Winter semester 2022/23

# 344.175 VL: Natural Language Processing
# Neural Networks for NLP – a Walkthrough

Navid Rekab-saz

Email: navid.rekabsaz@jku.at
Office hours: https://navid-officehours.youcanbook.me

JOHANNES KEPLER UNIVERSITY LINZ

Institute of Computational Perception

# Notation – recap

- $a \longrightarrow$ scalar
- $\boldsymbol{b} \longrightarrow$ vector
  - $i^{th}$ element of $\boldsymbol{b}$ is the scalar $b_i$
- $\boldsymbol{C} \longrightarrow$ matrix
  - $i^{th}$ vector of $\boldsymbol{C}$ is $\boldsymbol{c}_i$
  - $j^{th}$ element of the $i^{th}$ vector of $\boldsymbol{C}$ is the scalar $c_{i,j}$
- Tensor: generalization of scalar, vector, matrix to any arbitrary dimension

# Probability

- Conditional probability, given two random variables $X$ and $Y$:

$$P(Y|X)$$

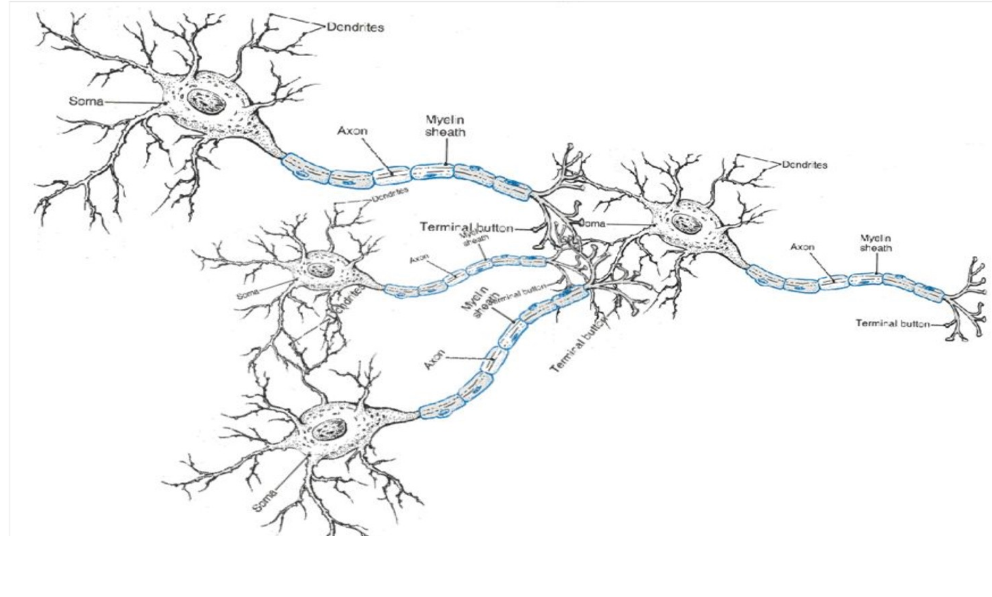
- Probability distribution
  - For a discrete random variable $Y$ with $K$ states (classes)
    - $0 \leq P(Y_i) \leq 1$
    - $\sum_{i=1}^{K} P(Y_i) = 1$
  - E.g. with $K = 4$ states: $[0.2 \quad 0.3 \quad 0.45 \quad 0.05]$

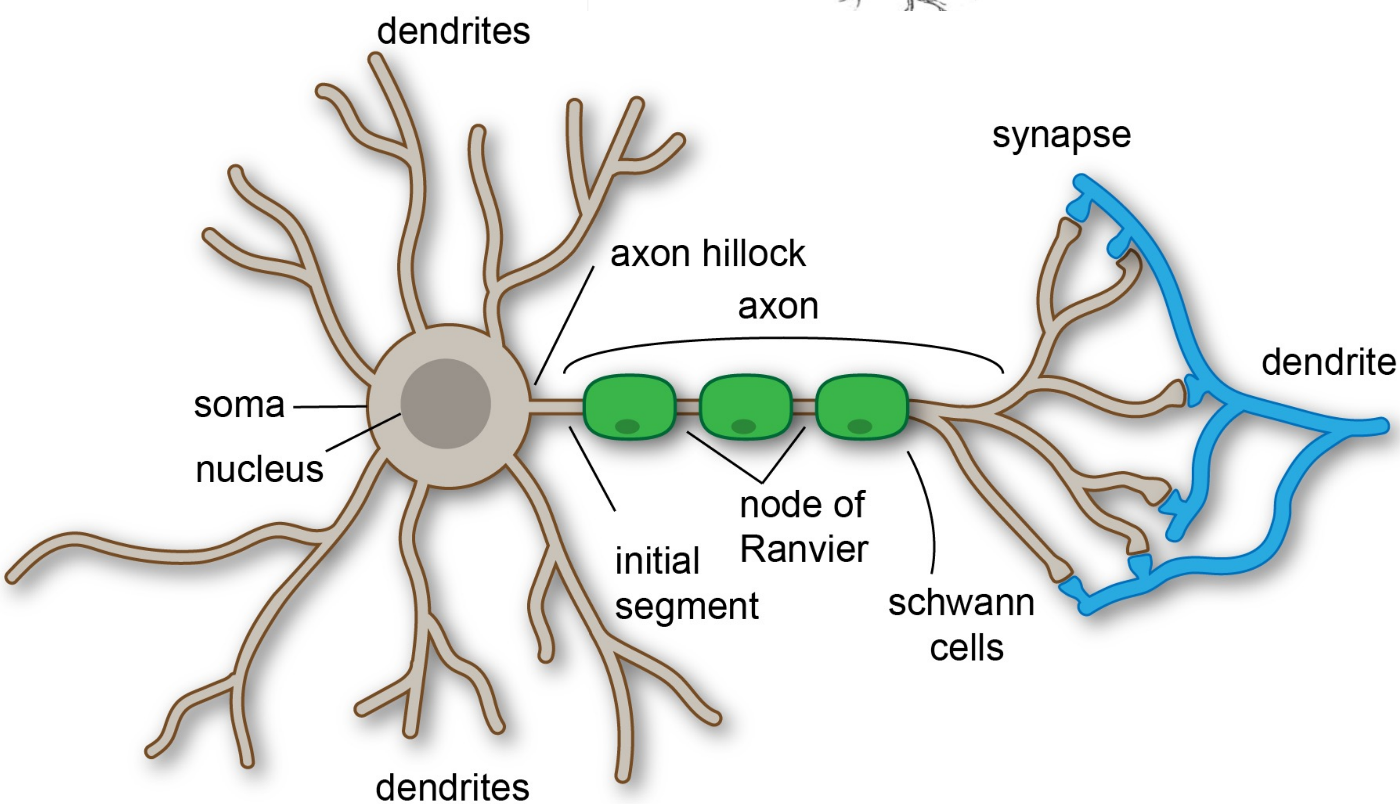
- Expected value over a set $\mathcal{D}$

$$\mathbb{E}_{\mathcal{D}}[f] = \frac{1}{|\mathcal{D}|} \sum_{x \in \mathcal{D}} f(x)$$

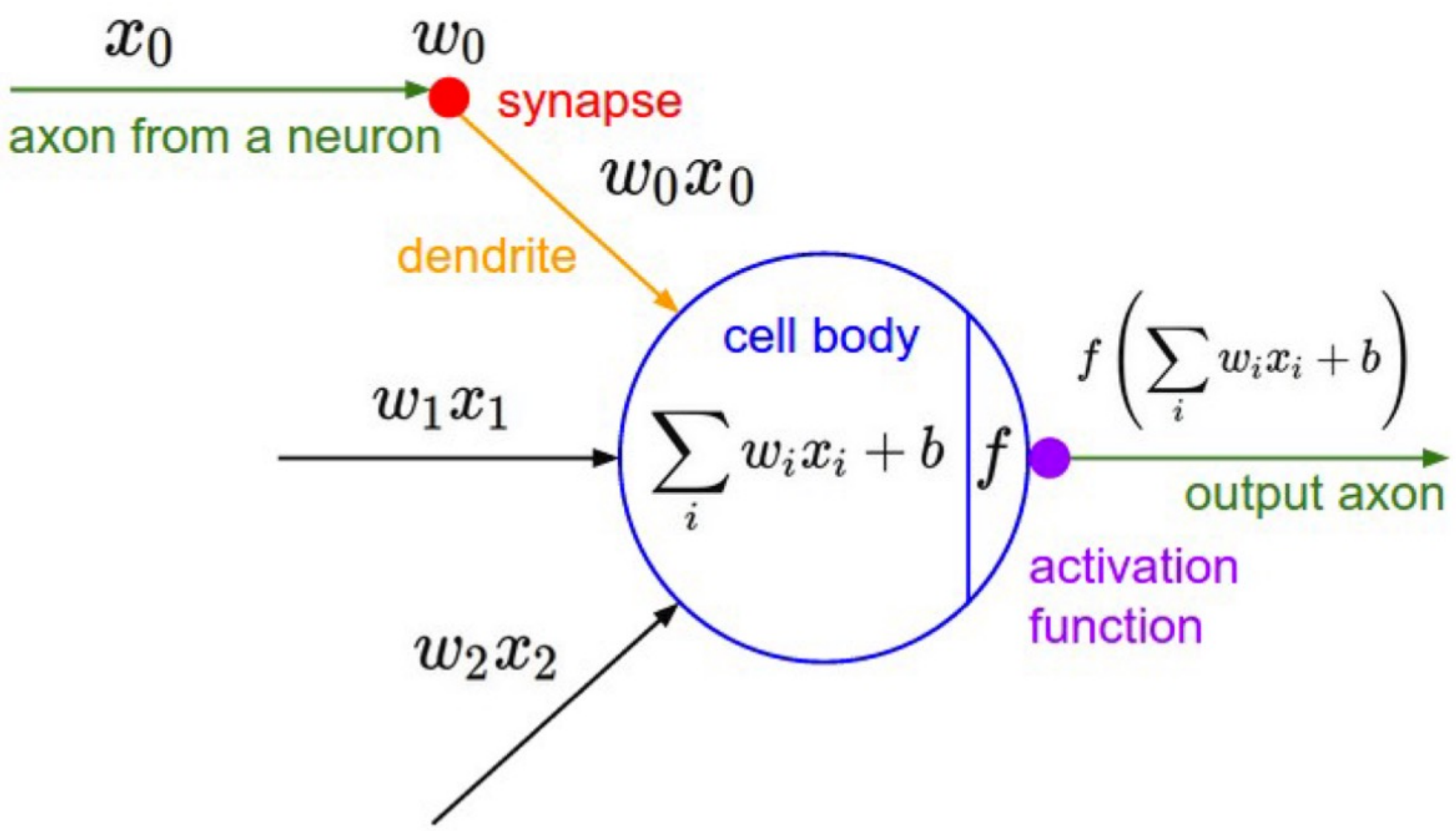
Note: The definition of expected value is not completely precise. Though, it suffices for our use in this lecture

# Neural Computation



# An Artificial Neuron




source

# Learning with Neural Networks

- Design the network's architecture
- Loop until some **exit criteria** are met
  - Sample a **(mini)batch** from training data $\mathcal{D}$
  - Execute **forward pass:** predict the output tensor of each given input tensor
  - Calculate **loss**
  - **Optimize** the network to reduce loss
    - Calculate the **gradient** of each parameter regarding the loss function using the **backpropagation** algorithm
    - **Update** parameters using their gradients

# Learning with Neural Networks

- Design the network's architecture
- Loop until some **exit criteria** are met
  - Sample a **(mini)batch** from training data $\mathcal{D}$
  - Execute **forward pass:** predict the output tensor of each given input tensor
  - Calculate **loss**
  - **Optimize** the network to reduce loss
    - Calculate the **gradient** of each parameter regarding the loss function using the **backpropagation** algorithm
    - **Update** parameters using their gradients

