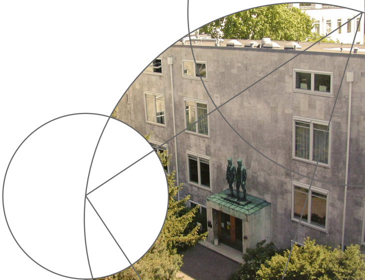
Deep Learning

Part I: Neural Networks

Christian Igel
Department of Computer Science



Outline

1. Neural Networks
2. Loss Functions and Encoding
3. Backpropagation & Gradient-based Learning
4. Regularization

Outline

1. Neural Networks
2. Loss Functions and Encoding
3. Backpropagation & Gradient-based Learning
4. Regularization

Outline

Neuroscience vs. machine learning

Two applications of neural networks:

Computational neuroscience: Modelling biological information processing to gain insights about biological information processing

Machine learning: Deriving learning algorithms (loosely) inspired by neural information processing to solve technical problems better than other methods

Feed-forward artificial neural networks

Different classes of NNs exist:

- feed-forward NNs $\longleftrightarrow$ recurrent networks
- supervised $\longleftrightarrow$ unsupervised learning

We

- concentrate on feed-forward NNs,
- consider regression and classification,
- just consider supervised learning.

That is, we use data to adapt (train) the parameters (weights) of a mathematical model.

Simple neuron models

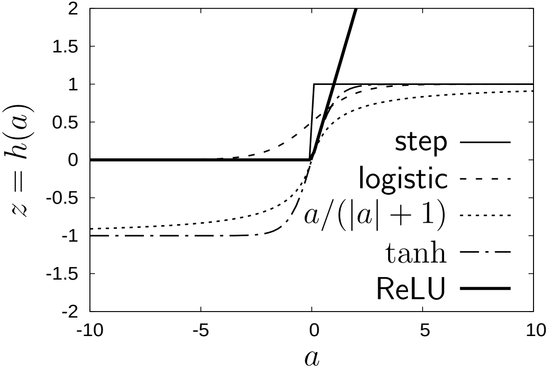
- Let the input be $x_1, \dots, x_d$ collected in the vector $\boldsymbol{x} \in \mathbb{R}^d$.
- Let the output of neuron i be denoted by $z_i(\boldsymbol{x})$. Often we omit writing the dependency on $\boldsymbol{x}$ to keept the notation uncluttered.
- "Integration": Computing weighted sum

$$a_i = \sum_{j=1}^{d} w_{ij} x_j + w_{i0}$$

  with bias (threshold, offset) parameter $w_{i0} \in \mathbb{R}$
- "Firing": Applying transfer function (activation function) h:

$$z_i = h(a_i) = h\left(\sum_{j=1}^{d} w_{ij} x_j + w_{i0}\right)$$

Activation functions



Step / threshold:

$$h(a) = \mathbb{I}\{a > 0\}$$

Fermi / logistic:

$$h(a) = \frac{1}{1 + \exp(-a)}$$

Hyperbolic tangens:

$$h(a) = \tanh(a)$$

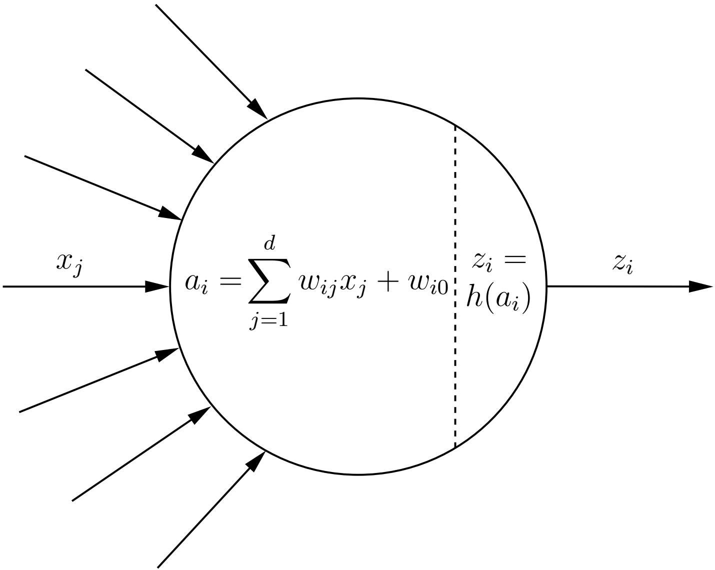
Alternative sigmoid:

