Neural Networks and Automatic Differentiation

Advanced Computer Vision

Niels Landwehr

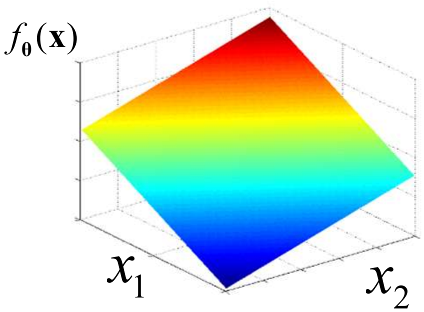
Overview

- Introduction: Computer Vision
- Data, Models, Optimization
- Neural Networks and Automatic Differentiation

Recap: Linear Models

- So far: linear models for regression and classification

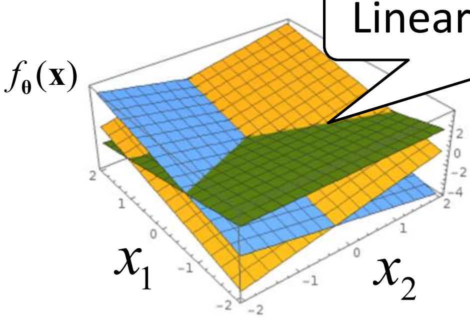
Linear regression



Number of attributes

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x} + b \quad \mathbf{w} \in \mathbb{R}^d, \quad b \in \mathbb{R}, \quad \boldsymbol{\theta} = (\mathbf{w}, b)$$

Linear classification



Linear class boundaries

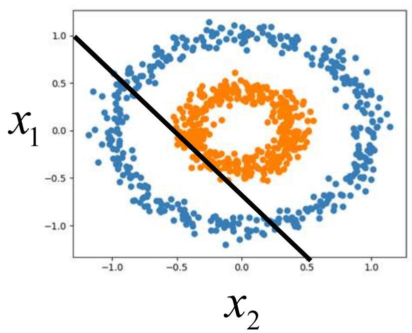
Number of classes

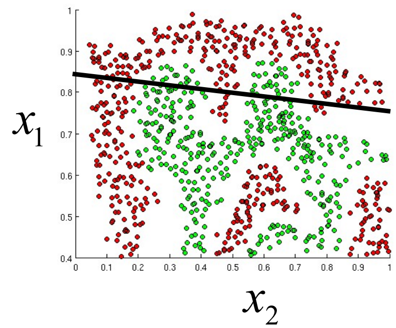
$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b} \quad \mathbf{W} \in \mathbb{R}^{k \times d}, \quad \mathbf{b} \in \mathbb{R}^k, \quad \boldsymbol{\theta} = (\mathbf{W}, \mathbf{b})$$

Vector of class scores.
Prediction = class with highest score

Limitations of Linear Models

- Expressivity of linear models is limited: in the real world, relationship between data and labels often highly nonlinear





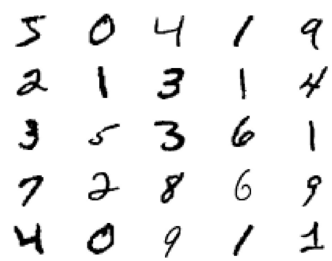
Classes (colors) not separable by linear model (black line)

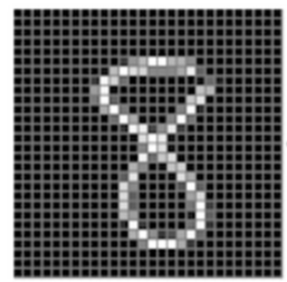
- Computer vision: training a linear model on pixel data will not get us very far
  – MNist benchmark data set (digits, 10 classes): linear model 92% accuracy, state of the art 99.x% accuracy
  – ImageNet benchmark data set (general objects, 1000 classes): linear model 10% top-5 accuracy, state of the art 98.x% top-5 accuracy

Linear Models in Computer Vision

- Computer vision: linear classification model encodes one „template“per class
- Example MNist: digits as 28x28 grayscale images, scaled, centered

10-class problem





- To apply linear model, „flatten“ the 28x28 images into 784-dimensional vector
- Linear classification model:

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b} \in \mathbb{R}^{10}$$

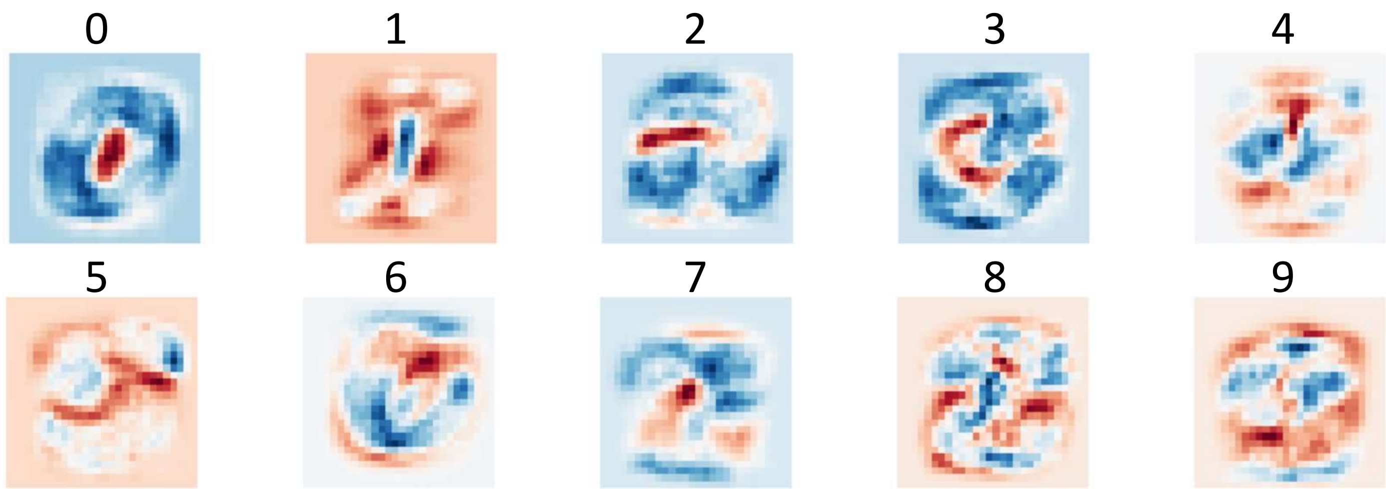
class scores

$$\mathbf{W} \in \mathbb{R}^{10\times 784}, \quad \mathbf{b} \in \mathbb{R}^{10}$$

To compute score of i-th class, multiply i-th row of $\mathbf{W}$ with input $\mathbf{x}$ (and add bias)

Linear Models as Templates

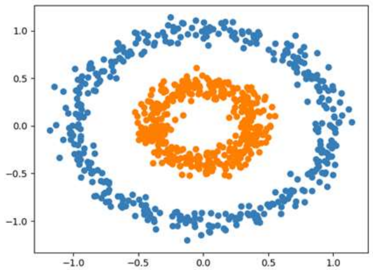
- Computer vision: linear classification model encodes one „template“ per class
- Rows of $\mathbf{W}$, reshaped to 28x28, blue = positive values, red = negative values:



- Each row matches on a certain visual pattern by „looking“ for a particular combination of present/missing pixel values in the image
- The more closely the image matches the pattern, the higher the class score
- Works reasonably well for MNist (centered, scaled digits). Not so well if there is large variation of shapes/appearances within class

Linear Models + Feature Maps

- One way of going beyond linear models: nonlinear feature maps
- Example:



Blue = class 1, red = class 2.
Not linearly separable.
Circle with appropriate radius r would separate classes.

- Idea: define feature map $\Phi : \mathbb{R}^2 \rightarrow \mathbb{R}^2, \quad \Phi(\mathbf{x}) = (x_1^2, x_2^2)$
- Feature map transforms each instance $\mathbf{x}$ into a transformed instance $\Phi(\mathbf{x})$
- Learn a linear model in the transformed space:

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{W}\Phi(\mathbf{x}) + \mathbf{b} \quad \mathbf{W} = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} -r^2 \\ 0 \end{pmatrix}$$

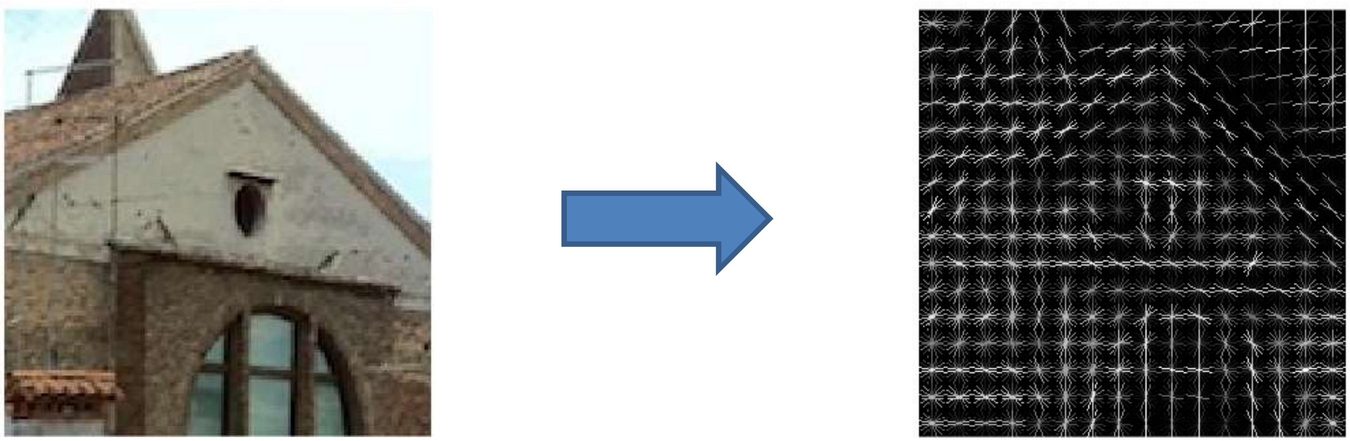
Works: score of class 1 larger than score of class 2 iff $x_1^2 + x_2^2 > r^2$

- Learning becomes easy with the right features!
- Where do the features come from?

