Statistical Machine Learning

Lecture 12: Neural Networks

Kristian Kersting
TU Darmstadt

Summer Term 2020

Today's Objectives

- Learn about Neural Networks
- Covered Topics
  - Learning representations
  - Single layer Perceptrons
  - Multilayer Perceptrons (MLPs)
  - Forward & Backpropagation
  - Efficient and Effective Gradient Descent
  - Theoretical Results
  - Applications

Outline

TECHNISCHE UNIVERSITÄT DARMSTADT

Outline

TECHNISCHE UNIVERSITÄT DARMSTADT

Learning Representations

- Up until now we had to come up with good features to solve our learning problems
  - For instance, in character image classification the features could be the number of grey pixels
- Feature selection is a laborious task. It is hard to choose the *right* features
- Try adding these two numbers in binary ...

$$0101101 + 1000001$$

- ... and now as decimals

$$45 + 65$$

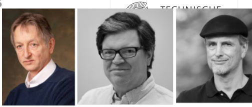
- Representation of your data matters
- Neural Networks learn complex data representations by combination of simpler ones

TECHNISCHE UNIVERSITÄT DARMSTADT

The Shift to Neural Networks

- The Big Shifts that lead to the current state of Neural Networks
  - Too little data $\Rightarrow$ Too much data
  - Linear and Convex $\Rightarrow$ Nonlinear and Nonconvex
  - Features intuitively obtainable (manually, automatic, indirectly via kernel) $\Rightarrow$ harder to obtain, key focus of learning
  - Optimization becomes easier by being deep (Whoops, did you see that coming?)
  - "Right number of parameters" $\Rightarrow$ "always too many"

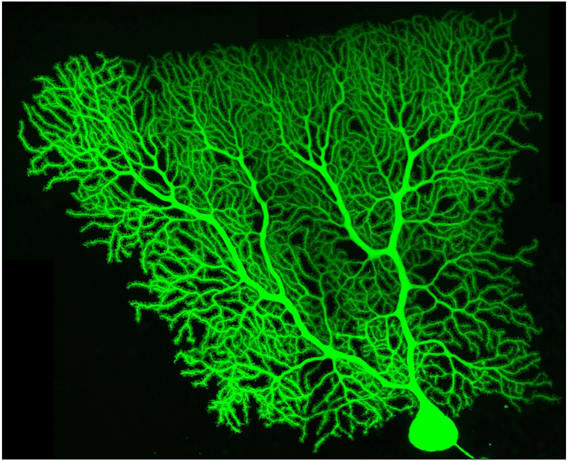
Neural Network History on One Slide



Geoffrey E. Hinton (1947-)
Yann LeCun (1960-)
Juergen Schmidhuber (1963-)
Ronald J. Williams (??-)

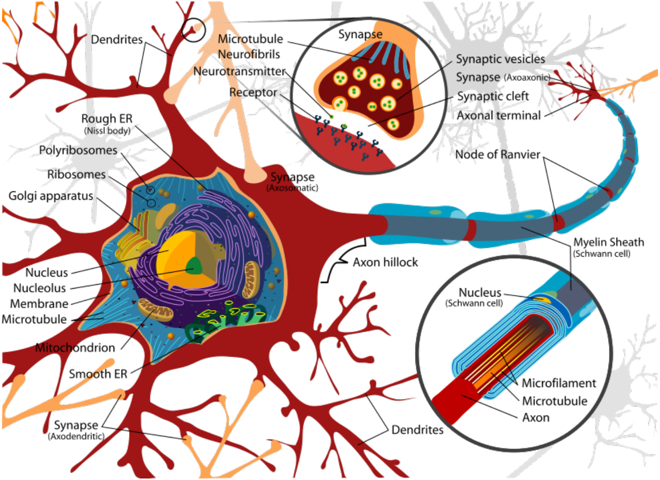
- **Precomputational (1888-)**: Neuron in Biology fully isolated by Ramon y Cajal...
- **Field Starts (1943-)**: McCullogh&Pitts Neuron and Networks
- **1st Hype (1957-)**: Rosenblatt's Perceptron
- **1st Winter (1969-)**: Papert/Minsky book perceptron with XOR example
- **2nd Hype (1986-1994)**: Rummelthart/Hinton/Williams *rediscover* Backpropagation
- **2nd Winter (1994-)**: Optimization is really hard, Kernels are better!
- **2007**: Rebooted by NIPS Workshop...
- **3rd Hype (2013-now)**: Amazing results in Computer Vision (ImageNet), Natural Language Processing, (Deep) Reinforcement Learning, ...

Neuron



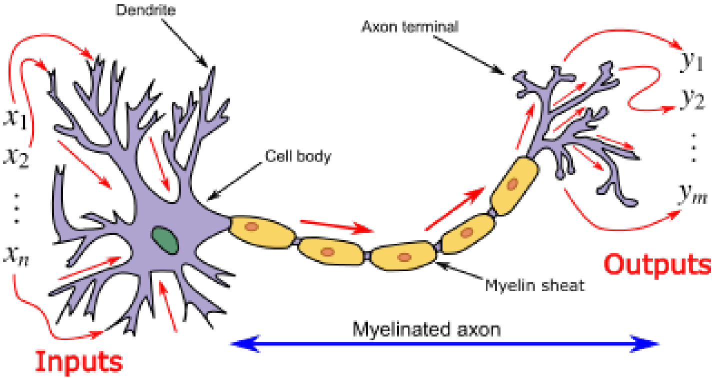
TECHNISCHE UNIVERSITÄT DARMSTADT

Neuron



TECHNISCHE UNIVERSITÄT DARMSTADT

Neuron



TECHNISCHE UNIVERSITÄT DARMSTADT

Biological Abstraction of a Neuron

- Abstract neuron model

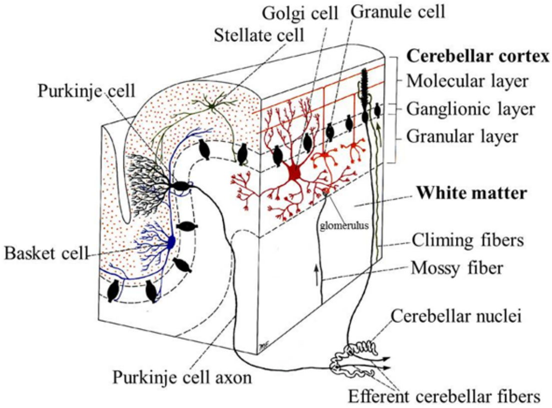
$$y = f\left(\sum_{i=1}^{n} \mathbf{w}_i x_i + w_0\right) = f\left(\mathbf{w}^\intercal \mathbf{x} + w_0\right) = f(\mathbf{w}_:^\intercal \mathbf{x}_:)$$

with

  - Input $\mathbf{x}_: = [\mathbf{x}^\intercal, 1]^\intercal$
  - Parameters/weights $\mathbf{w}_: = [\mathbf{w}^\intercal, w_0]^\intercal$
  - Bias/offset/threshold w_0
  - Activation function f

Biological Neural Network

- **Cerebellum**

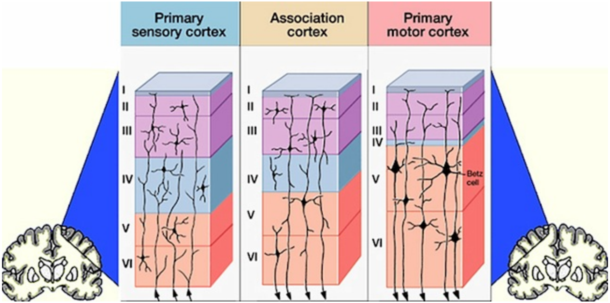


Adopted from: R.S.Snell. Clinical and functional histology for medical students, *Little, Brown and Company*, 1984

TECHNISCHE UNIVERSITÄT DARMSTADT

Biological Neural Network

- **Cerebrum**



I. Molecular Layer, II. External Granular Layer, III. External Pyramidal Layer, IV. Internal Granular Layer, V. Internal Pyramidal Layer, VI. Multiform Layer

Biological Abstraction of a Neural Network

- Neural networks in the brains are often determined by the sheets of tissue
  - Sheets = Vectors of Neurons
  - For simplicity in synthesis and analysis
- We pool neurons together in "layers" of *m* inputs and *n* outputs, where each layer has
  - Weight matrix $\mathbf{W} \in \mathbb{R}^{n \times m}$
  - Bias vector $\mathbf{w}_0 \in \mathbb{R}^{n \times 1}$
  - Input vector $\mathbf{x} \in \mathbb{R}^{m \times 1}$
  - Pre-activation vector
    $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{w}_0 = [\mathbf{W}, \mathbf{w}_0][\mathbf{x}^\intercal, 1]^\intercal = \mathbf{W}_{:}[\mathbf{x}^\intercal, 1]^\intercal \in \mathbb{R}^{n \times 1}$
  - Output vector $\mathbf{y} = \mathbf{f}(\mathbf{z})$, with $f : \mathbb{R}^{n \times 1} \rightarrow \mathbb{R}^{n \times 1}$

TECHNISCHE UNIVERSITÄT DARMSTADT

Outline

TECHNISCHE UNIVERSITÄT DARMSTADT

Reminder of Logistic Regression Classifier

- Model the class-posteriors as

$$p\left(C_1 \mid \mathbf{x}\right) = \sigma\left(\mathbf{w}^\intercal \mathbf{x} + w_0\right)$$