$$h(a) = \frac{a}{1 + |a|}$$

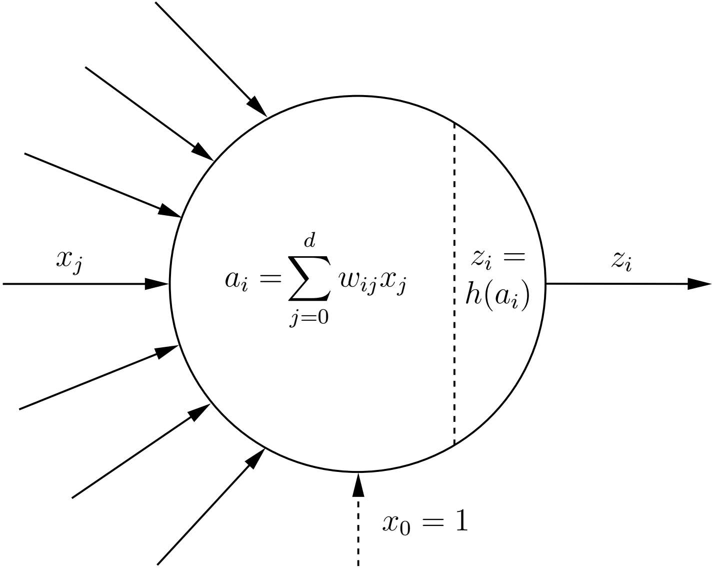
Rectified linear unit:

$$h(a) = \max(0, a)$$

Single neuron with bias

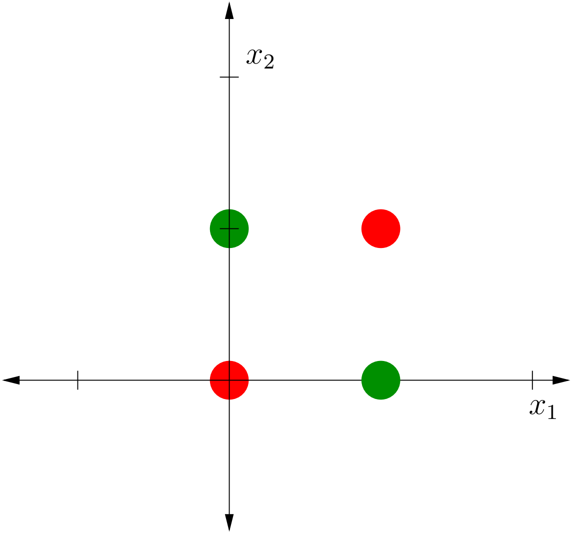


Single neuron with implicit bias



XOR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



Simple neural network models

- Neural network (NN): Set of connected neurons
- NN can be described by a weighted directed graph
  - Neurons are the nodes
  - Connections between neurons are the edges
  - Strength of connection from neuron j to neuron i is described by weight w_{ij}
  - All weights are collected in weight vector $\boldsymbol{w}$
- Neurons are numbered by integers
- Restriction to feed-forward NNs: We do not allow cycles in the connectivity graph
- NN represents mapping

