(Feed-Forward) Neural Networks

KGA

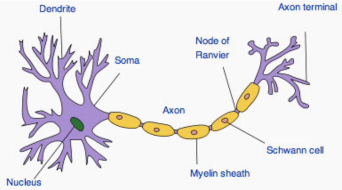
2016-12-06

Dr. Hajira Jabeen, Prof. Jens Lehmann

Outline

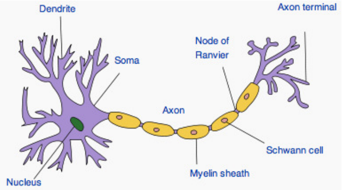
- In the previous lectures we have learned about tensors and factorization methods.
- RESCAL is a bilinear model for SRL that can be formulated as tensor factorization problem.
- Furthermore, we learned about optimization techniques which can be applied for learning score-based models.
- Today we will learn about a new class of algorithms which can be applied for SRL, namely neural networks.
- We will see how to apply them for SRL in the following lecture.

Artificial Neurons



source: Wikipedia

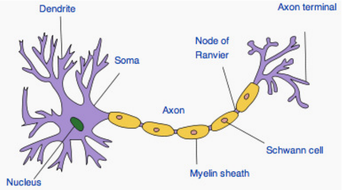
Artificial Neurons



source: Wikipedia

- There are many different types of neurons in the nervous system, and they are quite complicated.

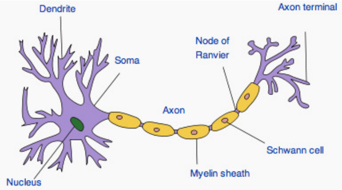
Artificial Neurons



source: Wikipedia

- There are many different types of neurons in the nervous system, and they are quite complicated.
- Neurons are connected to each other with synapses. Thus, they form a network. This complicates things even more.

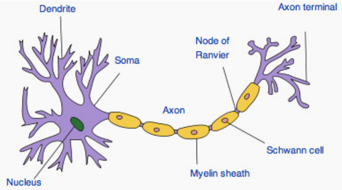
Artificial Neurons



source: Wikipedia

- There are many different types of neurons in the nervous system, and they are quite complicated.
- Neurons are connected to each other with synapses. Thus, they form a network. This complicates things even more.
- How to understand this system?
  Basic idea: reduce the neuron to its essentials.

Artificial Neurons



source: Wikipedia

- There are many different types of neurons in the nervous system, and they are quite complicated.
- Neurons are connected to each other with synapses. Thus, they form a network. This complicates things even more.
- How to understand this system?
  Basic idea: reduce the neuron to its essentials.
- There are a number of greatly simplified neuron models. One of the simplest models is by McCulloch and Pitts.

Artificial Neurons

- A neuron receives input from a number of other neurons.

Artificial Neurons

- A neuron receives input from a number of other neurons.
- These inputs come in the form of spikes – short pulses of electrical current. We average these spikes over time and represent them with a single number: the **spike frequency** ν.

Artificial Neurons

- A neuron receives input from a number of other neurons.
- These inputs come in the form of spikes – short pulses of electrical current. We average these spikes over time and represent them with a single number: the **spike frequency** ν.
- The spikes arrive at the neuron's membrane and alter the electrical potential at this point. For each neuron we keep track of its **membrane potential** u.

Artificial Neurons

- A neuron receives input from a number of other neurons.
- These inputs come in the form of spikes – short pulses of electrical current. We average these spikes over time and represent them with a single number: the **spike frequency** ν.
- The spikes arrive at the neuron's membrane and alter the electrical potential at this point. For each neuron we keep track of its **membrane potential** u.
- There are excitatory and inhibitory connections between neurons, and they can be of different strength. We model the effect of a connection on the membrane potential as the product of the **synaptic weight** w with the spike frequency.

Artificial Neurons

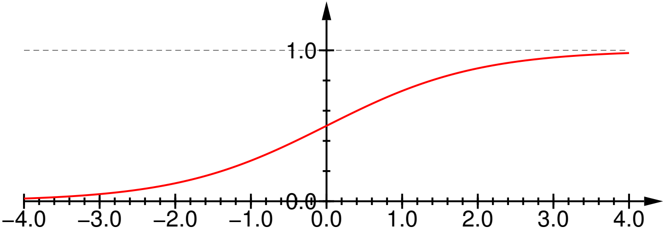
- A neuron receives input from a number of other neurons.
- These inputs come in the form of spikes – short pulses of electrical current. We average these spikes over time and represent them with a single number: the **spike frequency** ν.
- The spikes arrive at the neuron's membrane and alter the electrical potential at this point. For each neuron we keep track of its **membrane potential** u.
- There are excitatory and inhibitory connections between neurons, and they can be of different strength. We model the effect of a connection on the membrane potential as the product of the **synaptic weight** w with the spike frequency.
- The weight is positive for excitatory and negative for inhibitory connections.