- Maximize the likelihood

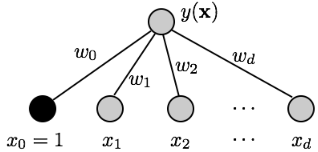
$$\begin{aligned} p\left(Y \middle| X; \mathbf{w}, w_0\right) &= \prod_{i=1}^{N} p\left(y_i \middle| \mathbf{x}_i; \mathbf{w}, w_0\right) \\ &= \prod_{i=1}^{N} p\left(C_1 \middle| \mathbf{x}_i; \mathbf{w}, w_0\right)^{1-y_i} p\left(C_2 \middle| \mathbf{x}_i; \mathbf{w}, w_0\right)^{y_i} \\ &= \prod_{i=1}^{N} \sigma\left(\mathbf{w}^\intercal \mathbf{x}_i + w_0\right)^{1-y_i} \left(1 - \sigma\left(\mathbf{w}^\intercal \mathbf{x}_i + w_0\right)\right)^{y_i} \end{aligned}$$

where $y_i = \{1, \mathbf{x}_i \text{ belongs to } C_2; 0, \mathbf{x}_i \text{ belongs to } C_1\}$

Logistic Regression Classifier as a Neural Network

TECHNISCHE UNIVERSITÄT DARMSTADT

- Single-Layer Network (without an hidden layer)



where $\mathbf{x}$ is the input layer, $\mathbf{w}$ are the weights and $y(\mathbf{x})$ is the output layer (here a single node)

- Linear output (linear regression function)

$$y(\mathbf{x}) = \mathbf{w}^\intercal \mathbf{x} + w_0 = \sum_{i=1}^{d} w_i x_i + w_0$$

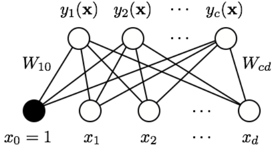
- Logistic output (classification)

$$y(\mathbf{x}) = \sigma(\mathbf{w}^\intercal \mathbf{x} + w_0)$$

TECHNISCHE UNIVERSITÄT DARMSTADT

Multi-Class Network

- Single-Layer Network with Multiple Outputs



- Multidimensional linear regression - linear output

$$y_k(\mathbf{x}) = \sum_{i=1}^{d} W_{ki} x_i$$

- Multi-class linear classification. Nonlinear extension is straightforward - logistic output

$$y_k(\mathbf{x}) = \sigma\left(\sum_{i=1}^{d} W_{ki} x_i\right)$$

The Least-Squares Loss Function

- In a supervised setting we have
  - N training data points $\mathbf{X} = \left[\mathbf{x}^1, \ldots, \mathbf{x}^N\right]$
  - For each data point there are c possible target values, $k \in 1, \ldots, c$, $\mathbf{T}_k = \left[t_k^1, \ldots, t_k^N\right]$
- With our model we can compute $y_k\left(\mathbf{x}^n; \mathbf{W}\right)$
- Least-squares error function

$$
\begin{aligned}
E\left(\mathbf{W}\right) &= \frac{1}{2} \sum_{n=1}^{N} \sum_{k=1}^{c} \left(y_k\left(\mathbf{x}^n; \mathbf{W}\right) - t_k^n\right)^2 \\
&= \frac{1}{2} \sum_{n=1}^{N} \sum_{k=1}^{c} \left(f\left(\sum_{i=1}^{d} W_{ki}\phi_i\left(\mathbf{x}^n\right)\right) - t_k^n\right)^2
\end{aligned}
$$

where $\phi_i\left(.\right)$ are arbitrary feature transformations

TECHNISCHE UNIVERSITÄT DARMSTADT

Learn the Weights with Gradient Descent

- Assume the output with a **linear activation**, i.e., $y_k\left(\mathbf{x}^n\right) = \sum_{i=1}^{d} W_{ki}\phi_i\left(\mathbf{x}^n\right)$

$$E\left(\mathbf{W}\right) = \sum_{n=1}^{N}\frac{1}{2}\sum_{k=1}^{c}\left(\sum_{i=1}^{d} W_{ki}\phi_i\left(\mathbf{x}^n\right) - t_k^n\right)^2 = \sum_{n=1}^{N} E^n\left(\mathbf{W}\right)$$

$$\frac{\partial E^n\left(\mathbf{W}\right)}{\partial W_{lj}} = \left(\sum_{i=1}^{d} W_{li}\phi_i\left(\mathbf{x}^n\right) - t_l^n\right)\phi_j\left(\mathbf{x}^n\right) = \left(y_l\left(\mathbf{x}^n\right) - t_l^n\right)\phi_j\left(\mathbf{x}^n\right)$$

- Update the weights with gradient descent

$$W_{lj} \leftarrow W_{lj} - \eta\left.\frac{\partial E\left(\mathbf{W}\right)}{\partial W_{lj}}\right|_{\mathbf{W}}$$

$$\frac{\partial E\left(\mathbf{W}\right)}{\partial W_{lj}} = \sum_{n=1}^{N}\frac{\partial E^n\left(\mathbf{W}\right)}{\partial W_{lj}}$$

- Computationally expensive if we use all the data points for gradient estimation (shortly we will see how to overcome this)

TECHNISCHE UNIVERSITÄT DARMSTADT

Learn the Weights with Gradient Descent

- Assume the output with a possible **non-linear activation**, i.e., $y_k(\mathbf{x}^n) = f(a_k) = f\left(\sum_{i=1}^{d} \mathbf{W}_{ki}\phi_i(\mathbf{x}^n)\right)$

$$\frac{\partial E^n(\mathbf{W})}{\partial W_{lj}} = f'(a_l)\left(y_l(\mathbf{x}^n) - t_l^n\right)\phi_j(\mathbf{x}^n)$$

- In a logistic neural network

$$\begin{aligned} f(a) &= \sigma(a) \\ \sigma'(a) &= \sigma(a)(1-\sigma(a)) \end{aligned}$$

TECHNISCHE UNIVERSITÄT DARMSTADT

Neural Networks

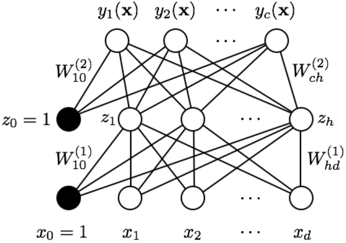
- NNs can be adapted to regression or classification
  - If we use a linear output node, we get a linear regression function
  - If we use a sigmoid output node, we get something similar to logistic regression
  - In either case, a classification can be obtained by taking the sign function
  - Nonetheless, at least classically, we don't use maximum likelihood, but a different learning criterion
- The actual power of NNs comes from extensions
  - Multi-class case
  - Multi-layer perceptron

Outline

TECHNISCHE UNIVERSITÄT DARMSTADT

Multi-Layer Perceptron

- Multi-Layer Network with Multiple Outputs



where **x** is the input layer, **z** is the hidden layer activation and **y** is the output layer

$$y_k(\mathbf{x}) = f^{(2)}\left(\sum_{i=0}^{h} W_{ki}^{(2)} \underbrace{f^{(1)}\left(\sum_{j=0}^{d} W_{ij}^{(1)} x_j\right)}_{z_i}\right)$$

Multi-Layer Perceptron

$$y_k(\mathbf{x}) = f^{(2)}\left(\sum_{i=0}^{h} W_{ki}^{(2)} f^{(1)}\left(\sum_{j=0}^{d} W_{ij}^{(1)} x_j\right)\right)$$

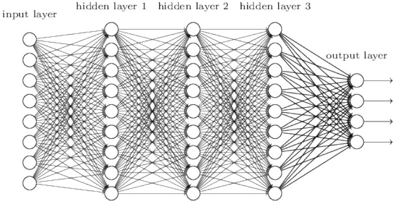
- $f^{(k)}$ are activation functions, for instance

$$f^{(1)}(a) = \sigma(a), \quad f^{(2)}(a) = a$$

- The hidden layer can have an arbitrary number of nodes h

Multi-Layer Perceptron

- There can also be multiple hidden layers with different sizes and activation functions $\Longrightarrow$ Multi-Layer Perceptron



$$y_k(\mathbf{x}) =$$

$$= f^{(N)}\left(\sum_{i_{N-1}=0}^{h_{N-1}} W^{(N)}_{k i_{N-1}} f^{(N-1)}\left(\sum_{i_{N-2}=0}^{h_{N-2}} W^{(N-1)}_{i_{N-1} i_{N-2}} f^{(N-2)}\left(\dots f^{(2)}\left(\sum_{i_1=0}^{h_1} W^{(2)}_{i_2 i_1} f^{(1)}\left(\sum_{i_0=0}^{d} W^{(1)}_{i_1 i_0} x_{i_0}\right)\right)\right)\right)\right)$$

[Michael Nielsen, neuralnetworksanddeeplearning.com]

Neural Networks Build Stacks of Features

- We can see a Multi-Layer network as a stack that builds features on top of features

$$y_k(\mathbf{x}) = f^{(N)}\left\{\sum_{i_{N-1}=0}^{h_{N-1}} W^{(N)}_{k i_{N-1}} \overbrace{f^{(N-1)}\left(\sum_{i_{N-2}=0}^{h_{N-2}} W^{(N-1)}_{i_{N-1} i_{N-2}} \underbrace{f^{(N-2)}\left(\ldots f^{(2)}\left(\sum_{i_1=0}^{h_1} W^{(2)}_{i_2 i_1} \overbrace{f^{(1)}\left(\sum_{i_0=0}^{d} W^{(1)}_{i_1 i_0} \underbrace{x_{i_0}}_{\phi^0_{i_0}}\right)}^{\phi^1_{i_1}}\right)\right)}_{\phi^{N-2}_{i_{N-2}}}\right)}^{\phi^{N-1}_{i_{N-1}}}\right\}$$

Universal Function Approximation - One Hidden Layer is Enough

George Cybenko (??)
Kurt Hornik (1963-)