$$f : \mathbb{R}^d \to \mathbb{R}^K$$

parameterized by $\boldsymbol{w}$: $f(\boldsymbol{x}_n; \boldsymbol{w})_i = y_i$

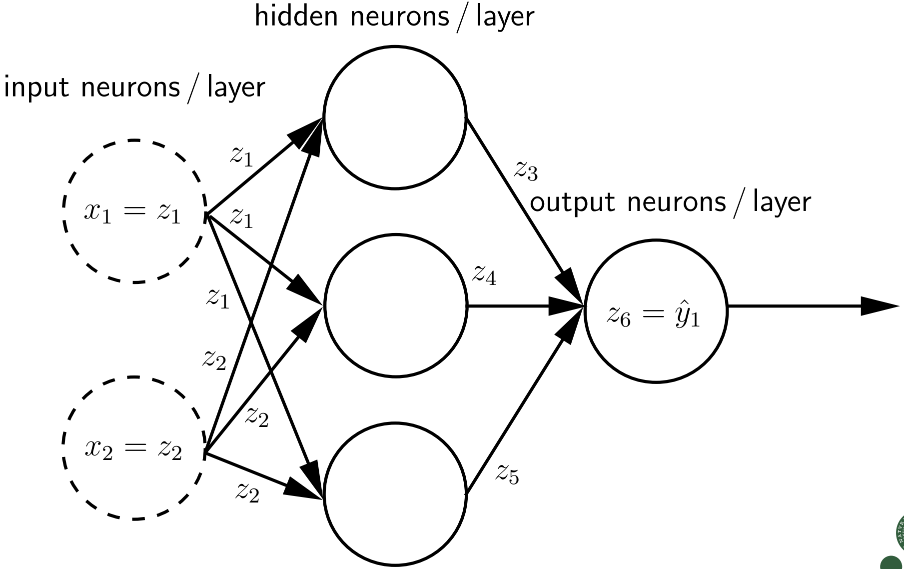
Notation I

- d input neurons, K output neurons, M *hidden* neurons
- Bishop notation: Activation function of hidden neurons is denoted by h, activation function of output neurons is denoted by σ
- Neuron i can get only input from neuron j if $j < i$, this ensures that the graph is acyclic

Notation II

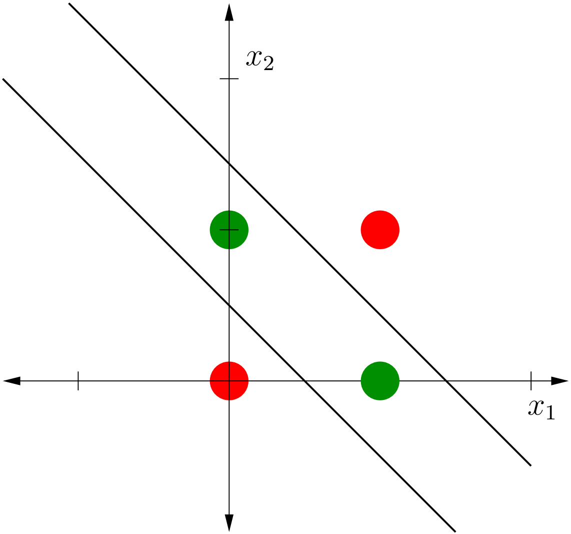
- Output of neuron i is denoted by z_i
- $z_0(\boldsymbol{x}) = 1$ ($w_{i0}z_0$ is the bias parameter of neuron i)
- $z_1(\boldsymbol{x}) = x_1, \dots, z_d(\boldsymbol{x}) = x_d$ (input neurons)
- $z_i(\boldsymbol{x}) = h\left(\sum_{0 \leq j < i} w_{ij} z_j\right)$ for $d < i \leq d + M$
- $z_i(\boldsymbol{x}) = \sigma\left(\sum_{0 \leq j < i} w_{ij} z_j\right)$ for $i > d + M$ (output neurons)
- $\hat{y}_1 = z_{1+M+d}(\boldsymbol{x}), \dots, \hat{y}_K = z_{K+M+d}(\boldsymbol{x})$