## Artificial Neural Networks

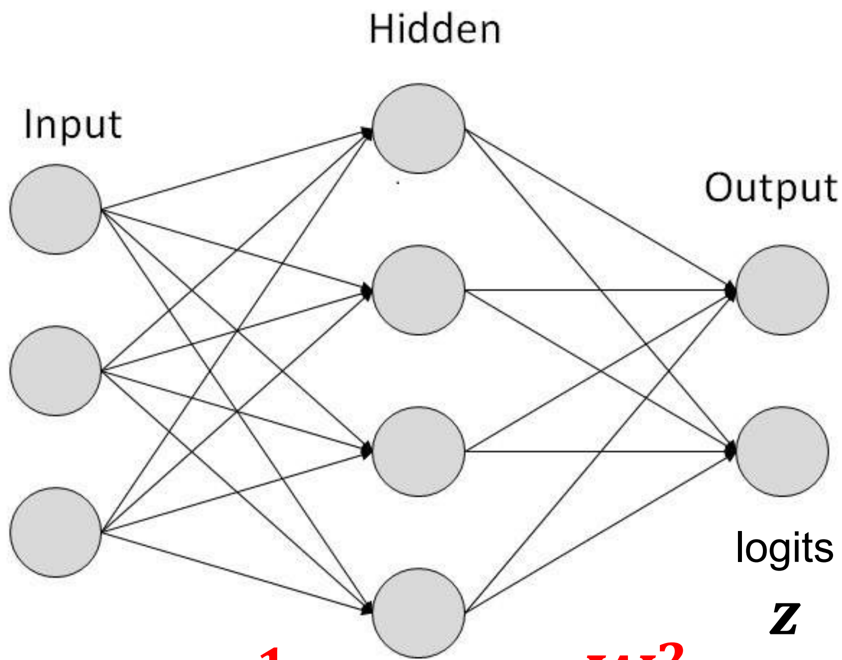
- Neural Networks are non-linear functions and universal approximators

- Neural networks can readily be defined as probabilistic models which estimate $P(Y|X)$

- Considering model parameter, $P(Y|X)$ can be written as $P(Y|\boldsymbol{x};\mathbb{W})$
  - $\boldsymbol{x}$ is an input vector and $\mathbb{W}$ is the set of model parameters
  - The model's predicted probability distribution is:

$$\hat{\boldsymbol{y}} = P(Y|\boldsymbol{x};\mathbb{W})$$

# A sample neural network (Multi Layer Perceptron)




input vector

$\boldsymbol{x}$

Predicted output probability distribution

$$\hat{\boldsymbol{y}} = P(Y|\boldsymbol{x}; \mathbb{W})$$

logits

$\boldsymbol{z}$

$\boldsymbol{W^1}$

size 3x4

$\boldsymbol{W^2}$

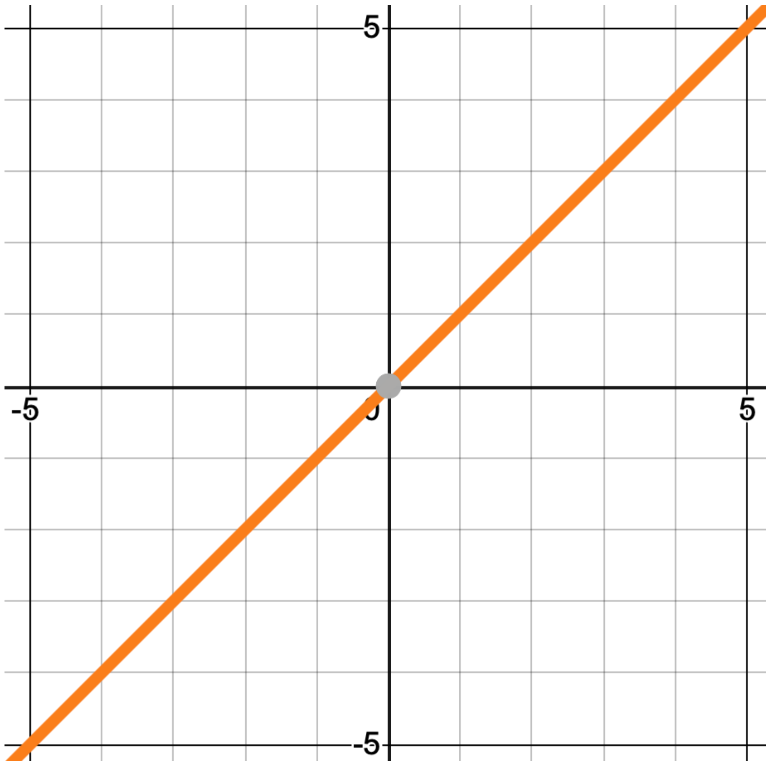
size 4x2

parameter matrices

Hidden nodes/layers apply non-linear functions to their inputs

# Linear

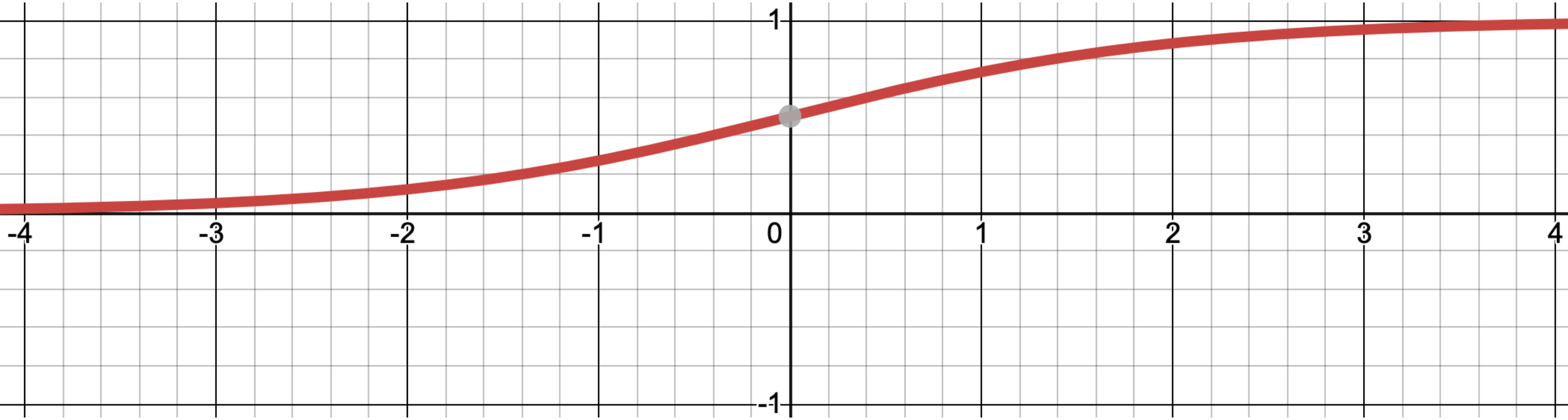
$$f(x) = x$$



## Non-linearities – Sigmoid

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

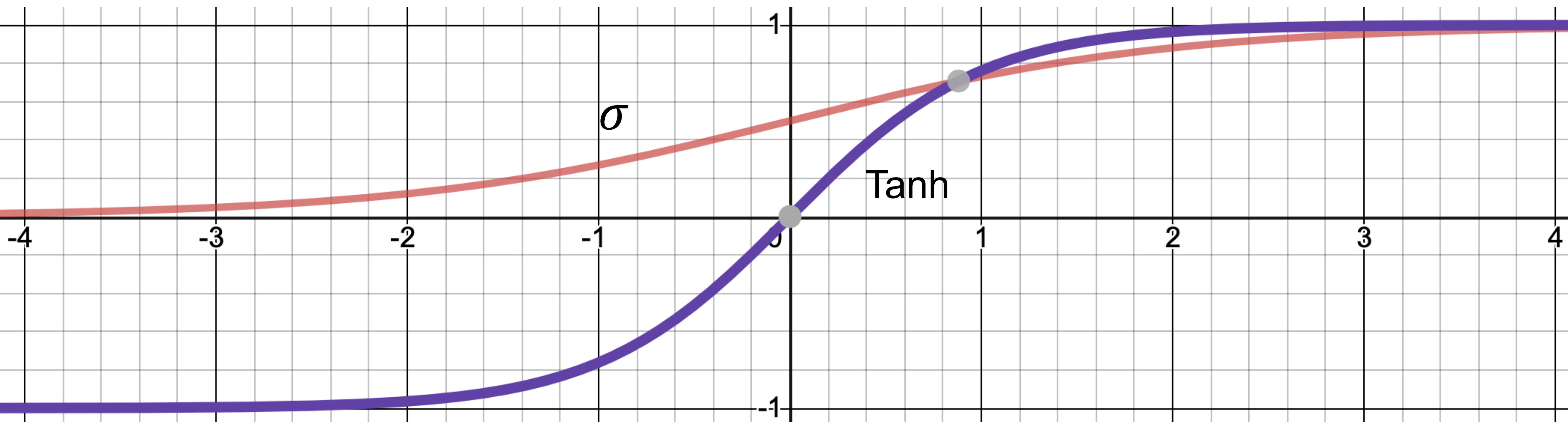
- squashes input between 0 and 1
- Output becomes like a probability value



# Hyperbolic Tangent (Tanh)

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

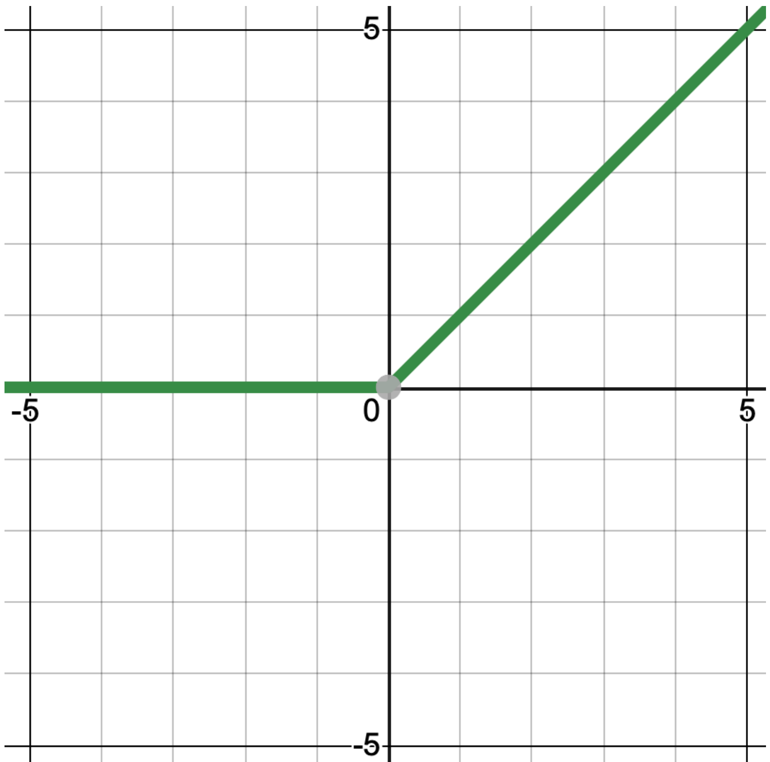
- squashes input between -1 and 1



## Rectified Linear Unit (ReLU)

$$\mathrm{ReLU}(x) = \max(0, x)$$

- fits to deep architectures, as it prevents vanishing gradient



# Examples

$$\boldsymbol{x} = \begin{bmatrix} 1 & 3 \end{bmatrix} \quad \boldsymbol{W} = \begin{bmatrix} 0.5 & -0.5 & 2 & 0 & 0 \\ 0 & 0 & 0 & 4 & -1 \end{bmatrix}$$

- Linear transformation $\boldsymbol{xW}$:

$$\boldsymbol{xW} = \begin{bmatrix} 1 & 3 \end{bmatrix} \begin{bmatrix} 0.5 & -0.5 & 2 & 0 & -1 \\ 0 & 0 & 0 & 4 & -1 \end{bmatrix} = \begin{bmatrix} \mathbf{0.5} & \mathbf{-0.5} & \mathbf{2} & \mathbf{12} & \mathbf{-4} \end{bmatrix}$$

