*** Applied Machine Learning Fundamentals ***
Neural Networks / Deep Learning

Clemens Biehl, Daniel Wehner

SAP SE

Winter term 2019/2020

Find all slides on GitHub

Lecture Overview

Unit I	Machine Learning Introduction
Unit II	Mathematical Foundations
Unit III	Bayesian Decision Theory
Unit IV	Probability Density Estimation
Unit V	Regression
Unit VI	Classification I
Unit VII	Evaluation
Unit VIII	**Classification II**
Unit IX	Clustering
Unit X	Dimensionality Reduction

Agenda for this Unit

Section:

Introduction

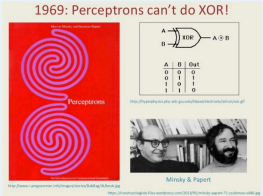
What is Deep Learning?

- 'Deep Learning' is a fancy new term for 'artificial neural networks'
- It is a **supervised** method and **model based**
- Artificial neural networks are inspired by the human brain
- Lots of different architectures exist:
  - **Multi-Layer perceptrons (MLPs)**
  - **Radial Basis Function Networks (RBFNs)**
  - **Convolutional neural networks (CNNs, ConvNets)**
  - **Recurrent neural networks (LSTMs, GRUs, etc.)**
  - **Residual networks (ResNets)**

History of Deep Learning

Early booming (1950s – early 1960s)

F. Rosenblatt suggests the **Perceptron** learning algorithm: `Click here!`



Setback I (mid 1960s – late 1970s)

M. Minsky and *S. Papert* (1969): Serious problems with perceptron algorithm. It cannot learn the **XOR problem**.

History of Deep Learning (Ctd.)

Renewed enthusiasm (1980s)

- New techniques available
- **Backpropagation** for deep nets

Setback II (1990s – mid 2000s)

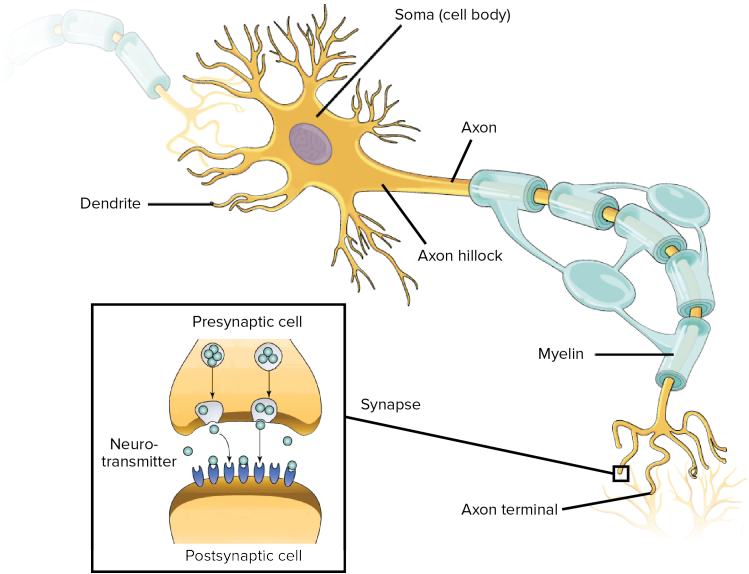
- Other techniques were considered superior (e. g. SVMs)
- CS journals rejected papers on neural networks

'Deep Learning' (since mid 2000)

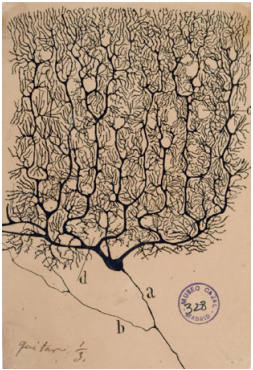
More data, faster computers, better optimization techniques...

Biological Motivation

- All neurons are connected and form a complex **network**
- **Transmitter chemicals** within the fluid of the brain influence the **electrical potential** inside the body of the neurons
- If the **membrane potential** reaches some threshold the neurons **fires** $\Rightarrow$ A pulse of fixed length is sent down the **axon**
- The axon connects the neuron with other neurons (via **synapses**)
- Probably there are 100 trillion (!!!) synapses in the human brain
- **Refractory period** after a neuron has fired


Soma (cell body)
Axon
Dendrite
Axon hillock
Myelin
Presynaptic cell
Synapse
Neuro-
transmitter
Axon terminal
Postsynaptic cell

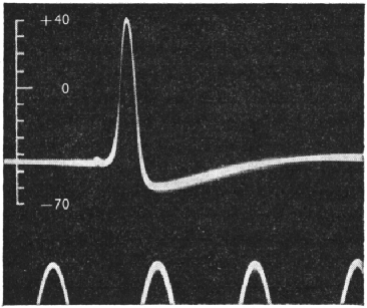
How can we know this?



- *Santiago Ramón y Cajal* made neurons visible by applying **Golgi's method**
- Golgi's method uses the Golgi stain to colorize the neurons
- Cajal establishes the **neuron doctrine** and won the Nobel price in 1906 for his work

How can we know this? (Ctd.)

- End of the 1940s *A. Hodgkin* and *A. Huxley* started investigating the electrical properties of neurons on the squid's[1] axon
- The right-hand-side image was the first **action potential** that was plotted



[1] lat.: *Loligo pealeii*

How do Humans / Animals learn?

