Machine Learning

A Bayesian and Optimization Perspective

Academic Press, 2015

Sergios Theodoridis[1]

[1] Dept. of Informatics and Telecommunications, National and Kapodistrian University of Athens, Athens, Greece.

Spring 2017, Version III

Chapter 18
Neural Networks and Deep Learning

Neural Networks

- Neural networks have a long history which goes back to the first attempts to understand how the human and mamal brain works and how what we call intelligence is formed.

- From a physiological point of view, one can trace the beginning of the field back to the work of Santiago Ramon y Cajal, who discovered that the basic building element of the brain is the neuron. The brain comprises approximately 60-100 billions neurons; that is, a number of the same order as the number of stars in our galaxy!

Neural Networks

- Neural networks have a long history which goes back to the first attempts to understand how the human and mamal brain works and how what we call intelligence is formed.
- From a physiological point of view, one can trace the beginning of the field back to the work of Santiago Ramon y Cajal, who discovered that the basic building element of the brain is the neuron. The brain comprises approximately 60-100 billions neurons; that is, a number of the same order as the number of stars in our galaxy!

Neural Networks

- Each neuron is connected with other neurons via elementary structural and functional units/links, known as synapses. It is estimated that there are 50-100 trillions of synapses. These links mediate information between connected neurons.
- The most common type of synapses are the chemical ones, which convert electric pulses, produced by a neuron, to a chemical signal and then back to an electrical one.
- Depending on the input pulse(s), a synapse is either activated or inhibited. Via these links, each neuron is connected to other neurons and this happens in a hierarchical way, in a layer-wise fashion.

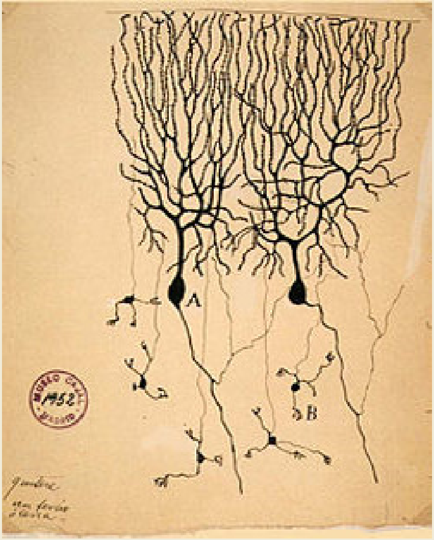
Neural Networks

- Each neuron is connected with other neurons via elementary structural and functional units/links, known as synapses. It is estimated that there are 50-100 trillions of synapses. These links mediate information between connected neurons.
- The most common type of synapses are the chemical ones, which convert electric pulses, produced by a neuron, to a chemical signal and then back to an electrical one.
- Depending on the input pulse(s), a synapse is either activated or inhibited. Via these links, each neuron is connected to other neurons and this happens in a hierarchical way, in a layer-wise fashion.

Neural Networks

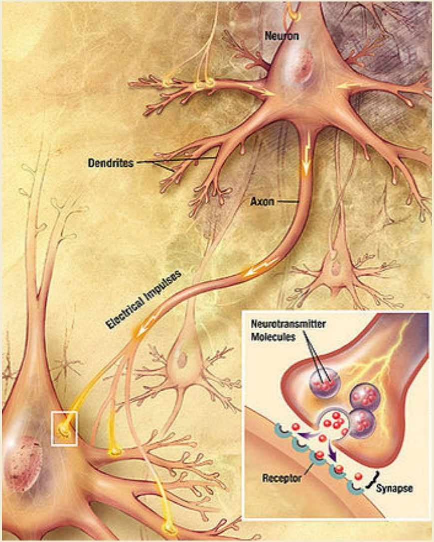
- Each neuron is connected with other neurons via elementary structural and functional units/links, known as synapses. It is estimated that there are 50-100 trillions of synapses. These links mediate information between connected neurons.
- The most common type of synapses are the chemical ones, which convert electric pulses, produced by a neuron, to a chemical signal and then back to an electrical one.
- Depending on the input pulse(s), a synapse is either activated or inhibited. Via these links, each neuron is connected to other neurons and this happens in a hierarchical way, in a layer-wise fashion.

The Neuron



Drawing of neurons in the pigeon cerebellum, by Santiago Ramón y Cajal in 1899 (http://en.wikipedia.org/wiki/Neuron).

The Neuron



A signal propagating down an axon to the cell body and dendrites of the next cell (http://en.wikipedia.org/wiki/Neuron).

Neural Networks

- A milestone from the learning theory's point of view occurred in 1943, when Warren McCulloch and Walter Pitts, developed a computational model for the basic neuron. Moreover, they provided results that tie neurophysiology with mathematical logic.
- They showed that given a sufficient number of neurons and adjusting appropriately the synaptic links, each one represented by a weight, one can compute, in principle, any computable function.
- As a matter of fact, it is generally accepted that this is the paper that gave birth to the fields of neural networks and artificial intelligence.

Neural Networks

- A milestone from the learning theory's point of view occurred in 1943, when Warren McCulloch and Walter Pitts, developed a computational model for the basic neuron. Moreover, they provided results that tie neurophysiology with mathematical logic.
- They showed that given a sufficient number of neurons and adjusting appropriately the synaptic links, each one represented by a weight, one can compute, in principle, any computable function.
- As a matter of fact, it is generally accepted that this is the paper that gave birth to the fields of neural networks and artificial intelligence.

Neural Networks

- A milestone from the learning theory's point of view occurred in 1943, when Warren McCulloch and Walter Pitts, developed a computational model for the basic neuron. Moreover, they provided results that tie neurophysiology with mathematical logic.
- They showed that given a sufficient number of neurons and adjusting appropriately the synaptic links, each one represented by a weight, one can compute, in principle, any computable function.
- As a matter of fact, it is generally accepted that this is the paper that gave birth to the fields of neural networks and artificial intelligence.

Neural Networks

- Frank Rosenblatt, borrowed the idea of a neuron model, as suggested by McCulloch and Pitts, and proposed a true learning machine, which **learns** from a set of training data.
- In the most basic version of operation, he used a single neuron and adopted a rule that can learn to separate data, which belong to two linearly separable classes. That is, he built a Pattern Recognition system.
- He called the basic neuron a perceptron and developed a rule/algorithm, the perceptron algorithm, for the respective training. The perceptron will be the kick-off point for our tour in this series of lectures.

Neural Networks

- Frank Rosenblatt, borrowed the idea of a neuron model, as suggested by McCulloch and Pitts, and proposed a true learning machine, which **learns** from a set of training data.
- In the most basic version of operation, he used a single neuron and adopted a rule that can learn to separate data, which belong to two linearly separable classes. That is, he built a Pattern Recognition system.
- He called the basic neuron a perceptron and developed a rule/algorithm, the perceptron algorithm, for the respective training. The perceptron will be the kick-off point for our tour in this series of lectures.

Neural Networks

- Frank Rosenblatt, borrowed the idea of a neuron model, as suggested by McCulloch and Pitts, and proposed a true learning machine, which **learns** from a set of training data.
- In the most basic version of operation, he used a single neuron and adopted a rule that can learn to separate data, which belong to two linearly separable classes. That is, he built a Pattern Recognition system.
- He called the basic neuron a perceptron and developed a rule/algorithm, the perceptron algorithm, for the respective training. The perceptron will be the kick-off point for our tour in this series of lectures.

The Perceptron And The Perceptron Rule

- Our starting point is the simple problem of a linearly separable two-class (ω_1, ω_2) classification task. In other words, we are given a set of training samples, $(y_n, \boldsymbol{x}_n),\ n = 1, 2, \ldots, N$, with $y_n \in \{-1, +1\}$, and it is assumed that there exists a hyperplane,

$$\boldsymbol{\theta}_*^T \boldsymbol{x} = 0: \quad \text{such that,}$$

$$\begin{aligned} \boldsymbol{\theta}_*^T \boldsymbol{x} &> 0, \quad \text{if} \quad \boldsymbol{x} \in \omega_1 \\ \boldsymbol{\theta}_*^T \boldsymbol{x} &< 0, \quad \text{if} \quad \boldsymbol{x} \in \omega_2. \end{aligned}$$

- In other words, such a hyperplane classifies correctly **all** the points in the training set. For notational simplification, the bias term of the hyperplane has been absorbed in $\boldsymbol{\theta}_*$.

The Perceptron And The Perceptron Rule

- Our starting point is the simple problem of a linearly separable two-class (ω_1, ω_2) classification task. In other words, we are given a set of training samples, $(y_n, \boldsymbol{x}_n),\ n = 1, 2, \ldots, N$, with $y_n \in \{-1, +1\}$, and it is assumed that there exists a hyperplane,

$$\boldsymbol{\theta}_*^T \boldsymbol{x} = 0 : \quad \text{such that,}$$

$$\begin{aligned} \boldsymbol{\theta}_*^T \boldsymbol{x} &> 0, \quad \text{if } \boldsymbol{x} \in \omega_1 \\ \boldsymbol{\theta}_*^T \boldsymbol{x} &< 0, \quad \text{if } \boldsymbol{x} \in \omega_2. \end{aligned}$$

- In other words, such a hyperplane classifies correctly **all** the points in the training set. For notational simplification, the bias term of the hyperplane has been absorbed in $\boldsymbol{\theta}_*$.

The Perceptron And The Perceptron Rule

- The goal now becomes that of developing an algorithm that **iteratively** computes a hyperplane that classifies correctly **all** the patterns from both classes. To this end, a cost function must first be adopted.

- Let the available estimate at the current iteration step of the unknown parameters be θ. Then, there are two possibilities:
  - all points are classified correctly; this means that a solution has been obtained.
  - θ classifies correctly some of the points and the rest are misclassified.

Let $\mathcal{Y}$ be the set of all misclassified samples.

The Perceptron And The Perceptron Rule

- The goal now becomes that of developing an algorithm that **iteratively** computes a hyperplane that classifies correctly **all** the patterns from both classes. To this end, a cost function must first be adopted.
- Let the available estimate at the current iteration step of the unknown parameters be $\boldsymbol{\theta}$. Then, there are two possibilities:
  - all points are classified correctly; this means that a solution has been obtained.
  - $\boldsymbol{\theta}$ classifies correctly some of the points and the rest are misclassified.

Let $\mathcal{Y}$ be the set of all misclassified samples.

The Perceptron And The Perceptron Rule

- The goal now becomes that of developing an algorithm that **iteratively** computes a hyperplane that classifies correctly **all** the patterns from both classes. To this end, a cost function must first be adopted.
- Let the available estimate at the current iteration step of the unknown parameters be $\boldsymbol{\theta}$. Then, there are two possibilities:
  - all points are classified correctly; this means that a solution has been obtained.
  - $\boldsymbol{\theta}$ classifies correctly some of the points and the rest are misclassified.

Let $\mathcal{Y}$ be the set of all misclassified samples.

The Perceptron And The Perceptron Rule

- The perceptron cost: This is defined as

$$J(\boldsymbol{\theta}) = -\sum_{n:\boldsymbol{x}_n \in \mathcal{Y}} y_n \boldsymbol{\theta}^T \boldsymbol{x}_n \tag{1}$$

where,

$$y_n = \begin{cases} +1, & \text{if } \boldsymbol{x} \in \omega_1 \\ -1, & \text{if } \boldsymbol{x} \in \omega_2 \end{cases}. \tag{2}$$

- The cost function is non-negative. Indeed, since the sum is over the misclassified points, if $\boldsymbol{x}_n \in \omega_1$ (ω_2) then $\boldsymbol{\theta}^T \boldsymbol{x}_n < \ (>) \ 0$ rendering the product $-y_n \boldsymbol{\theta}^T \boldsymbol{x}_n > 0$. The cost function becomes zero, if there are no misclassified points, i.e., $\mathcal{Y} = \emptyset$, which corresponds to a solution.

The Perceptron And The Perceptron Rule

- The perceptron cost: This is defined as

$$J(\boldsymbol{\theta}) = -\sum_{n:\boldsymbol{x}_n \in \mathcal{Y}} y_n \boldsymbol{\theta}^T \boldsymbol{x}_n \quad (1)$$

where,

$$y_n = \begin{cases} +1, & \text{if } \boldsymbol{x} \in \omega_1 \\ -1, & \text{if } \boldsymbol{x} \in \omega_2 \end{cases}. \quad (2)$$

- The cost function is non-negative. Indeed, since the sum is over the misclassified points, if $\boldsymbol{x}_n \in \omega_1$ (ω_2) then $\boldsymbol{\theta}^T \boldsymbol{x}_n < \ (>)\ 0$ rendering the product $-y_n \boldsymbol{\theta}^T \boldsymbol{x}_n > 0$. The cost function becomes zero, if there are no misclassified points, i.e., $\mathcal{Y} = \emptyset$, which corresponds to a solution.

The Perceptron And The Perceptron Rule

- The perceptron cost function is not differentiable at all points. It is a continuous **piece-wise** linear function. Indeed, let us write it in a slightly different way,

$$J(\boldsymbol{\theta}) = \left(- \sum_{n: \boldsymbol{x}_n \in \mathcal{Y}} y_n \boldsymbol{x}_n^T \right) \boldsymbol{\theta},$$

  This is a linear function with respect to $\boldsymbol{\theta}$, as long as the number of misclassified points remains the same.
- However, as one slowly changes the value of θ, which corresponds to a change of the (direction/position of the hyperplane), there will be a point where the number of misclassified samples in $\mathcal{Y}$ suddenly changes; this is the time, where a sample changes its relative position with respect to the (moving) hyperplane. Hence, the set $\mathcal{Y}$ is modified.
- After this change, $J(\theta)$ will correspond to a new linear function.

The Perceptron And The Perceptron Rule

- The perceptron cost function is not differentiable at all points. It is a continuous **piece-wise** linear function. Indeed, let us write it in a slightly different way,

$$J(\boldsymbol{\theta}) = \left(- \sum_{n:\boldsymbol{x}_n \in \mathcal{Y}} y_n \boldsymbol{x}_n^T \right) \boldsymbol{\theta},$$

  This is a linear function with respect to $\boldsymbol{\theta}$, as long as the number of misclassified points remains the same.

- However, as one slowly changes the value of $\boldsymbol{\theta}$, which corresponds to a change of the (direction/position of the hyperplane), there will be a point where the number of misclassified samples in $\mathcal{Y}$ suddenly changes; this is the time, where a sample changes its relative position with respect to the (moving) hyperplane. Hence, the set $\mathcal{Y}$ is modified.

- After this change, $J(\boldsymbol{\theta})$ will correspond to a new linear function.

The Perceptron And The Perceptron Rule

- The perceptron cost function is not differentiable at all points. It is a continuous **piece-wise** linear function. Indeed, let us write it in a slightly different way,

$$J(\boldsymbol{\theta}) = \left(- \sum_{n:\boldsymbol{x}_n \in \mathcal{Y}} y_n \boldsymbol{x}_n^T \right) \boldsymbol{\theta},$$

  This is a linear function with respect to $\boldsymbol{\theta}$, as long as the number of misclassified points remains the same.
- However, as one slowly changes the value of $\boldsymbol{\theta}$, which corresponds to a change of the (direction/position of the hyperplane), there will be a point where the number of misclassified samples in $\mathcal{Y}$ suddenly changes; this is the time, where a sample changes its relative position with respect to the (moving) hyperplane. Hence, the set $\mathcal{Y}$ is modified.
- After this change, $J(\boldsymbol{\theta})$ will correspond to a new linear function.

The Perceptron And The Perceptron Rule

- The Perceptron Algorithm: It can be shown that, starting from an arbitrary point, $\boldsymbol{\theta}^{(0)}$, the following iterative update,

$$\boldsymbol{\theta}^{(i)} = \boldsymbol{\theta}^{(i-1)} + \mu_i \sum_{n:\boldsymbol{x}_n \in \mathcal{Y}} y_n \boldsymbol{x}_n$$

converges after a finite number of steps. The parameter μ_i is the user-defined step-size, judicially chosen to guarantee convergence.

- Besides the previous scheme, another version of the algorithm considers one sample per iteration in a cyclic fashion, till the algorithm converges.

The Perceptron And The Perceptron Rule

- The Perceptron Algorithm: It can be shown that, starting from an arbitrary point, $\boldsymbol{\theta}^{(0)}$, the following iterative update,

$$\boldsymbol{\theta}^{(i)} = \boldsymbol{\theta}^{(i-1)} + \mu_i \sum_{n:\boldsymbol{x}_n \in \mathcal{Y}} y_n \boldsymbol{x}_n$$

  converges after a finite number of steps. The parameter μ_i is the user-defined step-size, judicially chosen to guarantee convergence.

- Besides the previous scheme, another version of the algorithm considers one sample per iteration in a cyclic fashion, till the algorithm converges.

The Perceptron And The Perceptron Rule

- Let us denote by $y_{(i)},\ \boldsymbol{x}_{(i)}$, $(i) \in \{1, 2, \ldots, N\}$, the training pair that is presented in the algorithms at the ith iteration step.
- Pattern-by-Pattern Perceptron Algorithm: In this formulation, the algorithm becomes,

$$\boldsymbol{\theta}^{(i)} = \begin{cases} \boldsymbol{\theta}^{(i-1)} + \mu_i y_{(i)} \boldsymbol{x}_{(i)}, & \text{if } \boldsymbol{x}_{(i)} \text{ is misclassified by } \boldsymbol{\theta}^{(i-1)}, \\ \boldsymbol{\theta}^{(i-1)}, & \text{otherwise.} \end{cases}$$

- In other words, starting from an initial estimate, e.g., taken to be equal to zero, $\boldsymbol{\theta}^{(0)} = \mathbf{0}$, we test each one of the samples, $\boldsymbol{x}_n,\ n = 1, 2, \ldots, N$. Once all samples have been considered, we say that one epoch has been completed.
- If no convergence has been attained, all samples are reconsidered in a second epoch and so on.

The Perceptron And The Perceptron Rule

- Let us denote by $y_{(i)},\ \boldsymbol{x}_{(i)}$, $(i) \in \{1, 2, \ldots, N\}$, the training pair that is presented in the algorithms at the ith iteration step.
- Pattern-by-Pattern Perceptron Algorithm: In this formulation, the algorithm becomes,

$$\boldsymbol{\theta}^{(i)} = \begin{cases} \boldsymbol{\theta}^{(i-1)} + \mu_i y_{(i)} \boldsymbol{x}_{(i)}, & \text{if } \boldsymbol{x}_{(i)} \text{ is misclassified by } \boldsymbol{\theta}^{(i-1)}, \\ \boldsymbol{\theta}^{(i-1)}, & \text{otherwise.} \end{cases}$$

- In other words, starting from an initial estimate, e.g., taken to be equal to zero, $\boldsymbol{\theta}^{(0)} = \mathbf{0}$, we test each one of the samples, $\boldsymbol{x}_n,\ n = 1, 2, \ldots, N$. Once all samples have been considered, we say that one epoch has been completed.
- If no convergence has been attained, all samples are reconsidered in a second epoch and so on.

The Perceptron And The Perceptron Rule

- Let us denote by $y_{(i)},\ \boldsymbol{x}_{(i)}$, $(i) \in \{1, 2, \ldots, N\}$, the training pair that is presented in the algorithms at the ith iteration step.
- Pattern-by-Pattern Perceptron Algorithm: In this formulation, the algorithm becomes,

$$\boldsymbol{\theta}^{(i)} = \begin{cases} \boldsymbol{\theta}^{(i-1)} + \mu_i y_{(i)} \boldsymbol{x}_{(i)}, & \text{if } \boldsymbol{x}_{(i)} \text{ is misclassified by } \boldsymbol{\theta}^{(i-1)}, \\ \boldsymbol{\theta}^{(i-1)}, & \text{otherwise.} \end{cases}$$

- In other words, starting from an initial estimate, e.g., taken to be equal to zero, $\boldsymbol{\theta}^{(0)} = \mathbf{0}$, we test each one of the samples, $\boldsymbol{x}_n,\ n = 1, 2, \ldots, N$. Once all samples have been considered, we say that one epoch has been completed.
- If no convergence has been attained, all samples are reconsidered in a second epoch and so on.

The Perceptron And The Perceptron Rule

- The previous algorithm is known as pattern-by-pattern or online mode of operation. Note that, the term "online" here indicates that the total number of data samples is fixed and the algorithm considers them in a cyclic fashion, epoch after epoch.

- After a successive finite number of epochs, the algorithm is guaranteed to converge. Note that for convergence, the sequence μ_i must be appropriately chosen. For the case of the perceptron algorithm, convergence is still guaranteed even if μ_i is a positive constant, $\mu_i = \mu > 0$, usually taken to be equal to one.

The Perceptron And The Perceptron Rule

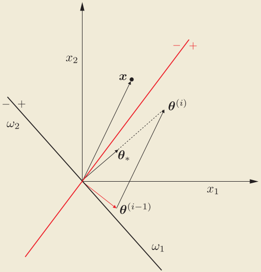
- The previous algorithm is known as pattern-by-pattern or online mode of operation. Note that, the term "online" here indicates that the total number of data samples is fixed and the algorithm considers them in a cyclic fashion, epoch after epoch.
- After a successive finite number of epochs, the algorithm is guaranteed to converge. Note that for convergence, the sequence μ_i must be appropriately chosen. For the case of the perceptron algorithm, convergence is still guaranteed even if μ_i is a positive constant, $\mu_i = \mu > 0$, usually taken to be equal to one.

