, \phi] = \phi_0 + \phi_1 x$

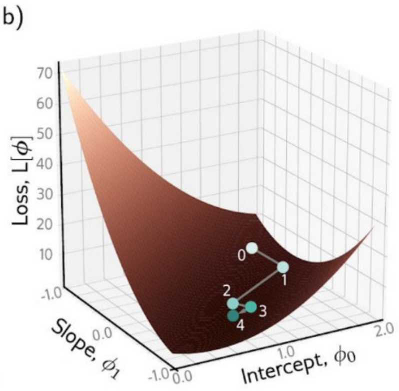
$$\mathrm{L}[\phi] = \sum_{i=1}^{I} \boxed{\ell_i} = \sum_{i=1}^{I} (\mathrm{f}[x_i, \phi] - y_i)^2 = \sum_{i=1}^{I} (\phi_0 + \phi_1 x_i - y_i)^2$$

while the *MSE loss* (right image) is defined as the individual contribution to the loss from the i-th training example. Finally, we simply compute the derivative of this loss function ($\frac{\partial L}{\partial \phi}$) w.r.t parameters ϕ as the sum of the derivatives of the individual contributions (we take the individual losses of l_i and take their derivative w.r.t. parameters $\phi_0, \phi_1, \phi_2, etc.$).

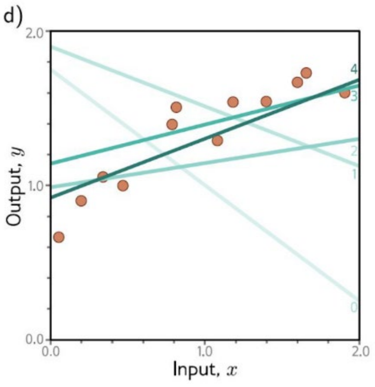
These below, instead, are visual representation of the loss:

The image (b) represents the iterations of the gradient descent. We start from point 0 and move in the steepest downhill direction until we can no more improve. The 2nd image (d) represents the model with the parameters at point 0 (lightest line) which is a very bad approximation. The darkest line is already a model with parameters which include a reasonable description of the training data.

b)

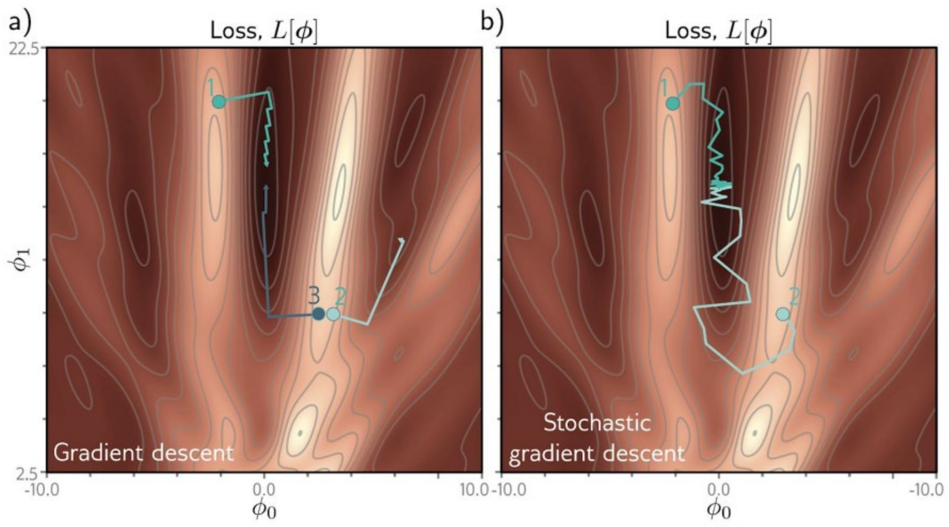


d)



2. Stochastic Gradient Descent

In SGD, we work in the same exact way as the normal Gradient Descent, but the gradient is computed for each element (x_i, y_i) in the dataset.



At each iteration, the algorithm chooses a random subset of the training data (known as minibatch/batch) and generally speaking, it works throughout the whole training examples (until it has used all the data from which then it starts sampling the full training dataset once again over and over until a fixed number of iterations).

A single pass through the entire dataset is referred as *epoch*.

