



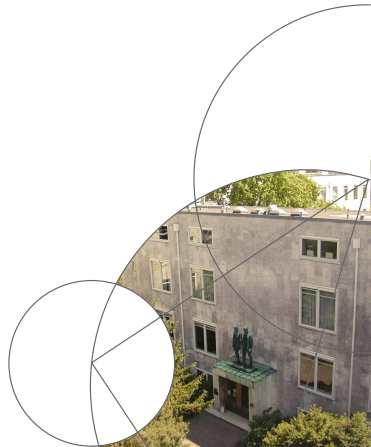
Faculty of Science



Deep Learning

Part I: Neural Networks

Christian Igel
Department of Computer Science



Warm-up: Chain rule

The *chain rule* for computing the derivative of a composition of two functions,

$$\frac{\partial f(g(x))}{\partial x} = f'(g(x))g'(x)$$

with $f'(x) = \frac{\partial f(x)}{\partial x}$ and $g'(x) = \frac{\partial g(x)}{\partial x}$, can be extended to:

$$\frac{\partial f(g_1(x), g_2(x), \dots, g_n(x))}{\partial x} = \sum_{i=1}^n \frac{\partial f(g_1(x), \dots, g_n(x))}{\partial g_i(x)} \frac{\partial g_i(x)}{\partial x}$$



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



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 start with

- feed-forward NNs,
- supervised learning.

Neural network learning: Using 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 $\mathbf{x} \in \mathbb{R}^d$.
- Let the output of neuron i be denoted by $z_i(\mathbf{x})$. Often we omit writing the dependency on \mathbf{x} to keep the notation uncluttered.
- “Integration”: Computing weighted sum

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

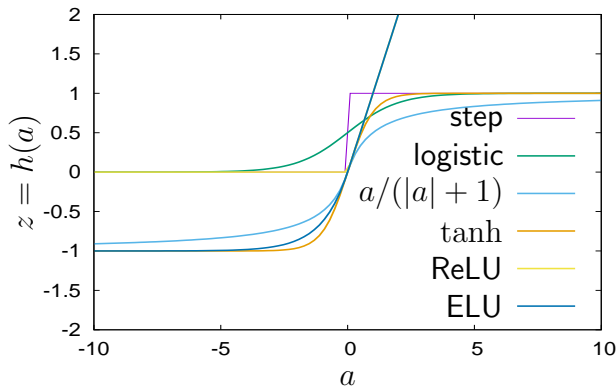
with bias (threshold, offset) parameter $w_{i0} \in \mathbb{R}$

- “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:

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

Hyperbolic tangents:

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

Alternative sigmoid:

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

Rectified linear unit (ReLU):

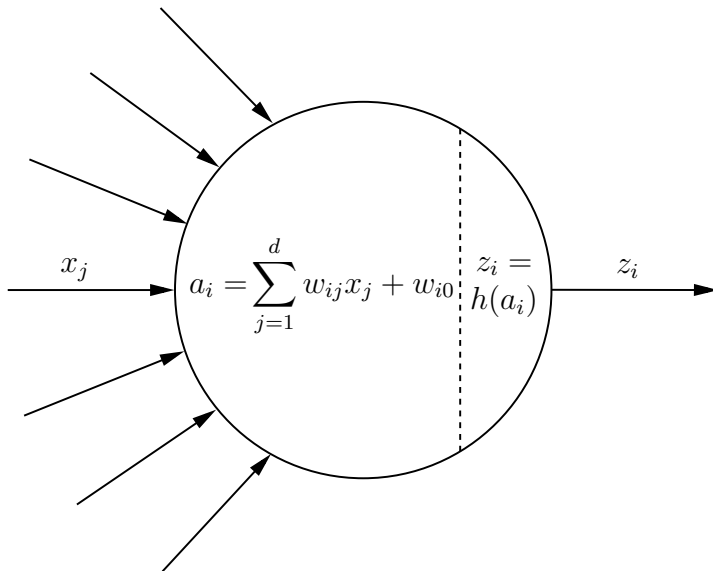
$$h(a) = \max(0, a)$$

Exponential LU (ELU, $\alpha > 0$):