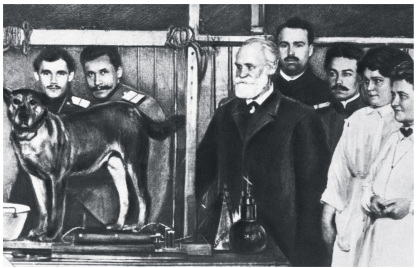
- **Idea:** Mechanism of learning is **association**
- **Hebbian learning:** If the firing of one neuron repeatedly assists in firing another neuron the synaptic connection will be strengthened

*'When an axon of cell A is near enough to excite a cell B and repeatedly or persistently **takes part in firing it**, some growth process or **metabolic change** takes place in one or both cells such that A's **efficiency**, as one of the cells firing B, is **increased**.'*

*'The general idea is an old one, that any two cells or systems of cells that are **repeatedly active at the same** time will tend to become **'associated'**, so that activity in one facilitates activity in the other.'* *Hebb*

Classical / Pavlovian Conditioning

- Dog salivates when given food
- Food is an **unconditioned stimulus (US)**
- Salivation in response to food is **unconditioned response (UR)**
- Food is paired with the sound of a bell
- Bell is **conditioned stimulus (CS)**
- Bell will eventually elicit salivation event without food
- Salivation is **conditioned response (CR)**

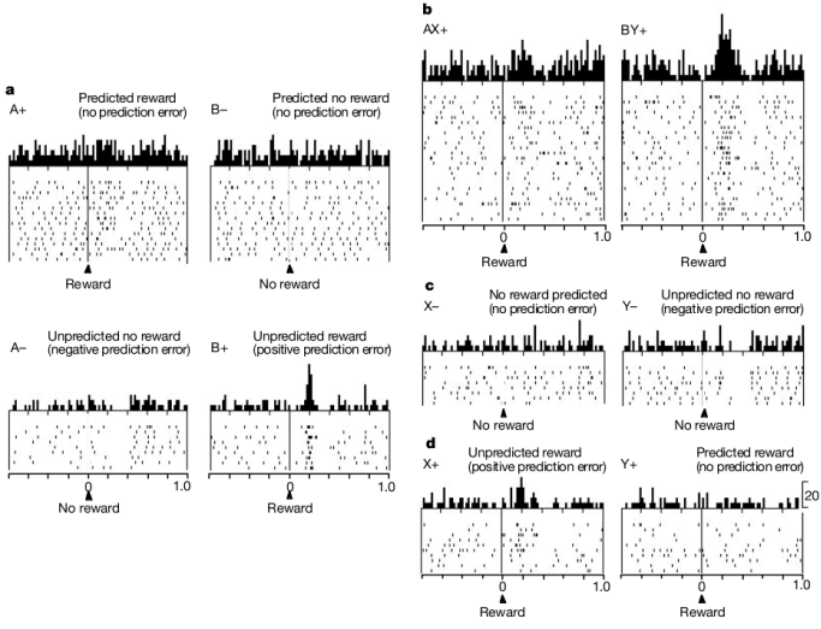


Classical / Pavlovian Conditioning (Ctd.)

Blocking

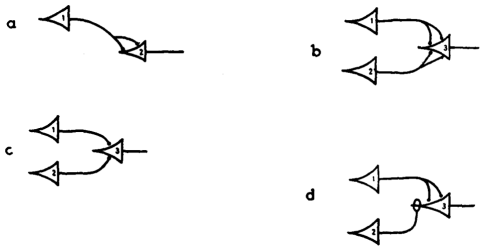
Group A	train N+	train LN+	test L-	$\Rightarrow$ no conditioning
Group B		train LN+	test L-	$\Rightarrow$ conditioning

- CS is a light (L), a noise (N), or a combination of both (LN)
- US is a mild shock that is paired with the CS in the training phase (+)
- In all conditions, after training fear response is tested when only L is presented without shock (-)
- Group B shows conditioning; Group A does not: **N blocks L**
- This is hard to explain with Hebbian learning
- **Idea:** Learning only happens if there is a **prediction error**

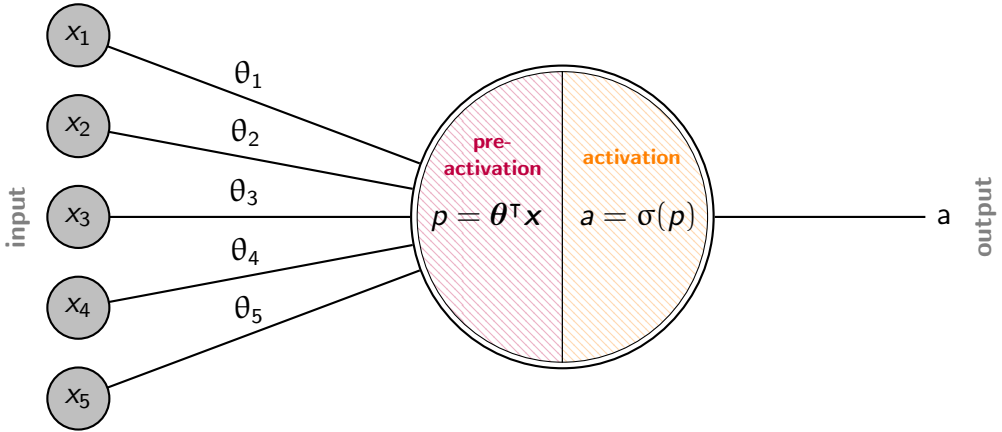

a
A+
Predicted reward
(no prediction error)
Reward
B–
Predicted no reward
(no prediction error)
No reward
A–
Unpredicted no reward
(negative prediction error)
0
1.0
No reward
B+
Unpredicted reward
(positive prediction error)
0
1.0
Reward
b
AX+
0
1.0
Reward
BY+
0
1.0
Reward
c
X–
No reward predicted
(no prediction error)
No reward
Y–
Unpredicted no reward
(negative prediction error)
No reward
d
X+
Unpredicted reward
(positive prediction error)
0
1.0
Reward
Y+
Predicted reward
(no prediction error)
20
0
1.0
Reward

