# Introduction to Deep Learning

## Chapter 3: Basic Backpropagation 2

**Bernd Bischl**

Department of Statistics – LMU Munich

WS 2021/2022

# BACKWARD COMPUTATION AND CACHING

In the XOR example, we computed:

$$\frac{\partial L(y, f(\mathbf{x}))}{\partial W_{11}} = \frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} \cdot \frac{\partial f_{out}}{\partial f_{in}} \cdot \frac{\partial f_{in}}{\partial z_{1,out}} \cdot \frac{\partial z_{1,out}}{\partial z_{1,in}} \cdot \frac{\partial z_{1,in}}{\partial W_{11}}$$

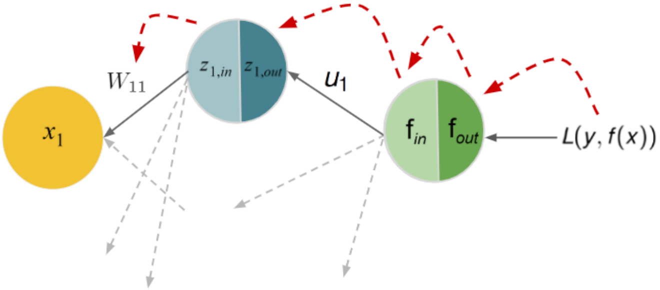



**Figure:** All five terms of our chain rule

# BACKWARD COMPUTATION AND CACHING

Next, let us compute:

$$\frac{\partial L\left(y, f(\mathbf{x})\right)}{\partial W_{21}} = \frac{\partial L\left(y, f(\mathbf{x})\right)}{\partial f_{out}} \cdot \frac{\partial f_{out}}{\partial f_{in}} \cdot \frac{\partial f_{in}}{\partial z_{1,out}} \cdot \frac{\partial z_{1,out}}{\partial z_{1,in}} \cdot \frac{\partial z_{1,in}}{\partial W_{21}}$$

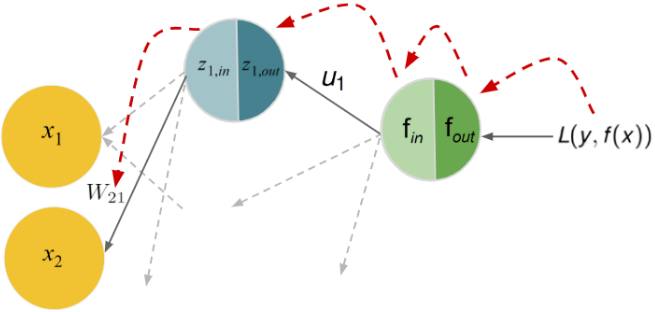



**Figure:** All five terms of our chain rule

# BACKWARD COMPUTATION AND CACHING

- Examining the two expressions:

$$\frac{\partial L(y, f(\mathbf{x}))}{\partial W_{11}} = \frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} \cdot \frac{\partial f_{out}}{\partial f_{in}} \cdot \frac{\partial f_{in}}{\partial z_{1,out}} \cdot \frac{\partial z_{1,out}}{\partial z_{1,in}} \cdot \frac{\partial z_{1,in}}{\partial W_{11}}$$

$$\frac{\partial L(y, f(\mathbf{x}))}{\partial W_{21}} = \frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} \cdot \frac{\partial f_{out}}{\partial f_{in}} \cdot \frac{\partial f_{in}}{\partial z_{1,out}} \cdot \frac{\partial z_{1,out}}{\partial z_{1,in}} \cdot \frac{\partial z_{1,in}}{\partial W_{21}}$$

- We see that there is significant overlap / redundancy in the two expressions. A huge chunk of the second expression has already been computed while computing the first one.
- **Again**: Simply cache these subexpressions instead of recomputing them each time.

# BACKWARD COMPUTATION AND CACHING

- Let

$$\delta_1 = \frac{\partial L(y, f(\mathbf{x}))}{\partial z_{1,in}} = \frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} \cdot \frac{\partial f_{out}}{\partial f_{in}} \cdot \frac{\partial f_{in}}{\partial z_{1,out}} \cdot \frac{\partial z_{1,out}}{\partial z_{1,in}}$$

  be cached. $\delta_1$ can also be seen as an **error signal** that represents how much the loss $L$ changes when the input $z_{1,in}$ changes.
- The two expressions of the previous slide now become,

$$\frac{\partial L(y, f(\mathbf{x}))}{\partial W_{11}} = \delta_1 \cdot \frac{\partial z_{1,in}}{\partial W_{11}} \quad \text{and} \quad \frac{\partial L(y, f(\mathbf{x}))}{\partial W_{21}} = \delta_1 \cdot \frac{\partial z_{1,in}}{\partial W_{21}}$$

  where $\delta_1$ is simply "plugged in".
- As you can imagine, caching subexpressions in this way and plugging in where needed can result in **massive** gains in efficiency for deep and "wide" neural networks.
- In fact, this simple algorithm, which was first applied to neural networks way back in 1985, is **still** the biggest breakthrough in deep learning.