Artificial Neurons

- A neuron receives input from a number of other neurons.
- These inputs come in the form of spikes – short pulses of electrical current. We average these spikes over time and represent them with a single number: the **spike frequency** ν.
- The spikes arrive at the neuron's membrane and alter the electrical potential at this point. For each neuron we keep track of its **membrane potential** u.
- There are excitatory and inhibitory connections between neurons, and they can be of different strength. We model the effect of a connection on the membrane potential as the product of the **synaptic weight** w with the spike frequency.
- The weight is positive for excitatory and negative for inhibitory connections.
- The absolute value of the weight is small for weak connections and large for strong connections.

Artificial Neurons

When the neuron's membrane potential exceeds a threshold then the neuron emits a spike (which can propagate to multiple receivers) and resets its membrane potential. The spike frequency as a function of the incoming power is a non-linear **transfer function** (also simply called **non-linearity**):



We call such a function a **sigmoid** or **sigmoidal function**. The standard formula is

$$\nu = \sigma(u) = \frac{1}{1 + e^{-u}} \quad .$$

Artificial Neurons

- Now assume we have a pool of neurons, numbered $1, \ldots, n$. Let u_i be the membrane potential and ν_i be the firing rate of neuron number i, and let w_{ji} be the synaptic weight of the connection from i to j (which is zero if the neurons are not connected).

Artificial Neurons

- Now assume we have a pool of neurons, numbered $1, \ldots, n$. Let u_i be the membrane potential and ν_i be the firing rate of neuron number i, and let w_{ji} be the synaptic weight of the connection from i to j (which is zero if the neurons are not connected).
- Then we arrive at the model

$$
\begin{aligned}
u_j &\leftarrow \sum_{i=1}^{n} w_{ji} \cdot \nu_i \\
\nu_j &\leftarrow \sigma(u_j) \quad .
\end{aligned}
$$

Artificial Neurons

- Now assume we have a pool of neurons, numbered $1, \ldots, n$. Let u_i be the membrane potential and ν_i be the firing rate of neuron number i, and let w_{ji} be the synaptic weight of the connection from i to j (which is zero if the neurons are not connected).
- Then we arrive at the model

$$u_j \leftarrow \sum_{i=1}^{n} w_{ji} \cdot \nu_i$$
$$\nu_j \leftarrow \sigma(u_j) \quad .$$

- This model draws inspiration from biology. However, it is so abstract that in the end it has little in common with its biological counterpart. It should rather be viewed as a computational unit in a mathematical learning machine.

Artificial Neurons

- The relation $u_j \leftarrow \sum_{i=1}^{n} w_{ji} \cdot \nu_i$ is familiar: if ν_i are the inputs, then this is a linear function, which can be understood as a Perceptron model.

Artificial Neurons

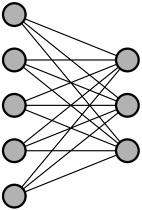
- The relation $u_j \leftarrow \sum_{i=1}^{n} w_{ji} \cdot \nu_i$ is familiar: if ν_i are the inputs, then this is a linear function, which can be understood as a Perceptron model.
- The sigmoid does not change the way decisions are made: $\nu_j \leftarrow \sigma\left(\sum_{i=1}^{n} w_{ji} \cdot \nu_i\right)$. The threshold changes to $1/2$ (instead of zero), but the decision boundary remains the same.

Artificial Neurons

- The relation $u_j \leftarrow \sum_{i=1}^{n} w_{ji} \cdot \nu_i$ is familiar: if ν_i are the inputs, then this is a linear function, which can be understood as a Perceptron model.
- The sigmoid does not change the way decisions are made: $\nu_j \leftarrow \sigma\left(\sum_{i=1}^{n} w_{ji} \cdot \nu_i\right)$. The threshold changes to $1/2$ (instead of zero), but the decision boundary remains the same.
- In other words, the Perceptron is a model of a single neuron.

Artificial Neurons

- The relation $u_j \leftarrow \sum_{i=1}^{n} w_{ji} \cdot \nu_i$ is familiar: if ν_i are the inputs, then this is a linear function, which can be understood as a Perceptron model.
- The sigmoid does not change the way decisions are made: $\nu_j \leftarrow \sigma\left(\sum_{i=1}^{n} w_{ji} \cdot \nu_i\right)$. The threshold changes to 1/2 (instead of zero), but the decision boundary remains the same.
- In other words, the Perceptron is a model of a single neuron.
- Usually many neurons process the input, so we have multiple Perceptrons in parallel:



Layered Neural Networks

- Let $\nu^{(0)} \in \mathbb{R}^m$ denote the vector of firing rates coming from the inputs (sensors, data), and let $\nu^{(1)} \in \mathbb{R}^n$ denote the vector of firing rates of the neurons. Let $W \in \mathbb{R}^{n \times m}$ be the matrix with entries w_{ji}. Also, let σ denote the component-wise application of the transfer function.