- Universal Function Approximation Theorem
  - One hidden layer can represent every function arbitrarily accurate (Cybenko/Hornik)
- Even though true, we would need an exponential number of units. Instead, multiple layers allow for a similar effect with less units

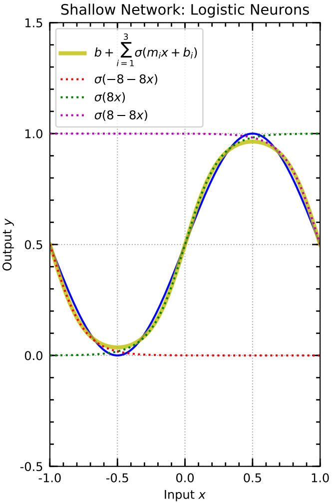
$$\#\text{regions} = O\left(\binom{n}{d}^{d(l-1)} n^d\right) \qquad \text{with} \begin{cases} n & \text{Number of neurons per layer} \\ l & \text{Number of hidden layers} \\ d & \text{Number of inputs} \end{cases}$$

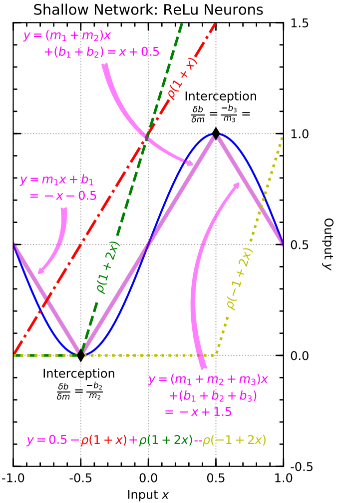
- Exponential growth in regions

$d = 1$	$l = 1$	$l = 2$	$\dots$	$l = k$
Regions	$O(n)$	$O(n^2)$	$\dots$	$O(n^k)$

Kurt Hornik et. al., “Multilayer feedforward networks are universal approximators”, 1989
G. Cybenko. Approximation by superpositions of a sigmoidal function. Mathematics of Control, Signals and Systems, 2(4):303-314, 1989.
Guido Montufar et.al., “On the Number of Linear Regions of Deep Neural Networks”, 2014

Universal Function Approximation Illustrated





Model Type and Model Class

- **Model type**: Choice of nonlinear parametric model
  - E.g., $\mathbf{y} = \mathbf{W}_3\mathbf{f}_2(\mathbf{W}_2\mathbf{f}_2(\mathbf{W}_1\mathbf{x}))$
  - Determined by
    1. Choice of topology: How are the neural layers connected and how many neurons per layer?
    2. Choice of neural elements: How do you model the neuron?
  - Little catch: EVERYTHING in ML was at some point called a neural network...
    - e.g., $f(z) = z$ is a linear network, RBFs, etc.
  - Activation function $f(z) = \phi(z)$ is just a feature function
- **Model class**: Number of hidden neurons, number of layers
  - E.g., $\dim\, \mathbf{f}_1(\mathbf{z})$

Model Type - Topologies

- Feedforward neural network: Acyclic directed graphs, e.g.,
    1. Multi-Layered Perceptrons: fully connected
    2. Convolutional neural networks: smartly pruned with weight-sharing
- Recurrent neural networks: Cyclic directed graphs with internal states, e.g., $y = f(\mathbf{z})$, $\mathbf{z}_{t+1} = f(\mathbf{x}, \mathbf{z}_t)$

TECHNISCHE UNIVERSITÄT DARMSTADT

Outline

TECHNISCHE UNIVERSITÄT DARMSTADT

Output Neurons

- Problem class determines the type for the output neurons
  - Linear for regression

$$\mathbf{f}(\mathbf{z}) = \mathbf{z}, \quad p(\mathbf{y}|\mathbf{z}) = \mathcal{N}(\mathbf{y}|\mathbf{z}, \sigma^2\mathbf{I})$$

    - E.g. from RL: to model a Gaussian stochastic policy the outputs can be the mean and the variance
  - Sigmoid for classification

$$f(z) = \sigma(z) \equiv \frac{1}{1 + \exp(-z)}, \quad p(y|z) = \sigma(z)^y (1 - \sigma(z))^{1-y}$$

  - Categorical Distribution/Softmax for multiclass-classification

$$f_i(\mathbf{z}) = \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)} \equiv p(y = i|\mathbf{z})$$

- All have probabilistic interpretations

TECHNISCHE UNIVERSITÄT DARMSTADT

Loss Functions

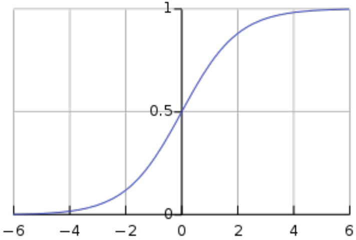
- The type of the output neuron is linked to the problem we want to solve, and so is the loss function
  - **Regression**
    - Linear output neuron $\Rightarrow$ Squared loss
  - **Classification**
    - Linear output $\Rightarrow$ Hinge loss
    - Sigmoid $\Rightarrow$ Nonlinear log-likelihood
  - **Multi-Class-Classification**
    - Softmax $\Rightarrow$ Nonlinear log-likelihood
- All derivable from maximum likelihood

TECHNISCHE UNIVERSITÄT DARMSTADT

Activation Functions

- Sigmoid

$$f(z) = \sigma(z)$$
$$f'(z) = \sigma(z)(1 - \sigma(z))$$



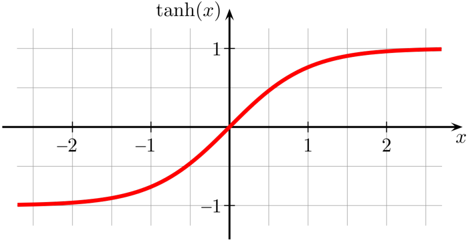
- What is the problem the sigmoid?
- The derivative is almost zero everywhere $\Longrightarrow$ zero gradient during backpropagation

UNIVERSITÄT DARMSTADT

Activation Functions

- Hyperbolic Tangent - tanh

$$f(z) = \tanh(z)$$
$$f'(z) = 1 - \tanh^2(z)$$

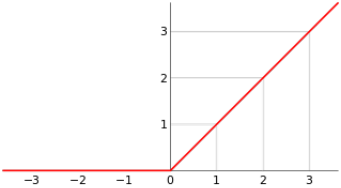


Activation Functions

- Rectified Linear Unit - ReLU

$$f(z) = \max(0, z)$$

$$f'(z) = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases}$$



- A bad initialization of the parameters can lead to a zero gradient
- In practice initialize the bias to a positive value

TECHNISCHE UNIVERSITÄT DARMSTADT

Activation Functions

- Hidden units may be chosen more freely - because we don't fully understand what they do!
  - $f'(z)$ determines how much a role that neuron plays in learning
- All technical choices remain voodoo...
- There are however best practices and heuristics on which to use

Demonstration

- `https://playground.tensorflow.org/`
- Classification problem
  - Linear separable dataset (third option)
    - with linear activation
    - non-linear activation
  - XOR dataset (second option)
    - with linear activation
    - with non-linear activation

TECHNISCHE UNIVERSITÄT DARMSTADT

Outline

TECHNISCHE UNIVERSITÄT DARMSTADT

Forward and Backpropagation

- In **forward propagation** compute
  - Activations at each hidden layer
  - Output(s) at the output layer
  - Resulting loss function
- In **backward propagation** (backpropagation) update the parameters
  - Compute the *contribution* of each parameter to the loss (gradient)
  - Update each parameter with gradient descent

Backpropagation

- Also known as *backprop*
- Gradient descent with chain rule
  - f is a function of one variable $f(a(x))$

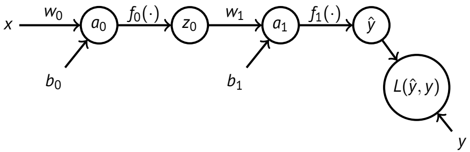
$$\frac{\partial f(a(x))}{\partial x} = \frac{\partial f(a(x))}{\partial a(x)} \frac{\partial a(x)}{\partial x}$$

  - f is a function of two variables $f(a(x), b(x))$

$$\frac{\partial f(a(x), b(x))}{\partial x} = \frac{\partial f(a(x))}{\partial a(x)} \frac{\partial a(x)}{\partial x} + \frac{\partial f(b(x))}{\partial b(x)} \frac{\partial b(x)}{\partial x}$$

- Invented in ML by a ton of people: Amari 1969, Werbos 1975, Rummelhardt et al 1989
- Known in control already in the 1950s, e.g., Bryson 1957
- Core Problems
  - Easy (Matrix): $\partial L / \partial \mathbf{W}_{:k}$, Hard (Tensor): $\partial \mathbf{a}_k / \partial \mathbf{W}_{:k}$

Example

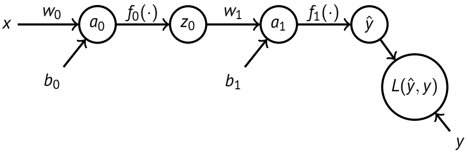


- What is $\frac{\partial L}{\partial w_0}$?

$$\frac{\partial L}{\partial w_0} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial a_1} \frac{\partial a_1}{\partial z_0} \frac{\partial z_0}{\partial a_0} \frac{\partial a_0}{\partial w_0}$$

TECHNISCHE UNIVERSITÄT DARMSTADT

Example



Forward pass

$$
\begin{aligned}
a_0 &= w_0 x + b_0 \\
z_0 &= f\left(a_0\right) \\
a_1 &= w_1 z_0 + b_1 \\
\hat{y} &= f_1\left(a_1\right)
\end{aligned}
$$