Artificial Neurons [McCulloch and Pitts 1943]

- 1943 *W. S. McCulloch* and *W. H. Pitts* designed the first 'artificial neuron'
- These neurons can represent logical functions (e. g. b – `OR`, c – `AND`)



Perceptron [Rosenblatt 1957]



Perceptron (Ctd.)

- The neuron receives an input-vector $\boldsymbol{x}$

$$\boldsymbol{x} = (x_1, x_2, \ldots, x_m)^\intercal$$

- Each input signal is weighted by a factor[2] θ_j

$$\boldsymbol{\theta} = (\theta_1, \theta_2, \ldots, \theta_m)^\intercal$$

- We compute the **pre-activation** and the **activation**:

$$p = \boldsymbol{\theta}^\intercal \boldsymbol{x} + b = \sum_{j=1}^{m} \theta_j x_j + b \qquad a = \sigma(p) \qquad (1)$$

[2] weight of synaptic strength

Perceptron (Ctd.)

- The simplest activation function is to use a threshold ρ:[3]

$$\sigma(p) = \begin{cases} 0 & \text{for } p \leqslant \rho \\ 1 & \text{for } p > \rho \end{cases}$$

- Quick example: $\boldsymbol{x} = (1, 0, 0.5)^\intercal; \boldsymbol{\theta} = (1, -0.5, -1)^\intercal; \rho = 0$

$$p = \sum_{j=1}^{3} \theta_j x_j = 1 \cdot 1 + (-0.5) \cdot 0 + (-1) \cdot 0.5 = 0.5$$

$$a = \sigma_{\rho=0}(0.5) = 1$$

[3] Not used, since not differentiable; alternatives later

Perceptron Learning

- Learning means choosing the correct weights $\boldsymbol{\theta}^*$ from a set of possible hypotheses $\mathcal{H}$ (**hypothesis space**):

$$\mathcal{H} = \{\boldsymbol{\theta} | \boldsymbol{\theta} \in \mathbb{R}^m\}$$

- How to learn the weights from a data set $\mathcal{D}$?
- **Algorithm outline:**
  1. Pick a training example $\boldsymbol{x} \in \mathcal{D}$
  2. Calculate the activation a for that training example
  3. Update the weights θ based on the error

Perceptron Learning (Ctd.)

- Let the error be denoted by δ
- How can we compute the error? We need a loss function $\mathcal{J}(\boldsymbol{\theta})$:

$$\mathcal{J}(\boldsymbol{\theta}) = \frac{1}{2} \sum_{i=1}^{N} (h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)})^2 \tag{2}$$

- Again, we use **gradient descent**: Compute gradient and go into the negative direction:

$$\boldsymbol{\theta}^{(t+1)} \longleftarrow \boldsymbol{\theta}^{(t)} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}) \tag{3}$$

$\Rightarrow$ cf. slides 'Regression'

Algorithm 1: Perceptron Learning Algorithm

Input: Training data $\mathcal{D}$, convergence threshold ε

```
// initialization
```

set all weights $\boldsymbol{\theta}^{(0)}$ to small random numbers

for $t \in \{0, 1, \ldots, \infty\}$ **do**

  pick a sample $\langle \boldsymbol{x}, y \rangle \in \mathcal{D}$ randomly

  ```
  // predict the class label
  ```

  compute the activation $a = \sigma(\boldsymbol{\theta}^{\intercal} \boldsymbol{x})$

  ```
  // stochastic gradient descent:  update based on prediction error
  ```

  $\boldsymbol{\theta}^{(t+1)} \longleftarrow \boldsymbol{\theta}^{(t)} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta})$

  if $\|\boldsymbol{\theta}^{(t+1)} - \boldsymbol{\theta}^{(t)}\| \leqslant \varepsilon$ **then**

    ```
    // convergence
    ```

    break

return $\boldsymbol{\theta}$

Perceptron Convergence Theorem

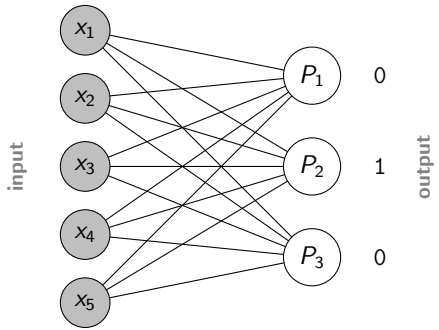
Perceptron Convergence Theorem

If the training data is **linearly separable**, then the perceptron learning algorithm is going to **converge after a finite amount of time** and classifies **all training data examples correctly**.

