Mathematics of Data: From Theory to Computation

Prof. Volkan Cevher
volkan.cevher@epfl.ch

A lightning tour through the optimization of deep neural networks

Laboratory for Information and Inference Systems (LIONS)
École Polytechnique Fédérale de Lausanne (EPFL)

EE-556 (Fall 2024)

License Information for Mathematics of Data Slides

- This work is released under a Creative Commons License with the following terms:
- **Attribution**
  - The licensor permits others to copy, distribute, display, and perform the work. In return, licensees must give the original authors credit.
- **Non-Commercial**
  - The licensor permits others to copy, distribute, display, and perform the work. In return, licensees may not use the work for commercial purposes – unless they get the licensor's permission.
- **Share Alike**
  - The licensor permits others to distribute derivative works only under a license identical to the one that governs the licensor's work.
- Full Text of the License

- This recitation

  1. Brief intro into tensors
  2. Backpropagation
  3. Automatic Differentiation & PyTorch
  4. Deep Learning Building Blocks

Outline

Recall: Definition and representation of deep neural networks

Training deep networks

Computational infrastructure

Deep Learning Toolkit

Tensors

- Tensors provide a natural and concise mathematical represention of **data** ($\mathbf{a}$) and **parameters** ($\mathbf{X}_l, \mu_l$ where l indicates the layers).
- Tensors are **multidimensional arrays** and are a generalization of:
  1. scalars - **tensors** with no *indices*; i.e., zeroth-rank tensor.
  2. vectors - **tensors** with exactly one *index*; i.e., first-rank tensor.
  3. matrices - **tensors** with exactly two *indices*; i.e., second-rank tensor.
  4. etc.

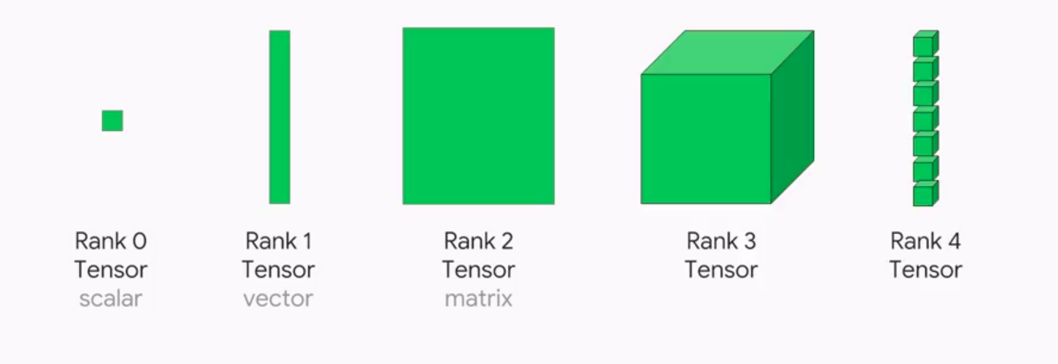


Figure: From [1]

Outline

Recall: Definition and representation of deep neural networks

Training deep networks

Computational infrastructure

Deep Learning Toolkit

Recall: Basic Neural Network

1-hidden-layer neural network with m neurons (fully-connected architecture):

- Parameters: $\mathbf{X}_1 \in \mathbb{R}^{m \times d}$, $\mathbf{X}_2 \in \mathbb{R}^{c \times m}$ (weights), $\mu_1 \in \mathbb{R}^m, \mu_2 \in \mathbb{R}^c$ (biases)
- Activation function: $\sigma : \mathbb{R} \to \mathbb{R}$

$$h_{\mathbf{x}}(\mathbf{a}) := \begin{bmatrix} \ \mathbf{X}_2 \ \end{bmatrix} \underbrace{\overset{\text{activation}}{\overset{\downarrow}{\sigma}} \left(\overset{\text{weight}}{\overset{\downarrow}{\begin{bmatrix} \ \mathbf{X}_1 \ \end{bmatrix}}} \overset{\text{input}}{\overset{\downarrow}{\begin{bmatrix} \mathbf{a} \end{bmatrix}}} + \overset{\text{bias}}{\overset{\downarrow}{\begin{bmatrix} \mu_1 \end{bmatrix}}} \right)}_{\text{hidden layer = learned features}} + \overset{\text{bias}}{\overset{\downarrow}{\begin{bmatrix} \mu_2 \end{bmatrix}}}, \qquad \mathbf{x} := [\mathbf{X}_1, \mathbf{X}_2, \mu_1, \mu_2]$$

recursively repeat activation + affine transformation to obtain "deeper" networks.

Minimization of the loss function

In order to use first order methods, we need to derive the gradient

$$\nabla_{\boldsymbol{x}} R_n(\boldsymbol{x}) := \frac{1}{n}\sum_{i=1}^{n} \nabla_{\boldsymbol{x}} L(h(\boldsymbol{a}_i;\boldsymbol{x}), b_i) := \frac{1}{n}\sum_{i=1}^{n} \nabla_{\boldsymbol{x}} L_i(\boldsymbol{x}) \tag{1}$$

where $\boldsymbol{x} = [\mathbf{X}_1, \mu_1, \ldots, \mathbf{X}_k, \mu_k]$ are the weights and biases of the network. For convenience we sometimes also write $h_{\boldsymbol{x}}(\boldsymbol{a})$ instead of $h(\boldsymbol{a}; x)$.

Minimization of the loss function

In order to use first order methods, we need to derive the gradient

$$\nabla_{\boldsymbol{x}} R_n(\boldsymbol{x}) := \frac{1}{n}\sum_{i=1}^{n} \nabla_{\boldsymbol{x}} L(h(\boldsymbol{a}_i;\boldsymbol{x}), b_i) := \frac{1}{n}\sum_{i=1}^{n} \nabla_{\boldsymbol{x}} L_i(\boldsymbol{x}) \tag{1}$$

where $\boldsymbol{x} = [\mathbf{X}_1, \mu_1, \ldots, \mathbf{X}_k, \mu_k]$ are the weights and biases of the network. For convenience we sometimes also write $h_{\boldsymbol{x}}(\boldsymbol{a})$ instead of $h(\boldsymbol{a}; x)$.

