Deep Learning
Summer semester ‘24

3. Optimization & Training

Machine Learning Components

- Any ML algorithm/approach has to have the following three components:
  - **Model**
  - **Objective**
  - **Optimization algorithm**

Machine Learning Components

- Any ML algorithm/approach has three components:

1. Model

- A set of functions among which we're looking for the „best" one

$$H = \{h(\mathbf{x}|\boldsymbol{\theta})\}_{\boldsymbol{\theta}}$$

- **Hypothesis** h = a concrete function obtained for some concrete values of $\boldsymbol{\theta}$
- Model = set of hypotheses

Machine Learning Components

- Any ML algorithm/approach has three components:

2. Objective

- We're looking from the best hypothesis h in the model H = $\{h(\mathbf{x}|\boldsymbol{\theta})\}_{\boldsymbol{\theta}}$
  - Q: But „best" according to what?
- **Objective** J is a function that quantifies how good/bad a hypothesis h is
  - Usually J is a „loss function" that we're minimizing
- We're looking for h (that is, values of parameters $\boldsymbol{\theta}$) that maximize or minimize the objective J

$$h^* = \text{argmin}_{h \in H} J(h(\mathbf{x}|\boldsymbol{\theta}))$$
$$\boldsymbol{\theta}^* = \text{argmin}_{\boldsymbol{\theta}} J(h(\mathbf{x}|\boldsymbol{\theta}))$$

- ML thus amounts to solving optimization problems

Machine Learning Components

- Any ML algorithm/approach has three components:

3. Optimization algorithm

- An exact algorithm that we use to solve the optimization problem

$$\boldsymbol{\theta}^* = \text{argmin}_{\boldsymbol{\theta}} J(h(\mathbf{x}|\boldsymbol{\theta}))$$

- Selection/type of the optimization algorithm depends on the two functions – the model H and the objective *J*

Optimization of a DL model

- $D = \{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}_{i = \{1, \ldots, B\}}$ → **training dataset**
  - We rarely/never optimize based on the whole training dataset at once, but on the small subset of B examples, called batch, one batch at a time

- $h(\mathbf{x}|\boldsymbol{\theta}) = lay_n(lay_{n-1}(\ldots(lay_1(\mathbf{x}|\boldsymbol{\theta}_{L1})|\boldsymbol{\theta}_{L2})\ldots)|\boldsymbol{\theta}_{Ln})$
  - Our DL **model** (aka „architecture”), composition of parameterized functions

- $L(h(\mathbf{x}|\boldsymbol{\theta}), \mathbf{y})$ → loss function (for a single instance)
- $J = \frac{1}{B}\sum_{i=1}^{B} L(h(\mathbf{x}^{(i)}|\boldsymbol{\theta}), \mathbf{y}^{(i)})$ → objective function to minimize w.r.t. $\boldsymbol{\theta}$

$$\boldsymbol{\theta}^* = \text{argmin}_{\boldsymbol{\theta}}\, J$$

Fahrplan

- **Gradient-Based Optimization**
- **Backpropagation**
- **Automatic Differentiation**
- **Training in Batches**
- **Regularization**

Gradient-Based Optimization

- We resort to (in DL, typically unconstrained) numerical optimization

Numerical Optimization

Numerical optimization refers to optimizing real-valued functions $f(\boldsymbol{\theta}): \mathbb{R}^n \rightarrow \mathbb{R}$, $\boldsymbol{\theta} = \theta_1, \theta_2, \ldots, \theta_n \in \mathbb{R}$. This means finding values $\theta_1, \theta_2, \ldots, \theta_n$ for which f obtains the minimal or maximal value.

- Concretely, optimization of deep NNs relies on gradient-based optimization, i.e., variants of gradient descent
- **Gradient descent** – optimization algorithm that uses function differentiation (w.r.t. parameters) to find the minimum of a function

Gradient-Based Optimization

- Objective *J* needs to be differentiable* w.r.t. all parameters $\boldsymbol{\theta} = \{\theta_1, \theta_2, \ldots, \theta_n\}$

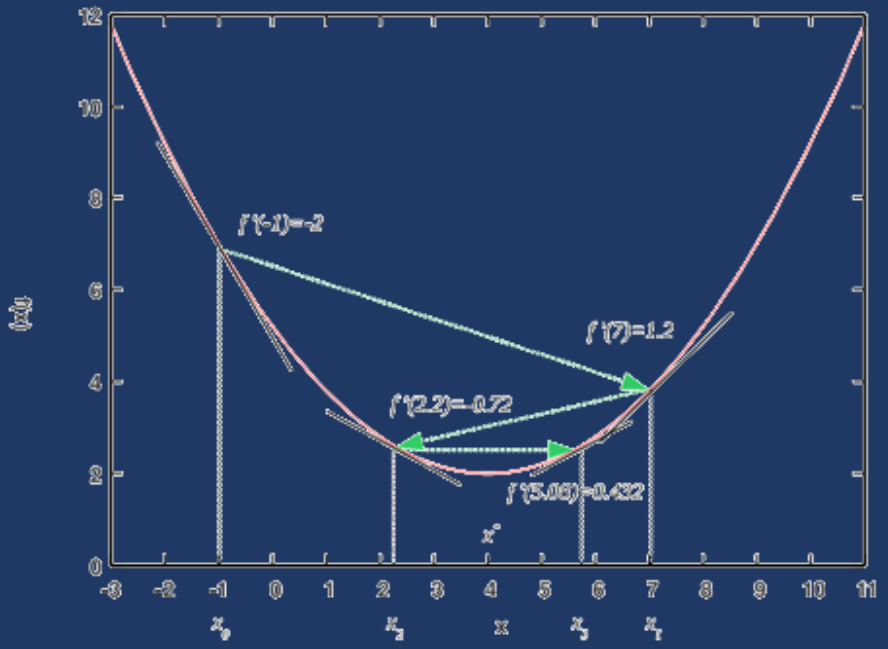
Gradient of a differentiable function

A function of multiple parameters $f(\boldsymbol{\theta} = \theta_1, \theta_2, \ldots, \theta_n)$ is differentiable if its **gradient** $\nabla_{\boldsymbol{\theta}} f$ – a vector of **partial derivatives** $\nabla_{\boldsymbol{\theta}} f = [\frac{\partial f}{\partial \theta_1}, \frac{\partial f}{\partial \theta_2}, \ldots, \frac{\partial f}{\partial \theta_n}]$ – exists for every point of the input domain.

Gradient-Based Optimization

- **Gradient descent** is a method that moves the parameter values in the direction opposite of the function's gradient in the current point
  - This is guaranteed to lead to a global **minimum** only for **convex** functions*

- Objectives of DL models are never globally convex
  - No guarantee of „global" minimum
  - But we hope for a good enough „local" minimum, i.e., to find such values θ for which J is „small enough"



Gradient Descent

Gradient Descent

Gradient descent (sometimes also called steepest descent) is an iterative algorithm for (continuous) optimization that finds a minimum of a convex (single) differentiable function.