Multi-layer perceptron network

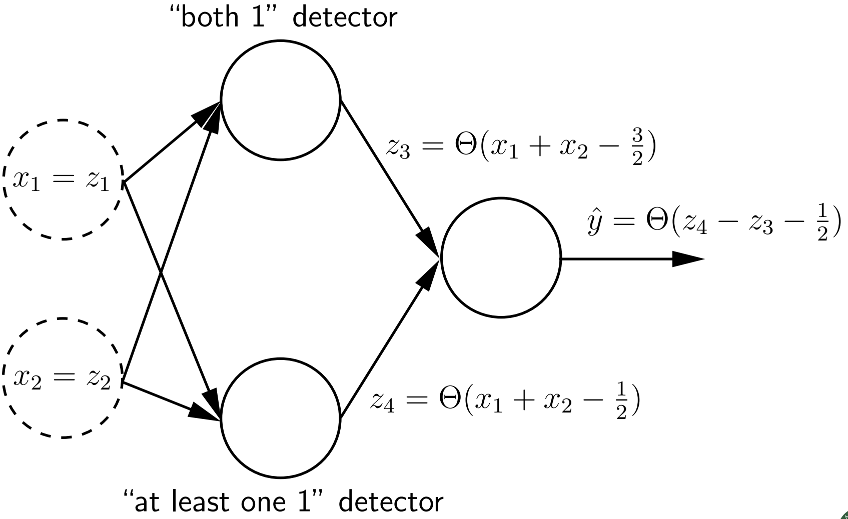


XOR revisited

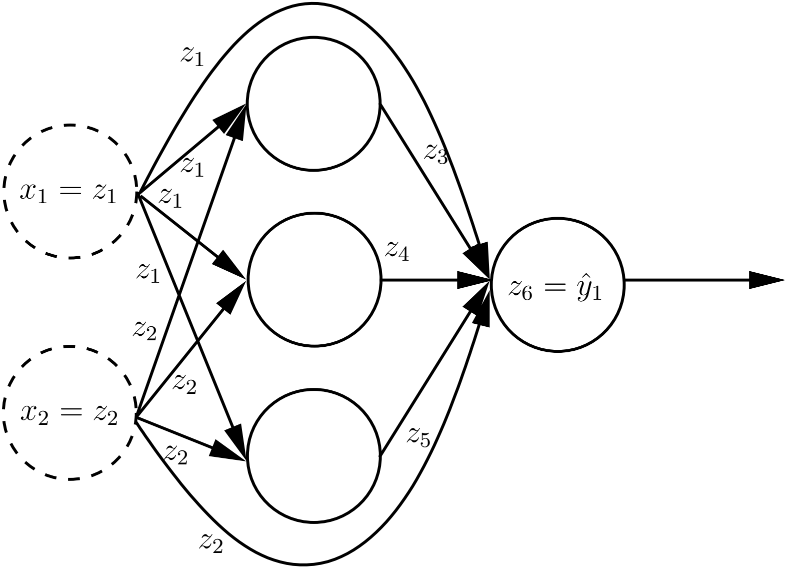
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



Multi-layer perceptron solving XOR



Multi-layer perceptron network with shortcuts



Outline

Regression

- NN shall learn function

$$f : \mathbb{R}^d \to \mathbb{R}^K$$

$\Rightarrow$ d input neurons, K output neurons

- Training data $S = \{(\boldsymbol{x}_1, \boldsymbol{y}_1), \dots, (\boldsymbol{x}_N, \boldsymbol{y}_N)\}$, $\boldsymbol{x}_i \in \mathbb{R}^d$, $\boldsymbol{y}_i \in \mathbb{R}^K$, $1 \leq i \leq N$
- Sum-of-squares error

$$E = \frac{1}{2} \sum_{n=1}^{N} \|f(\boldsymbol{x}_n; \boldsymbol{w}) - \boldsymbol{y}_n\|^2 = \frac{1}{2} \sum_{n=1}^{N} \sum_{i=1}^{K} ([f(\boldsymbol{x}_n; \boldsymbol{w})]_i - [\boldsymbol{y}_n]_i)^2$$

- Usually linear output neurons $\sigma(a) = a$

Sum-of-squares and maximum likelihood

W.l.o.g. $d = 1$, $S = \{(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_N, y_N)\}$. We assume that the observations t given an input $\boldsymbol{x}$ are normally distributed (with variance s^2) around the model $f(\boldsymbol{x}; \boldsymbol{w})$:

$$p(y|\boldsymbol{x}; \boldsymbol{w}) = \frac{1}{s\sqrt{2\pi}} \exp \frac{-(y - f(\boldsymbol{x}; \boldsymbol{w}))^2}{2s^2}$$

Likelihood and negative log-likelihood:

$$p(S|\boldsymbol{w}) = \prod_{n=1}^{N} \frac{1}{s\sqrt{2\pi}} \exp \frac{-(y_n - f(\boldsymbol{x_n}; \boldsymbol{w}))^2}{2s^2}$$

$$-\ln p(S|\boldsymbol{w}) = \frac{1}{2s^2} \sum_{n=1}^{N} (y_n - f(\boldsymbol{x_n}; \boldsymbol{w}))^2 + N \ln(s\sqrt{2\pi})$$

As blue terms are independent of $\boldsymbol{w}$, minimizing the sum-of-squares error corresponds to maximum likelihood estimation under the Gaussian noise assumption.

