Deep Learning

2. Deep Feedforward Networks

Lars Schmidt-Thieme

Information Systems and Machine Learning Lab (ISMLL)
Institute for Computer Science
University of Hildesheim, Germany

Syllabus

Tue. 21.4.	(1)	1. Supervised Learning (Review 1)
Tue. 28.4.	(2)	2. Neural Networks (Review 2)
Tue. 5.5.	(3)	3. Regularization
Tue. 12.5.	(4)	4. Optimization
Tue. 19.5.	(5)	5. Convolutional Neural Networks
Tue. 26.5.	(6)	6. Recurrent Neural Networks
Tue. 2.6.	—	— *Pentecoste Break* —
Tue. 9.6.	(7)	7. Autoencoders
Tue. 16.6.	(8)	8. Generative Adversarial Networks
Tue. 23.6.	(9)	9. Recent Advances
Tue. 30.6.	(10)	10. Engineering Deep Learning Models
Tue. 7.7.	(11)	tbd.
Tue. 14.7.	(12)	Q & A

Outline

1. What is a Neural Network?
2. An example: XOR
3. Loss and Output Layer
4. Basic Feedforward Network Architecture
5. Backpropagation

Outline

What is a Deep Feedforward Network (DFN)?

- **Feedforward networks**
  (aka **feedforward neural networks** or **multilayer perceptrons**)
- Given a function $\mathbf{y} = \mathbf{f}^*(\mathbf{x})$ that maps input $\mathbf{x}$ to output $\mathbf{y}$
- A DFN defines a parametric mapping $\hat{\mathbf{y}} = \mathbf{f}(\mathbf{x}; \theta)$ with parameters θ
- Aim is to learn θ such as $\mathbf{f}(\mathbf{x}; \theta)$ *best* approximates $\mathbf{f}^*(\mathbf{x})$!

Why *Feedforward*?

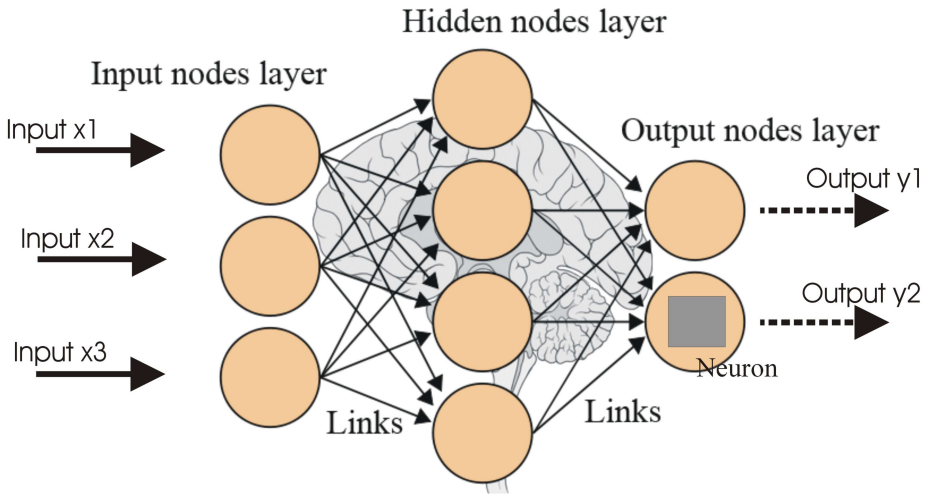
- Given a Feedforward Network $\hat{\mathbf{y}} = \mathbf{f}(\mathbf{x}; \theta)$
  - Input $\mathbf{x}$, then pass through a chain of steps before outputting $\mathbf{y}$
- Example $f^1(x)$, $f^2(x)$ and $f^3(x)$ can be chained as:
  - $f(x) = f^3(f^2(f^1(x)))$
    - x is the zero-th layer, or the **input** layer
    - f^1 is the first layer, or the **first hidden** layer
    - f^2 is the second layer, or a **second hidden** layer
    - f^3 is the last layer, or the **output** layer
- No feedback exists between the steps of the chain
  - Feedback connections yield the **Recurrent Neural Network**
- Number of hidden layers define the **depth** of the network
- Dimensionality of the hidden layers defines the **width** of the network

Why *Neural*?

- Loosely inspired by neuroscience, hence Artificial Neural Network
- Each hidden layer node resembles a neuron
- Input to a neuron are the synaptic connections from the previous attached neuron
- Output of a neuron is an aggregation of the input vector
- Signal propagates forward in a chain of "Neuron"-to-"Neuron" transmissions
- However, modern Deep Learning research is steered mainly by mathematical and engineering principles!

Why *Network*?

- A feed-forward network is an **acyclic directed graph**, but
  - Graph nodes are structured in layers
  - Directed links between nodes are **parameters/weights**
  - Each node is a computational functions
  - No inter-layer and intra-layer connections (but possible)
  - Input to the first layer is given (the features $\mathbf{x}$)
  - Output is the computation of the last layer (the target $\hat{\mathbf{y}}$)

