# Neural Networks

Lecture series „Machine Learning“

Niels Landwehr

Research Group „Data Science“
Institute of Computer Science
University of Hildesheim

# Agenda for Lecture

- Introduction: Neural networks
- Training neural networks
- Regularization for neural networks

# Agenda for Lecture

- Introduction: Neural networks
- Training neural networks
- Regularization for neural networks

# Recap: Linear Regression

- Review: Linear regression model for predicting real-valued targets

Function computes real-valued output based on instance $\mathbf{x} \in \mathbb{R}^M$

Function is **linear** in input: the output is obtained by multiplying each feature $x_m$ in the instance $\mathbf{x}$ by a corresponding model parameter $\theta_m$

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots \theta_M x_M$$

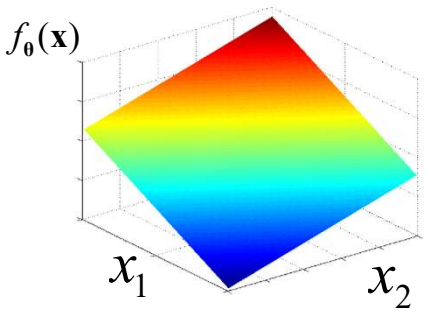
$$= \mathbf{x}^{\mathrm{T}} \boldsymbol{\theta}$$

Assuming a model parameter vector

$$\boldsymbol{\theta} = \begin{pmatrix} \theta_0 \\ \theta_1 \\ \ldots \\ \theta_M \end{pmatrix} \in \mathbb{R}^{M+1}$$

and constant attribute $x_0 = 1$

Visualization for *M=2*



# Recap: Linear Classification

- Review: Logistic regression model for predicting multiclass classification targets $y \in \{1,...,T\}$
- First, obtain a vector of $T$ class scores from $T$ linear models (rows of matrix $\mathbf{B}$)

$$f_{\boldsymbol{\theta}}^{(lin)}(\mathbf{x}) = \mathbf{B}\mathbf{x} + \mathbf{b} \qquad \mathbf{B} = \begin{pmatrix} \boldsymbol{\theta}_1^{\mathrm{T}} \\ ... \\ \boldsymbol{\theta}_T^{\mathrm{T}} \end{pmatrix} \in \mathbb{R}^{T \times M}, \mathbf{b} = \begin{pmatrix} b_1 \\ ... \\ b_T \end{pmatrix} \in \mathbb{R}^T, \quad \boldsymbol{\theta} = (\mathbf{B}, \mathbf{b})$$

- Then obtain class probabilities by transforming the class scores by a softmax function: Let $\mathbf{v} = f_{\boldsymbol{\theta}}^{(lin)}(\mathbf{x})$ denote the vector of class score, and define

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \mathrm{softmax}(\mathbf{v}) = \begin{pmatrix} \frac{e^{v_1}}{\sum_{t=1}^{T} e^{v_t}} \\ ... \\ \frac{e^{v_T}}{\sum_{t=1}^{T} e^{v_t}} \end{pmatrix}$$

Exponentiate each class score (to ensure that outputs are positive)

Divide by the sum of all exponentiated class scores (to ensure that outputs sum to one)

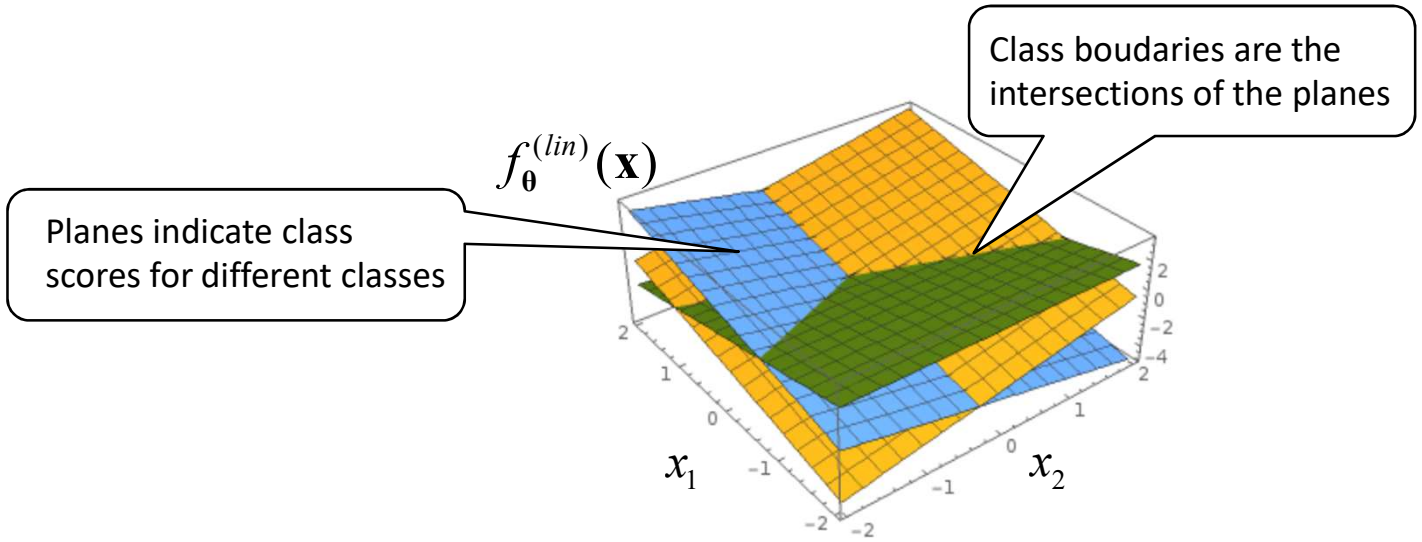
$$p(y = t \mid \mathbf{x}, \boldsymbol{\theta}) = f_{\boldsymbol{\theta}}(\mathbf{x})_t$$

$t$-th element in output $f_{\boldsymbol{\theta}}(\mathbf{x})$

# Recap: Linear Classification

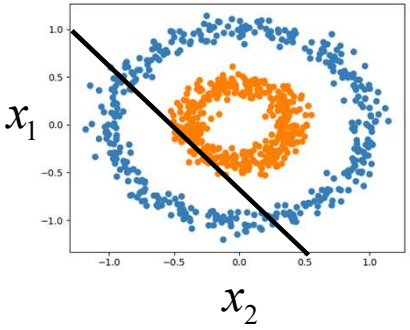
- The prediction for a novel test instance is then obtained as the most probable class, which is equivalent to the class that received the highest class score

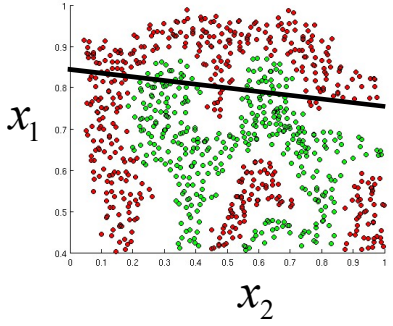
$$\hat{y} = \arg\max_{y} p(y \mid \mathbf{x}, \boldsymbol{\theta})$$



# 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)

- One way of going beyond linear models: nonlinear feature maps $\Phi : \mathbb{R}^M \rightarrow \mathbb{R}^{M'}$

$$\Phi(\mathbf{x}) = \begin{pmatrix} \Phi_1(\mathbf{x}) \\ \dots \\ \Phi_{M'}(\mathbf{x}) \end{pmatrix}$$

- For example, polynomial features, see lecture on regularization:

$$f_{\boldsymbol{\theta}}(\Phi(\mathbf{x})) = \theta_0 + \sum_{m=1}^{M} \theta_m x_m + \sum_{m=1}^{M} \sum_{l=1}^{M} \theta_{m,l} x_m x_l + \sum_{m=1}^{M} \sum_{l=1}^{M} \sum_{k=1}^{M} \theta_{m,l,k} x_m x_l x_k \quad (\text{degree } d = 3)$$

# Representation Learning

- Neural networks: instead of predefining a nonlinear feature map, „stack“ several modeling layers on top of each other to define an overall nonlinear model
  - Lower layers learn simple features from the data
  - Intermediate layers build on these simple features to form more complex features
  - Final layer performs actual prediction (classification or regression)

- Model can be trained **end-to-end**: all layers are jointly optimized to minimize a loss function on the predictions
- Usually leads to better results than pre-defined nonlinear feature maps (at least if enough data is available...)

- Also called **feature learning** or **representation learning**

- For many application domains, neural networks with many layers are the state-of-the-art solution. These are also called **deep learning** approaches.

# From Linear Models to Neural Networks

- Let's look at multiclass classification. A simple linear model is:

Output: vector of class probabilities

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

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

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

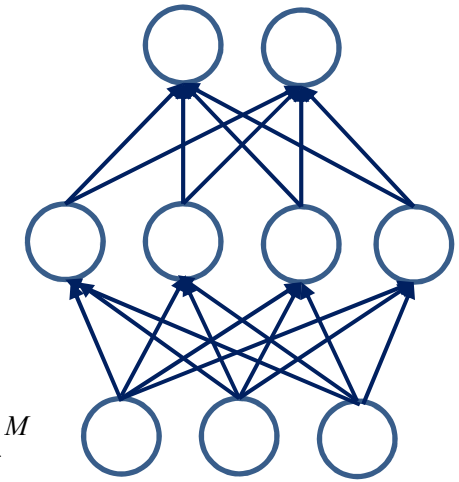
second linear model

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

first linear model

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

$\sigma(z)$ nonlinear function



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

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

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

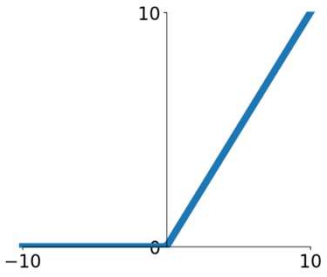
# Nonlinear Activation Function

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

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

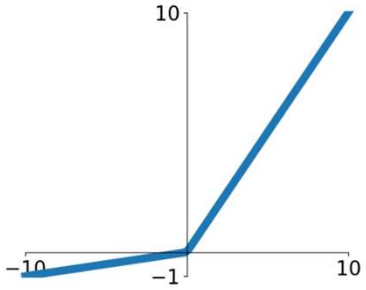
- Different nonlinear activation functions are used (non-exhaustive list):