- Non-linear transformation $\mathrm{ReLU}(\boldsymbol{xW})$:

$$\mathrm{ReLU}(\begin{bmatrix} 0.5 & -0.5 & 2 & 12 & -4 \end{bmatrix}) = \begin{bmatrix} \mathbf{0.5} & \mathbf{0.0} & \mathbf{2} & \mathbf{12} & \mathbf{0.0} \end{bmatrix}$$

- Non-linear transformation $\sigma(\boldsymbol{xW})$:

$$\sigma(\begin{bmatrix} 0.5 & -0.5 & 2 & 12 & -4 \end{bmatrix}) = \begin{bmatrix} \mathbf{0.62} & \mathbf{0.37} & \mathbf{0.88} & \mathbf{0.99} & \mathbf{0.11} \end{bmatrix}$$

- Non-linear transformation $\tanh(\boldsymbol{xW})$:

$$\tanh(\begin{bmatrix} 0.5 & -0.5 & 2 & 12 & -4 \end{bmatrix}) = \begin{bmatrix} \mathbf{0.46} & \mathbf{-0.46} & \mathbf{0.96} & \mathbf{0.99} & \mathbf{-0.99} \end{bmatrix}$$

# Learning with Neural Networks

- Design the network's architecture
- Loop until some **exit criteria** are met
  - Sample a **(mini)batch** from training data $\mathcal{D}$
  - Execute **forward pass:** predict the output tensor of each given input tensor
  - Calculate **loss**
  - **Optimize** the network to reduce loss
    - Calculate the **gradient** of each parameter regarding the loss function using the **backpropagation** algorithm
    - **Update** parameters using their gradients

# Early Stopping

- Run the model for several steps (epochs), and in each step evaluate the model on the validation set
- Store the model if the evaluation results improve
- At the end, take the stored model with the best validation results as the final model

# Learning with Neural Networks

- Design the network's architecture
- Loop until some **exit criteria** are met
  - Sample a **(mini)batch** from training data $\mathcal{D}$
  - Execute **forward pass:** predict the output tensor of each given input tensor
  - Calculate **loss**
  - **Optimize** the network to reduce loss
    - Calculate the **gradient** of each parameter regarding the loss function using the **backpropagation** algorithm
    - **Update** parameters using their gradients

## Toy neural network

- A sample neural network is going to calculate the following function:

$$z(x; \mathbb{W}) = {w_2}^2 * (2 * x * w_1 + w_0)$$

  - $x$ is input and $\mathbb{W}$ is the tensor of parameters
  - Parameters are initialized with

$$w_0 = 1 \quad w_1 = 3 \quad w_2 = 2$$

- A neural network first redefines this function as subfunctions of basic/atomic operations with new intermediary variables:*

$$a = 2 * x * w_1$$

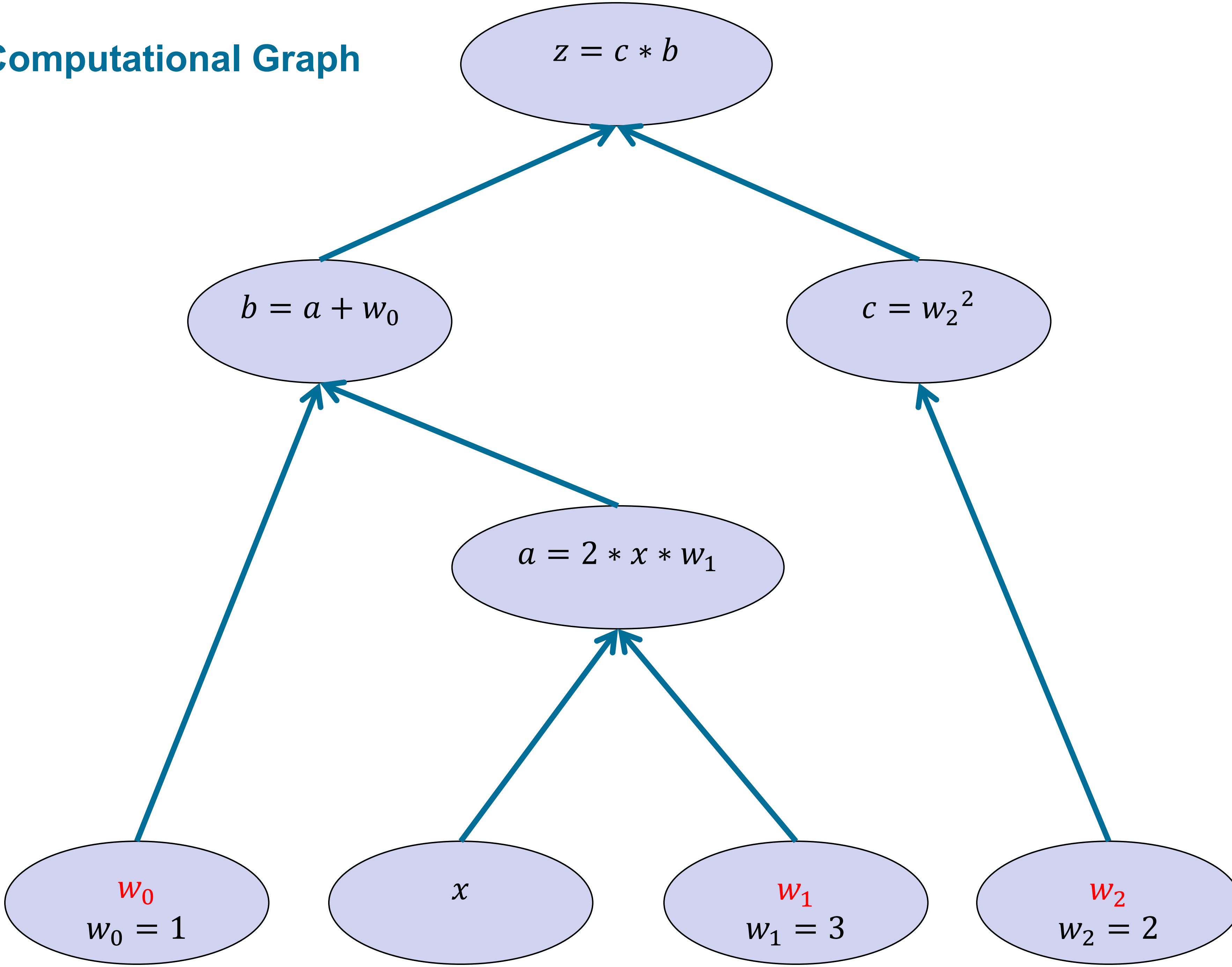
$$b = a + w_0$$

$$c = {w_2}^2$$

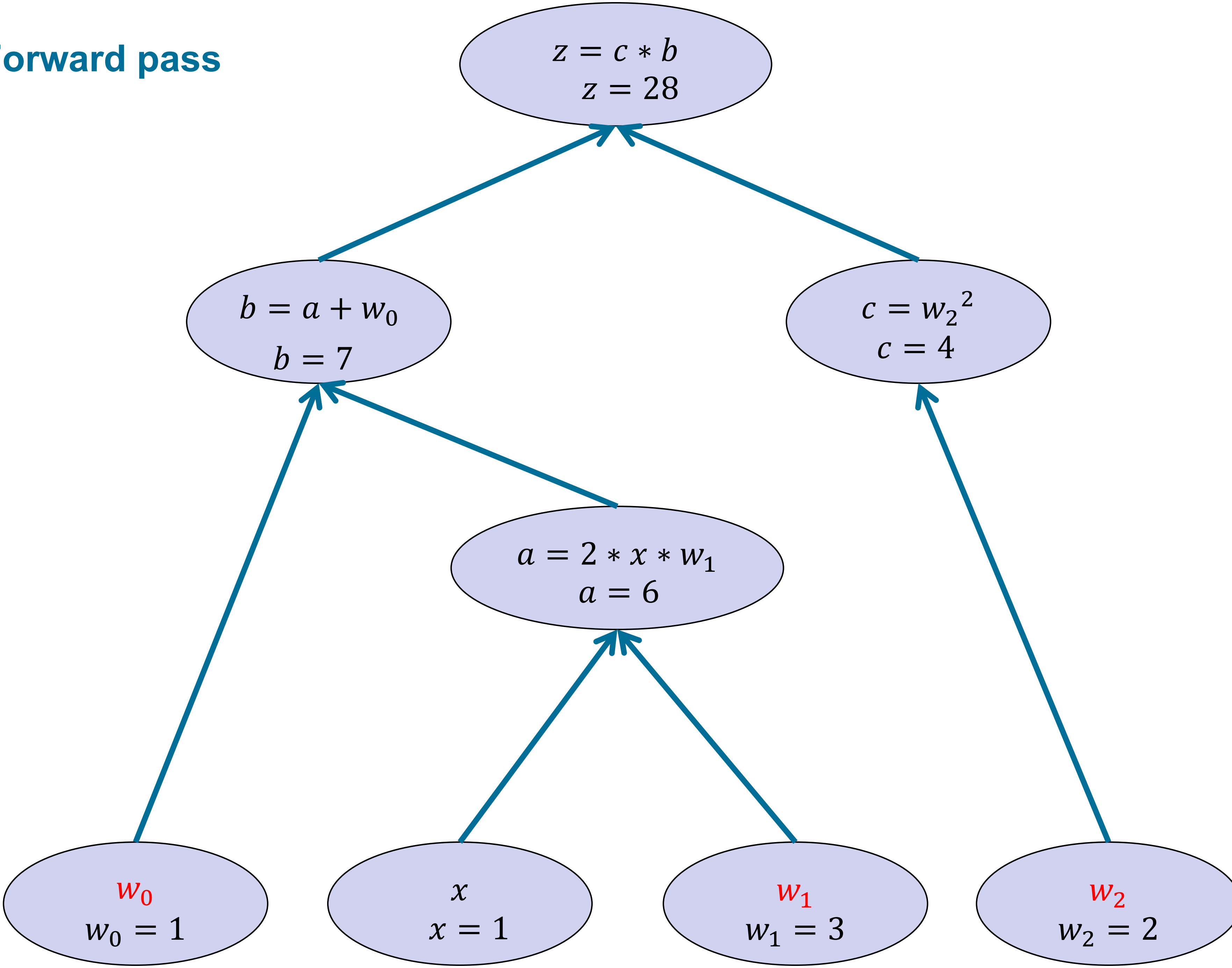
$$z = c * b$$

* To keep the example simple, the splitting is not applied to all basic operation