Layered Neural Networks

- Let $\nu^{(0)} \in \mathbb{R}^m$ denote the vector of firing rates coming from the inputs (sensors, data), and let $\nu^{(1)} \in \mathbb{R}^n$ denote the vector of firing rates of the neurons. Let $W \in \mathbb{R}^{n \times m}$ be the matrix with entries w_{ji}. Also, let σ denote the component-wise application of the transfer function.
- The computation can be written in compact form:

$$\nu^{(1)} = \sigma(W \cdot \nu^{(0)})$$

Layered Neural Networks

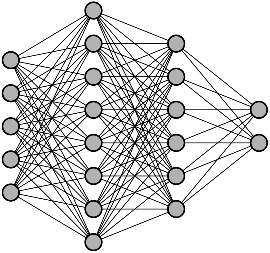
- Let $\nu^{(0)} \in \mathbb{R}^m$ denote the vector of firing rates coming from the inputs (sensors, data), and let $\nu^{(1)} \in \mathbb{R}^n$ denote the vector of firing rates of the neurons. Let $W \in \mathbb{R}^{n \times m}$ be the matrix with entries w_{ji}. Also, let σ denote the component-wise application of the transfer function.
- The computation can be written in compact form:

$$\nu^{(1)} = \sigma(W \cdot \nu^{(0)})$$

- Neurons can not only receive input from sensors, but also from other neurons. What if we feed the outputs into another layer of neurons?

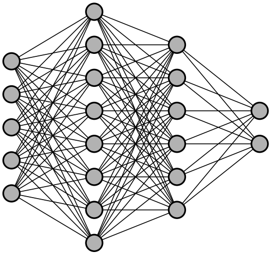
Layered Neural Networks

- The resulting architecture is called a (layered) feed-forward neural network, or multi layer Perceptron (MLP).



Layered Neural Networks

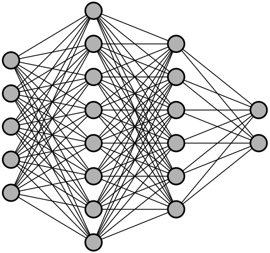
- The resulting architecture is called a (layered) feed-forward neural network, or multi layer Perceptron (MLP).



- This example network has an **input layer** with 5 nodes, two **hidden layers** with 8 and 6 neurons, and an **output layer** with 2 neurons.

Layered Neural Networks

- The resulting architecture is called a (layered) feed-forward neural network, or multi layer Perceptron (MLP).



- This example network has an **input layer** with 5 nodes, two **hidden layers** with 8 and 6 neurons, and an **output layer** with 2 neurons.
- The size of the input and output layers is determined by the problem (dimension of the vectors x and y), but number and size of the hidden layers is arbitrary.

Layered Neural Networks

- Now let $\nu^{(0)}$ denote the vector of inputs, let $\nu^{(i)}$ denote the vector of firing rates in layer i, and let $W^{(i)}$ denote the matrix of connections from layer $i-1$ to layer i. Then we have the overall model:

$$\nu^{(n)} = \sigma_{\text{out}}\left(W^{(n)} \cdot \sigma\Big(W^{(n-1)} \cdot \sigma\big(\dots \sigma(W^{(1)} \cdot \nu^{(0)}) \dots \big)\Big)\right) \ .$$

Layered Neural Networks

- Now let $\nu^{(0)}$ denote the vector of inputs, let $\nu^{(i)}$ denote the vector of firing rates in layer i, and let $W^{(i)}$ denote the matrix of connections from layer $i-1$ to layer i. Then we have the overall model:

$$\nu^{(n)} = \sigma_{\text{out}}\left(W^{(n)} \cdot \sigma\left(W^{(n-1)} \cdot \sigma\big(\ldots \sigma\big(W^{(1)} \cdot \nu^{(0)}\big) \ldots \big) \right) \right) .$$

- The model processes a data point x as follows:

Layered Neural Networks

- Now let $\nu^{(0)}$ denote the vector of inputs, let $\nu^{(i)}$ denote the vector of firing rates in layer i, and let $W^{(i)}$ denote the matrix of connections from layer $i-1$ to layer i. Then we have the overall model:

$$\nu^{(n)} = \sigma_{\text{out}}\left(W^{(n)} \cdot \sigma\left(W^{(n-1)} \cdot \sigma\big(\ldots \sigma\big(W^{(1)} \cdot \nu^{(0)}\big) \ldots\big)\right)\right) \ .$$

- The model processes a data point x as follows:
  - Set $\nu^{(0)} = x$.

Layered Neural Networks

- Now let $\nu^{(0)}$ denote the vector of inputs, let $\nu^{(i)}$ denote the vector of firing rates in layer i, and let $W^{(i)}$ denote the matrix of connections from layer $i-1$ to layer i. Then we have the overall model:

$$\nu^{(n)} = \sigma_{\text{out}}\left(W^{(n)} \cdot \sigma\left(W^{(n-1)} \cdot \sigma\left(\ldots \sigma(W^{(1)} \cdot \nu^{(0)}) \ldots\right)\right)\right) .$$

- The model processes a data point x as follows:
  - Set $\nu^{(0)} = x$.
  - Apply the model, i.e., compute $\nu^{(n)}$ from $\nu^{(0)}$.