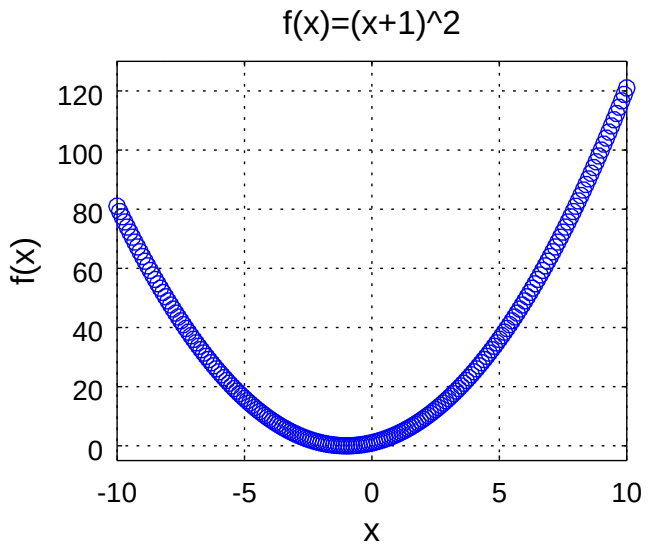


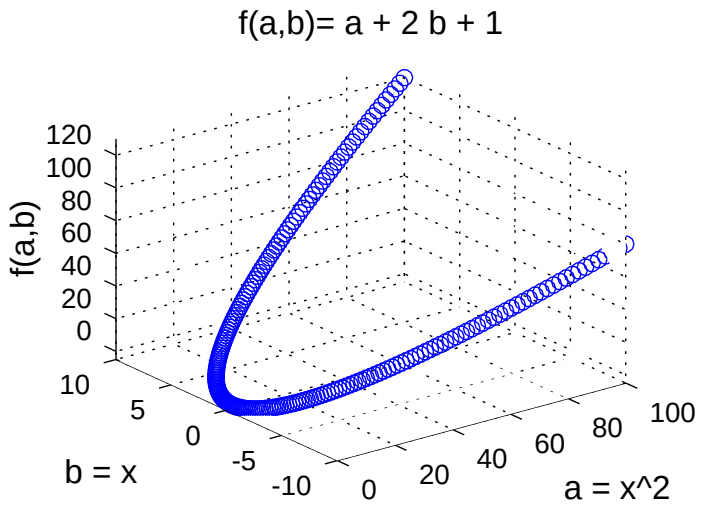
Nonlinear Mapping

- We can easily solve linear regression, but not every problem is linear.
- Can the function $f(x) = (x + 1)^2$ be approximated through a linear function?

Nonlinear Mapping

- We can easily solve linear regression, but not every problem is linear.
- Can the function $f(x) = (x+1)^2$ be approximated through a linear function?
- Yes, but only if we **map** the feature x into a new space:





Nonlinear Mapping (II)

- Which mapping $\phi(x)$ is the best?

There are various ways of designing $\phi(x)$:

1. Hand-craft (manually engineer) $\phi(\mathbf{x})$
2. Use a very generic $\phi(\mathbf{x})$, RBF or polynomial expansion
3. Parametrize and learn the mapping $\mathbf{f}(\mathbf{x}; \theta, \mathbf{w}) := \phi(\mathbf{x}; \theta)^\mathsf{T}\mathbf{w}$

Deep Feedforward Networks follow the third approach, where:

- the hidden layers (weights θ) learn the mapping $\phi(x; \theta)$
- the output layer (weights w) learns the function $g(z; w) := z^T w$

Nonlinear Mapping

- consider the function

$$f(x) = x^2 + 2e^x + 3x - 5$$

- from which latent features can it be linearly combined:
  A. x^2
  B. x^2, x, e^x
  C. x, e^x
  D. $x^2, x, e^x, \sin(x)$

Outline

An example - Learn XOR

- XOR is a function:

x_1	x_2	$y = f^*(x)$
0	0	0
0	1	1
1	0	1
1	1	0

- Can we learn a DFN $\hat{\mathbf{y}} = \mathbf{f}(\mathbf{x}; \theta)$ such that f resembles f^*?
- Our dataset
$\mathcal{D}^{\text{train}} := \{(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, 0), (\begin{pmatrix} 1 \\ 0 \end{pmatrix}, 1), (\begin{pmatrix} 0 \\ 1 \end{pmatrix}, 1), (\begin{pmatrix} 1 \\ 1 \end{pmatrix}, 0)\}$
- Leading to the optimization:

$$\arg\min_{\theta} J(\theta) := \frac{1}{4} \sum_{(x,y) \in \mathcal{D}^{\text{train}}} (y - f(x; \theta))^2$$

An example - Learn XOR (2)

- We will learn a simple DFN with one hidden layer:

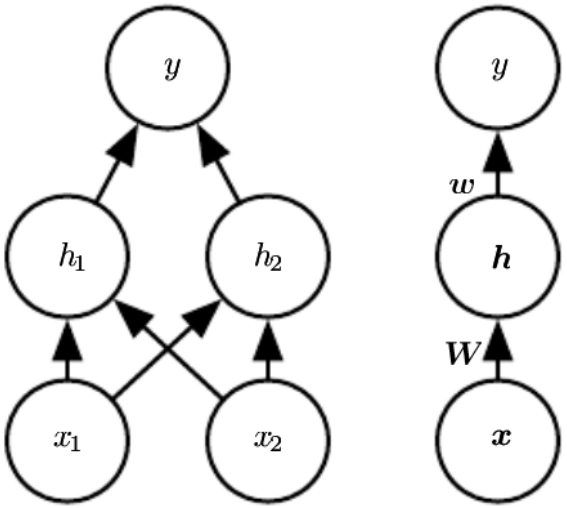


Figure 3: Left: Detailed, Right: Compact, Source: Goodfellow et al., 2016

- Two functions are chained $h = f^1(x; W, c)$ and $y = f^2(h; w, b)$
  - For n-th instance: Hidden-layer $h_i^{(n)} = g\left(W_{:,i}^T x^{(n)} + c_i\right)$
  - For n-th instance: output layer: $\hat{y}_n = w^T h^{(n)} + b$
  - $W \in \mathbb{R}^{2\times 2}, c \in \mathbb{R}^{2\times 1}, w \in \mathbb{R}^{2\times 1}, b \in \mathbb{R}$