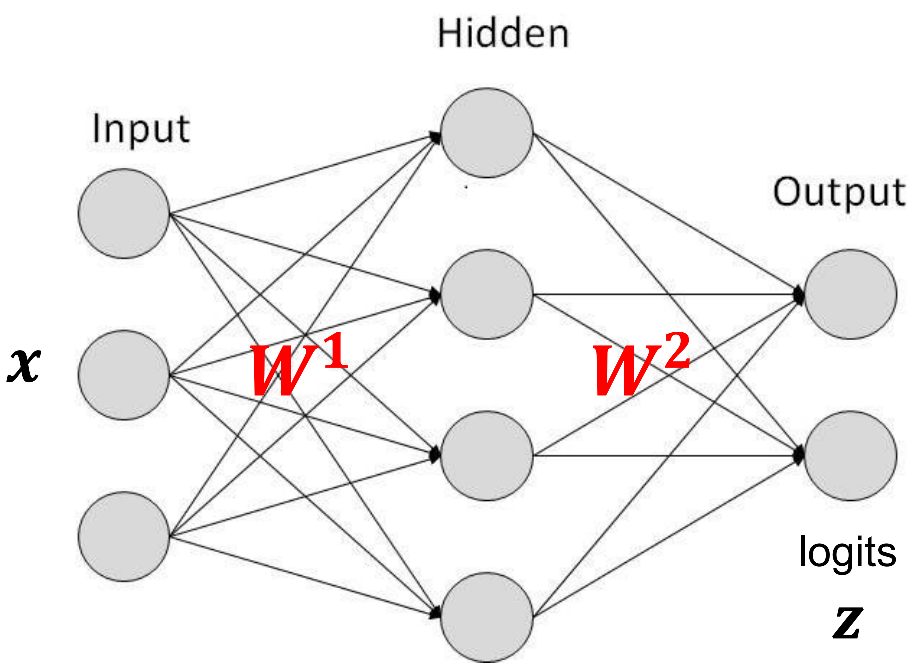
## Computational Graph



## Forward pass



# Output probability distribution




Predicted output probability distribution

$$\hat{\boldsymbol{y}} = P(Y|\boldsymbol{x};\mathbb{W})$$

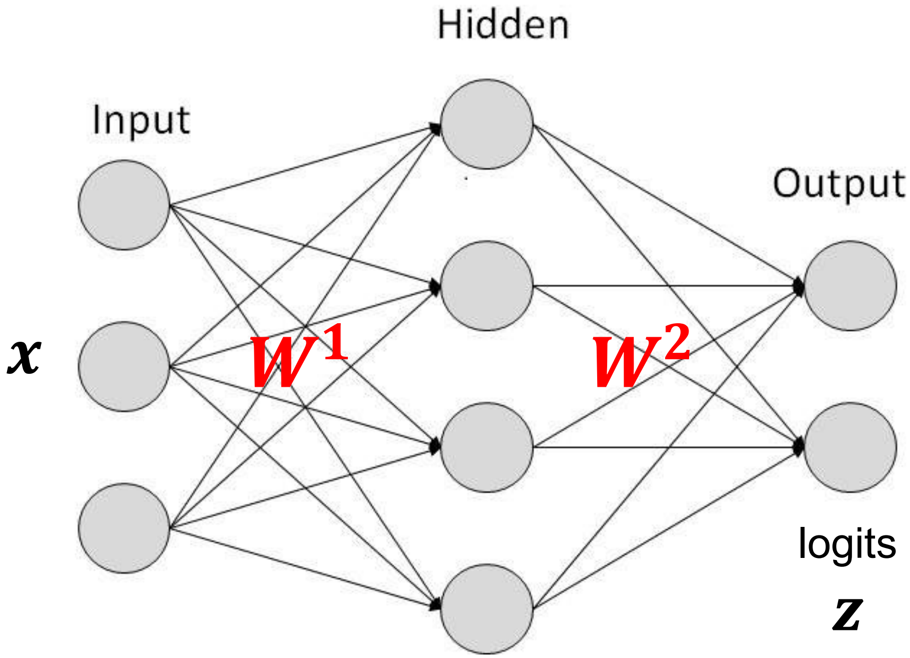
# Softmax

- As discussed, neural networks can readily turn to probabilistic models
- To do it, we need to transform the output vector $\boldsymbol{z}$ of a neural network with $K$ output classes to a probability distribution
  - In the context of neural networks, $\boldsymbol{z}$ is usually called **logits**
- softmax turns a vector to a probability distribution
  - $\boldsymbol{z}$ could be the output vector of a neural network

$$\text{softmax}(\boldsymbol{z})_l = \frac{e^{z_l}}{\sum_{i=1}^{K} e^{z_i}}$$

normalization term

# Output probability distribution




Predicted output probability distribution

$$\hat{\boldsymbol{y}} = P(Y|\boldsymbol{x}; \mathbb{W})$$

$$= \mathrm{softmax}(\boldsymbol{z})$$

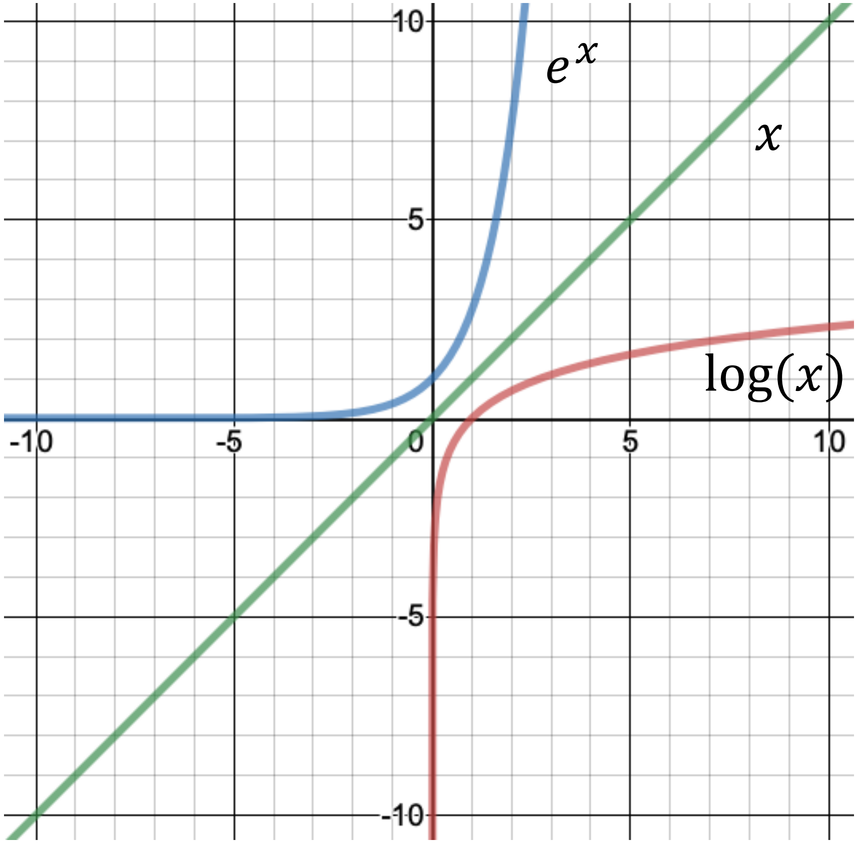
# Softmax – example

$$K = 4 \text{ classes}$$

$$\text{softmax}(\boldsymbol{z})_l = \frac{e^{z_l}}{\sum_{i=1}^{K} e^{z_i}}$$

$$\boldsymbol{z} = \begin{bmatrix} 1 \\ 2 \\ 5 \\ 6 \end{bmatrix}$$

$$\text{softmax}(\boldsymbol{z}) = \begin{bmatrix} 0.004 \\ 0.013 \\ 0.264 \\ 0.717 \end{bmatrix}$$



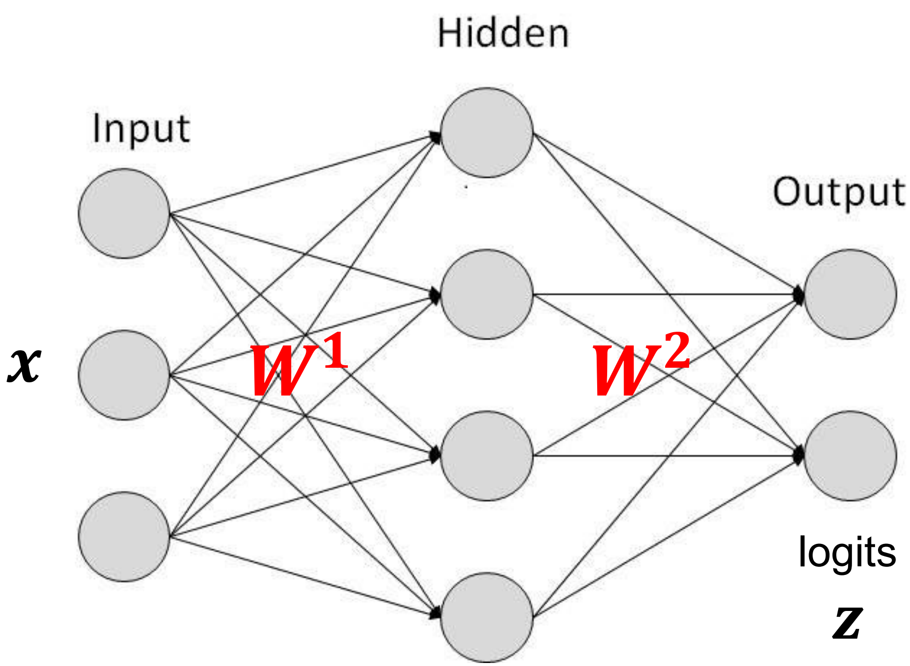
## Softmax characteristics

- The exponential function in softmax makes the maximum becomes much higher than the others
- Softmax identifies the "*max*" but in a "*soft*" way!
- Softmax imposes competition between the predicted output values, as in fact "*winner takes (almost) all!*"
  - Winner-takes-all is the case when one value is 1 and the rest are 0
  - Softmax provides a soft distribution of winner-takes-all
  - This resembles the competition between nearby neurons in the cortex

# Learning with Neural Networks

- Design the network's architecture
- Loop until some **exit criteria** are met
  - Sample a **(mini)batch** from training data $\mathcal{D}$
  - Execute **forward pass:** predict the output tensor of each given input tensor
  - Calculate **loss**
  - **Optimize** the network to reduce loss
    - Calculate the **gradient** of each parameter regarding the loss function using the **backpropagation** algorithm
    - **Update** parameters using their gradients

# Sample neural network




Predicted output probability distribution

$$\hat{\boldsymbol{y}} = P(Y|\boldsymbol{x}; \mathbb{W}) = \text{softmax}(\boldsymbol{z})$$

$\mathcal{L}$

Labels

$\boldsymbol{y}$

# Cross Entropy Loss

- Given a classification task with $K$ classes
  - known as multi-class classification
- $\hat{\boldsymbol{y}} \rightarrow$ predicted probability distribution of the classes
- $\boldsymbol{y} \rightarrow$ actual probability distribution of the classes (labels)
- Cross Entropy loss is defined as:

$$\mathcal{L} = -\mathbb{E}_{\mathcal{D}} \sum_{i=1}^{K} y_i \log \hat{y}_i$$

  - $\mathcal{D} \rightarrow$ the set of training data
- In neural networks, we can write it as:

$$\mathcal{L}(\mathbb{W}) = -\mathbb{E}_{\mathcal{D}} \sum_{i=1}^{K} y_i \log P(Y_i | \boldsymbol{x}; \mathbb{W})$$

# Cross Entropy Loss – example 1

- A multi-label scenario:

$$\hat{\boldsymbol{y}} = \begin{bmatrix} 0.004 \\ 0.013 \\ 0.264 \\ 0.717 \end{bmatrix} \qquad \boldsymbol{y} = \begin{bmatrix} 0 \\ 0.25 \\ 0 \\ 0.75 \end{bmatrix}$$

$$\mathcal{L} = -\sum_{i=1}^{K} y_i \log \hat{y}_i$$

$$\mathcal{L} = -(0\times \log 0.004 + 0.25\times \log 0.013 + 0\times \log 0.264 + 0.75\times \log 0.717)$$

$$\mathcal{L} = -(0-0.471 + 0-0.108)$$

$$\mathcal{L} = 0.579$$

# Cross Entropy Loss – example 2

- A single-label scenario:

$$\hat{\boldsymbol{y}} = \begin{bmatrix} 0.004 \\ 0.013 \\ 0.264 \\ 0.717 \end{bmatrix} \qquad \boldsymbol{y} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$$\mathcal{L} = -\sum_{i=1}^{K} y_i \log \hat{y}_i$$

$$\mathcal{L} = -(0\times \log 0.004 + 0\times \log 0.013 + 0\times \log 0.264 + 1\times \log 0.717)$$

$$\mathcal{L} = -(0 + 0 + 0 - 0.144)$$

$$\mathcal{L} = 0.144$$

# Negative Log Likelihood (NLL) Loss

- Single-label classification is the most common scenario

- In this case, we can simplify Cross Entropy formulation to

$$\mathcal{L}(\mathbb{W}) = -\mathbb{E}_{\mathcal{D}} \sum_{i=1}^{K} y_i \log P(Y_i | \boldsymbol{x}; \mathbb{W}) = -\mathbb{E}_{\mathcal{D}} \log P(Y_l | \boldsymbol{x}; \mathbb{W})$$