Generalization to multiple Classes

- A single neuron can only distinguish two classes
- If there are more than two classes: Simply use more perceptrons[4]
- Use **one-hot encoding** for the classes and **soft-max** as activation function (later)
- Example for three classes:

$\mathcal{C}_1$	1 0 0
$\mathcal{C}_2$	0 1 0
$\mathcal{C}_3$	0 0 1

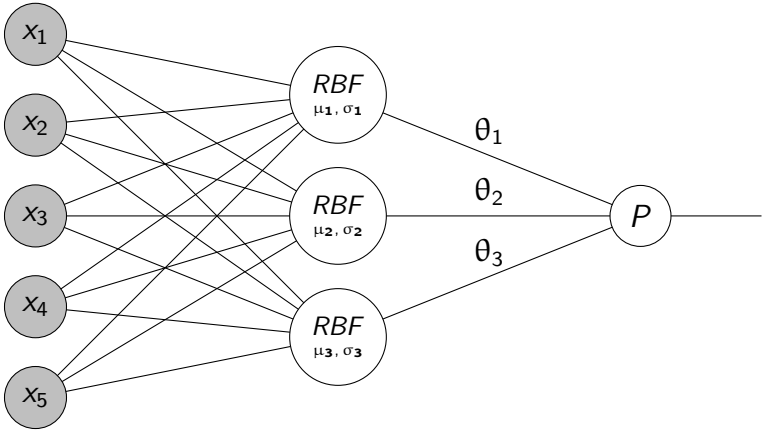


[4] This construct is still referred to as a perceptron.

What about non-linear Data Sets?

- If the data is not linearly separable then the perceptron cannot learn it
- Remember Marvin Minsky's/Seymour Papert's book *'Perceptrons'*
- What can we do?
  1. Add feature mapping $\Rightarrow$ **Radial basis function (RBF) networks**
  2. Add hidden layers $\Rightarrow$ **Multi-layer perceptrons (MLP)**

Radial Basis Function (RBF) Networks

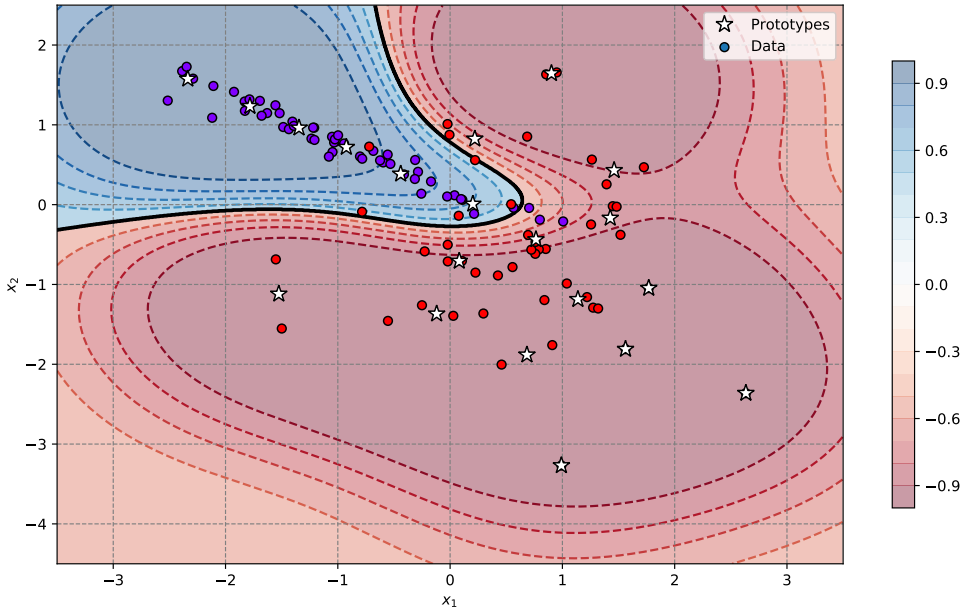


Inside an RBF Neuron

- Each RBF neuron computes the distance of the input $\boldsymbol{x}$ to an **internal prototype** vector:

$$\varphi_j(\boldsymbol{x}) = \exp\left\{-\frac{\|\boldsymbol{x} - \boldsymbol{z_j}\|^2}{2\sigma^2}\right\} \tag{4}$$

- These distances are then fed into the perceptron
- **How to get the prototypes?**
  - We can use clustering, e. g. **k-Means**, to get the locations $\boldsymbol{\mu}$ (this avoids useless prototypes in areas without data points)
  - Choose σ neither too small nor too big
- There are no weights $\boldsymbol{\theta}$ to tune for that neuron

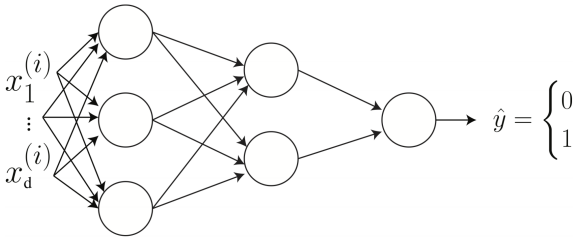

Prototypes
Data
x_1
x_2
2
1
0
−1
−2
−3
−4
−3
−2
−1
0
1
2
3
0.9
0.6
0.3
0.0
−0.3
−0.6
−0.9

Section:

Multi-Layer-Perceptrons (MLPs)

Overview: Multi-Layer Perceptron Representation

- A Multi-Layer Perceptron (MLP) can approximate any continuous function arbitrarily well (universal function approximator), given enough hidden units
- Hidden layers of the network can learn useful feature representations