Example (Naive computation of the gradient)

Let $h(\boldsymbol{a}; \boldsymbol{X}_1, \boldsymbol{X}_2) = \boldsymbol{X}_2^T \sigma(\boldsymbol{X}_1 \boldsymbol{a})$, and $L_i(\boldsymbol{X}_1, \boldsymbol{X}_2) = (b_i - \boldsymbol{X}_2^T \sigma(\boldsymbol{X}_1 \boldsymbol{a}_i))^2$ be the loss on a sample, then

$$\frac{\partial L_i}{\partial \boldsymbol{X}_2} = -2(b_i - \boldsymbol{X}_2^T \sigma(\boldsymbol{X}_1 \boldsymbol{a}_i))\sigma(\boldsymbol{X}_1 \boldsymbol{a}_i) \tag{2}$$

$$\frac{\partial L_i}{\partial \boldsymbol{X}_1} = -2(b_i - \boldsymbol{X}_2^T \sigma(\boldsymbol{X}_1 \boldsymbol{a}_i))\boldsymbol{X}_2 \odot \sigma'(\boldsymbol{X}_1 \boldsymbol{a}_i)\boldsymbol{a}_i^T \tag{3}$$

where $\odot$ denotes element-wise product of vectors.

Many similar terms in both derivatives $\Rightarrow$ Inefficient to compute them independently

Forward pass

Forward pass scheme
Input: $\boldsymbol{a}^{(\mathbf{0})} = \boldsymbol{a}$, $\mathbf{X}^{(l)}$ and $\mu^{(l)}$ for $l = 1, \ldots, k$.
1. **For** $l = 1, \ldots, k$ Compute $\boldsymbol{u}^{(l)} = \mathbf{X}^{(l)}\boldsymbol{a}^{(l-1)} + \mu^{(l)}$ Compute $\boldsymbol{a}^{(l)} = \sigma(\boldsymbol{u}^{(l)})$

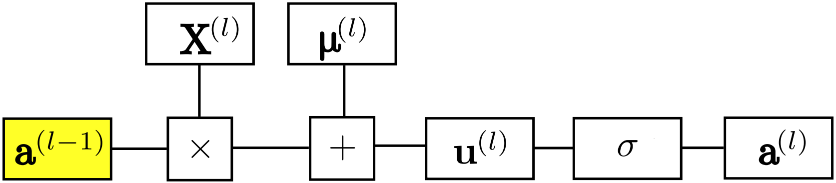


Figure: Computation of $\mathbf{u}^{(l)}$ and $\mathbf{a}^{(l)}$ starting from $\mathbf{a}^{(l-1)}$

Forward pass

Forward pass scheme
Input: $\boldsymbol{a}^{(\mathbf{0})} = \boldsymbol{a}$, $\mathbf{X}^{(l)}$ and $\mu^{(l)}$ for $l = 1, \ldots, k$.
1. **For** $l = 1, \ldots, k$ Compute $\boldsymbol{u}^{(l)} = \mathbf{X}^{(l)}\boldsymbol{a}^{(l-1)} + \mu^{(l)}$ Compute $\boldsymbol{a}^{(l)} = \sigma(\boldsymbol{u}^{(l)})$

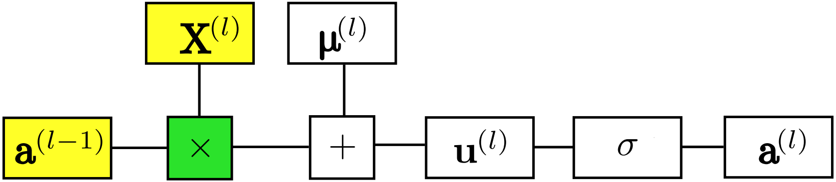


Figure: Computation of $\mathbf{u}^{(l)}$ and $\mathbf{a}^{(l)}$ starting from $\mathbf{a}^{(l-1)}$

Forward pass

Forward pass scheme
Input: $\boldsymbol{a}^{(\mathbf{0})} = \boldsymbol{a}$, $\mathbf{X}^{(l)}$ and $\mu^{(l)}$ for $l = 1, \ldots, k$.
1. For $l = 1, \ldots, k$ Compute $\boldsymbol{u}^{(l)} = \mathbf{X}^{(l)} \boldsymbol{a}^{(l-1)} + \mu^{(l)}$ Compute $\boldsymbol{a}^{(l)} = \sigma(\boldsymbol{u}^{(l)})$

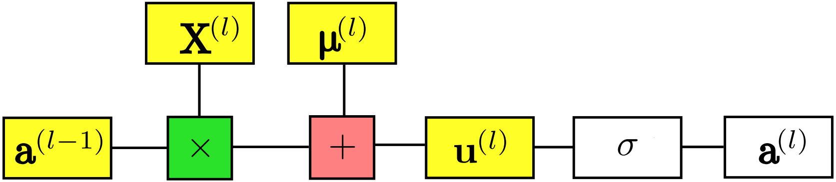


Figure: Computation of $\mathbf{u}^{(l)}$ and $\mathbf{a}^{(l)}$ starting from $\mathbf{a}^{(l-1)}$

Forward pass

Forward pass scheme
Input: $\boldsymbol{a}^{(\mathbf{0})} = \boldsymbol{a}$, $\mathbf{X}^{(l)}$ and $\mu^{(l)}$ for $l = 1, \ldots, k$.
1. For $l = 1, \ldots, k$ Compute $\boldsymbol{u}^{(l)} = \mathbf{X}^{(l)}\boldsymbol{a}^{(l-1)} + \mu^{(l)}$ Compute $\boldsymbol{a}^{(l)} = \sigma(\boldsymbol{u}^{(l)})$

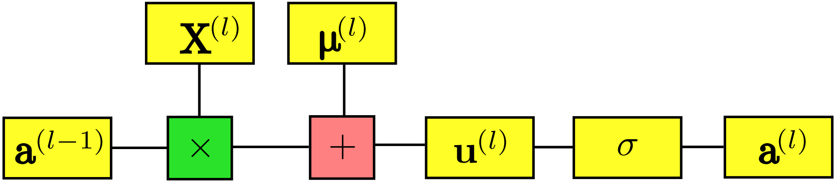


Figure: Computation of $\mathbf{u}^{(l)}$ and $\mathbf{a}^{(l)}$ starting from $\mathbf{a}^{(l-1)}$

Backward pass

Suppose $\frac{\partial L}{\partial \boldsymbol{a}^{(l)}}$ is given, as well as all pre-activation and hidden layer values.