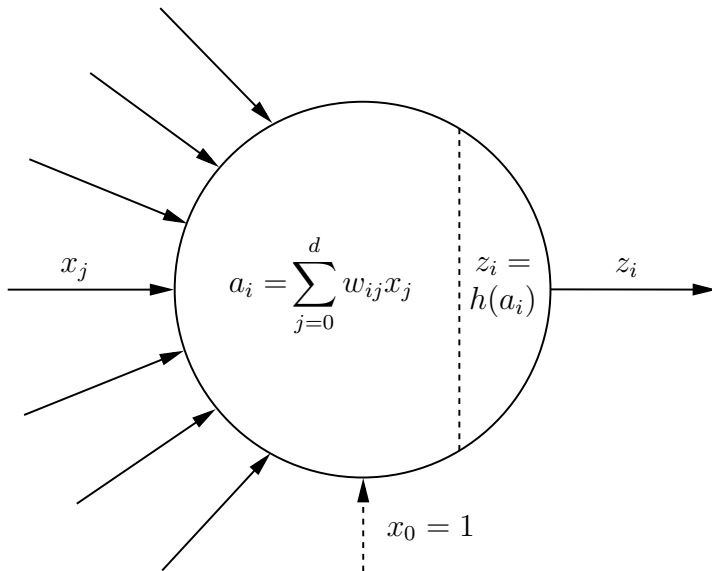
$$h(a) = \begin{cases} a & a \geq 0 \\ \alpha(e^a - 1) & a < 0 \end{cases}$$



Single neuron with bias

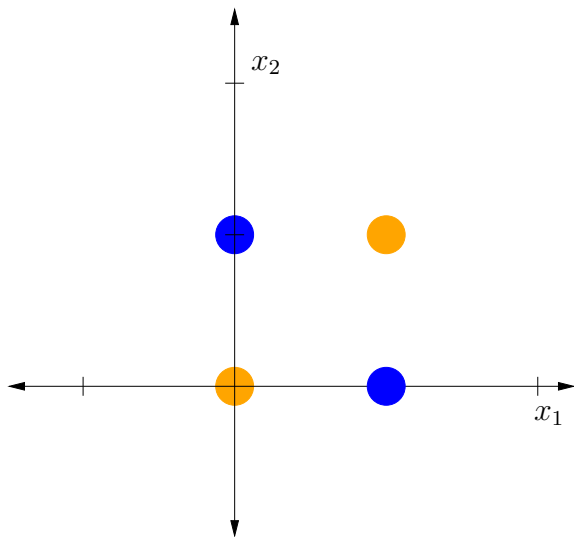


Single neuron with implicit bias



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/vertices and numbered by integers
 $V = \{0, 1, 2, \dots\}$
 - Connections between neurons are the edges A
 - Strength of connection $(j, i) \in A$ from neuron j to neuron i is described by weight w_{ij}
 - All weights are collected in weight vector \mathbf{w}
- Restriction to feed-forward NNs: We do not allow cycles in the connectivity graph
- NN represents mapping

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

parameterized by \mathbf{w} : $f(\mathbf{x}; \mathbf{w})_i = \hat{y}_i$



Notation I

- d input neurons, K output neurons, M hidden neurons:

$$V_{\text{input}} = \{1, \dots, d\}$$

$$V_{\text{hidden}} = \{d + 1, \dots, d + M\}$$

$$V_{\text{output}} = \{d + M + 1, \dots, d + M + K\}$$

That is:

$$V = \{0\} \cup V_{\text{input}} \cup V_{\text{hidden}} \cup V_{\text{output}} = \{0, 1, 2, \dots, d + M + K\}$$

- Activation function of neuron i is denoted by h_i
 - $h_i(a) = a$ for $i \in \{0\} \cup V_{\text{input}}$
 - Typically $h_i \neq h_j$ for $i \in V_{\text{hidden}}$ and $j \in V_{\text{output}}$
- Neuron i can get only input from neuron j if $j < i$, this ensures that the graph is acyclic