Overview: Multi-Layer Perceptron Representation

- An MLP with L hidden layers is a function:
  - $f : \mathcal{R}^n \rightarrow \mathcal{R}^m$
  - with parameter matrices $\boldsymbol{\Theta}^{(1)}$, $\boldsymbol{\Theta}^{(2)}$, $\ldots$, $\boldsymbol{\Theta}^{(L)}$
  - and non-linearities $g^{(1)}$, $g^{(2)}$, $\ldots$, $g^{(L)}$
  - where:

$$
\begin{aligned}
z^{(1)} &= \boldsymbol{\Theta}^{(1)} \boldsymbol{x} \\
z^{(2)} &= \boldsymbol{\Theta}^{(2)} g^{(1)}(\boldsymbol{z}^{(1)}) \\
&\ldots \\
z^{(L)} &= \boldsymbol{\Theta}^{(L-1)} g^{(L-1)}(\boldsymbol{z}^{(L-1)}) \\
\boldsymbol{h} &= g^{(L)}(\boldsymbol{z}^{(L)})
\end{aligned}
$$

Overview: Multi-Layer Perceptron Inference

- The forward pass (prediction) of an MLP can be computed as follows:

$$h_k(\boldsymbol{x}^{(i)}; \boldsymbol{\Theta}) = g^{(2)}\left(\sum_{l=0}^{h} \Theta_{kl}^{(2)} g^{(1)}\left(\sum_{j=0}^{m} \Theta_{lj}^{(1)} x_j^{(i)}\right)\right)$$

- This provides us with an output value for each class k, we could predict the class with the highest value $y = \underset{k}{\operatorname{argmax}}\, \boldsymbol{h}$
- How can we obtain the optimal weights $\boldsymbol{\Theta}$?

Overview: Multi-Layer Perceptron Learning

- We can optimize the parameters using gradient descent and a cost function
- The goal is to minimize the cost, e. g. $\mathcal{J}(\boldsymbol{\Theta}) = \frac{1}{2n} \sum_{i=1}^{n} (h_{\boldsymbol{\Theta}}(\widehat{\boldsymbol{x}}^{(i)}) - y^{(i)})^2$
- `Repeat until convergence {`
  $\boldsymbol{\Theta}^{(t+1)} \longleftarrow \boldsymbol{\Theta}^{(t)} - \alpha \nabla_{\boldsymbol{\Theta}} \mathcal{J}(\boldsymbol{\Theta}^{(t)})$ *// simultaneously update all* θ_j
  `}`
- How do we get the gradient $\nabla_{\boldsymbol{\Theta}} \mathcal{J}(\boldsymbol{\Theta})$ with respect to all weights $\boldsymbol{\Theta}_{\boldsymbol{ij}}^{(\text{layer})}$?

Overview: Multi-Layer Perceptron Learning

- We start by randomly initializing the network weights
- Do not set an initial value of 0 for all weights, as this will lead to problems in backpropagation (the gradients will all be the same), but initialize to small random numbers, e. g. by sampling from $\mathcal{N}(0, 0.1)$
- We then perform a forward pass through the network, i. e. make predictions on our training set
- Using the predictions, we compute a scalar loss value $J(\boldsymbol{\Theta})$
- We calculate the gradients of the loss with respect to each network parameter and update all network parameters using gradient descent

Backpropagation

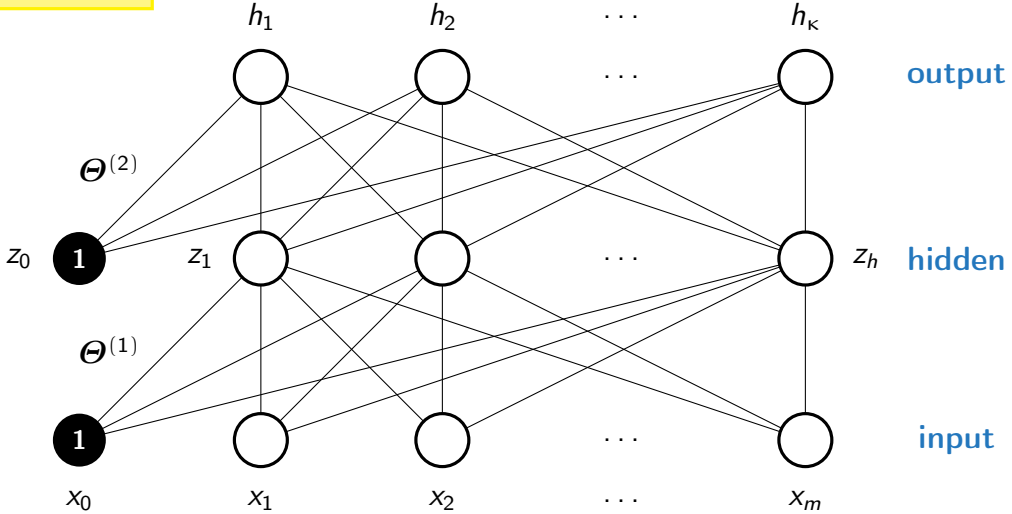
- In order to update the weights, we first have to perform a **forward pass**:

$$h_k(\boldsymbol{x}^{(i)}; \boldsymbol{\Theta}) = g^{(2)}\left(\sum_{l=0}^{h} \Theta_{kl}^{(2)} g^{(1)}\left(\sum_{j=0}^{m} \Theta_{lj}^{(1)} x_j^{(i)}\right)\right)$$

$$z_l = g^{(1)}\left(\sum_{j=0}^{m} \Theta_{lj}^{(1)} x_j^{(i)}\right) \qquad \text{activation}$$

- $g(\cdot) \equiv$ activation function, e. g. sigmoid, tanh, ReLU
- $\boldsymbol{\Theta}$ are the network parameters (to be learned)

This is a fully connected neural network



Backpropagation (Ctd.)