- **Goal**: obtain $\frac{\partial L}{\partial \mathbf{X}^{(l)}}$, $\frac{\partial L}{\partial \mu^{(l)}}$ and $\frac{\partial L}{\partial \boldsymbol{a}^{(l-1)}}$.

Backward pass

Suppose $\frac{\partial L}{\partial \boldsymbol{a}^{(l)}}$ is given, as well as all pre-activation and hidden layer values.

- **Goal**: obtain $\frac{\partial L}{\partial \mathbf{X}^{(l)}}$, $\frac{\partial L}{\partial \mu^{(l)}}$ and $\frac{\partial L}{\partial \boldsymbol{a}^{(l-1)}}$.

1.

$$\boldsymbol{u}^{(l)} = \mathbf{X}^{(l)} \boldsymbol{a}^{(l-1)} + \mu^{(l)} \Rightarrow \begin{cases} \frac{\partial L}{\partial \mathbf{X}^{(l)}} & = \frac{\partial L}{\partial \boldsymbol{u}^{(l)}} (\boldsymbol{a}^{(l-1)})^T \\ \frac{\partial L}{\partial \mu^{(l)}} & = \frac{\partial L}{\partial \boldsymbol{u}^{(l)}} \end{cases} \qquad \text{(chain rule)}$$

Backward pass

Suppose $\frac{\partial L}{\partial \boldsymbol{a}^{(l)}}$ is given, as well as all pre-activation and hidden layer values.

- **Goal**: obtain $\frac{\partial L}{\partial \mathbf{X}^{(l)}}$, $\frac{\partial L}{\partial \mu^{(l)}}$ and $\frac{\partial L}{\partial \boldsymbol{a}^{(l-1)}}$.

1.
$$\boldsymbol{u}^{(l)} = \mathbf{X}^{(l)} \boldsymbol{a}^{(l-1)} + \mu^{(l)} \Rightarrow \begin{cases} \frac{\partial L}{\partial \mathbf{X}^{(l)}} & = \frac{\partial L}{\partial \boldsymbol{u}^{(l)}} (\boldsymbol{a}^{(l-1)})^T \\ \frac{\partial L}{\partial \mu^{(l)}} & = \frac{\partial L}{\partial \boldsymbol{u}^{(l)}} \end{cases} \qquad \text{(chain rule)}$$

2.
$$\boldsymbol{a}^{(l)} = \sigma(\boldsymbol{u}^{(l)}) \Rightarrow \frac{\partial L}{\partial \boldsymbol{u}^{(l)}} = \frac{\partial L}{\partial \boldsymbol{a}^{(l)}} \odot \sigma'(\boldsymbol{u}^{(l)}) \qquad \text{(chain rule)}$$

Where $\odot$ is the Hadamard product (element-wise product).

Backward pass

Suppose $\frac{\partial L}{\partial \boldsymbol{a}^{(l)}}$ is given, as well as all pre-activation and hidden layer values.

- **Goal**: obtain $\frac{\partial L}{\partial \mathbf{X}^{(l)}}$, $\frac{\partial L}{\partial \mu^{(l)}}$ and $\frac{\partial L}{\partial \boldsymbol{a}^{(l-1)}}$.

1.
$$\boldsymbol{u}^{(l)} = \mathbf{X}^{(l)}\boldsymbol{a}^{(l-1)} + \mu^{(l)} \Rightarrow \begin{cases} \frac{\partial L}{\partial \mathbf{X}^{(l)}} & = \frac{\partial L}{\partial \boldsymbol{u}^{(l)}}(\boldsymbol{a}^{(l-1)})^T \\ \frac{\partial L}{\partial \mu^{(l)}} & = \frac{\partial L}{\partial \boldsymbol{u}^{(l)}} \end{cases} \qquad \text{(chain rule)}$$

2.
$$\boldsymbol{a}^{(l)} = \sigma(\boldsymbol{u}^{(l)}) \Rightarrow \frac{\partial L}{\partial \boldsymbol{u}^{(l)}} = \frac{\partial L}{\partial \boldsymbol{a}^{(l)}} \odot \sigma'(\boldsymbol{u}^{(l)}) \qquad \text{(chain rule)}$$

   Where $\odot$ is the Hadamard product (element-wise product).

3. Finally we have
$$\boldsymbol{u}^{(l)} = \mathbf{X}^{(l)}\boldsymbol{a}^{(l-1)} + \mu^{(l)} \Rightarrow \frac{\partial L}{\partial \boldsymbol{a}^{(l-1)}} = (\mathbf{X}^{(l)})^T \frac{\partial L}{\partial \boldsymbol{u}^{(l)}} \qquad \text{(chain rule)}$$

Backward pass

Backward pass scheme
Input: Gradient of the loss w.r.t. the last layer values $\partial L / \partial \boldsymbol{a}^{(k)}$
1. **For** $l = k, \ldots, 1$ Compute $\frac{\partial L}{\partial \boldsymbol{u}^{(l)}} = \frac{\partial L}{\partial \boldsymbol{a}^{(l)}} \odot \sigma'(\boldsymbol{u}^{(l)})$ Compute $\frac{\partial L}{\partial \mathbf{X}^{(l)}} = \frac{\partial L}{\partial \boldsymbol{u}^{(l)}} (\boldsymbol{a}^{(l-1)})^T, \frac{\partial L}{\partial \mu^{(l)}} = \frac{\partial L}{\partial \boldsymbol{u}^{(l)}}$ Compute $\frac{\partial L}{\partial \boldsymbol{a}^{(l-1)}} = (\mathbf{X}^{(l)})^T \frac{\partial L}{\partial \boldsymbol{u}^{(l)}}$