**ReLU: Rectified linear unit**



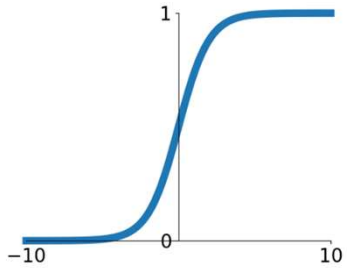

$$\sigma(z) = \max(0, z)$$

Leaky ReLU



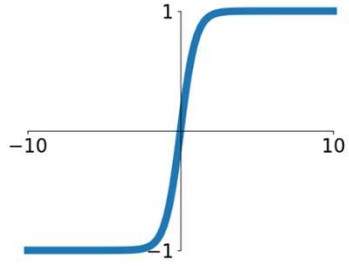

$$\sigma(z) = \max(0.1z, z)$$

Sigmoid




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

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 for classification)**

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

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

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

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

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

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

# Multilayer Perceptrons as Graphs

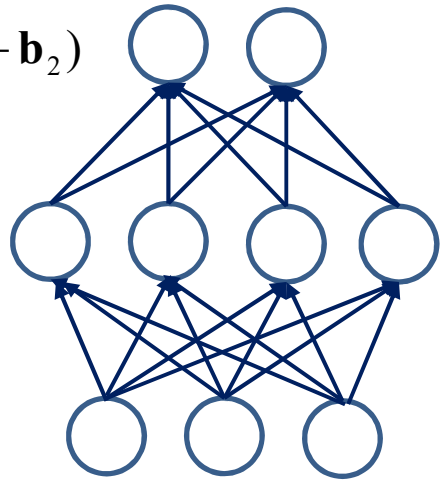
- A Multilayer Perceptron can be visualized as a directed acyclic graph (DAG)
- Graph is organized into layers that are fully connected
  - There are *D+2* layers, called „input“, „hidden“ (*D* layers) and „output“
  - Input layer has *M* nodes, hidden layer *i* has $k_i$ nodes, output has *T* nodes

Example: $M = 3, \quad T = 2, \quad k_1 = 4, \quad D = 1$

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

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

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



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

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

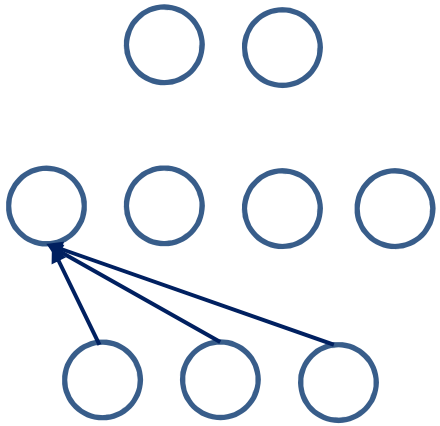
# Multilayer Perceptrons as Graphs

- 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 $\mathbf{b}_i$:

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

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

# Agenda for Lecture

- Introduction: Neural networks
- Training neural networks
- Regularization for neural networks

# Cross-Entropy Loss for Training

- Multilayer perceptron: function $f_{\boldsymbol{\theta}} : \mathbb{R}^M \rightarrow \mathbb{R}^T$ with parameters $\boldsymbol{\theta} = \{\mathbf{W}_i, \mathbf{b}_i\}_{i=1}^{D+1}$
- Classification scenario: output $f_{\boldsymbol{\theta}}(\mathbf{x})$ is a vector of probabilities for the *T* classes

- Assume training data $\mathcal{D} = \{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_N, y_N)\}$, with $y_n \in \{1, ..., T\}$
- We define the **cross-entropy loss function** by

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

where $p(y = y_n \mid \mathbf{x}_n, \boldsymbol{\theta})$ is the $y_n$-th element in the class probability vector $f_{\boldsymbol{\theta}}(\mathbf{x}_n)$

- As usual, the overall loss is the average over individual training instances:

$$L(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_n), y_n)$$

- Equivalent to maximizing the conditional likelihood $p(\mathbf{y} \mid \mathbf{x}, \boldsymbol{\theta}) = \prod_{n=1}^{N} p(y_n \mid \mathbf{x}_n, \boldsymbol{\theta})$
- Regularization can be added, see below

# Neural Networks For Regression

- The outlined multilayer perceptron can be easily adapted for regression
  - Using $T = 1$ and removing the final softmax operation yields a model $f_{\boldsymbol{\theta}} : \mathbb{R}^M \to \mathbb{R}$ that directly returns a continuous prediction
  - As loss function, standard regression losses such as squared loss or absolute loss can be used:

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

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

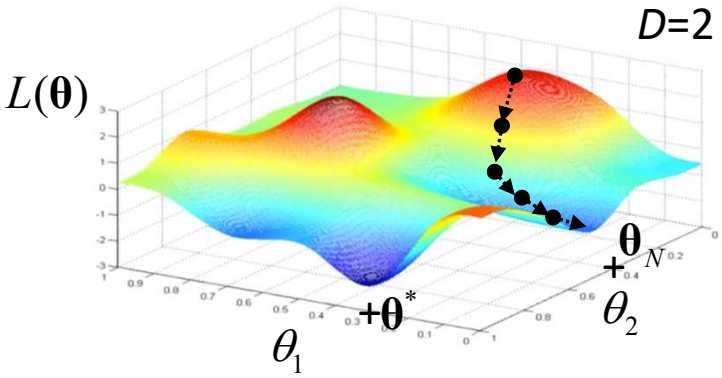
- Leads to a flexible non-linear model for solving regression tasks

# Parameter Optimization

- Learning a multilayer perceptron from data: solving optimization problem (see below for adding a regularizer)

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

- Generally, neural networks are optimized with gradient descent-based methods (see lecture on linear regression):




**Gradient descent algorithm**

1. $\boldsymbol{\theta}_0 = \text{randomInitialization}()$
2. for $i = 0,...,i_{max}$:
3. $\quad \boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta \nabla L(\boldsymbol{\theta}_i)$
4. $\quad$ if $L(\boldsymbol{\theta}_i) - L(\boldsymbol{\theta}_{i+1}) < \epsilon$ :
5. $\quad\quad$ return $\boldsymbol{\theta}_{i+1}$
6. raise Exception("Not converged in $i_{max}$ iterations")

- **However, for neural networks loss is not a convex function in model parameters**
- Gradient descent only finds local optimum, but not a huge problem in practice

# Stochastic Gradient Descent

- Gradient descent: in each iteration, change the parameter vector $\boldsymbol{\theta}$ in the direction of the negative gradient of the loss function
- The loss function and thus the gradient are defined by a sum over the data set

$$\nabla L(\boldsymbol{\theta}) = \left( \frac{1}{N} \sum_{n=1}^{N} \nabla \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_n), y_n) \right)$$

- Computing the gradient scales linearly with the size of the data set. If the data set is large, even computing a single parameter update will be expensive
- **Idea: do not compute gradient on all of the data, but on (small) subset of $J$ instances**:

$$\nabla L(\boldsymbol{\theta}) \approx \nabla L_{\mathcal{B}}(\boldsymbol{\theta}) = \nabla \left( \frac{1}{J} \sum_{j=1}^{J} \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_{n_j}), y_{n_j}) \right) \qquad \mathcal{B} = \{n_1, ..., n_J\} \subset \{1, ..., N\}, \quad J << N$$

- Gradient approximated on a small random subset: noisy/stochastic gradient
- This is called (minibatch) **Stochastic Gradient Descent**, or **SGD**
- The subset $\{(\mathbf{x}_{n_1}, y_{n_1}), ..., (\mathbf{x}_{n_J}, y_{n_J})\}$ of data is called a **minibatch**

# Stochastic Gradient Descent

- What happens if we compute the gradient on a random subset?
  - Individual gradients will be „noisy“ in the sense that they have a strong random component
  - However, in expectation, gradient will still be identical to „full“ gradient computed on the entire data set:

$$\mathbb{E}\left[\left[\nabla\left(\frac{1}{J}\sum_{j=1}^{J}\ell(f_{\boldsymbol{\theta}}(\mathbf{x}_{n_j}),y_{n_j})\right)\right]\right]=\nabla\left(\frac{1}{N}\sum_{n=1}^{N}\ell(f_{\boldsymbol{\theta}}(\mathbf{x}_n),y_n)\right)$$

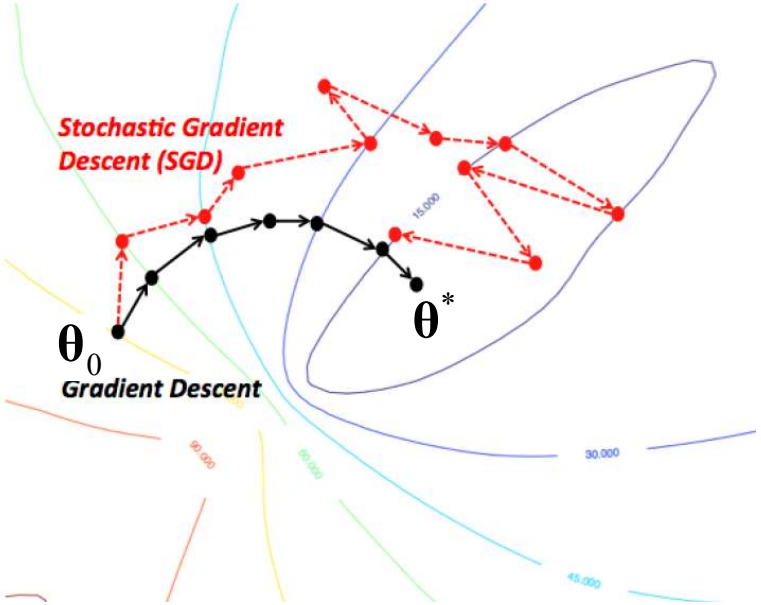
Expectation over draws of random subsets $\mathcal{B}=\{n_1,\ldots,n_J\}\subset\{1,\ldots,N\}$