Backward pass

$$
\begin{aligned}
&\frac{\partial L}{\partial \hat{y}} & \frac{\partial L}{\partial w_1} &= \frac{\partial L}{\partial a_1}\frac{\partial a_1}{\partial w_1} \\
\frac{\partial L}{\partial a_1} &= \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial a_1} & \frac{\partial L}{\partial b_1} &= \frac{\partial L}{\partial a_1}\frac{\partial a_1}{\partial b_1} \\
\frac{\partial L}{\partial z_0} &= \frac{\partial L}{\partial a_1}\frac{\partial a_1}{\partial z_0} & \frac{\partial L}{\partial w_0} &= \frac{\partial L}{\partial a_0}\frac{\partial a_0}{\partial w_0} \\
\frac{\partial L}{\partial a_0} &= \frac{\partial L}{\partial z_0}\frac{\partial z_0}{\partial a_0} & \frac{\partial L}{\partial b_0} &= \frac{\partial L}{\partial a_0}\frac{\partial a_0}{\partial b_0}
\end{aligned}
$$

TECHNISCHE UNIVERSITÄT DARMSTADT

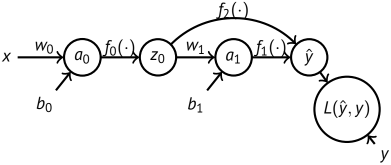
Example

- $L(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$
- $f_0 :=$ sigmoid activation, $f_1 :=$ linear activation
- $f_0(x) = \sigma(x)$, $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
- $f_1'(x) = 1$

$$
\begin{aligned}
\frac{\partial L}{\partial \hat{y}} &= y - \hat{y} & \frac{\partial L}{\partial w_1} &= (y - \hat{y})\, z_0 \\
\frac{\partial L}{\partial a_1} &= (y - \hat{y})\, f_1'(a_1) = (y - \hat{y}) \cdot 1 & \frac{\partial L}{\partial b_1} &= (y - \hat{y}) \cdot 1 \\
\frac{\partial L}{\partial z_0} &= (y - \hat{y})\, w_1 & \frac{\partial L}{\partial w_0} &= (y - \hat{y})\, w_1 f_0'(a_0)\, x \\
\frac{\partial L}{\partial a_0} &= (y - \hat{y})\, w_1 f_0'(a_0) & \frac{\partial L}{\partial b_0} &= (y - \hat{y})\, w_1 f_0'(a_0) \cdot 1
\end{aligned}
$$

Skip connections

- For parameters that are “closer” to the input, the gradient needs to flow from the loss until those parameters
- In **very** deep networks the application of the chain rule can lead to a zero gradient, and thus no learning occurs
- One solution is to use **skip connections**



$$
\begin{aligned}
\hat{y}(f_1, f_2) &= f_1(a_1) + f_2(z_0) \\
\frac{\partial \hat{y}}{\partial w_0} &= \frac{\partial \hat{y}}{\partial f_1}\frac{\partial f_1}{\partial w_0} + \frac{\partial \hat{y}}{\partial f_2}\frac{\partial f_2}{\partial w_0} \\
\frac{\partial \hat{y}}{\partial w_1} &= \frac{\partial \hat{y}}{\partial f_1}\frac{\partial f_1}{\partial w_1} + \frac{\partial \hat{y}}{\partial f_2}\underbrace{\frac{\partial f_2}{\partial w_1}}_{=0}
\end{aligned}
$$

TECHNISCHE UNIVERSITÄT DARMSTADT

Forward Propagation - the right way

- **Forward Propagation** through n layers

$$
\begin{aligned}
\mathbf{y} &= \mathbf{W}_{:n} \begin{bmatrix} \mathbf{a}_n^\intercal, & 1 \end{bmatrix}^\intercal \\
\mathbf{a}_n &= \mathbf{f}_{n-1}(\mathbf{z}_{n-1}) \\
\mathbf{z}_{n-1} &= \mathbf{W}_{:n-1} \begin{bmatrix} \mathbf{a}_{n-1}^\intercal, & 1 \end{bmatrix}^\intercal \\
\mathbf{a}_{n-1} &= \mathbf{f}_{n-2}(\mathbf{z}_{n-2}) \\
&\vdots \\
\mathbf{z}_2 &= \mathbf{W}_{:2} \begin{bmatrix} \mathbf{a}_2^\intercal, & 1 \end{bmatrix}^\intercal \\
\mathbf{a}_2 &= \mathbf{f}_1(\mathbf{z}_1) \\
\mathbf{z}_1 &= \mathbf{W}_{:1} \begin{bmatrix} \mathbf{a}_1^\intercal, & 1 \end{bmatrix}^\intercal \\
\mathbf{a}_1 &= \mathbf{x}
\end{aligned}
$$

Note: Bias vector $\mathbf{w}_k$ and weight matrix $\mathbf{W}_k$ yield $\mathbf{W}_{:k} = \begin{bmatrix} \mathbf{W}_k, & \mathbf{w}_k \end{bmatrix}$.
Where k indexes the layer

Backpropagation - the right way

Forwardpropagation		Backpropagation
$L(\mathbf{y}^{\mathrm{d}}, \mathbf{y}) = \frac{1}{2}(\mathbf{y}^{\mathrm{d}} - \mathbf{y})^{\mathsf{T}}(\mathbf{y}^{\mathrm{d}} - \mathbf{y})$		$\mathrm{d}L = -(\mathbf{y}^{\mathrm{d}} - \mathbf{y})^{\mathsf{T}}\, \mathrm{d}\mathbf{y}$
$\mathbf{y} = \mathbf{W}_{:n}\left[\mathbf{a}_n^{\mathsf{T}}, \quad 1\right]^{\mathsf{T}}$		$\mathrm{d}\mathbf{y} = \mathbf{W}_n\, \mathrm{d}\mathbf{a}_n$
$\mathbf{a}_n = \mathbf{f}_{n-1}(\mathbf{z}_{n-1})$		$\mathrm{d}\mathbf{a}_n = \mathbf{f}'_{n-1}(\mathbf{z}_{n-1})\, \mathrm{d}\mathbf{z}_{n-1}$
$\mathbf{z}_{n-1} = \mathbf{W}_{:n-1}\left[\mathbf{a}_{n-1}^{\mathsf{T}}, \quad 1\right]^{\mathsf{T}}$		$\mathrm{d}\mathbf{z}_{n-1} = \mathbf{W}_{n-1}\, \mathrm{d}\mathbf{a}_{n-1}$
$\mathbf{a}_{n-1} = \mathbf{f}_{n-2}(\mathbf{z}_{n-2})$		$\mathrm{d}\mathbf{a}_{n-1} = \mathbf{f}'_{n-2}(\mathbf{z}_{n-1})\, \mathrm{d}\mathbf{z}_{n-2}$
$\mathbf{z}_{n-2} = \mathbf{W}_{:n-2}\left[\mathbf{a}_{n-2}^{\mathsf{T}}, \quad 1\right]^{\mathsf{T}}$	$\Rightarrow$	$\mathrm{d}\mathbf{z}_{n-2} = \mathbf{W}_{n-2}\, \mathrm{d}\mathbf{a}_{n-2}$
$\vdots \quad \vdots$		$\vdots \quad \vdots$
$\mathbf{a}_2 = \mathbf{f}_1(\mathbf{z}_1)$		$\mathrm{d}\mathbf{a}_2 = \mathbf{f}'_1(\mathbf{z}_1)\, \mathrm{d}\mathbf{z}_1$
$\mathbf{z}_1 = \mathbf{W}_{:1}\left[\mathbf{a}_1^{\mathsf{T}}, \quad 1\right]^{\mathsf{T}}$		$\mathrm{d}\mathbf{z}_1 = \mathbf{W}_1\, \mathrm{d}\mathbf{a}_1$
$\mathbf{a}_1 = \mathbf{x}$		$\mathrm{d}\mathbf{a}_1 = \mathrm{d}\mathbf{x}$

TECHNISCHE UNIVERSITÄT DARMSTADT

Backpropagation - the right way

- Compute $\mathrm{D}_{\mathbf{z}_K} L$ from

$$
\begin{aligned}
\mathrm{d}L &= -(\mathbf{y}^{\mathrm{d}} - \mathbf{y})^{\intercal} \mathbf{W}_n \mathbf{f}'_{n-1}(\mathbf{z}_{n-1})\, \mathrm{d}\mathbf{z}_{n-1} && \text{for } K = n-1 \\
\mathrm{d}L &= -(\mathbf{y}^{\mathrm{d}} - \mathbf{y})^{\intercal} \mathbf{W}_n \mathbf{f}'_{n-1}(\mathbf{z}_{n-1}) \mathbf{W}_{n-1} \mathbf{f}'_{n-2}(\mathbf{z}_{n-2})\, \mathrm{d}\mathbf{z}_{n-2} && \text{for } K = n-2 \\
&\vdots && \vdots \\
\mathrm{d}L &= -(\mathbf{y}^{\mathrm{d}} - \mathbf{y})^{\intercal} \left(\prod_{k=n}^{K+1} \mathbf{W}_k \mathbf{f}'_{k-1}(\mathbf{z}_{k-1}) \right) \mathrm{d}\mathbf{z}_K && \text{for any } K
\end{aligned}
$$