The Perceptron And The Perceptron Rule

- The following figure provides a geometric interpretation of the perceptron rule. The sample $\boldsymbol{x}$ is misclassified by the hyperplane, $\boldsymbol{\theta}^{(i-1)}$. Since $\boldsymbol{x}$ lies in the $(-)$ side of the hyperplane and it is misclassified, it belongs to class ω_1. Hence, assuming $\mu = 1$, the applied correction by the algorithm is

$$\boldsymbol{\theta}^{(i)} = \boldsymbol{\theta}^{(i-1)} + \boldsymbol{x},$$

and its effect is to turn the hyperplane to the direction towards $\boldsymbol{x}$ so that to place it in the $(+)$ side of the new hyperplane, which is defined by the updated estimate $\boldsymbol{\theta}^{(i)}$.



The Artificial Neuron

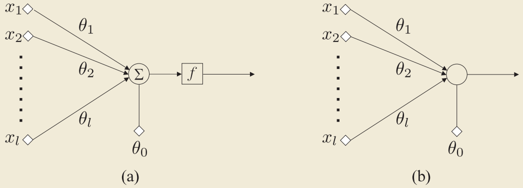
- Once the perceptron algorithm has run and converged, we have available the weights, $\theta_i, \ i = 1, 2, \ldots, l$, of the synapses of the associated neuron/perceptron as well as the bias term θ_0. These can now be used to classify unknown patterns.
- Basic neuron element: The features, $x_i, \ i = 1, 2, \ldots, l$, are applied to the input nodes. In turn, each feature is multiplied by the respective synapse (weight) and then the bias term is added on their linear combination. The outcome of this operation then goes through a nonlinear function, $f(\cdot)$, known as the activation function. In the more classical version, known as the McCulloch-Pitts neuron the activation function is the Heaviside one, i.e.,

$$f(z) = \begin{cases} 1 & \text{if } z > 0, \\ 0 & \text{if } z \leq 0. \end{cases}$$

The Artificial Neuron

- Once the perceptron algorithm has run and converged, we have available the weights, $\theta_i, \; i = 1, 2, \ldots, l$, of the synapses of the associated neuron/perceptron as well as the bias term θ_0. These can now be used to classify unknown patterns.
- Basic neuron element: The features, $x_i, \; i = 1, 2, \ldots, l$, are applied to the input nodes. In turn, each feature is multiplied by the respective synapse (weight) and then the bias term is added on their linear combination. The outcome of this operation then goes through a nonlinear function, $f(\cdot)$, known as the activation function. In the more classical version, known as the McCulloch-Pitts neuron the activation function is the Heaviside one, i.e.,

$$f(z) = \begin{cases} 1 & \text{if } z > 0, \\ 0 & \text{if } z \leq 0. \end{cases}$$



Feed-Forward Multilayer Neural Networks

- A single neuron realizes a hyperplane,

$$\theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_l x_l + \theta_0 = 0,$$

  in the input (feature) space. We will now see how to combine neurons, in a layer-wise fashion, in order to construct nonlinear classifiers. We will follow a simple constructive proof, which unveils certain aspects of neural networks.

- As a staring point, we consider classes, in the feature space, which are formed by unions of polyhedral regions, as shown in the figure below,

Feed-Forward Multilayer Neural Networks

- A single neuron realizes a hyperplane,

$$\theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_l x_l + \theta_0 = 0,$$

  in the input (feature) space. We will now see how to combine neurons, in a layer-wise fashion, in order to construct nonlinear classifiers. We will follow a simple constructive proof, which unveils certain aspects of neural networks.

- As a staring point, we consider classes, in the feature space, which are formed by unions of polyhedral regions, as shown in the figure below,

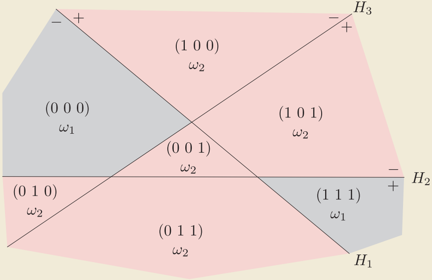
Feed-Forward Multilayer Neural Networks

- A single neuron realizes a hyperplane,

$$\theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_l x_l + \theta_0 = 0,$$

  in the input (feature) space. We will now see how to combine neurons, in a layer-wise fashion, in order to construct nonlinear classifiers. We will follow a simple constructive proof, which unveils certain aspects of neural networks.

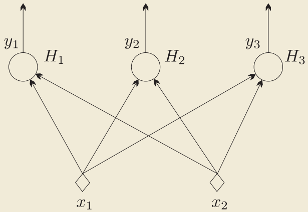
- As a staring point, we consider classes, in the feature space, which are formed by unions of polyhedral regions, as shown in the figure below,



Classes are formed by union of polyhedral regions. Regions are labeled according to the side they lie, with respect to the three lines, H_1, H_2, H_3. The number "1" indicates the (+) side and the "0" the (-) side. The class ω_1 consists of the union of the (000) and (111) regions.

Feed-Forward Multilayer Neural Networks

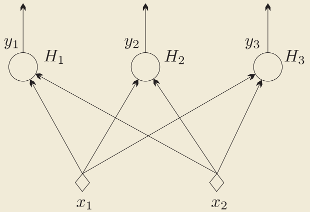
- The figure below shows three neurons, realizing the three hyperplanes, H_1, H_2, H_3, of the previous figure, respectively.



- The corresponding outputs, denoted as $y_1,\ y_2,\ y_3$, form the label of the region associated with the input pattern, which is applied on the input nodes. Indeed, if the weights of the synapses have been appropriately set, then if a pattern originates from the region, say, (010), then the first neuron on the left will fire a zero ($y_1 = 0$), the second an one ($y_2 = 1$) and the rightmost a zero ($y_3 = 0$).
- In other words, this layer of neurons forms a mapping of the input space into the 3-D (three neurons) one. We refer to this as the first hidden layer.

Feed-Forward Multilayer Neural Networks

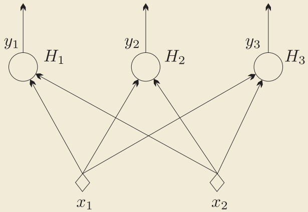
- The figure below shows three neurons, realizing the three hyperplanes, H_1, H_2, H_3, of the previous figure, respectively.



- The corresponding outputs, denoted as $y_1,\ y_2,\ y_3$, form the label of the region associated with the input pattern, which is applied on the input nodes. Indeed, if the weights of the synapses have been appropriately set, then if a pattern originates from the region, say, (010), then the first neuron on the left will fire a zero ($y_1 = 0$), the second an one ($y_2 = 1$) and the rightmost a zero ($y_3 = 0$).
- In other words, this layer of neurons forms a mapping of the input space into the 3-D (three neurons) one. We refer to this as the first hidden layer.

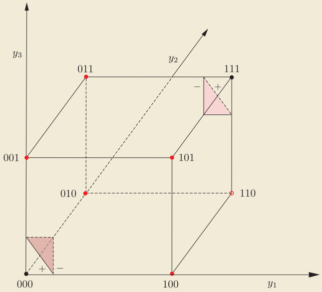
Feed-Forward Multilayer Neural Networks

- The figure below shows three neurons, realizing the three hyperplanes, H_1, H_2, H_3, of the previous figure, respectively.



- The corresponding outputs, denoted as $y_1,\ y_2,\ y_3$, form the label of the region associated with the input pattern, which is applied on the input nodes. Indeed, if the weights of the synapses have been appropriately set, then if a pattern originates from the region, say, (010), then the first neuron on the left will fire a zero ($y_1 = 0$), the second an one ($y_2 = 1$) and the rightmost a zero ($y_3 = 0$).
- In other words, this layer of neurons forms a mapping of the input space into the 3-D (three neurons) one. We refer to this as the first hidden layer.

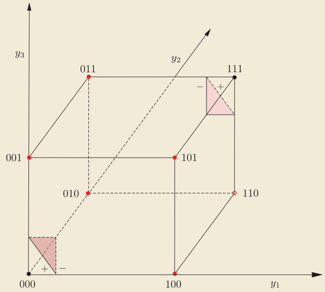
- More specifically, the mapping is performed on the vertices of the **unit** cube in $\mathbb{R}^3$, as shown below



- The neurons of the first hidden layer perform a mapping from the input feature space to the vertices of a unit hypercube. Each region is mapped into a single vertex. Each vertex of the hypercube is now linearly separable from all the rest and can be separated by a (hyper)plane realized by a neuron.
- If p instead of three neurons are used, the mapping is on the vertices of the p-dimensional unit cube.

Feed-Forward Multilayer Neural Networks

- More specifically, the mapping is performed on the vertices of the **unit** cube in $\mathbb{R}^3$, as shown below

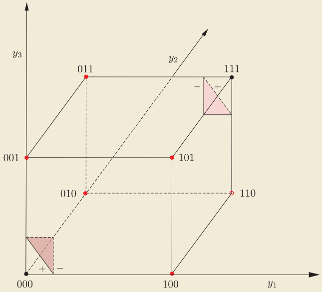


- The neurons of the first hidden layer perform a mapping from the input feature space to the vertices of a unit hypercube. Each region is mapped into a single vertex. Each vertex of the hypercube is now linearly separable from all the rest and can be separated by a (hyper)plane realized by a neuron.

- If p instead of three neurons are used, the mapping is on the vertices of the p-dimensional unit cube.

Feed-Forward Multilayer Neural Networks

- More specifically, the mapping is performed on the vertices of the **unit** cube in $\mathbb{R}^3$, as shown below



- The neurons of the first hidden layer perform a mapping from the input feature space to the vertices of a unit hypercube. Each region is mapped into a single vertex. Each vertex of the hypercube is now linearly separable from all the rest and can be separated by a (hyper)plane realized by a neuron.
- If p instead of three neurons are used, the mapping is on the vertices of the p-dimensional unit cube.

Feed-Forward Multilayer Neural Networks

- An alternative way to view this mapping is as a new representation of the input patterns in terms of **code words**. For three neurons, we can form 2^3 binary code-words, each corresponding to a vertex of the unit cube, which can represent $2^3 - 1 = 7$ regions (there is one remaining vertex, i.e., (110), which does not correspond to any region).
- Moreover, this mapping encodes information concerning some **structure** of the input data; that is, information relating on how the input patterns are grouped together in the feature space in different regions.

Feed-Forward Multilayer Neural Networks

- An alternative way to view this mapping is as a new representation of the input patterns in terms of **code words**. For three neurons, we can form 2^3 binary code-words, each corresponding to a vertex of the unit cube, which can represent $2^3 - 1 = 7$ regions (there is one remaining vertex, i.e., (110), which does not correspond to any region).
- Moreover, this mapping encodes information concerning some **structure** of the input data; that is, information relating on how the input patterns are grouped together in the feature space in different regions.

Feed-Forward Multilayer Neural Networks

- We will now use this new representation as input, which feeds the neurons of a second layer, which is constructed as follows.
- We choose all regions which belong to one class. Assume that regions (000) and (111) define class ω_1. Recall that, each of the two corresponding vertices is now linearly separable from the rest.
- This means that we can use a neuron/perceptron in the transformed space, which will place one vertex in the (+) side and the rest in the (-) one, as shown in the last figure. The resulting structure/network is shown in the figure in the next slide.

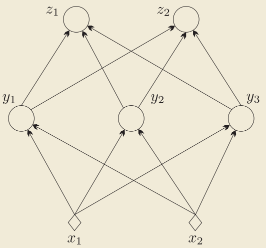
Feed-Forward Multilayer Neural Networks

- We will now use this new representation as input, which feeds the neurons of a second layer, which is constructed as follows.
- We choose all regions which belong to one class. Assume that regions (000) and (111) define class ω_1. Recall that, each of the two corresponding vertices is now linearly separable from the rest.
- This means that we can use a neuron/perceptron in the transformed space, which will place one vertex in the (+) side and the rest in the (-) one, as shown in the last figure. The resulting structure/network is shown in the figure in the next slide.

Feed-Forward Multilayer Neural Networks

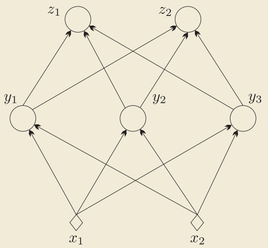
- We will now use this new representation as input, which feeds the neurons of a second layer, which is constructed as follows.
- We choose all regions which belong to one class. Assume that regions (000) and (111) define class ω_1. Recall that, each of the two corresponding vertices is now linearly separable from the rest.
- This means that we can use a neuron/perceptron in the transformed space, which will place one vertex in the (+) side and the rest in the (-) one, as shown in the last figure. The resulting structure/network is shown in the figure in the next slide.

Feed-Forward Multilayer Neural Networks



- The resulting network has a second layer of hidden neurons. The output z_1 of the left neuron will fire an “1” only if the input pattern originates from the region 000 and it will be at “0” for all other patterns. For the neuron on the right, the output z_2 will be “1” for all the patterns coming from region (111) and zero for all the rest.

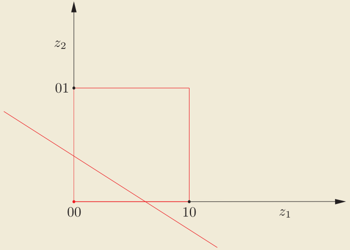
Feed-Forward Multilayer Neural Networks



- Note that, this second layer of neurons has performed a second mapping, this time to the unit rectangle in the $\mathbb{R}^2$. This mapping provides a new representation of the input patterns, and this representation **encodes information related to the classes of the regions**.

Feed-Forward Multilayer Neural Networks

- The following figure shows the last reported mapping to the corners of the unit rectangle in the (z_1, z_2) space.

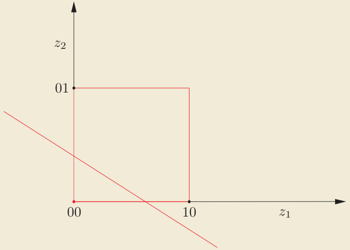


Patterns form class ω_1 are mapped either to (01) or to (10) and patterns from class ω_2 are mapped to (00). Thus the classes have now become linearly separable and can be separated via a straight line realized by a neuron.

- This is very interesting; by successive mappings, we have transformed our originally nonlinearly separable task, to one which is linearly separable. Indeed, the point (00) can be linearly separated from (01) and (10) and this can be realized by an extra neuron operating in the (z_1, z_2) space. The latter is known as the output neuron, since it provides the final classification decision. The final network is shown in the next figure.

Feed-Forward Multilayer Neural Networks

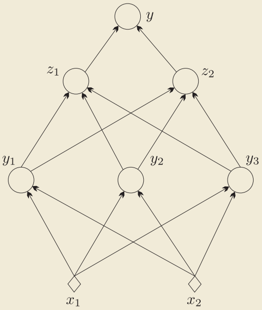
- The following figure shows the last reported mapping to the corners of the unit rectangle in the (z_1, z_2) space.



Patterns form class ω_1 are mapped either to (01) or to (10) and patterns from class ω_2 are mapped to (00). Thus the classes have now become linearly separable and can be separated via a straight line realized by a neuron.

- This is very interesting; by successive mappings, we have transformed our originally nonlinearly separable task, to one which is linearly separable. Indeed, the point (00) can be linearly separated from (01) and (10) and this can be realized by an extra neuron operating in the (z_1, z_2) space. The latter is known as the output neuron, since it provides the final classification decision. The final network is shown in the next figure.

Feed-Forward Multilayer Neural Networks



A three layer feedforward neural network. It comprises the input (non-processing) layer, two hidden layers and one output layer of neurons. Such a three layer NN can solve **any** classification task, where classes are formed by unions of polyhedral regions.

- We say that this network of neurons is a feed-forward one, since information flows in the forward direction from the input to the output layer. It comprises the input layer, which is a non-processing one, two hidden layers (the term hidden is self-explained) and one output layer. We call such a Neural Network (NN) a three layer network, without counting the input layer of non-processing nodes.

Feed-Forward Multilayer Neural Networks

- We have constructively shown that a three layer feed-forward NN can, in principle, solve any classification task whose classes are formed by union of polyhedral regions. The generalization to multiclass cases is straightforward, by employing more output neurons depending on the number of classes.

- Note that in some cases, one hidden layer of nodes may be sufficient. For example, this would be the case if class ω_1 was the union of (000) and (100) regions. Then these two vertices could be separated from the rest via a single plane and a second hidden layer of neurons would not be required (check why).

Feed-Forward Multilayer Neural Networks

- We have constructively shown that a three layer feed-forward NN can, in principle, solve any classification task whose classes are formed by union of polyhedral regions. The generalization to multiclass cases is straightforward, by employing more output neurons depending on the number of classes.
- Note that in some cases, one hidden layer of nodes may be sufficient. For example, this would be the case if class ω_1 was the union of (000) and (100) regions. Then these two vertices could be separated from the rest via a single plane and a second hidden layer of neurons would not be required (check why).

Feed-Forward Multilayer Neural Networks

- What we have said so far was a theoretical construction in order to highlight some analogies to the multilayer neural architecture of our brain and the concept of different representations of the input patterns via the various layers.
- In practice, when the data "live" in high dimensional spaces, there is no way of implementing neurons analytically so as to realize the hyperplanes. Furthermore, in real life, classes are not necessarily formed by union of polyhedral regions and more important classes do overlap.
- Hence, one needs to devise a training procedure based on a cost function.

Feed-Forward Multilayer Neural Networks

- What we have said so far was a theoretical construction in order to highlight some analogies to the multilayer neural architecture of our brain and the concept of different representations of the input patterns via the various layers.
- In practice, when the data "live" in high dimensional spaces, there is no way of implementing neurons analytically so as to realize the hyperplanes. Furthermore, in real life, classes are not necessarily formed by union of polyhedral regions and more important classes do overlap.
- Hence, one needs to devise a training procedure based on a cost function.

Feed-Forward Multilayer Neural Networks

- What we have said so far was a theoretical construction in order to highlight some analogies to the multilayer neural architecture of our brain and the concept of different representations of the input patterns via the various layers.
- In practice, when the data "live" in high dimensional spaces, there is no way of implementing neurons analytically so as to realize the hyperplanes. Furthermore, in real life, classes are not necessarily formed by union of polyhedral regions and more important classes do overlap.
- Hence, one needs to devise a training procedure based on a cost function.

The Backpropagation Algorithm

- A feed-forward neural network (NN) consists of a number of layers of neurons and each neuron is determined by the corresponding set of synaptic weights and its bias term. Networks with more than two hidden layers, are known as **deep networks**.
- From this point of view, a NN realizes a nonlinear parametric function, $\hat{y} = f_{\boldsymbol{\theta}}(\boldsymbol{x})$, where $\boldsymbol{\theta}$ stands for all the weights present in the network. Thus, training a NN seems not to be any different than training any other parametric prediction model.
- All is needed is a) a set of training samples, b) a loss function, $\mathcal{L}(y, \hat{y})$, and c) an iterative scheme, e.g., gradient descent, to perform the optimization of the associated empirical loss function,

$$J(\boldsymbol{\theta}) = \sum_{n=1}^{N} \mathcal{L}\big(y_n, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\big).$$

The Backpropagation Algorithm

- A feed-forward neural network (NN) consists of a number of layers of neurons and each neuron is determined by the corresponding set of synaptic weights and its bias term. Networks with more than two hidden layers, are known as **deep networks**.
- From this point of view, a NN realizes a nonlinear parametric function, $\hat{y} = f_{\boldsymbol{\theta}}(\boldsymbol{x})$, where $\boldsymbol{\theta}$ stands for all the weights present in the network. Thus, training a NN seems not to be any different than training any other parametric prediction model.
- All is needed is a) a set of training samples, b) a loss function, $\mathcal{L}(y, \hat{y})$, and c) an iterative scheme, e.g., gradient descent, to perform the optimization of the associated empirical loss function,

$$J(\boldsymbol{\theta}) = \sum_{n=1}^{N} \mathcal{L}\big(y_n, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\big).$$

The Backpropagation Algorithm

- A feed-forward neural network (NN) consists of a number of layers of neurons and each neuron is determined by the corresponding set of synaptic weights and its bias term. Networks with more than two hidden layers, are known as **deep networks**.
- From this point of view, a NN realizes a nonlinear parametric function, $\hat{y} = f_{\boldsymbol{\theta}}(\boldsymbol{x})$, where $\boldsymbol{\theta}$ stands for all the weights present in the network. Thus, training a NN seems not to be any different than training any other parametric prediction model.
- All is needed is a) a set of training samples, b) a loss function, $\mathcal{L}(y, \hat{y})$, and c) an iterative scheme, e.g., gradient descent, to perform the optimization of the associated empirical loss function,

$$J(\boldsymbol{\theta}) = \sum_{n=1}^{N} \mathcal{L}\big(y_n, f_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\big).$$

The Backpropagation Algorithm

- A difficulty with training NNs lies in their multilayer structure that **complicates** the computation of the involved gradients, which are needed for the optimization. Moreover, the McCulloch-Pitts neuron involves the discontinuous Heaviside activation function, which is not differentiable.
- A first step in developing a practical algorithm for training a NN is to replace the Heaviside activation function with a differentiable one.