- In each iteration GD moves the values of parameters $\boldsymbol{\theta} = \{\theta_1, \theta_2, \ldots, \theta_n\}$ in the direction **opposite** to the gradient in the current point

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \eta \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}^{(k)})$$

- $\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$ – value of the gradient (a vector of same dimensionality as $\boldsymbol{\theta}$) of the function f in the point $\boldsymbol{\theta}$
- η – learning rate, defines by how much to move the parameters in the direction opposite of the gradient

Gradient-Based Optimization

- So, what we need to compute for gradient descent is

$$\nabla_{\boldsymbol{\theta}} J = \nabla_{\boldsymbol{\theta}} \left[\frac{1}{B} \sum_{i=1}^{B} \mathrm{L}(\mathrm{h}(\mathbf{x}^{(\mathrm{i})} | \boldsymbol{\theta}), \mathbf{y}^{(\mathrm{i})})\right]$$

- Or, put differently, $\frac{\partial J}{\partial \theta_{\mathrm{i}}}$ for each parameter θ_{i} in $\boldsymbol{\theta}$

$$= \frac{\partial}{\partial \theta_{\mathrm{i}}} \left[\frac{1}{B} \sum_{i=1}^{B} \mathrm{L}(\mathrm{h}(\mathbf{x}^{(\mathrm{i})} | \boldsymbol{\theta}), \mathbf{y}^{(\mathrm{i})})\right]$$

$$= \frac{1}{B} \sum_{i=1}^{B} \frac{\partial}{\partial \theta_{\mathrm{i}}} \mathrm{L}(\mathrm{h}(\mathbf{x}^{(\mathrm{i})} | \boldsymbol{\theta}), \mathbf{y}^{(\mathrm{i})})$$

Gradient-Based Optimization

- So, to update some parameter θ_i we would need to compute in closed-form the partial derivative of the loss L w.r.t. θ_i: $\frac{\partial L}{\partial \theta_i}$

- But our L is a complex composition of parametrized functions (i.e., model layers)
  - Because it's computed on the output of the model, $h(\mathbf{x}^{(i)}|\boldsymbol{\theta})$

- In other words:

$$\frac{\partial J}{\partial \theta_i} = \frac{1}{B}\sum_{i=1}^{B} \frac{\partial}{\partial \theta_i} L(\text{lay}_n(\text{lay}_{n-1}(\ldots(\text{lay}_1(\mathbf{x}|\boldsymbol{\theta}_{L1})|\boldsymbol{\theta}_{L2})\ldots)|\boldsymbol{\theta}_{Ln}), \mathbf{y}^{(i)})$$

Fahrplan

- Gradient-Based Optimization
- Backpropagation
- Automatic Differentiation
- Training in Batches
- Regularization

Backpropagation

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. Nature, 323(6088), 533-536.

$$\frac{\partial}{\partial \theta_i} L(h(\mathbf{x}^{(i)} | \boldsymbol{\theta}), \mathbf{y}^{(i)})$$

$$= \frac{\partial}{\partial \theta_i} L(lay_n(lay_{n-1}(\ldots(lay_1(\mathbf{x} | \boldsymbol{\theta}_{L1}) | \boldsymbol{\theta}_{L2})\ldots) | \boldsymbol{\theta}_{Ln}), \mathbf{y}^{(i)})$$

- Let $\boldsymbol{\theta}_{ij}$ denote the j-th parameter of the i-th layer of the model
- Computing $\frac{\partial L}{\partial \theta_{ij}}$ in closed form for params $\boldsymbol{\theta}_{Nj}$ of the last layer is easy
- But it gets progressively more cumbersome and difficult the „deeper” in the model the layer of the parameter is

Backpropagation

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. Nature, 323(6088), 533-536.

- Computing $\frac{\partial L}{\partial \theta}$ in closed form gets progressively more difficult the „further away" the parameter (i.e., its layer) is from the loss"

- **Backpropagation** leverages the **chain rule of differentiation** to avoid the difficult computation of closed-form gradients for „deeper" parameters
  - Gradients of parameters from k-th layer are estimated from gradients of parameters from layer k+1

$$\frac{\partial L}{\partial \theta_{ij}} = \frac{\partial L}{\partial h} \frac{\partial h}{\partial lay_n} \frac{\partial lay_n}{\partial lay_{n\_1}} \cdots \frac{\partial lay_{i+1}}{\partial lay_i} \frac{\partial lay_i}{\partial \theta_{ij}}$$

Backpropagation

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. Nature, 323(6088), 533-536.

$$\frac{\partial L}{\partial \theta_{ij}} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial lay_n}\frac{\partial lay_n}{\partial lay_{n\_1}} \cdots \frac{\partial lay_{i+1}}{\partial lay_i}\frac{\partial lay_i}{\partial \theta_{ij}}$$

- For some (j-th) parameter $\theta_{n',j}$ of the last, n-th layer:

$$\frac{\partial L}{\partial \theta_{n'j}} = \underbrace{\frac{\partial L}{\partial h}\frac{\partial h}{\partial lay_n}}_{\delta_N}\frac{\partial lay_n}{\partial \theta_{n'j}}$$

Backpropagation

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. Nature, 323(6088), 533-536.

$$\frac{\partial L}{\partial \theta_{ij}} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial lay_n}\frac{\partial lay_n}{\partial lay_{n\_1}} \cdots \frac{\partial lay_{i+1}}{\partial lay_i}\frac{\partial lay_i}{\partial \theta_{ij}}$$

- For some (j-th) parameter $\theta_{n\_1,j}$ of the penultimate, (n-1)-th layer:

$$\frac{\partial L}{\partial \theta_{N\_1,j}} = \underbrace{\underbrace{\frac{\partial L}{\partial h}\frac{\partial h}{\partial lay_n}}_{\delta_N}\frac{\partial lay_n}{\partial lay_{n\_1}}}_{\delta_{N-1}}\frac{\partial lay_{n\_1}}{\partial \theta_{n\_1,j}}$$

Backpropagation

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. Nature, 323(6088), 533-536.

$$\frac{\partial L}{\partial \theta_{n\_1'j}} = \delta_N \frac{\partial lay_n}{\partial lay_{n\_1}} \frac{\partial lay_{n\_1}}{\partial \theta_{n\_1'j}}$$

…

$$\frac{\partial L}{\partial \theta_{i'j}} = \delta_{i+1} \frac{\partial lay_{i+1}}{\partial lay_i} \frac{\partial lay_i}{\partial \theta_{i'j}}$$

…

$$\frac{\partial L}{\partial \theta_{1'j}} = \delta_2 \frac{\partial lay_2}{\partial lay_1} \frac{\partial lay_1}{\partial \theta_{1'j}}$$

- With backprop we avoid having to explicitly compute derivatives for all layers/parameters
- But we have to compute gradients in the inverse order of layers ☺
- (part of the) gradient of a subsequent layer needed for the computation of the gradient of the preceding layer

Backpropagation – example

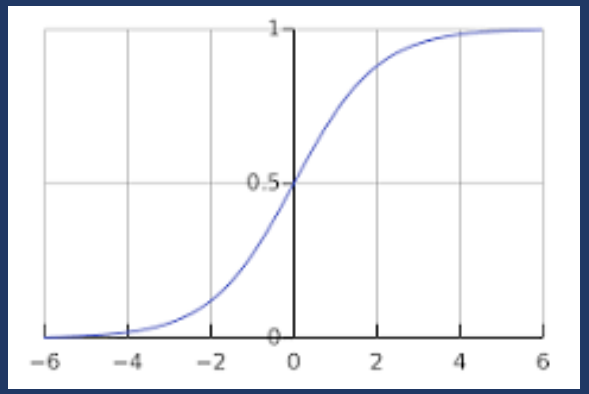
- Model: 2-layer feed-forward network with sigmoid activation
  - scalar output

$$h(\mathbf{x}|\boldsymbol{\theta}) = \sigma(\sigma(\mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2)$$

  - $\mathbf{x} \in \mathbb{R}^d$
  - $\boldsymbol{\theta} = \{ \mathbf{W}_1 \in \mathbb{R}^{d \times H}, \mathbf{b}_1 \in \mathbb{R}^H, \mathbf{W}_2 \in \mathbb{R}^{H \times 1}, \mathbf{b}_2 \in \mathbb{R} \}$

- Loss function: binary cross-entropy loss (BCE)
  - $L(h(\mathbf{x}|\boldsymbol{\theta}), y) = -[\, y \, ln(h(\mathbf{x}|\boldsymbol{\theta})) + (1 - y) \, ln(1-h(\mathbf{x}|\boldsymbol{\theta}))]$

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



Backpropagation – example

- Model: $h(\mathbf{x}|\boldsymbol{\theta}) = \sigma(\sigma(\mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + b_2)$

- Loss function: $L(h(\mathbf{x}|\boldsymbol{\theta}), y) = -[\, y \ln h + (1 - y) \ln(1-h)]$
- Last (second) layer parameters:
- $\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial h} \frac{\partial h}{\partial lay_2} \frac{\partial lay_2}{\partial W_2}$ and $\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial h} \frac{\partial h}{\partial lay_2} \frac{\partial lay_2}{\partial b_2}$

