Deep Learning

03 – Gradient-Based Training

Part 1: Backpropagation

Prof. Dr. Rainer Gemulla

Universität Mannheim

Version: 2024-1

Backpropagation

- **Backpropagation** is an algorithm to compute gradients
  - Origins in the 60s in control theory
  - Rediscovered many times
  - Used for neural networks since the 80s

- Given a compute graph, performs
  1. Forward pass to compute (all) output(s) (**forward propagation**)
  2. Backward pass to compute (all) gradient(s) (**backward propagation**)

- For us: compute graph typically represents
  - Output $\hat{y}$ of an FNN (given $\boldsymbol{x}$, $\boldsymbol{\theta}$)
  - Loss L of an FNN (given $(\boldsymbol{x}, y)$, $\boldsymbol{\theta}$)
  - Cost function J for an FNN (given $\{ (\boldsymbol{x}_i, y_i) \}$, $\boldsymbol{\theta}$)

- And we are interested in gradients (as we will see)
  - W.r.t. weights ($\nabla_{\boldsymbol{\theta}} J$): e.g., for gradient-based training
  - W.r.t. intermediate outputs ($\nabla_{\boldsymbol{z}} L$): e.g., for model debugging
  - W.r.t. inputs ($\nabla_{\boldsymbol{x}} L$ of $\nabla_{\boldsymbol{x}} \hat{y}$): e.g., for sensitivity analysis or adversarial training

Recap: Gradient

- For functions with multiple inputs, there are multiple **partial derivatives**; e.g.,

$$
\begin{aligned}
f &= x_1^2 + 5x_1x_2 \\
\frac{\partial}{\partial x_1} f &= 2x_1 + 5x_2 \\
\frac{\partial}{\partial x_2} f &= 5x_1
\end{aligned}
$$

- We can gather them all in a single vector, the **gradient** of f

$$\nabla_{\boldsymbol{x}^\top} f \stackrel{\text{def}}{=} \begin{pmatrix} \frac{\partial}{\partial x_1} J & \frac{\partial}{\partial x_2} f & \cdots & \frac{\partial}{\partial x_n} f \end{pmatrix}$$

- For the example above, we obtain

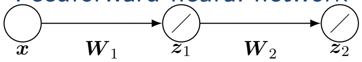
$$\nabla_{\boldsymbol{x}^\top} f = \begin{pmatrix} 2x_1 + 5x_2 & 5x_1 \end{pmatrix} \qquad \nabla_{\boldsymbol{x}} f = \begin{pmatrix} 2x_1 + 5x_2 \\ 5x_1 \end{pmatrix}$$

Numerator layout (row) Denominator layout (column)

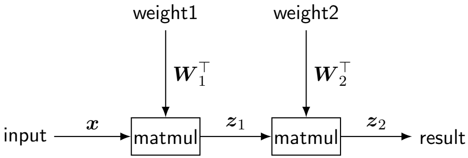
Compute graphs

- Backpropagation generally operates on a **compute graph**
- Directed, acyclic graph that models a computation (as a *data flow program*)
- Vertices correspond to operations
- Edges correspond to data passed between operations (typically tensor-valued)
- Multiple **sources** (no incoming edge): inputs, weights, ...
- One **sink** (no outgoing edge): result

Feedforward neural network

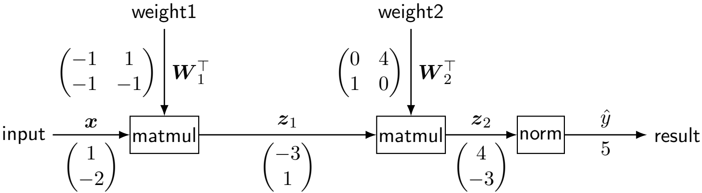


Corresponding compute graph



Forward propagation (example)

- Compute graph for example output $\hat{y}$



- **Forward propagation: inputs $\to$ result**

- Edges transport values

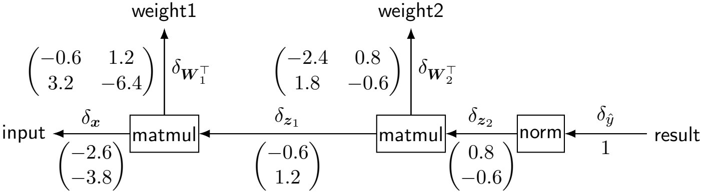
- For example:
  1. Provide inputs $\boldsymbol{x}$, $\boldsymbol{W}_1^\top$, $\boldsymbol{W}_2^\top$
  2. Evaluate first matmul: $\boldsymbol{z}_1 = \boldsymbol{W}_1^\top \boldsymbol{x}$
  3. Evaluate second matmul: $\boldsymbol{z}_2 = \boldsymbol{W}_2^\top \boldsymbol{z}_1$
  4. Evaluate norm: $\hat{y} = \|\boldsymbol{z}_2\|$