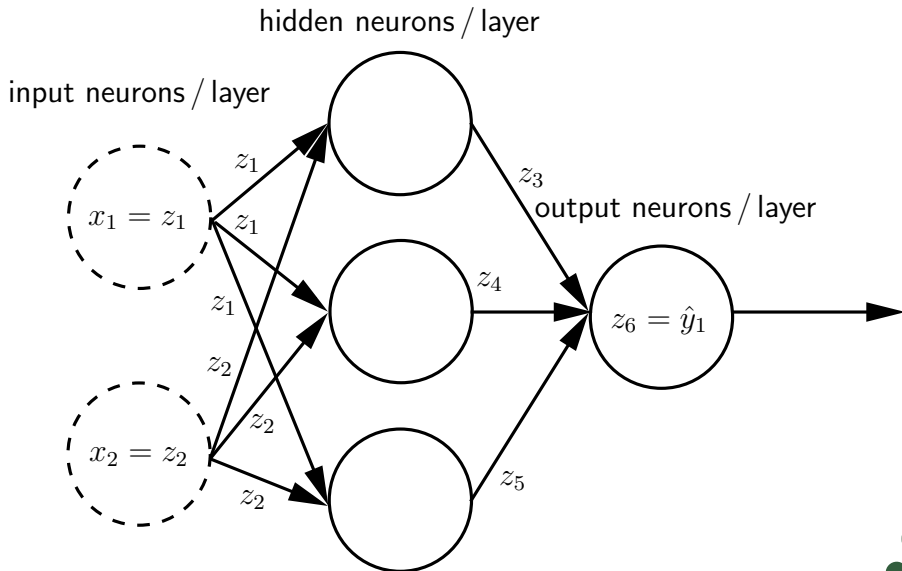
Notation II

- Activation of neuron $i > d$ is $a_i = \sum_{(j,i) \in A} w_{ij} z_j$
- Output of neuron i is denoted by z_i
- $z_i(a_i) = h_i(a_i)$
 - $z_0(\cdot) = 1$ ($w_{i0} z_0$ is the bias parameter of neuron i)
 - $z_i(\cdot) = x_i$ for $i \in V_{\text{input}} = \{1, \dots, d\}$

- Output of the network: $f(\mathbf{x}; \mathbf{w}) = \begin{pmatrix} z_{M+d+1} \\ z_{M+d+2} \\ \vdots \\ z_{M+d+K} \end{pmatrix} = \begin{pmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_K \end{pmatrix}$