Linear Models + Feature Maps

- Explicit nonlinear features, for example polynomial features $x_1^2, x_2^2, x_1 x_2, \ldots$
- Kernel trick: implicitly map instances by means of a **kernel** (details: ML lecture)
- But which feature map or kernel is good for a particular problem?
- Domain-specific features, e.g. for computer vision (not really used anymore):

HoGs („histogram of oriented gradients“) features: On small image patches (e.g., 8x8 pixels), compute histograms of edge directions and strengths



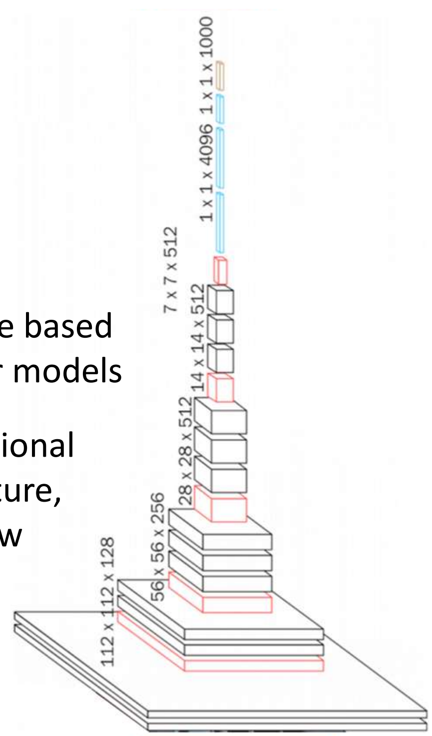
Learning Features

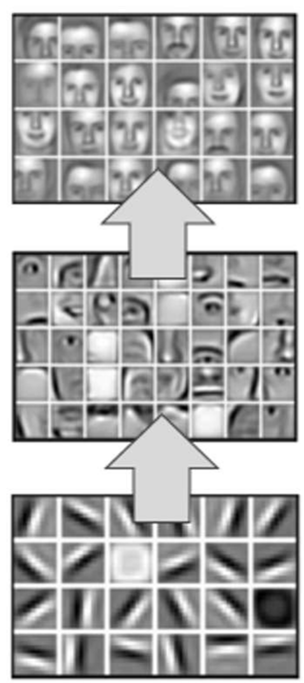
- **Neural networks**: models consisting of several stacked layers
- Lower layers perform feature extraction, highest layer prediction
- All layer are learned „end-to-end“ on raw data, no feature engineering needed

Example: Face recognition with deep convolutional neural network

layers are based on linear models

convolutional architecture, see below





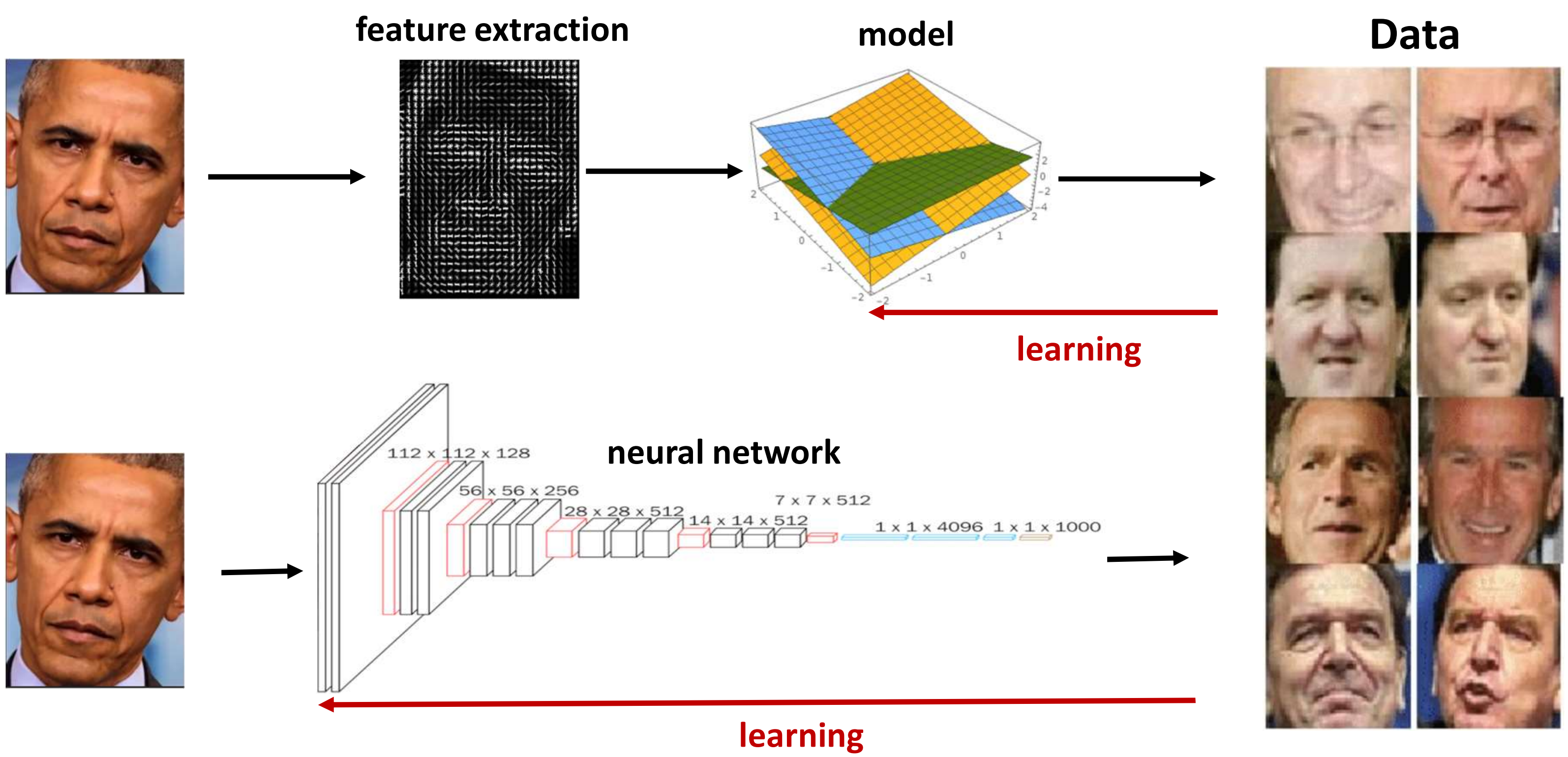
High-level layers represent identities of persons

Intermediate layers combine the low-level features to more informative features such as face parts

Low-level layers extract simple features such as edges or circles from raw pixels

Neural Networks: End-to-End Learning

- **All layer are learned „end-to-end“ on raw data, no feature engineering needed**
- Also called „feature learning“ or „representation learning“
- Advantage: neural network learns to extract features most helpful for task



From Linear Models to Neural Networks

- So far: linear model (k is number of classes)

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}$$

Output: vector (of class scores)

$$\mathbf{x} \in \mathbb{R}^{d}, \quad \mathbf{W} \in \mathbb{R}^{k \times d}, \quad \mathbf{b} \in \mathbb{R}^{k}, \quad \boldsymbol{\theta} = (\mathbf{W}, \mathbf{b})$$

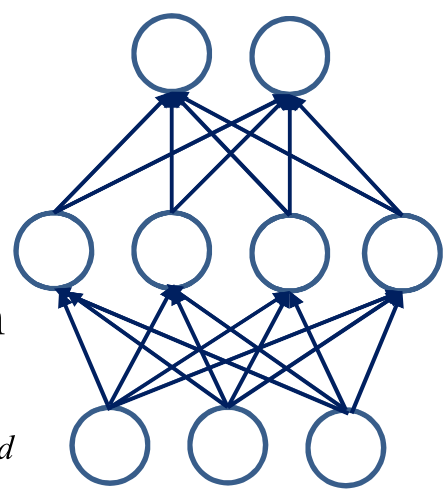
- **Idea**: stack multiple linear models to increase representational power

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \overbrace{\mathbf{W}_2 \mathbf{z}_1 + \mathbf{b}_2}^{\text{second linear model}}$$

$$\mathbf{z}_1 = \sigma(\overbrace{\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1}^{\text{first linear model}})$$

$\sigma(z)$ nonlinear function

$$\mathbf{x} \in \mathbb{R}^{d}$$



$$\mathbf{W}_2 \in \mathbb{R}^{k \times k_1}, \quad \mathbf{b}_2 \in \mathbb{R}^{k}$$

$$\mathbf{W}_1 \in \mathbb{R}^{k_1 \times d}, \quad \mathbf{b}_1 \in \mathbb{R}^{k_1}$$

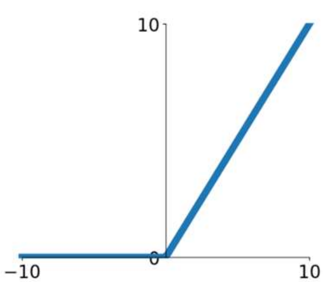
Nonlinear Activation Function

- Why nonlinear activation $\sigma(z)$? Without nonlinear activation, the entire model would stay linear

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{W}_2(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 = \underbrace{\mathbf{W}_2\mathbf{W}_1}_{\mathbf{W} \in \mathbb{R}^{k \times d}}\mathbf{x} + \underbrace{(\mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2)}_{\mathbf{b} \in \mathbb{R}^{k}} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

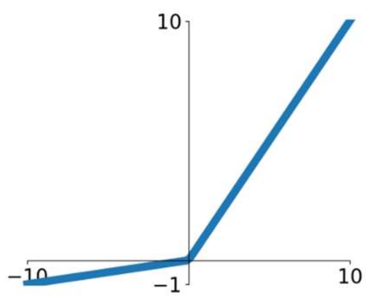
- Different nonlinear activation functions are used (non-exhaustive list):

ReLU: Rectified linear unit



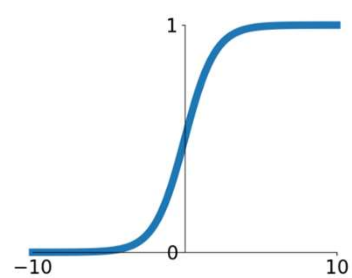
$\sigma(z) = \max(0, z)$

Leaky ReLU



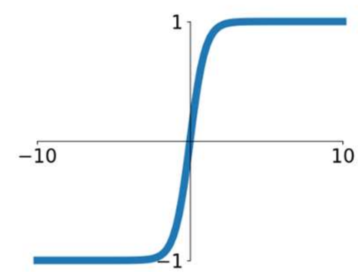
$\sigma(z) = \max(0.1z, z)$

Sigmoid



$\sigma(z) = \dfrac{1}{1 + e^{-z}}$

tanh



$\sigma(z) = \tanh(z)$

Multilayer Perceptron

- We can stack any number of layers to get a more expressive model
- In general, this leads to a so-called Multilayer Perceptron:

Definition (Multilayer Perceptron)

Let $\mathbf{x} \in \mathbb{R}^d$ denote the input vector, let $D \in \mathbb{N}$ denote the number of hidden layers, $k_1, \ldots, k_D \in \mathbb{N}$ denote the number of units in the hidden layers, and let $k \in \mathbb{N}$ denote the number of classes. Let $\sigma(z) : \mathbb{R} \to \mathbb{R}$ be an activation function.
Let $\mathbf{W}_1 \in \mathbb{R}^{k_1 \times d}, \mathbf{W}_2 \in \mathbb{R}^{k_2 \times k_1}, \ldots, \mathbf{W}_D \in \mathbb{R}^{k_D \times k_{D-1}}, \mathbf{W}_{D+1} \in \mathbb{R}^{k \times k_D}$ denote weight matrices and $\mathbf{b}_1 \in \mathbb{R}^{k_1}, \ldots, \mathbf{b}_D \in \mathbb{R}^{k_D}, \mathbf{b}_{D+1} \in \mathbb{R}^{k}$ denote bias vectors.