  - where $l$ is the index of the correct class

- This loss function is known as Negative Log Likelihood (NLL)
  - NLL is a special case of Cross Entropy

## NLL + softmax

- What happens when we use NLL and softmax in the output layer of a neural network?

$$\mathcal{L}(\mathbb{W}) = -\mathbb{E}_{\mathcal{D}} \log P(Y_l | \boldsymbol{x}; \mathbb{W}) = -\mathbb{E}_{\mathcal{D}} \log \mathrm{softmax}(\boldsymbol{z})_l$$

$\boldsymbol{z} \rightarrow$ output vector before softmax (logits)

$$\mathcal{L}(\mathbb{W}) = -\mathbb{E}_{\mathcal{D}} \log \frac{e^{z_l}}{\sum_{i=1}^{K} e^{z_i}} = -\mathbb{E}_{\mathcal{D}} \left[ \log e^{z_l} - \log \sum_{i=1}^{K} e^{z_i} \right]$$

$$\mathcal{L}(\mathbb{W}) = -\mathbb{E}_{\mathcal{D}} \left[ z_l - \log \sum_{i=1}^{K} e^{z_i} \right]$$

This term is (almost) equal to $\max(\boldsymbol{z})$

# NLL + softmax – example 1

$$\mathcal{L} = -\left[z_l - \log \sum_{i=1}^{K} e^{z_i}\right]$$

$$\boldsymbol{z} = [1 \quad 2 \quad 0.5 \quad 6]$$

- If the correct class is the first one, $l = 1$:

$$\mathcal{L} = -[1 - \log(e^1 + e^2 + e^{0.5} + e^6)] = -1 + 6.02 = \mathbf{5.02}$$

- If the correct class is the third one, $l = 3$:

$$\mathcal{L} = -[0.5 - \log(e^1 + e^2 + e^{0.5} + e^6)] = -0.5 + 6.02 = \mathbf{5.52}$$

- If the correct class is the fourth one, $l = 4$:

$$\mathcal{L} = -[6 - \log(e^1 + e^2 + e^{0.5} + e^6)] = -6 + 6.02 = \mathbf{0.02}$$

# NLL + softmax – example 2

$$\mathcal{L} = -\left[z_l - \log \sum_{i=1}^{K} e^{z_i}\right]$$

$$\boldsymbol{z} = \begin{bmatrix} 1 & 2 & 5 & 6 \end{bmatrix}$$

- If the correct class is the first one, $l = 1$:

$$\mathcal{L} = -[1 - \log(e^1 + e^2 + e^5 + e^6)] = -1 + 6.33 = \mathbf{5.33}$$

- If the correct class is the third one, $l = 3$:

$$\mathcal{L} = -[5 - \log(e^1 + e^2 + e^5 + e^6)] = -5 + 6.33 = \mathbf{1.33}$$

- If the correct class is the fourth one, $l = 4$:

$$\mathcal{L} = -[6 - \log(e^1 + e^2 + e^5 + e^6)] = -6 + 6.33 = \mathbf{0.33}$$

# Learning with Neural Networks

- Design the network's architecture
- Loop until some **exit criteria** are met
  - Sample a **(mini)batch** from training data $\mathcal{D}$
  - Execute **forward pass:** predict the output tensor of each given input tensor
  - Calculate **loss** function of the (mini)batch
  - **Optimize** the network to reduce loss
    - Calculate the **gradient** of each parameter regarding the loss function using the **backpropagation** algorithm
    - **Update** parameters using their gradients

# Toy neural network

$$z(x; \mathbb{W}) = {w_2}^2 * (2 * x * w_1 + w_0)$$

- Initialization: $w_0 = 1 \quad w_1 = 3 \quad w_2 = 2$
- Intermediary variables:

$$a = 2 * x * w_1$$

$$b = a + w_0$$

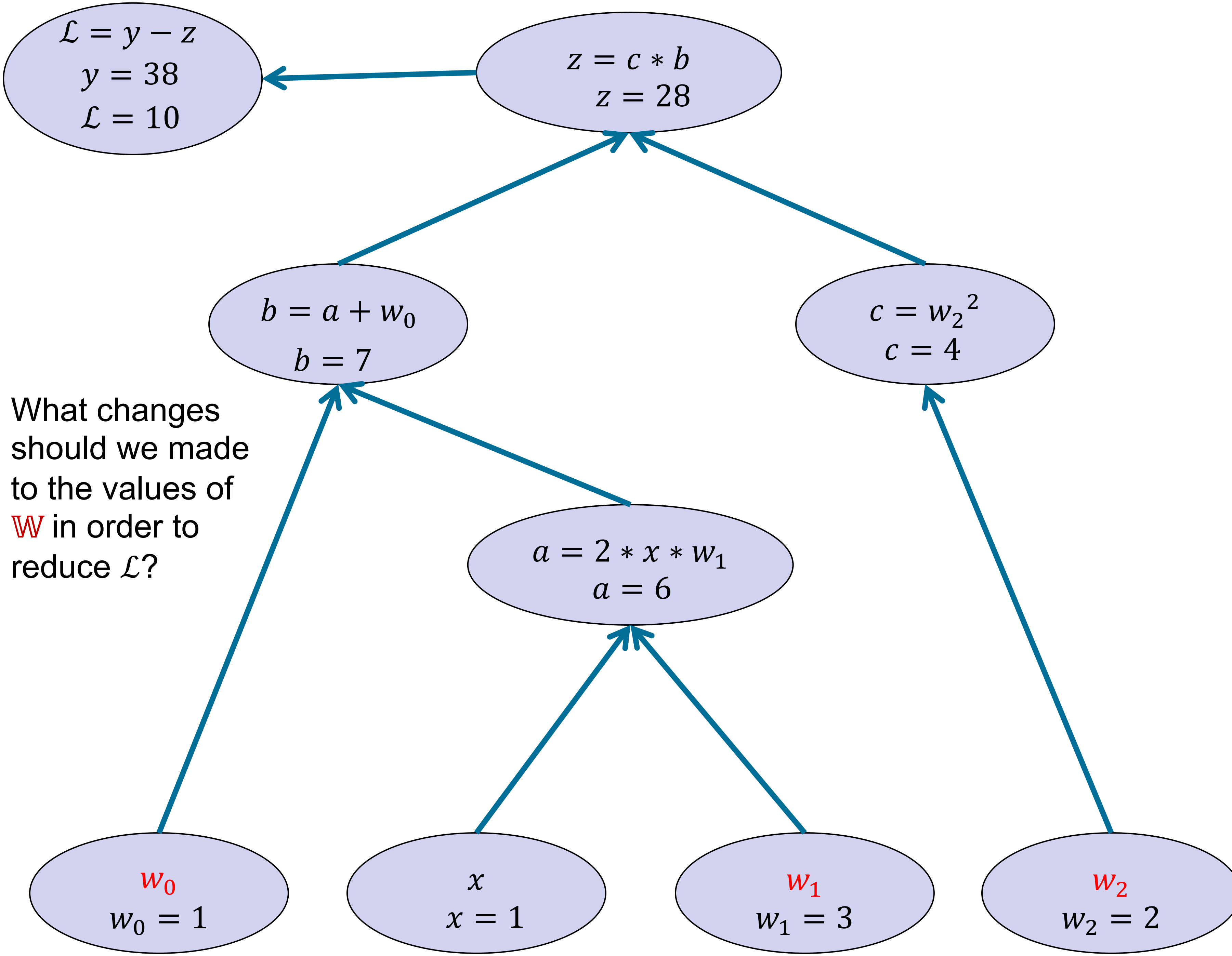
$$c = {w_2}^2$$

$$z = c * b$$

- An "imaginary" loss:

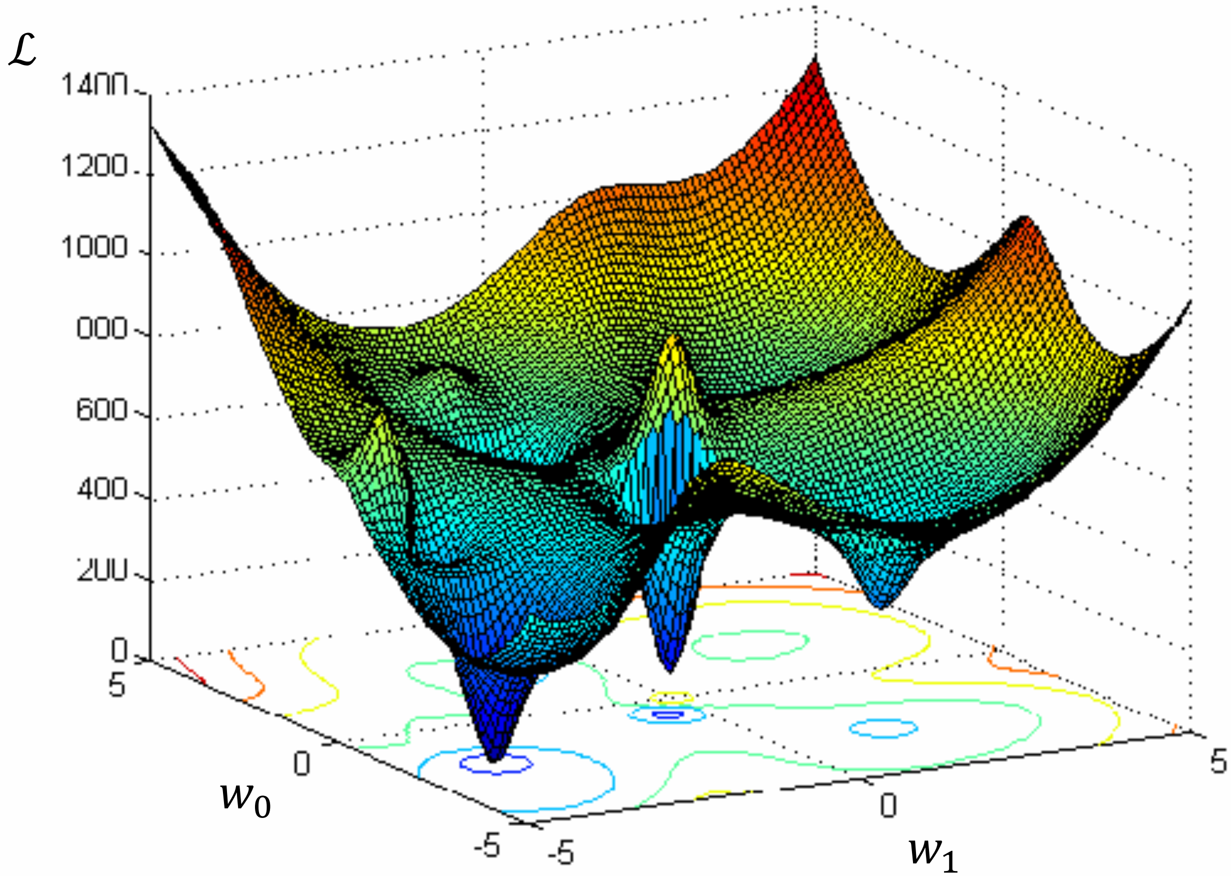
$$\mathcal{L} = y - z$$

For the current datapoint $x$ we have $y = 38$




What changes should we made to the values of $\mathbb{W}$ in order to reduce $\mathcal{L}$?