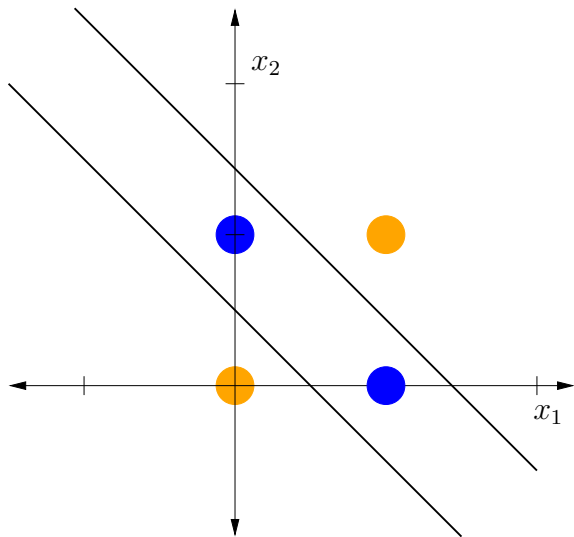


Multi-layer perceptron network



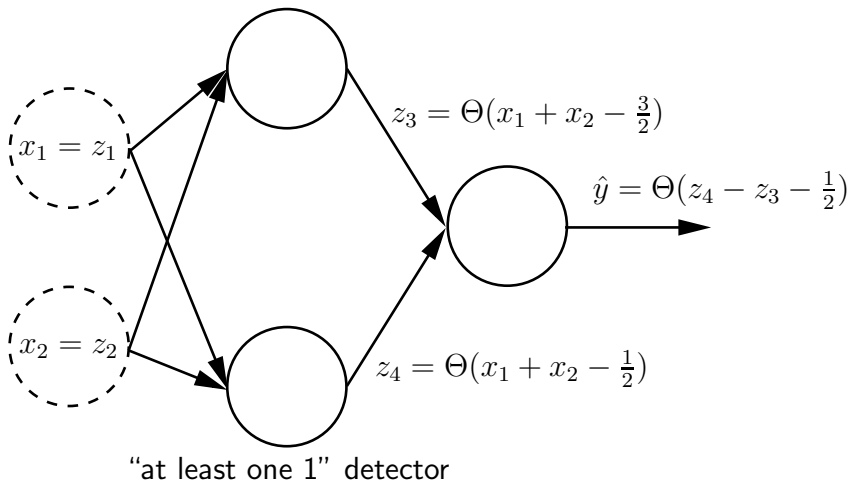
XOR revisited

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

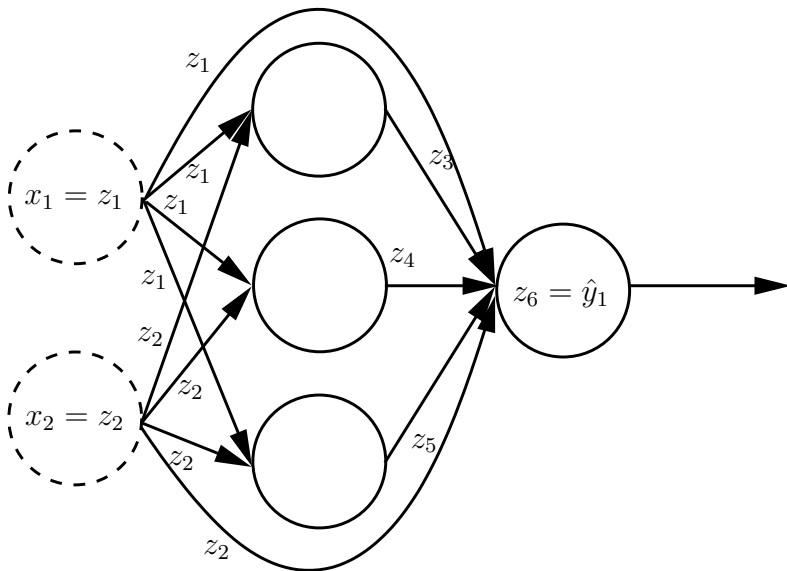


Multi-layer perceptron solving XOR

“both 1” detector



Multi-layer perceptron network with shortcuts



Outline

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



Regression

- NN shall learn function

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

$\Rightarrow d$ input neurons, K output neurons

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

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

- Usually linear output neurons: $h_i(a) = a$ for $i \in V_{\text{output}}$



Sum-of-squares and maximum likelihood

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

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

Likelihood and negative log-likelihood:

$$p(S|\mathbf{w}) = \prod_{n=1}^N \frac{1}{s\sqrt{2\pi}} \exp \frac{-(y_n - f(\mathbf{x}_n; \mathbf{w}))^2}{2s^2}$$
$$-\ln p(S|\mathbf{w}) = \frac{1}{2s^2} \sum_{n=1}^N (y_n - f(\mathbf{x}_n; \mathbf{w}))^2 + N \ln(s\sqrt{2\pi})$$

As **blue terms** are independent of \mathbf{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:

$$p(y|\mathbf{x}; \mathbf{w}) = f(\mathbf{x}; \mathbf{w})^y [1 - f(\mathbf{x}; \mathbf{w})]^{1-y} = \begin{cases} f(\mathbf{x}) & \text{for } y = 1 \\ 1 - f(\mathbf{x}) & \text{for } y = 0 \end{cases}$$

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

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

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 j th component of \mathbf{y}_i is one, if \mathbf{x}_i belongs to the j th class, and zero otherwise.
- Example: If $K = 4$ and \mathbf{x}_i belongs to third class, then $\mathbf{y}_i = (0, 0, 1, 0)^\top$.