The function $f_{\boldsymbol{\theta}} : \mathbb{R}^d \to \mathbb{R}^k$ given by

$$\mathbf{z}_0 = \mathbf{x}$$

$$\mathbf{z}_i = \sigma(\mathbf{W}_i \mathbf{z}_{i-1} + \mathbf{b}_i) \qquad i \in \{1, \ldots, D\} \qquad \text{"Activations at layer } i\text{"}$$

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{W}_{D+1} \mathbf{z}_D + \mathbf{b}_{D+1}$$

is called a multilayer perceptron. Its parameters are $\boldsymbol{\theta} = \{\mathbf{W}_i, \mathbf{b}_i\}_{i=1}^{D+1}$.

Multilayer Perceptrons as Graphs

- A Multilayer Perceptron can be visualized as a directed acyclic graph (DAG)
- Graph is organized into layers that are fully connected
  - There are *D+2* layers, called „input“, „hidden“ (*D* layers) and „output“
  - Input layer has *d* nodes, hidden layer *i* has k_i nodes, output has *k* nodes

$$\text{Example: } d = 3, \;\; k = 2, \;\; k_1 = 4, \;\; D = 1$$

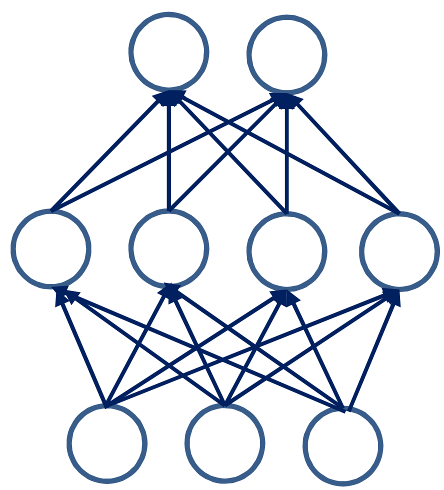
$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{W}_2 \mathbf{z}_1 + \mathbf{b}_2$$

$$\mathbf{W}_2 \in \mathbb{R}^{k \times k_1}, \;\; \mathbf{b}_2 \in \mathbb{R}^{k}$$

$$\mathbf{z}_1 = \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)$$

$$\mathbf{W}_1 \in \mathbb{R}^{k_1 \times d}, \;\; \mathbf{b}_1 \in \mathbb{R}^{k_1}$$

$$\mathbf{x} \in \mathbb{R}^{d}$$



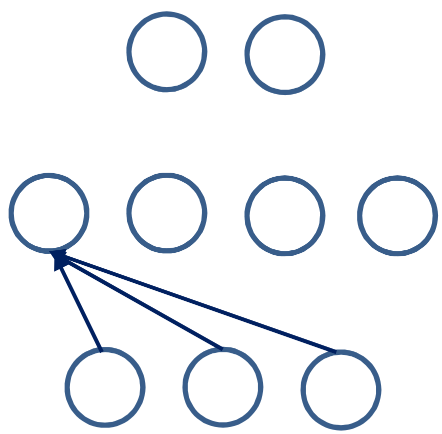
Multilayer Perceptrons as Graphs

- Nodes in the graph are sometimes called „neurons“. The elements $z_{i,j}$ of the vector $\mathbf{z}_i$ are called „activations“ of the neurons in layer *i*
- The activation of a single neuron $z_{i,j}$ can be computed from its inputs, the corresponding entries in the weight matrix $\mathbf{W}_i$ and the bias vector $\mathbf{b}_i$:

$$z_{i,j} = \sigma(\mathbf{w}_{i,j}\mathbf{z}_{i-1} + b_{i,j}) \qquad \mathbf{w}_{i,j} = \text{j-th row of } \mathbf{W}_i, \;\; \mathrm{b}_{i,j} = \text{j-th entry in } \mathbf{b}_i$$

weights of neuron

bias of neuron



- In this sense, computations are local to nodes.
- Any DAG structure is possible, simply compute activations node-by-node

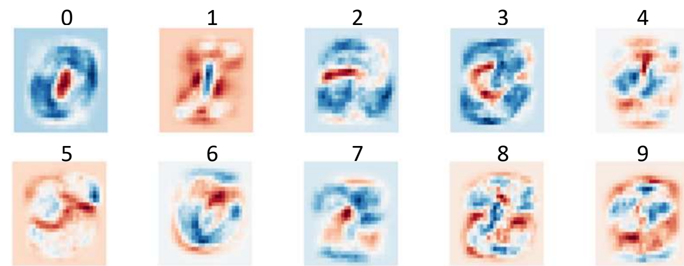
Neurons Match Patterns

- Reminder: rows in a linear model match one class "template" each

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}$$

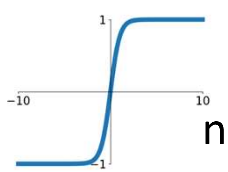
Output: vector (of class scores)

$$\mathbf{W} \in \mathbb{R}^{k \times d}, \quad \mathbf{b} \in \mathbb{R}^{k}, \quad \boldsymbol{\theta} = (\mathbf{W}, \mathbf{b})$$

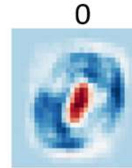


MNist: class templates

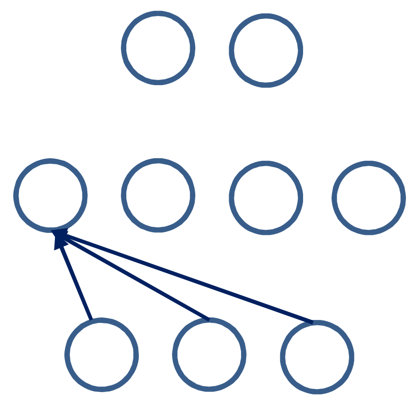
- Neurons match patterns (of the input or some other layer) in a similar way



$$z_{i,j} = \sigma(\mathbf{w}_{i,j}\mathbf{z}_{i-1} + b_{i,j})$$



neuron „activates“ if pattern matches



- Neurons in higher layers recombine simple features to complex ones

Preview: Neural Networks For Image Data

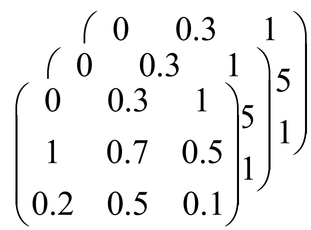
- So far: Multilayer perceptron
  - very simple model structure: stacked fully connected layers
  - ok for simple vector-based inputs
  - not very suitable for image data, where input is a 3D-tensor

$$\mathcal{X} = \mathbb{R}^{m \times l \times d}$$

m = image height

l = image width

d = number of channels



- Next week: **convolutional neural network architectures** for computer vision problems, take 3D-structure of images into account.

Recap: Cross-Entropy Loss

- Multilayer perceptron: function $f_{\boldsymbol{\theta}} : \mathbb{R}^d \to \mathbb{R}^k$ with parameters $\boldsymbol{\theta} = \{\mathbf{W}_i, \mathbf{b}_i\}_{i=1}^{D+1}$
- Classification scenario: output are scores for the *k* classes
- Regression scenario: *k* = 1, the single output is the predicted real value

- As for a linear model, we can use cross-entropy loss for classification where data are $(\mathbf{x}_1, \ldots, \mathbf{x}_n),\ (y_1, \ldots, y_n),\ y_i \in \{c_1, \ldots, c_k\}$

Transform class scores to probabilities:

probability for *j*-th class

$$p(y = c_j \mid \mathbf{x}, \boldsymbol{\theta}) = \frac{\exp(f_{\boldsymbol{\theta}}(\mathbf{x})_j)}{\sum_{i=1}^{k} \exp(f_{\boldsymbol{\theta}}(\mathbf{x})_i)}$$

score for class *j*

Cross-entropy loss: negative log-probability of observed class:

loss on *i*-th instance

$$\ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i) = -\log p(y = y_i \mid \mathbf{x}_i, \boldsymbol{\theta})$$

-log probability of label y_i according to model

Recap: Regression Loss

- Multilayer perceptron: function $f_{\boldsymbol{\theta}} : \mathbb{R}^d \to \mathbb{R}^k$ with parameters $\boldsymbol{\theta} = \{\mathbf{W}_i, \mathbf{b}_i\}_{i=1}^{D+1}$
- Classification scenario: output are scores for the *k* classes
- Regression scenario: *k* = 1, the single output is the predicted real value

- As for a linear model, we can use mean squared or mean average loss for regression where data are $(\mathbf{x}_1, \ldots, \mathbf{x}_n),\ (y_1, \ldots, y_n),\ y_i \in \mathbb{R}$

loss on *i*-th instance

$$\ell_1(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i) = | f_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i |$$

$$\ell_2(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i) = (f_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i)^2$$

absolute or squared difference between predicted and observed output

Parameter Optimization

- Learning a multilayer perceptron from data: solving optimization problem

$$\boxed{\textbf{Optimization:}\quad \boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} L(\boldsymbol{\theta})}$$

$$L(\boldsymbol{\theta}) = \underbrace{\frac{1}{n}\sum_{i=1}^{n} \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i)}_{\text{Loss}} + \lambda \underbrace{R(\boldsymbol{\theta})}_{\text{Regularization}}$$

Loss: model predictions should match training data

Regularization: prevent model from doing **too** well on the training data, prefer simpler models

- Optimization carried out by stochastic gradient descent in parameters $\boldsymbol{\theta}$
- Regularization can be for example L1 or L2

L2-regularizer $R(\boldsymbol{\theta}) = \sum_j \theta_j^2$

L1-regularizer $R(\boldsymbol{\theta}) = \sum_j |\theta_j|$

θ_j: any model parameter, e.g. for multilayer perceptron entries in any $\mathbf{W}_i, \mathbf{b}_i$

Optimization: Gradient

- For optimization with stochastic gradient descent, need to derive the gradient $\nabla L(\boldsymbol{\theta})$ of the loss in the model parameters $\boldsymbol{\theta}$
- Derive gradient manually?
  - Difficult for large, complex models with millions of parameters
  - Not flexible: for any change in model or loss function, would need to rederive gradient