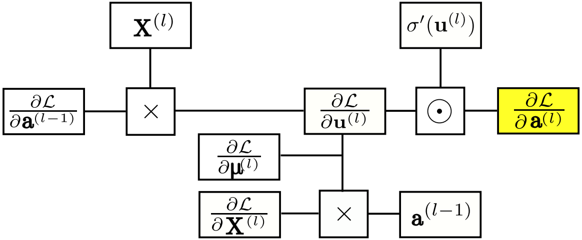


Figure: Computation of $\frac{\partial L}{\partial \mu^{(l)}}$, $\frac{\partial L}{\partial \mathbf{X}^{(l)}}$ and $\frac{\partial L}{\partial \boldsymbol{a}^{(l-1)}}$ starting from $\frac{\partial L}{\partial \boldsymbol{a}^{(l)}}$

Backward pass

Backward pass scheme
Input: Gradient of the loss w.r.t. the last layer values $\partial L / \partial \boldsymbol{a}^{(k)}$
1. For $l = k, \ldots, 1$ Compute $\frac{\partial L}{\partial \boldsymbol{u}^{(l)}} = \frac{\partial L}{\partial \boldsymbol{a}^{(l)}} \odot \sigma'(\boldsymbol{u}^{(l)})$ Compute $\frac{\partial L}{\partial \mathbf{X}^{(l)}} = \frac{\partial L}{\partial \boldsymbol{u}^{(l)}} (\boldsymbol{a}^{(l-1)})^T$, $\frac{\partial L}{\partial \mu^{(l)}} = \frac{\partial L}{\partial \boldsymbol{u}^{(l)}}$ Compute $\frac{\partial L}{\partial \boldsymbol{a}^{(l-1)}} = (\mathbf{X}^{(l)})^T \frac{\partial L}{\partial \boldsymbol{u}^{(l)}}$

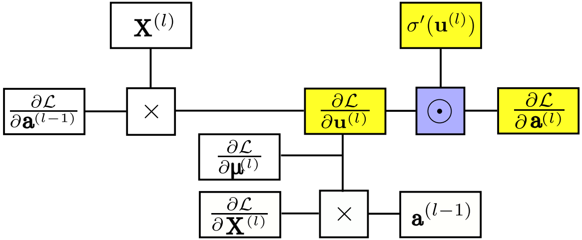


Figure: Computation of $\frac{\partial L}{\partial \mu^{(l)}}$, $\frac{\partial L}{\partial \mathbf{X}^{(l)}}$ and $\frac{\partial L}{\partial \boldsymbol{a}^{(l-1)}}$ starting from $\frac{\partial L}{\partial \boldsymbol{a}^{(l)}}$

Backward pass

Backward pass scheme
Input: Gradient of the loss w.r.t. the last layer values $\partial L / \partial \boldsymbol{a}^{(k)}$
1. For $l = k, \ldots, 1$ Compute $\frac{\partial L}{\partial \boldsymbol{u}^{(l)}} = \frac{\partial L}{\partial \boldsymbol{a}^{(l)}} \odot \sigma'(\boldsymbol{u}^{(l)})$ Compute $\frac{\partial L}{\partial \mathbf{X}^{(l)}} = \frac{\partial L}{\partial \boldsymbol{u}^{(l)}} (\boldsymbol{a}^{(l-1)})^T$, $\frac{\partial L}{\partial \mu^{(l)}} = \frac{\partial L}{\partial \boldsymbol{u}^{(l)}}$ Compute $\frac{\partial L}{\partial \boldsymbol{a}^{(l-1)}} = (\mathbf{X}^{(l)})^T \frac{\partial L}{\partial \boldsymbol{u}^{(l)}}$

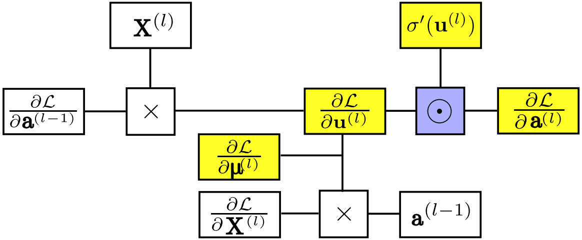


Figure: Computation of $\frac{\partial L}{\partial \mu^{(l)}}$, $\frac{\partial L}{\partial \mathbf{X}^{(l)}}$ and $\frac{\partial L}{\partial \boldsymbol{a}^{(l-1)}}$ starting from $\frac{\partial L}{\partial \boldsymbol{a}^{(l)}}$

Backward pass

Backward pass scheme
Input: Gradient of the loss w.r.t. the last layer values $\partial L/\partial \boldsymbol{a}^{(k)}$
1. **For** $l = k, \ldots, 1$ Compute $\frac{\partial L}{\partial \boldsymbol{u}^{(l)}} = \frac{\partial L}{\partial \boldsymbol{a}^{(l)}} \odot \sigma'(\boldsymbol{u}^{(l)})$ Compute $\frac{\partial L}{\partial \mathbf{X}^{(l)}} = \frac{\partial L}{\partial \boldsymbol{u}^{(l)}}(\boldsymbol{a}^{(l-1)})^T$, $\frac{\partial L}{\partial \mu^{(l)}} = \frac{\partial L}{\partial \boldsymbol{u}^{(l)}}$ Compute $\frac{\partial L}{\partial \boldsymbol{a}^{(l-1)}} = (\mathbf{X}^{(l)})^T \frac{\partial L}{\partial \boldsymbol{u}^{(l)}}$

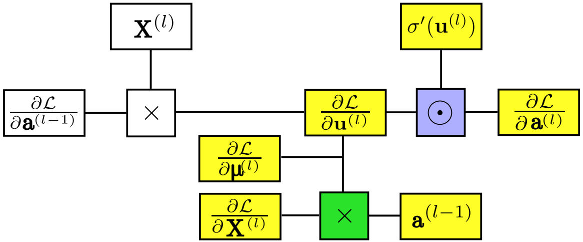


Figure: Computation of $\frac{\partial L}{\partial \mu^{(l)}}$, $\frac{\partial L}{\partial \mathbf{X}^{(l)}}$ and $\frac{\partial L}{\partial \boldsymbol{a}^{(l-1)}}$ starting from $\frac{\partial L}{\partial \boldsymbol{a}^{(l)}}$