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

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

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

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



Multi-class classification: Soft-max

The *soft-max* activation function

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

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



Outline

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



Gradient descent

- Consider learning by iteratively changing the weights

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

- Simplest choice is (steepest) gradient descent

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

with learning rate $\eta > 0$

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

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

with momentum parameter $\mu \geq 0$



Backpropagation I

Let the activation functions be differentiable. From

$$z_i = h(a_i) \quad a_i = \sum_{(j,i) \in A} w_{ij} z_j$$

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

we get for $(j, i) \in A$ 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 , \mathbf{x} for \mathbf{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** $i \in V_{\text{output}} = \{M + d + 1, \dots, M + d + K\}$ we have:

$$\delta_i = \frac{\partial E}{\partial a_i} = \frac{\partial z_i}{\partial a_i} \frac{\partial E}{\partial z_i} = h'_i(a_i) \frac{\partial E}{\partial z_i}$$

If $h_i(a) = a$, i.e., the output is linear and $h'_i(a) = 1$, and $E = \frac{1}{2} \|f(\mathbf{x}; \mathbf{w}) - \mathbf{y}\|^2$, we get:

$$E = \frac{1}{2} \sum_{i=1}^K ([f(\mathbf{x}; \mathbf{w})]_i - y_i)^2 = \frac{1}{2} \sum_{i \in V_{\text{output}}} (z_i - y_{i-M-d})^2$$

$$\delta_i = \frac{\partial}{\partial z_i} \frac{1}{2} \sum_{j \in V_{\text{output}}} (z_j - y_{j-M-d})^2 = \frac{\partial}{\partial z_i} \frac{1}{2} (z_i - y_{i-M-d})^2 \Rightarrow$$

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



Backpropagation IV

For the δ of a **hidden unit** $i \in V_{\text{hidden}} = \{d + 1, \dots, M + d\}$, we need the chain rule again

$$\delta_i = \frac{\partial E}{\partial a_i} = \sum_{k \text{ with } (i,k) \in A} \frac{\partial E}{\partial a_k} \frac{\partial a_k}{\partial a_i} = \sum_{k \text{ with } (i,k) \in A} \frac{\partial E}{\partial a_k} \frac{\partial a_k}{\partial z_i} \frac{\partial z_i}{\partial a_i}$$

and obtain:

$$\delta_i = h'_i(a_i) \sum_{k \text{ with } (i,k) \in A} w_{ki} \delta_k$$



Backpropagation V

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

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



Other error functions

For an output unit $i \in V_{\text{output}} = \{d + M + 1, \dots, d + M + K\}$ 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



Linear algebra notation I (and a special layer definition)

- Layers collect neurons getting the same input and perform the same type of computation
 - Layer l gets only input from layers $l' < l$
 - The activation function of layer l is $h^{(l)}$
 - Inputs (external inputs as well as outputs from other neurons) to layer l are gathered in vector $\mathbf{x}^{(l)}$, the outputs of the neurons in layer l in vector $\mathbf{h}^{(l)}$
- If the outputs of layer l are the only inputs to layer $l + 1$, we have $\mathbf{h}^{(l)} = \mathbf{x}^{(l+1)}$



Linear algebra notation II

- Denoting the weights of the i th neuron in layer l by $\mathbf{w}_i^{(l)}$ and its bias by $b_i^{(l)}$ we get $h_i^{(l)}(\mathbf{x}) = h^{(l)} \left(\left[\mathbf{w}_i^{(l)} \right]^\top \mathbf{x} + b_i^{(l)} \right)$
- We stack the $\mathbf{w}_i^{(l)}$ of all neurons in layer l in matrix $\mathbf{W}^{(l)}$ and their bias parameters in vector $\mathbf{b}^{(l)}$ and write

$$h^{(l)}(\mathbf{x}^{(l)}) = h^{(l)}(\mathbf{W}^{(l)}\mathbf{x}^{(l)} + \mathbf{b}^{(l)}) \quad [\text{great for GPU}]$$