- Compute the network loss
- The loss function is given by: (assume square loss: $\ell = (h_k(\boldsymbol{x}^{(i)}; \boldsymbol{\Theta}) - y_k^{(i)})^2$)

$$\mathcal{J}^{(i)}(\boldsymbol{\Theta}) = \frac{1}{n} \sum_{i=1}^{n} \sum_{k=1}^{\kappa} \ell(h_k(\boldsymbol{x}^{(i)}; \boldsymbol{\Theta}), y_k^{(i)})$$

- Compute the error gradient w. r. t. $h_k(\boldsymbol{x}^{(i)}; \boldsymbol{\Theta})$:

$$\frac{\partial \mathcal{J}^{(i)}(\boldsymbol{\Theta})}{\partial h_k(\boldsymbol{x}^{(i)}; \boldsymbol{\Theta})} = \ell'(h_k(\boldsymbol{x}^{(i)}; \boldsymbol{\Theta}), y_k^{(i)}) \equiv \delta_k^{(i)}$$

Backpropagation (Ctd.)

- Compute the weight gradient for the output layer:

$$\begin{aligned}\frac{\partial \mathcal{J}^{(i)}(\boldsymbol{\Theta})}{\partial \Theta_{kl}^{(2)}} &= \frac{\partial \mathcal{J}(\boldsymbol{\Theta})}{\partial h_k(\boldsymbol{x}^{(i)}; \boldsymbol{\Theta})} \frac{\partial h_k(\boldsymbol{x}^{(i)}; \boldsymbol{\Theta})}{\partial \Theta_{kl}^{(2)}} \\ &= \ell'(h_k(\boldsymbol{x}^{(i)}; \boldsymbol{\Theta}), y_k^{(i)}) \cdot g'^{(2)} \left(\sum_{t=0}^{h} \Theta_{kt}^{(2)} z_t(\boldsymbol{x}^{(i)}) \right) \cdot z_l(\boldsymbol{x}^{(i)}) \\ &= \delta_k^{(i)} \cdot g'^{(2)} \left(\sum_{t=0}^{h} \Theta_{kt}^{(2)} z_t(\boldsymbol{x}^{(i)}) \right) \cdot z_l(\boldsymbol{x}^{(i)})\end{aligned}$$

Backpropagation (Ctd.)

- Compute the error gradient for the hidden layer:

$$\begin{aligned}
\frac{\partial \mathcal{J}^{(i)}(\boldsymbol{\Theta})}{\partial z_l} &= \sum_{k=1}^{K} \frac{\partial \mathcal{J}^{(i)}(\boldsymbol{\Theta})}{\partial h_k(\boldsymbol{x}^{(i)}; \boldsymbol{\Theta})} \frac{\partial h_k(\boldsymbol{x}^{(i)}; \boldsymbol{\Theta})}{\partial z_l} \\
&= \sum_{k=1}^{K} \ell'(h_k(\boldsymbol{x}^{(i)}; \boldsymbol{\Theta}), y^{(i)}) \cdot g'^{(2)} \left(\sum_{t=0}^{h} \Theta_{kt}^{(2)} z_t(\boldsymbol{x}^{(i)}) \right) \cdot \Theta_{kl}^{(2)} \\
&= \sum_{k=1}^{K} \delta_k^{(i)} \cdot g'^{(2)} \left(\sum_{t=0}^{h} \Theta_{kt}^{(2)} z_t(\boldsymbol{x}^{(i)}) \right) \cdot \Theta_{kl}^{(2)} \equiv \widehat{\delta}_l^{(i)}
\end{aligned}$$

Backpropagation (Ctd.)

- Compute the weight gradient for the hidden layer:

$$\frac{\partial \mathcal{J}^{(i)}(\boldsymbol{\Theta})}{\partial \Theta_{lj}^{(1)}} = \frac{\partial \mathcal{J}^{(i)}(\boldsymbol{\Theta})}{\partial z_l} \cdot g'^{(1)} \left(\sum_{t=0}^{m} \Theta_{jt}^{(1)} x_j^{(i)} \right) \cdot x_j^{(i)}$$

$$= \widehat{\delta}_l^{(i)} \cdot g'^{(1)} \left(\sum_{t=0}^{m} \Theta_{jt}^{(1)} x_j^{(i)} \right) \cdot x_j^{(i)}$$

- The weight derivatives are now used in the gradient descent update rule

Remarks

- As a sanity check, we can perform numeric gradient checking: Based on the fact that $f'(x) \approx \frac{1}{v} \cdot (f(x+v) - f(x))$, we can evaluate the loss function for slightly different weights and compare the difference to the gradient we computed using backprop
- There are a lot of hyperparameters for backprop and gradient descent (number of hidden layers, hidden layer dimensionality, activation functions, learning rate, batch size, number of training epochs, regularization, etc.) - you need to empirically find the best ones for your problem
- Play with some hyperparameters here: `https://playground.tensorflow.org`

Sigmoid Function

- $\sigma(x) = \frac{1}{1+e^{-x}}$
- $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
- Value range 0 to 1
- Problem: Gradient can become very small (vanishing gradient), in contrast to large gradients it's not clear how to clip the gradient on the lower end
- Problem: Output is not zero-centered (sigmoid output is always positive, gradients will be all positive or all negative which makes optimization harder)