1. $\frac{\partial L}{\partial h} = \frac{-\partial[\, y \ln h + (1 - y) \ln(1-h)]}{\partial h}$

$$= -\left(\frac{y}{h} + \frac{1-y}{1-h}(-1)\right) = \frac{h-y}{h(1-h)}$$

Backpropagation – example

- Model: $h(\mathbf{x}|\boldsymbol{\theta}) = \sigma(\sigma(\mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + b_2)$
  $= lay_2(lay_1(\mathbf{x}))$

$lay_1(\mathbf{a}) = \sigma(\mathbf{a}\mathbf{W}_1 + \mathbf{b}_1)$
$lay_2(\mathbf{a}) = \sigma(\mathbf{a}\mathbf{W}_2 + \mathbf{b}_2)$

- Last (second) layer parameters
- $\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial lay_2}\frac{\partial lay_2}{\partial W_2}$ and $\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial lay_2}\frac{\partial lay_2}{\partial b_2}$

2. $\frac{\partial h}{\partial lay_2} = 1$

$h = lay_2$

Output of lay_2 is the output of the whole model

Backpropagation – example

- Model: $h(\mathbf{x}|\theta) = \sigma(\sigma(\mathbf{x}\mathbf{W_1} + \mathbf{b_1})\mathbf{W_2} + b_2)$
  $= lay_2(lay_1(\mathbf{x}))$
- First, last (second) layer parameters
- $$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial lay_2}\frac{\partial lay_2}{\partial W_2}$$

$lay_1(\mathbf{a}) = \sigma(\mathbf{a}\mathbf{W_1} + \mathbf{b_1})$
$lay_2(\mathbf{a}) = \sigma(\mathbf{a}\mathbf{W_2} + \mathbf{b_2})$

$\sigma(a)' = \sigma(a) * (1 - \sigma(a))$

Sigmoid has a very nice derivative ☺

3. $$\frac{\partial lay_2}{\partial W_2} = \frac{\partial[\sigma(\sigma(\mathbf{x}\mathbf{W_1} + \mathbf{b_1})\mathbf{W_2} + b_2)]}{\partial W_2}$$

$$= lay_2 * (1 - lay_2)\frac{\partial[\boldsymbol{lay_1}\mathbf{W_2} + b_2]}{\partial W_2}$$

$$= lay_2 * (1 - lay_2) * \boldsymbol{lay_1}$$

vector

$$\frac{\partial lay_2}{\partial b_2} = lay_2 * (1 - lay_2) * 1$$

Backpropagation – example

- Model: $h(\mathbf{x}|\boldsymbol{\theta}) = \sigma(\sigma(\mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + b_2)$
- Loss function: $L(h(\mathbf{x}|\boldsymbol{\theta}), y) = -[\, y \ln h + (1 - y) \ln(1-h)]$
- Last (second) layer parameters:

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial lay_2}\frac{\partial lay_2}{\partial W_2}$$

$$= \underbrace{\frac{h-y}{h(1-h)} * 1 * lay_2 * (1 - lay_2)}_{\delta_2} * lay_1$$

Backpropagation – example

- Model: $h(\mathbf{x}|\boldsymbol{\theta}) = \sigma(\sigma(\mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + b_2)$

- Loss function: $L(h(\mathbf{x}|\boldsymbol{\theta}), y) = -[\, y \, ln \, h + (1 - y) \, ln(1-h)]$
- First layer parameters:
- $$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial h} \frac{\partial h}{\partial lay_2} \frac{\partial lay_2}{\partial lay_1} \frac{\partial lay_1}{\partial W_1}$$

$$\frac{\partial L}{\partial W_1} = \mathbf{x}^T * \boldsymbol{\delta}_2 * \mathbf{W}_2 * lay_1 * (1 - lay_1)$$

1. $$\frac{\partial lay_2}{\partial lay_1} = lay_2 * (1 - lay_2) * \mathbf{W}_2$$
2. $$\frac{\partial lay_1}{\partial W_1} = lay_1 * (1 - lay_1) * \mathbf{x}^T$$

Explanation video (from 9:00)

Fahrplan

- Gradient-Based Optimization
- Backpropagation
- Automatic Differentiation
- Training in Batches
- Regularization

Automatic Differentiation

- In our backpropagation example, we manually differentiated
  - Tedious, error-prone

- Other options (all in principle „automatic”)
  - Numerical differentiation
    - (–) Numerical instabilities, problem-specific selection of learning rates

  - Symbolic differentiation
    - Automation of manual diff., computer applies diff. rules step by step
    - Result is an explicit (symbolic, closed form) derivative: (–) expression swell
    - (–) Model has to implemented with „pure functions”, no common programming constructs loops, conditions, ... (no discrete computation steps)
    - Example library: SymPy

  - **Automatic differentiation**

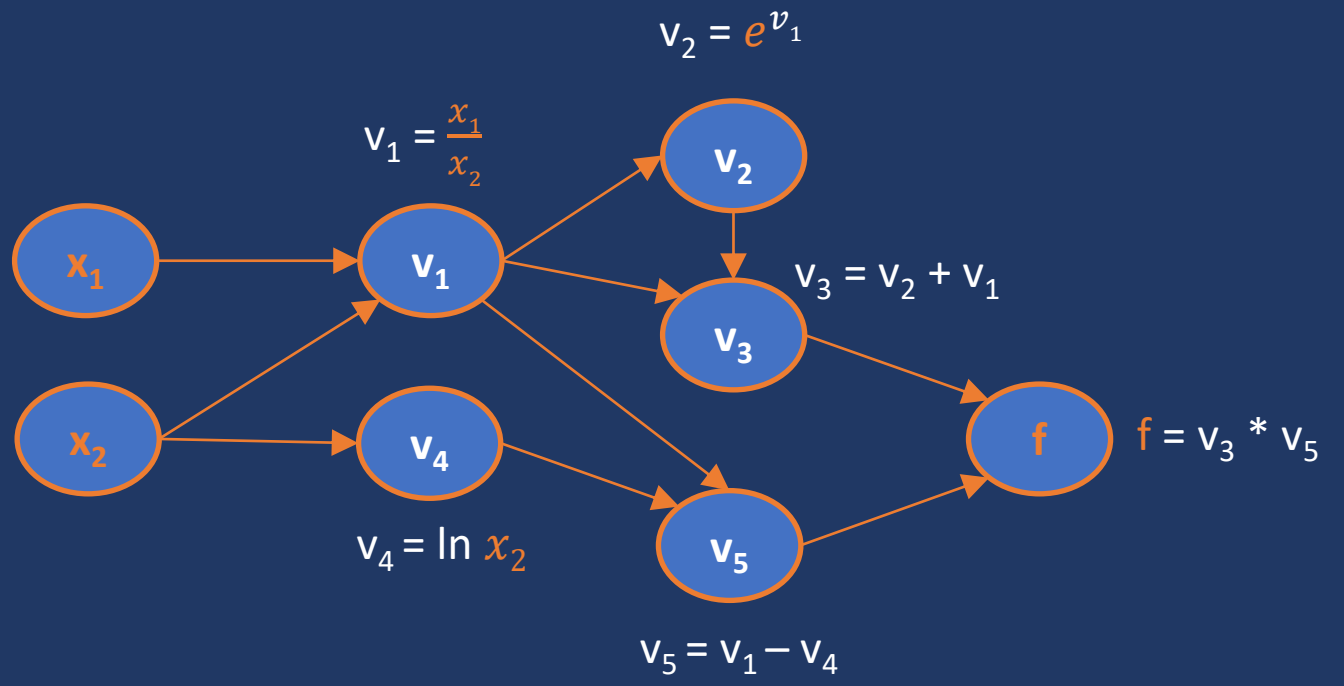
Automatic Differentiation

- Does not need the symbolic formula of the derivative
  - only computes values of the derivatives in concrete points
- **Computation graph** = intermediate variables in the code and how they are computed from one another
- Computation graph then used to propagate computation of gradients
  - Forward mode
  - Reverse mode