Layered Neural Networks

- Now let $\nu^{(0)}$ denote the vector of inputs, let $\nu^{(i)}$ denote the vector of firing rates in layer i, and let $W^{(i)}$ denote the matrix of connections from layer $i-1$ to layer i. Then we have the overall model:

$$\nu^{(n)} = \sigma_{\text{out}}\left(W^{(n)} \cdot \sigma\left(W^{(n-1)} \cdot \sigma\left(\dots \sigma(W^{(1)} \cdot \nu^{(0)}) \dots\right)\right)\right) .$$

- The model processes a data point x as follows:
  - Set $\nu^{(0)} = x$.
  - Apply the model, i.e., compute $\nu^{(n)}$ from $\nu^{(0)}$.
  - Output $\hat{y} = \nu^{(n)}$ (which is hopefully close to the true label y).

Layered Neural Networks

$$\nu^{(n)} = \sigma_{\text{out}}\left(W^{(n)} \cdot \sigma\left(W^{(n-1)} \cdot \sigma\big(\ldots \sigma(W^{(1)} \cdot \nu^{(0)}) \ldots\big)\right)\right)$$

- Hidden layers employ a sigmoid transfer function. The transfer function σ_{out} of the output layer is chosen task specific:

Layered Neural Networks

$$\nu^{(n)} = \sigma_{\text{out}}\left(W^{(n)} \cdot \sigma\left(W^{(n-1)} \cdot \sigma\left(\ldots \sigma(W^{(1)} \cdot \nu^{(0)}) \ldots\right)\right)\right)$$

- Hidden layers employ a sigmoid transfer function. The transfer function σ_{out} of the output layer is chosen task specific:
  - Regression problems usually need an unbounded range of values. Then a sigmoid is not appropriate. The identity function is used in this case (so-called "linear" output neurons).

Layered Neural Networks

$$\nu^{(n)} = \sigma_{\text{out}}\left(W^{(n)} \cdot \sigma\left(W^{(n-1)} \cdot \sigma\left(\dots \sigma(W^{(1)} \cdot \nu^{(0)}) \dots \right) \right) \right)$$

- Hidden layers employ a sigmoid transfer function. The transfer function σ_{out} of the output layer is chosen task specific:
  - Regression problems usually need an unbounded range of values. Then a sigmoid is not appropriate. The identity function is used in this case (so-called "linear" output neurons).
  - For classification the range of values does not matter. Either linear or sigmoid output layer neurons can be used.

Layered Neural Networks

$$\nu^{(n)} = \sigma_{\text{out}}\left(W^{(n)} \cdot \sigma\left(W^{(n-1)} \cdot \sigma\left(\dots \sigma(W^{(1)} \cdot \nu^{(0)}) \dots \right) \right) \right)$$

- Hidden layers employ a sigmoid transfer function. The transfer function σ_{out} of the output layer is chosen task specific:
  - Regression problems usually need an unbounded range of values. Then a sigmoid is not appropriate. The identity function is used in this case (so-called "linear" output neurons).
  - For classification the range of values does not matter. Either linear or sigmoid output layer neurons can be used.
- The linear function $u = W \cdot \nu$ is usually extended to an affine function $u = W \cdot \nu + b$ by means of a so-called **bias neuron**. This neuron has a constant firing rate of one and is input to all other neurons, with connection weights b_i.

Layered Neural Networks

$$\nu^{(n)} = \sigma_{\text{out}}\left(W^{(n)} \cdot \sigma\left(W^{(n-1)} \cdot \sigma\left(\ldots\sigma(W^{(1)} \cdot \nu^{(0)})\ldots\right)\right)\right)$$

- Hidden layers employ a sigmoid transfer function. The transfer function σ_{out} of the output layer is chosen task specific:
  - Regression problems usually need an unbounded range of values. Then a sigmoid is not appropriate. The identity function is used in this case (so-called "linear" output neurons).
  - For classification the range of values does not matter. Either linear or sigmoid output layer neurons can be used.
- The linear function $u = W \cdot \nu$ is usually extended to an affine function $u = W \cdot \nu + b$ by means of a so-called **bias neuron**. This neuron has a constant firing rate of one and is input to all other neurons, with connection weights b_i.
- This is effectively the same as embedding affine functions into

Layered Neural Networks

- A layered neural network alternates the application of two types of transformations:

Layered Neural Networks

- A layered neural network alternates the application of two types of transformations:
  1. A linear map: left multiplication with the matrix $W^{(i)}$. This matrix is a parameter of the model, so it can be subject to learning.

Layered Neural Networks

- A layered neural network alternates the application of two types of transformations:
  1. A linear map: left multiplication with the matrix $W^{(i)}$. This matrix is a parameter of the model, so it can be subject to learning.
  2. A non-linear function: component-wise transfer function σ. This function is fixed. It has no parameters that can be adjusted.