Rectified Linear Unit (ReLU)

non-linear activation function:

$$\text{relu}(z) := \max\{0, z\}$$

node:

$$f(z) := \text{relu}(Wz) = \max\{0, Wz\} = (\max\{0, (W_{k,.}^T z)\})_{k=1:K}$$

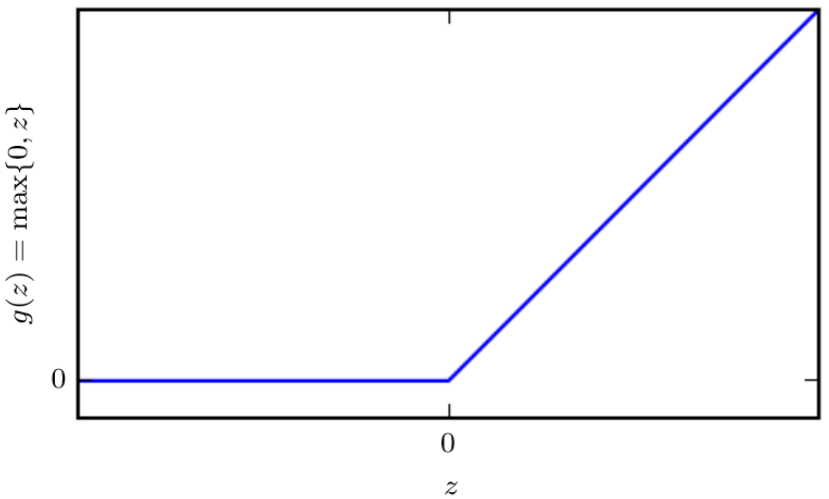


Figure 4: The ReLU activation, Source: Goodfellow et al., 2016

"Deus ex machina" solution?

Suppose I *magically* found out that:

$$W = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \; c = \begin{pmatrix} 0 \\ -1 \end{pmatrix}, \; w = \begin{pmatrix} 1 \\ -2 \end{pmatrix}, \; b = 0$$

We would later on see an optimization technique called backpropagation to learn the network parameters.

XOR Solution - Hidden Layer Computations

$$
\begin{aligned}
h_1^{(1)} &= g\left(W_{:,1}^T x_1 + c\right) = g\left(\begin{pmatrix}1 & 1\end{pmatrix}\begin{pmatrix}0\\0\end{pmatrix} + 0\right) = g(0) = 0\\
h_2^{(1)} &= g\left(W_{:,2}^T x_1 + c\right) = g\left(\begin{pmatrix}1 & 1\end{pmatrix}\begin{pmatrix}0\\0\end{pmatrix} - 1\right) = g(-1) = 0\\
h_1^{(2)} &= g\left(W_{:,1}^T x_2 + c\right) = g\left(\begin{pmatrix}1 & 1\end{pmatrix}\begin{pmatrix}0\\1\end{pmatrix} + 0\right) = g(1) = 1\\
h_2^{(2)} &= g\left(W_{:,2}^T x_2 + c\right) = g\left(\begin{pmatrix}1 & 1\end{pmatrix}\begin{pmatrix}0\\1\end{pmatrix} - 1\right) = g(0) = 0\\
h_1^{(3)} &= g\left(W_{:,1}^T x_3 + c\right) = g\left(\begin{pmatrix}1 & 1\end{pmatrix}\begin{pmatrix}1\\0\end{pmatrix} + 0\right) = g(1) = 1\\
h_2^{(3)} &= g\left(W_{:,2}^T x_3 + c\right) = g\left(\begin{pmatrix}1 & 1\end{pmatrix}\begin{pmatrix}1\\0\end{pmatrix} - 1\right) = g(0) = 0\\
h_1^{(4)} &= g\left(W_{:,1}^T x_4 + c\right) = g\left(\begin{pmatrix}1 & 1\end{pmatrix}\begin{pmatrix}1\\1\end{pmatrix} + 0\right) = g(2) = 2\\
h_2^{(4)} &= g\left(W_{:,2}^T x_4 + c\right) = g\left(\begin{pmatrix}1 & 1\end{pmatrix}\begin{pmatrix}1\\1\end{pmatrix} - 1\right) = g(1) = 1
\end{aligned}
$$

XOR Solution - Output Layer Computations

$$
\begin{aligned}
\hat{y}^{(1)} &= w^T h^{(1)} + b = \begin{pmatrix} 1 & -2 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} + 0 = 0 \\
\hat{y}^{(2)} &= w^T h^{(2)} + b = \begin{pmatrix} 1 & -2 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} + 0 = 1 \\
\hat{y}^{(3)} &= w^T h^{(3)} + b = \begin{pmatrix} 1 & -2 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} + 0 = 1 \\
\hat{y}^{(4)} &= w^T h^{(4)} + b = \begin{pmatrix} 1 & -2 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \end{pmatrix} + 0 = 0
\end{aligned}
$$

The computations of the final layer match exactly those of the XOR function.

