Feedforward neural networks

Deep learning reading group

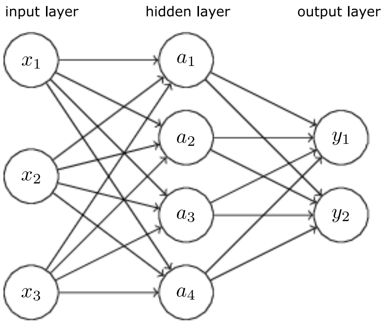
Lisa Handl

Max Planck Institute for Informatics

May 11, 2017

Vanilla Neural Network

- Also: single hidden layer backpropagation network
- Input layer: fixed numbers
- Hidden layer
  - $z_i = \sum_j w_{i,j} x_j + b_i$ weighted input
  - $a_i = \sigma(z_i)$ activation
  - σ activation function
- Output layer
  - Like hidden layer
  - Sometimes different activation function

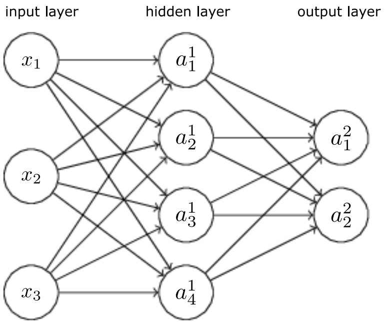


adapted from [1]

[1] M. A. Nielsen, Neural Networks and Deep Learning, `http://neuralnetworksanddeeplearning.com`

Vanilla Neural Network

- Also: single hidden layer backpropagation network
- Input layer: fixed numbers
- Hidden layer
  - $z_i = \sum_j w_{i,j} x_j + b_i$ weighted input
  - $a_i = \sigma(z_i)$ activation
  - σ activation function
- Output layer
  - Like hidden layer
  - Sometimes different activation function



adapted from [1]

[1] M. A. Nielsen, Neural Networks and Deep Learning, http://neuralnetworksanddeeplearning.com

Types of Neurons

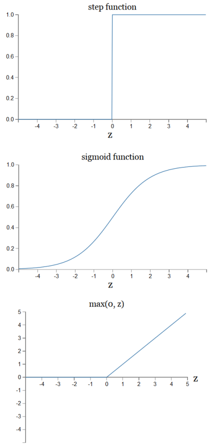
- Perceptron
  - $\sigma(z) = \mathbb{1}_{(0,\infty)}(z)$

  $\Rightarrow$ Binary decisions

  $\Rightarrow$ Multilayer perceptron (MLP)

- Sigmoid neuron
  - $\sigma(z) = \frac{1}{1+e^{-z}}$
  - Smooth version of perceptron

- Rectified linear unit
  - $\sigma(z) = \max\{0, z\}$
  - No saturation for large z

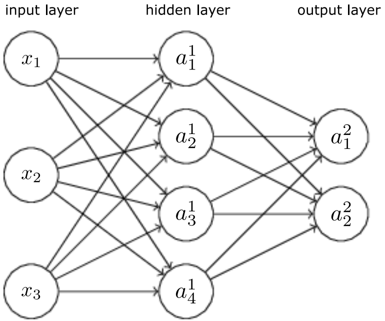


[1]

[1] M. A. Nielsen, Neural Networks and Deep Learning, `http://neuralnetworksanddeeplearning.com`

Output Layer

- $z_i^2 = \sum_j w_{i,j}^2 a_j^1 + b_i^2$
  weighted input
- Transformation depends on type of output
- For regression:
  - linear unit ($a_i^2 = z_i^2$)
  - sigmoid neuron
- For classification:
  - K neurons for K classes (modeling class probabilities)
  - e.g., softmax function
    $a_i^2 = \exp(z_i^2) / \sum_k \exp(z_k^2)$

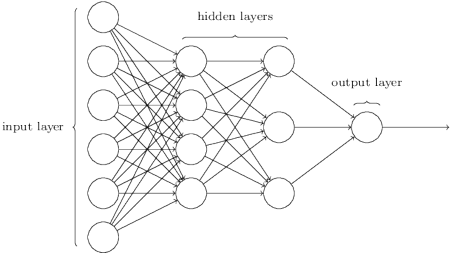


adapted from [1]

[1] M. A. Nielsen, Neural Networks and Deep Learning, http://neuralnetworksanddeeplearning.com

Design Choices

- Number of hidden neurons / layers
- Number and placement of edges
- Choice of activation functions

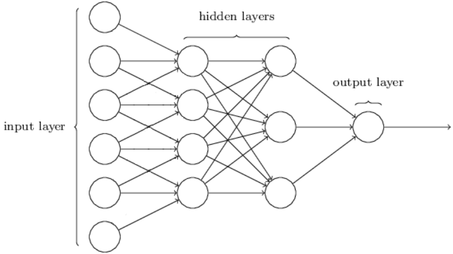


[1]

[1] M. A. Nielsen, Neural Networks and Deep Learning, http://neuralnetworksanddeeplearning.com