The Backpropagation Algorithm

- A difficulty with training NNs lies in their multilayer structure that **complicates** the computation of the involved gradients, which are needed for the optimization. Moreover, the McCulloch-Pitts neuron involves the discontinuous Heaviside activation function, which is not differentiable.
- A first step in developing a practical algorithm for training a NN is to replace the Heaviside activation function with a differentiable one.

The Backpropagation Algorithm

- A first step in developing a practical algorithm for training a NN is to replace the Heaviside activation function with a differentiable one.
- The logistic sigmoid neuron: One possibility is to adopt the logistic sigmoid function, i.e.,

$$f(z) = \sigma(z) := \frac{1}{1 + \exp(-az)}.$$

- Hyperbolic tangent function: Another alternative is

$$f(z) = a \tanh\left(\frac{cz}{2}\right), \text{ where } c \text{ and } a \text{ are controling parameters.}$$

The Backpropagation Algorithm

- A first step in developing a practical algorithm for training a NN is to replace the Heaviside activation function with a differentiable one.
- The logistic sigmoid neuron: One possibility is to adopt the logistic sigmoid function, i.e.,

$$f(z) = \sigma(z) := \frac{1}{1 + \exp(-az)}.$$

- Hyperbolic tangent function: Another alternative is

$$f(z) = a \tanh\left(\frac{cz}{2}\right),$$ where c and a are controling parameters.

The Backpropagation Algorithm

- A first step in developing a practical algorithm for training a NN is to replace the Heaviside activation function with a differentiable one.
- The logistic sigmoid neuron: One possibility is to adopt the logistic sigmoid function, i.e.,

$$f(z) = \sigma(z) := \frac{1}{1 + \exp(-az)}.$$

- Hyperbolic tangent function: Another alternative is

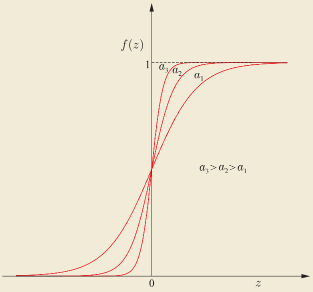
$$f(z) = a \tanh\left(\frac{cz}{2}\right),$$ where c and a are controling parameters.

- A first step in developing a practical algorithm for training a NN is to replace the Heaviside activation function with a differentiable one.
- The logistic sigmoid neuron: One possibility is to adopt the logistic sigmoid function, i.e.,

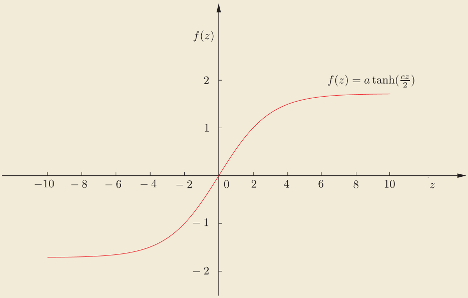
$$f(z) = \sigma(z) := \frac{1}{1 + \exp(-az)}.$$

- Hyperbolic tangent function: Another alternative is

$$f(z) = a \tanh\left(\frac{cz}{2}\right),$$ where c and a are controling parameters.



Logistic sigmoid function



Hyperbolic tangent function

The Backpropagation Algorithm

- The Gradient Descent Scheme: Having adopted a differentiable activation function, we are ready to proceed with developing the gradient descent iterative scheme for the minimization of the cost function. We will formulate the task in a general framework.

- What is more important is to grasp the rationale behind the algorithm and not the details.

The Backpropagation Algorithm

- The Gradient Descent Scheme: Having adopted a differentiable activation function, we are ready to proceed with developing the gradient descent iterative scheme for the minimization of the cost function. We will formulate the task in a general framework.
- What is more important is to grasp the rationale behind the algorithm and not the details.

The Backpropagation Algorithm

- Let $(\boldsymbol{y}_n, \boldsymbol{x}_n),\ n = 1, 2, \ldots, N$, be the set of training samples. Note that we have assumed multiple output variables, assembled as a vector. We assume that the network comprises L layers; $L - 1$ hidden and one output layers. Each layer consists of $k_r,\ r = 1, 2, \ldots, L$, neurons. Thus, the output vectors are:

$$\boldsymbol{y}_n = [y_{n1}, y_{n2}, \ldots, y_{nk_L}]^T \in \mathbb{R}^{k_L},\ n = 1, 2, \ldots, N.$$

  For the sake of the mathematical derivations, we also denote the number of input nodes as k_0; i.e., $k_0 = l$, where l is the dimensionality of the input feature space.

- Let $\boldsymbol{\theta}_j^r$ denote the synaptic weights associated with the jth neuron in the rth layer, with $j = 1, 2, \ldots, k_r$ and $r = 1, 2, \ldots, L$, where the bias term is included in $\boldsymbol{\theta}_j^r$, i.e.,

$$\boldsymbol{\theta}_j^r := [\theta_{j0}^r, \theta_{j1}^r, \ldots, \theta_{jk_{r-1}}^r]^T.$$

The Backpropagation Algorithm

- Let $(\boldsymbol{y}_n, \boldsymbol{x}_n),\ n = 1, 2, \ldots, N$, be the set of training samples. Note that we have assumed multiple output variables, assembled as a vector. We assume that the network comprises L layers; $L-1$ hidden and one output layers. Each layer consists of $k_r,\ r = 1, 2, \ldots, L$, neurons. Thus, the output vectors are:

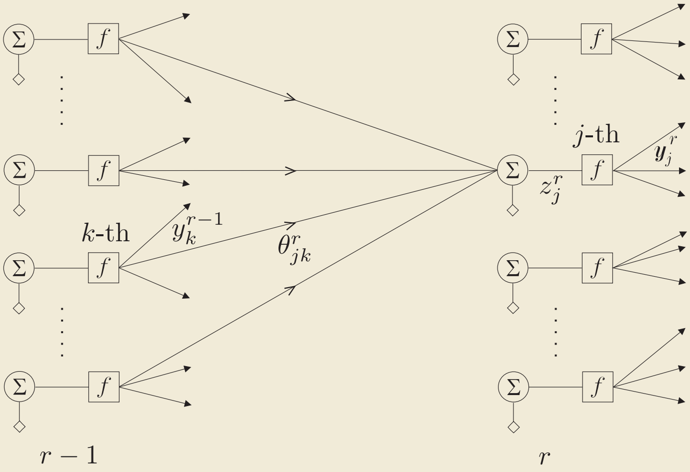
$$\boldsymbol{y}_n = [y_{n1}, y_{n2}, \ldots, y_{nk_L}]^T \in \mathbb{R}^{k_L},\ n = 1, 2, \ldots, N.$$

  For the sake of the mathematical derivations, we also denote the number of input nodes as k_0; i.e., $k_0 = l$, where l is the dimensionality of the input feature space.

- Let $\boldsymbol{\theta}_j^r$ denote the synaptic weights associated with the jth neuron in the rth layer, with $j = 1, 2, \ldots, k_r$ and $r = 1, 2, \ldots, L$, where the bias term is included in $\boldsymbol{\theta}_j^r$, i.e.,

$$\boldsymbol{\theta}_j^r := [\theta_{j0}^r, \theta_{j1}^r, \ldots, \theta_{jk_{r-1}}^r]^T.$$

The Backpropagation Algorithm



The links and the associated variables of the jth neuron at the rth layer.

The Backpropagation Algorithm

- The basic iterative step for the gradient decent scheme is written as

$$\boldsymbol{\theta}_j^r(new) = \boldsymbol{\theta}_j^r(old) + \Delta\boldsymbol{\theta}_j^r,$$

where

$$\Delta\boldsymbol{\theta}_j^r = -\mu \frac{\partial J}{\partial \boldsymbol{\theta}_j^r}\Big|_{\boldsymbol{\theta}_j^r(old)}.$$

The parameter μ is the user-defined step size (it can also be iteration-dependent) and J denotes the cost function.

The Backpropagation Algorithm

- For example, if the squared error loss is adopted, we have

$$J(\boldsymbol{\theta}) = \sum_{n=1}^{N} J_n(\boldsymbol{\theta}),$$

and

$$J_n(\boldsymbol{\theta}) = \frac{1}{2} \sum_{k=1}^{k_L} \left(\hat{y}_{nk} - y_{nk}\right)^2,$$

where $\hat{y}_{nk},\ k = 1, 2, \ldots, k_L$, are the estimates provided at the corresponding output nodes of the network. We will consider them as the elements of a corresponding vector, $\hat{\boldsymbol{y}}_n$.

The Backpropagation Algorithm

- The main difficulty in the backpropagation algorithm lies in the computation of the gradients. Note that the output of the network relates **directly** to the parameters associated with the neurons of the last (output) layer. Thus, the computation of the corresponding gradients poses no problems. Business as usual.
- However, the output of the network is related **indirectly** with the parameters of the neurons comprising the hidden layers. This is because the outputs/responses of the hidden layers are transformed by the neurons of the layers above. The closer to the input is a layer, the more transformations the respected neuron responses undergo, as they propagate through the layers higher in the hierarchy.

The Backpropagation Algorithm

- The main difficulty in the backpropagation algorithm lies in the computation of the gradients. Note that the output of the network relates **directly** to the parameters associated with the neurons of the last (output) layer. Thus, the computation of the corresponding gradients poses no problems. Business as usual.
- However, the output of the network is related **indirectly** with the parameters of the neurons comprising the hidden layers. This is because the outputs/responses of the hidden layers are transformed by the neurons of the layers above. The closer to the input is a layer, the more transformations the respected neuron responses undergo, as they propagate through the layers higher in the hierarchy.

The Backpropagation Algorithm

To compute the gradients, two types of computations are performed:

- Forward Computations: For a given input, $\boldsymbol{x}_n$, employ the currently available estimates of the parameters and compute the output of the network, say, $\hat{y}_n$, which depends of the current estimates.
- Backward Computations: Using the desired response, y_n, and the predicted one, $\hat{y}_n$, compute the corresponding gradients of the cost function w.r. to all the parameters. To this end, the computations propagate backwards:
  - Compute the gradients of the parameters of the neurons of the last layer, L.
  - Using the previously computed gradients and the chain rule of derivation (to account for the imposed, by the network, transformations), compute the gradients of the parameters of the neurons of layer $L-1$.
  - The above procedure caries on, backwards, till all the gradients, including those in the first hidden layer, have been computed.

The Backpropagation Algorithm

To compute the gradients, two types of computations are performed:

- Forward Computations: For a given input, $\boldsymbol{x}_n$, employ the currently available estimates of the parameters and compute the output of the network, say, $\hat{y}_n$, which depends of the current estimates.
- Backward Computations: Using the desired response, y_n, and the predicted one, $\hat{y}_n$, compute the corresponding gradients of the cost function w.r. to all the parameters. To this end, the computations propagate backwards:
  - Compute the gradients of the parameters of the neurons of the last layer, L.
  - Using the previously computed gradients and the chain rule of derivation (to account for the imposed, by the network, transformations), compute the gradients of the parameters of the neurons of layer $L-1$.
  - The above procedure caries on, backwards, till all the gradients, including those in the first hidden layer, have been computed.

The Backpropagation Algorithm

To compute the gradients, two types of computations are performed:

- Forward Computations: For a given input, $\boldsymbol{x}_n$, employ the currently available estimates of the parameters and compute the output of the network, say, $\hat{y}_n$, which depends of the current estimates.
- Backward Computations: Using the desired response, y_n, and the predicted one, $\hat{y}_n$, compute the corresponding gradients of the cost function w.r. to all the parameters. To this end, the computations propagate backwards:
  - Compute the gradients of the parameters of the neurons of the last layer, L.
  - Using the previously computed gradients and the chain rule of derivation (to account for the imposed, by the network, transformations), compute the gradients of the parameters of the neurons of layer $L-1$.
  - The above procedure caries on, backwards, till all the gradients, including those in the first hidden layer, have been computed.

The Backpropagation Algorithm

To compute the gradients, two types of computations are performed:

- Forward Computations: For a given input, $\boldsymbol{x}_n$, employ the currently available estimates of the parameters and compute the output of the network, say, $\hat{y}_n$, which depends of the current estimates.
- Backward Computations: Using the desired response, y_n, and the predicted one, $\hat{y}_n$, compute the corresponding gradients of the cost function w.r. to all the parameters. To this end, the computations propagate backwards:
    - Compute the gradients of the parameters of the neurons of the last layer, L.
    - Using the previously computed gradients and the chain rule of derivation (to account for the imposed, by the network, transformations), compute the gradients of the parameters of the neurons of layer $L-1$.
    - The above procedure caries on, backwards, till all the gradients, including those in the first hidden layer, have been computed.

The Backpropagation Algorithm

To compute the gradients, two types of computations are performed:

- Forward Computations: For a given input, $\boldsymbol{x}_n$, employ the currently available estimates of the parameters and compute the output of the network, say, $\hat{y}_n$, which depends of the current estimates.
- Backward Computations: Using the desired response, y_n, and the predicted one, $\hat{y}_n$, compute the corresponding gradients of the cost function w.r. to all the parameters. To this end, the computations propagate backwards:
    - Compute the gradients of the parameters of the neurons of the last layer, L.
    - Using the previously computed gradients and the chain rule of derivation (to account for the imposed, by the network, transformations), compute the gradients of the parameters of the neurons of layer $L-1$.
    - The above procedure caries on, backwards, till all the gradients, including those in the first hidden layer, have been computed.

- **Computation of the gradients**: Let z_{nj}^r denote the output of the linear combiner of the jth neuron in the rth layer at time instant n, when the pattern $\boldsymbol{x}_n$ appears at the input nodes. Then, we can write that

$$z_{nj}^r = \sum_{m=1}^{k_{r-1}} \theta_{jm}^r y_{nm}^{r-1} + \theta_{j0}^r = \sum_{m=0}^{k_{r-1}} \theta_{jm}^r y_{nm}^{r-1} = \boldsymbol{\theta}_j^{rT} \boldsymbol{y}_n^{r-1}, \quad (3)$$

where by definition

$$\boldsymbol{y}_n^{r-1} := [1, y_{n1}^{r-1}, \ldots, y_{nk_{r-1}}^{r-1}]^T,$$

and $y_{n0}^r \equiv 1, \ \forall \ r, n$. For the neurons at the output layer, $r = L$, $y_{nm}^L = \hat{y}_{nm}, \ m = 1, 2, \ldots, k_L$, and for $r = 1$, we have $y_{nm}^0 = x_{nm}, \ m = 1, 2, \ldots, k_0$; that is, y_{nm}^0 are set equal to the input feature values.

- Hence, we can now write that

$$\frac{\partial J_n}{\partial \boldsymbol{\theta}_j^r} = \frac{\partial J_n}{\partial z_{nj}^r} \frac{\partial z_{nj}^r}{\partial \boldsymbol{\theta}_j^r} = \frac{\partial J_n}{\partial z_{nj}^r} \boldsymbol{y}_n^{r-1}, \text{ and } \delta_{nj}^r := \frac{\partial J_n}{\partial z_{nj}^r}.$$

- Then we have

$$\Delta \boldsymbol{\theta}_j^r = -\mu \sum_{n=1}^{N} \delta_{nj}^r \boldsymbol{y}_n^{r-1}, \ r = 1, 2, \ldots, L. \quad (4)$$

- **Computation of the gradients**: Let z_{nj}^r denote the output of the linear combiner of the jth neuron in the rth layer at time instant n, when the pattern $\boldsymbol{x}_n$ appears at the input nodes. Then, we can write that

$$z_{nj}^r = \sum_{m=1}^{k_{r-1}} \theta_{jm}^r y_{nm}^{r-1} + \theta_{j0}^r = \sum_{m=0}^{k_{r-1}} \theta_{jm}^r y_{nm}^{r-1} = \boldsymbol{\theta}_j^{rT} \boldsymbol{y}_n^{r-1}, \quad (3)$$

where by definition

$$\boldsymbol{y}_n^{r-1} := [1, y_{n1}^{r-1}, \dots, y_{nk_{r-1}}^{r-1}]^T,$$

and $y_{n0}^r \equiv 1, \ \forall\ r, n$. For the neurons at the output layer, $r = L$, $y_{nm}^L = \hat{y}_{nm}, \ m = 1, 2, \dots, k_L$, and for $r = 1$, we have $y_{nm}^0 = x_{nm}, \ m = 1, 2, \dots, k_0$; that is, y_{nm}^0 are set equal to the input feature values.

- Hence, we can now write that

$$\frac{\partial J_n}{\partial \boldsymbol{\theta}_j^r} = \frac{\partial J_n}{\partial z_{nj}^r} \frac{\partial z_{nj}^r}{\partial \boldsymbol{\theta}_j^r} = \frac{\partial J_n}{\partial z_{nj}^r} \boldsymbol{y}_n^{r-1}, \text{ and } \delta_{nj}^r := \frac{\partial J_n}{\partial z_{nj}^r}.$$

- Then we have

$$\Delta \boldsymbol{\theta}_j^r = -\mu \sum_{n=1}^{N} \delta_{nj}^r \boldsymbol{y}_n^{r-1}, \ r = 1, 2, \dots, L. \quad (4)$$

- **Computation of the gradients**: Let z_{nj}^r denote the output of the linear combiner of the jth neuron in the rth layer at time instant n, when the pattern $\boldsymbol{x}_n$ appears at the input nodes. Then, we can write that

$$z_{nj}^r = \sum_{m=1}^{k_{r-1}} \theta_{jm}^r y_{nm}^{r-1} + \theta_{j0}^r = \sum_{m=0}^{k_{r-1}} \theta_{jm}^r y_{nm}^{r-1} = \boldsymbol{\theta}_j^{rT} \boldsymbol{y}_n^{r-1}, \tag{3}$$

where by definition

$$\boldsymbol{y}_n^{r-1} := [1, y_{n1}^{r-1}, \ldots, y_{nk_{r-1}}^{r-1}]^T,$$

and $y_{n0}^r \equiv 1, \ \forall \ r, n$. For the neurons at the output layer, $r = L$, $y_{nm}^L = \hat{y}_{nm}, \ m = 1, 2, \ldots, k_L$, and for $r = 1$, we have $y_{nm}^0 = x_{nm}, \ m = 1, 2, \ldots, k_0$; that is, y_{nm}^0 are set equal to the input feature values.

- Hence, we can now write that

$$\frac{\partial J_n}{\partial \boldsymbol{\theta}_j^r} = \frac{\partial J_n}{\partial z_{nj}^r} \frac{\partial z_{nj}^r}{\partial \boldsymbol{\theta}_j^r} = \frac{\partial J_n}{\partial z_{nj}^r} \boldsymbol{y}_n^{r-1}, \text{ and } \delta_{nj}^r := \frac{\partial J_n}{\partial z_{nj}^r}.$$

- Then we have

$$\Delta \boldsymbol{\theta}_j^r = -\mu \sum_{n=1}^{N} \delta_{nj}^r \boldsymbol{y}_n^{r-1}, \ r = 1, 2, \ldots, L. \tag{4}$$

- **Computation of δ_{nj}^{r}**: Here is where the heart of the backpropagation algorithm beats. For the computation of the gradients, δ_{nj}^{r}, one starts at the last layer, $r = L$, and proceeds backwards towards $r = 1$; this philosophy justifies the name given to the algorithm.
  - $r = L$: We have that
$$\delta_{nj}^{L} = \frac{\partial J_n}{\partial z_{nj}^{L}}.$$
    For the squared error loss function,
$$J_n = \frac{1}{2}\sum_{k=1}^{k_L}\left(f(z_{nk}^{L}) - y_{nk}\right)^2.$$
    Hence,
$$\begin{aligned}\delta_{nj}^{L} &= (\hat{y}_{nj} - y_{nj})f^{'}(z_{nj}^{L}),\\ &= e_{nj}f^{'}(z_{nj}^{L}),\ j = 1, 2, \ldots, k_L.\end{aligned} \quad (5)$$
    where $f^{'}(\cdot)$ denotes the derivative of $f(\cdot)$, and e_{nj} is the error associated with the jth output variable at time n. Note that for the last layer, the computation of the gradient is straightforward.

The Backpropagation Algorithm

- **Computation of δ_{nj}^r**: Here is where the heart of the backpropagation algorithm beats. For the computation of the gradients, δ_{nj}^r, one starts at the last layer, $r = L$, and proceeds backwards towards $r = 1$; this philosophy justifies the name given to the algorithm.
  - $r = L$: We have that
$$\delta_{nj}^L = \frac{\partial J_n}{\partial z_{nj}^L}.$$
    For the squared error loss function,
$$J_n = \frac{1}{2}\sum_{k=1}^{k_L}\big(f(z_{nk}^L) - y_{nk}\big)^2.$$
    Hence,
$$\begin{aligned}\delta_{nj}^L &= (\hat{y}_{nj} - y_{nj})f^{'}(z_{nj}^L),\\ &= e_{nj}f^{'}(z_{nj}^L),\ j = 1, 2, \ldots, k_L.\end{aligned} \qquad (5)$$
    where $f^{'}(\cdot)$ denotes the derivative of $f(\cdot)$, and e_{nj} is the error associated with the jth output variable at time n. Note that for the last layer, the computation of the gradient is straightforward.