Backward pass

Backward pass scheme
Input: Gradient of the loss w.r.t. the last layer values $\partial L / \partial \boldsymbol{a}^{(k)}$
1. For $l = k, \ldots, 1$ Compute $\frac{\partial L}{\partial \boldsymbol{u}^{(l)}} = \frac{\partial L}{\partial \boldsymbol{a}^{(l)}} \odot \sigma'(\boldsymbol{u}^{(l)})$ Compute $\frac{\partial L}{\partial \mathbf{X}^{(l)}} = \frac{\partial L}{\partial \boldsymbol{u}^{(l)}} (\boldsymbol{a}^{(l-1)})^T$, $\frac{\partial L}{\partial \mu^{(l)}} = \frac{\partial L}{\partial \boldsymbol{u}^{(l)}}$ Compute $\frac{\partial L}{\partial \boldsymbol{a}^{(l-1)}} = (\mathbf{X}^{(l)})^T \frac{\partial L}{\partial \boldsymbol{u}^{(l)}}$

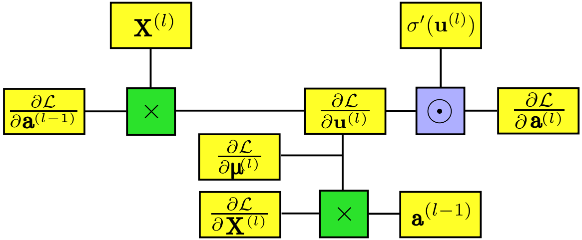


Figure: Computation of $\frac{\partial L}{\partial \mu^{(l)}}$, $\frac{\partial L}{\partial \mathbf{X}^{(l)}}$ and $\frac{\partial L}{\partial \boldsymbol{a}^{(l-1)}}$ starting from $\frac{\partial L}{\partial \boldsymbol{a}^{(l)}}$

Backpropagation

- Recursive computation of the derivative $\nabla_{\boldsymbol{x}} L_i(\boldsymbol{x})$

1. **Forward pass:** Compute all pre-activation and hidden layer values
2. **Backward pass:** Compute the derivative of L_i with respect to the weights and biases, from last to first layer.

Complexity of computing $\nabla_{\boldsymbol{x}} L_i(\boldsymbol{x})$

Method	Complexity
Naive derivative	$\mathcal{O}(k^2 m^2)$
Backpropagation	$\mathcal{O}(k m^2)$

Where m is number of neurons per layer and k is the number of layers.

Remarks:
- Complexity is reduced by reusing computations at each step (memoization).
- The backpropagation has the same complexity as the forward pass (but different constants).

Outline

Recall: Definition and representation of deep neural networks

Training deep networks

Computational infrastructure

Deep Learning Toolkit

Automatic Differentiation

- *Automatic differentiation* is a computational technique to compute the *exact* gradient of a function by keeping track of its inputs and intermediate values

Automatic Differentiation

- *Automatic differentiation* is a computational technique to compute the *exact* gradient of a function by keeping track of its inputs and intermediate values
- This removes the tedious manual derivation of the gradient and if implemented in a certain way, also reduces the backward pass complexity from $\mathcal{O}(km^2)$ to $\mathcal{O}(km)$
- For a thorough survey and explanation, see [5]

Automatic Differentiation (AD)

Table: Reverse mode AD example, with $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ evaluated at $(x_1, x_2) = (2, 5)$. After the forward evaluation of the primals on the left, the adjoint operations on the right are evaluated in reverse. Note that both $\frac{\partial y}{\partial x_1}$ and $\frac{\partial y}{\partial x_2}$ are computed in the same reverse pass, starting from the adjoint $\bar{v}_5 = \bar{y} = \frac{\partial y}{\partial y} = 1$. From [5]

Forward Primal Trace

$$
\begin{array}{lll}
v_{-1} = x_1 & & = 2 \\
v_0 & = x_2 & = 5 \\
\hline
v_1 & = \ln v_{-1} & = \ln 2 \\
v_2 & = v_{-1} \times v_0 & = 2 \times 5 \\
v_3 & = \sin v_0 & = \sin 5 \\
v_4 & = v_1 + v_2 & = 0.693 + 10 \\
v_5 & = v_4 - v_3 & = 10.693 + 0.959 \\
\hline
y & = v_5 & = 11.652
\end{array}
$$

Reverse Adjoint (Derivative) Trace

$$
\begin{array}{llll}
\boldsymbol{\bar{x}_1} & \boldsymbol{= \bar{v}_{-1}} & & \boldsymbol{= 5.5} \\
\boldsymbol{\bar{x}_2} & \boldsymbol{= \bar{v}_0} & & \boldsymbol{= 1.716} \\
\hline
\bar{v}_{-1} & = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}} & = \bar{v}_{-1} + \bar{v}_1 / v_{-1} & = 5.5 \\
\bar{v}_0 & = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0} & = \bar{v}_0 + \bar{v}_2 \times v_{-1} & = 1.716 \\
\bar{v}_{-1} & = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}} & = \bar{v}_2 \times v_0 & = 5 \\
\bar{v}_0 & = \bar{v}_3 \frac{\partial v_3}{\partial v_0} & = \bar{v}_3 \times \cos v_0 & = -0.284 \\
\bar{v}_2 & = \bar{v}_4 \frac{\partial v_4}{\partial v_2} & = \bar{v}_4 \times 1 & = 1 \\
\bar{v}_1 & = \bar{v}_4 \frac{\partial v_4}{\partial v_1} & = \bar{v}_4 \times 1 & = 1 \\
\bar{v}_3 & = \bar{v}_5 \frac{\partial v_5}{\partial v_3} & = \bar{v}_5 \times (-1) & = -1 \\
\bar{v}_4 & = \bar{v}_5 \frac{\partial v_5}{\partial v_4} & = \bar{v}_5 \times 1 & = 1 \\
\hline
\bar{v}_5 & = \bar{y} & = 1 &
\end{array}
$$

Autograd and Differentiable Programming

- Automatic differentiation + automatic construction of a computational graph from code
- Pedagogic version of autograd (with very readable code) available on github
- Industrial strength implementations used by Facebook and Google also available
- For cool applications on the extreme end see differentiable graphic rendering and differentiable convex optimization solvers.