The update rule for the model parameters $\phi_{t+1} \longleftarrow \phi_t - \alpha \cdot \sum_{i \in B_t} \frac{\partial \ell_i[\phi_t]}{\partial \phi}$

When we refer to $i \in B_t$ we are referring to the sum over the minibatch (or batch) of data.

2.1 Pros, Scheduled LR, Momentum, Nesterov Momentum

Although it adds noise to the trajectory, the updates tend to be sensible even if they are not optimal.

It is less computationally expensive to compute the gradient from a subset of the training data and not the whole dataset, avoiding local minima (in principle).

Another advantage of SGD is its application along with a Learning Rate schedule. This learning rate starts at a high value and slowly decreases by a constant factor every N epochs. This is applied due to the fact that in later stages of training, we are roughly in the right spot for our parameters and we decrease the LR by a very small amount in order to avoid drastically changes.

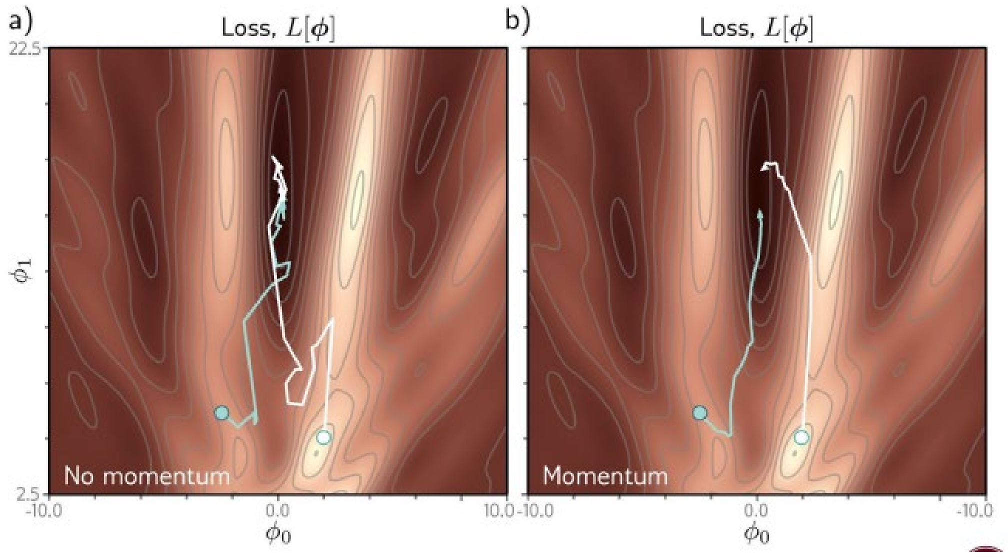
Another modification of SGD is Momentum, where we update the parameters of our model with a weighted combination of the gradient computed from the current batch and the direction moved in the previous step:

$$\mathbf{m}_{t+1} \leftarrow \beta \cdot \mathbf{m}_t + (1-\beta) \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}$$
$$\boldsymbol{\phi}_{t+1} \leftarrow \boldsymbol{\phi}_t - \alpha \cdot \mathbf{m}_{t+1},$$

where $\mathbf{m}_t$ is the momentum (which drives the update at iteration t), $\beta \in [0, 1)$ controls the degree to which the gradient is smoothed over time, and α is the learning rate.

This ensures that the overall effect is a smoother trajectory.

Figure 6.7 Stochastic gradient descent with momentum. a) Regular stochastic descent takes a very indirect path toward the minimum. b) With a momentum term, the change at the current step is a weighted combination of the previous change and the gradient computed from the batch. This smooths out the trajectory and increases the speed of convergence.



As for the Nesterov Accelerated Momentum, the gradients are computed at $\phi_t - \alpha \cdot m_t$ and not just ϕ_t. One way to look at this is that the gradient term now corrects the path provided by momentum alone. The figure below is really auto explicative:

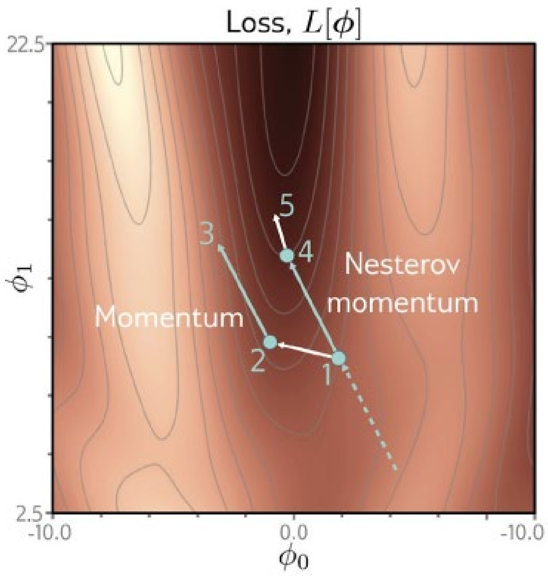


Figure 6.8 Nesterov accelerated momentum. The solution has traveled along the dashed line to arrive at point 1. A traditional momentum update measures the gradient at point 1, moves some distance in this direction to point 2, and then adds the momentum term from the previous iteration (i.e., in the same direction as the dashed line), arriving at point 3. The Nesterov momentum update first applies the momentum term (moving from point 1 to point 4) and then measures the gradient and applies an update to arrive at point 5.

The most common used SGD is the one called Adam. Here, though, the gradient descent is calculated with a fixed step size that makes large adjustments to parameters associated with large gradients, while it makes small adjustments to parameters associated with small gradients.

One way to solve this is to normalize the gradients such that we move a fixed distance in each direction of the gradient itself. This can be visualized in the plot down below (read also description for a full understanding on what's happening in figure C and D):

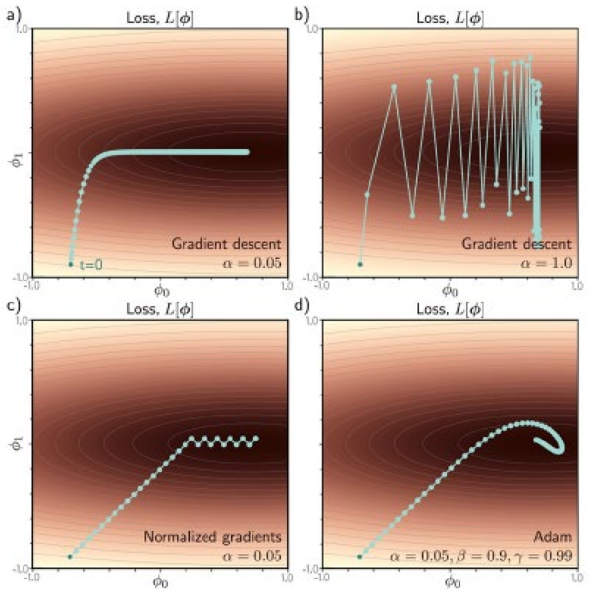


Figure 6.9 Adaptive moment estimation (Adam). a) This loss function changes quickly in the vertical direction but slowly in the horizontal direction. If we run full-batch gradient descent with a learning rate that makes good progress in the vertical direction, then the algorithm takes a long time to reach the final horizontal position. b) If the learning rate is chosen so that the algorithm makes good progress in the horizontal direction, it overshoots in the vertical direction and becomes unstable. c) A straightforward approach is to move a fixed distance along each axis at each step so that we move downhill in both directions. This is accomplished by normalizing the gradient magnitude and retaining only the sign. However, this does not usually converge to the exact minimum but instead oscillates back and forth around it (here between the last two points). d) The Adam algorithm uses momentum in both the estimated gradient and the normalization term, which creates a smoother path.

The choices of learning algorithm, batch size, learning rate schedule and momentum coefficients are all considered hyperparameters of the training algorithm. It's very common to train many models with different hyperparameters and then choose the best one. This is known as hyperparameters search.

3. Backpropagation

If we consider a simple network $f[x, \phi]$ with multivariate input x, parameters ϕ and three hidden layers h_1, h_2, h_3. Then:

$$h1 = a[\beta_0 + \Omega_0 x]$$
$$h2 = a[\beta_1 + \Omega_1 x]$$
$$h3 = a[\beta_2 + \Omega_2 x]$$
$$f[x, \phi] = [\beta_3 + \Omega_3 \mathrm{h}_3]$$

We also have a total loss computed as the sum over the individual loss terms ℓ_i:

$$L[\phi] = \sum_{i=1}^{I} \ell_i$$

- Let's assume we use some sort of Stochastic Gradient Descent

$$\boldsymbol{\phi}_{t+1} \longleftarrow \boldsymbol{\phi}_t - \alpha \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\boldsymbol{\phi}_t]}{\partial \boldsymbol{\phi}}$$