Design Choices

- Number of hidden neurons / layers
- Number and placement of edges
- Choice of activation functions

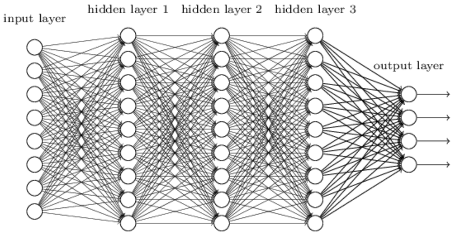


[1]

[1] M. A. Nielsen, Neural Networks and Deep Learning, `http://neuralnetworksanddeeplearning.com`

Design Choices

- Number of hidden neurons / layers
- Number and placement of edges
- Choice of activation functions

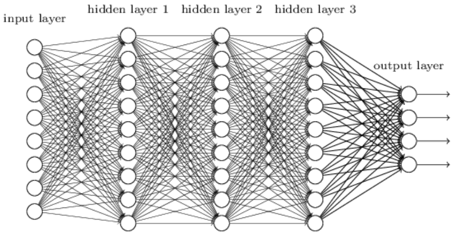


[1]

[1] M. A. Nielsen, Neural Networks and Deep Learning, http://neuralnetworksanddeeplearning.com

Design Choices

- Number of hidden neurons / layers
- Number and placement of edges
- Choice of activation functions



[1]

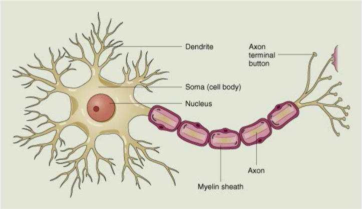
- Restriction: feedforward $=$ no feedback connections

  $\Rightarrow$ Directed acyclic graph
- Weights ($w^{\ell}_{i,j}$) and biases (b^{ℓ}_{i}) are learned from data

[1] M. A. Nielsen, Neural Networks and Deep Learning, http://neuralnetworksanddeeplearning.com

Motivation: The Human Brain

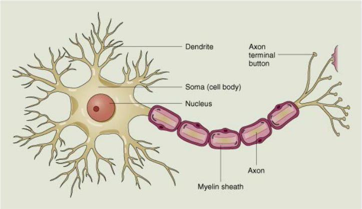
- Dendrites receive input from connected cells (post-synaptic potential)
- Soma integrates inputs and fires (action potential) if a threshold is exceeded



- The higher the added inputs, the higher are firing frequency and released level of neurotransmitter

Motivation: The Human Brain

- Dendrites receive input from connected cells (post-synaptic potential)
- Soma integrates inputs and fires (action potential) if a threshold is exceeded



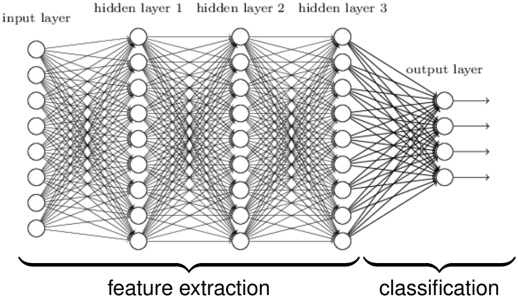
- The higher the added inputs, the higher are firing frequency and released level of neurotransmitter

⇒ This is only for *motivation*!

Artificial neural networks should be seen as statistical models, not as models for the human brain!

What Happens Inside a Neural Network?

- Each hidden neuron projects inputs along some direction and fits a sigmoid along this direction
- Can be interpreted as learning features
- The last layer is the actual predictor / classifier



[1]

[1] M. A. Nielsen, Neural Networks and Deep Learning, `http://neuralnetworksanddeeplearning.com`

Universality Theorem

- Neural networks can approximate any (Borel-measurable) function f between finite-dimensional spaces:

  For any $\varepsilon > 0$ there exists a neural network g such that $|f(x) - g(x)| < \varepsilon$ for all possible inputs x

- This holds even when only one hidden layer is used

- Formal proofs: Cybenko (1989) [2], Hornik et al. (1989) [3]

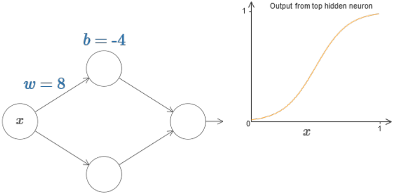
---

[2] G. Cybenko, Approximation by superpositions of a sigmoidal function, Math. Control Signals systems 2 (1989)

[3] K. Hornik, M. Stinchcombe, H. White, Multilayer feedforward networks are universal approximators, Neural Networks 2 (1989)

An Intuitive Explanation – 1D

- Sigmoids can approximate step functions
- For $\sigma(wx + b)$ the „step" is at $s = -b/w$

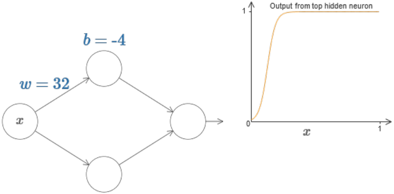


[1]