  - This means that in expectation gradient descent will move towards minimum

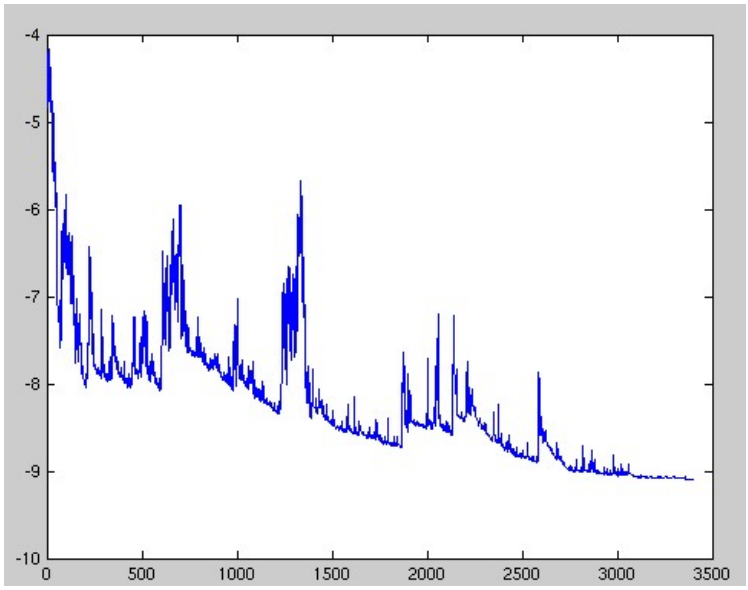
- Typically, small subsets are used, e.g. *J=32,64,128*, or *256* instances (while size of data set can be 1.000.000+ instances)
- Can be even a single instance per batch, *J=1*
- Gradient steps orders of magnitude faster, can run many more steps

# Stochastic Gradient Descent

- Trajectory to the minimum will be more noisy/random
- Loss on all of the data not guaranteed to fall at every SGD step, but will still fall on average




Path in parameter space SGD vs full gradient descent



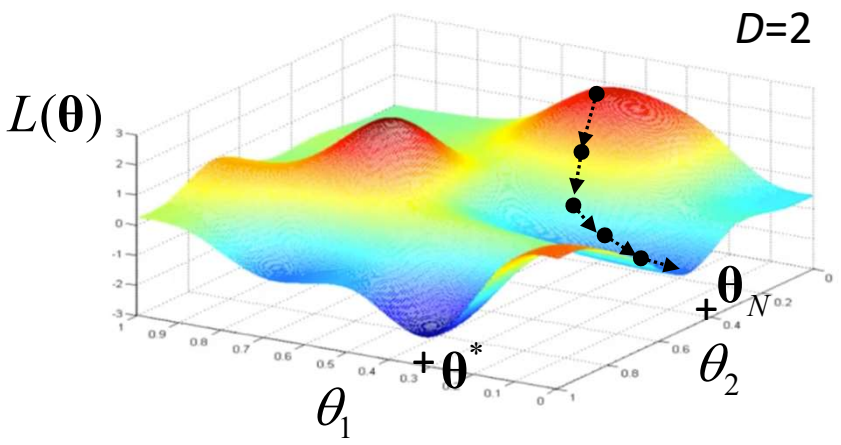

Loss on complete data set over SGD iterations

# Stochastic Gradient Descent Algorithm

- Assuming a batch size of $J$, the stochastic gradient descent algorithm looks as follows:

**Stochastic gradient descent algorithm**

1. $\boldsymbol{\theta}_0 = \text{randomInitialization}()$
2. for $i = 0,...,i_{max}$:
3. Select random minibatch $\mathcal{B} \subset \{1,...,N\}$
4. $\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \eta \nabla L_{\mathcal{B}}(\boldsymbol{\theta}_i)$
5. return $\boldsymbol{\theta}_{i_{max}}$

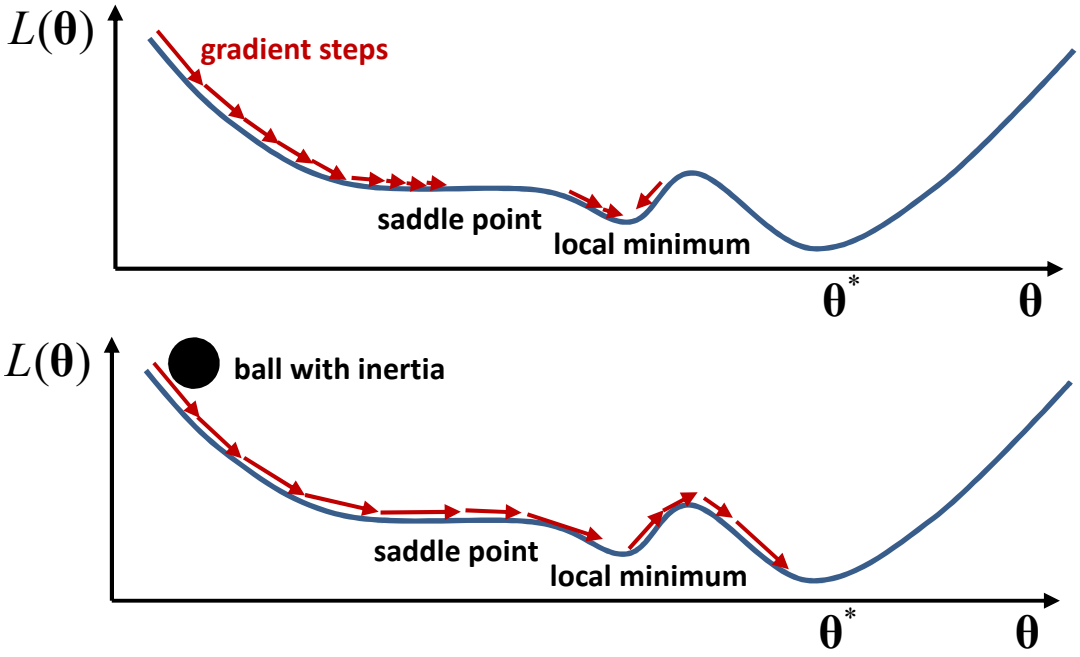



- Note that for stochastic gradient descent, there is no obvious stopping criterion: loss over minibatches fluctuates strongly, and even loss over all data varies (see last slide)

- Run for a fixed number of steps, or monitor performance on external validation set (see below), or use more advanced heuristics for deciding when to stop

# Momentum in Stochastic Gradient Descent

- One way of improving convergence of SGD is to use so-called **momentum**




**Without momentum:**
step is local negative slope.
Gets stuck at saddle point
or in local minimum

**With momentum:**
Imagine rolling a ball (with inertia) down the loss surface.
Inertia carries the ball through saddle point and over local minimum

- Momentum in SGD: take a step into the direction of the local negative slope plus an exponentially decaying average of earlier steps

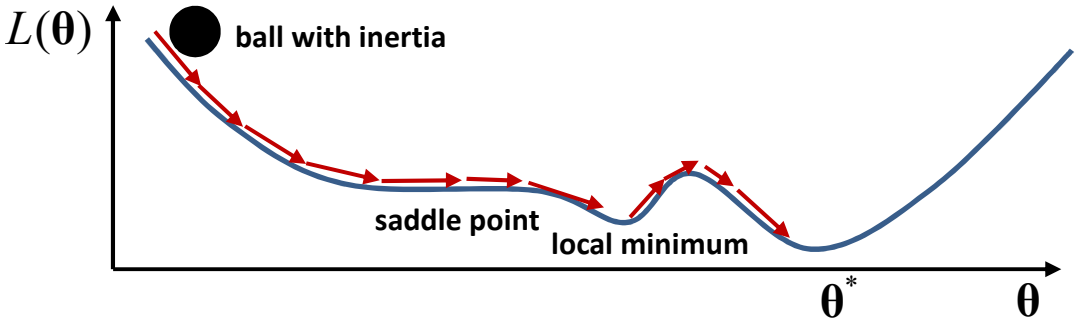
# Momentum in Stochastic Gradient Descent

- Momentum in gradient descent: take a step into the direction of the local negative slope plus an exponentially decaying average of earlier steps
- In practice, almost always use SGD with momentum

**Gradient descent with momentum**

1. $\boldsymbol{\theta}_0 = \text{randomInitialization}()$
2. $v_0 = 0$
3. for $i = 0, \ldots, i_{max}$:
   - $v_{i+1} = -\eta \nabla L(\boldsymbol{\theta}_i) + \gamma v_i$ (e.g. $\gamma = 0.9$)
   - $\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i + v_{i+1}$
4. return $\boldsymbol{\theta}_{i_{max}}$

next update: negative slope plus discounted old update




**With momentum:**
Imagine rolling a ball (with inertia) down the loss surface. Inertia carries the ball through saddle point and over local minimum

# Optimization: Gradient

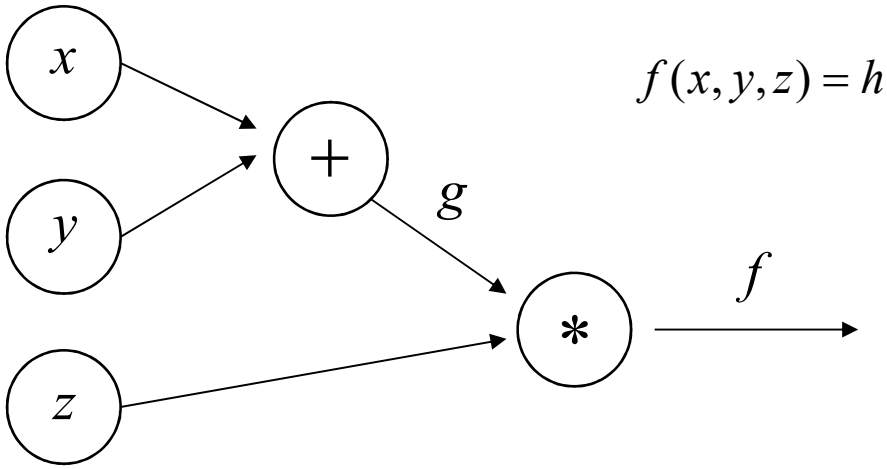
- For optimization, need to compute the gradient $\nabla L_{\mathcal{B}}(\boldsymbol{\theta})$

- 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 re-derive gradient

- Instead: derive gradient algorithmically using **automatic differentiation**, also known (in this specific case) as **backpropagation**
  - Automatic differentiation is a general, flexible approach to compute the gradient of any function $L : \mathbb{R}^M \to \mathbb{R}$ at a specific point $\boldsymbol{\theta} \in \mathbb{R}^M$
  - Applying automatic differentiation to the loss function $L(\boldsymbol{\theta})$ enables us to compute gradients automatically for any model architecture and loss function
  - Such automatic differentiation routines are a core functionality of all modern neural network 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:




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

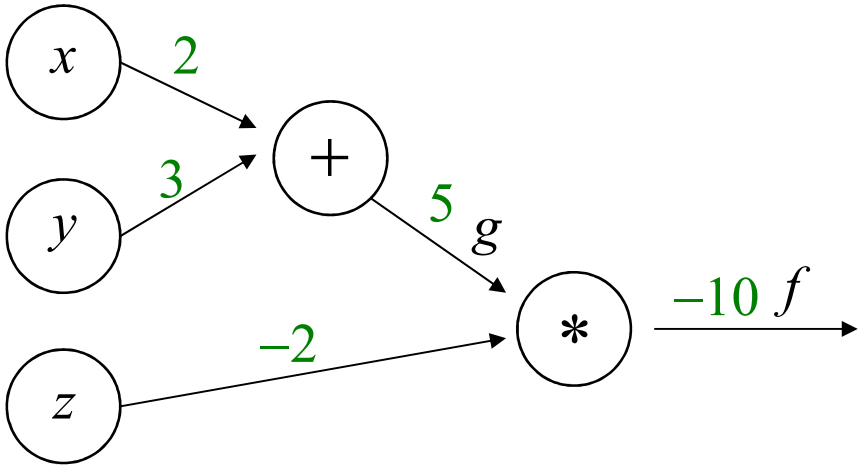
- We are interested in the gradient of the function, that is, the vector of partial derivatives at a specific point

- For example, at $x = 2, y = 3, z = -2$

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

# Forward Pass

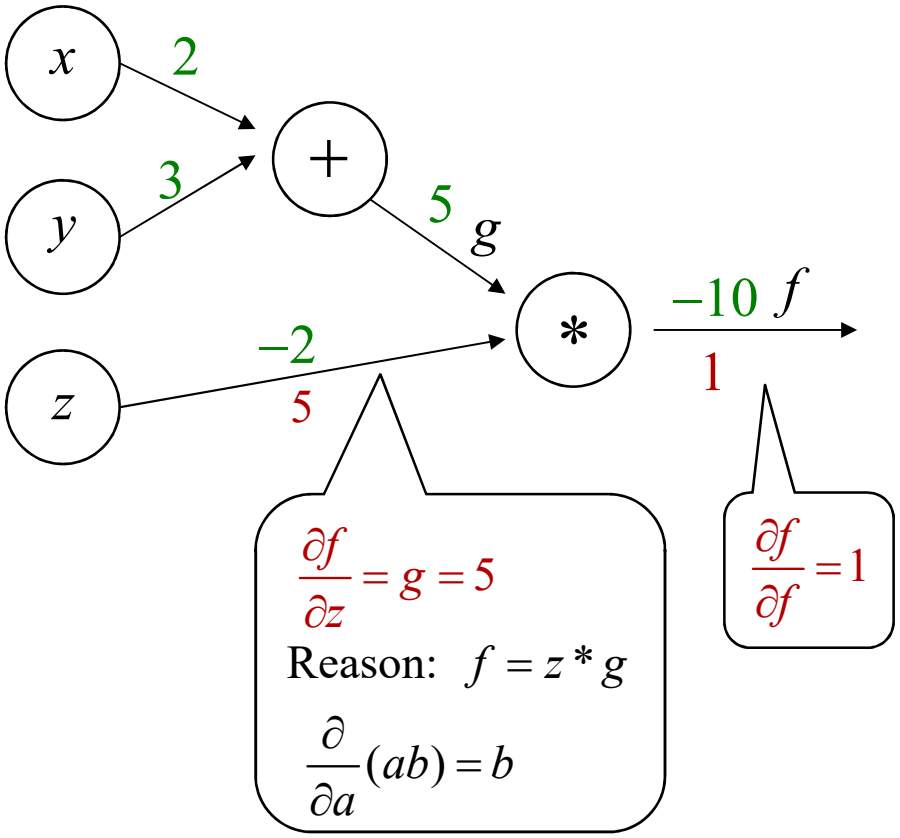
- We can compute the function value at position (2,3,-2) by a forward pass:




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

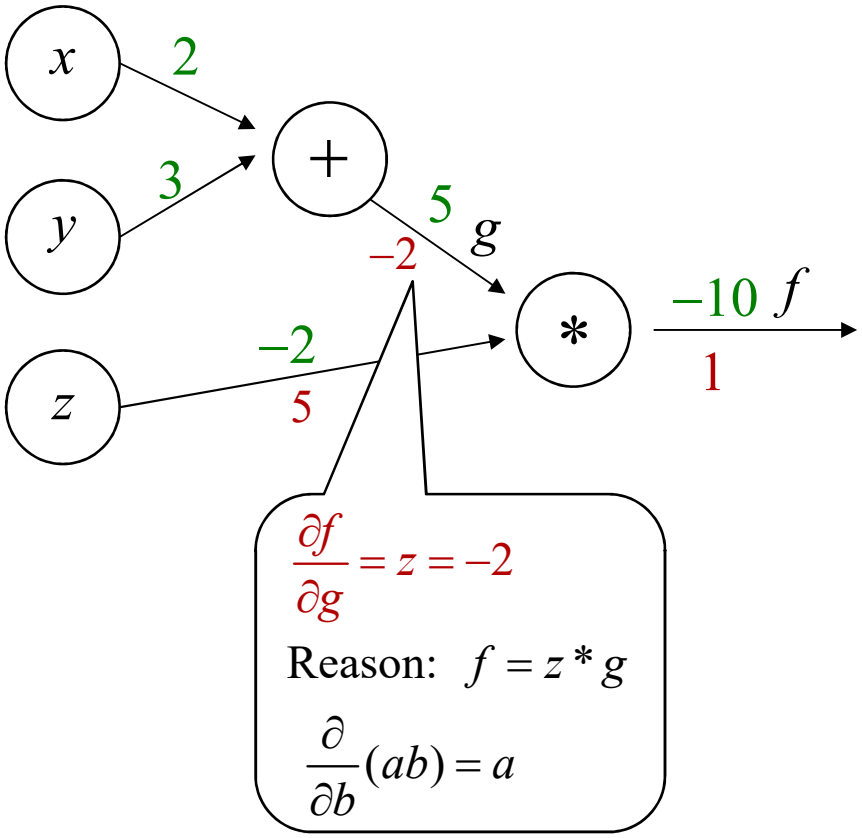
# Example: Backpropagation

- 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



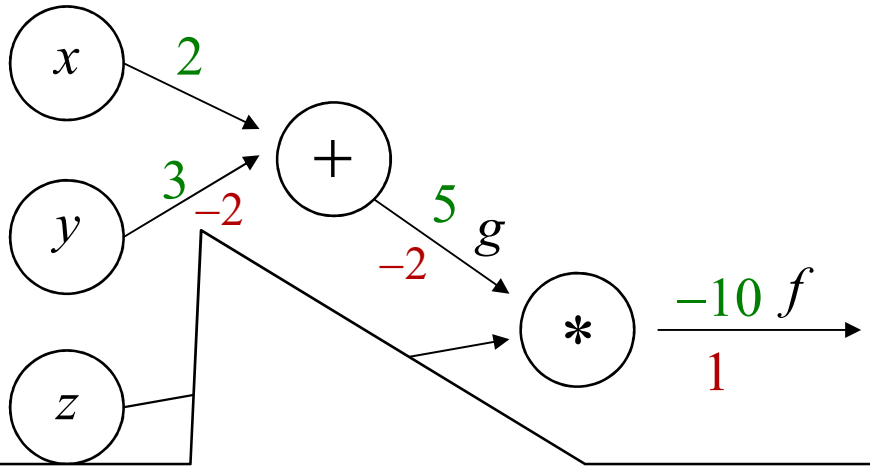
# Example: Backpropagation

- 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



# 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




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

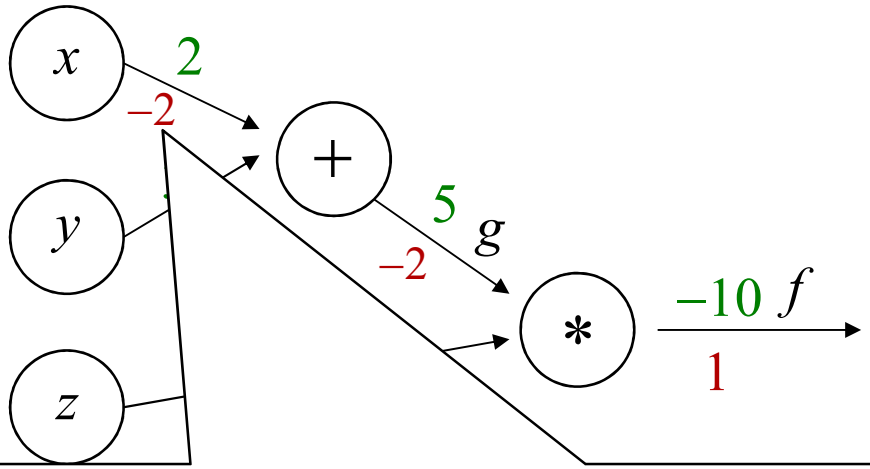
downstream gradient | local gradient | upstream gradient

Chain rule: $f = h(z, g(x,y))$

$$\frac{\partial h}{\partial y}(x,y,z) = \frac{\partial h}{\partial g}(z, g(x,y))\frac{\partial g}{\partial y}(x,y)$$

# 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




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

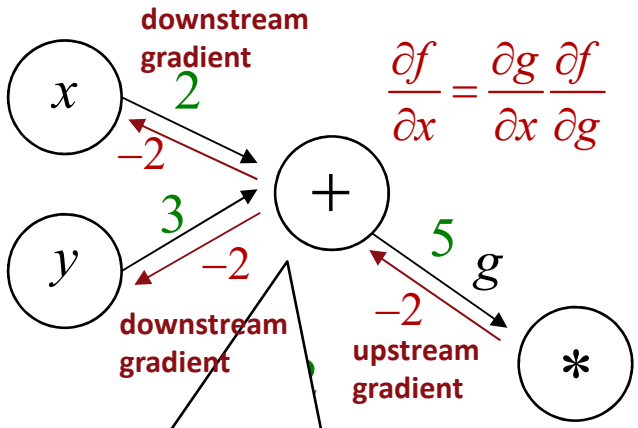
downstream gradient | local gradient | upstream gradient

Chain rule: $f = h(z, g(x, y))$

$$\frac{\partial h}{\partial x}(x, y, z) = \frac{\partial h}{\partial g}(z, g(x, y))\frac{\partial g}{\partial x}(x, y)$$