Outline

Maximum Likelihood as Objective

- The loss can be expressed in probabilistic terms as

$$J(\theta) = -\mathbf{E}_{(x,y)\sim p_{data}} \log p_{\text{model}}(y \mid x)$$

- If our model outputs normal uncertainty:

$$p_{\text{model}}(y \mid x) = \mathcal{N}(y; f(x;\theta), \sigma^2)$$

$$\rightsquigarrow \quad J(\theta) = \frac{1}{2}\mathbf{E}_{(x,y)\sim p_{data}}(y - f(x;\theta))^2 + \text{const}$$

- the model just outputs the mean $f(x;\theta)$,
  σ^2 is its error variance.

Output Layer — Gaussian Output Distribution

- Affine transformation without nonlinearity
  - Given features h, produces $\hat{y} = w^T h + b$
  - activation function is the identify $a(h) := h$

- Interpreted as mean of a conditional Gaussian distribution
  - $p(y \mid x) = \mathcal{N}(y; \hat{y}, \sigma^2), \quad \hat{y} := f(x; \theta)$

Bernoulli Output Distributions

- Binary target variables follow a Bernoulli distribution
  $P(y = 1) = p,\ P(y = 0) = 1 - p$
- Train a DFN such that $\hat{y} = f(x; \theta) \in [0, 1]$
- Naive Option: clip a linear output layer:
  - $P(y = 1 \mid x) = \max\left\{0, \min\left\{1, w^T h + b\right\}\right\}$
- What is the problem with the clipped linear output layer?

Bernoulli Output Distributions (2)

- Use a smooth sigmoid output unit:

$$\begin{aligned} \hat{y} &= \sigma(z) = \frac{e^z}{e^z + 1} \\ z &= w^T h + b \end{aligned}$$

- The loss for a DFN $f(x;\theta)$ with a sigmoid output is:

$$J(\theta) = \sum_{n=1}^{N} -y_n \log(f(x_n;\theta)) - (1 - y_n) \log(1 - f(x_n,\theta))$$

- Also called **cross entropy**

Multinoulli Output Distribution

- For multi-category targets $\hat{y}_c = P(y = c \mid x),\ c \in \{1, \ldots, C\}$
- last latent layer: unnormalized log probabilities:

$$z_c = \log \tilde{P}(y = c \mid x) := w_c^T h + b$$

- yields probabilities:

$$P(y = c \mid x) := \text{softmax}(z)_c := \frac{e^{z_c}}{\sum_d e^{z_d}}$$

- Minimizing the log-likelihood loss:

$$\begin{aligned} J(\theta) &= \sum_{n=1}^{N} \sum_{c=1}^{C} -\mathbb{I}(y_n = c) \log P(y = c \mid x) \\ &= -\sum_{n=1}^{N} \sum_{c=1}^{C} \mathbb{I}(y_n = c) \left(z_c - \log \sum_d e^{z_d} \right) \end{aligned}$$

Outline

Types of Hidden Units

- Question: Can we use no activation function,i.e., only purely linear layers $h = W^T x + b$?

Types of Hidden Units

- Question: Can we use no activation function,i.e., only purely linear layers $h = W^T x + b$?
- Remember the most used hidden layer is ReLU:

$$h = \text{relu}(W^T x + b) = \max(0, W^T x + b)$$

- Alternatively, the sigmoid function:

$$h = \sigma(z)$$

- or, the hyperbolic tangent:

$$h = \tanh(z) = 2\sigma(2z) - 1$$

Architecture of Hidden Layers

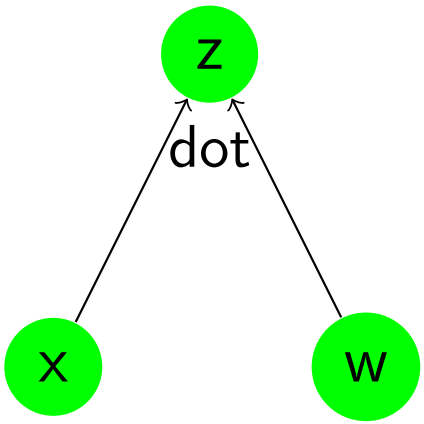
A DFN with L hidden layers:

$$
\begin{aligned}
h^{(1)} &= g^{(1)}(W^{(1)^T} x + b^{(1)}) \\
h^{(2)} &= g^{(2)}(W^{(2)^T} h^{(1)} + b^{(2)}) \\
&\dots \\
h^{(L)} &= g^{(L)}(W^{(L)^T} h^{(L-1)} + b^{(L)})
\end{aligned}
$$