for all other $K \in \{1, 2, \ldots, n\}$

Backpropagation - the right way

- **Outer Layer**
  Using
  $$d\mathbf{y} = (d\mathbf{W}_{:n})\,\mathbf{a}_{:n} = \operatorname{dvec}(\mathbf{W}_{:n})\,\mathbf{a}_n = (\mathbf{a}_{:n}^\top \otimes I)\operatorname{dvec}(\mathbf{W}_{:n}),$$
  we can check
  $$dL = -(\mathbf{y}^{\mathrm{d}} - \mathbf{y})^\top d\mathbf{y} = -(\mathbf{y}^{\mathrm{d}} - \mathbf{y})^\top (\mathbf{a}_{:n}^\top \otimes \mathbf{I})\operatorname{dvec}(\mathbf{W}_{:n}).$$
  Here, the Kronecker product $\mathbf{a}^\top \otimes \mathbf{I} = \begin{bmatrix} a_1\mathbf{I}, & \cdots, & a_m\mathbf{I} \end{bmatrix}$, the rules
  $$(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = \mathbf{AC} \otimes \mathbf{BD}, \qquad \alpha \otimes \mathbf{b} = \alpha\mathbf{b}$$
  $$\Longrightarrow \mathbf{b}^\top(\mathbf{c}^\top \otimes \mathbf{D}) = (1 \otimes \mathbf{b}^\top)(\mathbf{c}^\top \otimes \mathbf{D}) = \mathbf{c}^\top \otimes \mathbf{b}^\top\mathbf{D}$$
  and, thus, $DL = -\mathbf{a}_{:n}^\top \otimes (\mathbf{y}^{\mathrm{d}} - \mathbf{y})^\top$. Unvectorizing yields
  $$\frac{\partial L}{\partial \mathbf{W}_n} = \operatorname{vec}^{-1}_{\dim \mathbf{W}_n}(DL\,(\mathbf{W}_n)^\top) = -(\mathbf{y}^{\mathrm{d}} - \mathbf{y})\begin{bmatrix} \mathbf{a}_1^\top, & 1 \end{bmatrix}.$$
  The unvectorizing is commonly done by `reshape`.

TECHNISCHE UNIVERSITÄT DARMSTADT

Backpropagation - the right way

- **Hidden Layers and Input Layer**: Even $\mathrm{D}_{\mathbf{W}_K} L$ is easy using $\mathrm{d}\mathbf{z}_{K+1} = \left(\mathbf{a}_K^\intercal \otimes I\right) \mathrm{dvec}\, \mathbf{W}_K$

$$\frac{\partial L}{\partial \mathbf{W}_K} = -\left(\left(\prod_{k=K}^{n-1} \mathbf{W}_{k+1}^\intercal \mathbf{f}_k'(\mathbf{z}_k)\right)(\mathbf{y}^{\mathrm{d}} - \mathbf{y})\right)\left[\mathbf{a}_K^\intercal, \quad 1\right]$$

as $\mathbf{a}_1^\intercal = [\mathbf{x}^\intercal, 1]$ is the input layer, we also have the input layer that way

- It is computationally much more efficient to do

$$\frac{\partial L}{\partial \mathbf{W}_K} = -\left(\left(\bigodot_{k=K}^{n-1} \mathbf{W}_{k+1}^\intercal \mathbf{f}_{k\mathrm{diag}}'(\mathbf{z}_k)\right)(\mathbf{y}^{\mathrm{d}} - \mathbf{y})\right)\left[\mathbf{a}_K^\intercal, \quad 1\right]$$

with Hadamard product $[a_1, \ldots, a_n] \odot [b_1, \ldots, b_n] = [a_1 b_1, \ldots, a_n b_n]$, e.g., * in Python

Backpropagation

- Multi-layer perceptrons are usually trained using backpropagation
  - Non-convex, many local optima
  - Can get stuck in poor local optima
  - The design of a working backprop algorithm is somewhat of an *art*
  - Because of that, their use was in absolute winter between ~2000 and 2014
- Nonetheless, when these methods work, they work very well

TECHNISCHE UNIVERSITÄT DARMSTADT

Another Way to Compute the Gradients

- How would you compute the gradients without using backpropagation?
- We can see the loss as a function of the parameters, i.e., $L = L(\boldsymbol{w})$
- Using the definition of finite differences we compute the change in each parameter w_j as

$$\frac{\partial L}{\partial w_j} \approx \frac{L\left(\boldsymbol{w} + \epsilon \boldsymbol{u}_j\right) - L\left(\boldsymbol{w}\right)}{\epsilon}$$

where ϵ is a small perturbation and $\boldsymbol{u}_j$ is a unit vector in the j direction

Another Way to Compute the Gradients

$$\frac{\partial L}{\partial w_j} \approx \frac{L\left(\mathbf{w} + \epsilon \mathbf{u}_j\right) - L\left(\mathbf{w}\right)}{\epsilon}$$

- If a network as M parameters, how many times do you need to forward propagate to compute $L\left(\mathbf{w} + \epsilon \mathbf{u}_j\right)$ and $L\left(\mathbf{w}\right)$?
  - Exactly M times! If M is very large (for instance millions of parameters) it is very costly!

- With backpropagation, using the chain rule we can compute the partial derivatives with just one forward and one backward pass

TECHNISCHE UNIVERSITÄT DARMSTADT

Outline

TECHNISCHE UNIVERSITÄT DARMSTADT

Gradient Descent

$$\mathbf{W}_{:}^{k+1} = \mathbf{W}_{:}^{k} - \alpha \nabla_{\mathbf{W}_{:}} L$$

- Learning rate α
- Gradient from Backpropagation $\nabla_{\mathbf{W}_{:}} L$
- Questions
  - When to update **W**?
  - How to choose α?
  - How to initialize **W**?

When to Update W?

- **Full** Gradient Descent
  - Use the **whole** training set $\{(\boldsymbol{x}_i, \boldsymbol{y}_i)\}_{i=1,\ldots,n}$

$$\nabla_W J = \frac{1}{n} \sum_{i=1}^{n} \nabla_W L_f(\mathbf{x}_i, \mathbf{y}_i, \mathbf{W})$$

  - Computationally expensive for a large n
- **Stochastic** Gradient Descent (SGD)
  - Use **one data point** of the training set

$$\nabla_W J \approx \nabla_W L_f(\mathbf{x}_i, \mathbf{y}_i, \mathbf{W})$$

  - Needs adaptive learning rate η_t with $\sum_{t=1}^{\infty} \eta_t = \infty$ and $\sum_{t=1}^{\infty} \eta_t^2 < \infty$
  - High variance gradient estimation

TECHNISCHE UNIVERSITÄT DARMSTADT

When to Update W?

- **Mini-Batch** Gradient Descent
  - Use a batch of the training set

$$\nabla_W J \approx \frac{1}{k} \sum_{i=1}^{k} \nabla_W L_f(\mathbf{x}_i, \mathbf{y}_i, \mathbf{W})$$

with $k < n$

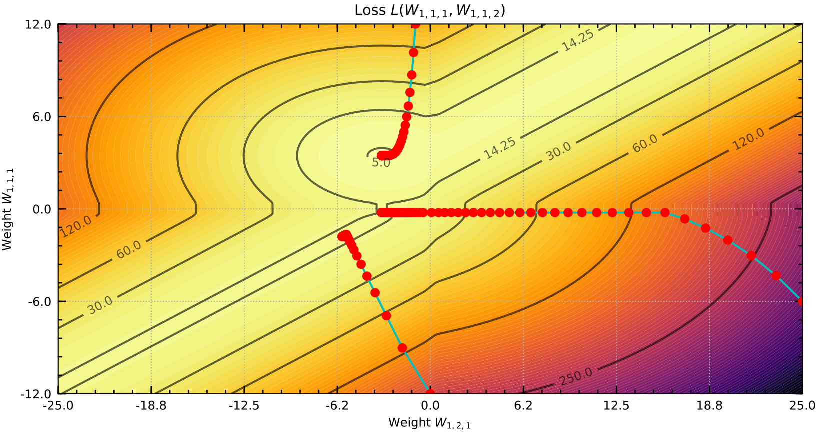
TECHNISCHE UNIVERSITÄT DARMSTADT

When to Update W?

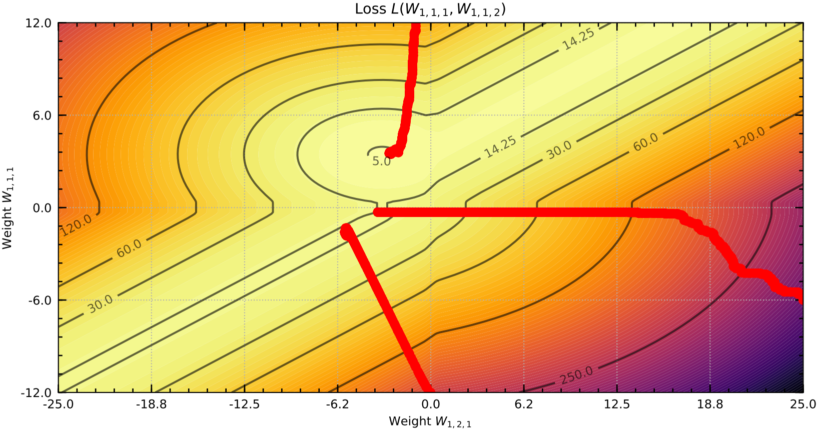
- Which one to choose?
  - Collecting data can introduce a strong bias in successive data samples
  - Updates for mini-batches will also be biased, leading to poor convergence due to big oscillations in weight updates
  - In practice: balance mini-batches approximately by random shuffling of the training data
- Side note: nowadays, when you read the term Stochastic Gradient Descent (SGD), most of the times it is referring to Mini-batch gradient descent