Computation Graph

- Example function of two variables: $f(x_1, x_2) = \left(e^{\frac{x_1}{x_2}} + \frac{x_1}{x_2}\right) * \left(\frac{x_1}{x_2} - \ln x_2\right)$
- We introduce variables for intermediate steps

- $v_1 = \frac{x_1}{x_2}$
- $v_2 = e^{\frac{x_1}{x_2}} = e^{v_1}$
- $v_3 = v_2 + v_1$
- $v_4 = \ln x_2$
- $v_5 = \frac{x_1}{x_2} - \ln x_2 = v_1 - v_4$
- $f = v_3 * v_5$

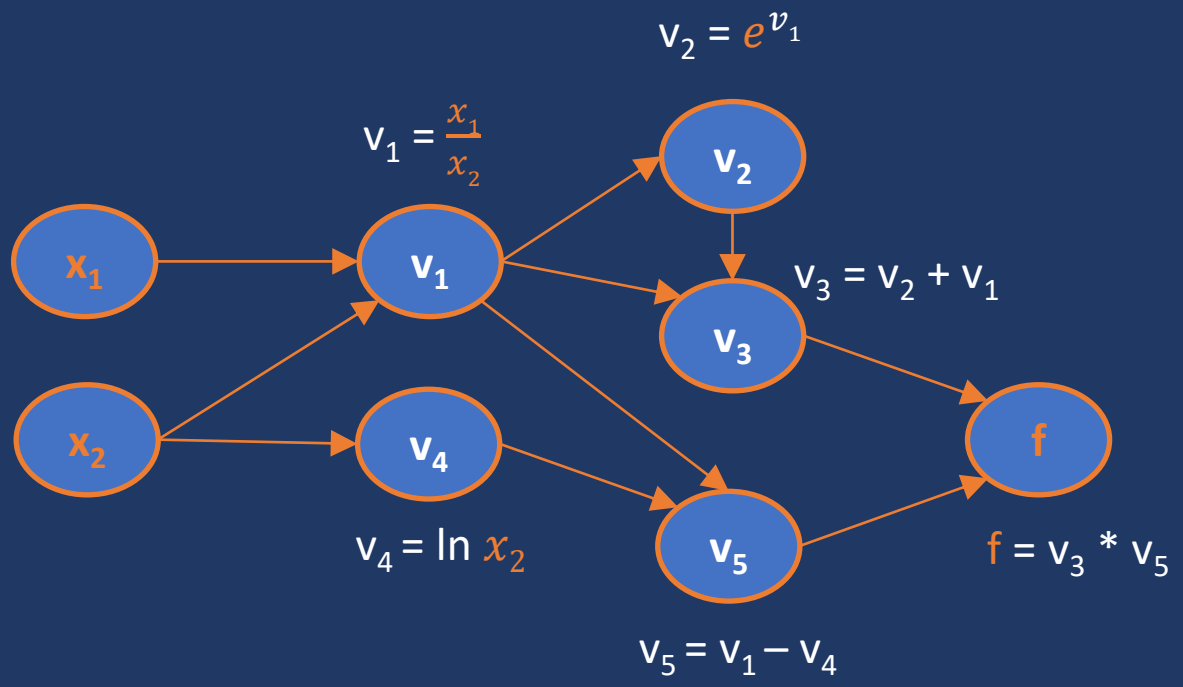


Automatic Differentiation: Forward Mode

- Forward mode
  - For each input variable, we compute both the value of each node as well as the value of the derivative of the intermediate node w.r.t that variable

- Start from: $(x_1, x_2) = (0.5, 1)$, compute $\frac{\partial f}{\partial x_1}$
- We compute v_i and $v'_i = \frac{\partial v_i}{\partial x_1}$

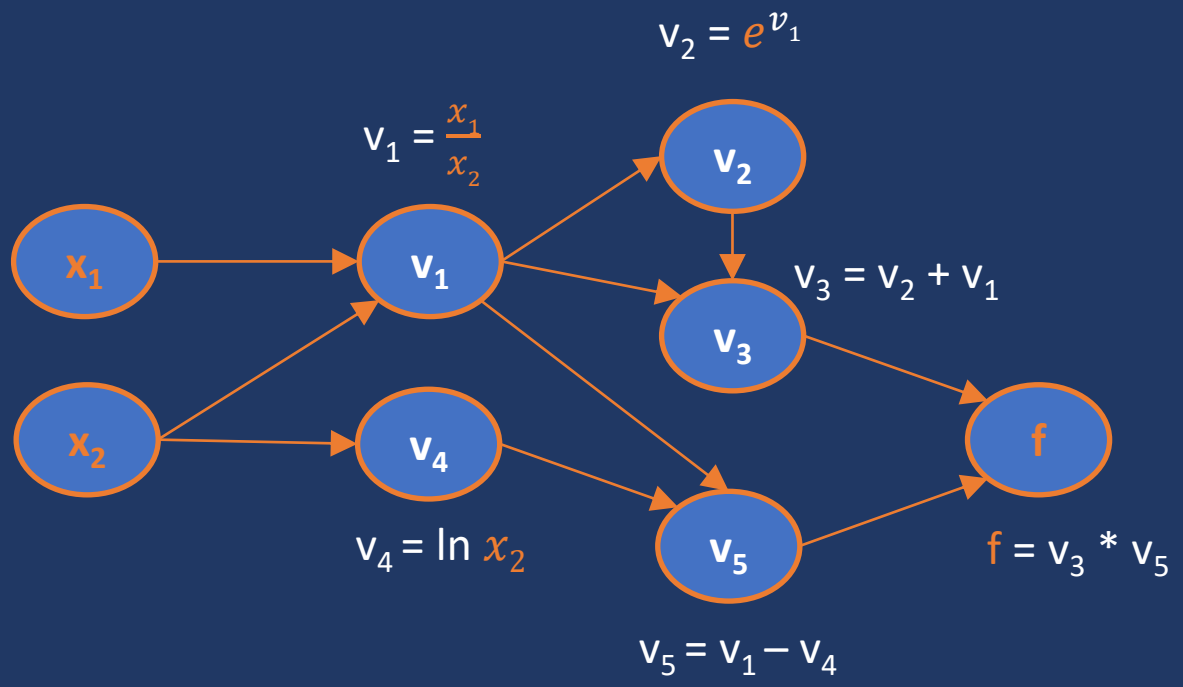
- $v_1 = 0.5$, $v'_1 = 1/x_2 = 1$
- $v_2 = e^{v1} = 1.64$, $v'_2 = e^{v1} * v'_1 = 1.65$
- $v_3 = 2.14$, $v'_3 = v'_1 + v'_2 = 2.65$
- $v_4 = 0$, $v'_4 = 0$
- $v_5 = 0.5$, $v'_5 = v'_1 - v'_4 = 1$
- $f = 1.07$ $\frac{\partial f}{\partial x_1} = v'_3 * v_5 + v'_5 * v_3 = 3.46$



Automatic Differentiation: Forward Mode

- Forward mode
  - For each input variable, we compute both the value of each node as well as the value of the derivative of that node w.r.t that variable