Outline

Pytorch

- Popular machine learning framework developed by Facebook
- Key innovations: APIs (module structure, dataset) and dynamic graphs (helps debugging, later adopted by tensorflow as well)
- Other frameworks worth mentioning: tensorflow (Google), mxnet (Microsoft) and Flux.jl (for julia)
- *very good* manual and tutorial at https://pytorch.org/docs/stable/index.html
- Two introductory notebooks are provided in this supplementary

Deep Learning Building Blocks: Linear Layers

- $f_l : \mathbb{R}^n \to \mathbb{R}^m$
- $f_l(\mathbf{a}) = \mathbf{X}_l \mathbf{a} + \mu_l$
- Question: How shall we modify the previous 'Bias' class to implement a linear layer?

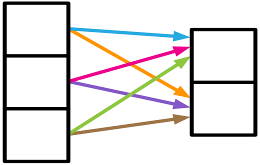


Figure: Linear Layer, from [4]

Deep Learning Building Blocks: Linear Layers

- $f_l : \mathbb{R}^n \to \mathbb{R}^m$
- $f_l(\mathbf{a}) = \mathbf{X}_l\mathbf{a} + \mu_l$
- pytorch implementation: `torch.nn.Linear`
- Multi-layer perceptron (MLP): Stack several (≥ 2) llinear layers, interleaved with activation functions.

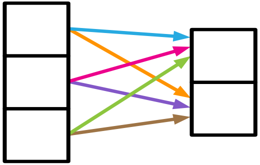
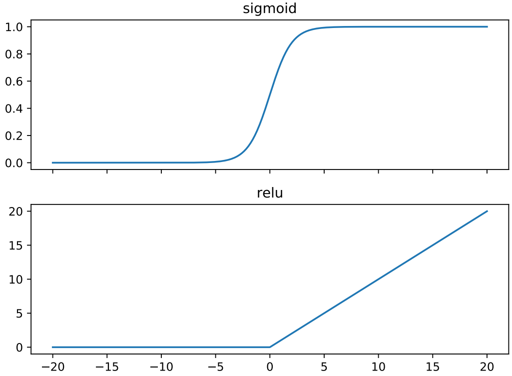


Figure: Linear Layer, from [4]

Deep Learning Building Blocks: Activation functions

- Non-linear functions that are applied element-wise and give the neural network its expressivity
- MLP without nonlinearity is just a factored linear layer
  $f_{\text{total}}(\boldsymbol{a}) = \mathbf{X}_{\text{total}}\boldsymbol{a} + \mu_{\text{total}} = \mathbf{X}_2\mathbf{X}_1\boldsymbol{a} + \mathbf{X}_2\mu_1 + \mu_2$
- Historically sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$ was common,but due to optimization issues, nowadays the **re**ctified **l**inear **u**nit (RELU) $\sigma(x) = \mathrm{relu}(x) = \max(x, 0)$ is the most common
- $f_{\text{total}}(\boldsymbol{a}) = \mathbf{X}_2\sigma(\mathbf{X}_1\boldsymbol{a} + \mu_1) + \mu_2$ is the minimal "deep" neural network, the "deep" refers to the nonlinearity "hiding" the inner projection
- `torch.nn.ReLu` and `torch.nn.Sigmoid` respectively
- Question: How can we implement an MLP class?



Deep Learning Building Blocks: Loss functions

- Express the task that your model is intended to perform on the data
- For regression, a sensible default is the mean square error $MSE(a,b) = \frac{1}{N}\sum_{i=1}^{N}(a_i - b_i)^2$
- For classification, a sensible default is cross-entropy $H(a,b) = -\sum_{i=1}^{N} a_i \log b_i$ with $a, b \in [0,1]$
- `torch.nn.MSELoss` and `torch.nn.CrossEntropyLoss` respectively

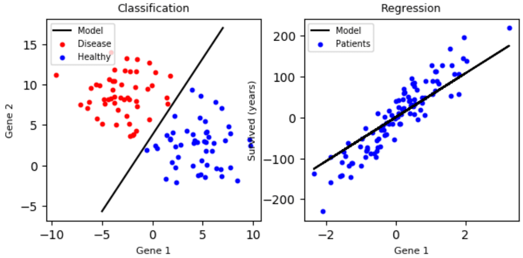


Figure: From [7]

Deep Learning Building Blocks: Convolutional Layers

- apply an MLP across spatial locations of an image to learn a filter
- $f_l : \mathbb{R}^{n \times H \times W} \to \mathbb{R}^{m \times H-k+1 \times W-k+1}$
- $f_l(\mathbf{a}) = \begin{bmatrix} \sum_{i=1}^{n} X_{l,1,i} \star a_i + \mu_1 \\ \vdots \\ \sum_{i=1}^{n} X_{l,o,i} \star a_i + \mu_o \end{bmatrix}$
- $x \star a$ denotes the cross correlation operator, i.e. a sliding window inner product: $(x \star a)_i = \sum_{j=1}^{k} x_{i+j-1} a_j$. The window size k is also called kernel size
- *reduces* spatial dimensions, effectively *subsampling* the input, other parameters include stride and dilation (can find explanations online)
- PyTorch implementation: `torch.nn.ConvNd` for $N \in \{1, 2, 3\}$ dimensional convolution

Convolution layer

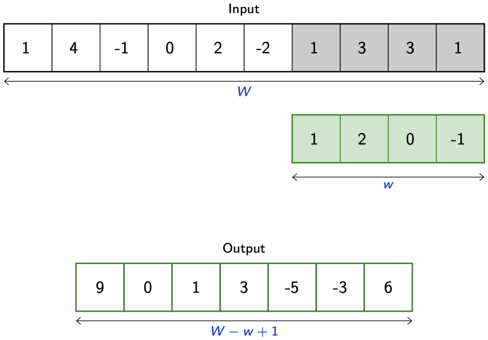


Figure: Convolution operation, from [6]