Forward propagation

- Operators are evaluated in **topological order** ("forwards")
  - Whenever an operator is evaluated, all its inputs must be available
  - Computation is **local**: only input values are required (the remainder of the compute graph does not matter)
- Inputs and/or outputs are generally tensor-valued
  - E.g., $\mathrm{matmul}(\boldsymbol{A}, \boldsymbol{B}) = \boldsymbol{A}\boldsymbol{B}$ takes two 2D tensors and produces a 2D tensor
  - Note: our visual representation of compute graph does not indicate which input is $\boldsymbol{A}$ and which is $\boldsymbol{B}$, but the actual compute graph does (and must do so)
- Intermediate results may need to be kept
  - To evaluate subsequent operators
  - To enable gradient computation with backpropagation
- Parallel processing is possible
  - Each operator can be evaluated as soon as all its inputs available
  - E.g., transformer encoders can operate on all inputs in parallel
  - E.g., RNN encoders must process inputs sequentially

Backward propagation (example)

- Backward graph for example output $\hat{y}$



- **Backward propagation: result $\rightarrow$ gradients**
- Edges transport gradients
  - Consider edge e and define

    $$\delta_e \stackrel{\text{def}}{=} \textbf{gradient of result w.r.t. values on edge } e$$

    evaluated at the provided inputs
- For example:
  1. Compute all values of forward pass (not shown above)
  2. $\delta_{\hat{y}} = \nabla_{\hat{y}}\text{result} = \nabla_{\hat{y}}\hat{y} = 1$
  3. $\delta_{\boldsymbol{z}_2}$ (discussed later)
  4. $\delta_{\boldsymbol{W}_2^\top}$ and $\delta_{\boldsymbol{z}_1}$ (discussed later)
  5. $\delta_{\boldsymbol{W}_1^\top}$ and $\delta_{\boldsymbol{x}}$ (discussed later)

Backward propagation

- $\delta_e \stackrel{\text{def}}{=}$ **gradient of result w.r.t. values on edge e**
- Key insight of backpropagation
  - Gradients δ_e can be computed incrementally (akin to forward pass, but in reverse order)
- Operators are evaluated in **reverse topological order** (“backwards”)
  - When operator evaluated, its output gradient(s) must be available
  - Computation is **local**: only input values and output gradient(s) are required (the remainder of the compute graph does not matter)
  - Recall: intermediate outputs of forward pass required $\rightarrow$ memory consumption (or recompute)
- Gradients are generally tensor-valued
  - Convention: same shape as values in forward pass
- Intermediate results may need to be kept
  - To evaluate gradient for prior operators
  - To debug/analyze models
- Parallel processing is possible (as before)

Gradient (single univariate function)

input $\xrightarrow{u} \boxed{f} \xrightarrow{y}$ result

- Output: $y = f(u)$
- Gradient $\delta_y \stackrel{\text{def}}{=} \nabla_y y = 1$
- Gradient $\delta_u \stackrel{\text{def}}{=} \nabla_u y = \frac{\partial}{\partial u} f(u) = f'(u)$
- Example
  - $y = \sigma(u) = \sigma(0)$ (logistic function)
  - $\delta_u = \sigma'(u) = \sigma(u)(1 - \sigma(u)) = \sigma(0)(1 - \sigma(0))$

input $\xrightarrow[0]{u} \boxed{\sigma} \xrightarrow[0.5]{y}$ result

Forward pass

input $\xleftarrow[0.25]{\delta_u} \boxed{\sigma} \xleftarrow[1]{\delta_y}$ result

Backward pass

Gradient (composition of two univariate functions)

- Let's add another operator g in front

$$\text{input} \xrightarrow{v} \boxed{g} \xrightarrow{u} \boxed{f} \xrightarrow{y} \text{result}$$