where $h^{(l)}$ acts component-wise

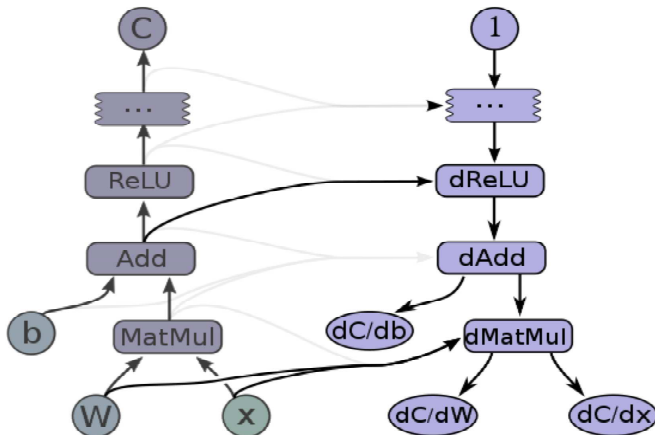
- Assume implicit bias parameters and a batch of inputs $\mathbf{x}_1^{(l)}, \dots, \mathbf{x}_B^{(l)}$ gathered as rows in a data matrix $\mathbf{X}^{(l)}$, the output of the layer can be computed for the full batch as:

$$\mathbf{H}^{(l)}(\mathbf{X}^{(l)}) = h^{(l)}(\mathbf{W}^{(l)}[\mathbf{X}^{(l)}]^\top) \quad [\text{great for GPU}]$$



Symbolic differentiation

Many modern machine learning frameworks can compute gradients automatically:



Abadi et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv:1603.04467*, 2016



Mini-batch learning

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

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

Mini-batch learning: Choose a subset

$$S_m = \{(\mathbf{x}_{i_1}, \mathbf{y}_{i_1}), \dots, (\mathbf{x}_{i_B}, \mathbf{y}_{i_B})\},$$
$$1 \leq i_1 \leq \dots \leq i_B \leq N, \text{ and update}$$

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

typically with a smaller learning rate η



“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

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

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



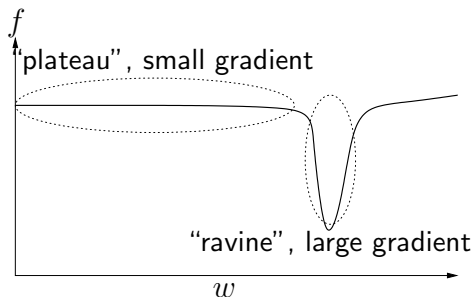
Efficient gradient-based optimization

- 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/mini-batch learning: Adam (from “adaptive moments”)

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



Basic problem



- The idea is – instead of directly using the magnitude of the partial derivative – to adjust the update in the direction of the partial derivative for each weight individually based on feedback from previous iterations.



Rescaling by second raw moment I

- **Goal:** Iterative minimization of $f(\mathbf{w})$
- Taylor approximation:

$$f(\mathbf{w} + \Delta\mathbf{w}) \approx f(\mathbf{w}) + \Delta\mathbf{w}^\top \nabla f(\mathbf{w})$$

leads to update step $\mathbf{w}_i^{(t+1)} \leftarrow \mathbf{w}_i^{(t)} + \Delta\mathbf{w}^{(t)}$ in iteration t

$$\Delta\mathbf{w}^{(t)} = \operatorname{argmin}_{\Delta\mathbf{w}} \{ \Delta\mathbf{w}^\top \nabla f(\mathbf{w}^{(t)}) \quad \text{s.t.} \quad \|\Delta\mathbf{w}^{(t)}\| \leq \alpha \} \quad ,$$

where α defines a trust region (in which we trust the approximation)

- Solution is co-linear with gradient and has maximum length:

$$\Delta\mathbf{w}^{(t)} = -\alpha \frac{\nabla f(\mathbf{w}^{(t)})}{\|\nabla f(\mathbf{w}^{(t)})\|}$$



Rescaling by second raw moment II

- We have unbiased estimates $\nabla \tilde{f}(\mathbf{w}^{(t)})$ instead of $\nabla f(\mathbf{w}^{(t)})$, typically gradients computed on mini-batches:

$$\Delta \mathbf{w} = -\alpha \frac{\mathbb{E}[\nabla \tilde{f}(\mathbf{w}^{(t)})]}{\|\mathbb{E}[\nabla \tilde{f}(\mathbf{w}^{(t)})]\|} = -\alpha \frac{\mathbb{E}[\nabla \tilde{f}(\mathbf{w}^{(t)})]}{\sqrt{\mathbb{E}[\nabla \tilde{f}(\mathbf{w}^{(t)})]^\top \mathbb{E}[\nabla \tilde{f}(\mathbf{w}^{(t)})]}}$$

- Concretele, for neural networks $\nabla_i \tilde{f}(\mathbf{w}^{(t)})$ is the partial derivative of the error function estimate (the error over a mini-batch) w.r.t. weight w_i at iteration t



Rescaling by second raw moment III

Now we do the following . . .