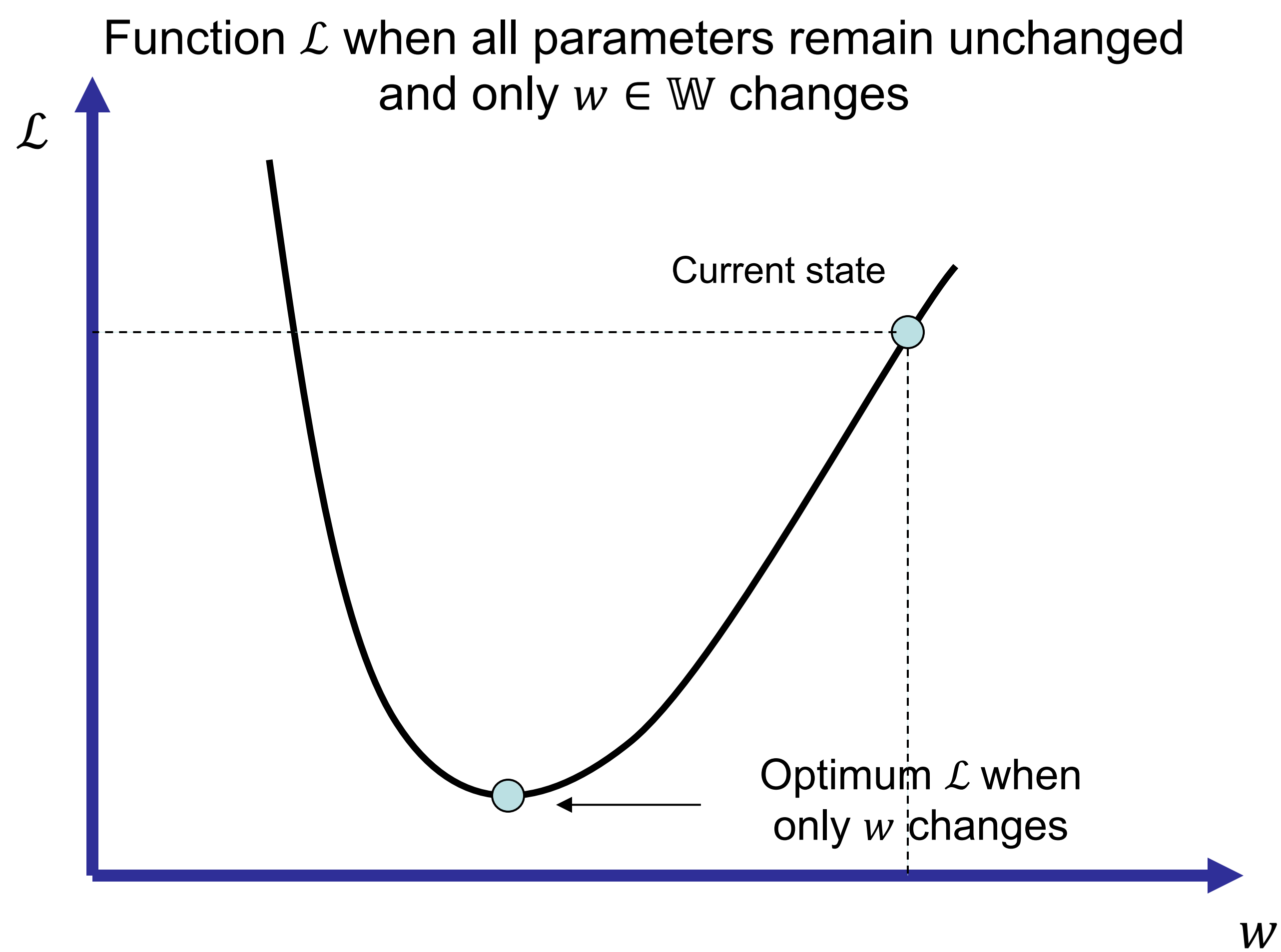
# Optimization




Source: https://www.semanticscholar.org/paper/Novel-composition-test-functions-for-numerical-Liang-Suganthan/2004e25c7239c71b18b8fa3a0fea78721f6ca69e

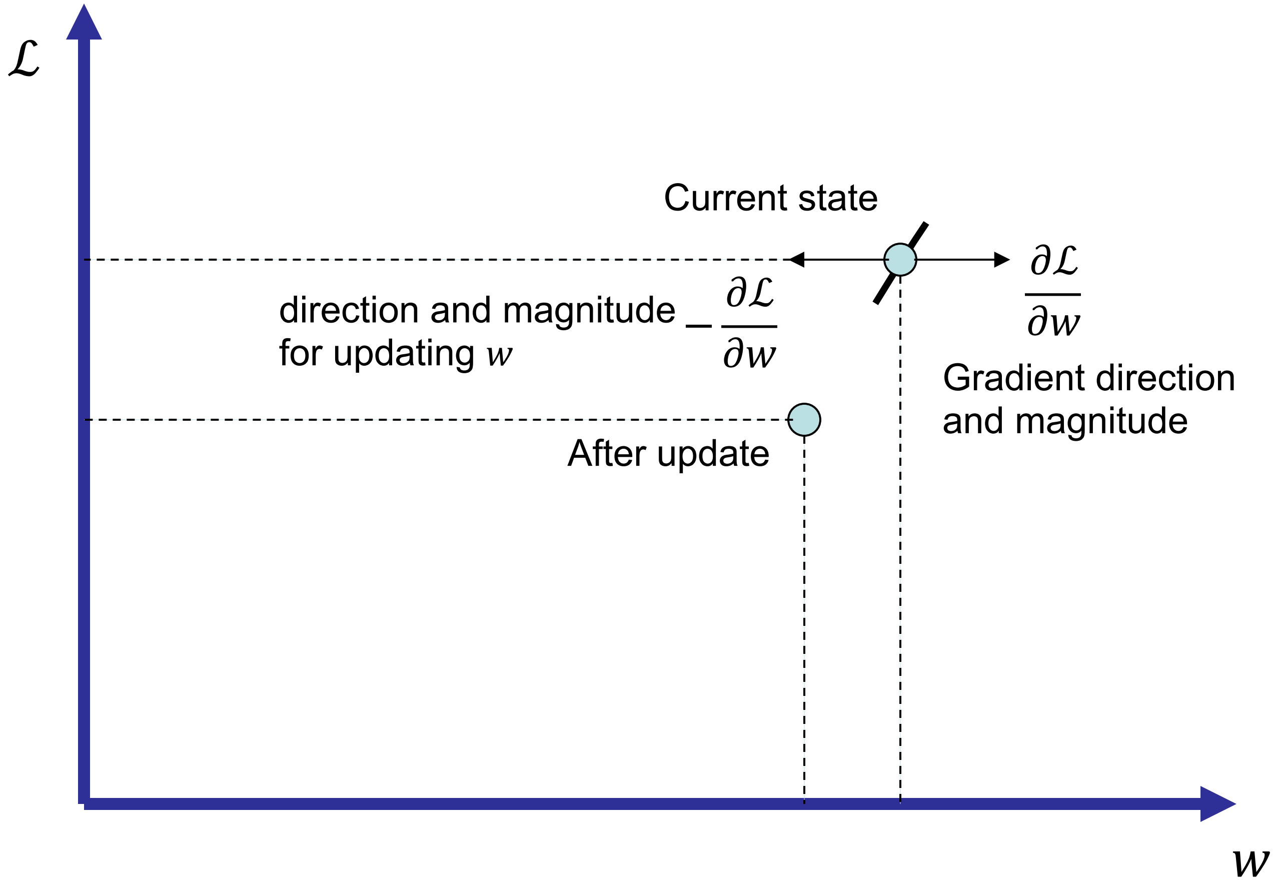
## Gradient-based optimization

- Assumption 1: optimize $\mathcal{L}$ in respect to each parameter $w \in \mathbb{W}$ independently regardless of other parameters



# Gradient Descent optimization

- Assumption 2: decide about your course of change for $w \in \mathbb{W}$ according to the local changes in $\mathcal{L}$



# Gradient Descent optimization

- We hence need the derivatives of $\mathcal{L}$ in respect to each $w \in \mathbb{W}$:

$$\nabla_{\mathbb{W}}\mathcal{L} = \left[\frac{\partial \mathcal{L}}{\partial w_0} \quad \frac{\partial \mathcal{L}}{\partial w_1} \quad \frac{\partial \mathcal{L}}{\partial w_2} \; \ldots\right]$$

- $\nabla_{\mathbb{W}}\mathcal{L}$ is often called gradient tensor, whose elements are the partial derivatives of $\mathcal{L}$ in respect to each parameter:

# Gradient Descent algorithm

- A model with parameters $\mathbb{W}$ at time step t → $\mathbb{W}^{(t)}$, **learning rate** $\eta$, and set of datapoints $\mathcal{D}$
- Loop for some epochs
  - Compute gradient tensor $\mathbb{G}$ of parameters $\mathbb{W}$ averaged over datapoints $\mathcal{D}$:

$$\mathbb{G} \leftarrow \frac{1}{|\mathcal{D}|} \nabla_{\mathbb{W}} \sum_{(\boldsymbol{x},y)\in\mathcal{D}} \mathcal{L}(\boldsymbol{x}, y; \mathbb{W})$$

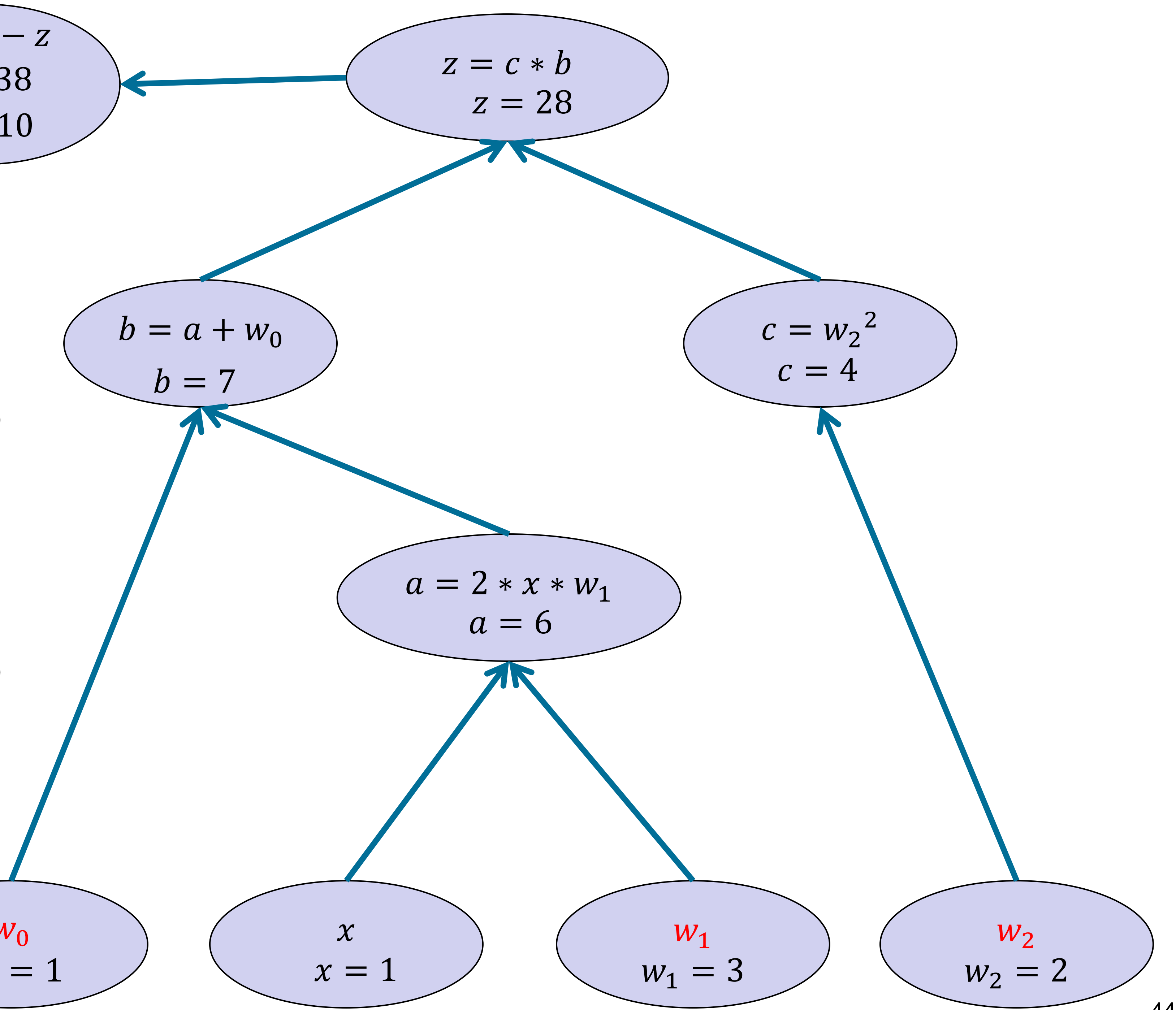
  - Update the parameters by taking steps in the opposite direction of the gradient tensor multiplied by $\eta$:

$$\mathbb{W}^{(t+1)} \leftarrow \mathbb{W}^{(t)} - \eta\mathbb{G}$$

  - Reduce learning rate (**annealing**) if some criteria are met or according to a scheduler