Binary classification

For binary classification, assume $\mathcal{Y} = \{0, 1\}$, the output is in $[0, 1]$, and the target follows a Bernoulli distribution:

$$p(y|\boldsymbol{x};\boldsymbol{w}) = f(\boldsymbol{x};\boldsymbol{w})^y [1 - f(\boldsymbol{x};\boldsymbol{w})]^{1-y}$$

Negative logarithm of $p(S|\boldsymbol{w}) = \prod_{n=1}^{N} p(y_n|\boldsymbol{x}_n;\boldsymbol{w})$ leads to *cross-entropy* error function:

$$-\ln p(S|\boldsymbol{w}) = -\sum_{n=1}^{N} \{y_n \ln f(\boldsymbol{x}_n;\boldsymbol{w}) + (1-y_n)\ln(1-f(\boldsymbol{x}_n;\boldsymbol{w}))\}$$

Use sigmoid mapping to $[0, 1]$ as output activation function.

Multi-class classification: One-hot

For K classes, use one-hot encoding (1 out of K encoding):

- The jth component of $\boldsymbol{y}_i$ is one, if $\boldsymbol{x}_i$ belongs to the jth class, and zero otherwise.
- Example: If $K = 4$ and $\boldsymbol{x}_i$ belongs to third class, then $\boldsymbol{y}_i = (0, 0, 1, 0)^\mathsf{T}$.

With $\sum_{k=1}^{K} [f(\boldsymbol{x}; \boldsymbol{w})]_k = 1$ and $\forall k : [f(\boldsymbol{x}; \boldsymbol{w})]_k \geq 0$

$$p(\boldsymbol{y}|\boldsymbol{x}; \boldsymbol{w}) = \prod_{k=1}^{K} [f(\boldsymbol{x}; \boldsymbol{w})]_k^{[\boldsymbol{y}]_k}$$

gives negative log likelihood (cross-entropy for multiple classes):

$$-\ln p(S|\boldsymbol{w}) = -\sum_{n=1}^{N} \sum_{k=1}^{K} [\boldsymbol{y}_n]_k \ln[f(\boldsymbol{x}_n; \boldsymbol{w})]_k$$

Multi-class classification: Soft-max

The *soft-max* activation function

$$[f(\boldsymbol{x};\boldsymbol{w})]_j = \sigma(a_{M+d+j}) = \frac{\exp a_{M+d+j}}{\sum_{k=1}^{K} \exp a_{M+d+k}}$$

naturally extends the logistic function to multiple classes and ensures that $\sum_{j=k}^{K}[f(\boldsymbol{x};\boldsymbol{w})]_k = 1$.

Outline

Gradient descent

- Consider learning by iteratively changing the weights

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} + \Delta\boldsymbol{w}^{(t)}$$

- Simplest choice is (steepest) gradient descent

$$\Delta\boldsymbol{w}^{(t)} = -\eta \nabla E|_{\boldsymbol{w}^{(t)}}$$

  with learning rate $\eta > 0$

- Often a *momentum term* is added to improve the performance

$$\Delta\boldsymbol{w}^{(t)} = -\eta \nabla E|_{\boldsymbol{w}^{(t)}} + \mu \Delta\boldsymbol{w}^{(t-1)}$$

  with momentum parameter $\mu \geq 0$

Backpropagation I

Let h and σ be differentiable. From

$$z_i = h(a_i) \qquad a_i = \sum_{j<i} w_{ij} z_j$$

$$E = \sum_{n=1}^{N} E^n \qquad \text{e.g.} \qquad \sum_{n=1}^{N} \underbrace{\frac{1}{2} \|\boldsymbol{y}_n - f(\boldsymbol{x}_n \,|\, \boldsymbol{w})\|^2}_{E^n}$$

we get the partial derivatives:

$$\frac{\partial E}{\partial w_{ij}} = \sum_{n=1}^{N} \frac{\partial E^n}{\partial w_{ij}}$$

In the following, we derive $\frac{\partial E^n}{\partial w_{ij}}$; the index n is omitted to keep the notation uncluttered (i.e., we write E for E^n, $\boldsymbol{x}$ for $\boldsymbol{x}_n$, etc.).