- Replace $\mathbb{E}[\nabla \tilde{f}(\mathbf{w})^\top] \mathbb{E}[\nabla \tilde{f}(\mathbf{w})]$ by $\mathbb{E}[(\nabla \tilde{f}(\mathbf{w}))^2]$. We have

$$\mathbb{E}[(\nabla \tilde{f}(\mathbf{w}))^2] = \mathbb{E}[\nabla \tilde{f}(\mathbf{w})^\top \nabla \tilde{f}(\mathbf{w})] \geq \mathbb{E}[\nabla \tilde{f}(\mathbf{w})^\top] \mathbb{E}[\nabla \tilde{f}(\mathbf{w})] ,$$

which might increase numerical stability.

- Add a positive constant ϵ to $\mathbb{E}[\nabla \tilde{f}(\mathbf{w})^2]$ to avoid division by zero.
- Consider each weight w_i independently.
- Use a running average ($\beta \in]0, 1]$)

$$v_i^{(t+1)} = (1 - \beta) v_i^{(t)} + \beta \left(\nabla_i \tilde{f}(\mathbf{w}^{(t)}) \right)^2$$

as estimate for $\mathbb{E}[\nabla \tilde{f}(\mathbf{w})^2]$.



RMSprop

... then we arrive at the RMSprop update rule:

$$w_i^{(t+1)} \leftarrow w_i^{(t)} - \alpha \frac{\nabla_i \tilde{f}(\mathbf{w}^{(t)})}{\sqrt{v_i^{(t+1)} + \epsilon}}$$

Tieleman, Hinton. Lecture 6.5. RMSPROP: Divide the gradient by a running average of its recent magnitude.
COURSERA: Neural Networks for Machine Learning, 2012

Lafond, Vasilache, Bottou. Diagonal Rescaling For Neural Networks. *arXiv:1705.09319*



Adam algorithm

Algorithm 1: Adam algorithm

```
1 init.  $\mathbf{w}^{(0)}, \beta_1, \beta_2, \alpha, \epsilon; t \leftarrow 1; \mathbf{v}^{(0)}, \hat{\mathbf{v}}^{(0)}, \mathbf{m}^{(0)}, \hat{\mathbf{m}}^{(0)} \leftarrow \mathbf{0}$ 
2 while stopping criterion not met do
3   foreach  $w_i$  do
4      $g_i^{(t)} \leftarrow \nabla_i \tilde{f}(\mathbf{w}^{(t)})$ 
5      $m_i^{(t+1)} \leftarrow \beta_1 \cdot m_i^{(t)} + (1 - \beta_1) \cdot g_i^{(t)}$ 
6      $v_i^{(t+1)} \leftarrow \beta_2 \cdot v_i^{(t)} + (1 - \beta_2) \cdot (g_i^{(t)})^2$ 
7      $\hat{m}_i^{(t+1)} \leftarrow m_i^{(t+1)} / (1 - \beta_1^t)$ 
8      $\hat{v}_i^{(t+1)} \leftarrow v_i^{(t+1)} / (1 - \beta_2^t)$ 
9      $w_i^{(t+1)} \leftarrow w_i^{(t)} - \alpha \cdot \hat{m}_i^{(t+1)} / (\sqrt{\hat{v}_i^{(t+1)}} + \epsilon)$ 
10   $t \leftarrow t + 1$ 
```

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 division by zero
α	\mathbb{R}^+	0.001	learning rate upper bound on update