# Example: Backpropagation

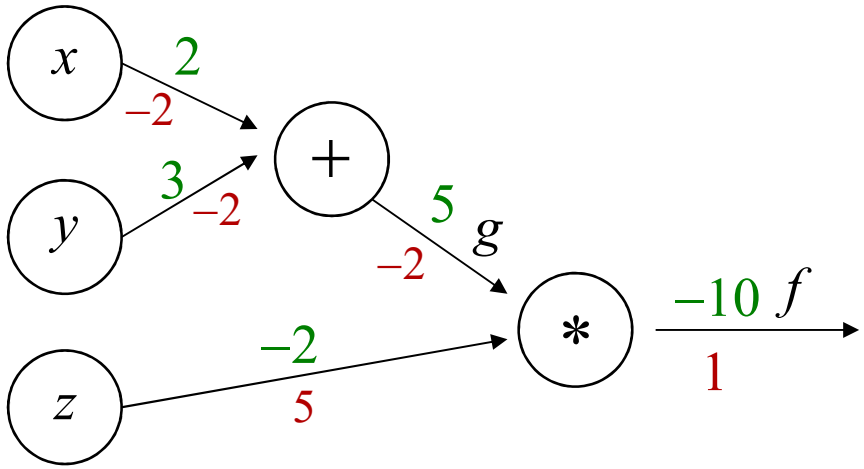
- Idea behind backpropagation: compute derivatives by propagating back through the graph, using local derivatives of primitive operations and chain rule
- Compute derivative of $f$ with respect to output of every node




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



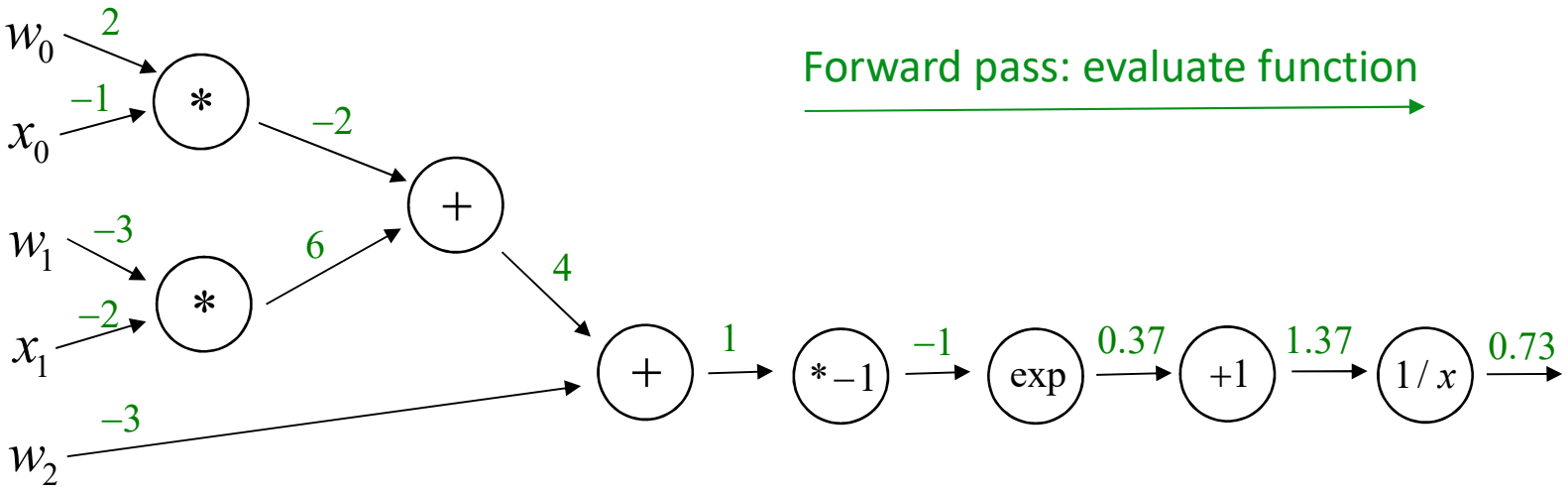

**Result: all partial derivates have been computed. Gradient is vector of these derivatives**

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

# Example: Backpropagation

- A slightly more complex example: backpropagation for the function

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



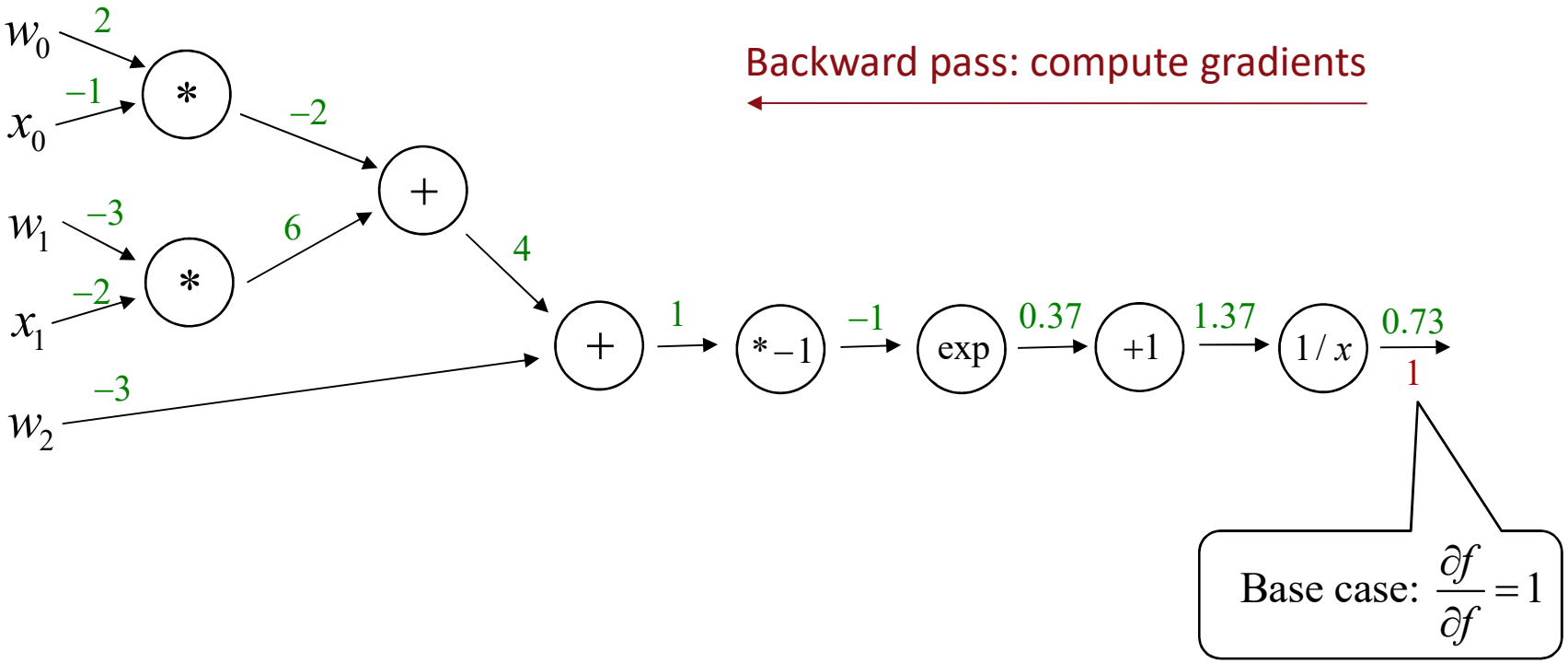

- Point at which function is evaluated and gradient computed:

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

# Example: Backpropagation

- A slightly more complex example: backpropagation for the function

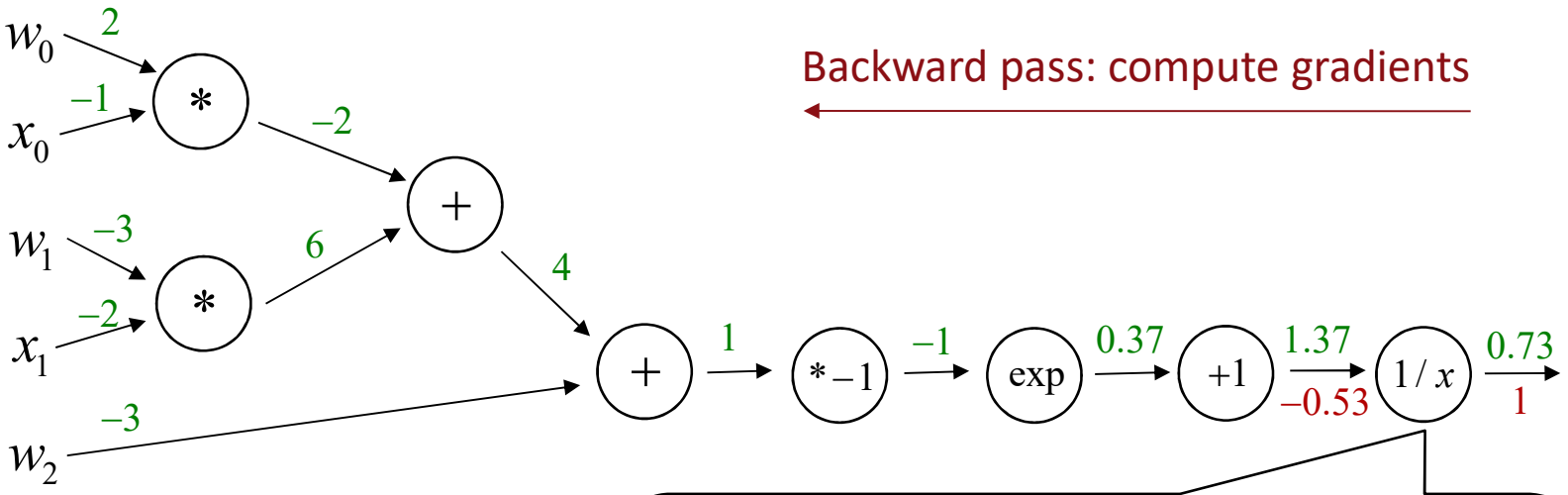
$$f(x_0, x_1, w_0, w_1, w_2) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$