Sigmoid Function

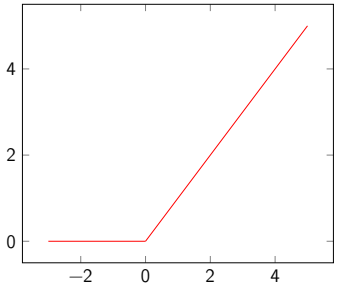
- *Convergence is usually faster if the average of each input variable over the training set is close to zero. To see this, consider the extreme case where all the inputs are positive. Weights to a particular node in the first weight layer are updated by an amount proportional to δx where δ is the (scalar) error at that node and x is the input vector (see equations (5) and (10)). When all of the components of an input vector are positive, all of the updates of weights that feed into a node will have the same sign (i.e. sign(δ)). As a result, these weights can only all decrease or all increase together for a given input pattern. Thus, if a weight vector must change direction it can only do so by zigzagging which is inefficient and thus very slow.* - Yann LeCun et al., Efficient BackProp, 1998

tanh

- $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- $\tanh'(x) = 1 - \tanh(x)^2$
- Value range -1 to 1
- Zero-centered
- Still suffers from the vanishing gradient problem

Rectified Linear Unit (ReLU)

- $ReLU(x) = max(0, x)$
- $ReLU'(x) = \begin{cases} 0 & x < 0 \\ 1 & x > 0 \end{cases}$



Rectified Linear Unit (ReLU)

- ReLU not lead to vanishing gradients
- ReLU is very efficient to compute
- Use ReLU as the hidden layer activation function!
- But: Pay attention to the initialization of your parameters to avoid *"dying ReLUs"* (parameter settings where single neurons will always output 0)

Softmax

- $softmax_i(x) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$
- $\frac{\partial s_i}{\partial a_j} = \begin{cases} s_i(1 - s_j) & i = j \\ -s_i \cdot s_j & i \neq j \end{cases}$
- Used to squash the last layer's activations into a probability distribution

Section:

Further Network Architectures

Convolutional Neural Networks

Motivation

- A large fully-connected MLP network can have a lot of parameters $\rightarrow$ it might be too complex / overfit or be computationally inefficient for some tasks
- For some problems the input position may not matter in every case (e. g. when classifying emails as spam or not spam, we don't care if the word *the* is at input position 3 or 5)
- The MLP always needs a fixed-size input vector - but we might have variable-size input data (e. g. images in different resolutions, text sequences of different lengths, etc.)

Convolutional Neural Networks

Idea

- *"A convolutional neural network is designed to identify indicative local predictors in a large structure, and combine them to produce a fixed size vector representation of the structure, capturing these local aspects that are most informative for the prediction task at hand."* - Yoav Goldberg

Convolutional Neural Networks

Representation

- A CNN usually consists of multiple convolutional layers
- A convolutional layer consists of a convolution (a.k.a. filter), a non-linear activation function and a pooling operation
- Pooling extracts the most important features independent of their input position

Convolutional Neural Networks

Representation

- The convolution operation:

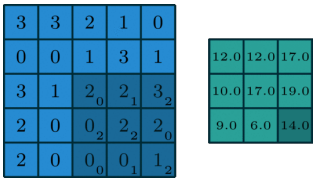
$$(f \star g)[i] = \sum_{w=-W}^{W} f[i-w]g[w]$$

  - $\star$ is the convolution operator,
  - f is the input to the convolutional layer,
  - i is the current position in the input,
  - W is the filter size / window size,
  - g is the filter (a.k.a. *kernel*)

Convolutional Neural Networks

Representation

- The convolution operation: $(f \star g)[i] = \sum_{n=-N}^{N} f[i-n]g[n]$
- We stride a filter across the input data (e. g. image pixels) with a certain stride size and multiply the input with the filter weights at each local position



Convolutional Neural Networks

Representation

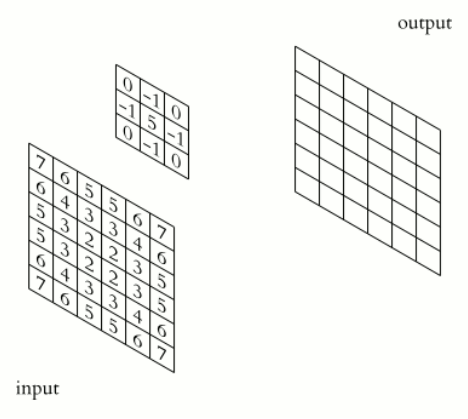
- The outputs of the convolutions are passed through non-linear activation functions
- Pooling is applied to extract only the most important activations:
  - If $c_1, c_2, \ldots, c_N \in \mathcal{R}$ are the outputs of the convolution, a *max pooling* operation will output: $\max_i c_i$
- There are no parameters / weights involved in the pooling

Convolutional Neural Networks

Remarks

- Usually many (hundreds or thousands) filters are applied simultaneously
- We need to choose the number and size of filters, the stride (step-size) with which to slide them over the input, the activation functions, the pooling (max, k-max, mean)
- Multiple convolutional layers can be applied subsequently to extract useful structures from the data (e. g. detecting edges, substructures)
- Sparse connectivity and parameter-sharing make CNNs very efficient, computation can be parallelized

