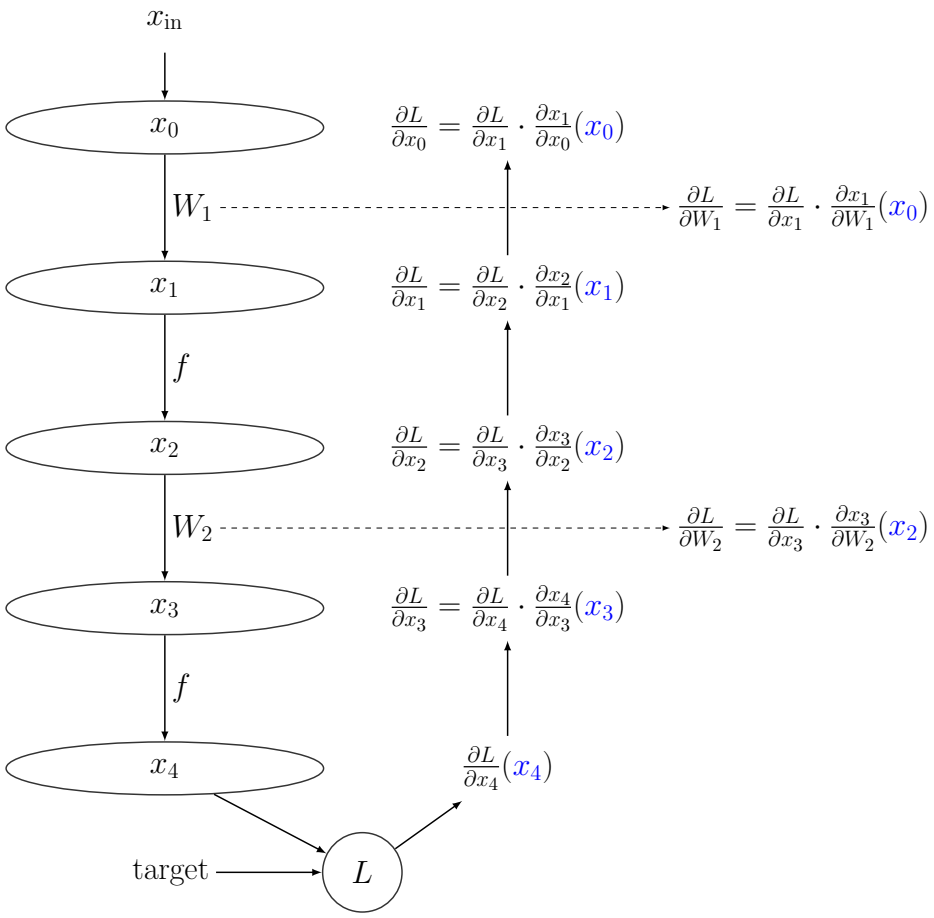
Backpropagation refresher

Luke Rast



In this visualization of backpropagation, input training data, x_{in}, is fed into the network at the top. It goes through a series of weight multiplication, W_i, and activation function, f, steps to produce outputs, x_4. These are compared to the training targets through a loss function, L. The gradients of L are then back-propagated upward, multiplying at each step by gradient of the layer's output with respect to its inputs $\frac{\partial x_{i+1}}{\partial x_i}$.

There are two things to note in the visualization. First, there are two different types of arrows in the backward step. The solid arrows leading upward correspond to the backpropagation through the layers, while the dashed arrows to the right allow us to compute, given the backpropagated gradients, the gradients with respect to the weights themselves. Second, the blue arguments correspond to the results from the forward pass that are necessary to compute the gradients in the corresponding

layer. These results should be stored in memory between forward and reverse passes, so we don't have to recompute them. Note that if the W_i are truly weight matrix multiplications, the backprop gradients of these layers will not depend on the input x_i, however, these x_i are still necessary for computation of the weight gradients.

This visualization corresponds directly to the pytorch API. Setting the `Weight.requires_grad` flag to True when initializing the weight tensors tells pytorch to track the inputs and gradients of these operations. This property is then inherited by downstream operations, which will also do the tracking. Once the Loss is computed, we can then run `Loss.backward()` which will perform backpropagation along the solid arrows. Then `Weight.grad` gives the weight gradients, through the dashed arrows. Finally, once the weights have been updated, we *have to* run `Weight.grad.zero_()` for each of our tracked weights to clear their cache of input values.