# 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

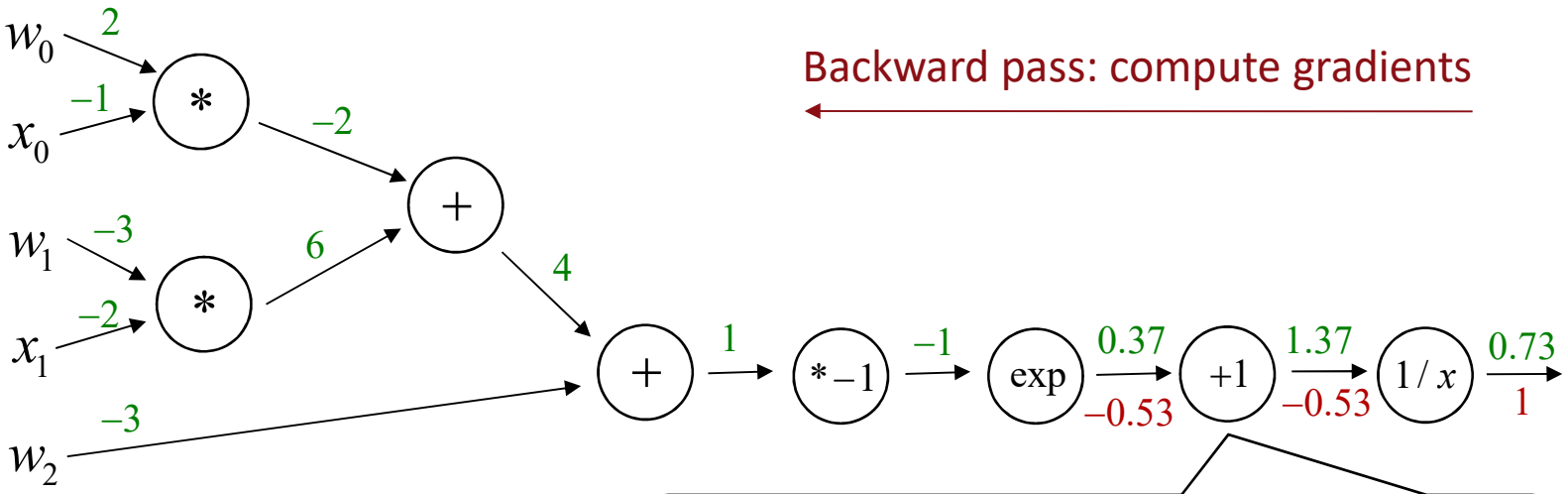
Local gradient: $\frac{\partial}{\partial x}\left[\frac{1}{x}\right] = -\frac{1}{x^2}$

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

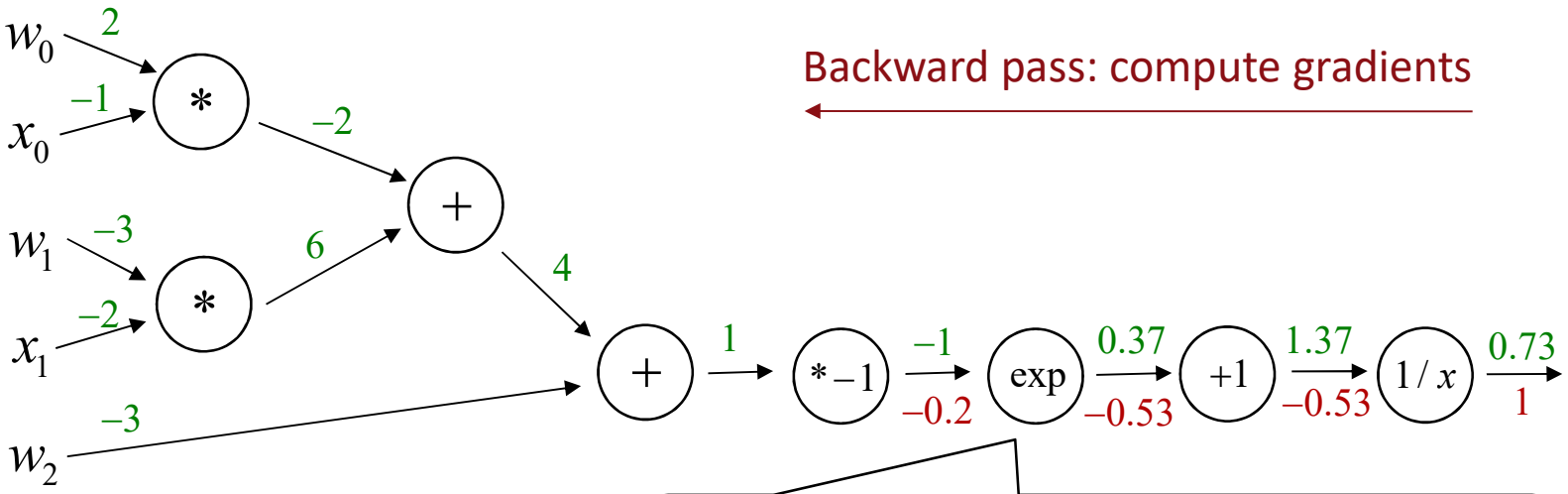
Local gradient: $\frac{\partial}{\partial x}[x+1] = 1$

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

# 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

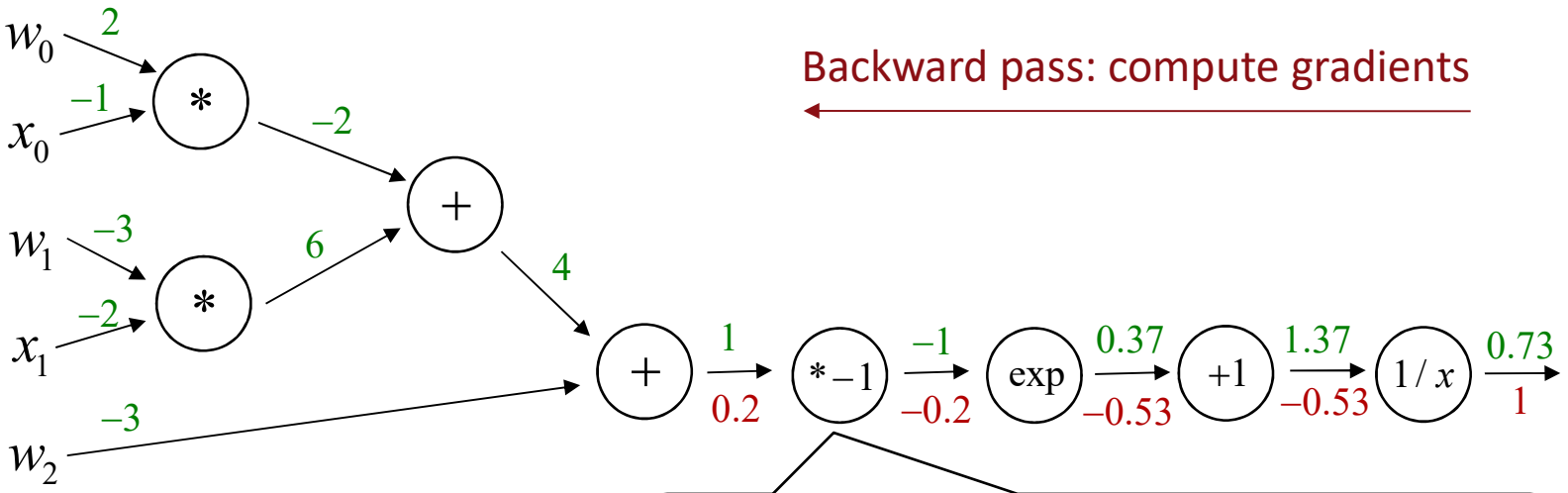
Local gradient: $\frac{\partial}{\partial x}\exp(x) = \exp(x)$

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

# Example: Backpropagation

- A slightly more complex example: backpropagation for the function

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




Upstream gradient: -0.2

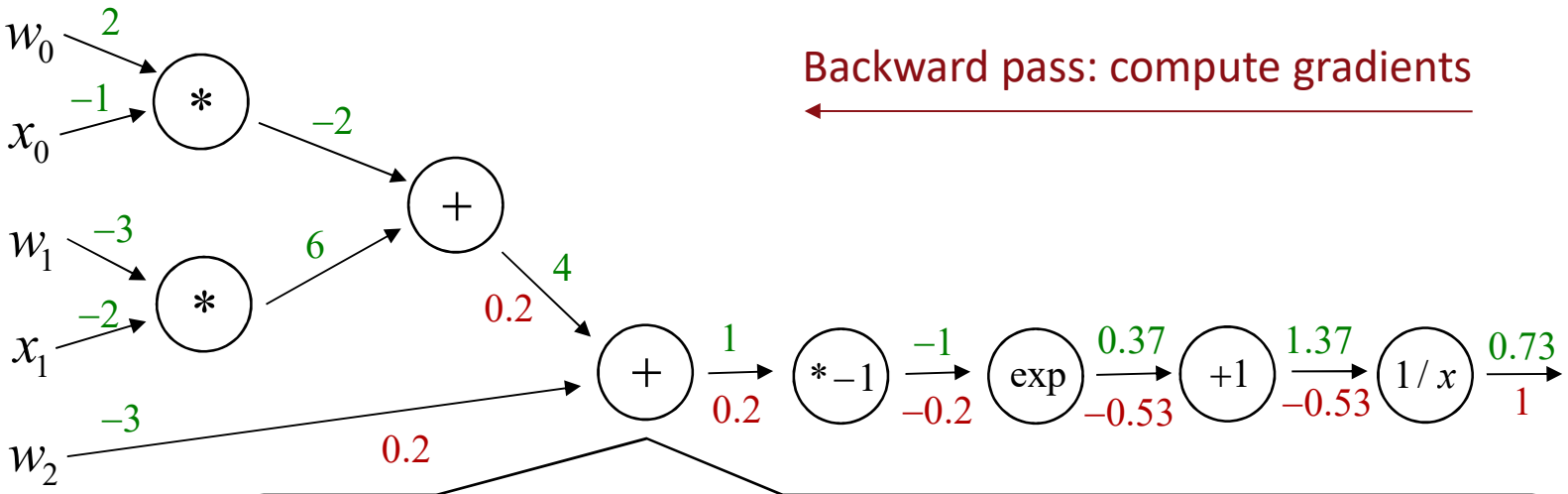
Local gradient: $\frac{\partial}{\partial x}[-x] = -1$