- Output: $y = f(u) = f(g(v))$
  - **Function composition**
- Gradient: $\delta_u \stackrel{\text{def}}{=} \nabla_u y = \frac{\partial}{\partial u} f(u) = f'(u) = f'(u)$
  - Same computation as before (but u now output of g)
  - Need to retain u in forward pass to compute $f'(u)$
- Gradient

$$\delta_v \stackrel{\text{def}}{=} \nabla_v y = \underbrace{\frac{\partial}{\partial v} f(g(v)) = g'(v) f'(g(v))}_{\text{chain rule}} = g'(v) f'(u)$$

$$= g'(v)\delta_u$$

  - **Observe: that's a local computation at g**
  - Need: $\delta_u \rightarrow$ passed backwards from subsequent operators
  - Need: $v \rightarrow$ computed in forward pass
  - Need: $g' \rightarrow$ determined by g

Example

Forward pass

$$\text{input} \xrightarrow[1]{v} \boxed{\log_2} \xrightarrow[0]{u} \boxed{\sigma} \xrightarrow[0.5]{y} \text{result}$$

Backward pass

$$\text{input} \xleftarrow[0.36]{\delta_v} \boxed{\log_2} \xleftarrow[0.25]{\delta_u} \boxed{\sigma} \xleftarrow[1]{\delta_y} \text{result}$$

- $\delta_y = \nabla_y y = 1$
- $\delta_u = \nabla_u y = \sigma'(u)\delta_y = \sigma(u)(1 - \sigma(u))\delta_y = 0.25 \cdot 1 = 0.25$
- $\delta_v = \nabla_v y = \log_2'(v)\delta_u = \frac{1}{v\log(2)}\delta_u \approx 1.44 \cdot 0.25 = 0.36$

Gradient (composition of univariate functions)

- This generalizes; e.g., consider n operators

$$\text{input} \xrightarrow{z_0 = x} \boxed{f_1} \xrightarrow{z_1} \boxed{f_2} \xrightarrow{z_2} \cdots \xrightarrow{z_{n-1}} \boxed{f_n} \xrightarrow{z_n = y} \text{result}$$

- We have

$$y = f_n(f_{n-1}(\cdots(f_1(x))\cdots))$$

- At each operator f_i, the required gradient can be computed as follows:

$$\begin{aligned} \delta_{z_{i-1}} \overset{\text{def}}{=} \nabla_{z_{i-1}} y = & \underbrace{\frac{\partial y}{\partial z_{i-1}} = \frac{\partial y}{\partial z_i} \frac{\partial z_i}{\partial z_{i-1}}}_{\textbf{chain rule}} \\ = & \underbrace{f_i'(z_{i-1})}_{\text{local derivative}} \cdot \underbrace{\delta_{z_i}}_{\text{output derivative}} \end{aligned}$$

Overall gradient

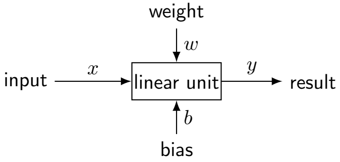
- Let's derive an expression for the gradients individually

$$
\begin{aligned}
\delta_{z_n} &= 1 \\
\delta_{z_{n-1}} &= f'_n(z_{n-1})\delta_{z_n} &&= & f'_n(z_{n-1}) \\
\delta_{z_{n-2}} &= f'_{n-1}(z_{n-2})\delta_{z_{n-1}} &&= & f'_{n-1}(z_{n-2})f'_n(z_{n-1}) \\
\delta_{z_{n-3}} &= f'_{n-2}(z_{n-3})\delta_{z_{n-2}} &&= & f'_{n-2}(z_{n-3})f'_{n-1}(z_{n-2})f'_n(z_{n-1}) \\
&\vdots
\end{aligned}
$$

- **Gradient is product of local gradients along the path from the result to the resp. edge**
- Backpropagation avoids repeated computations by
  1. Proceeding backwards
  2. Using the chain rule

Gradient (multiple inputs)

- Operators often have multiple inputs; e.g., a simple linear unit



- In the forward pass, the operator computes

$$y = f(x, w, b) = wx + b$$

- In the backward pass, we compute gradients of result w.r.t. each edge as before (using the chain rule)

$$\begin{aligned}
\delta_y &= 1 \\
\delta_x &= \nabla_x y = \nabla_x f(w, x, b) \cdot \delta_y = w \cdot 1 \\
\delta_w &= \nabla_w y = \nabla_w f(w, x, b) \cdot \delta_y = x \cdot 1 \\
\delta_b &= \nabla_b y = \nabla_b f(w, x, b) \cdot \delta_y = 1 \cdot 1
\end{aligned}$$