Layered Neural Networks

- A layered neural network alternates the application of two types of transformations:
  1. A linear map: left multiplication with the matrix $W^{(i)}$. This matrix is a parameter of the model, so it can be subject to learning.
  2. A non-linear function: component-wise transfer function σ. This function is fixed. It has no parameters that can be adjusted.
- The non-linearities are not adaptive, but they are nevertheless important! Without them the model would collapse into the linear map $W = W^{(n)} \cdot ... \cdot W^{(1)}$. Then all computations were linear.

Layered Neural Networks

- A layered neural network alternates the application of two types of transformations:
  1. A linear map: left multiplication with the matrix $W^{(i)}$. This matrix is a parameter of the model, so it can be subject to learning.
  2. A non-linear function: component-wise transfer function σ. This function is fixed. It has no parameters that can be adjusted.
- The non-linearities are not adaptive, but they are nevertheless important! Without them the model would collapse into the linear map $W = W^{(n)} \cdot \ldots \cdot W^{(1)}$. Then all computations were linear.
- It turns out that a neural network with sigmoid transfer functions is indeed far more powerful than a linear model. In a sense, it can compute "everything".

Universal Approximation Property

Theorem. Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a continuous, non-constant, bounded, and monotonically increasing function. Let $K \subset \mathbb{R}^m$ be compact, and let $\mathcal{C}(K)$ denote the space of continuous functions $K \to \mathbb{R}$. Then, given a function $g \in \mathcal{C}(K)$ and an accuracy $\varepsilon > 0$, there exists a hidden layer size $N \in \mathbb{N}$ and a set of coefficients $w_i^{(1)} \in \mathbb{R}^m$, $w_i^{(2)}, b_i \in \mathbb{R}$ (for $i \in \{1, \ldots, N\}$), such that

$$f : K \to \mathbb{R} ; \quad f(x) = \sum_{i=1}^{N} w_i^{(2)} \cdot \sigma\Big((w_i^{(1)})^T x + b_i \Big)$$

is an ε-approximation of g, that is,

$$\|f - g\|_\infty := \max_{x \in K} |f(x) - g(x)| < \varepsilon \ .$$

Universal Approximation Property

Theorem. Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a continuous, non-constant, bounded, and monotonically increasing function. Let $K \subset \mathbb{R}^m$ be compact, and let $\mathcal{C}(K)$ denote the space of continuous functions $K \to \mathbb{R}$. Then, given a function $g \in \mathcal{C}(K)$ and an accuracy $\varepsilon > 0$, there exists a hidden layer size $N \in \mathbb{N}$ and a set of coefficients $w_i^{(1)} \in \mathbb{R}^m$, $w_i^{(2)}, b_i \in \mathbb{R}$ (for $i \in \{1, \ldots, N\}$), such that

$$f : K \to \mathbb{R} \, ; \quad f(x) = \sum_{i=1}^{N} w_i^{(2)} \cdot \sigma\Big((w_i^{(1)})^T x + b_i \Big)$$

is an ε-approximation of g, that is,

$$\|f - g\|_\infty := \max_{x \in K} |f(x) - g(x)| < \varepsilon \ .$$

Corollary. The theorem extends trivially to multiple outputs.

Universal Approximation Property

Theorem. Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a continuous, non-constant, bounded, and monotonically increasing function. Let $K \subset \mathbb{R}^m$ be compact, and let $\mathcal{C}(K)$ denote the space of continuous functions $K \to \mathbb{R}$. Then, given a function $g \in \mathcal{C}(K)$ and an accuracy $\varepsilon > 0$, there exists a hidden layer size $N \in \mathbb{N}$ and a set of coefficients $w_i^{(1)} \in \mathbb{R}^m$, $w_i^{(2)}, b_i \in \mathbb{R}$ (for $i \in \{1, \ldots, N\}$), such that

$$f : K \to \mathbb{R} ; \quad f(x) = \sum_{i=1}^{N} w_i^{(2)} \cdot \sigma\Big((w_i^{(1)})^T x + b_i\Big)$$

is an ε-approximation of g, that is,

$$\|f - g\|_\infty := \max_{x \in K} |f(x) - g(x)| < \varepsilon \ .$$

Corollary. The theorem extends trivially to multiple outputs.

Corollary. Neural networks with a single sigmoidal hidden layer and linear output layer are universal approximators.

Universal Approximation Property

- This means that for a given target function g there exists a sequence of networks $\left(f_k\right)_{k\in\mathbb{N}}$ that converges (pointwise) to the target function.

Universal Approximation Property

- This means that for a given target function g there exists a sequence of networks $\left(f_k\right)_{k \in \mathbb{N}}$ that converges (pointwise) to the target function.
- Usually, as the networks come closer and closer to g, they will need more and more hidden neurons.