TECHNISCHE UNIVERSITÄT DARMSTADT

Full gradient descent

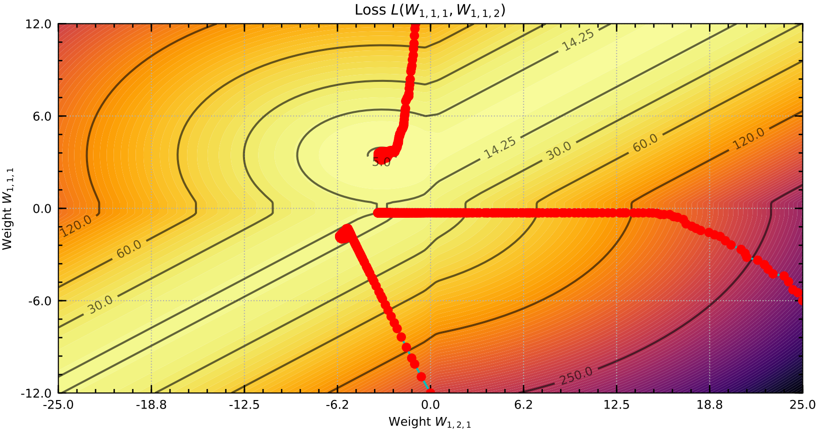


Stochastic Gradient Descent



TECHNISCHE UNIVERSITÄT DARMSTADT

Mini-Batch Gradient Descent

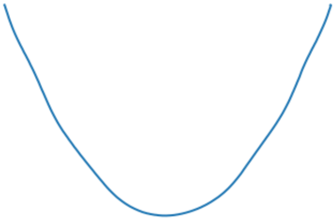


25% of the data

TECHNISCHE UNIVERSITÄT DARMSTADT

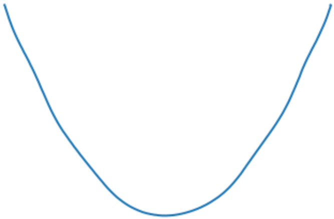
How to choose the learning rate α?

- **Very small** learning rate



How to choose the learning rate α?

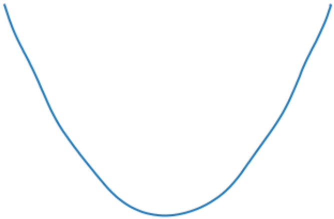
- **Good** learning rate



TECHNISCHE UNIVERSITÄT DARMSTADT

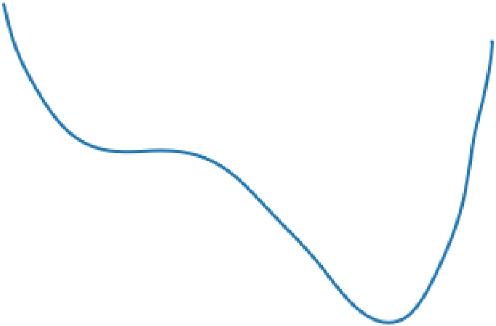
How to choose the learning rate α?

- **Large** learning rate



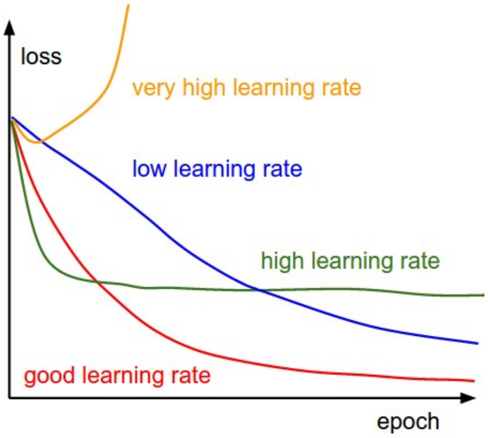
Plateaus and Valleys

- The learning rate should adapt to be larger in flat regions, but smaller inside the valley



TECHNISCHE UNIVERSITÄT DARMSTADT

Effect of the Learning Rate



[cs231n.github.io]

Learning Rate Adaptation - Momentum

- **Insight**: Running average $\bar{m}_0 = 0, \bar{m}_{k+1} = \gamma_k \bar{m}_k + (1-\gamma_k) m_k$
  - Geometric Average (Constant γ): $\bar{m}_{k+1} = (1-\gamma) \sum_{i=1}^{k} \gamma^{k-i} m_i$
  - Arithmetic Average ($\gamma_k = (k-1)/k$): $\bar{m}_{k+1} = (1/k) \sum_{i=1}^{k} m_i$
- **Practically:** Applied to Momentum Terms

$$\mathbf{M}_{k+1} = \gamma_k \mathbf{M}_k + (1-\gamma_k) \nabla J(\mathbf{W}_k)$$
$$\mathbf{W}_{k+1} = \mathbf{W}_k - \alpha_k \mathbf{M}_{k+1}$$

with $\mathbf{M}_0 = 0$

- **Physics-equivalent**: Move from 1st to 2nd Order ODE

TECHNISCHE UNIVERSITÄT DARMSTADT

Learning Rate Adaptation - Adadelta

- **Insight**: In plateaus, take large steps as they do not have much risk. In steep areas take smaller steps
- **Practically**: Normalize by running average of gradient norm

$$
\begin{aligned}
\mathbf{G}_k &= \nabla J(\mathbf{W}_k) \\
\mathbf{V}_{k+1} &= \gamma \mathbf{V}_k + (1-\gamma)\mathbf{G}_k \odot \mathbf{G}_k \\
\mathbf{W}_{k+1,ij} &= \mathbf{W}_{k+1,ij} - \frac{\alpha_k}{\sqrt{\mathbf{V}_{k,ij} + \epsilon}} \mathbf{G}_{k,ij}
\end{aligned}
$$

with a small ϵ to prevent division by zero and $\mathbf{V}_0 = \mathbf{0}$

- **Note**: Two versions exit (ϵ inside and outside root but in fraction)

[Zeiler, 2012, ADADELTA - An Adaptive Learning Rate Method]

TECHNISCHE UNIVERSITÄT DARMSTADT

Learning Rate Adaptation - Adam

- **Insight**: Combine Momentum Term with Adagrad
- **Practically**: Just combine both equations

$$
\begin{aligned}
\mathbf{G}_k &= \nabla J(\mathbf{W}_k) \\
\mathbf{V}_{k+1} &= \gamma_1 \mathbf{V}_k + (1-\gamma_1)\mathbf{G}_k \odot \mathbf{G}_k \\
\mathbf{M}_{k+1} &= \gamma_2 \mathbf{M}_k + (1-\gamma_2)\mathbf{G}_k \\
\mathbf{W}_{k+1,ij} &= \mathbf{W}_{k+1,ij} - \frac{\alpha_k}{\sqrt{\eta_{\gamma_1 k}\mathbf{V}_{k,ij} + \epsilon}} \eta_{\gamma_1 k}\mathbf{M}_{k+1,ij}
\end{aligned}
$$

  with a ϵ to prevent division by zero
- Initialization $\mathbf{V}_0 = \mathbf{0}$, $\mathbf{M}_0 = 0$ leads to underestimation fixed by

$$\eta_{\gamma_i k} = \frac{1}{1-\gamma_i^k}$$

- Choose $\gamma_1 = 0.9$, $\gamma_2 = 0.999$ and $\epsilon = 10^{-8}$. Not too sensitive to parameter changes
- **Note**: Violates convergence guarantees...

[Kingma et. al, 2015, Adam: A Method for Stochastic Optimization]

Better Directions for Small Networks

- **Hessian Approaches**
  - With Hessian $\mathbf{H} = \nabla^2 J$ you second order descent with $\delta \mathbf{w} = H^{-1} \nabla J$
  - Estimate Hessian from Gradient with Broyden−Fletcher−Goldfarb−Shanno (BFGS)
  - Use line search instead of learning rate
  - **Problem:** Too expensive for big networks
- **Conjugate gradient**
  - Momentum term with variable update rate, e.g.,

$$\delta \mathbf{w}_t = \nabla J(\mathbf{w}_t) + \frac{\nabla J(\mathbf{w}_t)^\intercal \nabla J(\mathbf{w}_t)}{\nabla J(\mathbf{w}_{t-1})^\intercal \nabla J(\mathbf{w}_{t-1})} \delta \mathbf{w}_t$$

    with Powell restarts (van der Smagt, 1994)
  - **Problem:** Fights stochastic gradient descent

TECHNISCHE UNIVERSITÄT DARMSTADT

Better Directions for Small Networks

- **Levenberg-Marquart**
  - Linearize network

$$f(\mathbf{x}_i, \mathbf{w}) = f(\mathbf{x}_i, \mathbf{w}_0) + \nabla_{\mathbf{w}} f(\mathbf{x}_i, \mathbf{w})\big|_{\mathbf{w}=\mathbf{w}_0}^{\mathsf{T}} \delta\mathbf{w} = \mathbf{f}_{i0} + \mathbf{J}_i \delta\mathbf{w}$$

    and solve regularized least squares problem

$$J \approx \frac{1}{2}\left(\mathbf{y} - (\mathbf{f}_0 + \mathbf{J}\delta\mathbf{w})\right)^{\mathsf{T}}\left(\mathbf{y} - (\mathbf{f}_0 + \mathbf{J}\delta\mathbf{w})\right) + \frac{1}{2}\delta\mathbf{w}^{\mathsf{T}}\mathbf{W}\delta\mathbf{w}$$

    which yields $\delta\mathbf{w} = (\mathbf{J}^{\mathsf{T}}\mathbf{J} + \mathbf{W})^{-1}\mathbf{J}_i^{\mathsf{T}}(\mathbf{y} - \mathbf{f}_0)$
  - Basically Gauss-Newton Method
  - **Levenberg $\mathbf{W} = \lambda\mathbf{I}$** keeps matrix invertible
  - **Marquardt $\mathbf{W} = \lambda \mathrm{diag}(\mathbf{J}^{\mathsf{T}}\mathbf{J})$**
- Adadelta approximates Levenberg's Method parameterwise

How to Initialize W?