[1] M. A. Nielsen, Neural Networks and Deep Learning, http://neuralnetworksanddeeplearning.com

An Intuitive Explanation – 1D

- Sigmoids can approximate step functions
- For $\sigma(wx + b)$ the „step“ is at $s = -b/w$

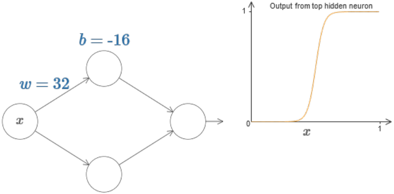


[1]

[1] M. A. Nielsen, Neural Networks and Deep Learning, http://neuralnetworksanddeeplearning.com

An Intuitive Explanation – 1D

- Sigmoids can approximate step functions
- For $\sigma(wx + b)$ the „step“ is at $s = -b/w$

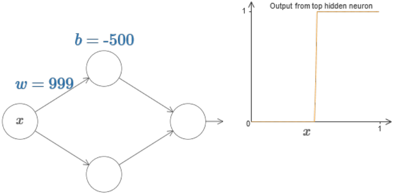


1

[1] M. A. Nielsen, Neural Networks and Deep Learning, `http://neuralnetworksanddeeplearning.com`

An Intuitive Explanation – 1D

- Sigmoids can approximate step functions
- For $\sigma(wx + b)$ the „step“ is at $s = -b/w$

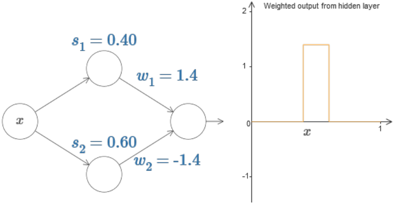


[1]

[1] M. A. Nielsen, Neural Networks and Deep Learning, `http://neuralnetworksanddeeplearning.com`

An Intuitive Explanation – 1D

- Sigmoids can approximate step functions
- For $\sigma(wx + b)$ the „step“ is at $s = -b/w$
- Two neurons can approximate a bump



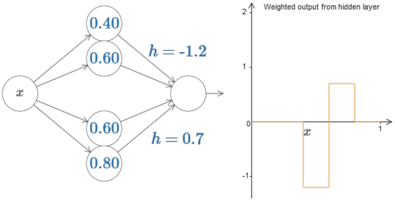
[1]

---

[1] M. A. Nielsen, Neural Networks and Deep Learning, `http://neuralnetworksanddeeplearning.com`

An Intuitive Explanation – 1D

- Sigmoids can approximate step functions
- For $\sigma(wx + b)$ the „step“ is at $s = -b/w$
- Two neurons can approximate a bump

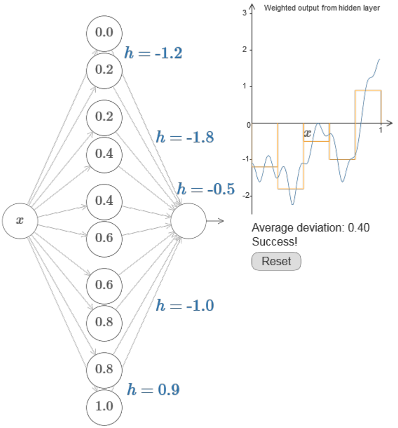


[1]

[1] M. A. Nielsen, Neural Networks and Deep Learning, http://neuralnetworksanddeeplearning.com

An Intuitive Explanation – 1D

- Sigmoids can approximate step functions
- For $\sigma(wx + b)$ the „step“ is at $s = -b/w$
- Two neurons can approximate a bump
- Enough neurons can approximate any piecewise constant function



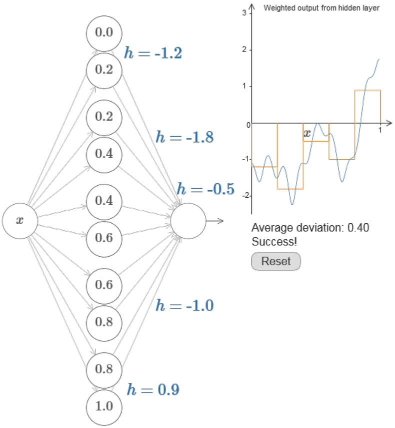
[1]

[1] M. A. Nielsen, Neural Networks and Deep Learning, `http://neuralnetworksanddeeplearning.com`

An Intuitive Explanation – 1D

- Sigmoids can approximate step functions
- For $\sigma(wx + b)$ the „step“ is at $s = -b/w$
- Two neurons can approximate a bump
- Enough neurons can approximate any piecewise constant function

$\Rightarrow$ Online: clickable graphs!