Backpropagation II

We want

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial a_i} \frac{\partial a_i}{\partial w_{ij}}$$

and define:

$$\delta_i := \frac{\partial E}{\partial a_i}$$

With

$$\frac{\partial a_i}{\partial w_{ij}} = z_j$$

we get:

$$\frac{\partial E}{\partial w_{ij}} = \delta_i z_j$$

Backpropagation III

For an output unit $M + d < i \leq d + M + K$ we have:

$$\delta_i = \frac{\partial E}{\partial a_i} = \frac{\partial z_i}{\partial a_i}\frac{\partial E}{\partial z_i} = \sigma'(a_i)\frac{\partial E}{\partial z_i} = \sigma'(a_i)\frac{\partial E}{\partial \hat{y}_{i-M-d}}$$

If $\sigma(a) = a$, i.e., the output is linear and $\sigma'(a) = 1$, and $E = \frac{1}{2}\|\boldsymbol{y} - \hat{\boldsymbol{y}}\|^2$, we get:

$$E = \frac{1}{2}\sum_{i=1}^{K}(\hat{y}_i - y_i)^2 = \frac{1}{2}\sum_{i=M+d+1}^{d+M+K}(\underbrace{\hat{y}_{i-M-d}}_{z_i} - y_{i-M-d})^2$$

$$\delta_i = \frac{\partial}{\partial z_i}\frac{1}{2}\sum_{j=M+d+1}^{d+M+K}(z_j - y_{j-M-d})^2 = \frac{\partial}{\partial z_i}\frac{1}{2}(z_i - y_{i-M-d})^2 \Rightarrow$$

$$\delta_i = z_i - y_{i-M-d}$$

Backpropagation IV

To get the δs for a hidden unit $i \in \{d+1, \ldots, M+d\}$, we need the chain rule again

$$\delta_i = \frac{\partial E}{\partial a_i} = \sum_{k=i+1}^{M+d+K} \frac{\partial E}{\partial a_k} \frac{\partial a_k}{\partial a_i} = \sum_{k=i+1}^{M+d+K} \frac{\partial E}{\partial a_k} \frac{\partial a_k}{\partial z_i} \frac{\partial z_i}{\partial a_i}$$

and obtain:

$$\delta_i = h'(a_i) \sum_{k=i+1}^{M+d+K} w_{ki} \delta_k$$

Backpropagation V

For each training pattern $(\boldsymbol{x}, \boldsymbol{y})$:

- *Forward pass (determines output of network given $\boldsymbol{x}$):*
  1. Compute $z_{d+1}, \dots, z_{d+M+K}$ in sequential order
  2. $z_{M-K+1}, \dots, z_M$ define $\hat{\boldsymbol{y}} = f(\boldsymbol{x} \mid \boldsymbol{w})$
- *Backward pass (determines partial derivatives):*
  1. After a forward pass, compute $\delta_d, \dots, \delta_{d+M+K}$ in reverse order
  2. Compute the partial derivatives according to $\partial E / \partial w_{ij} = \delta_i z_j$

Other error functions

For an output unit $M + d < i \leq d + M + K$ we get

$$\delta_i = z_i - y_{i-M-d}$$

for

- Sum-of-squares error and linear output neurons
- Cross-entropy error and single logistic output neuron
- Cross-entropy error for multiple classes and soft-max output

Online vs. batch learning iteration

Batch learning: Compute the gradients over all N training samples and update

$$\Delta \boldsymbol{w}^{(t)} = -\eta \nabla E|_{\boldsymbol{w}^{(t)}}$$

Online learning: Choose a pattern $(\boldsymbol{x}_n, \boldsymbol{y}_n)$, $1 \leq n \leq N$, (e.g., randomly) and update

$$\Delta \boldsymbol{w}^{(t)} = -\eta \nabla E^n|_{\boldsymbol{w}^{(t)}}$$

with a smaller learning rate η

Mini-batch learning: Choose a subset $S_m = \{(\boldsymbol{x}_{i_1}, \boldsymbol{y}_{i_1}), \dots, (\boldsymbol{x}_{i_B}, \boldsymbol{y}_{i_B})\}$, $1 \leq i_1 \leq \dots \leq i_B \leq N$, and update

$$\Delta \boldsymbol{w}^{(t)} = -\eta \sum_{(\boldsymbol{x}_n, \boldsymbol{y}_n) \in S_B} \nabla E^n|_{\boldsymbol{w}^{(t)}}$$