Universal Approximation Property

- This means that for a given target function g there exists a sequence of networks $\left(f_k\right)_{k\in\mathbb{N}}$ that converges (pointwise) to the target function.
- Usually, as the networks come closer and closer to g, they will need more and more hidden neurons.
- A network with fixed layer sizes can only model a subspace of all continuous functions. Its dimensionality is limited by the number of weights.

Universal Approximation Property

- This means that for a given target function g there exists a sequence of networks $\left(f_k\right)_{k\in\mathbb{N}}$ that converges (pointwise) to the target function.
- Usually, as the networks come closer and closer to g, they will need more and more hidden neurons.
- A network with fixed layer sizes can only model a subspace of all continuous functions. Its dimensionality is limited by the number of weights.
- The continuous functions form an infinite dimensional vector space. Therefore arbitrarily large hidden layer sizes are needed.

Universal Approximation Property

- This means that for a given target function g there exists a sequence of networks $\left(f_k\right)_{k\in\mathbb{N}}$ that converges (pointwise) to the target function.
- Usually, as the networks come closer and closer to g, they will need more and more hidden neurons.
- A network with fixed layer sizes can only model a subspace of all continuous functions. Its dimensionality is limited by the number of weights.
- The continuous functions form an infinite dimensional vector space. Therefore arbitrarily large hidden layer sizes are needed.
- The universal approximation property is not as special as it seems. For example, polynomials are universal approximators (Weierstraß theorem).

Other Neural Networks

In this lecture we cover only feed-forward neural networks, because this is the most basic and most relevant class for supervised (and unsupervised) learning and the class applied to SRL. A broad range of other network types has been developed.

Other Neural Networks

In this lecture we cover only feed-forward neural networks, because this is the most basic and most relevant class for supervised (and unsupervised) learning and the class applied to SRL. A broad range of other network types has been developed.

- The linear+sigmoid processing model can be replaced by other functions. For example, this leads to radial basis function models.

Other Neural Networks

In this lecture we cover only feed-forward neural networks, because this is the most basic and most relevant class for supervised (and unsupervised) learning and the class applied to SRL. A broad range of other network types has been developed.

- The linear+sigmoid processing model can be replaced by other functions. For example, this leads to radial basis function models.
- **Convolutional Neural Networks (CNNs)** are inspired by the organization of the animal visual cortex. Each neuron receives input only from a "local patch" (corresponding to the receptive field in real neurons).

Other Neural Networks

In this lecture we cover only feed-forward neural networks, because this is the most basic and most relevant class for supervised (and unsupervised) learning and the class applied to SRL. A broad range of other network types has been developed.

- The linear+sigmoid processing model can be replaced by other functions. For example, this leads to radial basis function models.
- **Convolutional Neural Networks (CNNs)** are inspired by the organization of the animal visual cortex. Each neuron receives input only from a "local patch" (corresponding to the receptive field in real neurons).
- Synapses can form loops. This requires the introduction of time delays. Then we speak of **Recurrent Neural Networks (RNNs).** These are even more powerful models: they are not simple mappings, but stateful computers.

Other Neural Networks

In this lecture we cover only feed-forward neural networks, because this is the most basic and most relevant class for supervised (and unsupervised) learning and the class applied to SRL. A broad range of other network types has been developed.

- The linear+sigmoid processing model can be replaced by other functions. For example, this leads to radial basis function models.
- **Convolutional Neural Networks (CNNs)** are inspired by the organization of the animal visual cortex. Each neuron receives input only from a "local patch" (corresponding to the receptive field in real neurons).
- Synapses can form loops. This requires the introduction of time delays. Then we speak of **Recurrent Neural Networks (RNNs).** These are even more powerful models: they are not simple mappings, but stateful computers.
- **Auto-encoders, (restricted) Boltzmann machines**, and **self-organizing maps** are used for unsupervised learning

Other Neural Networks

In this lecture we cover only feed-forward neural networks, because this is the most basic and most relevant class for supervised (and unsupervised) learning and the class applied to SRL. A broad range of other network types has been developed.

- The linear+sigmoid processing model can be replaced by other functions. For example, this leads to radial basis function models.
- **Convolutional Neural Networks (CNNs)** are inspired by the organization of the animal visual cortex. Each neuron receives input only from a "local patch" (corresponding to the receptive field in real neurons).
- Synapses can form loops. This requires the introduction of time delays. Then we speak of **Recurrent Neural Networks (RNNs).** These are even more powerful models: they are not simple mappings, but stateful computers.
- **Auto-encoders, (restricted) Boltzmann machines**, and **self-organizing maps** are used for unsupervised learning

Deep Learning

- Many NNs used e.g. for image processing are really deep, i.e., they consist of 10 layers or more.

Deep Learning

- Many NNs used e.g. for image processing are really deep, i.e., they consist of 10 layers or more.
- We speak of **deep learning**. This has become one of the dominant buzz-words of the field (next to big data).