# Learning with Neural Networks

- Design the network's architecture
- Loop until some **exit criteria** are met
  - Sample a **(mini)batch** from training data $\mathcal{D}$
  - Execute **forward pass:** predict the output tensor of each given input tensor
  - Calculate **loss**
  - **Optimize** the network to reduce loss
    - Calculate the **gradient** of each parameter regarding the loss function using the **backpropagation** algorithm
    - **Update** parameters using their gradients




$$\frac{\partial \mathcal{L}}{\partial w_0} = ?$$

$$\frac{\partial \mathcal{L}}{\partial w_1} = ?$$

$$\frac{\partial \mathcal{L}}{\partial w_2} = ?$$

# Chain rule

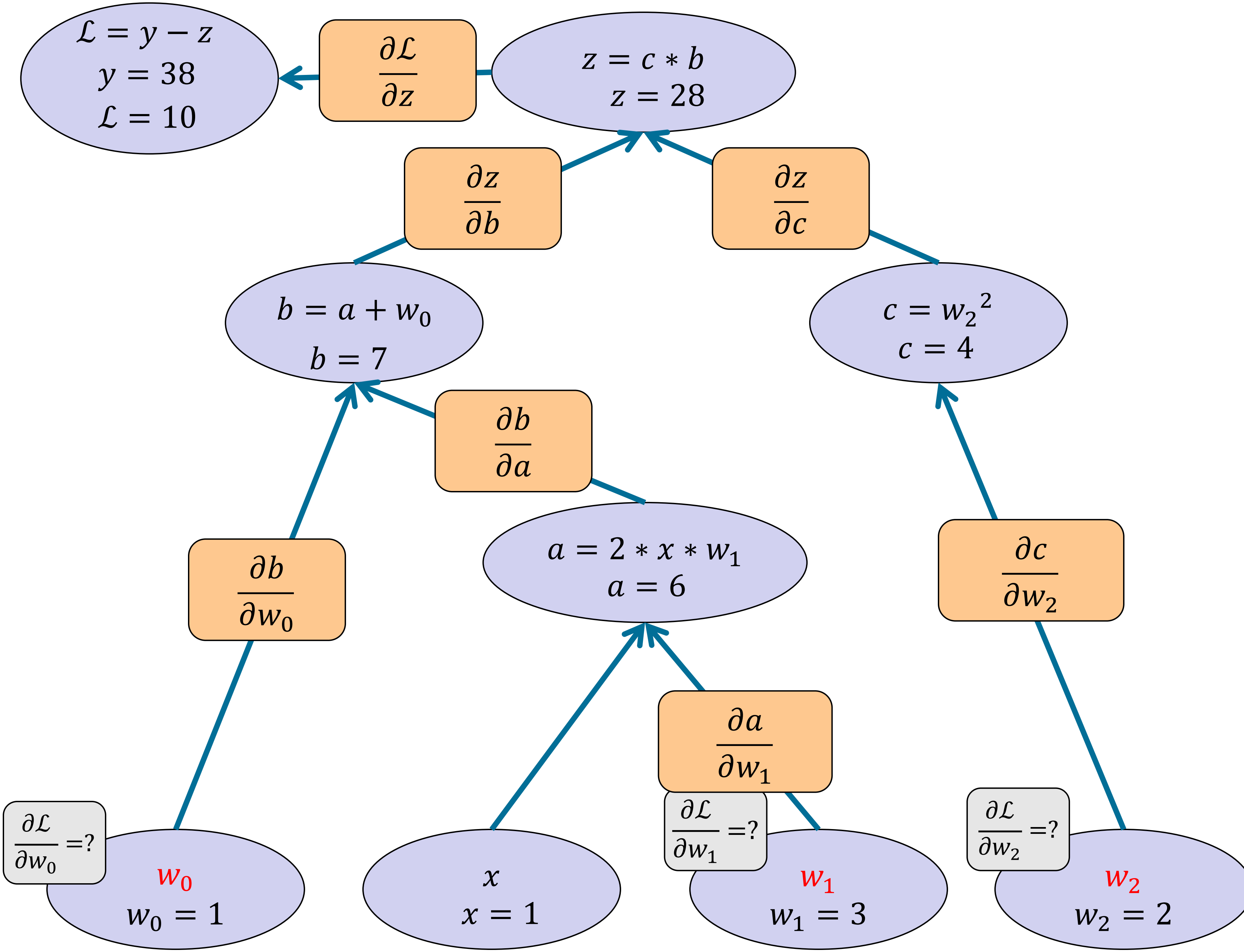
- Gradient tensor: $\nabla_{\mathbb{W}}\mathcal{L} = \left[\frac{\partial\mathcal{L}}{\partial w_0} =? \quad \frac{\partial\mathcal{L}}{\partial w_1} =? \quad \frac{\partial\mathcal{L}}{\partial w_2} =?\right]$
- Partial derivatives can be calculated using local derivates and the chain rule:

$$\frac{\partial\mathcal{L}}{\partial w_0} = \frac{\partial\mathcal{L}}{\partial z}\frac{\partial z}{\partial b}\frac{\partial b}{\partial w_0}$$

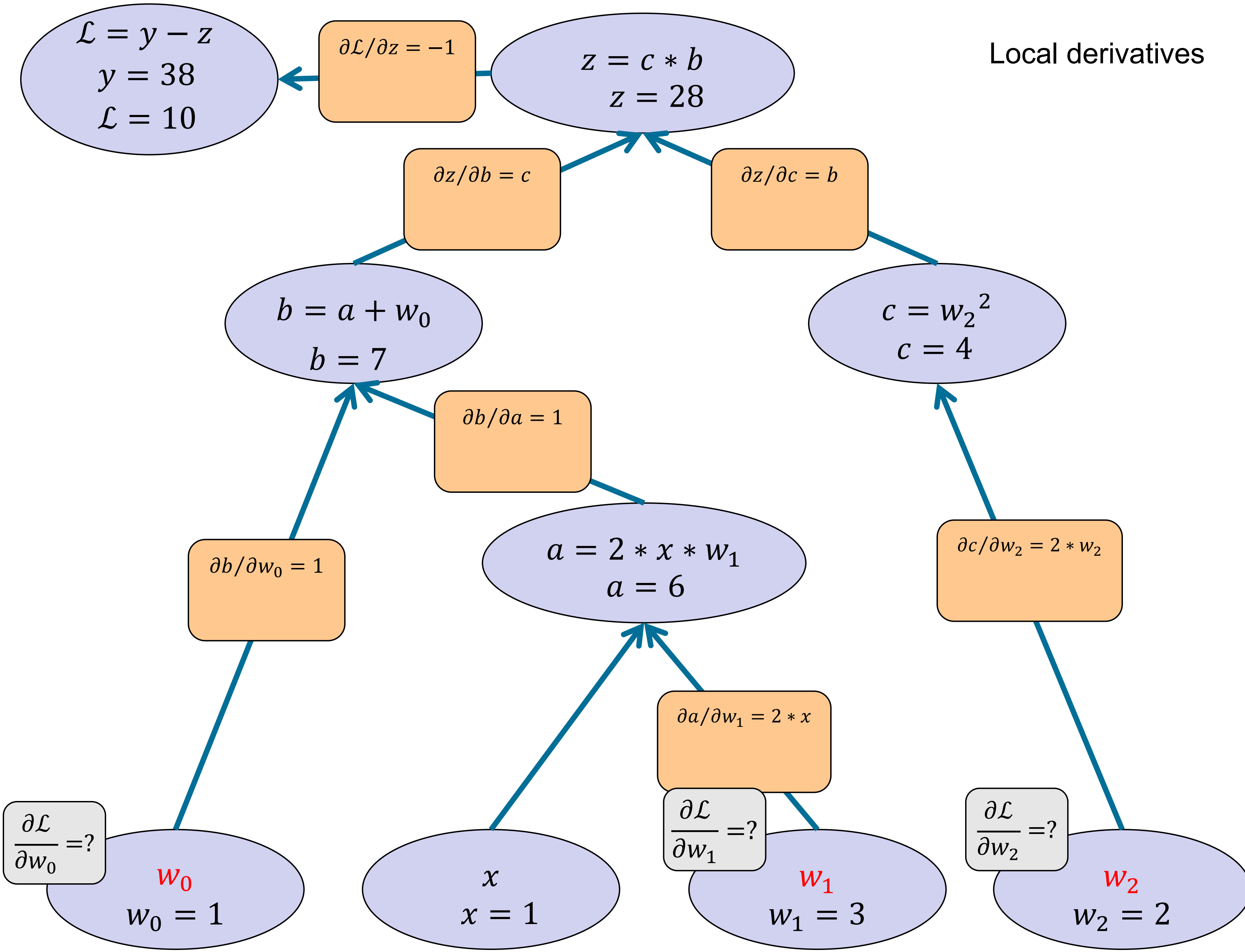
$$\frac{\partial\mathcal{L}}{\partial w_1} = \frac{\partial\mathcal{L}}{\partial z}\frac{\partial z}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{\partial w_1}$$

$$\frac{\partial\mathcal{L}}{\partial w_2} = \frac{\partial\mathcal{L}}{\partial z}\frac{\partial z}{\partial c}\frac{\partial c}{\partial w_2}$$

- Local derivates are pre-defined on each atomic operation in the neural computation graph



$\mathcal{L} = y - z$
$y = 38$
$\mathcal{L} = 10$
$\frac{\partial \mathcal{L}}{\partial z}$
$z = c * b$
$z = 28$
$\frac{\partial z}{\partial b}$
$\frac{\partial z}{\partial c}$
$b = a + w_0$
$b = 7$
$c = {w_2}^2$
$c = 4$
$\frac{\partial b}{\partial a}$
$\frac{\partial b}{\partial w_0}$
$a = 2 * x * w_1$
$a = 6$
$\frac{\partial c}{\partial w_2}$
$\frac{\partial a}{\partial w_1}$
$\frac{\partial \mathcal{L}}{\partial w_0} =?$
$w_0$
$w_0 = 1$
$x$
$x = 1$
$\frac{\partial \mathcal{L}}{\partial w_1} =?$
$w_1$
$w_1 = 3$
$\frac{\partial \mathcal{L}}{\partial w_2} =?$
$w_2$
$w_2 = 2$