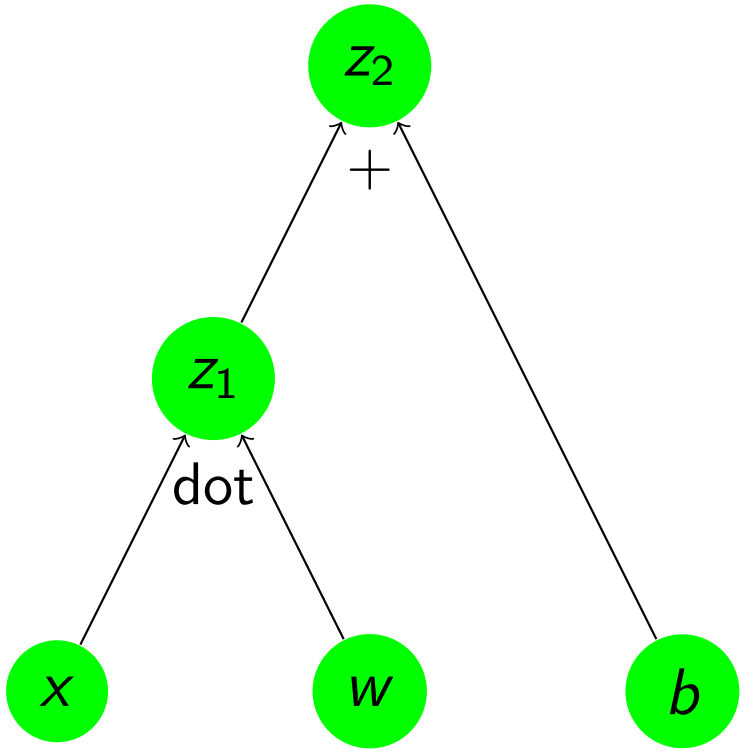
Different layers can have different activation functions.

Outline

Computational Graphs

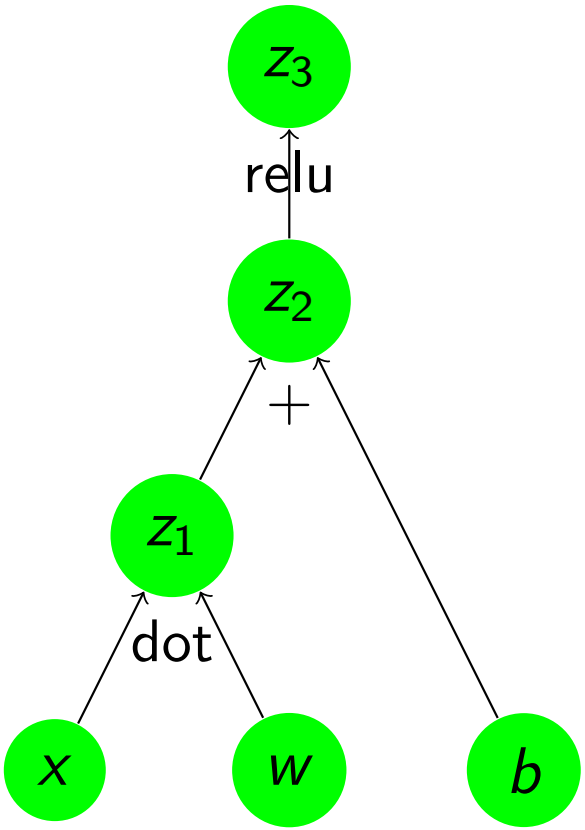


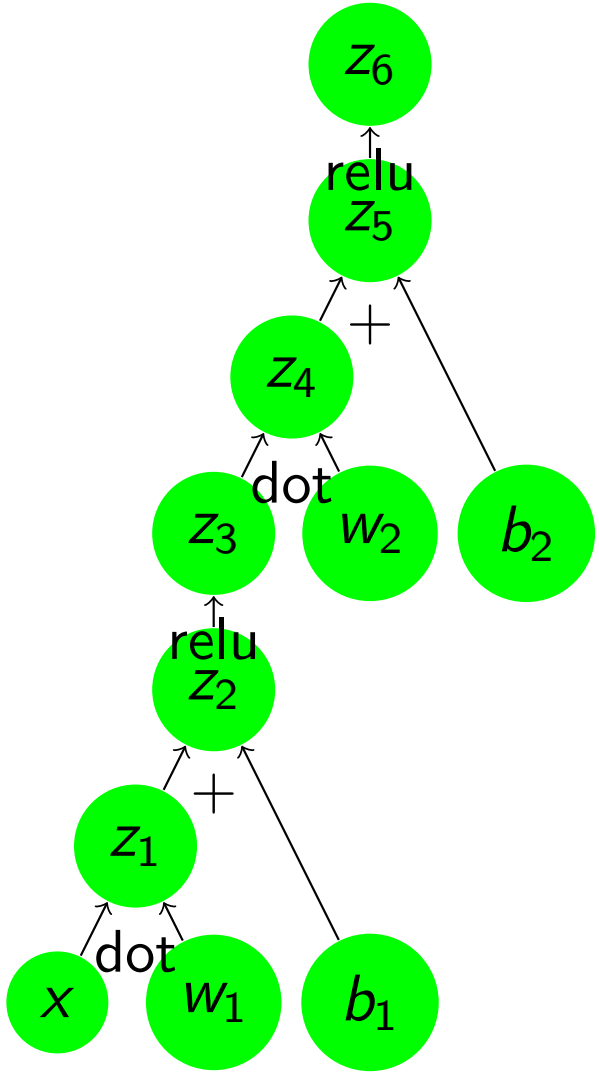
$$z = x^T w$$



$$z_2 = z_1 + b = x^T w + b$$

Computational Graphs





$$z_3 = \text{relu}(x^T w + b)$$

$$z_6 = \text{relu}(w_2^T \text{relu}(x^T w_1 + b_1) + b_2)$$

Forward Computation

- computational graph (Z, E), a DAG.
  - Z a set of node IDs.
  - $E \subseteq Z \times Z$ a set of directed edges.