Downstream gradient: $(-1) \cdot (-0.2) = 0.2$

# 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

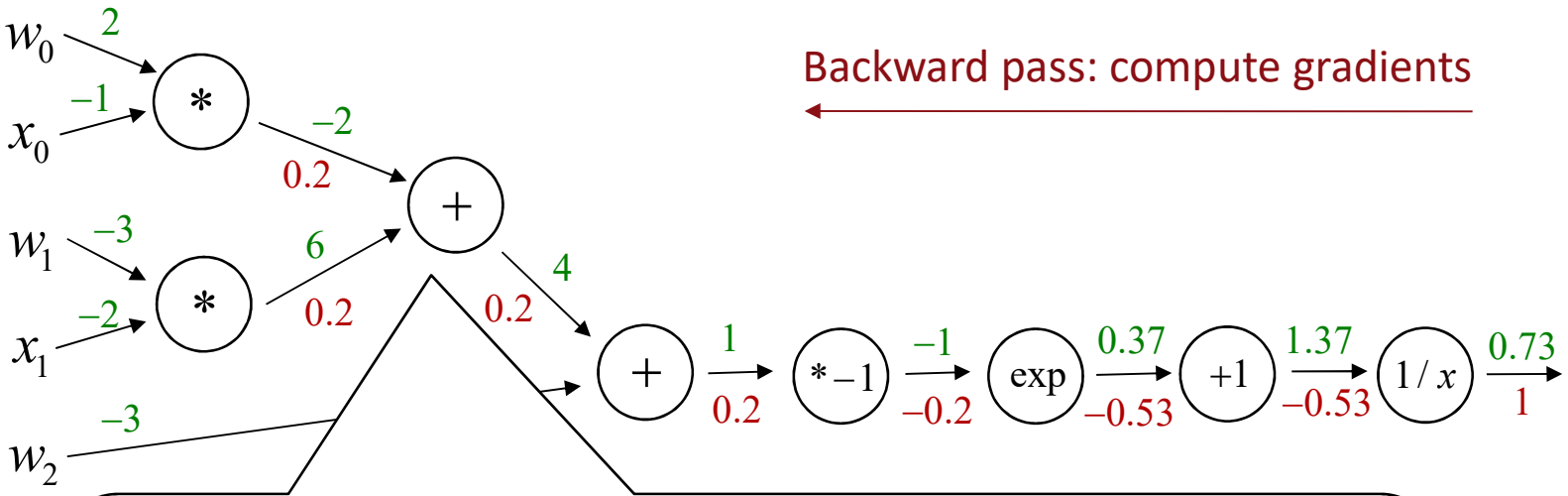
Local gradient: $\frac{\partial}{\partial x}[x + y] = 1, \quad \frac{\partial}{\partial y}[x + y] = 1$

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

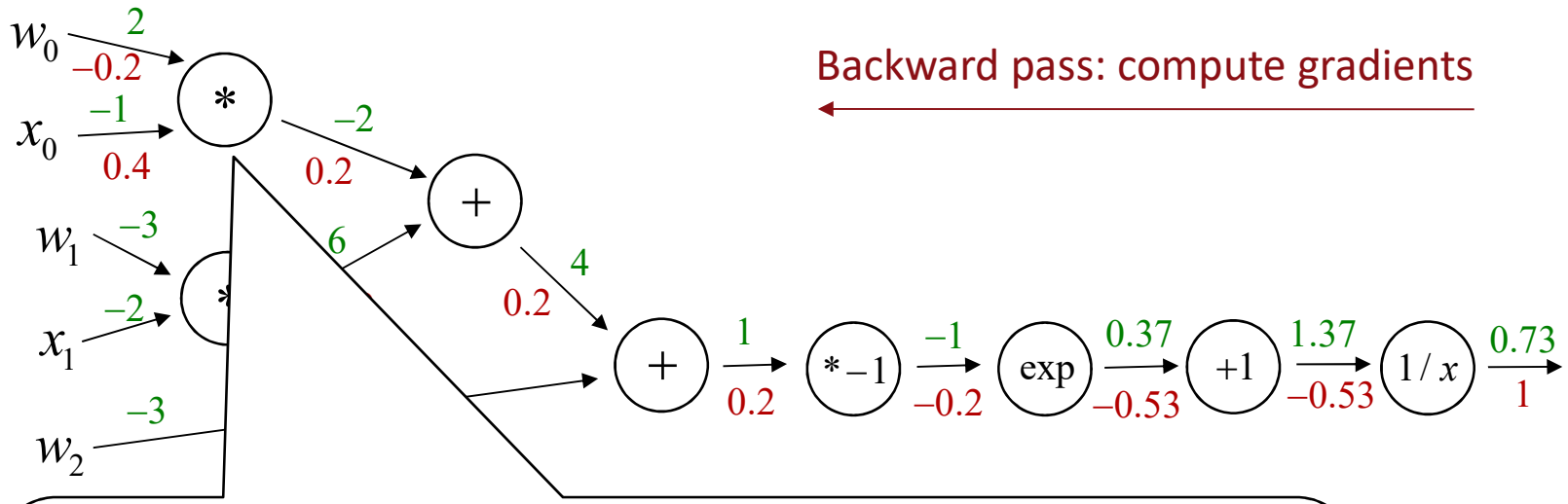
Local gradient: $\frac{\partial}{\partial x}[x+y]=1, \quad \frac{\partial}{\partial y}[x+y]=1$

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

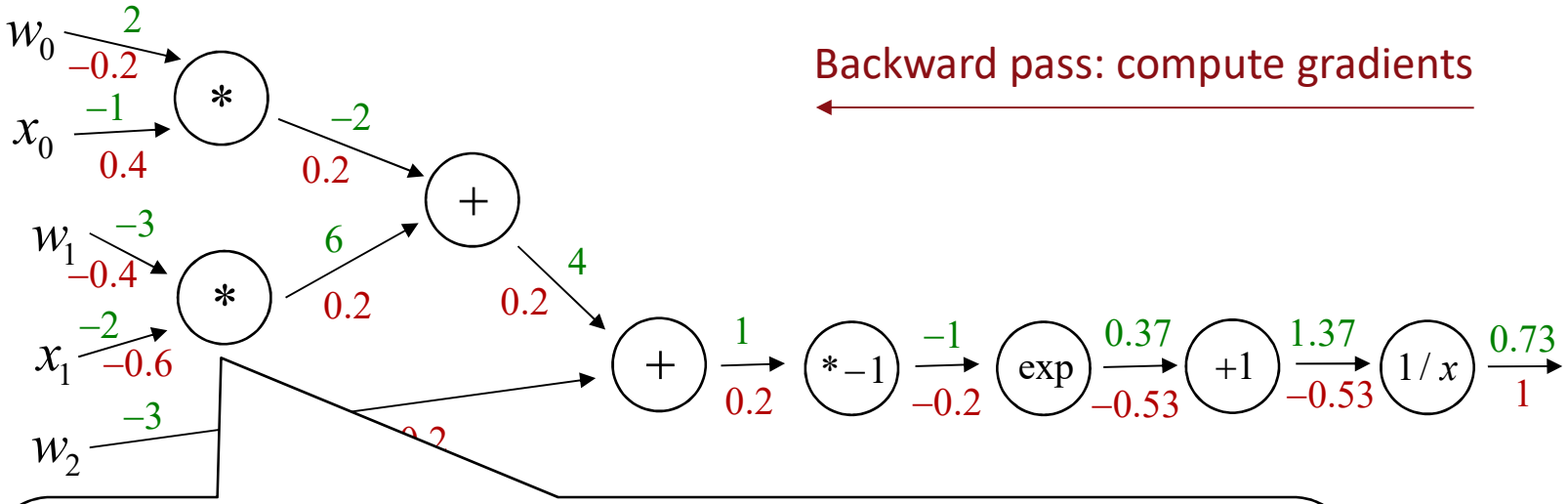
Local gradient: $\frac{\partial}{\partial x}[x \cdot y] = y, \quad \frac{\partial}{\partial y}[x \cdot y] = x$

Downstream gradients: $(-1) \cdot (0.2) = -0.2, \; 2 \cdot (0.2) = 0.4$

# 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

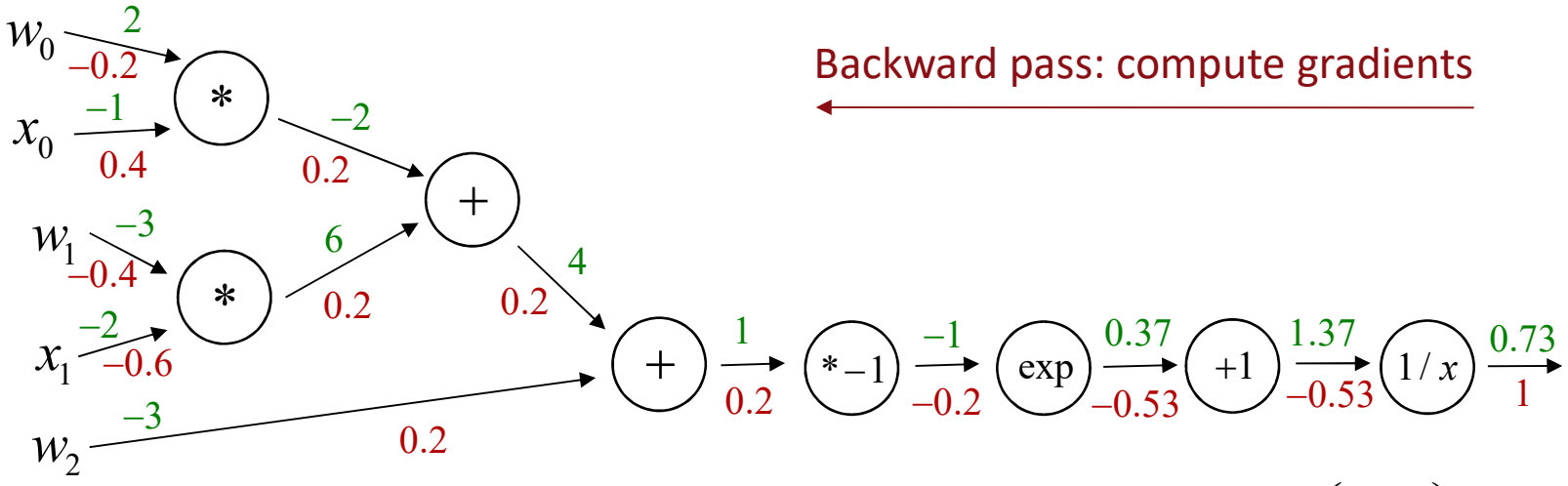
Local gradient: $\frac{\partial}{\partial x}[x \cdot y] = y, \quad \frac{\partial}{\partial y}[x \cdot y] = x$