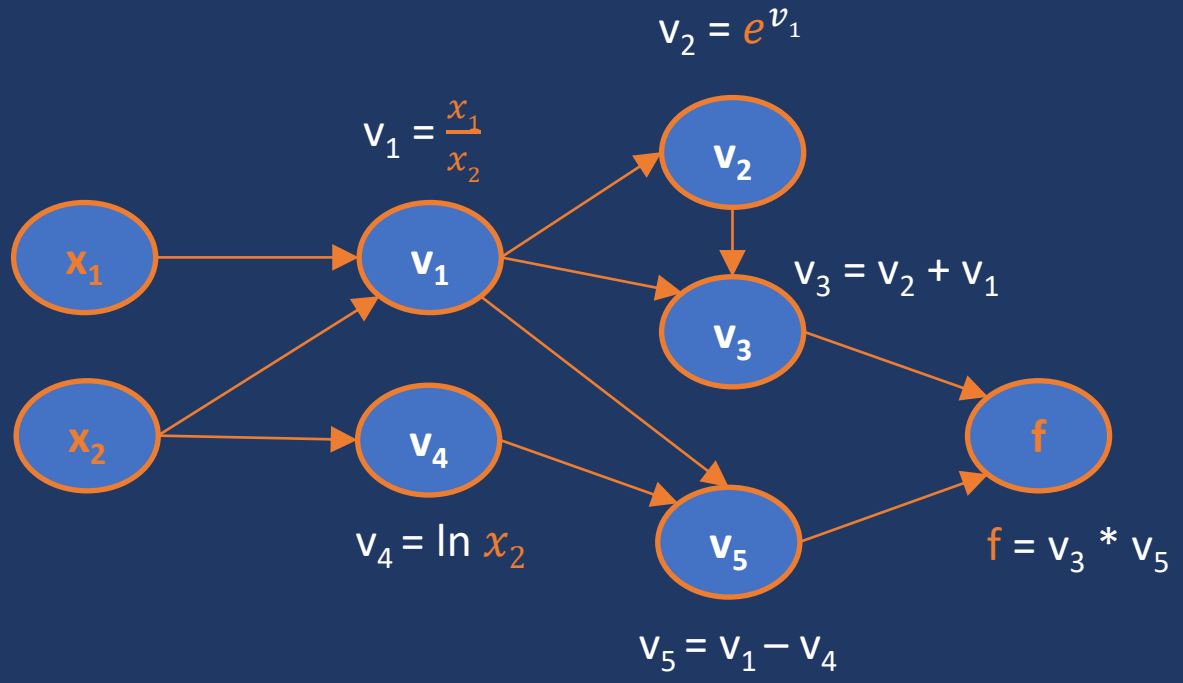
- One forward pass to compute $\frac{\partial f}{\partial x_1}$
- **Q:** Can we compute also $\frac{\partial g}{\partial x_1}$, for some other function $g(x_1, x_2)$ in the same pass?
  - Yes!
  - One joint computational graph for arbitrary number of functions over the same variables



Automatic Differentiation: Forward Mode

- Forward mode
  - For each input variable, we compute both the value of each node as well as the value of the derivative of the intermediate node w.r.t that variable

- One forward pass to compute $\frac{\partial f}{\partial x_1}$
- **Q:** Can we compute also $\frac{\partial f}{\partial x_2}$, in the same pass (while computing $\frac{\partial f}{\partial x_1}$)?
  - No*, we have to run two forward passes
  - Computation of partial derivatives of functions per different parameters is independent in forward mode



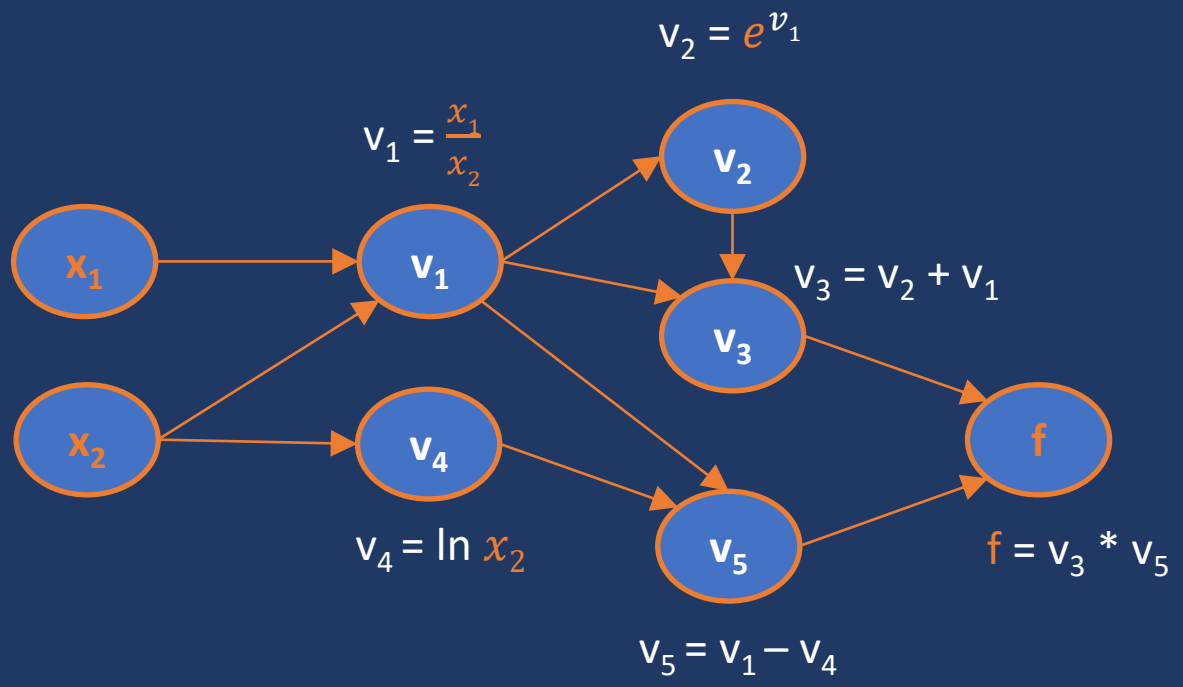
Automatic Differentiation

- Forward mode
  - Not suitable for **deep learning**!
  - Q: Why? Hint: how many paramaters do we have in DL models?
  - Forward mode good when
    - No. outputs >> no. of inputs/parameters
    - We need gradients of many different functions defined over the same small number of parameters
- **Reverse mode**
  - Start from end nodes of comp. graph and compute gradients backwards
  - Q: Familiar?

Automatic Differentiation: Forward Mode

- Reverse mode
  - Forward pass computes just the values
  - „Gradients” (actually adjoints) computed in a backward pass

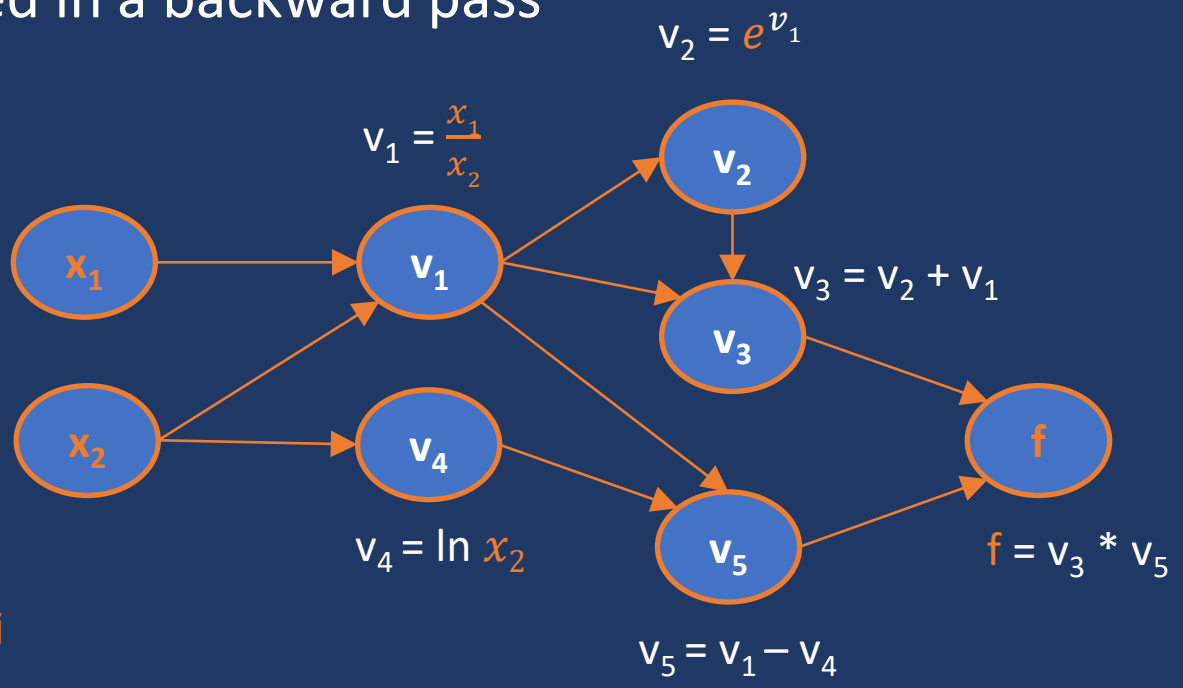
- $f(x_1, x_2) = \left(e^{\frac{x_1}{x_2}} + \frac{x_1}{x_2}\right) * \left(\frac{x_1}{x_2} - \ln x_2\right)$
- Start from: $(x_1, x_2) = (0.5, 1)$
- 1. Forward pass to compute the **values**
  - $v_1 = 0.5$,
  - $v_2 = e^{v1} = 1.64$,
  - $v_3 = 2.14$,
  - $v_4 = 0$,
  - $v_5 = 0.5$,
  - $f = 1.07$