- **Random Initialization**
  - Can lead to problems in gradient descent
  - For instance, large absolute values with sigmoid activation functions, or weights and biases negative or equal to zero in ReLU
- **Gaussian Initialization**
  - Weights $\mathbf{W}_{kij} \sim \mathcal{N}(0, m^{-1})$, Bias $\mathbf{w}_k \sim \mathcal{N}(0, 1)$
  - Basically normalization

TECHNISCHE UNIVERSITÄT DARMSTADT

How to Initialize W?

- **Xavier/Normalized Initialization**
  - Parameters W_j are initialized as

$$W_j \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right]$$

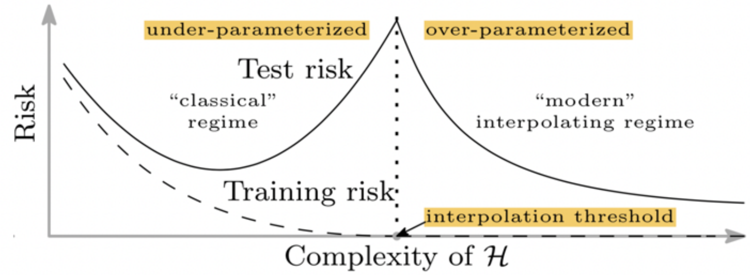
    where W_j are the weights connecting the previous hidden layer j and the next hidden layer $j+1$, n_j and n_{j+1} are the sizes of the previous and next layer, respectively, and U is the uniform distribution

  - Glorot et al, 2010, *Understanding the difficulty of training deep feedforward neural networks*

  - Note: Xavier initialization assumes the activation functions are symmetric and linear around 0, such as the tanh. For ReLUs it does not hold, as shown in He et al, 2015, *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*

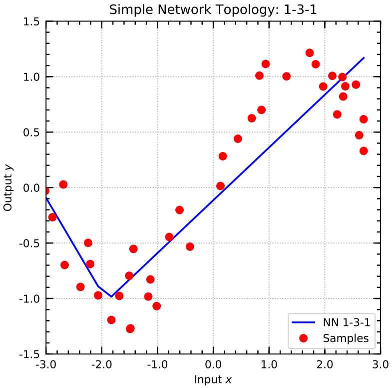
Outline

Risk vs Complexity



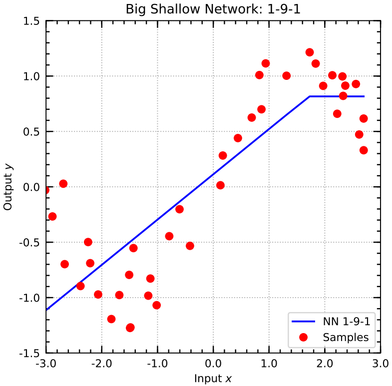
Shallow NN

- **Perfect Network Size**



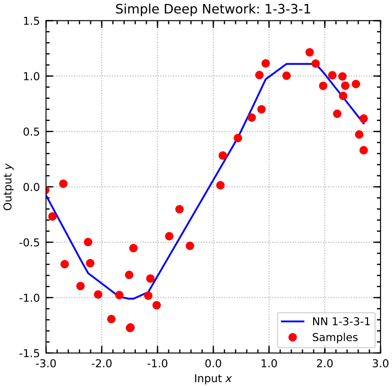
Shallow NN

- **Too Big Network**: Prone to overfitting?



Deep NN

- Deep Network with Equally Many Linear Regions

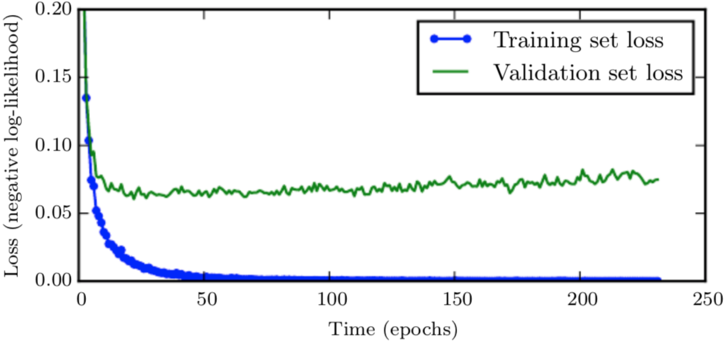


Neural Networks and Overfitting

- Neural Networks can contain hundreds, thousands and (sometimes) even millions of parameters
- In most cases we do not have datasets with millions of datapoints
- Neural Networks are prone to overfit
- Fight overfitting with an algorithmic realization of a prior
  - Regularization
  - Early stopping
  - Input noise augmentation
  - Dropout

Early Stopping

- Stop the training when the validation error starts rising again...



[Goodfellow et al, 2016, Deep Learning]

Weight Decay

- Ridge Loss $J(\mathbf{w}) = L(\mathbf{w}) + \lambda \mathbf{w}^\intercal \mathbf{w}$ yields **weight decay**

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k \left(\nabla_{\mathbf{w}} L(\mathbf{w}_k) + \lambda \mathbf{w}_k \right) = \left(1 - \lambda \alpha_k\right) \mathbf{w}_k + \alpha_k \nabla_{\mathbf{w}} L(\mathbf{w}_k)$$

Input Noise Augmentation

- Adding noise ϵ_i to inputs $\mathbf{x}_i$ reduces the chance of overfitting

$$\tilde{\mathbf{x}}_i = \mathbf{x}_i + \epsilon_i$$

TECHNISCHE UNIVERSITÄT DARMSTADT

Dropout

- Focus more effectively on the relevant neurons and prune others
- Zero out weights intermittently and let a subset of neurons predict
- Practically

$$a_i = f_i(z) d_i$$
$$\text{with } d_i \in \{0, 1\}$$
$$\text{and } p(d_i = 1) = p_{\mathrm{dropout}} = 0.5$$

[Srivastava et al, 2014, Dropout: A Simple Way to Prevent Neural Networks from Overfitting]

TECHNISCHE UNIVERSITÄT DARMSTADT

Improve Training - Batch normalization

- **Covariate Shift**
  - Change in input distribution makes learning hard
  - Problematic with mini-batches
  - Hidden values change as their preceding layers change
- Fought by **Batch Normalization**

$$\tilde{x}_i = \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

  - Like dropout with better performance?
  - Similar to normalization in Ridge regression
  - More complex: Removal of batch normalization

TECHNISCHE UNIVERSITÄT DARMSTADT

Outline

Why These Improvements in Performance?

- Features are learned rather than hand-crafted
- More layers capture more invariances (Razavian, Azizpour, Sullivan, Carlsson, *CNN Features off-the-shelf: an Astounding Baseline for Recognition*. CVPRW'14)
- More data to train deeper networks
- More computing power (GPUs)
- Better regularization methods: dropout
- New nonlinearities: max pooling, ReLU
- However, the theoretical understanding of deep networks remains shallow

Theoretical Results in Deep Learning

- Approximation, depth, width and invariance theory
  - Perceptrons and multilayer feedforward networks are universal approximators: Cybenko 1989, Hornik 1989, Hornik 1991, Barron 1993
  - Scattering networks are deformation stable for Lipschitz non-linearities: Bruna-Mallat 2013, Wiatowski 2015, Mallat 2016
- Generalization and regularization theory
  - Number of training examples grows exponentially with network size: Bartlett 2003
  - Distance and margin preserving embeddings: Giryes 2015, Sokolik 2016
  - Geometry, generalization bounds and depth efficiency: Montufar 2015, Neyshabur 2015, Shashua 2014/15/16

Theoretical Results in Deep Learning - References

- Cybenko, Approximations by superpositions of sigmoidal functions, Mathematics of Control, Signals, and Systems, 2 (4), 303-314, 1989
- Hornik, Stinchcombe and White. Multilayer feedforward networks are universal approximators, Neural Networks, 2(3), 359-366, 1989
- Hornik, Approximation Capabilities of Multilayer Feedforward Networks, Neural Networks, 4(2), 251–257, 1991
- Barron, Universal approximation bounds for superpositions of a sigmoidal function. IEEE Transactions on Information Theory, 39(3):930–945, 1993
- Bruna and Mallat, Invariant scattering convolution networks. Trans. PAMI, 35(8):1872–1886, 2013
- Wiatowski, Boelcskei, A mathematical theory of deep convolutional neural networks for feature extraction. arXiv 2015
- Mallat, Understanding deep convolutional networks. Phil. Trans. R. Soc. A, 374(2065), 2016

Theoretical Results in Deep Learning - References

- Bartlett and Maass, Vapnik-Chervonenkis dimension of neural nets. The handbook of brain theory and neural networks, pages 1188– 1192, 2003
- Giryes, Sapiro, A Bronstein, Deep Neural Networks with Random Gaussian Weights: A Universal Classification Strategy? arXiv:1504.08291
- Sokolic, Margin Preservation of Deep Neural Networks, 2015
- Montufar, Geometric and Combinatorial Perspectives on Deep Neural Networks, 2015
- Neyshabur, The Geometry of Optimization and Generalization in Neural Networks: A Path-based Approach, 2015

Outline

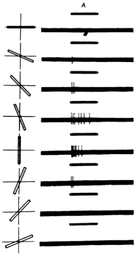
Hubel and Wiesel Receptive Fields

- D. H. Hubel and T. N. Wiesel, 1959, *Receptive fields of single neurones in the cat's striate cortex*
- The striate cortex is the first part of the visual cortex that processes visual information
  - A cat was shown a set of images (bars) with different orientations
  - Response in the striate cortex. Cells are activated with a vertical line

Deep Neural Networks
Convolutional Networks II

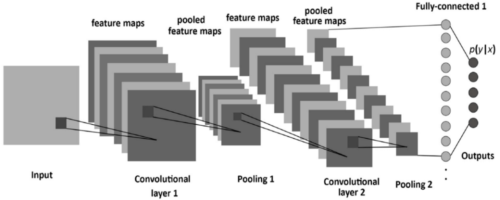
Bhiksha Raj

1



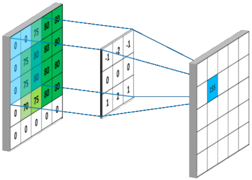
Convolutional Neural Networks (CNNs)

- CNNs are particularly suited for feature extraction in spatially correlated data, such as images
- Typical CNN for image classification task



Convolutional Neural Networks (CNNs)

- Features maps are computed by applying convolutional kernels to the input or feature maps



- Pooling reduces dimensionality. For instance, max_pooling(k) takes the pixel with largest value among k neighboring pixels

Why use Convolutions?