Deep Learning

- Many NNs used e.g. for image processing are really deep, i.e., they consist of 10 layers or more.
- We speak of **deep learning**. This has become one of the dominant buzz-words of the field (next to big data).
- A central concept of deep learning is that lower layers extract basic features (e.g., edge detectors), while higher layers compose them to complex features (complex cells, invariant object detectors).

Deep Learning

- Many NNs used e.g. for image processing are really deep, i.e., they consist of 10 layers or more.
- We speak of **deep learning**. This has become one of the dominant buzz-words of the field (next to big data).
- A central concept of deep learning is that lower layers extract basic features (e.g., edge detectors), while higher layers compose them to complex features (complex cells, invariant object detectors).
- This is in rough correspondence with our understanding of how the visual cortex processes images.

Deep Learning

- Many NNs used e.g. for image processing are really deep, i.e., they consist of 10 layers or more.
- We speak of **deep learning**. This has become one of the dominant buzz-words of the field (next to big data).
- A central concept of deep learning is that lower layers extract basic features (e.g., edge detectors), while higher layers compose them to complex features (complex cells, invariant object detectors).
- This is in rough correspondence with our understanding of how the visual cortex processes images.
- Recently deep learning revolutionized a lot of fields like image and language procession, machine translations etc. It was also part of Alpha-Go.

From Models to Learners

The class of functions represented by neural networks is “rich enough” to represent the solution to any problem, provided that the network is “big enough”.

From Models to Learners

The class of functions represented by neural networks is “rich enough” to represent the solution to any problem, provided that the network is “big enough”.

- With a large enough hidden layer the network can approximate the optimal hypothesis arbitrarily well.

From Models to Learners

The class of functions represented by neural networks is “rich enough” to represent the solution to any problem, provided that the network is “big enough”.

- With a large enough hidden layer the network can approximate the optimal hypothesis arbitrarily well.
- However, this is not helpful in practice. It does not tell us how to actually set the network size, let alone the weights.

From Models to Learners

The class of functions represented by neural networks is “rich enough” to represent the solution to any problem, provided that the network is “big enough”.

- With a large enough hidden layer the network can approximate the optimal hypothesis arbitrarily well.
- However, this is not helpful in practice. It does not tell us how to actually set the network size, let alone the weights.
- Until now we have defined neural networks as a class of models. We do not have a learning rule yet.

From Models to Learners

The class of functions represented by neural networks is “rich enough” to represent the solution to any problem, provided that the network is “big enough”.

- With a large enough hidden layer the network can approximate the optimal hypothesis arbitrarily well.
- However, this is not helpful in practice. It does not tell us how to actually set the network size, let alone the weights.
- Until now we have defined neural networks as a class of models. We do not have a learning rule yet.
- Neural networks are trained based on **stochastic gradient descent** as described in the following.

(Online) Steepest Descent Training

- Let w denote a vector collecting all weights of a neural network. This is a "linearized" version of all of its weight matrices.

(Online) Steepest Descent Training

- Let w denote a vector collecting all weights of a neural network. This is a “linearized” version of all of its weight matrices.
- Let f_w be the mapping represented by the network for particular weights w. Let

$$E(w) = \frac{1}{|S|} \sum_{i \in S} L(f_w(x_i), y_i)$$

denote the error of the network, as a function of the weights.

(Online) Steepest Descent Training

- Let w denote a vector collecting all weights of a neural network. This is a "linearized" version of all of its weight matrices.
- Let f_w be the mapping represented by the network for particular weights w. Let

$$E(w) = \frac{1}{|S|} \sum_{i \in S} L(f_w(x_i), y_i)$$

denote the error of the network, as a function of the weights.
- The sum may run over the whole data set ($|S| = n$, batch mode), over small subsets ($|S| \ll n$, mini batches), or only over a single example ($|S| = 1$, online mode).

(Online) Steepest Descent Training

$$E(w) = \frac{1}{|S|} \sum_{i \in S} L(f_w(x_i), y_i)$$

- The batch error is what we have called the empirical risk w.r.t. the loss function L.

(Online) Steepest Descent Training

$$E(w) = \frac{1}{|S|} \sum_{i \in S} L(f_w(x_i), y_i)$$

- The batch error is what we have called the empirical risk w.r.t. the loss function L.
- Its minimization is a straightforward learning strategy.

(Online) Steepest Descent Training

$$E(w) = \frac{1}{|S|} \sum_{i \in S} L(f_w(x_i), y_i)$$

- The batch error is what we have called the empirical risk w.r.t. the loss function L.
- Its minimization is a straightforward learning strategy.
- This is usually what we want when training a neural network. So why care about online and mini batch errors?

(Online) Steepest Descent Training

$$E(w) = \frac{1}{|S|} \sum_{i \in S} L(f_w(x_i), y_i)$$

- The batch error is what we have called the empirical risk w.r.t. the loss function L.
- Its minimization is a straightforward learning strategy.
- This is usually what we want when training a neural network. So why care about online and mini batch errors?
- The reason is that the online error is much faster to compute, namely by a factor of n (size of the data set). Thus its use allows for many more gradient descent steps.