Automatic Differentiation: Forward Mode

- Reverse mode
  - Forward pass computes just the values
  - „Gradients” (actually adjoints) computed in a backward pass
- $f(x_1, x_2) = \left(e^{\frac{x_1}{x_2}} + \frac{x_1}{x_2}\right) * \left(\frac{x_1}{x_2} - ln\ x_2\right)$
- Start from: $(x_1, x_2) = (0.5, 1)$
- 2. Backward pass to compute **adjoints**
  - Adjoint $\overline{v_i}$ of the node v_i is $\frac{\partial f}{\partial v_i}$
  - Adjoints of parent nodes v_i computed from adjoints of their children nodes v_j

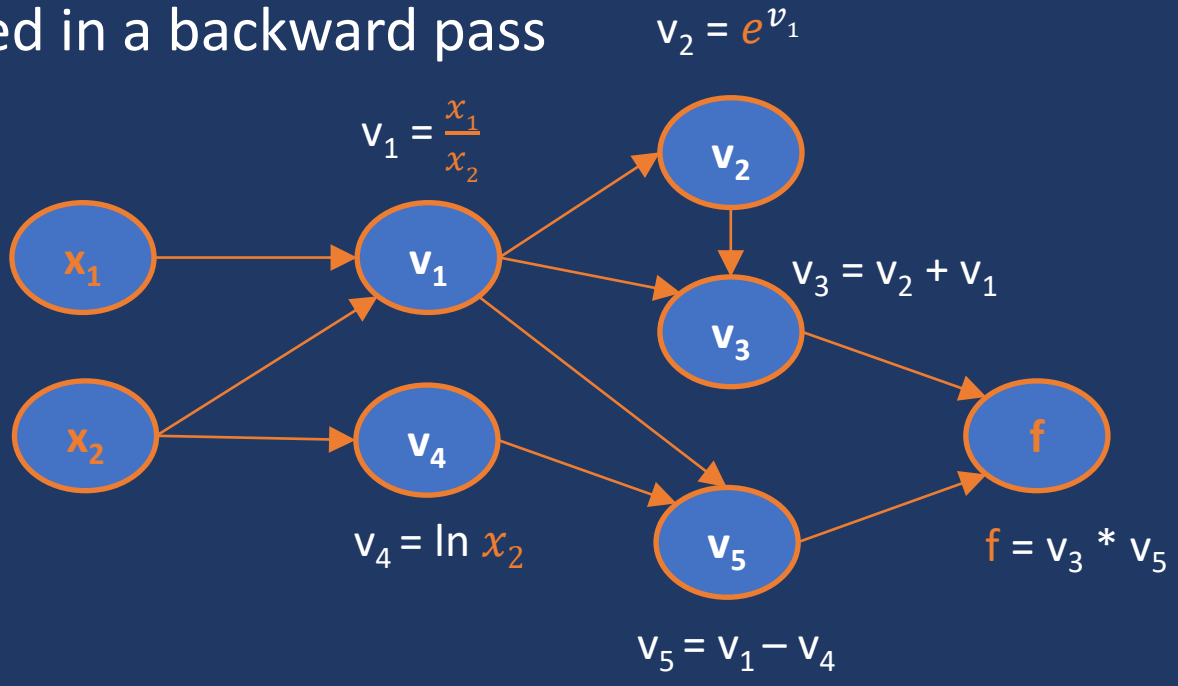
$$\overline{v_i} = \sum_{v_j\ child\ of\ v_i} \left(\overline{v_j} * \frac{\partial v_j}{\partial v_i}\right)$$



Automatic Differentiation: Forward Mode

- Reverse mode
  - Forward pass computes just the values
  - „Gradients” (actually adjoints) computed in a backward pass

- $f(x_1, x_2) = \left(e^{\frac{x_1}{x_2}} + \frac{x_1}{x_2}\right) * \left(\frac{x_1}{x_2} - \ln x_2\right)$
- $\bar{v_5} = \frac{\partial f}{\partial v_5} = v_3 = 2.14$
- $\bar{v_3} = \frac{\partial f}{\partial v_3} = v_5 = 0.5$
- $\bar{v_2} = \bar{v_3} * \frac{\partial v_3}{\partial v_2} = 0.5 * 1 = 0.5$
- $\bar{v_4} = \bar{v_5} * \frac{\partial v_5}{\partial v_4} = 2.14 * (-1) = -2.14$
- $\bar{v_1} = \bar{v_2} * \frac{\partial v_2}{\partial v_1} + \bar{v_3} * \frac{\partial v_3}{\partial v_1} + \bar{v_5} * \frac{\partial v_5}{\partial v_1}$
  $= 0.5 * 1.64 + 0.5 * 1 + 2.14 * 1 = 3.46$

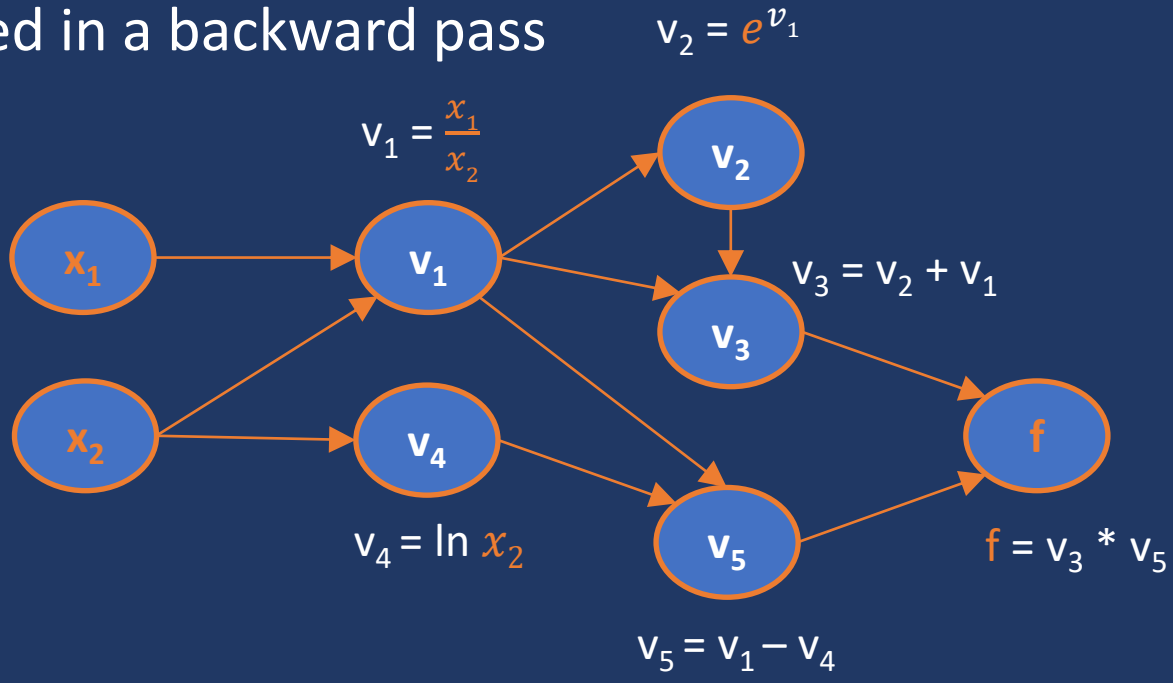


$$\bar{v_i} = \sum_{v_j \, child \, of \, v_i} \left(\bar{v_j} * \frac{\partial v_j}{\partial v_i}\right)$$