- Therefore we need to compute the derivatives:

$$\frac{\partial \ell_i}{\partial \boldsymbol{\beta}_k} \quad \text{and} \quad \frac{\partial \ell_i}{\partial \boldsymbol{\Omega}_k}$$

- This is done through Backpropapagation, a.k.a. Backprop

3.1 Forward Pass

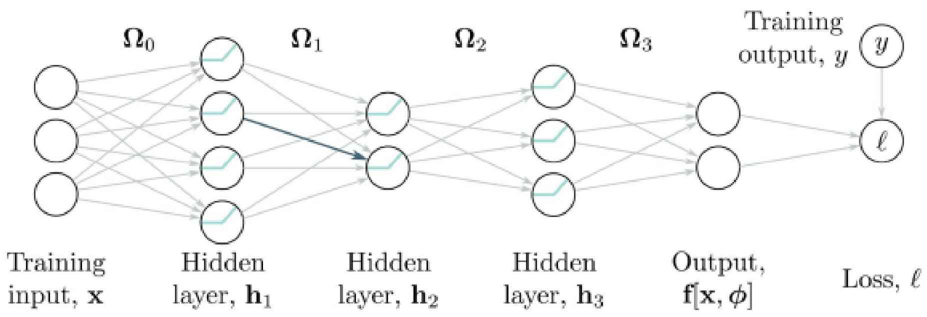

Ω_0
Ω_1
Ω_2
Ω_3
Training output, y
y
ℓ
Training input, $\mathbf{x}$
Hidden layer, $\mathbf{h}_1$
Hidden layer, $\mathbf{h}_2$
Hidden layer, $\mathbf{h}_3$
Output, $\mathbf{f}[\mathbf{x}, \phi]$
Loss, ℓ

Figure 7.1 Backpropagation forward pass. The goal is to compute the derivatives of the loss ℓ with respect to each of the weights (arrows) and biases (not shown). In other words, we want to know how a small change to each parameter will affect the loss. Each weight multiplies the hidden unit at its source and contributes the result to the hidden unit at its destination. Consequently, the effects of any small change to the weight will be scaled by the activation of the source hidden unit. For example, the blue weight is applied to the second hidden unit at layer 1; if the activation of this unit doubles, then the effect of a small change to the blue weight will double too. Hence, to compute the derivatives of the weights, we need to calculate and store the activations at the hidden layers. This is known as the *forward pass* since it involves running the network equations sequentially.