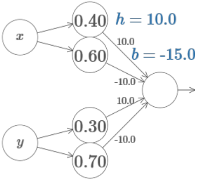
[1]

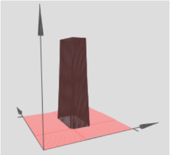
---

[1] M. A. Nielsen, Neural Networks and Deep Learning, http://neuralnetworksanddeeplearning.com

An Intuitive Explanation – 2D

- Four neurons can approximate a tower function



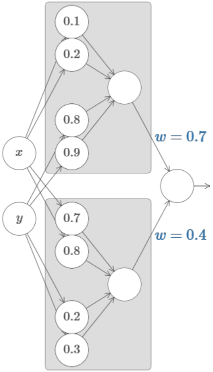


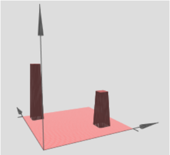
[1]

[1] M. A. Nielsen, Neural Networks and Deep Learning, http://neuralnetworksanddeeplearning.com

An Intuitive Explanation – 2D

- Four neurons can approximate a tower function
- Tower functions can approximate any continuous function in 2D



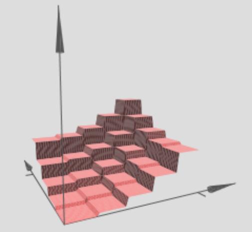


[1]

[1] M. A. Nielsen, Neural Networks and Deep Learning, http://neuralnetworksanddeeplearning.com

An Intuitive Explanation – 2D

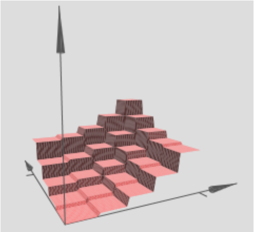
- Four neurons can approximate a tower function
- Tower functions can approximate any continuous function in 2D


[1]

[1] M. A. Nielsen, Neural Networks and Deep Learning, http://neuralnetworksanddeeplearning.com

An Intuitive Explanation – 2D

- Four neurons can approximate a tower function
- Tower functions can approximate any continuous function in 2D
- The same works in higher dimensions

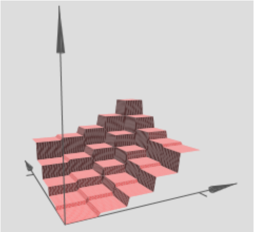


[1]

[1] M. A. Nielsen, Neural Networks and Deep Learning, http://neuralnetworksanddeeplearning.com

An Intuitive Explanation – 2D

- Four neurons can approximate a tower function
- Tower functions can approximate any continuous function in 2D
- The same works in higher dimensions
- Drawback: Many hidden neurons are needed for a good approximation!



1

[1] M. A. Nielsen, Neural Networks and Deep Learning, http://neuralnetworksanddeeplearning.com

Training Neural Networks

- Parameters to learn:
  - The weights $w^{\ell}_{i,j}$
  - The biases b^{ℓ}_{i}
- Minimize cost function
  - Measures difference between data and modeled distribution
  - Choice depends on the output unit
  - Optimized via backpropagation
    (clever way to do gradient descent)

Training Neural Networks

- Parameters to learn:
  - The weights $w^{\ell}_{i,j}$
  - The biases b^{ℓ}_{i}
- Minimize cost function
  - Measures difference between data and modeled distribution
  - Choice depends on the output unit
  - Optimized via backpropagation
    (clever way to do gradient descent)
- Cost function is usually non-convex

$\Rightarrow$ No convergence guarantees!

$\Rightarrow$ Initial values matter!

Cost Functions and Maximum Likelihood

- Usually the negative log-likelihood
  - Define $p_{\text{model}}(y \mid x)$
  - Minimize $-\frac{1}{N}\sum_{i=1}^{N} \log p_{\text{model}}(y_i \mid x_i)$

Cost Functions and Maximum Likelihood

- Usually the negative log-likelihood
  - Define $p_{\text{model}}(y \mid x)$
  - Minimize $-\frac{1}{N}\sum_{i=1}^{N} \log p_{\text{model}}(y_i \mid x_i)$
- Linear output unit: squared error
  - $C(w, b) = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}(x_i) - y_i)^2$
  - Model assumption: $p_{\text{model}}(y \mid x) = f_{\mathrm{N}(\hat{y}(x),1)}(y)$

Cost Functions and Maximum Likelihood

- Usually the negative log-likelihood
  - Define $p_{\text{model}}(y \mid x)$
  - Minimize $-\frac{1}{N}\sum_{i=1}^{N} \log p_{\text{model}}(y_i \mid x_i)$
- Linear output unit: squared error
  - $C(w, b) = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}(x_i) - y_i)^2$
  - Model assumption: $p_{\text{model}}(y \mid x) = f_{\text{N}(\hat{y}(x),1)}(y)$

$$
\begin{aligned}
-\log\left(\prod_{i=1}^{N} p(y_i \mid x_i)\right) &= -\log\left(\prod_{i=1}^{N} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}(y_i - \hat{y}_i(x_i))^2\right)\right) \\
&= -N\log\left(\frac{1}{\sqrt{2\pi}}\right) + \frac{1}{2}\sum_{i=1}^{N}(y_i - \hat{y}_i(x_i))^2
\end{aligned}
$$