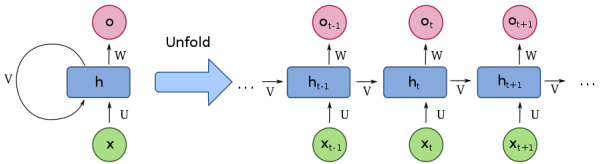
Convolution Animation



Recurrent Neural Networks

Overview

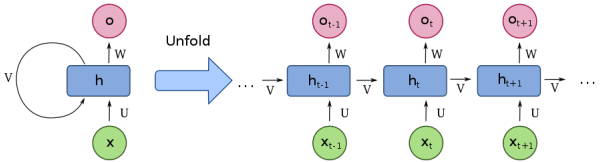
- Recurrent Neural Networks (RNNs) can be used to process sequences (e. g. for sequence labelling tasks like named entity recognition, for sequence transduction tasks like machine translation, for sequence classification, etc.)
- They are similar to feed-forward networks, but have a recurrent loop in the computational graph



Recurrent Neural Networks

Representation

$$\boldsymbol{h_t} = \sigma_h(\boldsymbol{U}\boldsymbol{x} + \boldsymbol{V}\boldsymbol{h}_{t-1} + \boldsymbol{b}_h)$$
$$\boldsymbol{o_t} = \sigma_o(\boldsymbol{W}\boldsymbol{h_t} + \boldsymbol{b}_o)$$



Recurrent Neural Networks

Extensions

- Bi-directionality: Run the RNN from left-to-right and right-to-left and concatenate the hidden states $\overrightarrow{h_t}$ and $\overleftarrow{h_t}$ for both directions
- Gating: Gated Recurrent Units (GRUs) and Long-Short Term Memory (LSTM) Networks add additional parameters to RNNs to better control what is stored in the hidden state h and to prevent vanishingly small gradients for long sequences
- Skip-connections through time
- Attention (re-expressing inputs and outputs in terms of a weighted combination with the other inputs)

Recurrent Neural Networks

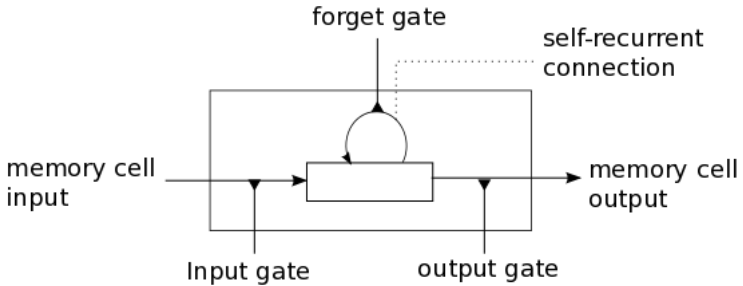
Extensions

- Long-Short Term Memory Network (Hochreiter, Schmidhuber, 1997):
  - Input gate $\boldsymbol{i}_t = \sigma(\boldsymbol{W}_i \boldsymbol{x}_t + \boldsymbol{U}_i \boldsymbol{h}_{t-1})$
  - Forget gate $\boldsymbol{f}_t = \sigma(\boldsymbol{W}_f \boldsymbol{x}_t + \boldsymbol{U}_f \boldsymbol{h}_{t-1})$
  - Output gate $\boldsymbol{o}_t = \sigma(\boldsymbol{W}_o \boldsymbol{x}_t + \boldsymbol{U}_o \boldsymbol{h}_{t-1})$
  - New memory cell $\widetilde{\boldsymbol{c}_t} = \tanh(\boldsymbol{W}_c \boldsymbol{x}_t + \boldsymbol{U}_c \boldsymbol{h}_{t-1})$
  - Final memory cell $\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \widetilde{\boldsymbol{c}_t}$
  - Final hidden state $\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t)$
- σ denotes the sigmoid activation function on this slide, $\odot$ is element-wise mutliplication (a.k.a. Hadamard product)

Recurrent Neural Networks

Extensions

- Long-Short Term Memory Network (Hochreiter, Schmidhuber, 1997):



Section:

Wrap-Up

Summary

- Neural Networks are powerful models for pattern recognition
- The Perceptron can classify all training examples correctly iff the training data is linearly separable
- Multi-Layer Perceptrons are universal function approximators
- Backpropagation is a recursive procedure based on the chain rule of calculus to obtain the gradients for neural network learning
- Different neural network architectures like CNNs and RNNs exist for solving different kinds of problems

Self-Test Questions

1. What is the relation between neural networks and logistic regression?
2. What is a perceptron? Which problems can it solve and which not?
3. Why do we often use multiple layers used instead of a simple Perceptron?
4. How does backpropagation work? How does a neural network learn?
5. What are advantages and disadvantages of using neural networks?
6. What are CNNs and RNNs? For which tasks are they suitable?

What's next...?

Unit I	Machine Learning Introduction
Unit II	Mathematical Foundations
Unit III	Bayesian Decision Theory
Unit IV	Probability Density Estimation
Unit V	Regression
Unit VI	Classification I
Unit VII	Evaluation
Unit VIII	**Classification II**
Unit IX	Clustering
Unit X	Dimensionality Reduction

Recommended Literature and further Reading

[1] Deep Learning
Ian Goodfellow et al. MIT Press. 2016.
→ Link, cf. chapters 6 *Deep Feedforward Networks*, especially chapter 6.5

[2] Pattern Recognition and Machine Learning
Christopher Bishop. Springer. 2006.
→ Link, cf. chapter 5 *Neural Networks*, especially chapter 5.3

[3] Backpropagation calculus
Grant Saunderson. YouTube. 2017.
→ Link

Thank you very much for the attention!

Topic: *** Applied Machine Learning Fundamentals *** Neural Networks / Deep Learning
Term: Winter term 2019/2020

Contact:
Clemens Biehl
Moodle Forum

Do you have any questions?