- **Consider each input separately and reuse incoming δ-value**

Gradient (multiple outputs)

- Operators may have multiple outputs; e.g., consider
  - E.g., operator $f(x)$ may output n values, say, $z_1 = f_1(x)$, ..., $z_n = f_n(x)$
  - During backpropagation, we obtain δ_{z_1}, ..., δ_{z_n}
  - We are interested in

$$\delta_x = \nabla_x y = \underbrace{\frac{\partial y}{\partial x} = \sum_{k=1}^{n} \frac{\partial y}{\partial z_k} \frac{\partial z_k}{\partial x}}_{\text{multivariate chain rule}}$$

$$= \sum_{k=1}^{n} f_k'(x) \delta_{z_k}$$

  - **Consider each output independently and sum up**

Gradient (multiple uses)

- Sometimes an operator's output is "used" multiple times
  - E.g., the output of an operator $g(x)$ is used n times
  - That's equivalent to a single operator f with n identical outputs (i.e., $z_k = f_k(x) = g(x)$), each being used once
  - Using the results from the previous slide with f defined in this way:

$$\begin{aligned}\delta_x = \nabla_x y = \sum_{k=1}^{n} f'_k(x)\delta_{z_k} &= \sum_{k=1}^{n} g'(x)\delta_{z_k} \\ = g'(x)\sum_{k=1}^{n}\delta_k\end{aligned}$$

  - **Sum up all incoming δ-values and proceed as before**

Gradient computation in general

- Consider an operator $\boldsymbol{f} : \mathbb{R}^I \to \mathbb{R}^O$
- Forward pass: $\boldsymbol{v}^{\text{out}} = \boldsymbol{f}(\boldsymbol{v}^{\text{in}})$ with $\boldsymbol{v}^{\text{in}} \in \mathbb{R}^I$ and $\boldsymbol{v}^{\text{out}} \in \mathbb{R}^O$

$$\xrightarrow{\boldsymbol{v}^{\text{in}}} \boldsymbol{f} \xrightarrow{\boldsymbol{v}^{\text{out}}}$$

- Backward pass: $\boldsymbol{\delta}^{\text{in}} = \boldsymbol{J}_{\boldsymbol{f}}(\boldsymbol{v}^{\text{in}})^\top \boldsymbol{\delta}^{\text{out}}$ with $\boldsymbol{\delta}^{\text{out}} \in \mathbb{R}^O$ and $\boldsymbol{\delta}^{\text{in}} \in \mathbb{R}^I$

$$\xleftarrow{\boldsymbol{\delta}^{\text{in}}} \boldsymbol{J}_{\boldsymbol{f}} \xleftarrow{\boldsymbol{\delta}^{\text{out}}}$$

  where we use the **Jacobian** $\boldsymbol{J}_{\boldsymbol{f}} \in \mathbb{R}^{O \times I}$ given by

$$\boldsymbol{J}_{\boldsymbol{f}} = \nabla_{\boldsymbol{v}_{\text{in}}^\top} \boldsymbol{f} = \begin{pmatrix} \nabla_{\boldsymbol{v}_{\text{in}}^\top} v_1^{\text{out}} \\ \vdots \\ \nabla_{\boldsymbol{v}_{\text{in}}^\top} v_O^{\text{out}} \end{pmatrix} = \begin{pmatrix} \frac{\partial}{\partial v_1^{\text{in}}} v_1^{\text{out}} & \cdots & \frac{\partial}{\partial v_I^{\text{in}}} v_1^{\text{out}} \\ \vdots & \ddots & \vdots \\ \frac{\partial}{\partial v_1^{\text{in}}} v_O^{\text{out}} & \cdots & \frac{\partial}{\partial v_I^{\text{in}}} v_O^{\text{out}} \end{pmatrix}$$

- Intuitively, $f'(v^{\text{in}})\delta^{\text{out}}$ now becomes $\boldsymbol{J}_{\boldsymbol{f}}(\boldsymbol{v}^{\text{in}})^\top \boldsymbol{\delta}^{\text{out}}$
  - Can be derived by “rewriting” the discussions on multiple inputs/outputs from the previous slides into matrix form
- More in exercises and tutorials