Deep Learning Building Blocks: Attention Layers

- $f_l : \mathbb{R}^{T,n} \to \mathbb{R}^{T,m}$,explicitly maps between sets
- $f_l(\mathbf{a}) = \mathrm{Attention}(\mathbf{a})X_v\mathbf{a}$
- where the $\mathrm{Attention}(mba)$ is a weight matrix defined rowise as $\mathrm{Attention}(\mathbf{a})_r = \mathrm{softmax}\left(\left(X_q\mathbf{a}\mathbf{a}^T X_k^T\right)_r\right)$
- originally conceived to help RNNs with long range dependencies, requires explicit order encoding
- currently *dominates* both RNNs and CNNs for sequence and vision tasks
- computationally and memory heavy in the original form$\mathcal{O}(T^2)$ but recent work improved this to $\approx \mathcal{O}(T)$.

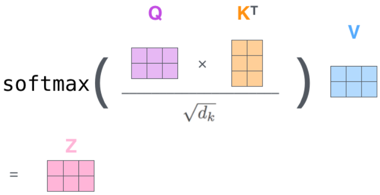


Figure: Attention Layer, from [3]

References I

[1] Tensors—Representation of Data In Neural Networks, Dec 2019.
[Online; accessed 22. Oct. 2020].
(Cited on page 5.)

[2] File:Recurrent neural network unfold.svg - Wikimedia Commons, Oct 2020.
[Online; accessed 15. Oct. 2020].
(Cited on page 40.)

[3] Jay Alammar.
The Illustrated Transformer, Oct 2020.
[Online; accessed 15. Oct. 2020].
(Cited on page 36.)

[4] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andrew Ballard, Justin Gilmer, George Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu.
Relational inductive biases, deep learning, and graph networks, 2018.
(Cited on pages 31, 32, and 42.)

References II

[5] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind.
Automatic differentiation in machine learning: a survey, 2018.
(Cited on pages 25, 26, and 27.)

[6] Francois Fleuret.
7.1. Transposed convolutions, 2023.
(Cited on pages 35 and 41.)

[7] petercour.
Machine Learning Classification vs Regression.
DEV Community, Jul 2019.
(Cited on page 34.)

Advanced material

Optional reading material for additional building blocks and the complexity of backpropagation.

Deep Learning Building Blocks: Recurrent Layers

- introduces hidden state $h_t \in R^H$ into the training process, generally trained through unrolling the computation graph across time steps T
- $f_l : \mathbb{R}^n \times \mathbb{R}^H \to \mathbb{R}^m \times \mathbb{R}^H$, but when training implicitly $\mathbb{R}^{T,n} \to \mathbb{R}^{T,m} \times \mathbb{R}^{T,H}$ since we unroll through time
- $f_l(\mathbf{a}_t), h_t = g(\mathbf{a}_{t-1}, h_{t-1})$
- h_0 is some initial value,often the zero vector
- $g(\dot{)}$ is usually the GRU or LSTM unit which uses an MLP to predict updates to the hidden state as well as the output

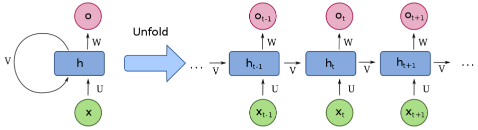


Figure: Recurrent Layer, from [2]

Deep Learning Building Blocks: Transposed-Convolutional Layers

- apply an MLP across spatial locations of an image to learn a filter
- $f_l : \mathbb{R}^{n \times H \times W} \to \mathbb{R}^{m \times H+k-1 \times W+k-1}$
- $f_l(\mathbf{a}) = \begin{bmatrix} \sum_{i=1}^{n} X_{l,1,i} \star a_i + \mu_1 \\ \vdots \\ \sum_{i=1}^{n} X_{l,o,i} \star a_i + \mu_o \end{bmatrix}$
- $x \star a$ denotes the transposed cross correlation operator: $(x \star a)_i = \sum_{j=1}^{k} x_j a_{i+j-1}$.
- *increases* spatial dimensions, effectively *oversampling* the input, other parameters include stride and dilation (can find explanations online)
- PyTorch implementation: `torch.nn.ConvTransposeNd` for $N \in \{1, 2, 3\}$ dimensional transposed convolution

Transposed convolution layer

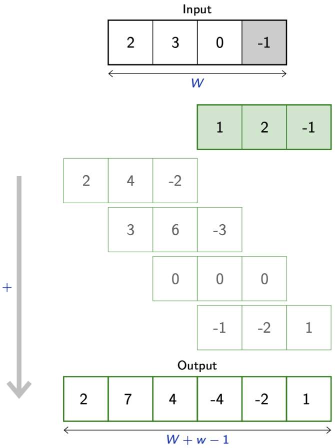


Figure: Transposed convolution operation, from [6]

Deep Learning Building Blocks: Graphical Layers

- generalizes attention to sparse,non-euclidean geometry and variable cardinality sets of inputs
- $f_l : E_{l-1}, V_{-1}, u_{-1} \to E_l, V_l, u^l$ where the E_i, V_i are the sets of edge and node attribute tensors respectively and u is a graph level attribute tensor (from [4])
- updates to e, v, u tensors follow the template $x = \mathrm{Agg}(\mathrm{Proj}(\mathrm{Neigh}(x)))$ i.e. we **agg**regate the **proj**ected elements of the **neigh**ourhood set of an element x.
- Examples for Agg are `sum`,`mean`,`max`, Proj is usually a neural network and the Neigh set are the nodes connected by an edge, to a node by various edges or all elements in the graph respectively for e, v, u.