(Online) Steepest Descent Training

- Assume we have computed the gradient of the error $\nabla_w E(w)$ with respect to the weights. Then we can perform a step of gradient descent with learning rate η to update the wights.

$$w \leftarrow w - \eta \cdot \nabla_w E(w) \ .$$

Backpropagation

Now we come to the computation of the error gradient $\nabla_w E(w)$.

Backpropagation

Now we come to the computation of the error gradient $\nabla_w E(w)$.

- The error is a simple sum over loss terms of the form $E(w) = L(f_w(x), y)$. We compute $\nabla_w E(w)$ in the following.

Backpropagation

Now we come to the computation of the error gradient $\nabla_w E(w)$.

- The error is a simple sum over loss terms of the form $E(w) = L(f_w(x), y)$. We compute $\nabla_w E(w)$ in the following.
- We write this error as

$$E(w) = L\left(\sigma\left(W^{(n)} \cdot \sigma\left(W^{(n-1)} \cdot \sigma\left(\ldots W^{(1)} \cdot x \ldots\right)\right)\right), y\right) \ ,$$

  where $W^{(k)}$ are the weight matrices and σ is the component-wise non-linearity.

Backpropagation

Now we come to the computation of the error gradient $\nabla_w E(w)$.

- The error is a simple sum over loss terms of the form $E(w) = L(f_w(x), y)$. We compute $\nabla_w E(w)$ in the following.
- We write this error as

$$E(w) = L\left(\sigma\left(W^{(n)} \cdot \sigma\left(W^{(n-1)} \cdot \sigma\left(\ldots W^{(1)} \cdot x \ldots\right)\right)\right), y\right) \ ,$$

  where $W^{(k)}$ are the weight matrices and σ is the component-wise non-linearity.
- The gradient can be calculated by the chain rule.

Backpropagation

Now we come to the computation of the error gradient $\nabla_w E(w)$.

- The error is a simple sum over loss terms of the form $E(w) = L(f_w(x), y)$. We compute $\nabla_w E(w)$ in the following.
- We write this error as

$$E(w) = L\left(\sigma\left(W^{(n)} \cdot \sigma\left(W^{(n-1)} \cdot \sigma\left(\dots W^{(1)} \cdot x \dots\right)\right)\right), y\right) \ ,$$

  where $W^{(k)}$ are the weight matrices and σ is the component-wise non-linearity.
- The gradient can be calculated by the chain rule.
- **Backpropagation** is an algorithm for doing this fast. It will be introduced in the next lecture.

Summary

- We have introduced a simple neuron model which is composed out of linear input and a non-linear transfer/activation function.

Summary

- We have introduced a simple neuron model which is composed out of linear input and a non-linear transfer/activation function.
- We have composed multiple neurons in parallel to layers, and multiple layers in sequence to feed forward neural networks (multi layer Perceptrons, MLPs).

Summary

- We have introduced a simple neuron model which is composed out of linear input and a non-linear transfer/activation function.
- We have composed multiple neurons in parallel to layers, and multiple layers in sequence to feed forward neural networks (multi layer Perceptrons, MLPs).
- Neural networks are universal function approximators.

Summary

- We have introduced a simple neuron model which is composed out of linear input and a non-linear transfer/activation function.
- We have composed multiple neurons in parallel to layers, and multiple layers in sequence to feed forward neural networks (multi layer Perceptrons, MLPs).
- Neural networks are universal function approximators.
- Networks can be trained by online or batch gradient descent.

Summary

- We have introduced a simple neuron model which is composed out of linear input and a non-linear transfer/activation function.
- We have composed multiple neurons in parallel to layers, and multiple layers in sequence to feed forward neural networks (multi layer Perceptrons, MLPs).
- Neural networks are universal function approximators.
- Networks can be trained by online or batch gradient descent.
- The error gradient can be computed efficiently with the backpropagation algorithm.

Summary

- We have introduced a simple neuron model which is composed out of linear input and a non-linear transfer/activation function.
- We have composed multiple neurons in parallel to layers, and multiple layers in sequence to feed forward neural networks (multi layer Perceptrons, MLPs).
- Neural networks are universal function approximators.
- Networks can be trained by online or batch gradient descent.
- The error gradient can be computed efficiently with the backpropagation algorithm.
- The weights usually end up in a local optimum, not in the global optimum.

Summary

- We have introduced a simple neuron model which is composed out of linear input and a non-linear transfer/activation function.
- We have composed multiple neurons in parallel to layers, and multiple layers in sequence to feed forward neural networks (multi layer Perceptrons, MLPs).
- Neural networks are universal function approximators.
- Networks can be trained by online or batch gradient descent.
- The error gradient can be computed efficiently with the backpropagation algorithm.
- The weights usually end up in a local optimum, not in the global optimum.

Acknowledgments

Acknowledgments: We thank Tobias Glasmachers for providing us the material for this class which was taken from his lecture *Machine Learning - Supervised Methods* at the Ruhr-Uni-Bochum.