Cost Functions and Maximum Likelihood

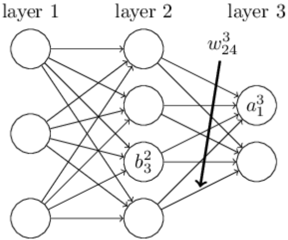
- Usually the negative log-likelihood
  - Define $p_{\text{model}}(y \mid x)$
  - Minimize $-\frac{1}{N}\sum_{i=1}^{N} \log p_{\text{model}}(y_i \mid x_i)$
- Linear output unit: squared error
  - $C(w, b) = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}(x_i) - y_i)^2$
  - Model assumption: $p_{\text{model}}(y \mid x) = f_{\mathrm{N}(\hat{y}(x),1)}(y)$
- Sigmoid unit: cross-entropy
  - $C(w, b) = \frac{1}{N}\sum_{i=1}^{N}(y_i \log(\hat{y}(x_i)) + (1 - y_i)\log(1 - \hat{y}(x_i)))$
  - Logistic linear regression model in hidden units

Cost Functions and Maximum Likelihood

- Usually the negative log-likelihood
  - Define $p_{\text{model}}(y \mid x)$
  - Minimize $-\frac{1}{N}\sum_{i=1}^{N} \log p_{\text{model}}(y_i \mid x_i)$
- Linear output unit: squared error
  - $C(w, b) = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}(x_i) - y_i)^2$
  - Model assumption: $p_{\text{model}}(y \mid x) = f_{\mathrm{N}(\hat{y}(x),1)}(y)$
- Sigmoid unit: cross-entropy
  - $C(w, b) = \frac{1}{N}\sum_{i=1}^{N}(y_i \log(\hat{y}(x_i)) + (1 - y_i)\log(1 - \hat{y}(x_i)))$
  - Logistic linear regression model in hidden units
- Softmax: log-likelihood cost function
  - $C(w, b) = -\frac{1}{N}\sum_{i=1}^{N} \log(\hat{y}_{y_i}(x_i))$

Backpropagation – Notation

- $w^{\ell}_{j,k}$: weight for the connection from neuron k in layer $\ell - 1$ to neuron j in layer ℓ
- b^{ℓ}_{j} and a^{ℓ}_{j}: bias and activation of neuron j in layer ℓ



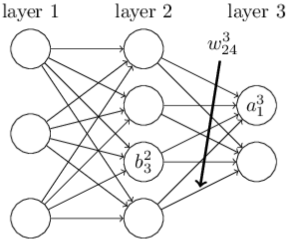
[1]

[1] M. A. Nielsen, Neural Networks and Deep Learning, `http://neuralnetworksanddeeplearning.com`

Backpropagation – Notation

- $w^{\ell}_{j,k}$: weight for the connection from neuron k in layer $\ell - 1$ to neuron j in layer ℓ
- b^{ℓ}_{j} and a^{ℓ}_{j}: bias and activation of neuron j in layer ℓ
- $z^{\ell}_{j} = \sum_k w^{\ell}_{j,k} a^{\ell-1}_{k} + b^{\ell}_{j}$ and $a^{\ell}_{j} = \sigma(z^{\ell}_{j})$

  $\Rightarrow z^{\ell} = w^{\ell} a^{\ell-1} + b^{\ell}$ and $a^{\ell} = \sigma(z^{\ell})$



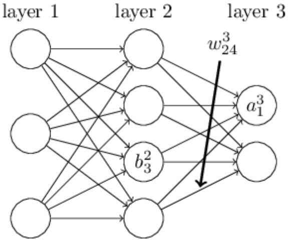
[1]

[1] M. A. Nielsen, Neural Networks and Deep Learning, `http://neuralnetworksanddeeplearning.com`

Backpropagation – Notation

- $w^{\ell}_{j,k}$: weight for the connection from neuron k in layer $\ell - 1$ to neuron j in layer ℓ
- b^{ℓ}_{j} and a^{ℓ}_{j}: bias and activation of neuron j in layer ℓ
- $z^{\ell}_{j} = \sum_k w^{\ell}_{j,k} a^{\ell-1}_{k} + b^{\ell}_{j}$ and $a^{\ell}_{j} = \sigma(z^{\ell}_{j})$

  $\Rightarrow z^{\ell} = w^{\ell} a^{\ell-1} + b^{\ell}$ and $a^{\ell} = \sigma(z^{\ell})$



[1]

- Goal: compute $\frac{\partial C}{\partial w^{\ell}_{j,k}}$ and $\frac{\partial C}{\partial b^{\ell}_{j}}$ to update parameters
- Here shown for only sigmoid neurons

[1] M. A. Nielsen, Neural Networks and Deep Learning, http://neuralnetworksanddeeplearning.com