  For every node $z \in Z$:
  - T_z: domain of the node (e.g., $\mathbb{R}^{17}$)

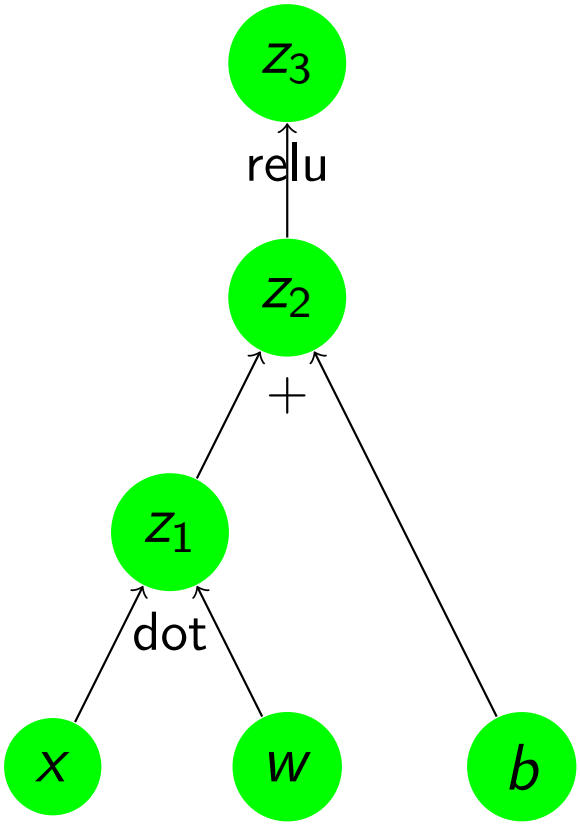
  and additionally for every non-root node $z \in Z$:
  - $f_z : \prod_{z' \in \text{fanin}(z)} T_{z'} \to T_z$ **node operation**

- **forward computation**:
  Given values $v_z \in T_z$ of all the root nodes $z \in Z$,
  compute a value for every node $z \in Z$ via

$$v_z := f_z(\underbrace{(v_{z'})_{z' \in \text{fanin}(z)}}_{=: v_{\text{fanin}(z)}})$$

Note: $\text{fanin}_{(Z,E)}(z) := \{z' \in Z \mid (z', z) \in E\}$ nodes with edges into z.

Forward Computation / Example



$$z_3 = \text{relu}(x^T w + b)$$

- types for each node:

$$T_x := \mathbb{R}^2, T_w := \mathbb{R}^2, T_b := \mathbb{R},$$
$$T_{z_1} = T_{z_2} = T_{z_3} := \mathbb{R}$$

- functions for each non-root node:

$$f_1(x, w) := x^T w, \quad f_2(z_1, b) = z_1 + b,$$
$$f_3(z_2) := \text{relu}(z_2)$$

- given values for all root nodes:

$$x = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, w = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, b = 0.5$$

- compute values for all non-root nodes:

$$z_1 = 1, \quad z_2 = 1.5, \quad z_3 = 1.5$$

Forward Computation / Algorithm

cg-forward$((Z, E, f), (v_z)_{z \in \text{roots}(Z,E)})$:
  for $z \in Z \setminus \text{roots}(Z, E)$:
    $v_z := f_z((v_{z'})_{z' \in \text{fanin}(z)})$
  return $(v_z)_{z \in Z}$

Note: $x^\frown y$ denotes the concatenation of two lists.

Forward Computation / Algorithm

cg-forward$((Z, E, f), (v_z)_{z \in \text{roots}(Z,E)})$:
  for $z \in Z \setminus \text{roots}(Z, E)$ in topological order:
    $v_z := f_z((v_{z'})_{z' \in \text{fanin}(z)})$
  return $(v_z)_{z \in Z}$

topological-order(Z, E) :
  $x := ()$
  while $Z \neq \emptyset$:
    choose $z \in \text{roots}(Z, E)$ arbitrarily
    delete z in graph (Z, E)
    $x := x^\frown(z)$
  return x

Note: $x^\frown y$ denotes the concatenation of two lists.

Gradients in Computational Graphs (1/2)

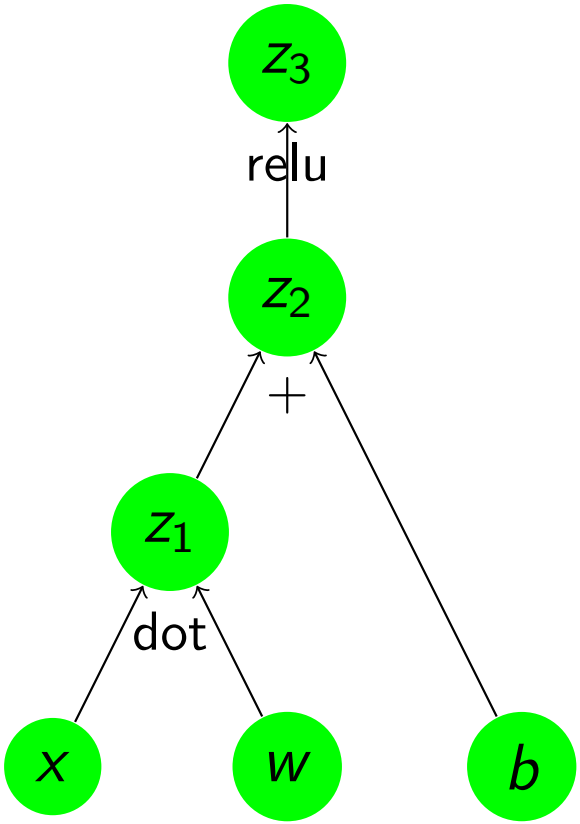
- lets assume every operation is differentiable and
  we have for every non-root node z its gradients functions $\nabla_{z'} z = \frac{\partial z}{\partial z'}$:

$$g_{z,z'} : \prod_{\tilde{z}' \in \text{fanin}(z)} T_{\tilde{z}'} \rightarrow T_z \times T_{z'}, \quad z' \in \text{fanin}(z)$$