"Vanishing gradient"

- Derivative of $h(a) = \frac{1}{1+\exp(-a)}$ is upper bounded by 0.25.
- What is the derivative of the rectified linear unit (ReLU) $h(a) = \max(0, a)$ for $a < 0$ and $a > 0$, respectively?
- Consider a deep neural network with many layers and

$$\delta_i = h'(a_i) \sum_{k=i+1}^{M+d+K} w_{ki}\delta_k \quad .$$

What happens to the magnitude of δ_i with increasing number of layers between neuron i and the output?

Efficient gradient-based optimization

- Vanilla steepest-descent is usually not the best choice for (batch) gradient-based learning
- Many powerful gradient-based search techniques exist
- Recent method for online/min-batch learning: Adam (from "adaptive moments")

Adam algorithm

Algorithm 1: Adam algorithm

init. $\boldsymbol{w}^{(0)}, \beta_1, \beta_2, \alpha, \epsilon$; $t \leftarrow 1$; $\boldsymbol{v}^{(t)}, \hat{\boldsymbol{v}}^{(t)}, \boldsymbol{m}^{(t)}, \hat{\boldsymbol{m}}^{(t)} \leftarrow \mathbf{0}$

while *stopping criterion not met* **do**

    foreach w_{ij} **do**

        $g_{ij}^{(t)} \leftarrow \partial f(\boldsymbol{w}^{(t)}) / \partial w_{ij}^{(t)}$

        $m_{ij}^{(t+1)} \leftarrow \beta_1 \cdot m_{ij}^{(t)} + (1 - \beta_1) \cdot g_{ij}^{(t)}$

        $v_{ij}^{(t+1)} \leftarrow \beta_2 \cdot v_{ij}^{(t)} + (1 - \beta_2) \cdot \left(g_{ij}^{(t)}\right)^2$

        $\hat{m}_{ij}^{(t+1)} \leftarrow m_{ij}^{(t+1)} / (1 - \beta_1^t)$

        $\hat{v}_{ij}^{(t+1)} \leftarrow v_{ij}^{(t+1)} / (1 - \beta_2^t)$

        $w_{ij}^{(t+1)} \leftarrow w_{ij}^{(t)} - \alpha \cdot \hat{m}_{ij}^{(t+1)} / \left(\sqrt{\hat{v}_{ij}^{(t+1)}} + \epsilon\right)$

    $t \leftarrow t + 1$

^{t}: power of t; $^{(t)}$: iteration step t

Adam default values

parameter	range	default	comment
β_1	$[0,1[$	0.9	first moment learning rate
β_2	$[0,1[$	0.999	second raw moment learning rate
ϵ	$\mathbb{R}^+$	10^{-8}	avoid devision by zero
α	$\mathbb{R}^+$	0.001	learning rate
			upper bound on update

- β_1 controls learning the gradient's geometric mean
- β_2 controls learning the gradient components' second raw moments
- $(1-\beta_1^t)$ and $(1-\beta_2^t)$ compensate for initialization bias
- ϵ avoids devision by zero (let's ignore it in the following)

Kingma, Lei Ba. Adam: A method for Stochastic Optimization. *ICLR*, 2015

Adam: Effects

- Update by $\alpha \cdot \hat{m}_{ij}^{(t+1)} / \sqrt{\hat{v}_{ij}^{(t+1)}}$
- Note $\sqrt{\mathbb{E}\{(g_{ij}^{(t)})^2\}} \geq \sqrt{\mathbb{E}\{g_{ij}^{(t)}\}^2} \geq \mathbb{E}\{g_{ij}^{(t)}\}$
- α plays the role of an upper bound on the steps (can be increased by $(1-\beta_1)/\sqrt{1-\beta_2}$)
- If for all t the $g_{ij}^{(t)}$ have the same sign (i.e., the steps for that weight go in the same direction), $\sqrt{\mathbb{E}\{g_{ij}^{(t)}\}^2} = |\mathbb{E}\{g_{ij}^{(t)}\}|$
- If for a weight the $g_{ij}^{(t)}$ often change sign, $\hat{m}_{ij}^{(t+1)} / \sqrt{\hat{v}_{ij}^{(t+1)}}$ gets small