3.2 Backward Pass

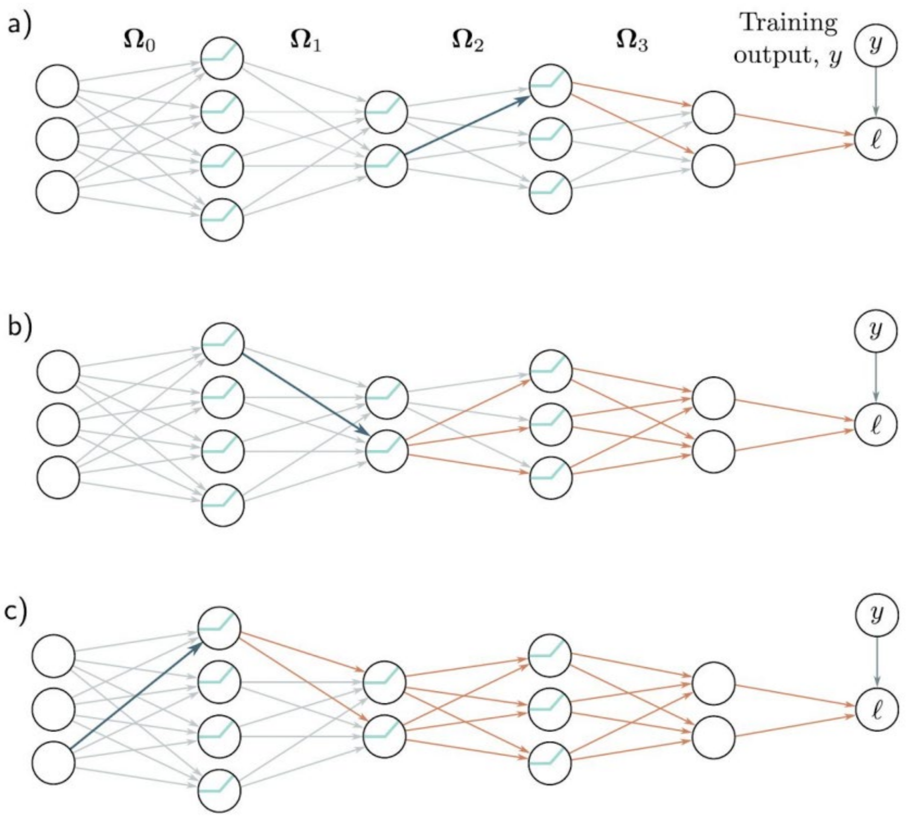
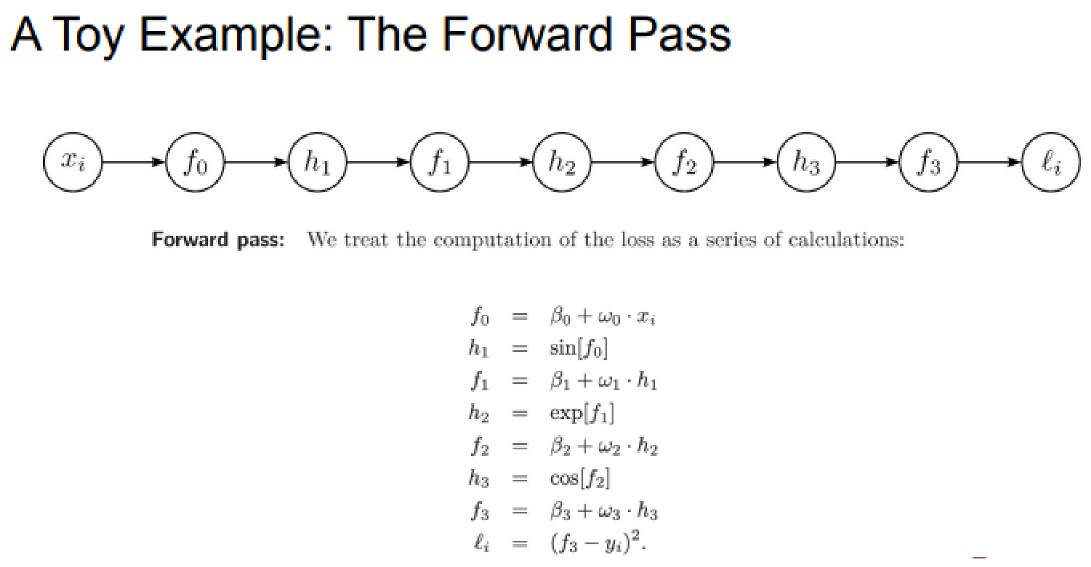


Figure 7.2 Backpropagation backward pass. a) To compute how a change to a weight feeding into layer $\mathbf{h}_3$ (blue arrow) changes the loss, we need to know how the hidden unit in $\mathbf{h}_3$ changes the model output $\mathbf{f}$ and how $\mathbf{f}$ changes the loss (orange arrows). b) To compute how a small change to a weight feeding into $\mathbf{h}_2$ (blue arrow) changes the loss, we need to know (i) how the hidden unit in $\mathbf{h}_2$ changes $\mathbf{h}_3$, (ii) how $\mathbf{h}_3$ changes $\mathbf{f}$, and (iii) how $\mathbf{f}$ changes the loss (orange arrows). c) Similarly, to compute how a small change to a weight feeding into $\mathbf{h}_1$ (blue arrow) changes the loss, we need to know how $\mathbf{h}_1$ changes $\mathbf{h}_2$ and how these changes propagate through to the loss (orange arrows). The backward pass first computes derivatives at the end of the network and then works backward to exploit the inherent redundancy of these computations.

Substantially, the goal of the forward pass it calculating the model's output based on the current set of weights and biases.

As for the backward pass, it is needed is to compute the derivatives w.r.t to the weights (and also biases) by starting from the loss function and propagate them backward through the network.

Essentially, it is needed for updating the model's parameters during training. By knowing how changes in parameters impact the loss, it leads the optimization algorithm (e.g., gradient descent) to adjust the weights and biases in a way that they do minimizes the loss, hence improving the model's performance over time.