The Backpropagation Algorithm

- **Computation of δ^r_{nj}**: Here is where the heart of the backpropagation algorithm beats. For the computation of the gradients, δ^r_{nj}, one starts at the last layer, $r = L$, and proceeds backwards towards $r = 1$; this philosophy justifies the name given to the algorithm.
  - $r = L$: We have that

    $$\delta^L_{nj} = \frac{\partial J_n}{\partial z^L_{nj}}.$$

    For the squared error loss function,

    $$J_n = \frac{1}{2}\sum_{k=1}^{k_L}\left(f(z^L_{nk}) - y_{nk}\right)^2.$$

    Hence,

    $$\begin{aligned}\delta^L_{nj} &= (\hat{y}_{nj} - y_{nj})f^{'}(z^L_{nj}),\\ &= e_{nj}f^{'}(z^L_{nj}),\ j = 1, 2, \ldots, k_L.\end{aligned} \quad (5)$$

    where $f^{'}(\cdot)$ denotes the derivative of $f(\cdot)$, and e_{nj} is the error associated with the jth output variable at time n. Note that for the last layer, the computation of the gradient is straightforward.

- **Computation of δ^r_{nj}** (continued):
  - $r < L$: Due to the successive dependence between the layers, the value of z^{r-1}_{nj} influences all the values $z^r_{nk},\ k = 1, 2, \ldots, k_r$ of the next layer. Employing the chain rule for differentiation, we get

$$\delta^{r-1}_{nj} = \frac{\partial J_n}{\partial z^{r-1}_{nj}} = \sum_{k=1}^{k_r} \frac{\partial J_n}{\partial z^r_{nk}} \frac{\partial z^r_{nk}}{\partial z^{r-1}_{nj}}, \tag{6}$$

or

$$\frac{\partial J_n}{\partial z^{r-1}_{nj}} = \sum_{k=1}^{k_r} \delta^r_{nk} \frac{\partial z^r_{nk}}{\partial z^{r-1}_{nj}}. \tag{7}$$

However,

$$\frac{\partial z^r_{nk}}{\partial z^{r-1}_{nj}} = \frac{\partial\left(\sum_{m=0}^{k_{r-1}} \theta^r_{km} y^{r-1}_{nm}\right)}{\partial z^{r-1}_{nj}}, \text{ where } y^{r-1}_{nm} = f(z^{r-1}_{nm}),$$

which leads to,

$$\frac{\partial z^r_{nk}}{\partial z^{r-1}_{nj}} = \theta^r_{kj} f'(z^{r-1}_{nj}),$$

and combining with (6)-(7), we obtain for $j = 1, 2, \ldots, k_{r-1}$,

$$\delta^{r-1}_{nj} = \left(\sum_{k=1}^{k_r} \delta^r_{nk}\theta^r_{kj}\right) f'(z^{r-1}_{nj}) := \delta^{r-1}_{nj} = e^{r-1}_{nj} f'(z^{r-1}_{nj}). \tag{8}$$

- **Computation of δ_{nj}^r** (continued):
  - $r < L$: Due to the successive dependence between the layers, the value of z_{nj}^{r-1} influences all the values $z_{nk}^r,\ k = 1, 2, \ldots, k_r$ of the next layer. Employing the chain rule for differentiation, we get

$$\delta_{nj}^{r-1} = \frac{\partial J_n}{\partial z_{nj}^{r-1}} = \sum_{k=1}^{k_r} \frac{\partial J_n}{\partial z_{nk}^r} \frac{\partial z_{nk}^r}{\partial z_{nj}^{r-1}}, \qquad (6)$$

or

$$\frac{\partial J_n}{\partial z_{nj}^{r-1}} = \sum_{k=1}^{k_r} \delta_{nk}^r \frac{\partial z_{nk}^r}{\partial z_{nj}^{r-1}}. \qquad (7)$$

However,

$$\frac{\partial z_{nk}^r}{\partial z_{nj}^{r-1}} = \frac{\partial \left(\sum_{m=0}^{k_{r-1}} \theta_{km}^r y_{nm}^{r-1} \right)}{\partial z_{nj}^{r-1}}, \text{ where } y_{nm}^{r-1} = f(z_{nm}^{r-1}),$$

which leads to,

$$\frac{\partial z_{nk}^r}{\partial z_{nj}^{r-1}} = \theta_{kj}^r f^{'}(z_{nj}^{r-1}),$$

and combining with (6)-(7), we obtain for $j = 1, 2, \ldots, k_{r-1}$,

$$\delta_{nj}^{r-1} = \left(\sum_{k=1}^{k_r} \delta_{nk}^r \theta_{kj}^r \right) f^{'}(z_{nj}^{r-1}) := \delta_{nj}^{r-1} = e_{nj}^{r-1} f^{'}(z_{nj}^{r-1}). \qquad (8)$$

The Backpropagation Algorithm

- **Computation of** δ_{nj}^r (continued):
  - $r < L$: Due to the successive dependence between the layers, the value of z_{nj}^{r-1} influences all the values $z_{nk}^r,\ k = 1, 2, \ldots, k_r$ of the next layer. Employing the chain rule for differentiation, we get

$$\delta_{nj}^{r-1} = \frac{\partial J_n}{\partial z_{nj}^{r-1}} = \sum_{k=1}^{k_r} \frac{\partial J_n}{\partial z_{nk}^r} \frac{\partial z_{nk}^r}{\partial z_{nj}^{r-1}}, \tag{6}$$

or

$$\frac{\partial J_n}{\partial z_{nj}^{r-1}} = \sum_{k=1}^{k_r} \delta_{nk}^r \frac{\partial z_{nk}^r}{\partial z_{nj}^{r-1}}. \tag{7}$$

However,

$$\frac{\partial z_{nk}^r}{\partial z_{nj}^{r-1}} = \frac{\partial \left(\sum_{m=0}^{k_{r-1}} \theta_{km}^r y_{nm}^{r-1} \right)}{\partial z_{nj}^{r-1}}, \text{ where } y_{nm}^{r-1} = f(z_{nm}^{r-1}),$$

which leads to,

$$\frac{\partial z_{nk}^r}{\partial z_{nj}^{r-1}} = \theta_{kj}^r f'(z_{nj}^{r-1}),$$

and combining with (6)-(7), we obtain for $j = 1, 2, \ldots, k_{r-1}$,

$$\delta_{nj}^{r-1} = \left(\sum_{k=1}^{k_r} \delta_{nk}^r \theta_{kj}^r \right) f'(z_{nj}^{r-1}) := \delta_{nj}^{r-1} = e_{nj}^{r-1} f'(z_{nj}^{r-1}). \tag{8}$$

- **Computation of** δ^r_{nj} (continued):
  - $r < L$: Due to the successive dependence between the layers, the value of z^{r-1}_{nj} influences all the values $z^r_{nk},\ k = 1, 2, \ldots, k_r$ of the next layer. Employing the chain rule for differentiation, we get

$$\delta^{r-1}_{nj} = \frac{\partial J_n}{\partial z^{r-1}_{nj}} = \sum_{k=1}^{k_r} \frac{\partial J_n}{\partial z^r_{nk}} \frac{\partial z^r_{nk}}{\partial z^{r-1}_{nj}}, \tag{6}$$

or

$$\frac{\partial J_n}{\partial z^{r-1}_{nj}} = \sum_{k=1}^{k_r} \delta^r_{nk} \frac{\partial z^r_{nk}}{\partial z^{r-1}_{nj}}. \tag{7}$$

However,

$$\frac{\partial z^r_{nk}}{\partial z^{r-1}_{nj}} = \frac{\partial\left(\sum_{m=0}^{k_{r-1}} \theta^r_{km} y^{r-1}_{nm}\right)}{\partial z^{r-1}_{nj}}, \text{ where } y^{r-1}_{nm} = f(z^{r-1}_{nm}),$$

which leads to,

$$\frac{\partial z^r_{nk}}{\partial z^{r-1}_{nj}} = \theta^r_{kj} f'(z^{r-1}_{nj}),$$

and combining with (6)-(7), we obtain for $j = 1, 2, \ldots, k_{r-1}$,

$$\delta^{r-1}_{nj} = \left(\sum_{k=1}^{k_r} \delta^r_{nk} \theta^r_{kj}\right) f'(z^{r-1}_{nj}) := \delta^{r-1}_{nj} = e^{r-1}_{nj} f'(z^{r-1}_{nj}). \tag{8}$$

- **Computation of** δ_{nj}^{r} (continued):
  - $r < L$: Due to the successive dependence between the layers, the value of z_{nj}^{r-1} influences all the values z_{nk}^{r}, $k = 1, 2, \ldots, k_r$ of the next layer. Employing the chain rule for differentiation, we get

$$\delta_{nj}^{r-1} = \frac{\partial J_n}{\partial z_{nj}^{r-1}} = \sum_{k=1}^{k_r} \frac{\partial J_n}{\partial z_{nk}^{r}} \frac{\partial z_{nk}^{r}}{\partial z_{nj}^{r-1}}, \qquad (6)$$

or

$$\frac{\partial J_n}{\partial z_{nj}^{r-1}} = \sum_{k=1}^{k_r} \delta_{nk}^{r} \frac{\partial z_{nk}^{r}}{\partial z_{nj}^{r-1}}. \qquad (7)$$

However,

$$\frac{\partial z_{nk}^{r}}{\partial z_{nj}^{r-1}} = \frac{\partial \left(\sum_{m=0}^{k_{r-1}} \theta_{km}^{r} y_{nm}^{r-1} \right)}{\partial z_{nj}^{r-1}}, \text{ where } y_{nm}^{r-1} = f(z_{nm}^{r-1}),$$

which leads to,

$$\frac{\partial z_{nk}^{r}}{\partial z_{nj}^{r-1}} = \theta_{kj}^{r} f'(z_{nj}^{r-1}),$$

and combining with (6)-(7), we obtain for $j = 1, 2, \ldots, k_{r-1}$,

$$\delta_{nj}^{r-1} = \left(\sum_{k=1}^{k_r} \delta_{nk}^{r} \theta_{kj}^{r} \right) f'(z_{nj}^{r-1}) := \delta_{nj}^{r-1} = e_{nj}^{r-1} f'(z_{nj}^{r-1}). \qquad (8)$$

- **The Gradient Descent Backpropagation Algorithm**
  - Initialization
    - Initialize all synaptic weights and biases randomly with small, but not very small, values. Select μ
    - Set $y^0_{nj} = x_{nj},\ j = 1, 2, \ldots, k_0 = l,\ n = 1, 2, \ldots, N.$
  - **Repeat**; Each repetition completes an epoch.
    - **For** $n = 1, 2, \ldots, N$, **Do**
      - **For** $r = 1, 2, \ldots, L$, **Do**; Forward computations.
      - **For** $j = 1, 2, \ldots, k_r$, **Do**
      - Compute z^r_{nj} from (3).
      - Compute $y^r_{nj} = f(z^r_{nj})$.
      - **End For**
      - **End For**
    - **End For**
    - **For** $j = 1, 2, \ldots, k_L$, **Do**
      - Compute δ^L_{nj} from (5).
    - **End For**
    - **For** $r = L, L-1, \ldots, 2$, **Do**; Backward computations.
      - **For** $j = 1, 2, \ldots, k_r$, **Do**
      - Compute δ^{r-1}_{nj} from (8).
      - **End For**
    - **End For**
  - **End For**
  - **For** $r = 1, 2, \ldots, L$, **Do**; Update the weights.
    - **For** $j = 1, 2, \ldots, k_r$, **Do**
      - Compute $\Delta\boldsymbol{\theta}^r_j$ from (4)
      - $\boldsymbol{\theta}^r_j = \boldsymbol{\theta}^r_j + \Delta\boldsymbol{\theta}^r_j$
    - **End For**
  - **End For**
  - **Until** a stop criterion is met.

The Backpropagation Algorithm

Some Remarks on the Backpropagation Algorithm

- One possibility to terminate the algorithm is to track the value of the cost function, and stop the algorithm when this gets smaller than a preselected threshold. An alternative path is to check for the gradient values and stop when these become small.
- As it is the case with all gradient descent schemes, the choice of the step size, μ, is very critical; it has to be small to guarantee convergence, but not too small, otherwise convergence speed slows down. Adaptive values of μ, whose value depends on the iteration are more appropriate. Soon, such techniques will be discussed.

The Backpropagation Algorithm

Some Remarks on the Backpropagation Algorithm

- One possibility to terminate the algorithm is to track the value of the cost function, and stop the algorithm when this gets smaller than a preselected threshold. An alternative path is to check for the gradient values and stop when these become small.
- As it is the case with all gradient descent schemes, the choice of the step size, μ, is very critical; it has to be small to guarantee convergence, but not too small, otherwise convergence speed slows down. Adaptive values of μ, whose value depends on the iteration are more appropriate. Soon, such techniques will be discussed.

The Backpropagation Algorithm

- Due to the highly nonlinear nature of the NN task, the cost function in the parameter space is, in general, of a complicated form and there exist local minima, where the algorithm can be trapped.
- If such a local minimum is deep enough, the obtained solution can be acceptable. However, this may not be the case and the solution can be trapped in a shallow minimum resulting in a bad solution.
- However, this "shallow minima" view has been challenged in the context of deep architectures. As we will discuss soon, in the case of networks with many layers, shallow minima may not necessarily be a major problem. Saddle points become the critical issue.
- In practice, random initialization of the weights is carried out. Yet, initialization remains a critical part of the algorithm.

The Backpropagation Algorithm

- Due to the highly nonlinear nature of the NN task, the cost function in the parameter space is, in general, of a complicated form and there exist local minima, where the algorithm can be trapped.
- If such a local minimum is deep enough, the obtained solution can be acceptable. However, this may not be the case and the solution can be trapped in a shallow minimum resulting in a bad solution.
- However, this "shallow minima" view has been challenged in the context of deep architectures. As we will discuss soon, in the case of networks with many layers, shallow minima may not necessarily be a major problem. Saddle points become the critical issue.
- In practice, random initialization of the weights is carried out. Yet, initialization remains a critical part of the algorithm.

The Backpropagation Algorithm

- Due to the highly nonlinear nature of the NN task, the cost function in the parameter space is, in general, of a complicated form and there exist local minima, where the algorithm can be trapped.
- If such a local minimum is deep enough, the obtained solution can be acceptable. However, this may not be the case and the solution can be trapped in a shallow minimum resulting in a bad solution.
- However, this "shallow minima" view has been challenged in the context of deep architectures. As we will discuss soon, in the case of networks with many layers, shallow minima may not necessarily be a major problem. Saddle points become the critical issue.
- In practice, random initialization of the weights is carried out. Yet, initialization remains a critical part of the algorithm.

The Backpropagation Algorithm

- Due to the highly nonlinear nature of the NN task, the cost function in the parameter space is, in general, of a complicated form and there exist local minima, where the algorithm can be trapped.
- If such a local minimum is deep enough, the obtained solution can be acceptable. However, this may not be the case and the solution can be trapped in a shallow minimum resulting in a bad solution.
- However, this "shallow minima" view has been challenged in the context of deep architectures. As we will discuss soon, in the case of networks with many layers, shallow minima may not necessarily be a major problem. Saddle points become the critical issue.
- In practice, random initialization of the weights is carried out. Yet, initialization remains a critical part of the algorithm.

The Backpropagation Algorithm

- Pattern-by-pattern operation: The previous scheme is of the batch type of operation, where the weights are updated once per epoch. The alternative route is the pattern-by-pattern/online mode of operation; the weights are updated at every time instant when a new pattern appears in the input.
- Mini-batch operation: There are also intermediate ways, where the update is performed every $N_1 < N$ samples; this technique is also referred as mini-batch mode of operation. Batch and mini-batch modes have an averaging effect on the the computation of the gradients.

The Backpropagation Algorithm

- Pattern-by-pattern operation: The previous scheme is of the batch type of operation, where the weights are updated once per epoch. The alternative route is the pattern-by-pattern/online mode of operation; the weights are updated at every time instant when a new pattern appears in the input.
- Mini-batch operation: There are also intermediate ways, where the update is performed every $N_1 < N$ samples; this technique is also referred as mini-batch mode of operation. Batch and mini-batch modes have an averaging effect on the the computation of the gradients.

Vanishing, Exploding and Unstable Gradients

- Due to the hierarchical computations of the gradients, it turns out that their computation involves a sequence of products of parameters with derivatives of the activation function (e.g., Eq. (8)). The closer to the input layer we are, the more products the computation of the respected gradients involve.

- Taking into account that the derivatives of the activation function can be less than one (e.g., for sigmoid functions can be very small), and if the parameters values are not very large, this can make the gradients, associated to the parameters in the lower layers, vanishingly small, especially if networks with many layers are involved. This can make learning extremely slow.

Vanishing, Exploding and Unstable Gradients

- Due to the hierarchical computations of the gradients, it turns out that their computation involves a sequence of products of parameters with derivatives of the activation function (e.g., Eq. (8)). The closer to the input layer we are, the more products the computation of the respected gradients involve.
- Taking into account that the derivatives of the activation function can be less than one (e.g., for sigmoid functions can be very small), and if the parameters values are not very large, this can make the gradients, associated to the parameters in the lower layers, vanishingly small, especially if networks with many layers are involved. This can make learning extremely slow.

The Backpropagation Algorithm

- On the other extreme, if the values of the parameter estimates happen to take large values, this may lead the values of the gradients to explode. As a result, this can disturb the learning process, by pushing the estimates to wrong regions in the parameters' space.
- Another related problem is that gradients in different layers can take values of different scales. Thus, some layers can learn faster that others, and this can make the learning process unstable.
- To cope with such difficulties, a number of modifications of the basic gradient scheme and a number of practical hints have been proposed.

The Backpropagation Algorithm

- On the other extreme, if the values of the parameter estimates happen to take large values, this may lead the values of the gradients to explode. As a result, this can disturb the learning process, by pushing the estimates to wrong regions in the parameters' space.
- Another related problem is that gradients in different layers can take values of different scales. Thus, some layers can learn faster that others, and this can make the learning process unstable.
- To cope with such difficulties, a number of modifications of the basic gradient scheme and a number of practical hints have been proposed.

The Backpropagation Algorithm

- On the other extreme, if the values of the parameter estimates happen to take large values, this may lead the values of the gradients to explode. As a result, this can disturb the learning process, by pushing the estimates to wrong regions in the parameters' space.
- Another related problem is that gradients in different layers can take values of different scales. Thus, some layers can learn faster that others, and this can make the learning process unstable.
- To cope with such difficulties, a number of modifications of the basic gradient scheme and a number of practical hints have been proposed.

Beyond The Basic Gradient Descent Scheme

- Gradient descent with a momentum term: One way to improve the convergence rate is to employ the so called momentum term, a. The correction term is now modified as

$$\Delta\boldsymbol{\theta}_j^r(new) = a\Delta\boldsymbol{\theta}_j^r(old) + \Delta\boldsymbol{\theta}_j^r$$

The effect is to increase the step size in regions, where the cost function exhibits low curvature.

- Indeed, assume that the gradient is approximately constant over a number of steps, say I. Then, it can be shown that

$$\Delta\boldsymbol{\theta}_j^r(I) \approx -\frac{\mu}{1-\alpha}\boldsymbol{g},$$

where $\boldsymbol{g}$ is the gradient over the I steps. That is, the use of the momentum term increases the correction by a factor $1-\alpha$. Note that adaptive versions for the momentum term a are possible and popular.

Beyond The Basic Gradient Descent Scheme

- Gradient descent with a momentum term: One way to improve the convergence rate is to employ the so called momentum term, a. The correction term is now modified as

$$\Delta\boldsymbol{\theta}_j^r(new) = a\Delta\boldsymbol{\theta}_j^r(old) + \Delta\boldsymbol{\theta}_j^r$$

The effect is to increase the step size in regions, where the cost function exhibits low curvature.

- Indeed, assume that the gradient is approximately constant over a number of steps, say I. Then, it can be shown that

$$\Delta\boldsymbol{\theta}_j^r(I) \approx -\frac{\mu}{1-\alpha}\boldsymbol{g},$$

where $\boldsymbol{g}$ is the gradient over the I steps. That is, the use of the momentum term increases the correction by a factor $1-\alpha$. Note that adaptive versions for the momentum term a are possible and popular.

Beyond The Basic Gradient Descent Scheme

- A number of alternatives have been proposed, and the topic of speeding up the convergence of the backpropagation algorithm has been a hot topic of research, and many variants have been proposed over the years. For example:
  - Newton-type and related simplified versions for computing the associated Hessian matrix.
  - A number of versions, using more recent results on optimization, have also been suggested; for example, schemes based on the ADAGARD or on the Nesterov rationale, which have been considered and discussed in the text in Chapter 8.