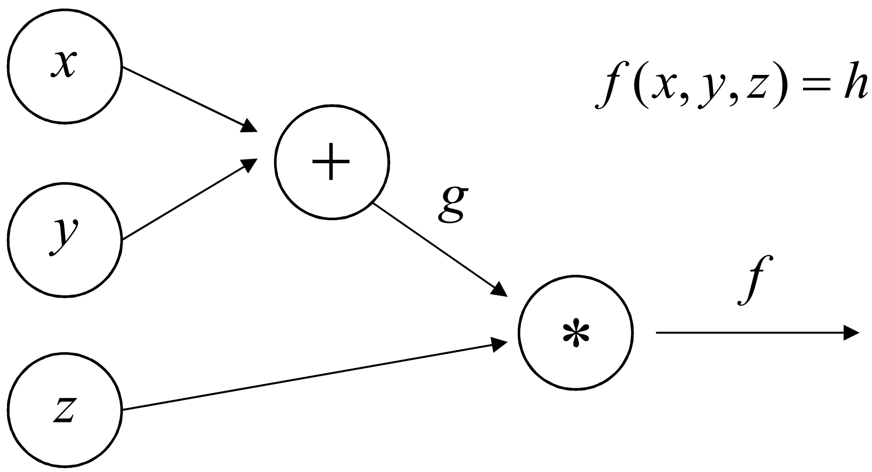
- Instead: derive gradient algorithmically using **automatic differentiation**, also known (in this specific case) as **backpropagation**

- Flexible approach that can compute gradients automatically for any model architecture and loss function

- Core functionality of all modern deep learning software frameworks

Automatic Differentiation / Backpropagation

- Example: simple multivariate function $f(x, y, z) = z(x + y)$
- Can write down function as a graph of primitive operations:



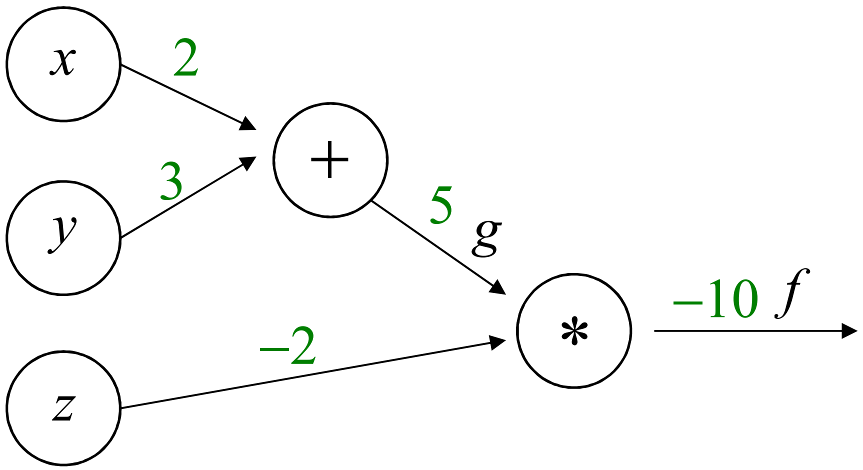
$$f(x, y, z) = h(z, g(x, y)) \quad h = "*", g = "+"$$

- We are interested in the partial derivatives at a specific point, for example, at $(x, y, z) = (2, 3, -2)$

$$\frac{\partial f}{\partial x}(2,3,-2) = -2, \qquad \frac{\partial f}{\partial y}(2,3,-2) = -2, \qquad \frac{\partial f}{\partial z}(2,3,-2) = 5$$

Forward Pass

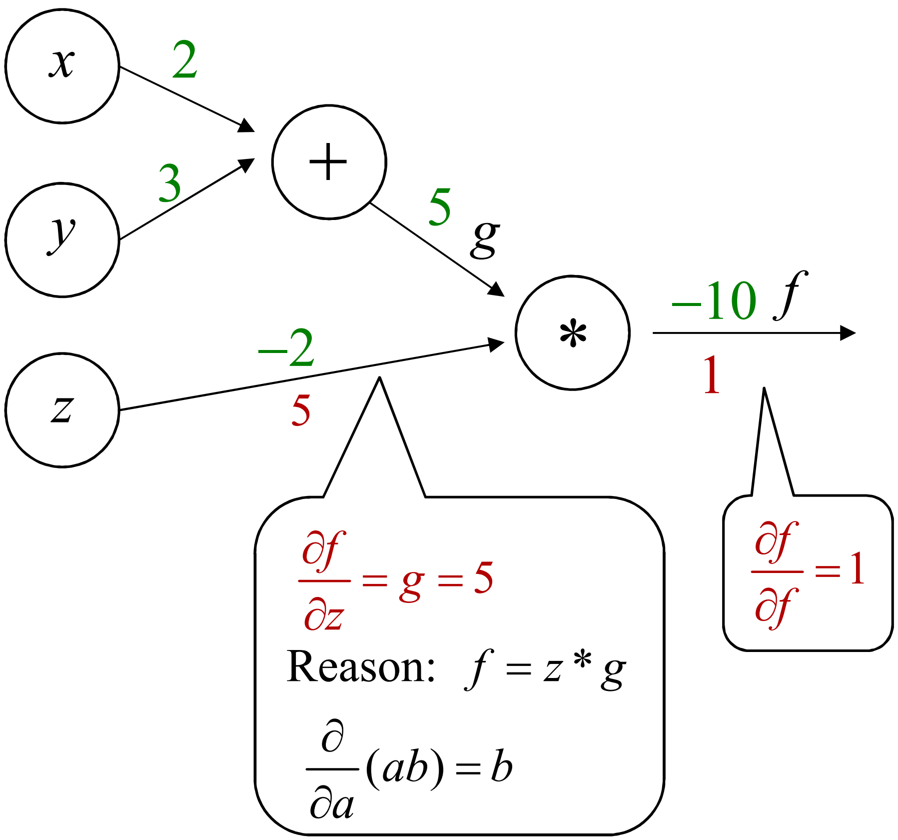
- We can compute the function value at position $(x, y, z) = (2, 3, -2)$ by a so-called forward pass:



- Computation „flows“ through the graph and is built up from primitive operations
- Outputs of inner nodes in the graph represent evaluations of subexpressions of the entire function

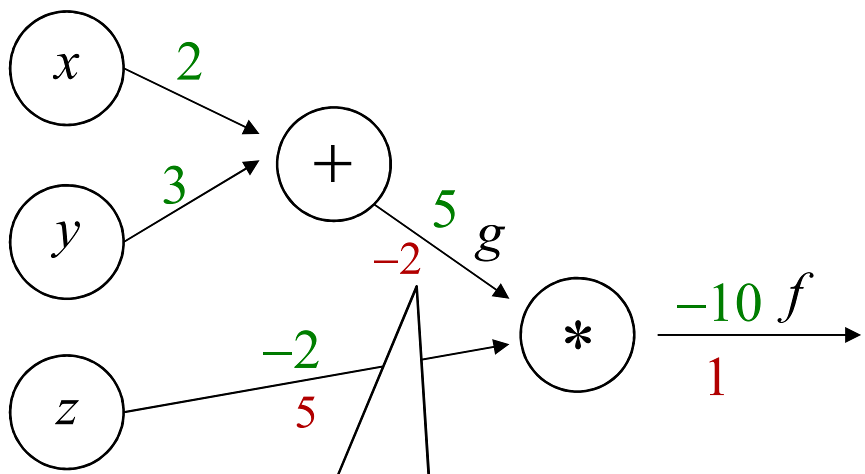
Example: Backpropagation

- Idea behind backpropagation: compute derivatives by propagating back through the graph, using local derivatives of primitive operations and chain rule
- Compute derivative of f with respect to output of every node



Example: Backpropagation

- Idea behind backpropagation: compute derivatives by propagating back through the graph, using local derivatives of primitive operations and chain rule
- Compute derivative of f with respect to output of every node



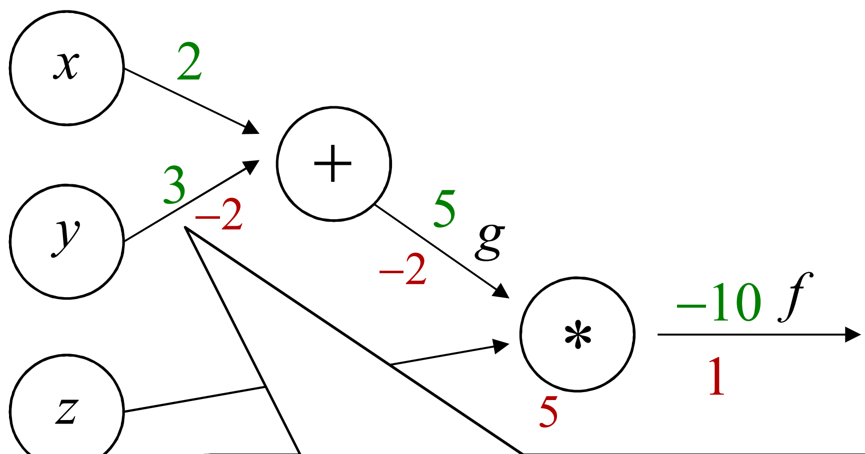
$$\frac{\partial f}{\partial g} = z = -2$$