Backpropagation – Equations

- Auxiliary term: $\delta_j^\ell = \frac{\partial C}{\partial z_j^\ell}$ is called the *error* of neuron j in layer ℓ (indicator for how far from convergence this neuron is)

Backpropagation – Equations

- Auxiliary term: $\delta_j^\ell = \frac{\partial C}{\partial z_j^\ell}$ is called the *error* of neuron j in layer ℓ (indicator for how far from convergence this neuron is)
- The δ_j^ℓ can be computed recursively, starting from the last layer
  1. $\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L)$ (last layer)
  2. $\delta_j^\ell = \sum_k w_{k,j}^{\ell+1} \delta_k^{\ell+1} \sigma'(z_j^\ell)$

  $\Rightarrow$ „backpropagation“

Backpropagation – Equations

- Auxiliary term: $\delta_j^\ell = \frac{\partial C}{\partial z_j^\ell}$ is called the *error* of neuron j in layer ℓ (indicator for how far from convergence this neuron is)
- The δ_j^ℓ can be computed recursively, starting from the last layer
  1. $\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$ (last layer)
  2. $\delta_j^\ell = \sum_k w_{k,j}^{\ell+1}\delta_k^{\ell+1}\sigma'(z_j^\ell)$

  $\Rightarrow$ „backpropagation“ $\qquad C(a^L) = C(\sigma(z^L))$
- Proof of 1.:

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} \overset{\text{chain rule}}{=} \sum_k \frac{\partial C}{\partial a_k^L} \underbrace{\frac{\partial a_k^L}{\partial z_j^L}}_{=0 \text{ for } k \neq j} = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$$

Backpropagation – Equations

- Auxiliary term: $\delta_j^\ell = \frac{\partial C}{\partial z_j^\ell}$ is called the *error* of neuron j in layer ℓ (indicator for how far from convergence this neuron is)
- The δ_j^ℓ can be computed recursively, starting from the last layer
  1. $\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$ (last layer)
  2. $\delta_j^\ell = \sum_k w_{k,j}^{\ell+1}\delta_k^{\ell+1}\sigma'(z_j^\ell)$

  $\Rightarrow$ „backpropagation“ $z_k^{\ell+1} = \sum_i w_{k,i}^{\ell+1}\sigma(z_i^l) + b_k^{\ell+1}$
- Proof of 2.:

$$\delta_j^\ell = \frac{\partial C}{\partial z_j^\ell} \overset{\text{chain rule}}{=} \sum_k \frac{\partial C}{\partial z_k^{\ell+1}} \frac{\partial z_k^{\ell+1}}{\partial z_j^\ell} = \sum_k \delta_k^{\ell+1} w_{k,j}^{\ell+1}\sigma'(z_j^l)$$

Backpropagation – Equations

- The partial derivatives can be written in terms of δ_j^ℓ
  - $\frac{\partial C}{\partial b_j^\ell} = \delta_j^\ell$
  - $\frac{\partial C}{\partial w_{j,k}^\ell} = a_k^{\ell-1}\delta_j^\ell = a_{\text{in}}\,\delta_{\text{out}}$
- Proof: again chain rule
- Theoretically, δ_j^ℓ and the partial derivatives have to be computed for each training sample (and averaged)

  $\Rightarrow$ Computationally not feasible
- In practice: stochastic gradient descent + parallelization

Backpropagation – Algorithm

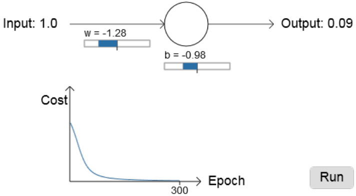
1. For each training sample in a mini-batch:
   - Fix weights and biases and compute weighted input (z_j^ℓ) and activation (a_j^ℓ) of each neuron (forward pass)
   - Compute the error of the last layer (δ_j^L) and backpropagate it through all previous layers to get the remaining δ_j^ℓ (backward pass)
2. Compute partial derivatives (average) and use them to update weights and biases (gradient descent step)
3. Go back to 1. or stop if the algorithm converged

(Python code without any specific libraries is online)

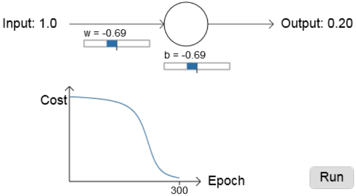
Learning Slowdown

- Small example with sigmoid neuron and squared error
- Single neuron, single sample: input 1, output 0

Initialized with $w = 0.6, b = 0.9$
(initial output: 0.82)



Initialized with $w = 2, b = 2$
(initial output: 0.98)



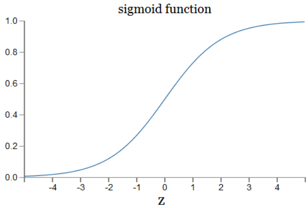
[1]

$\Rightarrow$ Learning can be slow if the initial values are far from the truth!

[1] M. A. Nielsen, Neural Networks and Deep Learning, http://neuralnetworksanddeeplearning.com

Why Does Learning Slow Down?