Beyond The Basic Gradient Descent Scheme

- A number of alternatives have been proposed, and the topic of speeding up the convergence of the backpropagation algorithm has been a hot topic of research, and many variants have been proposed over the years. For example:
  - Newton-type and related simplified versions for computing the associated Hessian matrix.
  - A number of versions, using more recent results on optimization, have also been suggested; for example, schemes based on the ADAGARD or on the Nesterov rationale, which have been considered and discussed in the text in Chapter 8.

Beyond The Basic Gradient Descent Scheme

- A number of alternatives have been proposed, and the topic of speeding up the convergence of the backpropagation algorithm has been a hot topic of research, and many variants have been proposed over the years. For example:
  - Newton-type and related simplified versions for computing the associated Hessian matrix.
  - A number of versions, using more recent results on optimization, have also been suggested; for example, schemes based on the ADAGARD or on the Nesterov rationale, which have been considered and discussed in the text in Chapter 8.

Selecting A Cost Function

- The choice of a loss function for the optimization is tightly related with the choice of the output activation function. A wrong combination can severely affect the learning performance of a network.
- A wrong combination: Let us select the squared error loss function and the logistic sigmoid function as the output nonlinearity, i.e.,

$$f(z) = \sigma(z) = \frac{1}{1 + \exp(-az)}, \quad J(e) = \frac{1}{2}(y - \hat{y})^2,$$

where a single output neuron is considered and the time index has been suppressed.

Selecting A Cost Function

- The choice of a loss function for the optimization is tightly related with the choice of the output activation function. A wrong combination can severely affect the learning performance of a network.
- A wrong combination: Let us select the squared error loss function and the logistic sigmoid function as the output nonlinearity, i.e.,

$$f(z) = \sigma(z) = \frac{1}{1 + \exp(-az)}, \quad J(e) = \frac{1}{2}(y - \hat{y})^2,$$

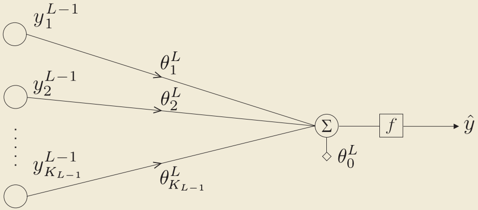
where a single output neuron is considered and the time index has been suppressed.

Selecting A Cost Function

- Assume L layers and let the vector of the parameters, associated with the single output neuron, be $\boldsymbol{\theta}^L$. The vector of the outputs of the previous $(L-1)$ layer is denoted as $\boldsymbol{y}^L := [y_1^{L-1}, y_2^{L-1}, \ldots, y_{k_{L-1}}^{L-1}]$, see figure below. Then, the output of the network will be

$$\hat{y} = \sigma(z^L), \quad z^L := \boldsymbol{\theta}^{L^T} \boldsymbol{y}^{L-1},$$

where the bias term has been included in the vector of parameters.



Selecting A Cost Function

- For the specific combination of loss and activation functions, it turns out that

$$\frac{\partial J}{\partial \boldsymbol{\theta}^L} = (y - \hat{y})\sigma'(z^L)\boldsymbol{y}^{L-1}.$$

- Observe that for values of z^L not close to zero, the derivative of the logistic sigmoid function takes very small values, due to its saturating nature. However, very small values of the gradient lead to considerable slow down of the convergence of the gradient descent type algorithms.
- In contrast, this is not the case, if the squared error loss function is combined with a linear activation function. This is a perfectly good combination (try it).

Selecting A Cost Function

- For the specific combination of loss and activation functions, it turns out that
$$\frac{\partial J}{\partial \boldsymbol{\theta}^L} = (y - \hat{y})\sigma'(z^L)\boldsymbol{y}^{L-1}.$$
- Observe that for values of z^L not close to zero, the derivative of the logistic sigmoid function takes very small values, due to its saturating nature. However, very small values of the gradient lead to considerable slow down of the convergence of the gradient descent type algorithms.
- In contrast, this is not the case, if the squared error loss function is combined with a linear activation function. This is a perfectly good combination (try it).

Selecting A Cost Function

- For the specific combination of loss and activation functions, it turns out that
$$\frac{\partial J}{\partial \boldsymbol{\theta}^L} = (y - \hat{y})\sigma'(z^L)\boldsymbol{y}^{L-1}.$$
- Observe that for values of z^L not close to zero, the derivative of the logistic sigmoid function takes very small values, due to its saturating nature. However, very small values of the gradient lead to considerable slow down of the convergence of the gradient descent type algorithms.
- In contrast, this is not the case, if the squared error loss function is combined with a linear activation function. This is a perfectly good combination (try it).

Cross-Entropy Loss Function

- If one adopts as target values, in a classification task, the $0, 1$ values, i.e., $y_n \in \{0, 1\}$, and assuming k_L output nodes, the cross-entropy cost is defined as

$$J = -\sum_{n=1}^{N}\sum_{k=1}^{k_L}\left(y_{nk}\ln \hat{y}_{nk} + (1-y_{nk})\ln(1-\hat{y}_{nk})\right),$$

where N is the number of the training points.

- The minimum of this cost function is achieved when $y_{nk} = \hat{y}_{nk}$. Viewing $\hat{y}_{nk}$ as the probability of observing an "1" at the respective node, then the probability $P(\boldsymbol{y}_n)$ is equal to

$$P(\boldsymbol{y}_n) = \prod_{k=1}^{k_L}(\hat{y}_{nk})^{y_{nk}}(1-\hat{y}_{nk})^{1-y_{nk}}.$$

Thus, the cross entropy can be interpreted as the negative log-likelihood function over the training samples.

Cross-Entropy Loss Function

- If one adopts as target values, in a classification task, the $0, 1$ values, i.e., $y_n \in \{0, 1\}$, and assuming k_L output nodes, the cross-entropy cost is defined as

$$J = -\sum_{n=1}^{N} \sum_{k=1}^{k_L} \left(y_{nk} \ln \hat{y}_{nk} + (1 - y_{nk}) \ln(1 - \hat{y}_{nk}) \right),$$

  where N is the number of the training points.

- The minimum of this cost function is achieved when $y_{nk} = \hat{y}_{nk}$. Viewing $\hat{y}_{nk}$ as the probability of observing an "1" at the respective node, then the probability $P(\boldsymbol{y}_n)$ is equal to

$$P(\boldsymbol{y}_n) = \prod_{k=1}^{k_L} (\hat{y}_{nk})^{y_{nk}} (1 - \hat{y}_{nk})^{1 - y_{nk}}.$$

  Thus, the cross entropy can be interpreted as the negative log-likelihood function over the training samples.

Cross-Entropy Loss Function

- It turns out that, combining the cross entropy with the logistic sigmoid activation in the output nodes renders the associated gradients independent of the respective derivative and the gradients depend solely on the errors committed.

Softmax Output Activation Function

- Softmax activation function: Although we have interpreted the outputs as probabilities, there is no guarantee that these add to one. This can be enforced if the activation function takes the form

$$\hat{y}_{nk} = \frac{\exp(z_{nk}^L)}{\sum_{m=1}^{k_L} \exp(z_{nm}^L)},$$

which is known as the softmax function.

- It turns out that, combining the softmax activation with the cross-entropy loss makes the gradients equal to

$$\frac{\partial J}{\partial \boldsymbol{\theta}^L} = (y - \hat{y})\boldsymbol{y}^{L-1},$$

where time and node indices have been suppressed. Thus, the gradients depend on the error and no derivative is involved.

Softmax Output Activation Function

- Softmax activation function: Although we have interpreted the outputs as probabilities, there is no guarantee that these add to one. This can be enforced if the activation function takes the form

$$\hat{y}_{nk} = \frac{\exp(z_{nk}^L)}{\sum_{m=1}^{k_L} \exp(z_{nm}^L)},$$

which is known as the softmax function.

- It turns out that, combining the softmax activation with the cross-entropy loss makes the gradients equal to

$$\frac{\partial J}{\partial \boldsymbol{\theta}^L} = (y - \hat{y})\boldsymbol{y}^{L-1},$$

where time and node indices have been suppressed. Thus, the gradients depend on the error and no derivative is involved.

Simulation Examples

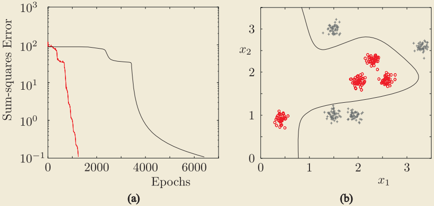
- Learning curves: The classification task consists of two classes, each being the union of four regions. Each region consists of normally distributed random vectors. A total of 400 training vectors were generated, 50 from each distribution. A multilayer perceptron with three neurons in the first and two neurons in the second hidden layer were used, with a single output neuron. The activation function was the logistic one with $a = 1$ and the desired outputs 1 and 0, respectively, for the two classes.
- The momentum and the adaptive momentum algorithms, as explained in the book, were used. The weights were initialized by a uniform pseudorandom distribution between 0 and 1. The obtained results are shown in the following figures.

Simulation Examples

- Learning curves: The classification task consists of two classes, each being the union of four regions. Each region consists of normally distributed random vectors. A total of 400 training vectors were generated, 50 from each distribution. A multilayer perceptron with three neurons in the first and two neurons in the second hidden layer were used, with a single output neuron. The activation function was the logistic one with $a = 1$ and the desired outputs 1 and 0, respectively, for the two classes.
- The momentum and the adaptive momentum algorithms, as explained in the book, were used. The weights were initialized by a uniform pseudorandom distribution between 0 and 1. The obtained results are shown in the following figures.

Simulation Examples

- Learning curves: The classification task consists of two classes, each being the union of four regions. Each region consists of normally distributed random vectors. A total of 400 training vectors were generated, 50 from each distribution. A multilayer perceptron with three neurons in the first and two neurons in the second hidden layer were used, with a single output neuron. The activation function was the logistic one with $a = 1$ and the desired outputs 1 and 0, respectively, for the two classes.
- The momentum and the adaptive momentum algorithms, as explained in the book, were used. The weights were initialized by a uniform pseudorandom distribution between 0 and 1. The obtained results are shown in the following figures.



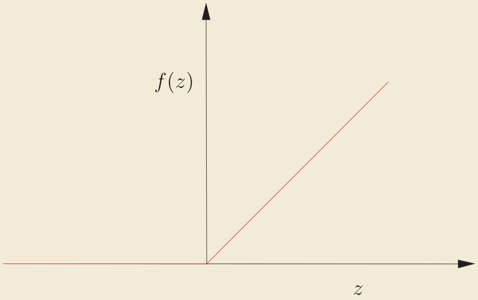
(a) Error convergence curves for the adaptive momentum (red line) and the momentum algorithms. Note that the adaptive momentum leads to faster convergence. (b) The classifier formed by the multilayer perceptron.

The Rectified Linear Unit (ReLU)

- Besides the two already mentioned activation functions, more recently, a new one has become very popular for use in the hidden layers, especially in the context of deep networks. The rectified linear unit (ReLU) is defined as

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

and it is shown in the figure



The Rectified Linear Unit (ReLU)

- It has been established, by now, that the use of the ReLU in the context of deep networks, with many layers, can improve the training time significantly.
- Observe that the ReLU has derivatives that their values remain large for large positive values of z; That is, in the region where the neuron remains active.
- Thus, it is advisable to initialize the respective biases to some positive small value, e.g., $\theta_0 = 0.1$; this increases the probability the the input to the activation has positive values.
- Note that at $z = 0$, the derivative is not defined; yet, in the extreme case that z is exactly zero, one can set the derivative either equal to zero or to one (for those familiar with the notion of subgradient, this makes sense!)

The Rectified Linear Unit (ReLU)

- It has been established, by now, that the use of the ReLU in the context of deep networks, with many layers, can improve the training time significantly.
- Observe that the ReLU has derivatives that their values remain large for large positive values of z; That is, in the region where the neuron remains active.
- Thus, it is advisable to initialize the respective biases to some positive small value, e.g., $\theta_0 = 0.1$; this increases the probability the the input to the activation has positive values.
- Note that at $z = 0$, the derivative is not defined; yet, in the extreme case that z is exactly zero, one can set the derivative either equal to zero or to one (for those familiar with the notion of subgradient, this makes sense!)

The Rectified Linear Unit (ReLU)

- It has been established, by now, that the use of the ReLU in the context of deep networks, with many layers, can improve the training time significantly.
- Observe that the ReLU has derivatives that their values remain large for large positive values of z; That is, in the region where the neuron remains active.
- Thus, it is advisable to initialize the respective biases to some positive small value, e.g., $\theta_0 = 0.1$; this increases the probability the the input to the activation has positive values.
- Note that at $z = 0$, the derivative is not defined; yet, in the extreme case that z is exactly zero, one can set the derivative either equal to zero or to one (for those familiar with the notion of subgradient, this makes sense!)

The Rectified Linear Unit (ReLU)

- It has been established, by now, that the use of the ReLU in the context of deep networks, with many layers, can improve the training time significantly.
- Observe that the ReLU has derivatives that their values remain large for large positive values of z; That is, in the region where the neuron remains active.
- Thus, it is advisable to initialize the respective biases to some positive small value, e.g., $\theta_0 = 0.1$; this increases the probability the the input to the activation has positive values.
- Note that at $z = 0$, the derivative is not defined; yet, in the extreme case that z is exactly zero, one can set the derivative either equal to zero or to one (for those familiar with the notion of subgradient, this makes sense!)

The Rectified Linear Unit (ReLU)

- The major disadvantage of the ReLU is that learning is freezing when $z < 0$. To bypass this obstacle, a number of variants have been proposed.
- Consider the function:

$$f(z) = \max(0, z) + \alpha \min(0, z).$$

  - When $\alpha = -1$, the resulting is known as the absolute value rectification.
  - When α is assigned a fixed small value, e.g., $\alpha = 0.01$, the resulting function is coined as the leaky ReLU.
  - When α is left as a parameter to be learned during the training, it is known as the parametric ReLU.
  - Maxout unit: In this variant, a fixed number of, say k, different ReLUs are employed, which are learned during the training. The output of the neuron is selected as the maximum one among the k ones.

The Rectified Linear Unit (ReLU)

- The major disadvantage of the ReLU is that learning is freezing when $z < 0$. To bypass this obstacle, a number of variants have been proposed.
- Consider the function:

$$f(z) = \max(0, z) + \alpha \min(0, z).$$

  - When $\alpha = -1$, the resulting is known as the absolute value rectification.
  - When α is assigned a fixed small value, e.g., $\alpha = 0.01$, the resulting function is coined as the leaky ReLU.
  - When α is left as a parameter to be learned during the training, it is known as the parametric ReLU.
  - Maxout unit: In this variant, a fixed number of, say k, different ReLUs are employed, which are learned during the training. The output of the neuron is selected as the maximum one among the k ones.

The Rectified Linear Unit (ReLU)

- The major disadvantage of the ReLU is that learning is freezing when $z < 0$. To bypass this obstacle, a number of variants have been proposed.
- Consider the function:

$$f(z) = \max(0, z) + \alpha \min(0, z).$$

  - When $\alpha = -1$, the resulting is known as the absolute value rectification.
  - When α is assigned a fixed small value, e.g., $\alpha = 0.01$, the resulting function is coined as the leaky ReLU.
  - When α is left as a parameter to be learned during the training, it is known as the parametric ReLU.
  - Maxout unit: In this variant, a fixed number of, say k, different ReLUs are employed, which are learned during the training. The output of the neuron is selected as the maximum one among the k ones.

The Rectified Linear Unit (ReLU)

- The major disadvantage of the ReLU is that learning is freezing when $z < 0$. To bypass this obstacle, a number of variants have been proposed.
- Consider the function:

$$f(z) = \max(0, z) + \alpha \min(0, z).$$

  - When $\alpha = -1$, the resulting is known as the absolute value rectification.
  - When α is assigned a fixed small value, e.g., $\alpha = 0.01$, the resulting function is coined as the leaky ReLU.
  - When α is left as a parameter to be learned during the training, it is known as the parametric ReLU.
  - Maxout unit: In this variant, a fixed number of, say k, different ReLUs are employed, which are learned during the training. The output of the neuron is selected as the maximum one among the k ones.

The Rectified Linear Unit (ReLU)

- The major disadvantage of the ReLU is that learning is freezing when $z < 0$. To bypass this obstacle, a number of variants have been proposed.
- Consider the function:

$$f(z) = \max(0, z) + \alpha \min(0, z).$$

  - When $\alpha = -1$, the resulting is known as the absolute value rectification.
  - When α is assigned a fixed small value, e.g., $\alpha = 0.01$, the resulting function is coined as the leaky ReLU.
  - When α is left as a parameter to be learned during the training, it is known as the parametric ReLU.
  - Maxout unit: In this variant, a fixed number of, say k, different ReLUs are employed, which are learned during the training. The output of the neuron is selected as the maximum one among the k ones.

The Rectified Linear Unit (ReLU)

- The major disadvantage of the ReLU is that learning is freezing when $z < 0$. To bypass this obstacle, a number of variants have been proposed.
- Consider the function:

$$f(z) = \max(0, z) + \alpha \min(0, z).$$

  - When $\alpha = -1$, the resulting is known as the absolute value rectification.
  - When α is assigned a fixed small value, e.g., $\alpha = 0.01$, the resulting function is coined as the leaky ReLU.
  - When α is left as a parameter to be learned during the training, it is known as the parametric ReLU.
  - Maxout unit: In this variant, a fixed number of, say k, different ReLUs are employed, which are learned during the training. The output of the neuron is selected as the maximum one among the k ones.

Which Activation Function Then?

- Which nonlinearity is the best? Unfortunately, there is not a universal answer to that. It depends on the data and the problem at hand. At the time of developing these slides, it seems that the ReLU versions are the preferable choice, for a number of mainstream applications.

Pruning a Network

- Pruning a Network: A crucial factor in training NNs is to decide the size of the network. The size is directly related to the number of weights to be estimated and we know that, in any parametric modeling method, if the number of free parameters is **large enough** with respect to the number of training data, **overfitting** is bound to happen.

- In practice, the classical approach is to start with a large enough number of neurons and then use a regularization technique to push the less informative weights to low values, which are then removed. To this end, there exists a number of different regularization approaches, which have been proposed over the years.

Pruning a Network

- Pruning a Network: A crucial factor in training NNs is to decide the size of the network. The size is directly related to the number of weights to be estimated and we know that, in any parametric modeling method, if the number of free parameters is **large** enough with respect to the number of training data, **overfitting** is bound to happen.
- In practice, the classical approach is to start with a large enough number of neurons and then use a regularization technique to push the less informative weights to low values, which are then removed. To this end, there exists a number of different regularization approaches, which have been proposed over the years.

Pruning a Network-Regularization

- Weight decay: This path refers to a typical cost function regularization via the Euclidean norm of the weights. Instead of minimizing a cost function, $J(\boldsymbol{\theta})$, its regularized version is used, i.e.,
$$J^{'}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\|^2.$$
- In general, it is not a good practice to include the bias terms in the norm. As it is the case with the ridge regression task, this affects the translation invariant properties of the network.
- Moreover, it is even better if one groups the parameters of different layers together and employs different regularizing constants for each group.
- More recently, the use of the sparsity promoting ℓ_1 norm has been proposed in places of the Euclidean norm.

Pruning a Network-Regularization

- Weight decay: This path refers to a typical cost function regularization via the Euclidean norm of the weights. Instead of minimizing a cost function, $J(\boldsymbol{\theta})$, its regularized version is used, i.e.,
$$J^{'}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\|^2.$$
- In general, it is not a good practice to include the bias terms in the norm. As it is the case with the ridge regression task, this affects the translation invariant properties of the network.
- Moreover, it is even better if one groups the parameters of different layers together and employs different regularizing constants for each group.
- More recently, the use of the sparsity promoting ℓ_1 norm has been proposed in places of the Euclidean norm.

Pruning a Network-Regularization

- Weight decay: This path refers to a typical cost function regularization via the Euclidean norm of the weights. Instead of minimizing a cost function, $J(\boldsymbol{\theta})$, its regularized version is used, i.e.,

$$J^{'}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\|^2.$$

- In general, it is not a good practice to include the bias terms in the norm. As it is the case with the ridge regression task, this affects the translation invariant properties of the network.
- Moreover, it is even better if one groups the parameters of different layers together and employs different regularizing constants for each group.
- More recently, the use of the sparsity promoting ℓ_1 norm has been proposed in places of the Euclidean norm.

Pruning a Network-Regularization

- Weight decay: This path refers to a typical cost function regularization via the Euclidean norm of the weights. Instead of minimizing a cost function, $J(\boldsymbol{\theta})$, its regularized version is used, i.e.,
$$J^{'}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\|^2.$$
- In general, it is not a good practice to include the bias terms in the norm. As it is the case with the ridge regression task, this affects the translation invariant properties of the network.
- Moreover, it is even better if one groups the parameters of different layers together and employs different regularizing constants for each group.
- More recently, the use of the sparsity promoting ℓ_1 norm has been proposed in places of the Euclidean norm.