Reason: $f = z * g$

$$\frac{\partial}{\partial b}(ab) = a$$

Example: Backpropagation

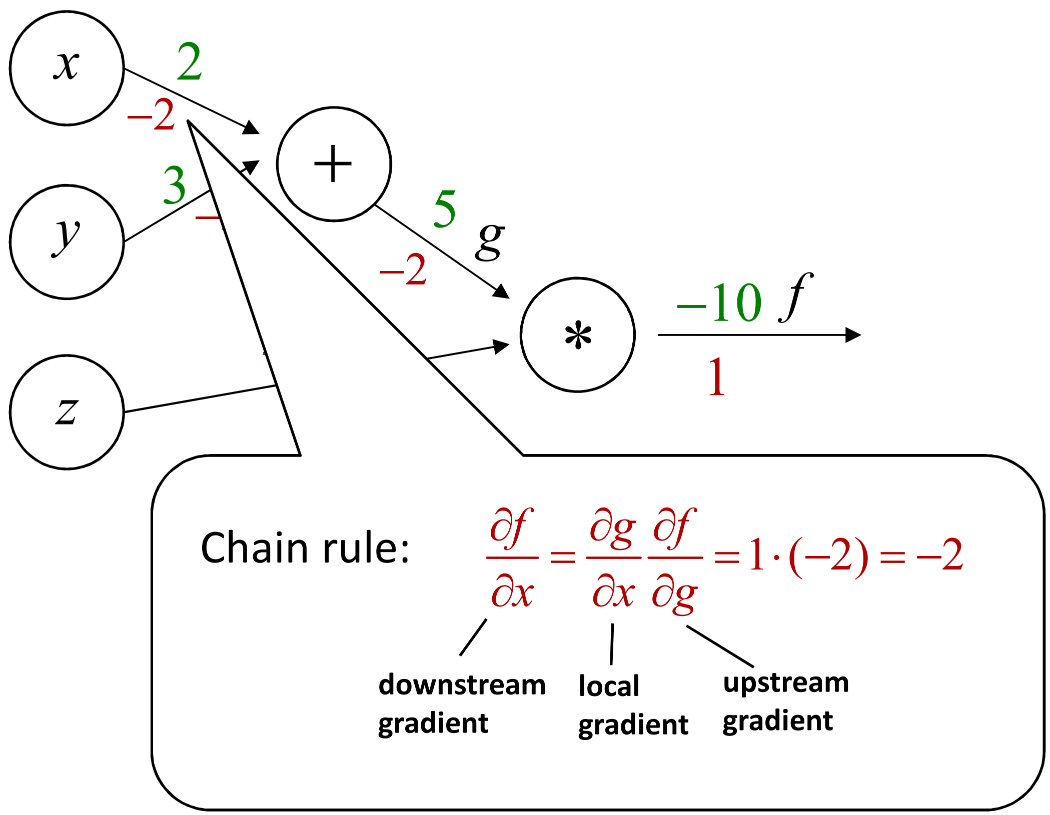
- Idea behind backpropagation: compute derivatives by propagating back through the graph, using local derivatives of primitive operations and chain rule
- Compute derivative of f with respect to output of every node



Chain rule: $$\underbrace{\frac{\partial f}{\partial y}}_{\text{downstream gradient}} = \underbrace{\frac{\partial g}{\partial y}}_{\text{local gradient}} \underbrace{\frac{\partial f}{\partial g}}_{\text{upstream gradient}} = 1 \cdot (-2) = -2$$

Example: Backpropagation

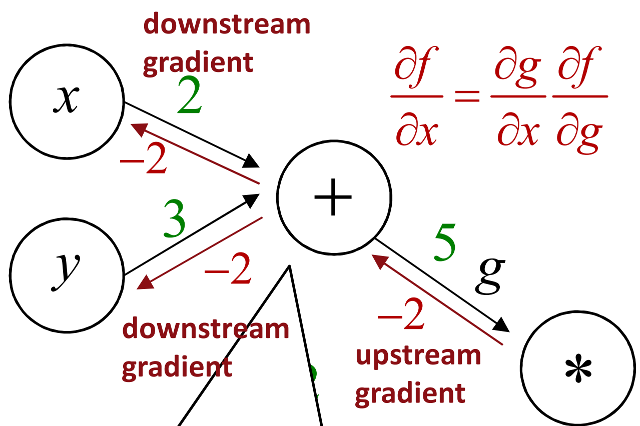
- Idea behind backpropagation: compute derivatives by propagating back through the graph, using local derivatives of primitive operations and chain rule
- Compute derivative of f with respect to output of every node



$$\frac{\partial f}{\partial x} = \frac{\partial g}{\partial x}\frac{\partial f}{\partial g} = 1 \cdot (-2) = -2$$

Example: Backpropagation

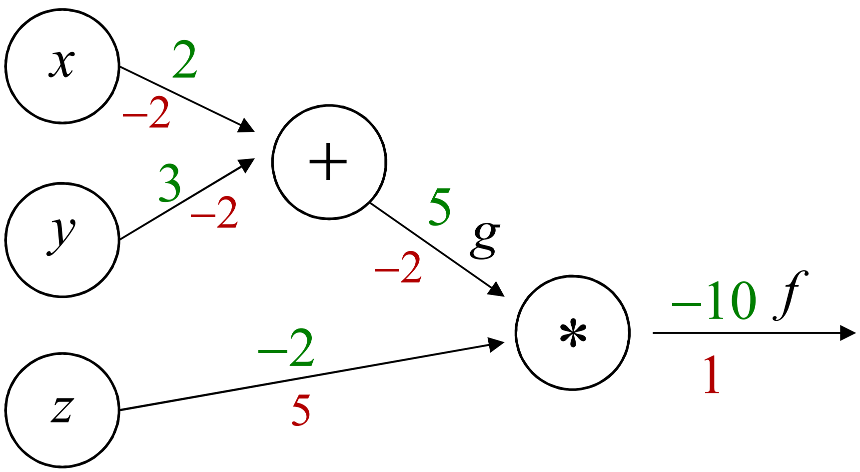
- Idea behind backpropagation: compute derivatives by propagating back through the graph, using local derivatives of primitive operations and chain rule
- Compute derivative of f with respect to output of every node



Backpropagation in the graph: start at the output. At every node, take the upstream gradient, multiply it with the local gradient, and propagate the result as downstream gradient

Example: Backpropagation

- Idea behind backpropagation: compute derivatives by propagating back through the graph, using local derivatives of primitive operations and chain rule
- Compute derivative of f with respect to output of every node



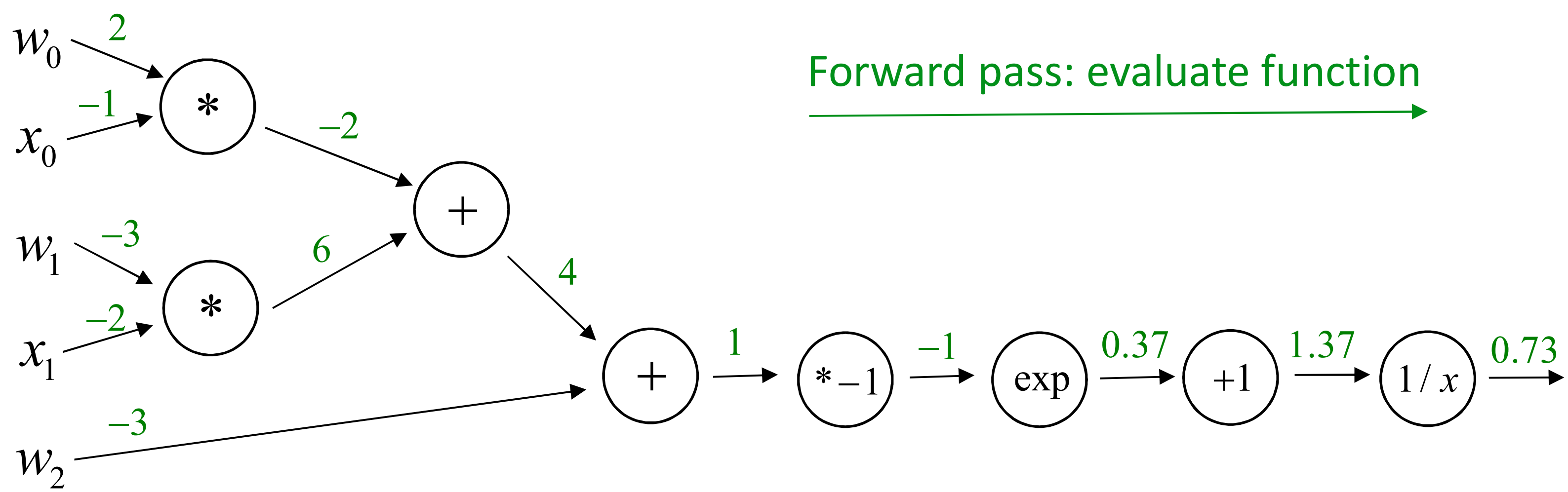
Result: all partial derivates have been computed. Gradient is vector of these derivatives

$$\frac{\partial f}{\partial x}(2,3,-2) = -2, \quad \frac{\partial f}{\partial y}(2,3,-2) = -2, \quad \frac{\partial f}{\partial z}(2,3,-2) = 5 \quad \nabla f(2,3,-2) = \begin{pmatrix} -2 \\ -2 \\ 5 \end{pmatrix}$$

Example: Backpropagation

- A slightly more complex example: backpropagation for the function

$$f(x_0, x_1, w_0, w_1, w_2) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$



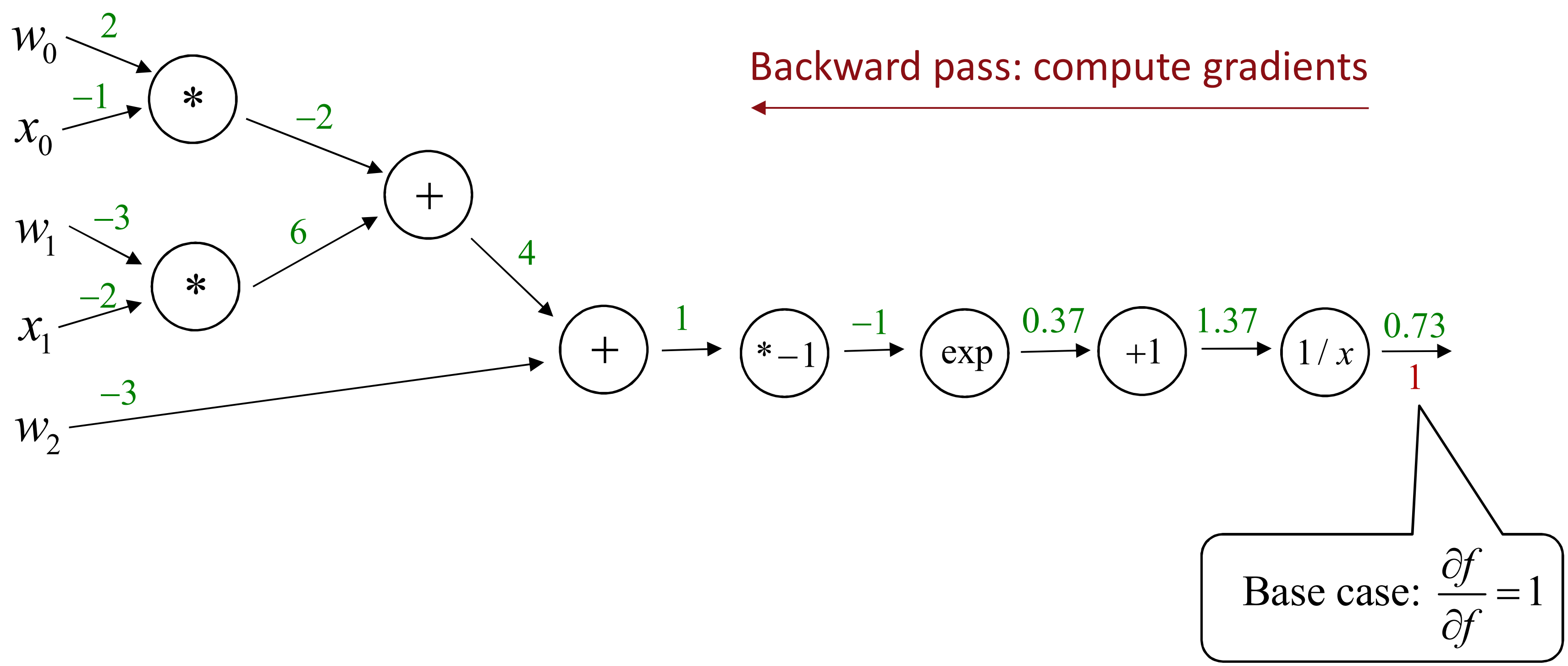
- Point at which function is evaluated and gradient computed:

$$x_0 = -1, \quad x_1 = -2, \quad w_0 = 2, \quad w_1 = -3, \quad w_2 = -3$$

Example: Backpropagation

- A slightly more complex example: backpropagation for the function

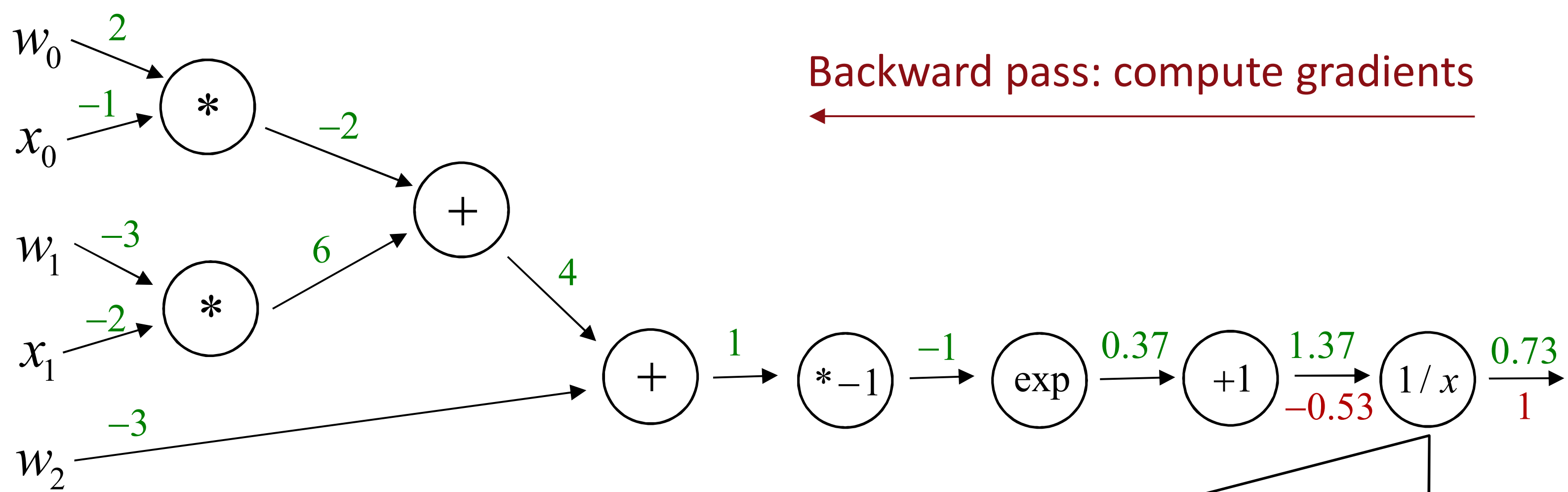
$$f(x_0, x_1, w_0, w_1, w_2) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$



Example: Backpropagation

- A slightly more complex example: backpropagation for the function

$$f(x_0, x_1, w_0, w_1, w_2) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$



Upstream gradient: 1

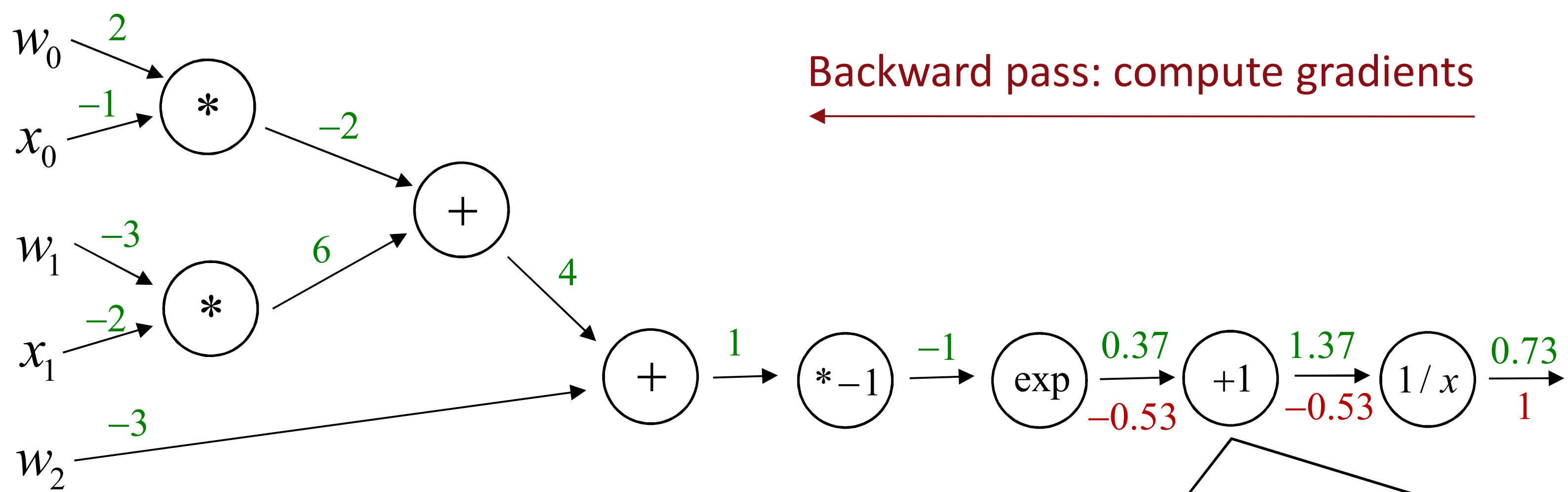
Local gradient: $\frac{\partial}{\partial x}\left[\frac{1}{x}\right] = -\frac{1}{x^2}$

Downstream gradient: $-\frac{1}{1.37^2} \cdot 1 = -0.53 \cdot 1 = -0.53$

Example: Backpropagation

- A slightly more complex example: backpropagation for the function

$$f(x_0, x_1, w_0, w_1, w_2) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$



Upstream gradient: -0.53

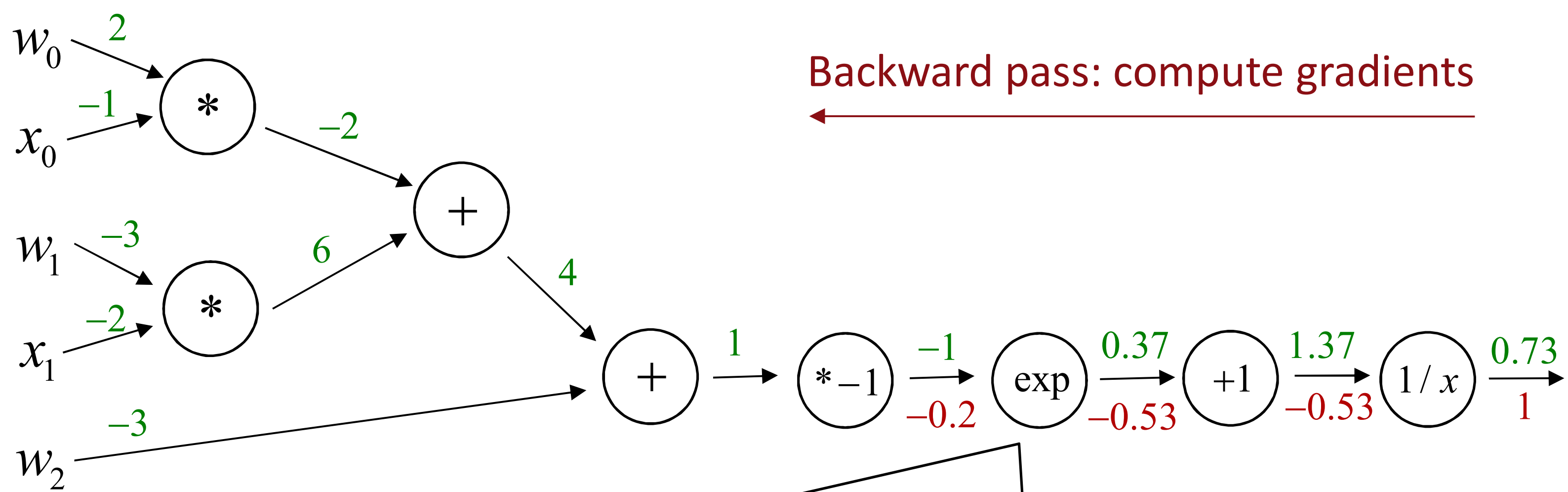
Local gradient: $\frac{\partial}{\partial x}[x+1] = 1$

Downstream gradient: $1 \cdot (\text{-}0.53) = -0.53$

Example: Backpropagation

- A slightly more complex example: backpropagation for the function

$$f(x_0, x_1, w_0, w_1, w_2) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$