- Recall:
  - $a = \sigma(z)$ and $z = wx + b$
  - $C = \frac{(y-a)^2}{2}$
- Partial derivatives:
  - $\frac{\partial C}{\partial w} = (a-y)\sigma'(z)x$
  - $\frac{\partial C}{\partial b} = (a-y)\sigma'(z)$



[1]

$\Rightarrow$ Small if $\sigma'(z)$ is small

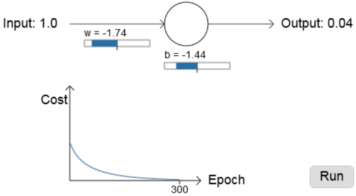
$\Rightarrow$ Little improvement in one gradient descent step

[1] M. A. Nielsen, Neural Networks and Deep Learning, `http://neuralnetworksanddeeplearning.com`

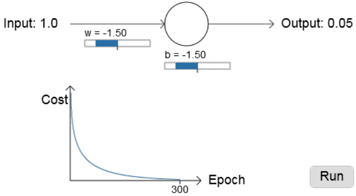
Way Out

- Cross-entropy cost function: $C = y\log(a) + (1-y)\log(1-a)$
- Partial derivatives: $\frac{\delta C}{\delta w} x(a-y)$ and $\frac{\delta C}{\delta b} = (a-y)$

Initialized with $w = 0.6, b = 0.9$
(initial output: 0.82)



Initialized with $w = 2, b = 2$
(initial output: 0.98)



[1]

$\Rightarrow$ No slowdown! (Only if we are already close to the true y)

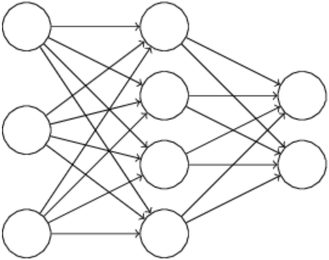
[1] M. A. Nielsen, Neural Networks and Deep Learning, `http://neuralnetworksanddeeplearning.com`

Remarks

- Learning slowdown can happen when neurons saturate
- Often, the flat regions come from exponentials in the activation functions
- The log in the log-likelihood often undoes these exponentials
  $\Rightarrow$ Use output units and cost functions as on Slide 12
- Rectified linear units do not saturate for large weighted inputs
  $\Rightarrow$ This is an advantage compared to sigmoid neurons

Regularization

- Neural networks have many parameters
  - 26 in our small example
  - In reality sometimes millions
- Overfitting is a problem
- Regularization can help to prevent overfitting
- Strategies:
  - Early stopping
  - Penalty
  - Dropout
  - Artificially expanding the data



[1]

[1] M. A. Nielsen, Neural Networks and Deep Learning, http://neuralnetworksanddeeplearning.com

Early Stopping

- Idea: Stop training before convergence
- Weights are usually initialized randomly near zero
  - Sigmoids are roughly linear close to zero
    $\Rightarrow$ Roughly linear model to start from
  - Learning adds nonlinearity where needed
- Stopping early means to shrink the model towards a more linear solution
- When to stop is determined via cross-validation
  - E.g., when the cross-validation error starts to increase

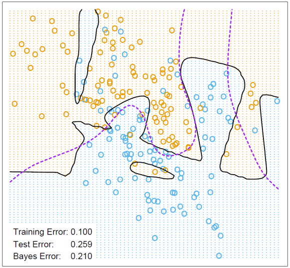
Penalty

- Add a penalty to the cost function to account for model complexity:
  $\widetilde{C}(w,b) = C(w,b) + \lambda J(w,b)$
- Common penalties:
  - Weight decay (L2): $J(w,b) = \sum_{\ell,i,j}(w_{i,j}^{\ell})^2$
  - L1 penalty: $J(w,b) = \sum_{\ell,i,j}|w_{i,j}^{\ell}|$
  - Weight elimination: $J(w,b) = \sum_{\ell,i,j}\frac{(w_{i,j}^{\ell})^2}{1+(w_{i,j}^{\ell})^2}$
- Cross-validation is used to choose λ

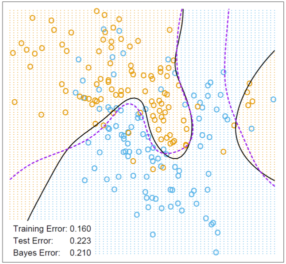
Example – Weight Decay

- Binary classification problem
- Vanilla neural network with 10 hidden neurons

Neural Network - 10 Units, No Weight Decay



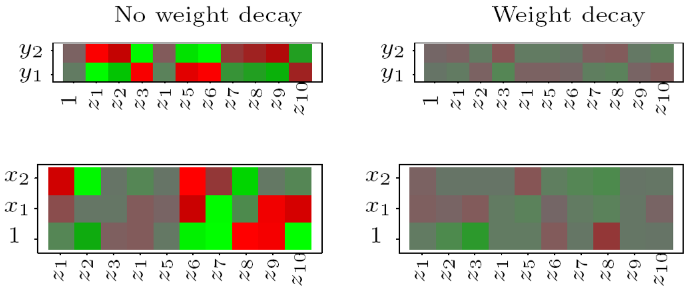
Neural Network - 10 Units, Weight Decay=0.02