- Instead of computing the pre-activation of a layer with a matrix multiplication between weights and the previous layer, CNNs employ a convolution operation
- **Convolution**

$$s(t) = (x * w)(t) = \int x(a)\, w(t-a)\, \mathrm{d}a$$

  where x is the input signal and w is often called the kernel

- Acts as a filter of the input

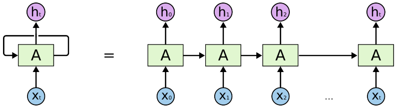
Why use CNNs instead of Fully Connected Networks?

- **Fully Connected Layers**
  - With high dimensional input data the number parameters explodes
    - Grey Image 1000 x 1000 pixels, hidden layer with 1000 units $\Longrightarrow$ 1 billion parameters (just for the first layer)
  - Does not extract local features, which is usually present in images
- **Convolutional Layers**
  - The learned parameters are the kernel weights, which are much smaller than the input and are shared over the whole input
  - Computes local features, since the output of a kernel involves a computation over adjacent pixels

TECHNISCHE UNIVERSITÄT DARMSTADT

Recurrent Neural Networks (RNNs)

- RNNs are networks with memory



$$h^t = f\left(h^{t-1}, x^t; W\right)$$

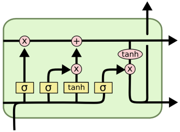
where ***h*** is the hidden layer, ***x*** is the input and ***W*** the parameters

- Used for time dependent / series data
  - Natural Language Processing
  - Speech Recognition
  - Dynamical Systems
  - Stock market
  - Brain-Computer Interface
  - ...

[colah.github.io]

Long Short-Term Memory Networks (LSTMs)

- Computing gradients in RNNs is done with Back-Propagation Through Time (BPTT). A parameter is updated by adding all the contributions to the loss over time
- BPTT in RNNs leads to vanishing and exploding gradients (Pascanu et al, 2013, On the difficulty of training recurrent neural networks)
- LSTMs fight the gradient problems with a different architecture that lets the gradient flow better in BPTT, and thus are capable of learning more effectively than traditional RNNs



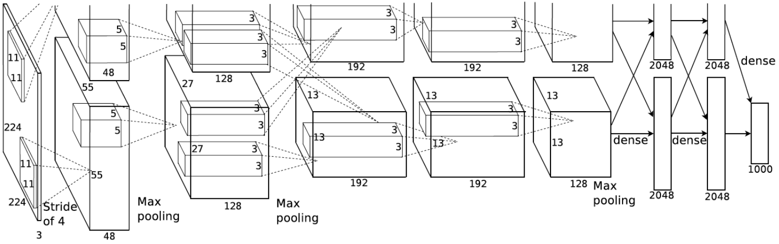
- For more information read Schmidhuber et al, 1997, Long Short-Term Memory

[colah.github.io]

Outline

Neural Networks in Computer Vision

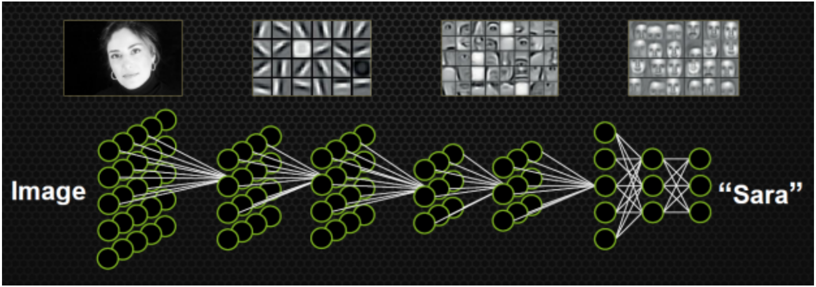
- Since 2012, CNNs have regained track in Computer Vision tasks after the achievement of AlexNet in the ImageNet Classification task
- Mainly from training in GPUs and using regularization techniques such as dropout



[Krizhevsky et al, 2012, ImageNet Classification with Deep Convolutional Neural Networks]

Neural Networks in Computer Vision

- The layers in CNNs learn interpretable representations

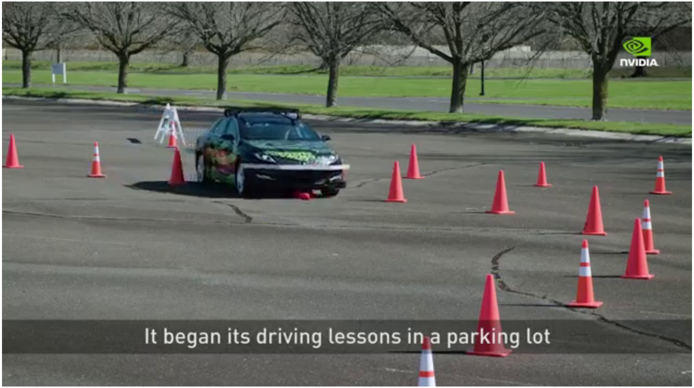


[Nvidia]

TECHNISCHE UNIVERSITÄT DARMSTADT

Neural Networks in Autonomous Systems

- End to End Learning for Self-Driving Cars

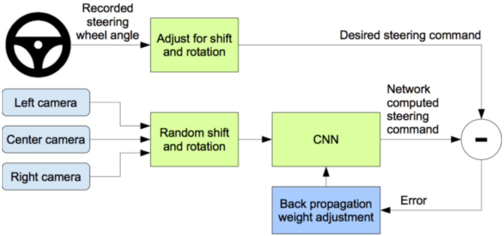


- https://www.youtube.com/watch?v=-96BEoXJMs0

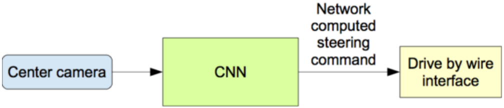
[Bojarski et al, 2016, End to End Learning for Self-Driving Cars]

Neural Networks in Autonomous Systems

- Training Network



- Prediction Network

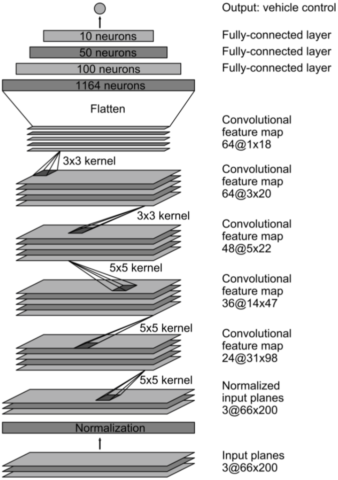


[Bojarski et al, 2016, End to End Learning for Self-Driving Cars]

TECHNISCHE UNIVERSITÄT DARMSTADT

Neural Networks in Autonomous Systems

- CNN Architecture



[Bojarski et al, 2016, End to End Learning for Self-Driving Cars]

TECHNISCHE UNIVERSITÄT DARMSTADT

Outline

1. Learning Representations and the Shift to Neural Networks
2. Single-Layer Neural Networks
3. Multi-Layer Neural Networks
4. Output Neurons and Activation Functions
5. Forward and Backpropagation
6. Gradient Descent
7. Overfitting
8. Theoretical Results
9. Other Network Architectures
10. Examples
11. **Wrap-Up**

11. Wrap-Up

You know now:

- What neural networks are and how they relate to the brain
- How neural networks build stacks of feature representations
- A network of one layer is enough, but in practice not a good idea
- How to do forward and backpropagation
- Different ways of doing fast gradient descent
  - Full, stochastic, mini-batch
  - Speedup training via learning rate adaptation
  - How to initialize the parameters
- Why neural networks overfit and what you can do to about it
- Why CNNs are used for spatial correlated data
- Why LSTMs are used for time series data

Self-Test Questions

- How does logistic regression relate to neural networks?
- How do neural networks relate to the brain?
- What kind of functions can single layer neural networks learn?
- Why do two layers help? How many layers do you need to represent arbitrary functions?
- Why were neural networks abandoned in the 1970s, and later in the 1990s? Why did neural networks re-awaken in the 2010s?
- What output layer and loss function to use given the task (regression, classification)?
- Why use a ReLU activation instead of sigmoid?
- Derive the equations for forward and backpropagation for a simple network
- What is mini-batch gradient descent? Why use it instead of SGD or full gradient descent?
- Why neural networks can overfit and what are the options to prevent it?

TECHNISCHE UNIVERSITÄT DARMSTADT

Acknowledgment and Extra Material

- Some ideas for these slides where taken from the Machine Learning lecture (SS 2017) from the University of Freiburg, and from Stanford lecture on Convolutional Neural Networks (http://cs231n.github.io/convolutional-networks/)
- Deep Learning Book, 2016, Goodfellow, Bengio, Courville
  - https://www.deeplearningbook.org/
- Neural Networks Playground
  - https://playground.tensorflow.org/
- Sebastian Ruder's blog has an overview on gradient descent optimization algorithms
  - http://ruder.io/optimizing-gradient-descent/
- Andrej Karpathy's blog has a recent overview on best practices to train neural networks
  - http://karpathy.github.io/2019/04/25/recipe/

Homework

- Reading Assignment for next lecture
  - Bishop 7.1