Upstream gradient: -0.53

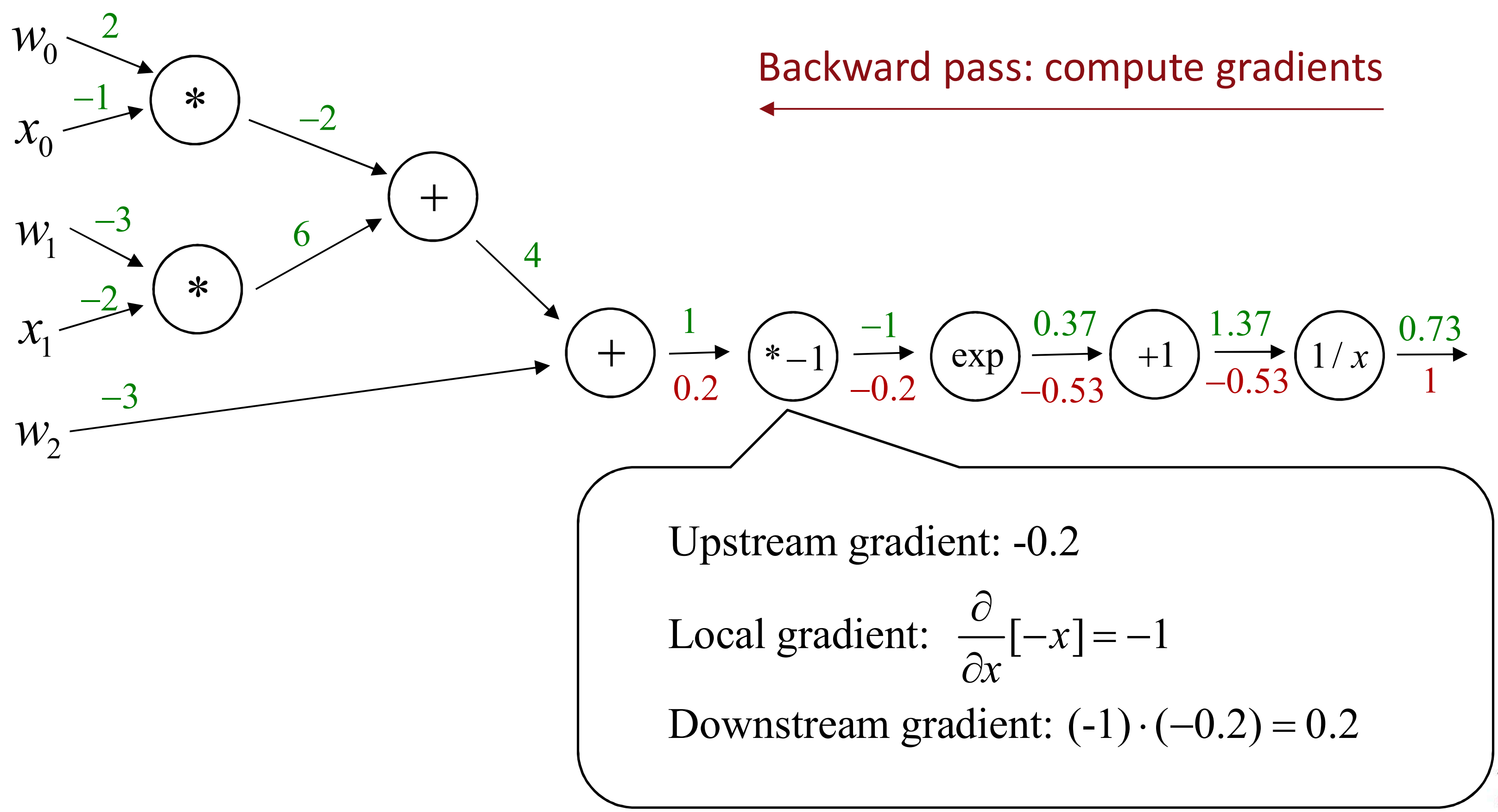
Local gradient: $\frac{\partial}{\partial x}\exp(x) = \exp(x)$

Downstream gradient: $\exp(\text{-}1) \cdot (-0.53) = -0.2$

Example: Backpropagation

- A slightly more complex example: backpropagation for the function

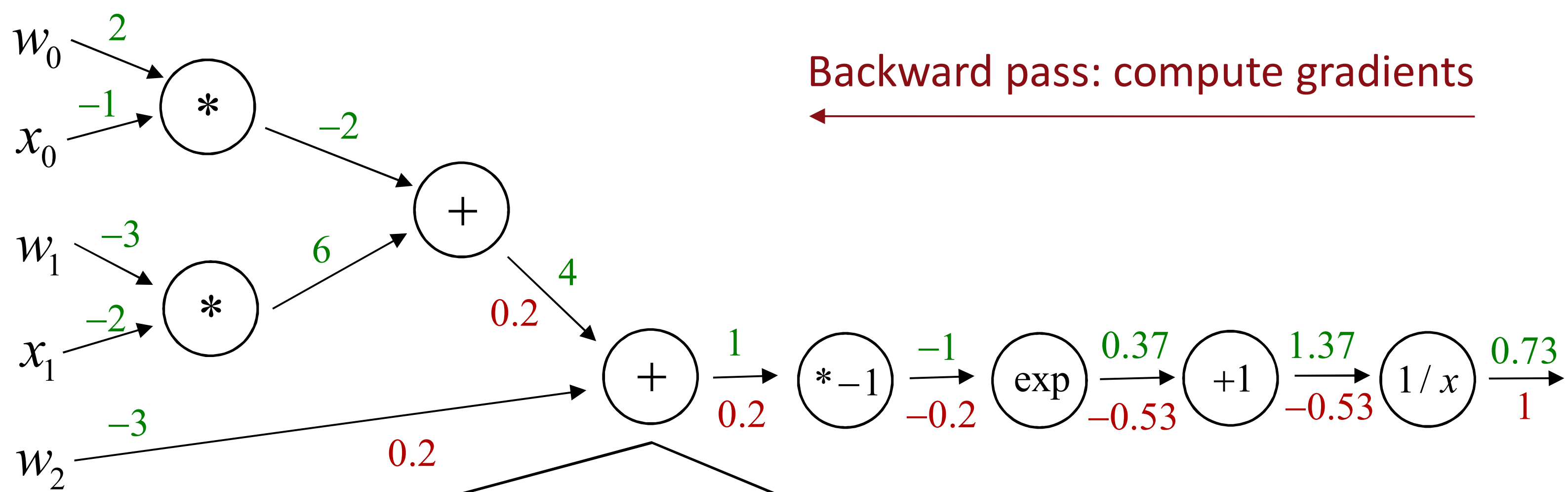
$$f(x_0, x_1, w_0, w_1, w_2) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$



Example: Backpropagation

- A slightly more complex example: backpropagation for the function

$$f(x_0, x_1, w_0, w_1, w_2) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$



Upstream gradient: 0.2

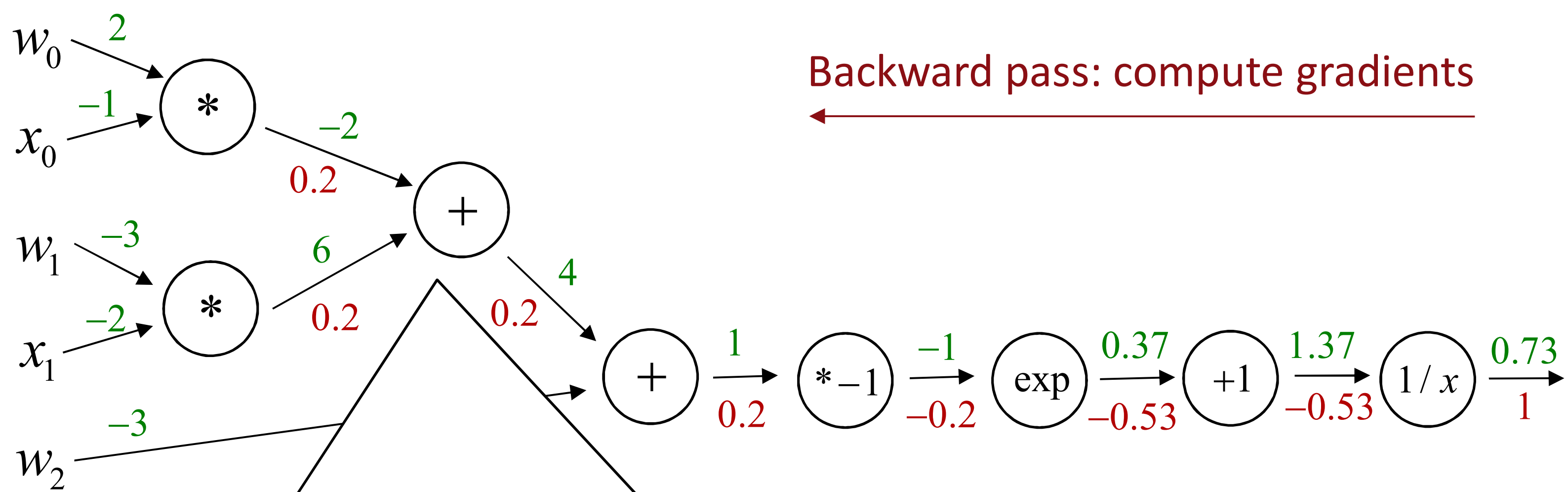
Local gradient: $\frac{\partial}{\partial x}[x + y] = 1, \quad \frac{\partial}{\partial y}[x + y] = 1$

Downstream gradients: $1 \cdot (0.2) = 0.2, \; 1 \cdot (0.2) = 0.2$

Example: Backpropagation

- A slightly more complex example: backpropagation for the function

$$f(x_0, x_1, w_0, w_1, w_2) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$



Upstream gradient: 0.2

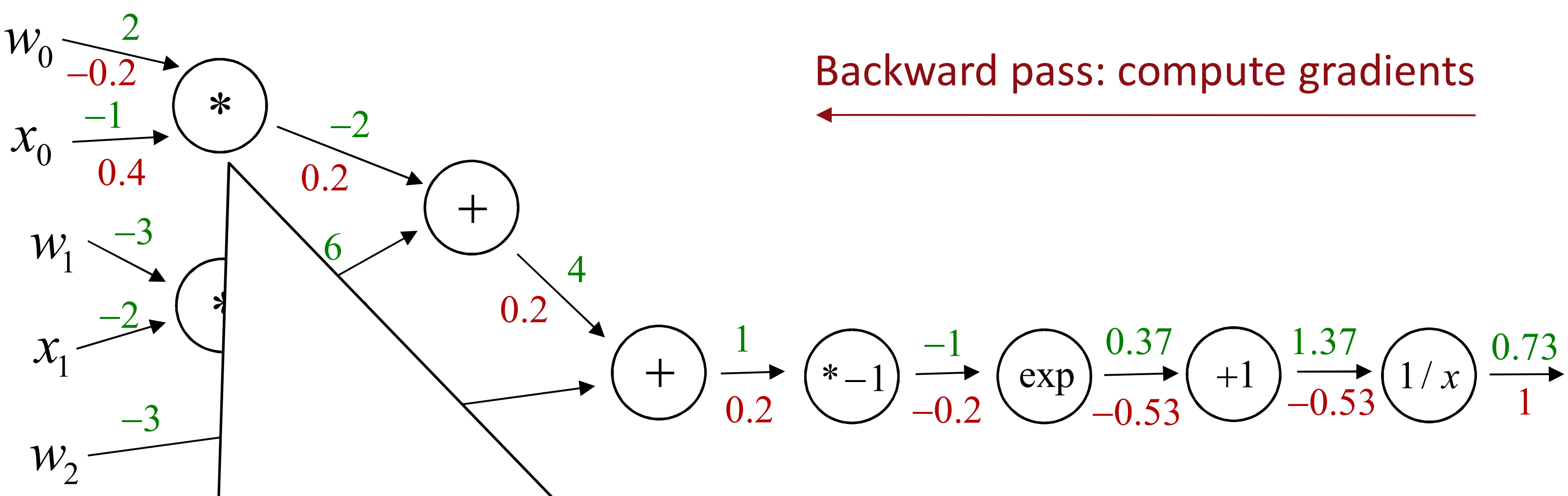
Local gradient: $\frac{\partial}{\partial x}[x + y] = 1, \quad \frac{\partial}{\partial y}[x + y] = 1$