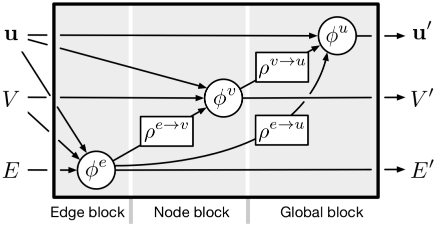


Figure: Graphical Layer, from [4]

Using networks for multi-class classification

Definition (Score-based classifier)

For a network $h : \mathbb{R}^d \to \mathbb{R}^c$ define the score-based classifier $i_h : \mathbb{R}^d \to \{1, \ldots, c\}$ as

$$i_h(\mathbf{a}) = \arg\max_{i \in \{1,\ldots,c\}} [h(\mathbf{a})]_i$$

One output per class, choose the class corresponding to the maximum output. Example:

$$f(\mathbf{x}_0) = \begin{bmatrix} 0.1 \\ -0.8 \\ \mathbf{1.4} \\ 1.1 \end{bmatrix} \qquad \Longrightarrow i_f(\mathbf{a}) = 3$$

Definition (Cross-entropy loss)

Let $\mathbf{a} \in \mathbb{R}^d$ be a sample with label $b \in \{1, \ldots, c\}$

$$L(h(\mathbf{a}), b) = -\log\left(\frac{\exp(h(\mathbf{a})_b)}{\sum_{j=1}^{c} \exp(h(\mathbf{a})_j)}\right)$$

$\mathbf{e}_i \in \mathbb{R}^c$ denotes the i-th canonical vector.

Most common loss for classification via ERM with neural networks. Example:

$$h(\mathbf{a}) = \begin{bmatrix} 0.1 \\ \mathbf{-0.8} \\ \mathbf{1.4} \\ 1.1 \end{bmatrix} \quad \begin{array}{l} L(f(\mathbf{a}), 2) = 2.95 \\ \\ L(f(\mathbf{a}), 3) = 0.75 \end{array}$$

Complexity of Backpropagation

The size of each layer (including input) is $\mathcal{O}(m)$, and the number of layers is $\mathcal{O}(k)$.

Forward pass scheme
1. For $l = 1, \ldots, k$ ▶ $\boldsymbol{u}^{(l)} = X^{(l)}\boldsymbol{a}^{(l-1)} + \mu^{(l)}$ ▶ $\boldsymbol{a}^{(l)} = \sigma(\boldsymbol{u}^{(l)})$

Backward pass scheme
1. For $l = k, \ldots, 1$ ▶ $\frac{\partial L}{\partial \boldsymbol{u}^{(l)}} = \frac{\partial L}{\partial \boldsymbol{a}^{(l)}} \odot \sigma'(\boldsymbol{u}^{(l)})$ ▶ $\frac{\partial L}{\partial X^{(l)}} = \frac{\partial L}{\partial \boldsymbol{u}^{(l)}}(\boldsymbol{a}^{(l-1)})^T$ ▶ $\frac{\partial L}{\partial \mu^{(l)}} = \frac{\partial L}{\partial \boldsymbol{u}^{(l)}}$ ▶ $\frac{\partial L}{\partial \boldsymbol{a}^{(l-1)}} = (X^{(l)})^T \frac{\partial L}{\partial \boldsymbol{u}^{(l)}}$

Complexity of Backpropagation

The size of each layer (including input) is $\mathcal{O}(m)$, and the number of layers is $\mathcal{O}(k)$.

Forward pass scheme
1. For $l = 1, \ldots, k$ ▶ $\boldsymbol{u}^{(l)} = X^{(l)}\boldsymbol{a}^{(l-1)} + \mu^{(l)} \Rightarrow \mathcal{O}(m^2)$ ▶ $\boldsymbol{a}^{(l)} = \sigma(\boldsymbol{u}^{(l)}) \Rightarrow \mathcal{O}(m)$

Forward pass is $\mathcal{O}(km^2)$

Backward pass scheme
1. For $l = k, \ldots, 1$ ▶ $\frac{\partial L}{\partial \boldsymbol{u}^{(l)}} = \frac{\partial L}{\partial \boldsymbol{a}^{(l)}} \odot \sigma'(\boldsymbol{u}^{(l)})$ ▶ $\frac{\partial L}{\partial X^{(l)}} = \frac{\partial L}{\partial \boldsymbol{u}^{(l)}} (\boldsymbol{a}^{(l-1)})^T$ ▶ $\frac{\partial L}{\partial \mu^{(l)}} = \frac{\partial L}{\partial \boldsymbol{u}^{(l)}}$ ▶ $\frac{\partial L}{\partial \boldsymbol{a}^{(l-1)}} = (X^{(l)})^T \frac{\partial L}{\partial \boldsymbol{u}^{(l)}}$

Complexity of Backpropagation

The size of each layer (including input) is $\mathcal{O}(m)$, and the number of layers is $\mathcal{O}(k)$.

Forward pass scheme

1. **For** $l = 1, \ldots, k$
   - $\boldsymbol{u}^{(l)} = X^{(l)}\boldsymbol{a}^{(l-1)} + \mu^{(l)} \Rightarrow \mathcal{O}(m^2)$
   - $\boldsymbol{a}^{(l)} = \sigma(\boldsymbol{u}^{(l)}) \Rightarrow \mathcal{O}(m)$

Forward pass is $\mathcal{O}(km^2)$

Backward pass scheme

1. **For** $l = k, \ldots, 1$
   - $\frac{\partial L}{\partial \boldsymbol{u}^{(l)}} = \frac{\partial L}{\partial \boldsymbol{a}^{(l)}} \odot \sigma'(\boldsymbol{u}^{(l)}) \Rightarrow \mathcal{O}(m)$
   - $\frac{\partial L}{\partial X^{(l)}} = \frac{\partial L}{\partial \boldsymbol{u}^{(l)}} (\boldsymbol{a}^{(l-1)})^T \Rightarrow \mathcal{O}(m^2)$
   - $\frac{\partial L}{\partial \mu^{(l)}} = \frac{\partial L}{\partial \boldsymbol{u}^{(l)}} \Rightarrow \mathcal{O}(1)$
   - $\frac{\partial L}{\partial \boldsymbol{a}^{(l-1)}} = (X^{(l)})^T \frac{\partial L}{\partial \boldsymbol{u}^{(l)}} \Rightarrow \mathcal{O}(m^2)$

Backward pass is $\mathcal{O}(km^2)$