- for any node then we can compute its gradient values w.r.t. its inputs:

$$w_{z,z'} = g_{z,z'}\left((v_{\tilde{z}'})_{\tilde{z}' \in \text{fanin}(z)}\right)$$

Gradient Computation / Example



$$z_3 = \text{relu}(x^T w + b)$$

- functions for each non-root node:

$$f_1(x, w) := x^T w, \quad f_2(z_1, b) = z_1 + b,$$
$$f_3(z_2) := \text{relu}(z_2)$$

- gradient functions for each non-root node:

$$g_{1,x}(x, w) := w, \quad g_{1,w}(x, w) := x,$$
$$g_{2,z_1}(z_1, b) := 1, \quad g_{2,b}(z_1, b) := 1$$
$$g_{3,z_2}(z_2) := \mathbb{I}(z_2 \geq 0)$$

- given values for all root nodes:

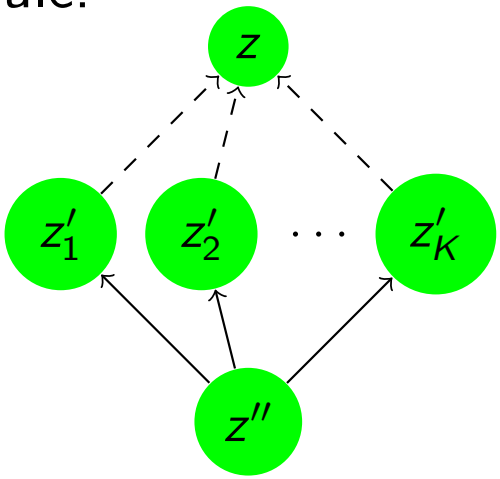
$$x = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, w = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, b = 0.5$$

- compute gradient values for all neighbors:

$$\nabla_{z_2} z_3 = 1, \nabla_{z_1} z_2 = 1, \nabla_{b_1} z_2 = 1,$$
$$\nabla_x z_1 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \nabla_w z_1 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

Gradients in Computational Graphs (2/2)

- for any subgraph $z'' \to z'_1, z'_2, \ldots, z'_K \to z$ by chain rule:



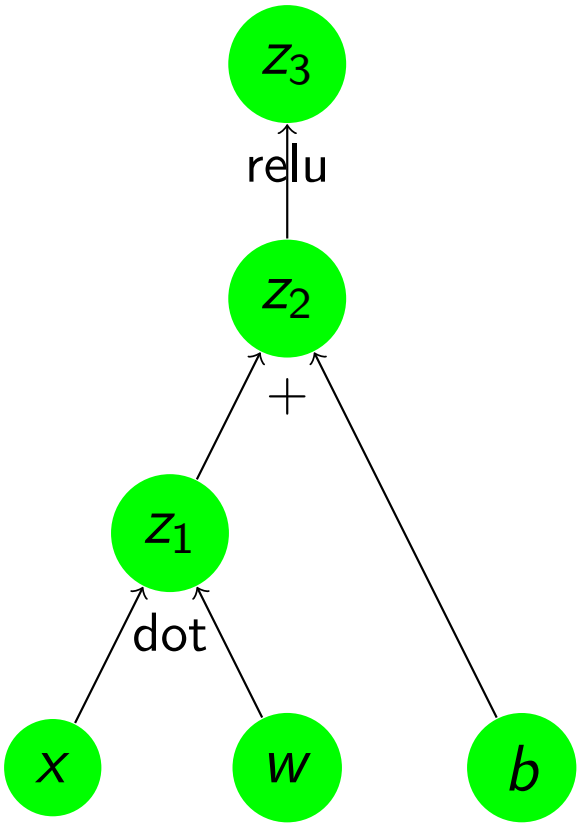
$$\nabla_{z''} z = \sum_{k=1}^{K} \nabla_{z'_k} z \, \nabla_{z''} z'_k = \sum_{k=1}^{K} \frac{\partial z}{\partial z'_k} \frac{\partial z'_k}{\partial z''}$$

$$w_{z,z''} = \sum_{k=1}^{K} w_{z,z'_k} \, g_{z'_k,z''}\left(v_{\text{fanin}(z'_k)}\right)$$

$$= \sum_{k=1}^{K} \sum_{i \in \dim T_{z'_k}} \left(w_{z,z'_k}\right)_{.,i} \, g_{z'_k,z''}\left(v_{\text{fanin}(z'_k)}\right)_{i,.} \in T_z \times T_{z''}$$

- this way, gradients between any two nodes in a computational graph can be computed automatically.

Gradient Computation / Example



$$z_3 = \text{relu}(x^T w + b)$$

- gradient values for all neighbors:

$$\nabla_{z_2} z_3 = 1, \nabla_{z_1} z_2 = 1, \nabla_{b_1} z_2 = 1,$$

$$\nabla_x z_1 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \nabla_w z_1 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

- gradient values for all node pairs:

$$\nabla_{z_1} z_3 = 1, \nabla_b z_3 = 1,$$

A.