Downstream gradients: $1 \cdot (0.2) = 0.2, \ 1 \cdot (0.2) = 0.2$

Example: Backpropagation

- A slightly more complex example: backpropagation for the function

$$f(x_0, x_1, w_0, w_1, w_2) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$



Upstream gradient: 0.2

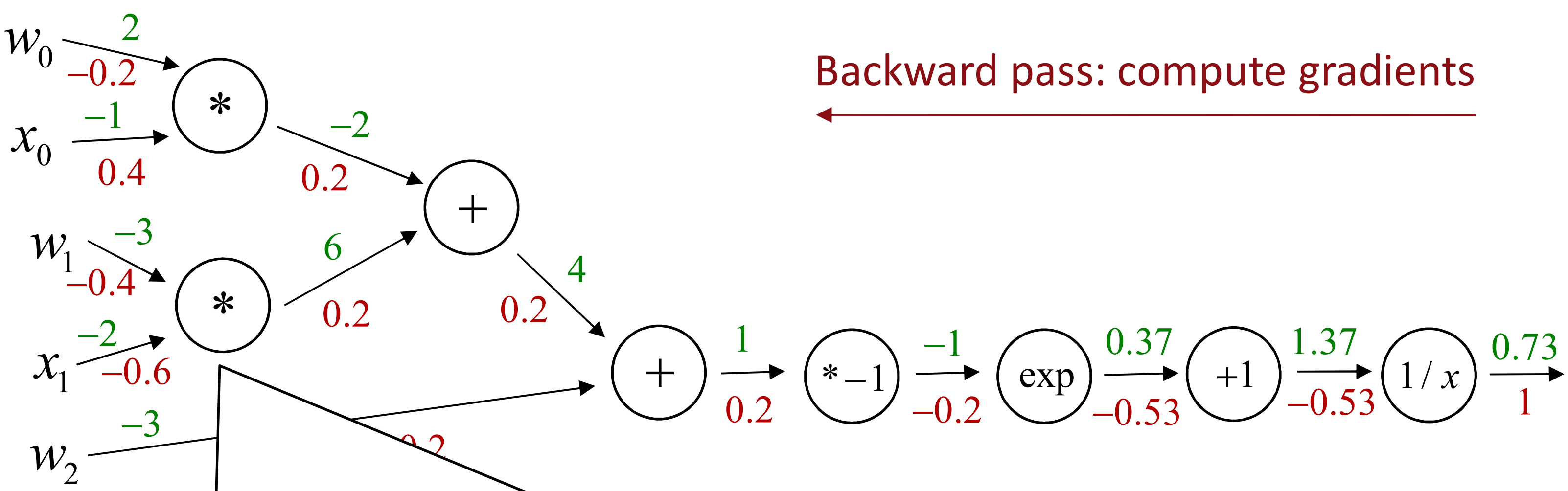
Local gradient: $\frac{\partial}{\partial x}[x \cdot y] = y, \quad \frac{\partial}{\partial y}[x \cdot y] = x$

Downstream gradients: $(\text{-}1) \cdot (0.2) = -0.2,\ \ 2 \cdot (0.2) = 0.4$

Example: Backpropagation

- A slightly more complex example: backpropagation for the function

$$f(x_0, x_1, w_0, w_1, w_2) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$



Upstream gradient: 0.2

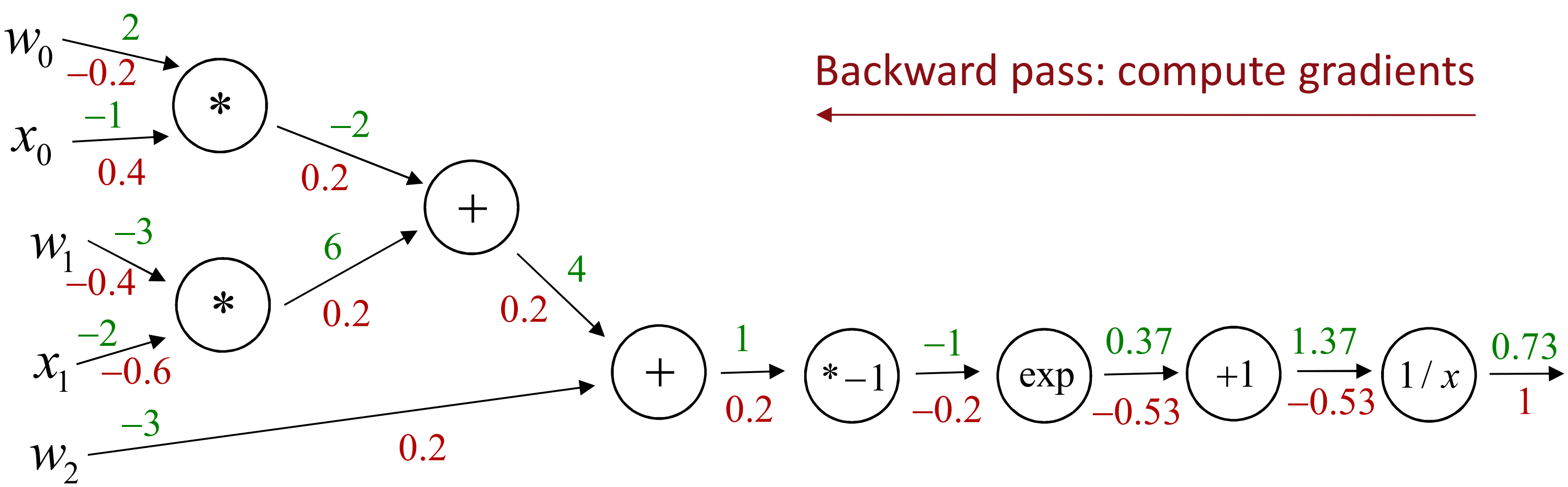
Local gradient: $\frac{\partial}{\partial x}[x \cdot y] = y, \quad \frac{\partial}{\partial y}[x \cdot y] = x$

Downstream gradients: $(-2) \cdot (0.2) = -0.4, \ (-3) \cdot (0.2) = -0.6$

Example: Backpropagation

- A slightly more complex example: backpropagation for the function

$$f(x_0, x_1, w_0, w_1, w_2) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$



- Gradient is vector of partial derivatives:

$$\nabla f(-1,-2,2,-3,-3) = \begin{pmatrix} 0.4 \\ -0.6 \\ -0.2 \\ -0.4 \\ 0.2 \end{pmatrix}$$

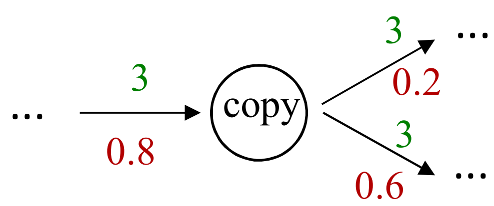
Note ordering of arguments

Backpropagation from Objective Function

- **Summary**: For neural network model, we need the gradient $\nabla L(\boldsymbol{\theta})$ of the objective function $L(\boldsymbol{\theta})$ in the model parameters $\boldsymbol{\theta}$
- The objective function $L(\boldsymbol{\theta})$ is a complex calculation: start with inputs, propagate through the layers of network using model parameters, and finally compute loss.
- This whole calculation can be cast into a computation graph as in the simple example, and derivatives with respect to model parameters (and also inputs) can be derived
- Note that the actual derivative calculation is not symbolic, but works with concrete numbers (gradient at a specific input point)
- Core functionality of deep learning frameworks such as Tensorflow or Pytorch: specificy neural network architecture and/or custom computational operations (custom losses, custom layers, custom regularizers, ...), framework will build computation graph and perform backpropagation
- Highly optimized implementations of backpropagation that run on the GPU

Backpropagation from Objective Function

- The provided examples only sketch the general idea, things that we did not cover include
  - what if a variable occurs in different places within a calculation? Introduce a „copy“ node that copies the variable in forward pass and adds the gradients in the backward pass



  - in practice, to improve computational efficiency we work directly with vectors/matrices/tensors and their respective derivatives rather than scalars (somewhat complicated)

Training a Feedforward Network

- Once we have the gradient, the neural network can be trained by stochastic gradient descent as discussed ealier

Gradient descent with momentum

1. $\boldsymbol{\theta}_0 = \text{randomInitialization()}$
2. $v_0 = 0$
3. for $i = 0, \ldots, N$:

   next update: negative slope plus discounted old update

   $\gamma \approx 0.9$

   $$v_{i+1} = -\eta \nabla L(\boldsymbol{\theta}_i) + \gamma v_i$$

   $$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i + v_{i+1}$$

4. return $\boldsymbol{\theta}_N$

$$\nabla L(\boldsymbol{\theta}) \approx \nabla \underbrace{\left(\frac{1}{m} \sum_{i=1}^{m} \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_{i_m}), y_{i_m}) + \lambda R(\boldsymbol{\theta}) \right)}_{\text{Use backpropagation to compute these terms}} \qquad \{i_1, \ldots, i_m\} \subset \{1, \ldots, n\}, \quad m << n$$

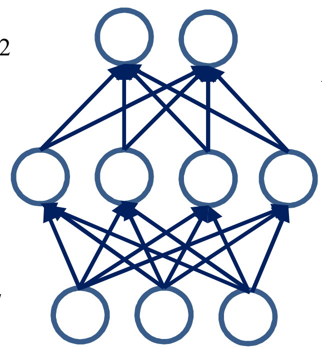
Summary Neural Networks So Far

- Multilayer perceptron: feedforward neural networks organized into stacked layers that are fully connected

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{W}_2 \mathbf{z}_1 + \mathbf{b}_2$$

$$\mathbf{W}_2 \in \mathbb{R}^{k \times k_1}, \quad \mathbf{b}_2 \in \mathbb{R}^{k}$$

$$\mathbf{z}_1 = \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)$$



$$\mathbf{W}_1 \in \mathbb{R}^{k_1 \times d}, \quad \mathbf{b}_1 \in \mathbb{R}^{k_1}$$

$$\mathbf{x} \in \mathbb{R}^{d}$$

Can be visualized as a regular graph structure

- Hyperparameters include number of layers, number of nodes per layers

- Loss functions for classification (cross-entropy) and regression (squared/absolute loss)
- Model parameters can be trained on data using stochastic gradient descent, gradients are automatically derived using backpropagation in compute graph