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

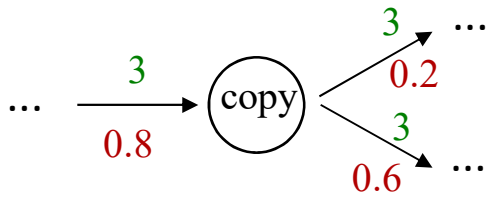
# Backpropagation from Objective Function

- For a neural network model, we need the gradient $\nabla L(\boldsymbol{\theta})$ of the objective function in the model parameters
- 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 (details are somewhat complicated)

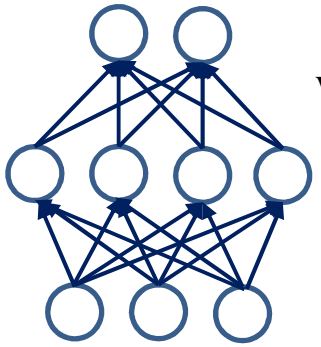
# Summary Neural Networks So Far

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

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

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

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



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

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

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

# Agenda for Lecture

- Introduction: Neural networks
- Training neural networks
- Regularization for neural networks

# Regularization for Neural Networks

- As for other machine learning models, **regularization** often improves predictive performance for neural networks

- The complexity of a neural network is strongly influenced by its architecture:
  - the number of layers and the number of neurons per layer determines how many parameters the model has
  - as discussed in earlier lectures, the number of parameters is one measure of the complexity of a model: more parameters can fit more complex functions, but are also more susceptible to overfitting

- Some regularization can therefore be imposed by choosing the neural network structure: limit the number of layers and number of nodes per layer to reduce the number of parameters

# Recap: Weight Regularization

- Neural networks can be regularized by a shrinkage term just like linear models
- Add a regularization term to loss function

$$L(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N} \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_n), y_n) + \lambda R(\boldsymbol{\theta})$$

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

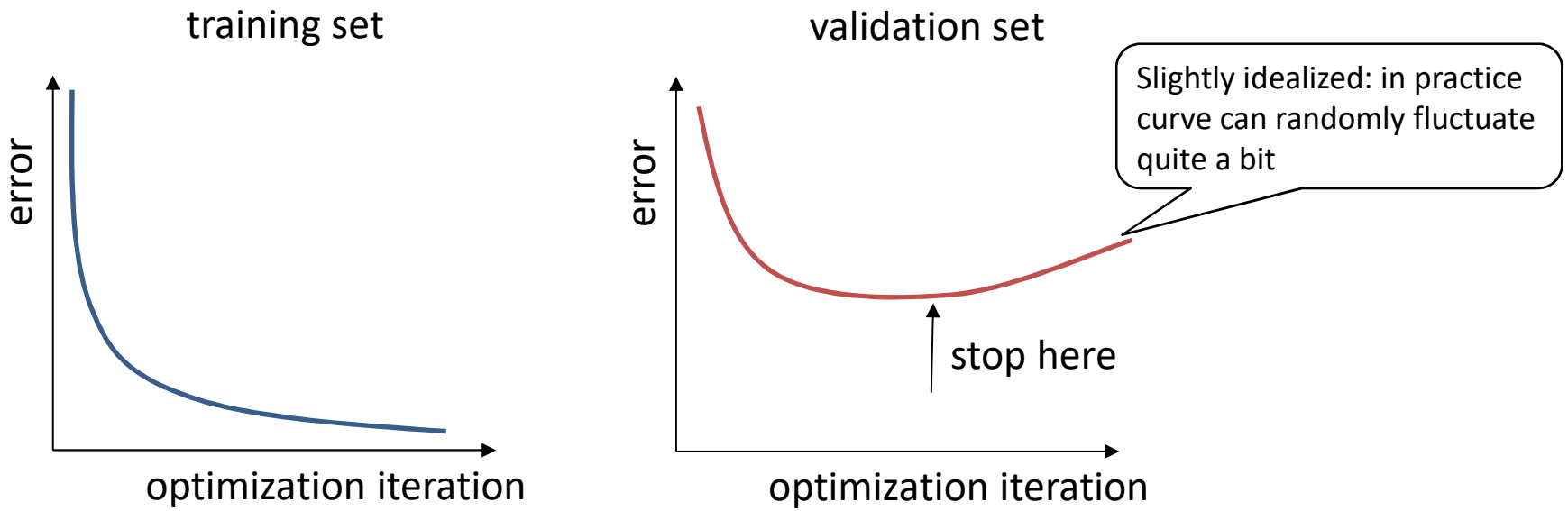
L1-regularizer $R(\boldsymbol{\theta}) = \sum_j |\theta_j|$

$\theta_j$: any model parameter, that is, all weights and biases in the neural network

- Strength of regularization $\lambda$ is a hyperparameter that must be tuned

# Early Stopping

- Simple practical regularization strategy: early stopping

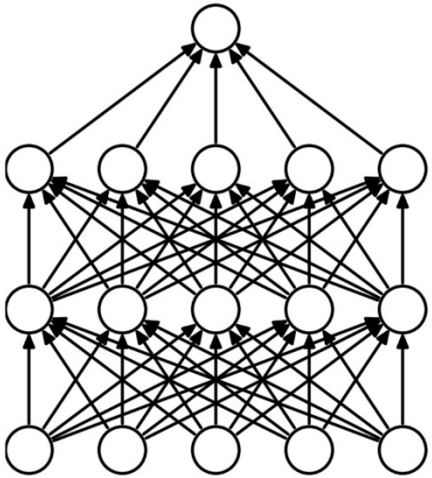



- Split off validation set and stop training when validation error increases
- Remember that for estimating future error of model, separate test set is needed

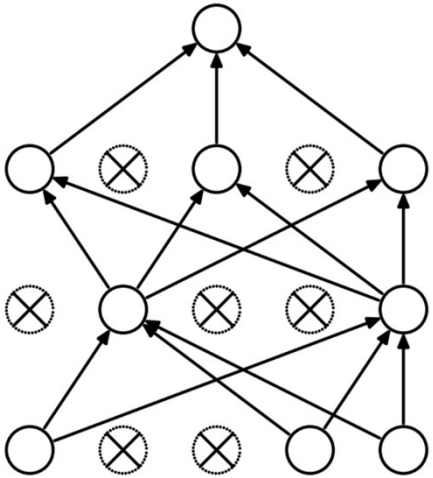
# Dropout

- **Dropout**: a widely used regularization method  for general neural networks

- In each forward pass (and corresponding backpropagation) during training, randomly set some neurons to zero in compute graph



without dropout

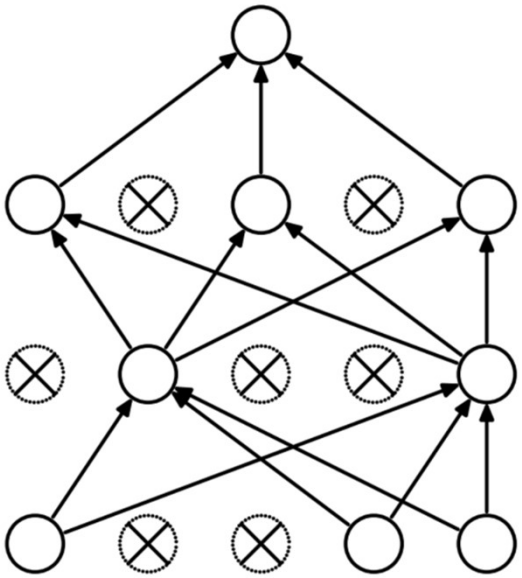


with dropout

- Implemented as dropout-layer with hyperparameter dropout probability (e.g. probability 0.5)

# Dropout: Motivation

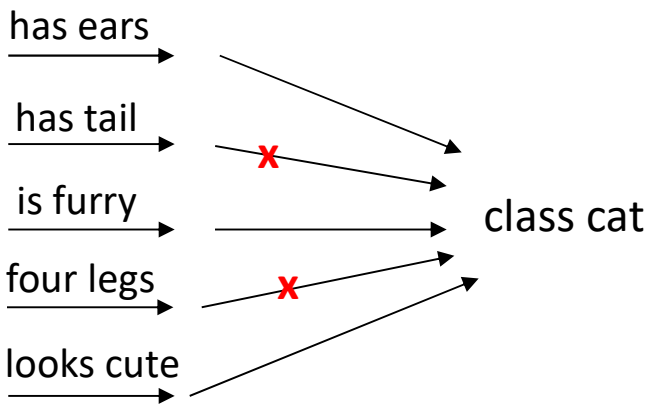
- What is the motivation behind dropout?



with dropout

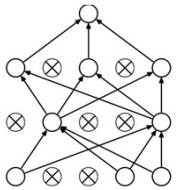
Network should learn a representation that is robust with respect to missing individual features

Also prevents co-adaptation of features (features that are always activated together and depend on each other)



# Dropout at Test Time

- What do we do with a model trained with dropout at test time?
- Keeping dropout at test time would make the output random:



$$\mathbf{z} \sim p(\mathbf{z}) \qquad \text{draw a random dropout mask } \mathbf{z}$$

$$y = f_{\boldsymbol{\theta}}(\mathbf{x}, \mathbf{z}) \qquad \text{compute output based on input } \mathbf{x} \text{ and mask } \mathbf{z}$$

- However, if we just remove dropout, the distribution of incoming activations to a node at test time will be very different from what it was at training time, because at training time many incoming activations were zero

- To resolve this, the activation of a node is multiplied with (1-*p*), where *p* is the dropout probability: scale down activation such that it matches expectation under dropout

# Summary: Neural Networks

- Neural networks are parameterized nonlinear models for supervised learning
- A neural network model consists of several stacked layers
  - first layer is the input, last layer is the output
  - activation values in one layer are computed by first computing a linear function of the input from the previous layers and then applying a nonlinearity such as ReLU
- Neural networks are trained by stochastic gradient descent
  - Compute approximation of gradient on small minibatches to improve efficiency
  - Gradient computed by automatic differentiation
  - Loss is non-convex, therefore only local optimum
- Neural networks (also called „Deep Learning“) are a large and important subfield of machine learning
  - Not just multilayer perceptrons: many specific architectures for vision, NLP, ...
  - Behind many recent breakthroughs in machine learning / AI
  - More details in specific lecture: „Deep Learning“, „Advanced Computer Vision",...