Pruning a Network-Regularization

- Weight elimination: Another path is include other functions than norms. An example is,

$$J'(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda \sum_{k=1}^{K} \frac{\theta_k^2}{\theta_h^2 + \theta_k^2}.$$

where K is the number of the weights involved and θ_h is a preselected threshold value.

- A careful look at this function reveals that, if $\theta_k < \theta_h$ the penalty term goes to zero very fast. In contrast, for values $\theta_k > \theta_h$, the penalty term tends to unity. In this way, less significant weights are pushed towards to zero.

Pruning a Network-Regularization

- Weight elimination: Another path is include other functions than norms. An example is,

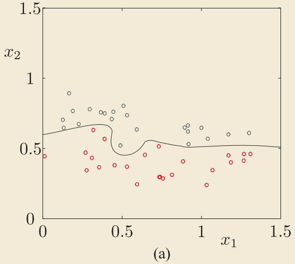
$$J'(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda \sum_{k=1}^{K} \frac{\theta_k^2}{\theta_h^2 + \theta_k^2}.$$

  where K is the number of the weights involved and θ_h is a preselected threshold value.

- A careful look at this function reveals that, if $\theta_k < \theta_h$ the penalty term goes to zero very fast. In contrast, for values $\theta_k > \theta_h$, the penalty term tends to unity. In this way, less significant weights are pushed towards to zero.

Simulation Examples

- Pruning of the network: The samples of the two classes are denoted by black and red "○" respectively. Figure (a) on the left corresponds to a multilayer perceptron with two hidden layers and 20 neurons in each of them, amounting to a total of 480 weights. Training was performed via the backpropagation algorithm. The overfitting nature of the resulting curve is readily observed. Figure (b) on the right corresponds to the same multilayer perceptron trained with a pruning algorithm. Finally, only 25 of the 480 weights have survived, and the curve is simplified to a straight line.



Pruning a Network-Regularization Via Noise Injection

- Adding some small noise to the input data turns out to be equivalent with modifying the cost function by adding an extra term, which acts as a regularizer.
- Adding some small noise to the unknown parameters, during their training, pertubes the solution. Using Taylor series expansion arguments around this perturbation, it turns out that this procedure is equivalent with regularizing the cost function via the norm of the gradient of the w.r. to the parameters.

Pruning a Network-Regularization Via Noise Injection

- Adding some small noise to the input data turns out to be equivalent with modifying the cost function by adding an extra term, which acts as a regularizer.
- Adding some small noise to the unknown parameters, during their training, pertubes the solution. Using Taylor series expansion arguments around this perturbation, it turns out that this procedure is equivalent with regularizing the cost function via the norm of the gradient of the w.r. to the parameters.

Regularization-Artificially Expanding The Data Set

- The source of overfitting is the relatively limited number of the training data compared to the size of the network. Thus, increasing the data size has an equivalent effect as regularization. It decreases overfitting.
- In certain applications, one can artificially generate more data. For example, in an OCR task, one can generate many characters in different rotations. Similar arguments hold for object recognition tasks.
- Soon, we will discuss ways of generating fake data.

Regularization-Artificially Expanding The Data Set

- The source of overfitting is the relatively limited number of the training data compared to the size of the network. Thus, increasing the data size has an equivalent effect as regularization. It decreases overfitting.
- In certain applications, one can artificially generate more data. For example, in an OCR task, one can generate many characters in different rotations. Similar arguments hold for object recognition tasks.
- Soon, we will discuss ways of generating fake data.

Regularization-Artificially Expanding The Data Set

- The source of overfitting is the relatively limited number of the training data compared to the size of the network. Thus, increasing the data size has an equivalent effect as regularization. It decreases overfitting.
- In certain applications, one can artificially generate more data. For example, in an OCR task, one can generate many characters in different rotations. Similar arguments hold for object recognition tasks.
- Soon, we will discuss ways of generating fake data.

Dropout

- This is the most recent technique that deals with overfitting, in the context of deep networks. The term dropout refers to dropping out/removing neurons and/or input nodes in a neural network.
- One starts with a large enough network, comprising, say, K nodes. Choose a training algorithm, e.g., any version of the gradient descent backpropagation algorithm. At each iteration step of the algorithm:
  - Retain each node (hidden or input), together with its incoming and outgoing connections, with probability P.
  - Train the remaining nodes according to the selected algorithm.

Dropout

- This is the most recent technique that deals with overfitting, in the context of deep networks. The term dropout refers to dropping out/removing neurons and/or input nodes in a neural network.
- One starts with a large enough network, comprising, say, K nodes. Choose a training algorithm, e.g., any version of the gradient descent backpropagation algorithm. At each iteration step of the algorithm:
  - Retain each node (hidden or input), together with its incoming and outgoing connections, with probability P.
  - Train the remaining nodes according to the selected algorithm.

Dropout

- This is the most recent technique that deals with overfitting, in the context of deep networks. The term dropout refers to dropping out/removing neurons and/or input nodes in a neural network.
- One starts with a large enough network, comprising, say, K nodes. Choose a training algorithm, e.g., any version of the gradient descent backpropagation algorithm. At each iteration step of the algorithm:
  - Retain each node (hidden or input), together with its incoming and outgoing connections, with probability P.
  - Train the remaining nodes according to the selected algorithm.

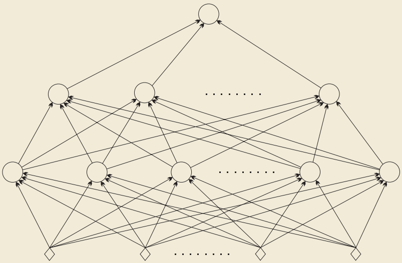
Dropout

- This is the most recent technique that deals with overfitting, in the context of deep networks. The term dropout refers to dropping out/removing neurons and/or input nodes in a neural network.
- One starts with a large enough network, comprising, say, K nodes. Choose a training algorithm, e.g., any version of the gradient descent backpropagation algorithm. At each iteration step of the algorithm:
  - Retain each node (hidden or input), together with its incoming and outgoing connections, with probability P.
  - Train the remaining nodes according to the selected algorithm.

Dropout

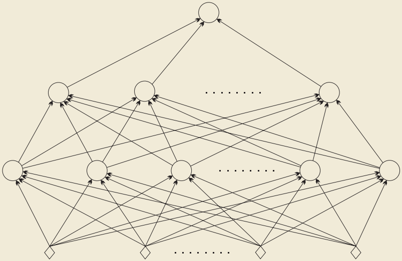
- Thus, at each iteration step, a different subnetwork is trained. In other words, at each iteration step, only the parameters of one subnetwork are updated. The parameters of the "removed" nodes are left unchanged, frozen at their current estimates.

Full Network And Subnetwork For The Dropout Method



Full network

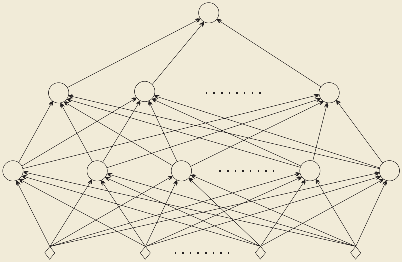
Full Network And Subnetwork For The Dropout Method



Full network

Red nodes and connections to be removed

Full Network And Subnetwork For The Dropout Method



Full network   Red nodes removed

Dropout

- Once training has been completed and convergence has been achieved, during the test phase, each parameter is multiplied by the probability P.
- Justification: At each iteration step, a different subnetwork is trained. This can be thought of as being equivalent of training a large number of networks (theoretically 2^K). Once training is over, one combines the trained subnetworks by an averaging rationale.
- This reminds of the bagging approach to combine predictors. Yet, it is different. In dropout, there is a large overlap and parameter sharing among the different subnetworks.
- Typical values for the probabilities are: For hidden layers $P = 0.5$ and for input units $P = 0.8$.

Dropout

- Once training has been completed and convergence has been achieved, during the test phase, each parameter is multiplied by the probability P.
- Justification: At each iteration step, a different subnetwork is trained. This can be thought of as being equivalent of training a large number of networks (theoretically 2^K). Once training is over, one combines the trained subnetworks by an averaging rationale.
- This reminds of the bagging approach to combine predictors. Yet, it is different. In dropout, there is a large overlap and parameter sharing among the different subnetworks.
- Typical values for the probabilities are: For hidden layers $P = 0.5$ and for input units $P = 0.8$.

Dropout

- Once training has been completed and convergence has been achieved, during the test phase, each parameter is multiplied by the probability P.
- Justification: At each iteration step, a different subnetwork is trained. This can be thought of as being equivalent of training a large number of networks (theoretically 2^K). Once training is over, one combines the trained subnetworks by an averaging rationale.
- This reminds of the bagging approach to combine predictors. Yet, it is different. In dropout, there is a large overlap and parameter sharing among the different subnetworks.
- Typical values for the probabilities are: For hidden layers $P = 0.5$ and for input units $P = 0.8$.

Dropout

- Once training has been completed and convergence has been achieved, during the test phase, each parameter is multiplied by the probability P.
- Justification: At each iteration step, a different subnetwork is trained. This can be thought of as being equivalent of training a large number of networks (theoretically 2^K). Once training is over, one combines the trained subnetworks by an averaging rationale.
- This reminds of the bagging approach to combine predictors. Yet, it is different. In dropout, there is a large overlap and parameter sharing among the different subnetworks.
- Typical values for the probabilities are: For hidden layers $P = 0.5$ and for input units $P = 0.8$.

Dropout

- Is this magic? A heuristic explanation on why this technique works is that it reduces co-adaptations of neurons, since at each iteration different neurons are updated. Thus, the network is forced to learn more robust features, that are useful in conjunction with the different subnetworks.
- In other words, the network is forced to learn while parts of the network are missing at random.
- Some more theoretically pleasing arguments have been given via a Bayesian inference view of a neural network.

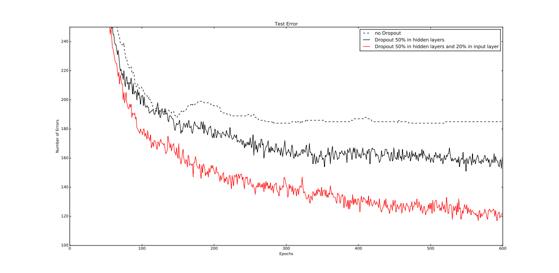
Dropout

- Is this magic? A heuristic explanation on why this technique works is that it reduces co-adaptations of neurons, since at each iteration different neurons are updated. Thus, the network is forced to learn more robust features, that are useful in conjunction with the different subnetworks.
- In other words, the network is forced to learn while parts of the network are missing at random.
- Some more theoretically pleasing arguments have been given via a Bayesian inference view of a neural network.

Dropout

- Is this magic? A heuristic explanation on why this technique works is that it reduces co-adaptations of neurons, since at each iteration different neurons are updated. Thus, the network is forced to learn more robust features, that are useful in conjunction with the different subnetworks.
- In other words, the network is forced to learn while parts of the network are missing at random.
- Some more theoretically pleasing arguments have been given via a Bayesian inference view of a neural network.

- Data Base used MNIST with 55000 handwritten digits for the training and 10000 handwritten digits for testing.
- A feedforward NN was used with 784-2000-2000-10 neurons.
- ReLU units were used for the hidden layers and ten softmax output units were employed.
- Cross entropy was used as a cost function, the mini batch size was 100 and the learning rate for the gradient backpropagation algorithm was set equal to 0.01.



Universal Approximation Property of Feed-Forward Neural Networks

- So far, we focused on how to train neural networks so that to learn a specific input-output mapping. Our interest now turns on **what** a neural network is capable to learn?
- To this end, some strong theoretical results have been developed and are still being developed.
- Let us consider a two-layer network, with one hidden layer and with a single output linear node. The output of the network is then written as

$$\hat{g}(\boldsymbol{x}) = \sum_{k=1}^{K} \theta_k^o f(\boldsymbol{\theta}_k^{hT} \boldsymbol{x}) + \theta_0^o,$$

where $\boldsymbol{\theta}_k^h$ denotes the synaptic weights and bias term defining the kth hidden neuron and the superscript "o" refers to the output neuron. Then, the following theorem holds true.

Universal Approximation Property of Feed-Forward Neural Networks

- So far, we focused on how to train neural networks so that to learn a specific input-output mapping. Our interest now turns on **what** a neural network is capable to learn?
- To this end, some strong theoretical results have been developed and are still being developed.
- Let us consider a two-layer network, with one hidden layer and with a single output linear node. The output of the network is then written as

$$\hat{g}(\boldsymbol{x}) = \sum_{k=1}^{K} \theta_k^o f(\boldsymbol{\theta}_k^{hT} \boldsymbol{x}) + \theta_0^o,$$

where $\boldsymbol{\theta}_k^h$ denotes the synaptic weights and bias term defining the kth hidden neuron and the superscript "o" refers to the output neuron. Then, the following theorem holds true.

Universal Approximation Property of Feed-Forward Neural Networks

- So far, we focused on how to train neural networks so that to learn a specific input-output mapping. Our interest now turns on **what** a neural network is capable to learn?
- To this end, some strong theoretical results have been developed and are still being developed.
- Let us consider a two-layer network, with one hidden layer and with a single output linear node. The output of the network is then written as

$$\hat{g}(\boldsymbol{x}) = \sum_{k=1}^{K} \theta_k^o f(\boldsymbol{\theta}_k^{hT} \boldsymbol{x}) + \theta_0^o,$$

where $\boldsymbol{\theta}_k^h$ denotes the synaptic weights and bias term defining the kth hidden neuron and the superscript "o" refers to the output neuron. Then, the following theorem holds true.

Universal Approximation Property of Feed-Forward Neural Networks

- So far, we focused on how to train neural networks so that to learn a specific input-output mapping. Our interest now turns on **what** a neural network is capable to learn?
- To this end, some strong theoretical results have been developed and are still being developed.
- Let us consider a two-layer network, with one hidden layer and with a single output linear node. The output of the network is then written as

$$\hat{g}(\boldsymbol{x}) = \sum_{k=1}^{K} \theta_k^o f(\boldsymbol{\theta}_k^{hT} \boldsymbol{x}) + \theta_0^o,$$

  where $\boldsymbol{\theta}_k^h$ denotes the synaptic weights and bias term defining the kth hidden neuron and the superscript "o" refers to the output neuron. Then, the following theorem holds true.

Universal Approximation Property of Feed-Forward Neural Networks

- Theorem: Let $g(\boldsymbol{x})$ be a continuous function defined in a compact (closed and bounded) subset $S \subset \mathbb{R}^l$ and any $\epsilon > 0$. Then there exists a $K(\epsilon)$ and a two-layer network of the previous form, so that

  $$|g(\boldsymbol{x}) - \hat{g}(\boldsymbol{x})| < \epsilon, \ \forall \boldsymbol{x} \in S.$$

  It has been shown that the approximation error decreases according to an $O(\frac{1}{K})$ rule.

- In other words, the input dimensionality does **not** enter into the scene and the error depends on the number of neurons used. The theorem states that a two-layer NN network is sufficient to approximate **any** continuous function.

Universal Approximation Property of Feed-Forward Neural Networks

- Theorem: Let $g(\boldsymbol{x})$ be a continuous function defined in a compact (closed and bounded) subset $S \subset \mathbb{R}^l$ and any $\epsilon > 0$. Then there exists a $K(\epsilon)$ and a two-layer network of the previous form, so that

  $$|g(\boldsymbol{x}) - \hat{g}(\boldsymbol{x})| < \epsilon, \ \forall \boldsymbol{x} \in S.$$

  It has been shown that the approximation error decreases according to an $O(\frac{1}{K})$ rule.

- In other words, the input dimensionality does **not** enter into the scene and the error depends on the number of neurons used. The theorem states that a two-layer NN network is sufficient to approximate **any** continuous function.

Universal Approximation Property of Feed-Forward Neural Networks

- However, what the theorem does not say is how big such a network should be, in terms of the required number of neurons in the single layer. It may be that a very large number of neurons is needed in order to obtain a good enough approximation.
- This is where the use of more layers can be advantageous. Using more layers, the overall number of neurons, needed to achieve certain approximation, may be much smaller.
- This is point that ignites our the interest for deep networks, involving many hidden layers.

Universal Approximation Property of Feed-Forward Neural Networks

- However, what the theorem does not say is how big such a network should be, in terms of the required number of neurons in the single layer. It may be that a very large number of neurons is needed in order to obtain a good enough approximation.
- This is where the use of more layers can be advantageous. Using more layers, the overall number of neurons, needed to achieve certain approximation, may be much smaller.
- This is point that ignites our the interest for deep networks, involving many hidden layers.

Universal Approximation Property of Feed-Forward Neural Networks

- However, what the theorem does not say is how big such a network should be, in terms of the required number of neurons in the single layer. It may be that a very large number of neurons is needed in order to obtain a good enough approximation.
- This is where the use of more layers can be advantageous. Using more layers, the overall number of neurons, needed to achieve certain approximation, may be much smaller.
- This is point that ignites our the interest for deep networks, involving many hidden layers.

The Need for Deep Architectures

- We have already discussed that each layer of a neural network provides a different description of the input patterns. In the context of our previous presentation:
  - The input layer described each pattern as a point in the feature space.
  - The first hidden layer of nodes formed a partition of the input space and placed each input point in one of the regions, using a coding scheme of zeros and ones at the outputs of the respective neurons. This can be considered as a more abstract **representation** of our input patterns.
  - The second hidden layer of nodes, based on the information provided by the previous layer, encoded information related to the classes; this is a further representation abstraction, which carries some type of **semantic meaning**. For example, it could provide information of whether a tumor is malignant or benign, in a related a medical application.

The Need for Deep Architectures

- We have already discussed that each layer of a neural network provides a different description of the input patterns. In the context of our previous presentation:
  - The input layer described each pattern as a point in the feature space.
  - The first hidden layer of nodes formed a partition of the input space and placed each input point in one of the regions, using a coding scheme of zeros and ones at the outputs of the respective neurons. This can be considered as a more abstract **representation** of our input patterns.
  - The second hidden layer of nodes, based on the information provided by the previous layer, encoded information related to the classes; this is a further representation abstraction, which carries some type of **semantic meaning**. For example, it could provide information of whether a tumor is malignant or benign, in a related a medical application.

The Need for Deep Architectures

- We have already discussed that each layer of a neural network provides a different description of the input patterns. In the context of our previous presentation:
  - The input layer described each pattern as a point in the feature space.
  - The first hidden layer of nodes formed a partition of the input space and placed each input point in one of the regions, using a coding scheme of zeros and ones at the outputs of the respective neurons. This can be considered as a more abstract **representation** of our input patterns.
  - The second hidden layer of nodes, based on the information provided by the previous layer, encoded information related to the classes; this is a further representation abstraction, which carries some type of **semantic meaning**. For example, it could provide information of whether a tumor is malignant or benign, in a related a medical application.

The Need for Deep Architectures

- We have already discussed that each layer of a neural network provides a different description of the input patterns. In the context of our previous presentation:
  - The input layer described each pattern as a point in the feature space.
  - The first hidden layer of nodes formed a partition of the input space and placed each input point in one of the regions, using a coding scheme of zeros and ones at the outputs of the respective neurons. This can be considered as a more abstract **representation** of our input patterns.
  - The second hidden layer of nodes, based on the information provided by the previous layer, encoded information related to the classes; this is a further representation abstraction, which carries some type of **semantic meaning**. For example, it could provide information of whether a tumor is malignant or benign, in a related a medical application.

The Need for Deep Architectures

- It turns out that, the previous reported hierarchical type of representations of the input patterns mimics the way that a mammal's brain follows in order to understand and sense the world around us; in the case of humans, this is the physical mechanism in the brain, which intelligence is built upon.
- The brain of the mammals is organized in a number of layers of neurons and each layer provides a different representation of the input percept. In this way, different levels of abstraction are formed, via a **hierarchy** of transformations.

The Need for Deep Architectures

- It turns out that, the previous reported hierarchical type of representations of the input patterns mimics the way that a mammal's brain follows in order to understand and sense the world around us; in the case of humans, this is the physical mechanism in the brain, which intelligence is built upon.
- The brain of the mammals is organized in a number of layers of neurons and each layer provides a different representation of the input percept. In this way, different levels of abstraction are formed, via a **hierarchy** of transformations.

The Need for Deep Architectures

- For example, in the primate visual system, this hierarchy involves first detection of edges, then formation of primitive shapes and every subsequent stage forms more complex visual shapes, till finally a **semantics concept** is formed; e.g., a car moving in a video scene, a person sitting in an image. The cortex of our brain can be seen as a multilayer architecture with 5-10 layers dedicated only to our visual system.
- An issue that is now raised is whether one can obtain an equivalent input-output representation via a relatively simple functional formulation, e.g., via networks with less than three layers of neurons/processing elements, maybe at the expense of more elements per layer.