$\Rightarrow$ Simpler solution / decision boundary with weight decay

Example – Weight Decay

- Binary classification problem
- Vanilla neural network with 10 hidden neurons

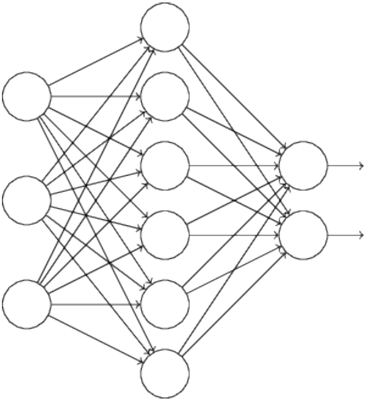


$\Rightarrow$ Simpler solution / decision boundary with weight decay

$\Rightarrow$ Similar but smaller weights

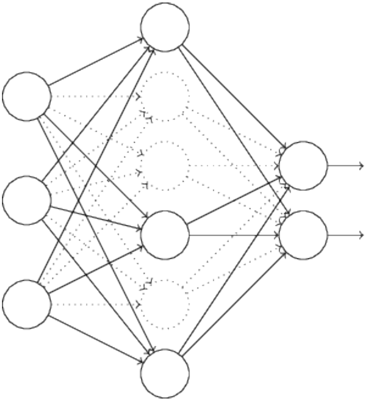
Dropout

- Radically different idea



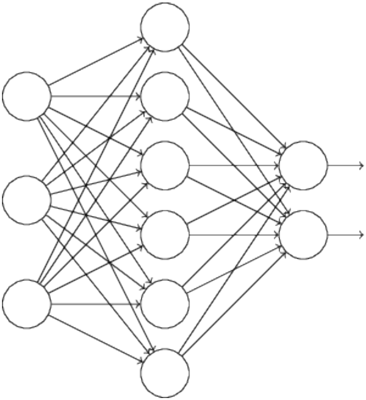
Dropout

- Radically different idea
- Randomly drop half of the hidden neurons for each gradient step



Dropout

- Radically different idea
- Randomly drop half of the hidden neurons for each gradient step
- After training, halve weights outgoing from hidden neurons in the complete network
- Similar to averaging over many networks



Dropout – Heuristic Explanation

> "This technique reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons. It is, therefore, forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons." [4]

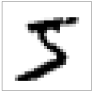
$\Rightarrow$ Robust to the loss of any individual piece of evidence

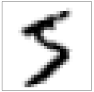
[4] A. Krizhevsky, I. Sutskever, G. Hinton, ImageNet Classification with Deep Convolutional Neural Networks, NIPS 2012

Artificially Expanding the Training Data

- Extremely large amounts of data prevent overfitting
- However, data is often hard to get
- Expand the data by slightly changing existing training samples

  $\Rightarrow$ Simulate real-world variation
- Examples:
  - Rotate / translate images of digits
  - Speed up / slow down speech recordings





[1]

[1] M. A. Nielsen, Neural Networks and Deep Learning, http://neuralnetworksanddeeplearning.com

Summary

- Although there are few to no theoretical guarantees, neural networks perform extremely well on many challenging problems

Summary

- Although there are few to no theoretical guarantees, neural networks perform extremely well on many challenging problems
- Neural networks are trained with backpropagation (stochastic gradient descent)
- Clever choices of neuron types and cost functions can ensure that they learn (reasonably) fast

Summary

- Although there are few to no theoretical guarantees, neural networks perform extremely well on many challenging problems
- Neural networks are trained with backpropagation (stochastic gradient descent)
- Clever choices of neuron types and cost functions can ensure that they learn (reasonably) fast
- Because of their flexibility, they are prone to overfitting
- Regularization can help to prevent this

Summary

- Although there are few to no theoretical guarantees, neural networks perform extremely well on many challenging problems
- Neural networks are trained with backpropagation (stochastic gradient descent)
- Clever choices of neuron types and cost functions can ensure that they learn (reasonably) fast
- Because of their flexibility, they are prone to overfitting
- Regularization can help to prevent this
- There are many design choices ($\Rightarrow$ future sessions)

Thank you for your attention!

Main References:

- Michael A. Nielsen, *Neural Networks and Deep Learning*, Determination Press, 2015.
  `http://neuralnetworksanddeeplearning.com`
- Ian Goodfellow, Yoshua Bengio and Aaron Courville, *Deep Learning*, MIT Press, 2016.
  `http://www.deeplearningbook.org`
- Thomas Lengauer, *Statistical Learning in Computational Biology*, Lecture at Saarland University, Winter Term 2016/17