Automatic Differentiation: Forward Mode

- Reverse mode
  - Forward pass computes just the values
  - „Gradients” (actually adjoints) computed in a backward pass

- $f(x_1, x_2) = (e^{\frac{x_1}{x_2}} + \frac{x_1}{x_2}) * (\frac{x_1}{x_2} - ln\ x_2)$
- …
- $\bar{v_4} = \bar{v_5} * \frac{\partial v_5}{\partial v_4} = 0.5 * (-1) = -0.5$
- $\bar{v_1} = \bar{v_2} * \frac{\partial v_2}{\partial v_1} + \bar{v_3} * \frac{\partial v_3}{\partial v_1} + \bar{v_5} * \frac{\partial v_5}{\partial v_1}$
  $= 0.5 * 1.64 + 0.5 * 1 + 2.14 * 1 = 3.46$
- $\bar{x_1} = \bar{v_1} * \frac{\partial v_1}{\partial x_1} = \bar{v_1} * 1/x_2 = 3.46 * 1 = 3.46$
- $\bar{x_2} = \bar{v_1} * \frac{\partial v_1}{\partial x_2} + \bar{v_4} * \frac{\partial v_4}{\partial x_2} = …$

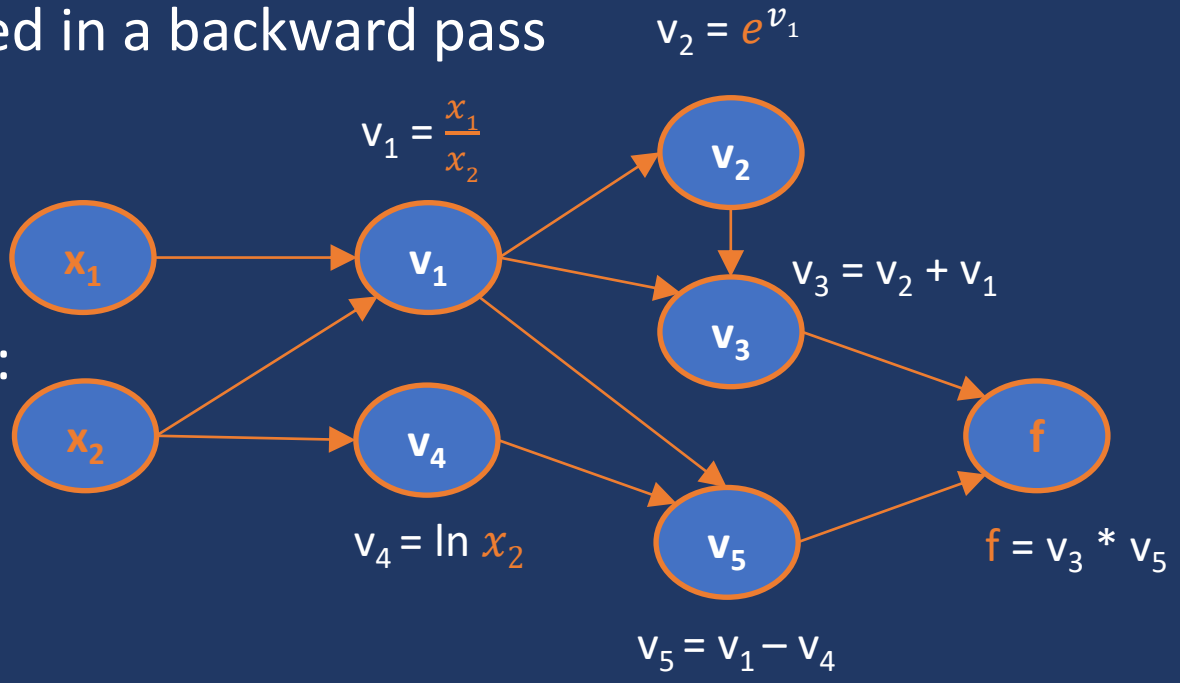


$$\bar{v_i} = \sum_{v_j\ child\ of\ v_i} (\bar{v_j} * \frac{\partial v_j}{\partial v_i})$$

Automatic Differentiation: Forward Mode

- Reverse mode
  - Forward pass computes just the values
  - „Gradients” (actually adjoints) computed in a backward pass

- Q: How many reverse/backward passes would we need if:
  - We have many variables/parameters: $x_1, x_2, \ldots, x_M$
    - **Just one**!
      - This is why it’s used in DL!
  - We have more than one target function: $f_1, f_2, \ldots, f_N$?

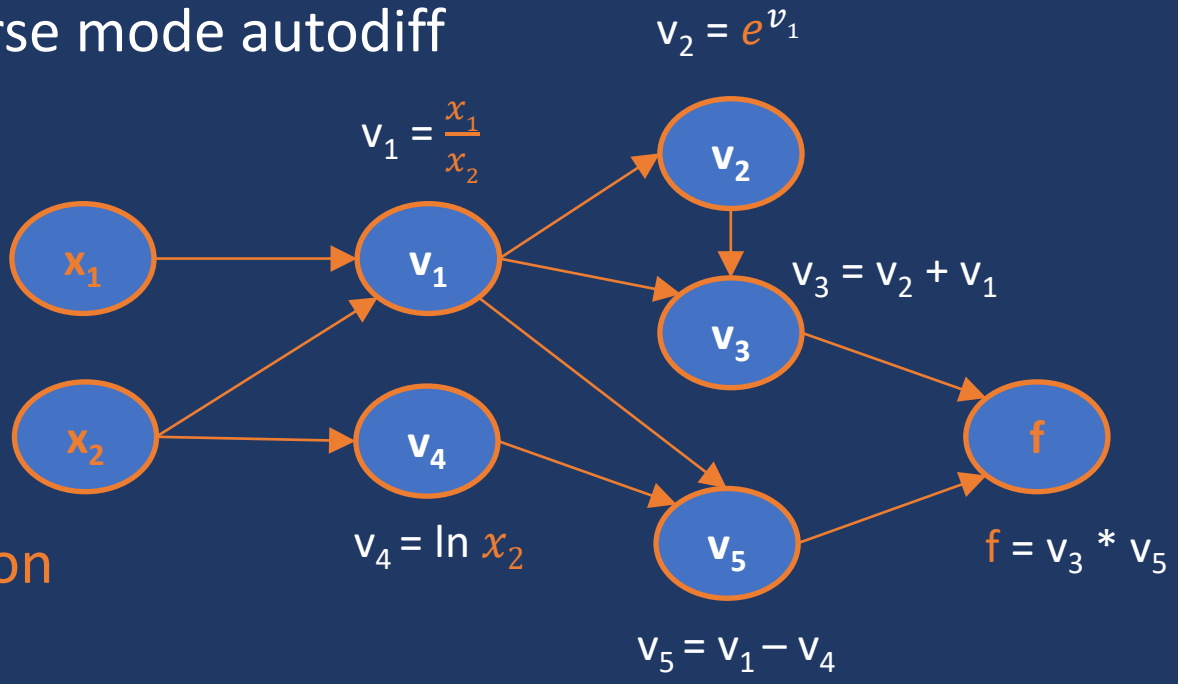


$$\overline{v_i} = \sum_{v_j \, child \, of \, v_i} \left(\overline{v_j} * \frac{\partial v_j}{\partial v_i} \right)$$

Reverse Mode Autodiff vs. Backpropagation?