The Need for Deep Architectures

- For example, in the primate visual system, this hierarchy involves first detection of edges, then formation of primitive shapes and every subsequent stage forms more complex visual shapes, till finally a **semantics concept** is formed; e.g., a car moving in a video scene, a person sitting in an image. The cortex of our brain can be seen as a multilayer architecture with 5-10 layers dedicated only to our visual system.
- An issue that is now raised is whether one can obtain an equivalent input-output representation via a relatively simple functional formulation, e.g., via networks with less than three layers of neurons/processing elements, maybe at the expense of more elements per layer.

The Need for Deep Architectures

- The answer to the first of the previously stated two point is yes, as long as the input-output dependence relation is simple enough. However, for more complex tasks, where more complex concepts have to be learned, e.g., recognition of a scene in a video recording, language and speech recognition, the underlying functional dependence is of very a complex nature that we are unable to express it analytically in a simple way.

The Need for Deep Architectures

- The answer to the second point, concerning networks, lies in what is known as compactness of representation. We say that a network, realizing an input-output functional dependence, is compact if it consists of relatively few free parameters (few computational elements) to be learned/tuned during the training phase. Thus, for a given number of training points, we expect compact representations to result in better generalization performance.

The Need for Deep Architectures

- Using networks with more layers can lead to more compact representations of the input-output relation. Results from the theory of circuits of Boolean functions suggest that a function, which can compactly be realized by, say, k layers of logic elements, may need an exponentially large number of elements if it is realized via $k-1$ layers.
- Some of these results have been generalized and are valid for learning algorithms in some special cases. For example, it has been shown that, for a class of deep networks and target functions, one needs a substantially smaller number of nodes to achieve a predefined accuracy compared to a shallow one.

The Need for Deep Architectures

- Using networks with more layers can lead to more compact representations of the input-output relation. Results from the theory of circuits of Boolean functions suggest that a function, which can compactly be realized by, say, k layers of logic elements, may need an exponentially large number of elements if it is realized via $k-1$ layers.
- Some of these results have been generalized and are valid for learning algorithms in some special cases. For example, it has been shown that, for a class of deep networks and target functions, one needs a substantially smaller number of nodes to achieve a predefined accuracy compared to a shallow one.

Learning Deep Networks

- A major drawback of multilayer NNs is that their training can become difficult. This drawback becomes more severe if more than two hidden layers are used. The more layers one uses, the more difficult the training becomes. Historically, in the 1990's, the effort to train large networks was, practically, abandoned.

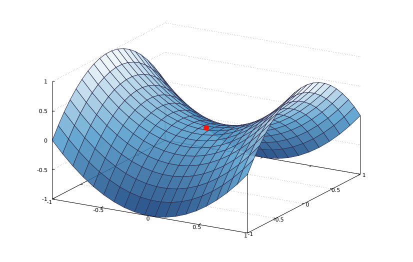
- For a long time, it was believed that, this was due to the existence of many local minima, which caused the learning algorithm to be trapped in a shallow one. To remedy such a drawback, the algorithm was randomly initialized from different points a number of times, hoping for the best result.

Learning Deep Networks

- A major drawback of multilayer NNs is that their training can become difficult. This drawback becomes more severe if more than two hidden layers are used. The more layers one uses, the more difficult the training becomes. Historically, in the 1990's, the effort to train large networks was, practically, abandoned.
- For a long time, it was believed that, this was due to the existence of many local minima, which caused the learning algorithm to be trapped in a shallow one. To remedy such a drawback, the algorithm was randomly initialized from different points a number of times, hoping for the best result.

Learning Deep Networks

- The view point concerning local minima is now challenged, as new results started coming out around 2015. Theoretical as well as experimental evidence point out that the major drawback lies not in the local minima but in the saddle points. At the time these slides are being developed, this is an ongoing and active research area.



Learning Deep Networks

- Under some simplifications, it has been shown that in large size networks most local minima yield low cost function values and result to similar performance. Moreover, the probability of finding a poor local minimum decreases fast as the size of the network increases ([Choromanska, et.al. 2015]).
- In high dimensional spaces, the major drawback seems to be posed by the proliferation of the saddle points. The existence of such points can slow down the convergence of the training algorithms dramatically (Dauphin, et.al, 2014]).

Learning Deep Networks

- Under some simplifications, it has been shown that in large size networks most local minima yield low cost function values and result to similar performance. Moreover, the probability of finding a poor local minimum decreases fast as the size of the network increases ([Choromanska, et.al. 2015]).
- In high dimensional spaces, the major drawback seems to be posed by the proliferation of the saddle points. The existence of such points can slow down the convergence of the training algorithms dramatically (Dauphin, et.al, 2014]).

Learning Deep Networks

- Although the exact effect of these findings on the gradient-type algorithms is not yet clear, it seems that they are finally able to escape such critical points, in spite of the very small values of the corresponding gradients ([Goodfellow, et. al. 2015]).
- A particularly interesting result has been derived in [Xie, et. al., 2017]. Focusing on a single hidden layer network, involving ReLU activations, they proved that, in spite of the nonconvexity of the cost function:
  - Under certain assumptions, there are no spurious local minima points.
  - If the squared norm of the gradient matrix is bounded by an ϵ, then the (squared) error on the training set is also bounded by $O(\epsilon)$.
  - The **generalization error** is bounded by $O(\epsilon + \frac{1}{\sqrt{N}})$, where N is the number of training data points.

Learning Deep Networks

- Although the exact effect of these findings on the gradient-type algorithms is not yet clear, it seems that they are finally able to escape such critical points, in spite of the very small values of the corresponding gradients ([Goodfellow, et. al. 2015]).
- A particularly interesting result has been derived in [Xie, et. al., 2017]. Focusing on a single hidden layer network, involving ReLU activations, they proved that, in spite of the nonconvexity of the cost function:
  - Under certain assumptions, there are no spurious local minima points.
  - If the squared norm of the gradient matrix is bounded by an ϵ, then the (squared) error on the training set is also bounded by $O(\epsilon)$.
  - The **generalization error** is bounded by $O(\epsilon + \frac{1}{\sqrt{N}})$, where N is the number of training data points.

Learning Deep Networks

- Although the exact effect of these findings on the gradient-type algorithms is not yet clear, it seems that they are finally able to escape such critical points, in spite of the very small values of the corresponding gradients ([Goodfellow, et. al. 2015]).
- A particularly interesting result has been derived in [Xie, et. al., 2017]. Focusing on a single hidden layer network, involving ReLU activations, they proved that, in spite of the nonconvexity of the cost function:
  - Under certain assumptions, there are no spurious local minima points.
  - If the squared norm of the gradient matrix is bounded by an ϵ, then the (squared) error on the training set is also bounded by $O(\epsilon)$.
  - The **generalization error** is bounded by $O(\epsilon + \frac{1}{\sqrt{N}})$, where N is the number of training data points.

Learning Deep Networks

- Although the exact effect of these findings on the gradient-type algorithms is not yet clear, it seems that they are finally able to escape such critical points, in spite of the very small values of the corresponding gradients ([Goodfellow, et. al. 2015]).
- A particularly interesting result has been derived in [Xie, et. al., 2017]. Focusing on a single hidden layer network, involving ReLU activations, they proved that, in spite of the nonconvexity of the cost function:
  - Under certain assumptions, there are no spurious local minima points.
  - If the squared norm of the gradient matrix is bounded by an ϵ, then the (squared) error on the training set is also bounded by $O(\epsilon)$.
  - The **generalization error** is bounded by $O(\epsilon + \frac{1}{\sqrt{N}})$, where N is the number of training data points.

Learning Deep Networks

- Although the exact effect of these findings on the gradient-type algorithms is not yet clear, it seems that they are finally able to escape such critical points, in spite of the very small values of the corresponding gradients ([Goodfellow, et. al. 2015]).
- A particularly interesting result has been derived in [Xie, et. al., 2017]. Focusing on a single hidden layer network, involving ReLU activations, they proved that, in spite of the nonconvexity of the cost function:
  - Under certain assumptions, there are no spurious local minima points.
  - If the squared norm of the gradient matrix is bounded by an ϵ, then the (squared) error on the training set is also bounded by $O(\epsilon)$.
  - The **generalization error** is bounded by $O(\epsilon + \frac{1}{\sqrt{N}})$, where N is the number of training data points.

Learning Deep Networks

- The previous result confirms what is known and observed in practice. Neural networks can perform well even without regularization, provided that they are trained with lots of training points. Of course, regularization improves the performance.
- Currently, the success of the neural networks seems to lie in the available computational power combined with the availability of large training data sets. The combination of ReLU activation functions with the dropout technique, together with some practical hints, concerning initialization, seem to offer the secret of their success.
- The use of appropriate pre-training techniques, as we will soon see, can also be beneficial in certain cases.

Learning Deep Networks

- The previous result confirms what is known and observed in practice. Neural networks can perform well even without regularization, provided that they are trained with lots of training points. Of course, regularization improves the performance.
- Currently, the success of the neural networks seems to lie in the available computational power combined with the availability of large training data sets. The combination of ReLU activation functions with the dropout technique, together with some practical hints, concerning initialization, seem to offer the secret of their success.
- The use of appropriate pre-training techniques, as we will soon see, can also be beneficial in certain cases.

Learning Deep Networks

- The previous result confirms what is known and observed in practice. Neural networks can perform well even without regularization, provided that they are trained with lots of training points. Of course, regularization improves the performance.
- Currently, the success of the neural networks seems to lie in the available computational power combined with the availability of large training data sets. The combination of ReLU activation functions with the dropout technique, together with some practical hints, concerning initialization, seem to offer the secret of their success.
- The use of appropriate pre-training techniques, as we will soon see, can also be beneficial in certain cases.

Features Via Convolutions

- A major class of NNs, known as convolutional neural networks, employ convolutions instead of multiply-add type of neurons. We will first review the "why" behind the convolutions.
- The input to any classifier/learner is presented with a set of features. Each input vector, x_n, in the training set is a point in the feature space. The features should encode, in a compact way, information that resides in the raw/sensed data and it is related to the learning task at hand.
- If, instead, the input to a neural network were the pixels of a 256×256 image, this would correspond to a vector in a space of dimension equal to 65536. If the first hidden layer had, say, 20000 nodes, this would amount to approximately 1.3 billion synapses! Adding more layers, this number would explode further.

Features Via Convolutions

- A major class of NNs, known as convolutional neural networks, employ convolutions instead of multiply-add type of neurons. We will first review the "why" behind the convolutions.
- The input to any classifier/learner is presented with a set of features. Each input vector, $\boldsymbol{x}_n$, in the training set is a point in the feature space. The features should encode, in a compact way, information that resides in the raw/sensed data and it is related to the learning task at hand.
- If, instead, the input to a neural network were the pixels of a 256×256 image, this would correspond to a vector in a space of dimension equal to 65536. If the first hidden layer had, say, 20000 nodes, this would amount to approximately 1.3 billion synapses! Adding more layers, this number would explode further.

Features Via Convolutions

- A major class of NNs, known as convolutional neural networks, employ convolutions instead of multiply-add type of neurons. We will first review the "why" behind the convolutions.
- The input to any classifier/learner is presented with a set of features. Each input vector, $\boldsymbol{x}_n$, in the training set is a point in the feature space. The features should encode, in a compact way, information that resides in the raw/sensed data and it is related to the learning task at hand.
- If, instead, the input to a neural network were the pixels of a 256×256 image, this would correspond to a vector in a space of dimension equal to 65536. If the first hidden layer had, say, 20000 nodes, this would amount to approximately 1.3 billion synapses! Adding more layers, this number would explode further.

Features Via Convolutions

- One among the most popular ways to generate features from an image (and not only) has traditionally being to “run” a filter over the image. Filtering exploits underlying correlations/relations among the pixels; different filters can extract different type of information.
- Example: Edge detection

Features Via Convolutions

- One among the most popular ways to generate features from an image (and not only) has traditionally being to "run" a filter over the image. Filtering exploits underlying correlations/relations among the pixels; different filters can extract different type of information.
- Example: Edge detection

Features Via Convolutions

- One among the most popular ways to generate features from an image (and not only) has traditionally being to "run" a filter over the image. Filtering exploits underlying correlations/relations among the pixels; different filters can extract different type of information.
- Example: Edge detection

$$H = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Features Via Convolutions

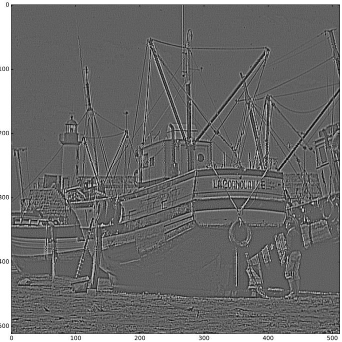
- One among the most popular ways to generate features from an image (and not only) has traditionally being to "run" a filter over the image. Filtering exploits underlying correlations/relations among the pixels; different filters can extract different type of information.
- Example: Edge detection

$$H = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Original image

Features Via Convolutions

- Example: Edge detection



Features Via Convolutions

- Convolution: In the current context, filtering will be viewed as a cross-correlation operation between the filer matrix, known as the kernel matrix, H, and the image array, I. The output matrix, O, is known as the feature map.
- Let us assume, for simplicity, that,

$$H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \text{ and } I = \begin{bmatrix} I(1,1) & I(1,2) & I(1,3) \\ I(2,1) & I(2,2) & I(2,3) \\ I(3,1) & I(3,2) & I(3,3) \end{bmatrix}.$$

- The convolution between the kernel matrix, H, and the image, I, will be the 2×2 feature map array, O, with elements

$$O(n,m) = \sum_{i=1}^{2} \sum_{j=1}^{2} h_{ij} I(n+i-1, m+j-1), \; n,m = 1,2.$$

Features Via Convolutions

- Convolution: In the current context, filtering will be viewed as a cross-correlation operation between the filer matrix, known as the kernel matrix, H, and the image array, I. The output matrix, O, is known as the the feature map.
- Let us assume, for simplicity, that,

$$H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \text{ and } I = \begin{bmatrix} I(1,1) & I(1,2) & I(1,3) \\ I(2,1) & I(2,2) & I(2,3) \\ I(3,1) & I(3,2) & I(3,3) \end{bmatrix}.$$

- The convolution between the kernel matrix, H, and the image, I, will be the 2×2 feature map array, O, with elements

$$O(n,m) = \sum_{i=1}^{2} \sum_{j=1}^{2} h_{ij} I(n+i-1, m+j-1), \; n,m = 1,2.$$

Features Via Convolutions

- Convolution: In the current context, filtering will be viewed as a cross-correlation operation between the filer matrix, known as the kernel matrix, H, and the image array, I. The output matrix, O, is known as the feature map.
- Let us assume, for simplicity, that,

$$H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \text{ and } I = \begin{bmatrix} I(1,1) & I(1,2) & I(1,3) \\ I(2,1) & I(2,2) & I(2,3) \\ I(3,1) & I(3,2) & I(3,3) \end{bmatrix}.$$

- The convolution between the kernel matrix, H, and the image, I, will be the 2×2 feature map array, O, with elements

$$O(n,m) = \sum_{i=1}^{2} \sum_{j=1}^{2} h_{ij} I(n+i-1, m+j-1), \; n,m = 1,2.$$

Features Via Convolutions

$$\begin{bmatrix} h_{11} \cdot I(1,1) & h_{12} \cdot I(1,2) & I(1,3) \\ h_{21} \cdot I(2,1) & h_{22} \cdot I(2,2) & I(2,3) \\ I(3,1) & I(3,2) & I(3,3) \end{bmatrix}, \quad O = \begin{bmatrix} O(1,1) & * \\ * & * \end{bmatrix}$$

Features Via Convolutions

$$\begin{bmatrix} I(1,1) & h_{11} \cdot I(1,2) & h_{12} \cdot I(1,3) \\ I(2,1) & h_{21} \cdot I(2,2) & h_{22} \cdot I(2,3) \\ I(3,1) & I(3,2) & I(3,3) \end{bmatrix}, \; O = \begin{bmatrix} O(1,1) & O(1,2) \\ * & * \end{bmatrix}$$

Features Via Convolutions

$$\begin{bmatrix} I(1,1) & I(1,2) & I(1,3) \\ h_{11} \cdot I(2,1) & h_{22} \cdot I(2,2) & I(2,3) \\ h_{21} \cdot I(3,1) & h_{22} \cdot I(3,2) & I(3,3) \end{bmatrix}, \ O = \begin{bmatrix} O(1,1) & O(1,2) \\ O(2,1) & * \end{bmatrix}$$

Features Via Convolutions

$$\begin{bmatrix} I(1,1) & I(1,2) & I(1,3) \\ I(2,1) & h_{11} \cdot I(2,2) & h_{21} \cdot I(2,3) \\ I(3,1) & h_{12} \cdot I(3,2) & h_{22} \cdot I(3,3) \end{bmatrix}, \quad O = \begin{bmatrix} O(1,1) & O(1,2) \\ O(2,1) & O(2,2) \end{bmatrix}$$

Convolutional Neural Networks (CNN)

- A breakthrough in training neural networks came in the late 1980s (Le Cun), when the feature generation step, via convolutions, was integrated as part of a neural network.
- Instead of using fixed kernel matrices, it was left to the network to learn the elements of the kernel matrix as part of the training process.
- Thus, the first layers of a neural network were dedicated to perform convolutions instead of simple multiply-add operations. These constitute the layers where the features are **learned** from the input raw data and feed subsequent layers of the NN.
- The basic steps performed in the front end convolution layers are:
  - The convolution step.
  - The nonlinearity step.
  - The pooling step.

Convolutional Neural Networks (CNN)

- A breakthrough in training neural networks came in the late 1980s (Le Cun), when the feature generation step, via convolutions, was integrated as part of a neural network.
- Instead of using fixed kernel matrices, it was left to the network to learn the elements of the kernel matrix as part of the training process.
- Thus, the first layers of a neural network were dedicated to perform convolutions instead of simple multiply-add operations. These constitute the layers where the features are **learned** from the input raw data and feed subsequent layers of the NN.
- The basic steps performed in the front end convolution layers are:
  - The convolution step.
  - The nonlinearity step.
  - The pooling step.

Convolutional Neural Networks (CNN)

- A breakthrough in training neural networks came in the late 1980s (Le Cun), when the feature generation step, via convolutions, was integrated as part of a neural network.
- Instead of using fixed kernel matrices, it was left to the network to learn the elements of the kernel matrix as part of the training process.
- Thus, the first layers of a neural network were dedicated to perform convolutions instead of simple multiply-add operations. These constitute the layers where the features are **learned** from the input raw data and feed subsequent layers of the NN.
- The basic steps performed in the front end convolution layers are:
  - The convolution step.
  - The nonlinearity step.
  - The pooling step.

Convolutional Neural Networks (CNN)

- A breakthrough in training neural networks came in the late 1980s (Le Cun), when the feature generation step, via convolutions, was integrated as part of a neural network.
- Instead of using fixed kernel matrices, it was left to the network to learn the elements of the kernel matrix as part of the training process.
- Thus, the first layers of a neural network were dedicated to perform convolutions instead of simple multiply-add operations. These constitute the layers where the features are **learned** from the input raw data and feed subsequent layers of the NN.
- The basic steps performed in the front end convolution layers are:
  - The convolution step.
  - The nonlinearity step.
  - The pooling step.

Convolutional Neural Networks (CNN)

- A breakthrough in training neural networks came in the late 1980s (Le Cun), when the feature generation step, via convolutions, was integrated as part of a neural network.
- Instead of using fixed kernel matrices, it was left to the network to learn the elements of the kernel matrix as part of the training process.
- Thus, the first layers of a neural network were dedicated to perform convolutions instead of simple multiply-add operations. These constitute the layers where the features are **learned** from the input raw data and feed subsequent layers of the NN.
- The basic steps performed in the front end convolution layers are:
  - The convolution step.
  - The nonlinearity step.
  - The pooling step.

Convolutional Neural Networks (CNN)

- A breakthrough in training neural networks came in the late 1980s (Le Cun), when the feature generation step, via convolutions, was integrated as part of a neural network.
- Instead of using fixed kernel matrices, it was left to the network to learn the elements of the kernel matrix as part of the training process.
- Thus, the first layers of a neural network were dedicated to perform convolutions instead of simple multiply-add operations. These constitute the layers where the features are **learned** from the input raw data and feed subsequent layers of the NN.
- The basic steps performed in the front end convolution layers are:
  - The convolution step.
  - The nonlinearity step.
  - The pooling step.

CNN: The Convolution Step

- The first layer in a CNN comprises the parameters of the kernel matrix. If the input nodes correspond to the (raw) pixels of an image array, the output of the convolutional layer is the corresponding feature map array.
- Thus, the parameters comprise the kernel array and they are **shared** among the input pixels; moreover, in place of the multiply-add operations convolutions are performed, instead.
- However, instead of a single kernel matrix, multiple ones are used; each one is expected to extract **different** type of information, to be encoded via a **different** feature map array. In the figure, three such kernel arrays are shown to "scan" the input image, searching for "hidden information".