- β_1 controls learning approximation of gradient's mean
- β_2 controls learning approximation of gradient components' 2nd raw moments
- $(1 - \beta_1^t)$ and $(1 - \beta_2^t)$ compensate for initialization bias
- ϵ avoids division by zero (let's ignore it in the following)



Adam: Observations I

- $m_i^{(t+1)}$ and $v_i^{(t+1)}$ are exponential moving averages
- Assume that the mini-batch gradients $\mathbf{g}^{(t)}$ ($\mathbf{g}^{(t)}$ collects all $g_i^{(t)}$) are drawn from distributions $p^{(t)}$, i.e., $\mathbf{g}^{(t)} \sim p^{(t)}$
- Assume $g_i^{(t)}$ stationary ($p^{(t)} = p$, $t = 1, \dots$):
 - $\mathbb{E}\{\hat{m}_i^{(t+1)}\} = \mathbb{E}\{g_i\}$
 - $\mathbb{E}\{\hat{v}_i^{(t+1)}\} = \mathbb{E}\{g_i^2\}$
(see slide "Adam: Initialization bias correction")
- Under the stationary assumption:

$$\sqrt{\mathbb{E}\{g_i^2\}} = \sqrt{\mathbb{E}\{|g_i|^2\}} \geq \sqrt{\mathbb{E}\{|g_i|\}^2} = \mathbb{E}\{|g_i|\} \geq |\mathbb{E}\{g_i\}|$$



Adam: Observations II

- Assuming $g_i^{(t)}$ stationary, $(1 - \beta_1) = \sqrt{1 - \beta_2}$, and $\epsilon = 0$, we have
 - $\hat{m}_i^{(t+1)} / \sqrt{\hat{v}_i^{(t+1)}}$ approximates $\mathbb{E}\{g_i\} / \sqrt{\mathbb{E}\{g_i^2\}}$
 - $|\mathbb{E}\{g_i\} / \sqrt{\mathbb{E}\{g_i^2\}}| = |\mathbb{E}\{g_i\}| / \sqrt{\mathbb{E}\{g_i^2\}} \leq 1$
 - α upper bounds the steps (for $(1 - \beta_1) \leq \sqrt{1 - \beta_2}$)
- Observation: $|\mathbb{E}\{g_i\}|$ is upper bounded by $\mathbb{E}\{|g_i|\}$ and reaches this bound if all g_i have the same signs (assuming that the signs are the only difference in the sampled g_i). The latter is the case if the steps for that weight go in the same direction – and accordingly the learning rate can be large.



Adam: Initialization bias correction

- We have:

$$v_i^{(t+1)} = (1-\beta_2) \sum_{i=1}^t \beta_2^{t-i} (g_i^{(i)})^2 = (1-\beta_2) \sum_{i=0}^{t-1} \beta_2^{t-i-1} (g_i^{(i+1)})^2$$

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

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



Outline

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



Weight-decay

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

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

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

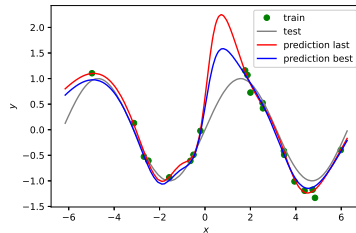
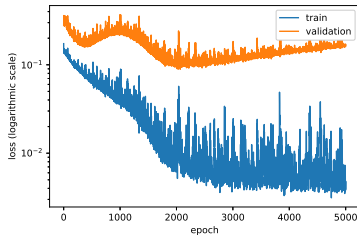
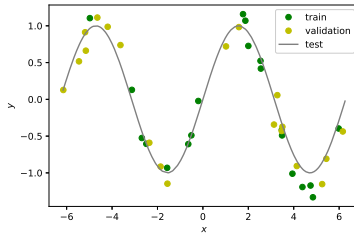
$$h_1, h_2, h_3, \dots$$

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



The secrets of successful shallow NN training

- Normalize the data component-wise to zero-mean and unit variance
- Use a single layer with “enough neurons”
- Magnitude of the weights is more important for the complexity of a layer than number of neurons (assuming sigmoidal activation functions):
 - Start with small weights
 - Employ early stopping
- Try shortcuts