Adam: Initialization bias correction

- We have:

$$v_{ij}^{(t+1)} = (1-\beta_2)\sum_{i=1}^{t}\beta_2^{t-i}\left(g_{ij}^{(i)}\right)^2 = (1-\beta_2)\sum_{i=0}^{t-1}\beta_2^{t-i-1}\left(g_{ij}^{(i+1)}\right)^2$$

Assume $\mathbb{E}\{\left(g_{ij}^{(t)}\right)^2\}$ to be stationary and recall from geometric series that $\sum_{i=0}^{t-1}\alpha^i = (1-\alpha^t)/(1-\alpha)$:

$$\begin{aligned}
\mathbb{E}\{v_{ij}^{(t+1)}\} &= \mathbb{E}\left\{(1-\beta_2)\sum_{i=0}^{t-1}\beta_2^{t-i-1}\left(g_{ij}^{(i+1)}\right)^2\right\}\\
&= \mathbb{E}\{\left(g_{ij}\right)^2\}(1-\beta_2)\sum_{i=0}^{t-1}\beta_2^{t-i-1}\\
&= \mathbb{E}\{\left(g_{ij}\right)^2\}(1-\beta_2)\sum_{i=0}^{t-1}\beta_2^{i} = \mathbb{E}\{\left(g_{ij}\right)^2\}(1-\beta_2^t)
\end{aligned}$$

Outline

Weight-decay

- The smaller the weights, the "more linear" is the neural network function.
- Thus, small $\|\boldsymbol{w}\|$ corresponds to smooth functions.
- Therefore, one can penalize large weights by optimizing

$$E + \gamma \frac{1}{2} \|\boldsymbol{w}\|^2$$

  with regularization hyperparameter $\gamma \geq 0$.
- Note: the weights of linear output neurons should not be considered when computing the norm of the weight vector.

Early stopping

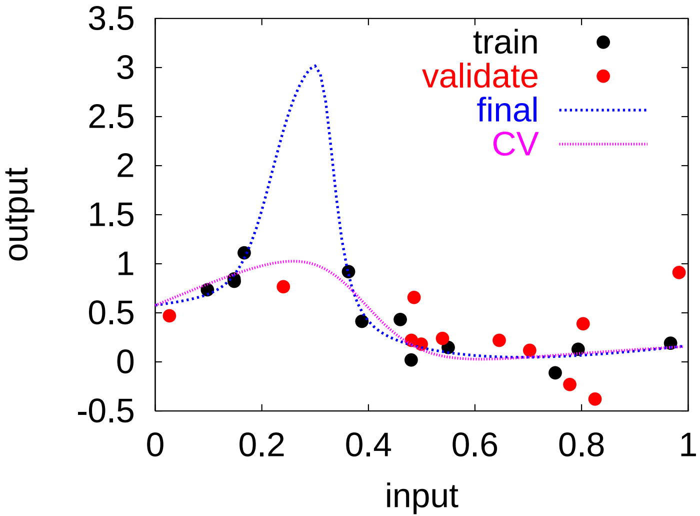
Early-stopping: the learning algorithm

- partitions sample S into training S_{train} and validation S_{val} data
- produces iteratively a sequence of hypotheses

$$h_1, h_2, h_3, \dots$$

  based on S_{train}
- monitors empirical risk $\mathcal{R}_{S_{\text{val}}}(h_i)$ on the validation data
- outputs the hypothesis h_i minimizing $\mathcal{R}_{S_{\text{val}}}(h_i)$.

Early stopping example



Neural network architecture

- Magnitude of the weights is more important for the complexity of a layer than number of neurons.
- Depth of network in general increases complexity.
- Training “deep” NNs implementing hierarchical processing is currently an active research field.

The secrets of successful shallow NN training

- Normalize the data component-wise to zero-mean and unit variance (using PCA for removing linear correlations helps)
- Use a single layer with “enough neurons”
- Start with small weights
- Employ early stopping
- Try shortcuts