- **Q:** How is reverse mode autodiff different from backpropagation?
  - Reverse mode autodiff is more general than backpropagation
  - Backpropagation a special case of reverse mode autodiff
    - Initially designed for FFNNs
    - One target function/loss (i.e., scalar)

- Q: Autodiff vs. Autograd?
  - Autograd is just the name of the popular autodiff Python implementation
  - Used also by PyTorch
    - `torch.autograd`



$$\overline{v_i} = \sum_{v_j \, child \, of \, v_i} \left(\overline{v_j} * \frac{\partial v_j}{\partial v_i} \right)$$

Automatic Differentiation in PyTorch

$f(x_1, x_2) = \left(e^{\frac{x_1}{x_2}} + \frac{x_1}{x_2}\right) * \left(\frac{x_1}{x_2} - \ln x_2\right)$

```
import torch

x1 = torch.tensor(0.5, requires_grad = True)
x2 = torch.tensor(1.0, requires_grad = True)

f = (torch.exp(x1/x2) + x1/x2) * (x1/x2 - torch.log(x2))

f.backward() # executes reverse mode autodiff

print(x1.grad)
print(x2.grad)
```

Fahrplan

- Gradient-Based Optimization & Backpropagation
- Automatic Differentiation
- Training in Batches
- Regularization

Stochastic Gradient Descent

- In Deep Learning, we never compute the exact gradient of the loss function on the whole training set D = $\{(\mathbf{x}_k, \mathbf{y}_k)\}_{k=1}^{N}$
  - Q: Why not?
  - Conceptual reason: gradient descent is guaranteed to lead to the closest local minimum (if η small enough)
  - Practical reason: we cannot fit all training examples into memory (GPU VRAM) at once*
- **Stochastic gradient descent** (SGD) – compute the loss, gradients, and update the parameters based on a single training instance
  - Repeat for all training instances
  - Order of instances random (hence the name stochastic)
  - Many parameter updates – slow training

Mini-Batch Gradient Descent

- **(Mini-)batch GD**: sweet spot between full GD and SGD
  - We train in the so-called mini-batches of B examples (e.g., B = 32)
  - Iteratively (mini-batch after mini-batch):
    1. Select B training examples from the training set D
    2. Compute the loss L_B and gradient $\nabla_{\boldsymbol{\theta}} L_B(\boldsymbol{\theta})$ based on B (using the reverse mode automatic differentiation)
    3. Update the parameters $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta \nabla_{\boldsymbol{\theta}} L_B(\boldsymbol{\theta}^{(t)})$
  - Batch-based GD – more resilient to local minima than GD and faster than SGD

- **Training epoch**: model updated on all mini-batches B from D,
  - Each training example part of exactly one mini-batch
  - It is common to train DL models for multiple epochs

Gradient Accumulation

- All instances of the batch B are „packed" into a single input tensor
  - Forward pass through the model simultaneous for instances in B
- In DL, we generally want to train on batches as large as possible
  - Limitation: VRAM of your GPU
  - Let B_P be the practical batch size, that is, the max. number of instances that fit into GPU memory at once
  - If B_P < desired batch size B, then we will resort to **gradient accumulation**
- **Gradient accumulation**
  - Accumulating (i.e., summing) gradients across $|B|/|B_P|$ batches of size $|B_P|$
  - Updating the parameters only at the end (learning rate needs to be adjusted*):

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \frac{\eta}{(|B|/|B_P|)} \sum_{B_P} \nabla_{\boldsymbol{\theta}} L_{BP}(\boldsymbol{\theta}^{(t)})$$

  - $|B|/|B_P|$ passes through the model (forward pass + reverse mode autodiff) for **one parameter update**

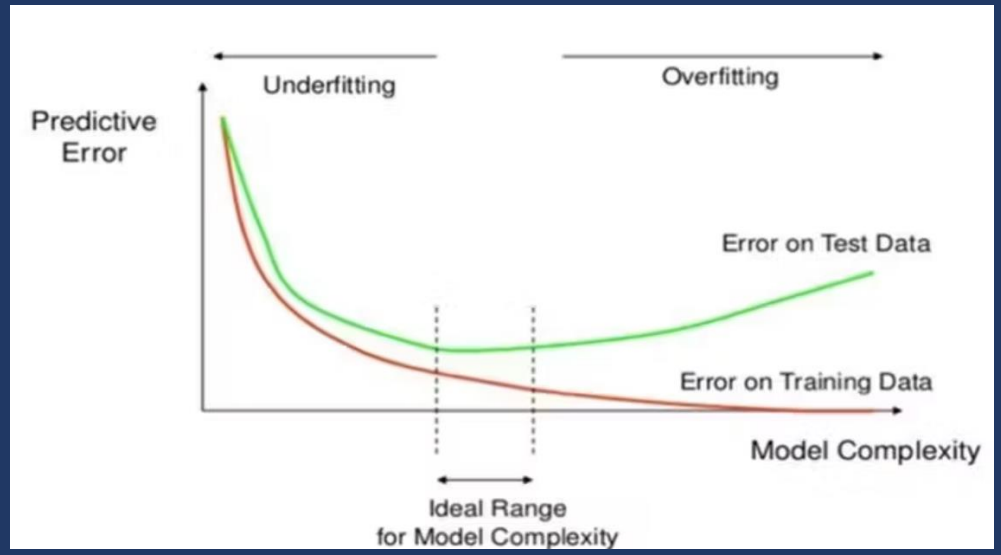
Fahrplan

- Gradient-Based Optimization & Backpropagation
- Automatic Differentiation
- Training in Batches
- Regularization

Regularization

- If complexity (sometimes in DL also called capacity) of the model $h(\mathbf{x}|\boldsymbol{\theta})$ is (much) larger than the data distribution we're modeling...
- ...model will likely overfit to training data and won't generalize well

- **Regularization** is an umbrella term for methods that try to prevent overfitting by reducing model complexity



Regularization

- **Regularization** is an umbrella term for methods that try to prevent overfitting by reducing model complexity
- Two most commonly used regularization techniques in Deep Learning:
  - **L2-Regularization** (called Ridge Regression in statistics)
  - **Dropout**

- **L2-Regularization**
  - Prevents parameters from getting large absolute values (which is what commonly happens when overfitting)
  - We minimize the objective: $J_R(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda * \|\boldsymbol{\theta}\|_2$
  - $\|\boldsymbol{\theta}\|_2$ – sum of Euclidean (L_2) norms of all parameter vectors and matrices

Dropout

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The journal of Machine Learning Research, 15(1), 1929-1958..

- **Regularization** by training multiple models (multiple different model instances and **ensembling** their predictions is effective
  - But this is very computationally prohibitive!
  - Especially if models have billions of parameters ☺

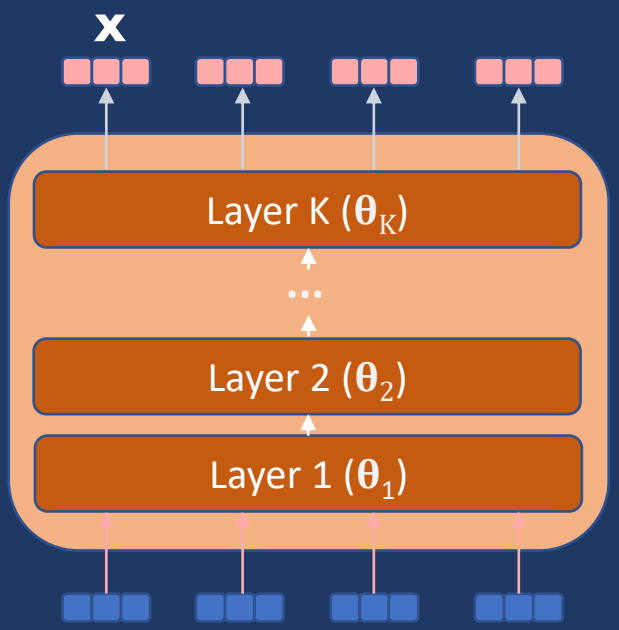
- **Dropout**: a regularization method that simulates training many (slightly) different models in a single training procedure
  - By means of randomly **dropping out** ”neurons” (zeroing out values in tensors)
  - Applied on per-layer basis, i.e., on the output of a layer

Dropout

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The journal of Machine Learning Research, 15(1), 1929-1958..

- Let **x** be any hidden representation, output of any layer of an arbitrary DL model
  - E.g., output of layer K
- Applying dropout on a layer means
  - To modify layer's output(s) **x** so that each element x_i becomes replaced with x'_i:

$x'_i = 0$ with dropout probability p or
$x'_i = x_i / (1-p)$ with the probability $(1-p)$



Deep Learning
Summer semester ‘24

3. Optimization & Training