# BACKWARD COMPUTATION AND CACHING

- On the other hand, if we had done a **forward** caching of the derivatives

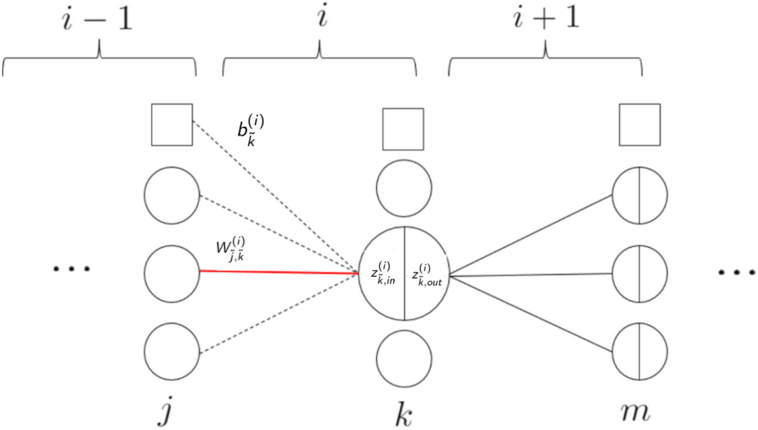
$$\frac{\partial L(y, f(\mathbf{x}))}{\partial W_{11}} = \left(\left(\left(\frac{\partial z_{1,in}}{\partial W_{11}} \frac{\partial z_{1,out}}{\partial z_{1,in}}\right) \frac{\partial f_{in}}{\partial z_{1,out}}\right) \frac{\partial f_{out}}{\partial f_{in}}\right) \frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}}$$

$$\frac{\partial L(y, f(\mathbf{x}))}{\partial W_{21}} = \left(\left(\left(\frac{\partial z_{1,in}}{\partial W_{21}} \frac{\partial z_{1,out}}{\partial z_{1,in}}\right) \frac{\partial f_{in}}{\partial z_{1,out}}\right) \frac{\partial f_{out}}{\partial f_{in}}\right) \frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}}$$

  there would be no common subexpressions to plug in. We would have to compute the entire expression for each and every weight!
- This would make the computations too inefficient to make gradient descent tractable for large neural networks.

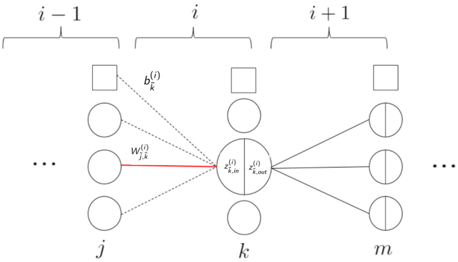
# BACKPROPAGATION: FORMALISM

- Let us now derive a general formulation of backpropagation.
- The neurons in layers $i-1$, $i$ and $i+1$ are indexed by $j$, $k$ and $m$, respectively.
- The output layer will be referred to as layer O.




Credit: Erik Hallström

# BACKPROPAGATION: FORMALISM




- Let $\delta_{\tilde{k}}^{(i)}$ (also: error signal) for a neuron $\tilde{k}$ in layer $i$ represent how much the loss $L$ changes when the input $z_{\tilde{k},in}^{(i)}$ changes:

$$\delta_{\tilde{k}}^{(i)} = \frac{\partial L}{\partial z_{\tilde{k},in}^{(i)}} = \frac{\partial L}{\partial z_{\tilde{k},out}^{(i)}} \frac{\partial z_{\tilde{k},out}^{(i)}}{\partial z_{\tilde{k},in}^{(i)}} = \sum_m \left( \frac{\partial L}{\partial z_{m,in}^{(i+1)}} \frac{\partial z_{m,in}^{(i+1)}}{\partial z_{\tilde{k},out}^{(i)}} \right) \frac{\partial z_{\tilde{k},out}^{(i)}}{\partial z_{\tilde{k},in}^{(i)}}$$

- Note: The sum in the expression above is over all the neurons in layer $i+1$. This is simply an application of the chain rule.