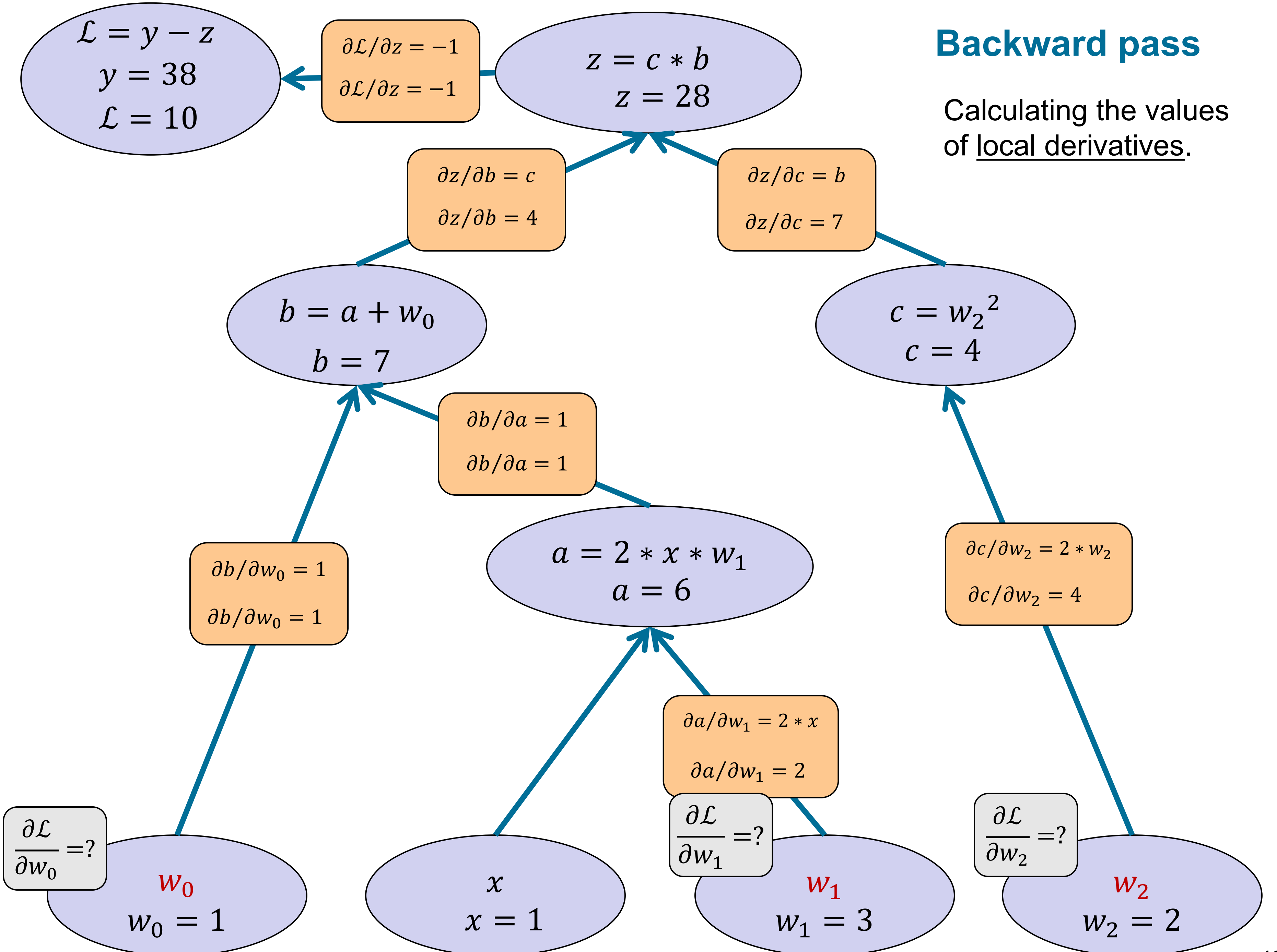
Local derivatives
$\mathcal{L} = y - z$
$y = 38$
$\mathcal{L} = 10$
$\partial\mathcal{L}/\partial z = -1$
$z = c * b$
$z = 28$
$\partial z/\partial b = c$
$\partial z/\partial c = b$
$b = a + w_0$
$b = 7$
$c = {w_2}^2$
$c = 4$
$\partial b/\partial a = 1$
$\partial b/\partial w_0 = 1$
$a = 2 * x * w_1$
$a = 6$
$\partial c/\partial w_2 = 2 * w_2$
$\partial a/\partial w_1 = 2 * x$
$\frac{\partial\mathcal{L}}{\partial w_0} = ?$
$w_0$
$w_0 = 1$
$x$
$x = 1$
$\frac{\partial\mathcal{L}}{\partial w_1} = ?$
$w_1$
$w_1 = 3$
$\frac{\partial\mathcal{L}}{\partial w_2} = ?$
$w_2$
$w_2 = 2$

# Backward pass

- Tracing the computation graph from top to bottom and calculating the values of local derivatives
- It means that:
  - We need to keep the values of all intermediate variables after forward pass
  - For the local derivative of every atomic operation, we now have a new stored value

## Backward pass

Calculating the values of local derivatives.



## Backpropagation

Calculating partial derivatives:

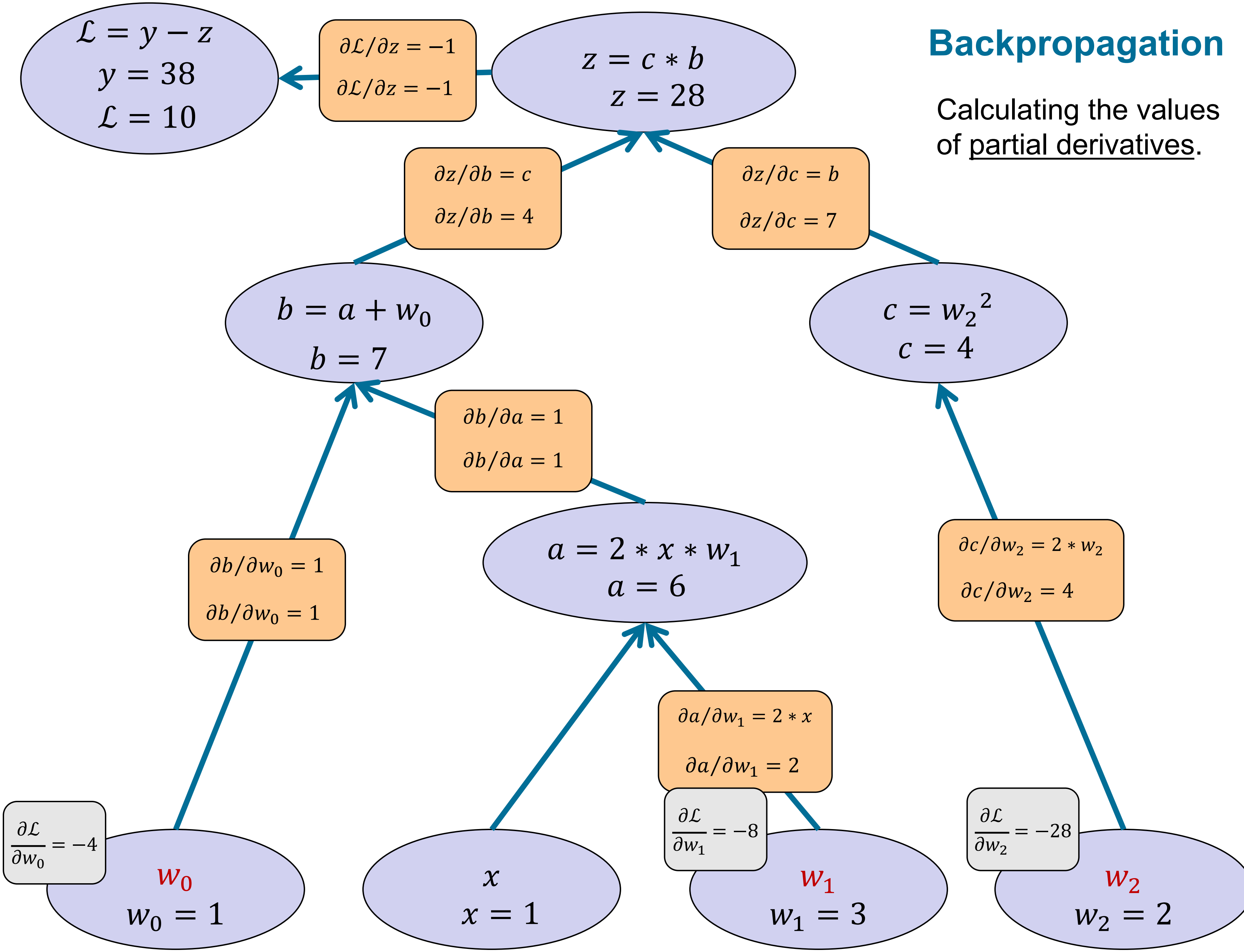
$$\frac{\partial \mathcal{L}}{\partial w_0} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial b}\,\frac{\partial b}{\partial w_0} = -1\times 4\times 1 = -4$$

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial b}\,\frac{\partial b}{\partial a}\frac{\partial a}{\partial w_1} = -1\times 4\times 1\times 2 = -8$$

$$\frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial c}\,\frac{\partial c}{\partial w_2} = -1\times 7\times 4 = -28$$

# Backpropagation

Calculating the values of partial derivatives.



# Learning with Neural Networks

- Design the network's architecture
- Loop until some **exit criteria** are met
  - Sample a **(mini)batch** from training data $\mathcal{D}$
  - Execute **forward pass:** predict the output tensor of each given input tensor
  - Calculate **loss** function of the (mini)batch
  - **Optimize** the network to reduce loss
    - Calculate the **gradient** of each parameter regarding the loss function using the **backpropagation** algorithm
    - **Update** parameters using their gradients

# Gradient Descent algorithm – recap

- A model with parameters $\mathbb{W}$ at time step t $\rightarrow \mathbb{W}^{(t)}$, **learning rate** $\eta$, and set of datapoints $\mathcal{D}$
- Loop for some epochs
  - Compute gradient tensor $\mathbb{G}$ of parameters $\mathbb{W}$ averaged over datapoints $\mathcal{D}$:

$$\mathbb{G} \leftarrow \frac{1}{|\mathcal{D}|} \nabla_{\mathbb{W}} \sum_{(\boldsymbol{x}, y) \in \mathcal{D}} \mathcal{L}(\boldsymbol{x}, y; \mathbb{W})$$

  - Update the parameters by taking steps in the <u>opposite</u> direction of the gradient tensor multiplied by $\eta$:

$$\mathbb{W}^{(t+1)} \leftarrow \mathbb{W}^{(t)} - \eta \mathbb{G}$$

  - Reduce learning rate (**annealing**) if some criteria are met or according to a scheduler

# Batch

- In (vanilla) Gradient Descent, first all data points are processed, and their gradients are aggregated, and then a small parameter update is made
  - Training can take very long time
  - Training is not stochastic

- Batch/Mini-batch
  - A (small) set of data to be processed together
  - Suitable for multi-processing capabilities of GPUs

- **Stochastic Gradient Descent**
  - In each step, we process a (mini-)batch of data, calculate their gradients, and update parameters
  - Typical setting for training deep learning models

# (Mini-batch) Stochastic Gradient Descent algorithm

- A model with parameters $\mathbb{W}$ at time step t $\rightarrow \mathbb{W}^{(t)}$, **learning rate** $\eta$, and set of datapoints $\mathcal{D}$
- Loop until some exit criteria are met
  - $\widehat{\mathcal{D}}$ is the set of datapoints in the **minibatch**
  - Compute gradient tensor $\mathbb{G}$ of parameters $\mathbb{W}$ averaged over batch datapoints $\widehat{\mathcal{D}}$:

$$\mathbb{G} \leftarrow \frac{1}{|\widehat{\mathcal{D}}|} \nabla_{\mathbb{W}} \sum_{(\boldsymbol{x}, y) \in \widehat{\mathcal{D}}} \mathcal{L}(\boldsymbol{x}, y; \mathbb{W})$$

  - Update the parameters by taking steps in the <u>opposite</u> direction of the gradient tensor multiplied by $\eta$:

$$\mathbb{W}^{(t+1)} \leftarrow \mathbb{W}^{(t)} - \eta \mathbb{G}$$

  - Reduce learning rate (**annealing**) if some criteria are met or according to a scheduler

# Other gradient-based optimizations

- Some limitations of the mentioned SGD algorithms
  - Choosing learning rate is hard
  - Choosing annealing method/rate is hard
  - Same learning rate is applied to all parameters
  - Can get trapped in non-optimal local minima and saddle points

- Some other commonly used algorithms:
  - Nestrov accelerated gradient
  - Adagrad
  - Adam