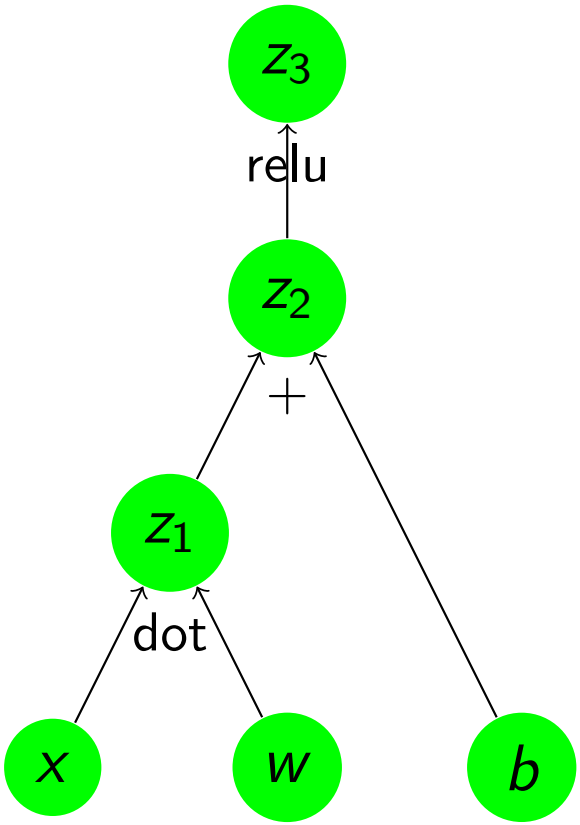
$$\nabla_x z_3 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \nabla_w z_3 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

B.

$$\nabla_x z_3 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \nabla_w z_3 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

$$\nabla_x z_2 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \nabla_w z_2 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

Gradient Computation / Example



$$z_3 = \text{relu}(x^T w + b)$$

- gradient values for all neighbors:

$$\nabla_{z_2} z_3 = 1, \nabla_{z_1} z_2 = 1, \nabla_{b_1} z_2 = 1,$$

$$\nabla_x z_1 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \nabla_w z_1 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

- gradient values for all node pairs:

$$\nabla_{z_1} z_3 = 1, \nabla_b z_3 = 1,$$

$$\nabla_x z_3 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \nabla_w z_3 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

$$\nabla_x z_2 = \begin{pmatrix} 1 \\ -1 \end{pmatrix}, \nabla_w z_2 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

Gradient Computation / Algorithm / Single Leaf

cg-gradient$((Z, E, f, g), (v_z)_{z \in \text{roots}(Z,E)})$:
  $v :=$ cg-forward$((Z, E, f), (v_z)_{z \in \text{roots}(Z,E)})$
  $z :=$ single leaf in (Z, E)
  $w_{z,z} := 1$
  for $z'' \in Z \setminus \{z\}$ in reverse topological order:
    $w_{z,z''} := 0$
    for $z' \in \text{fanout}(z'')$:
      $w_{z',z''} := g_{z',z''}((v_{\tilde{z}''})_{\tilde{z}'' \in \text{fanin}(z')})$
      $w_{z,z''} := w_{z,z''} + w_{z,z'} w_{z',z''}$
  return $(w_{z,z'})_{z' \in \text{roots}(Z,E)}$

- compute gradients $\nabla_{z'} z$ for single leaf node z and all root nodes z'
- take the subgraph on ancestors$(z) \cap$ descendants(Z_{in}) to compute all gradients $\nabla_{z'} z$ for nodes $z' \in Z_{\text{in}}$.

Gradient Computation

- automatic gradient computation in computational graphs combines
  - manually specified gradients of elementary functions,
  - the chain rule, and
  - a useful arrangement of computations
    - forward function computations
    - backwards gradient computations
    - compute each neighbor gradient once (for multiple leaf nodes)
- called **backpropagation** for feedforward neural networks
  - algorithm can be formulated without graph terminology

Summary

- Feedforward neural networks are models for supervised prediction (regression, classification).
- They consist of L stacked layers, each of the form

$$z_\ell(z_{\ell-1}) := \mathrm{a}(W_\ell^T z_{\ell-1})$$

  consisting of
  - a linear combination of the previous layer values with parameters W
  - a nonlinear function (**activation function**).
    - often just the **rectifying linear unit** $\mathrm{relu}(z) := \max\{0, z\}$
- The output layer contains an activation function that **reflects the target type** / output distribution:
  - identity: for continuous targets (with normally distributed uncertainty)
  - logistic function: for a bernoulli probability (binary classification)
  - softmax function: for a multinoulli probability (multi-class classification)

Summary (2/2)

- Any loss can be used, esp. the **negative log-likelihood**.
  - for classification problems: called **cross entropy**
- To learn a feedforward neural network, gradient-descent type algorithms can be used
  (esp. **stochastic gradient descent**).
- Gradients of neural networks
  — and more generally, any computational graph —
  can be computed automatically.
  - **backpropagation**

Further Readings

- Goodfellow et al. 2016, ch. 6
- Zhang et al. 2020, ch. 2.5, 3–5
- lecture Machine Learning, chapter B.2

Acknowledgement: An earlier version of the slides for this lecture have been written by my former postdoc Dr Josif Grabocka.

References

Charu C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. Springer International Publishing, 2018. ISBN 978-3-319-94462-3. doi: 10.1007/978-3-319-94463-0.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The Mit Press, Cambridge, Massachusetts, November 2016. ISBN 978-0-262-03561-3.

Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander Smola. *Dive into Deep Learning*. https://d2l.ai/, 2020.