# BACKPROPAGATION: FORMALISM

Using

$$
\begin{aligned}
z_{\tilde{k},out}^{(i)} &= \sigma(z_{\tilde{k},in}^{(i)}) \\
z_{m,in}^{(i+1)} &= \sum_k W_{k,m}^{(i+1)} z_{k,out}^{(i)} + b_m^{(i+1)}
\end{aligned}
$$

we get:

$$
\begin{aligned}
\delta_{\tilde{k}}^{(i)} &= \sum_m \left( \frac{\partial L}{\partial z_{m,in}^{(i+1)}} \frac{\partial z_{m,in}^{(i+1)}}{\partial z_{\tilde{k},out}^{(i)}} \right) \frac{\partial z_{\tilde{k},out}^{(i)}}{\partial z_{\tilde{k},in}^{(i)}} \\
&= \sum_m \left( \frac{\partial L}{\partial z_{m,in}^{(i+1)}} \frac{\partial \left( \sum_k W_{k,m}^{(i+1)} z_{k,out}^{(i)} + b_m^{(i+1)} \right)}{\partial z_{\tilde{k},out}^{(i)}} \right) \frac{\partial \sigma(z_{\tilde{k},in}^{(i)})}{\partial z_{\tilde{k},in}^{(i)}} \\
&= \sum_m \left( \frac{\partial L}{\partial z_{m,in}^{(i+1)}} W_{\tilde{k},m}^{(i+1)} \right) \sigma'(z_{\tilde{k},in}^{(i)}) = \sum_m \left( \delta_{\tilde{k}}^{(i+1)} W_{\tilde{k},m}^{(i+1)} \right) \sigma'(z_{\tilde{k},in}^{(i)})
\end{aligned}
$$

Therefore, we now have a **recursive definition** for the error signal of a neuron in layer $i$ in terms of the error signals of the neurons in layer $i+1$ and, by extension, layers {i+2, i+3 . . . , O}!

# BACKPROPAGATION: FORMALISM

- Given the error signal $\delta_{\tilde{k}}^{(i)}$ of neuron $\tilde{k}$ in layer $i$, the derivative of loss $L$ w.r.t. to the weight $W_{\tilde{j},\tilde{k}}$ is simply:

$$\frac{\partial L}{\partial W_{\tilde{j},\tilde{k}}^{(i)}} = \frac{\partial L}{\partial z_{\tilde{k},in}^{(i)}} \frac{\partial z_{\tilde{k},in}^{(i)}}{\partial W_{\tilde{j},\tilde{k}}^{(i)}} = \delta_{\tilde{k}}^{(i)} z_{\tilde{j},out}^{(i-1)}$$

  because $z_{\tilde{k},in}^{(i)} = \sum_j W_{j,\tilde{k}}^{(i)} z_{j,out}^{(i-1)} + b_{\tilde{k}}^{(i)}$

- Similarly, the derivative of loss $L$ w.r.t. bias $b_{\tilde{k}}^{(i)}$ is:

$$\frac{\partial L}{\partial b_{\tilde{k}}^{(i)}} = \frac{\partial L}{\partial z_{\tilde{k},in}^{(i)}} \frac{\partial z_{\tilde{k},in}^{(i)}}{\partial b_{\tilde{k}}^{(i)}} = \delta_{\tilde{k}}^{(i)}$$

# BACKPROPAGATION: FORMALISM

- We have seen how to compute the error signals for individual neurons. It can be shown that the error signal $\boldsymbol{\delta}^i$ for an entire layer $i$ can be computed as follows ($\odot$ = element-wise product):
  - $\boldsymbol{\delta}^{(O)} = \nabla_{f_{out}} L \odot \tau'(f_{in})$
  - $\boldsymbol{\delta}^{(i)} = W^{(i+1)} \boldsymbol{\delta}^{(i+1)} \odot \sigma'(z_{in}^{(i)})$
- As we have seen earlier, the error signal for a given layer $i$ depends recursively on the error signals of **later** layers {i+1, i+2, . . . , O}.
- Therefore, backpropagation works by computing and storing the error signals **backwards**. That is, starting at the output layer and ending at the first hidden layer. This way, the error signals of later layers **propagate backwards** to the earlier layers.
- The derivative of the loss $L$ w.r.t. a given weight is computed efficiently by plugging in the cached error signals thereby avoiding expensive and redundant computations.