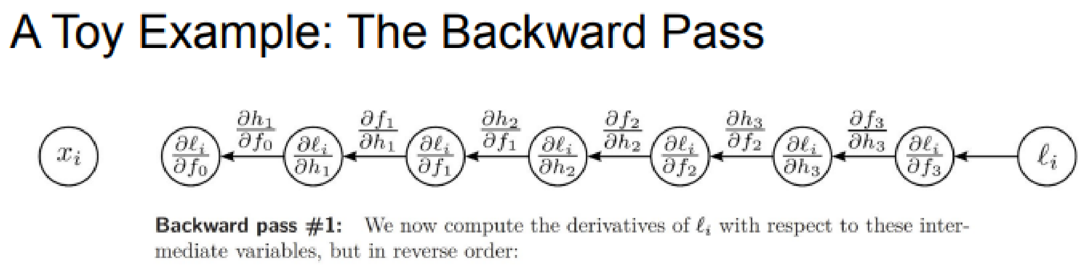


A Toy Example: The Forward Pass

$$\begin{tikzcd} x_i \arrow[r] & f_0 \arrow[r] & h_1 \arrow[r] & f_1 \arrow[r] & h_2 \arrow[r] & f_2 \arrow[r] & h_3 \arrow[r] & f_3 \arrow[r] & \ell_i \end{tikzcd}$$

Forward pass: We treat the computation of the loss as a series of calculations:

$$\begin{aligned} f_0 &= \beta_0 + \omega_0 \cdot x_i \\ h_1 &= \sin[f_0] \\ f_1 &= \beta_1 + \omega_1 \cdot h_1 \\ h_2 &= \exp[f_1] \\ f_2 &= \beta_2 + \omega_2 \cdot h_2 \\ h_3 &= \cos[f_2] \\ f_3 &= \beta_3 + \omega_3 \cdot h_3 \\ \ell_i &= (f_3 - y_i)^2. \end{aligned}$$



A Toy Example: The Backward Pass

$$\begin{tikzcd} x_i & \frac{\partial \ell_i}{\partial f_0} & \frac{\partial \ell_i}{\partial h_1} \arrow[l, "\frac{\partial h_1}{\partial f_0}"'] & \frac{\partial \ell_i}{\partial f_1} \arrow[l, "\frac{\partial f_1}{\partial h_1}"'] & \frac{\partial \ell_i}{\partial h_2} \arrow[l, "\frac{\partial h_2}{\partial f_1}"'] & \frac{\partial \ell_i}{\partial f_2} \arrow[l, "\frac{\partial f_2}{\partial h_2}"'] & \frac{\partial \ell_i}{\partial h_3} \arrow[l, "\frac{\partial h_3}{\partial f_2}"'] & \frac{\partial \ell_i}{\partial f_3} \arrow[l, "\frac{\partial f_3}{\partial h_3}"'] & \ell_i \arrow[l] \end{tikzcd}$$

Backward pass #1: We now compute the derivatives of ℓ_i with respect to these intermediate variables, but in reverse order:

4. Parameter initialization

If we consider that during a forward pass, each set of pre-activations f_k is computed as:

$$\begin{aligned} f_k &= \beta_k + \Omega_k \mathrm{h}_k \\ &= \beta_k + \Omega_k \mathrm{a}[f_{k-1}] \end{aligned}$$

Where a[·] applies the ReLU functions and Ω_k and β_k are respectively, the weights and biases of the network.

With this being said, imagine we initialize all the biases to zero and the elements of Ω_k according to a normal distribution with mean zero and variance σ^2.

If the variance σ^2 is then too small (e.g. 10^{-5}), each element of $\beta_k + \Omega_k \mathrm{h}_k$ will be a weighted sum of h_k, thus as we pass through the layers of a neural network, the pre-activation values (f_k) tend to become smaller due to the combination of small weights, ReLU activation, and initializations with low variances.

In contrary, if the variance σ^2 is pretty large (e.g. 10^5), the weights will be very large. It's true that the ReLU function will always clip the inputs as before, but if σ^2 is large, pre-activation values (f_k) tend to become bigger as we proceed through the network.

These two problems are denoted as vanishing gradient and exploding gradient, respectively, where the gradient magnitudes may decrease or increase uncontrollably during the backward pass.