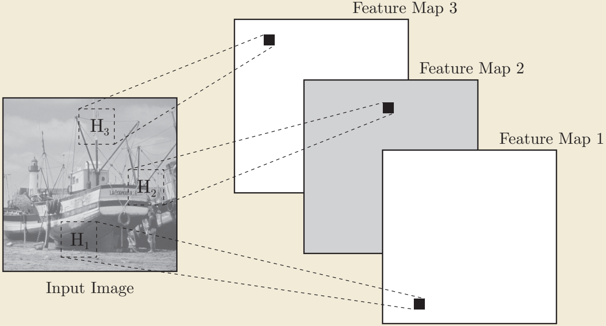
CNN: The Convolution Step

- The first layer in a CNN comprises the parameters of the kernel matrix. If the input nodes correspond to the (raw) pixels of an image array, the output of the convolutional layer is the corresponding feature map array.
- Thus, the parameters comprise the kernel array and they are **shared** among the input pixels; moreover, in place of the multiply-add operations convolutions are performed, instead.
- However, instead of a single kernel matrix, multiple ones are used; each one is expected to extract **different** type of information, to be encoded via a **different** feature map array. In the figure, three such kernel arrays are shown to "scan" the input image, searching for "hidden information".

CNN: The Convolution Step

- The first layer in a CNN comprises the parameters of the kernel matrix. If the input nodes correspond to the (raw) pixels of an image array, the output of the convolutional layer is the corresponding feature map array.
- Thus, the parameters comprise the kernel array and they are **shared** among the input pixels; moreover, in place of the multiply-add operations convolutions are performed, instead.
- However, instead of a single kernel matrix, multiple ones are used; each one is expected to extract **different** type of information, to be encoded via a **different** feature map array. In the figure, three such kernel arrays are shown to "scan" the input image, searching for "hidden information".

CNN: The Convolution Step



CNN: The Convolution Step

- There is strong evidence from the visual neuroscience that, similar computations are performed in the human brain. The idea of employing convolutions was first exploited in the neogognitron ([Fukushima]).
- Translation invariance: A welcome byproduct of the convolution step is that in this way, the network becomes invariant to translations. The same kernel matrix is slided all over the input image array. Thus, if an object has been moved within in an image, the only difference is that, in the feature map, the corresponding activity will move by the same amount of pixels.

CNN: The Convolution Step

- There is strong evidence from the visual neuroscience that, similar computations are performed in the human brain. The idea of employing convolutions was first exploited in the neogognitron ([Fukushima]).
- Translation invariance: A welcome byproduct of the convolution step is that in this way, the network becomes invariant to translations. The same kernel matrix is slided all over the input image array. Thus, if an object has been moved within in an image, the only difference is that, in the feature map, the corresponding activity will move by the same amount of pixels.

CNN: The Convolution Step

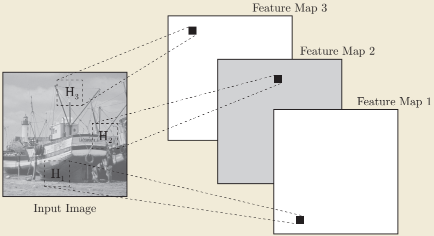
Bit of the jargon:

- Receptive field: Each pixel in the feature map array receives input from within a specific region of the previous (input) layer. This is known as the corresponding receptive field.

CNN: The Convolution Step

Bit of the jargon:

- Receptive field: Each pixel in the feature map array receives input from within a specific region of the previous (input) layer. This is known as the corresponding receptive field.
- Depth: This refers to the number of kernel matrices (filters) that are employed. For each filter, a corresponding feature map image array results.



The depth of the feature map array is **three**

CNN: The Convolution Step

Bit of the jargon (continued):

- Stride: This is the number of pixels by which one slides the filter matrix over the input matrix. When the stride is one, then we move the filters one pixel at a time. When the stride is two, then the filter jumps two pixels at a time as we slide them around. The larger the stride is the smaller the resulting feature maps become.
- Example for a 3×3 kernel matrix:

CNN: The Convolution Step

Bit of the jargon (continued):

- Stride: This is the number of pixels by which one slides the filter matrix over the input matrix. When the stride is one, then we move the filters one pixel at a time. When the stride is two, then the filter jumps two pixels at a time as we slide them around. The larger the stride is the smaller the resulting feature maps become.
- Example for a 3×3 kernel matrix:

CNN: The Convolution Step

Bit of the jargon (continued):

- Stride: This is the number of pixels by which one slides the filter matrix over the input matrix. When the stride is one, then we move the filters one pixel at a time. When the stride is two, then the filter jumps two pixels at a time as we slide them around. The larger the stride is the smaller the resulting feature maps become.
- Example for a 3×3 kernel matrix:

$$\text{Stride 1:} \quad \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}$$

CNN: The Convolution Step

Bit of the jargon (continued):

- Stride: This is the number of pixels by which one slides the filter matrix over the input matrix. When the stride is one, then we move the filters one pixel at a time. When the stride is two, then the filter jumps two pixels at a time as we slide them around. The larger the stride is the smaller the resulting feature maps become.
- Example for a 3×3 kernel matrix:

$$\text{Stride 1:} \quad \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}$$

CNN: The Convolution Step

Bit of the jargon (continued):

- Stride: This is the number of pixels by which one slides the filter matrix over the input matrix. When the stride is one, then we move the filters one pixel at a time. When the stride is two, then the filter jumps two pixels at a time as we slide them around. The larger the stride is the smaller the resulting feature maps become.
- Example for a 3×3 kernel matrix:

$$\text{Stride 1:} \quad \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}$$

CNN: The Convolution Step

Bit of the jargon (continued):

- Stride: This is the number of pixels by which one slides the filter matrix over the input matrix. When the stride is one, then we move the filters one pixel at a time. When the stride is two, then the filter jumps two pixels at a time as we slide them around. The larger the stride is the smaller the resulting feature maps become.
- Example for a 3×3 kernel matrix:

$$\text{Stride 1:} \quad \begin{bmatrix} * & * & * & * & * \\ \color{red}{*} & \color{red}{*} & \color{red}{*} & * & * \\ \color{red}{*} & \color{red}{*} & \color{red}{*} & * & * \\ \color{red}{*} & \color{red}{*} & \color{red}{*} & * & * \\ * & * & * & * & * \end{bmatrix}$$

CNN: The Convolution Step

Bit of the jargon (continued):

- Stride: This is the number of pixels by which one slides the filter matrix over the input matrix. When the stride is one, then we move the filters one pixel at a time. When the stride is two, then the filter jumps two pixels at a time as we slide them around. The larger the stride is the smaller the resulting feature maps become.
- Example for a 3×3 kernel matrix:

$$\text{Stride 1:} \quad \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}$$

CNN: The Convolution Step

Bit of the jargon (continued):

- Stride: This is the number of pixels by which one slides the filter matrix over the input matrix. When the stride is one, then we move the filters one pixel at a time. When the stride is two, then the filter jumps two pixels at a time as we slide them around. The larger the stride is the smaller the resulting feature maps become.
- Example for a 3×3 kernel matrix:

$$\text{Stride 1:} \quad \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}$$

CNN: The Convolution Step

Bit of the jargon (continued):

- Stride: This is the number of pixels by which one slides the filter matrix over the input matrix. When the stride is one, then we move the filters one pixel at a time. When the stride is two, then the filter jumps two pixels at a time as we slide them around. The larger the stride is the smaller the resulting feature maps become.
- Example for a 3×3 kernel matrix:

$$\text{Stride 1:} \quad \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ {\color{red}*} & {\color{red}*} & {\color{red}*} & * & * \\ {\color{red}*} & {\color{red}*} & {\color{red}*} & * & * \\ {\color{red}*} & {\color{red}*} & {\color{red}*} & * & * \end{bmatrix}$$

CNN: The Convolution Step

Bit of the jargon (continued):

- Stride: This is the number of pixels by which one slides the filter matrix over the input matrix. When the stride is one, then we move the filters one pixel at a time. When the stride is two, then the filter jumps two pixels at a time as we slide them around. The larger the stride is the smaller the resulting feature maps become.
- Example for a 3×3 kernel matrix:

$$\text{Stride 1:} \quad \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}$$

CNN: The Convolution Step

Bit of the jargon (continued):

- Stride: This is the number of pixels by which one slides the filter matrix over the input matrix. When the stride is one, then we move the filters one pixel at a time. When the stride is two, then the filter jumps two pixels at a time as we slide them around. The larger the stride is the smaller the resulting feature maps become.
- Example for a 3×3 kernel matrix:

$$\text{Stride 1:} \quad \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}$$

The resulting feature map array has size 9×9.

CNN: The Convolution Step

Bit of the jargon (continued):

- Stride: This is the number of pixels by which one slides the filter matrix over the input matrix. When the stride is one, then we move the filters one pixel at a time. When the stride is two, then the filter jumps two pixels at a time as we slide them around. The larger the stride is the smaller the resulting feature maps become.
- Example for a 3×3 kernel matrix:

$$\text{Stride 2:} \quad \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}$$

CNN: The Convolution Step

Bit of the jargon (continued):

- Stride: This is the number of pixels by which one slides the filter matrix over the input matrix. When the stride is one, then we move the filters one pixel at a time. When the stride is two, then the filter jumps two pixels at a time as we slide them around. The larger the stride is the smaller the resulting feature maps become.
- Example for a 3×3 kernel matrix:

$$\text{Stride 2:} \quad \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}$$

CNN: The Convolution Step

Bit of the jargon (continued):

- Stride: This is the number of pixels by which one slides the filter matrix over the input matrix. When the stride is one, then we move the filters one pixel at a time. When the stride is two, then the filter jumps two pixels at a time as we slide them around. The larger the stride is the smaller the resulting feature maps become.
- Example for a 3×3 kernel matrix:

$$\text{Stride 2:} \quad \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}$$

CNN: The Convolution Step

Bit of the jargon (continued):

- Stride: This is the number of pixels by which one slides the filter matrix over the input matrix. When the stride is one, then we move the filters one pixel at a time. When the stride is two, then the filter jumps two pixels at a time as we slide them around. The larger the stride is the smaller the resulting feature maps become.
- Example for a 3×3 kernel matrix:

$$\text{Stride 2:} \quad \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}$$

The resulting feature map array has size 4×4.

CNN: The Convolution Step

Bit of the jargon (continued):

- Zero-padding: Sometimes, we pad the input matrix with zeros around the border pixels, so that we can apply the filter to the bordering elements of the input image matrix.

CNN: The Convolution Step

Bit of the jargon (continued):

- Zero-padding: Sometimes, we pad the input matrix with zeros around the border pixels, so that we can apply the filter to the bordering elements of the input image matrix.

$$\text{Original array: } \begin{bmatrix} * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \\ * & * & * & * & * \end{bmatrix}$$

CNN: The Convolution Step

Bit of the jargon (continued):

- Zero-padding: Sometimes, we pad the input matrix with zeros around the border pixels, so that we can apply the filter to the bordering elements of the input image matrix.

After padding: $$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & * & * & 0 & 0 \\ 0 & 0 & * & * & * & * & * & 0 & 0 \\ 0 & 0 & * & * & * & * & * & 0 & 0 \\ 0 & 0 & * & * & * & * & * & 0 & 0 \\ 0 & 0 & * & * & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

CNN: The Nonlinearity Step

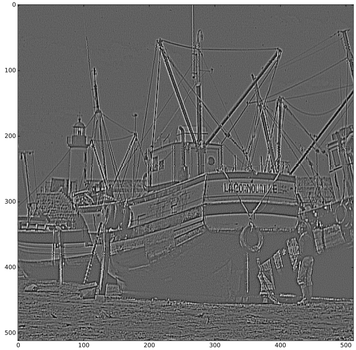
- Once the convolution step has been completed and feature maps have been produced, a nonlinearity is applied to each pixel/element of each feature map array. Typical nonlinearities used are the sigmoid functions or the ReLUs. The latter seem to be the preferable choice currently.
- Note that after convolution, some of the matrix elements can become negative. These are set equal to zero, after, e.g., the application of the ReLU.

CNN: The Nonlinearity Step

- Once the convolution step has been completed and feature maps have been produced, a nonlinearity is applied to each pixel/element of each feature map array. Typical nonlinearities used are the sigmoid functions or the ReLUs. The latter seem to be the preferable choice currently.

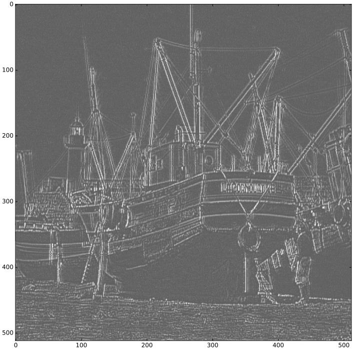
- Note that after convolution, some of the matrix elements can become negative. These are set equal to zero, after, e.g., the application of the ReLU.

CNN: The Nonlinearity Step



Feature map

CNN: The Nonlinearity Step



Feature map after the ReLU nonlinearity

CNN: The Pooling Step

- Pooling reduces the size of the feature map. To this end, one slides a window, e.g, 2×2, over the feature map, and for each location of the window a single value is selected. This is a downsampling operation. Pooling is also contributing in building into the network shift invariance properties [Bruna, et. al. 2013].
- There are different scenarios. In the max pooling, the maximum value is selected. In the average pooling, the average value is selected. Other variants do, also, exist.
- The max pooling for a window of size 2×2 is shown below:

$$\underbrace{\begin{bmatrix} 2 & 3 & \mathbf{7} & 1 & 4 & 5 \\ 4 & \mathbf{5} & 0 & 6 & \mathbf{7} & 1 \\ \mathbf{6} & 2 & 1 & 3 & 2 & 3 \\ 4 & 5 & 6 & \mathbf{8} & 4 & \mathbf{5} \\ \mathbf{5} & 3 & 2 & 1 & 2 & 1 \\ 4 & 2 & 1 & \mathbf{8} & \mathbf{6} & 3 \end{bmatrix}}_{\text{before pooling}} \qquad \underbrace{\begin{bmatrix} 5 & 7 & 7 \\ 6 & 8 & 5 \\ 4 & 8 & 6 \end{bmatrix}}_{\text{after pooling}}$$

CNN: The Pooling Step

- Pooling reduces the size of the feature map. To this end, one slides a window, e.g, 2×2, over the feature map, and for each location of the window a single value is selected. This is a downsampling operation. Pooling is also contributing in building into the network shift invariance properties [Bruna, et. al. 2013].
- There are different scenarios. In the max pooling, the maximum value is selected. In the average pooling, the average value is selected. Other variants do, also, exist.
- The max pooling for a window of size 2×2 is shown below:

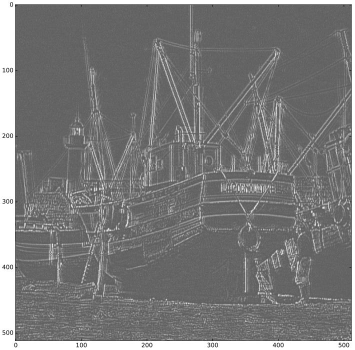
$$\underbrace{\begin{bmatrix} 2 & 3 & \mathbf{7} & 1 & 4 & 5 \\ 4 & \mathbf{5} & 0 & 6 & \mathbf{7} & 1 \\ \mathbf{6} & 2 & 1 & 3 & 2 & 3 \\ 4 & 5 & 6 & \mathbf{8} & 4 & \mathbf{5} \\ \mathbf{5} & 3 & 2 & 1 & 2 & 1 \\ 4 & 2 & 1 & \mathbf{8} & \mathbf{6} & 3 \end{bmatrix}}_{\text{before pooling}} \qquad \underbrace{\begin{bmatrix} 5 & 7 & 7 \\ 6 & 8 & 5 \\ 4 & 8 & 6 \end{bmatrix}}_{\text{after pooling}}$$

CNN: The Pooling Step

- Pooling reduces the size of the feature map. To this end, one slides a window, e.g, 2×2, over the feature map, and for each location of the window a single value is selected. This is a downsampling operation. Pooling is also contributing in building into the network shift invariance properties [Bruna, et. al. 2013].
- There are different scenarios. In the max pooling, the maximum value is selected. In the average pooling, the average value is selected. Other variants do, also, exist.
- The max pooling for a window of size 2×2 is shown below:

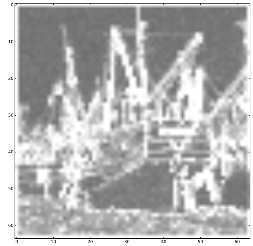
$$\underbrace{\begin{bmatrix} 2 & 3 & \mathbf{7} & 1 & 4 & 5 \\ 4 & \mathbf{5} & 0 & 6 & \mathbf{7} & 1 \\ \mathbf{6} & 2 & 1 & 3 & 2 & 3 \\ 4 & 5 & 6 & \mathbf{8} & 4 & \mathbf{5} \\ \mathbf{5} & 3 & 2 & 1 & 2 & 1 \\ 4 & 2 & 1 & \mathbf{8} & \mathbf{6} & 3 \end{bmatrix}}_{\text{before pooling}} \qquad \underbrace{\begin{bmatrix} 5 & 7 & 7 \\ 6 & 8 & 5 \\ 5 & 8 & 6 \end{bmatrix}}_{\text{after pooling}}$$

CNN: The Pooling Step



Feature map after the ReLU nonlinearity

CNN: The Pooling Step



Feature map after the ReLU+ max 8×8 pooling

The Full CNN

- The three stages discussed before, i.e, the convolution, the nonlinearity and the pooling steps, comprise a single layer of a convolutional network.
- In practice, a CNN comprises a series of such convolution layers. The first one is presented with the input image array. The second one receives as inputs the pooled features maps of the previous layer, and so on. Networks with 20-25 layers have been reported in practical applications.
- Finally, the feature map arrays of the last convolution layer are provided as inputs to a classifier. A softmax output NN is a popular choice, yet SVMs or other predictors can also be employed.
- Training of the full CNN takes place via a modified backpropagation algorithm. The modification is due to the weight sharing of the convolutional layers.

The Full CNN

- The three stages discussed before, i.e, the convolution, the nonlinearity and the pooling steps, comprise a single layer of a convolutional network.
- In practice, a CNN comprises a series of such convolution layers. The first one is presented with the input image array. The second one receives as inputs the pooled features maps of the previous layer, and so on. Networks with 20-25 layers have been reported in practical applications.
- Finally, the feature map arrays of the last convolution layer are provided as inputs to a classifier. A softmax output NN is a popular choice, yet SVMs or other predictors can also be employed.
- Training of the full CNN takes place via a modified backpropagation algorithm. The modification is due to the weight sharing of the convolutional layers.

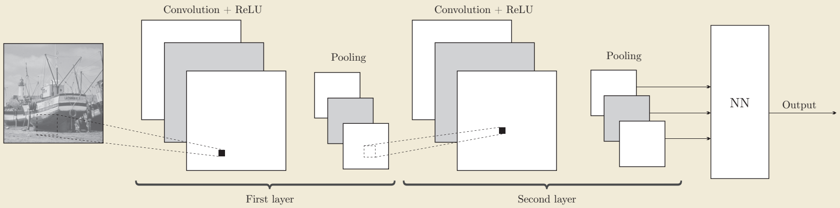
The Full CNN

- The three stages discussed before, i.e, the convolution, the nonlinearity and the pooling steps, comprise a single layer of a convolutional network.
- In practice, a CNN comprises a series of such convolution layers. The first one is presented with the input image array. The second one receives as inputs the pooled features maps of the previous layer, and so on. Networks with 20-25 layers have been reported in practical applications.
- Finally, the feature map arrays of the last convolution layer are provided as inputs to a classifier. A softmax output NN is a popular choice, yet SVMs or other predictors can also be employed.
- Training of the full CNN takes place via a modified backpropagation algorithm. The modification is due to the weight sharing of the convolutional layers.

The Full CNN

- The three stages discussed before, i.e, the convolution, the nonlinearity and the pooling steps, comprise a single layer of a convolutional network.
- In practice, a CNN comprises a series of such convolution layers. The first one is presented with the input image array. The second one receives as inputs the pooled features maps of the previous layer, and so on. Networks with 20-25 layers have been reported in practical applications.
- Finally, the feature map arrays of the last convolution layer are provided as inputs to a classifier. A softmax output NN is a popular choice, yet SVMs or other predictors can also be employed.
- Training of the full CNN takes place via a modified backpropagation algorithm. The modification is due to the weight sharing of the convolutional layers.

The Full CNN



Convolution + ReLU
Pooling
Convolution + ReLU
Pooling
NN
Output
First layer
Second layer

The Full CNN-An Example

The Full CNN

- Some “famous” CNNs:
  - Lanet: [Lecun] 1990’s. The LeNet architecture was used mainly for character recognition tasks such as reading zip codes, digits, etc.
  - Alexnet: [Krizhevsky, et. al.] 2012. The network has 60 million parameters and 500,000 neurons and it consists of five convolutional layers, and a two-layer NN with a softmax output.
  - ZF Net: [Zeidler-Fergus] 2013. It was an improvement on AlexNet by tweaking the architecture hyperparameters.
  - GoogLeNet: [Szegedy et al.] 2014.
  - ResNets [He, et.al.] 2015.
  - DenseNet: [Huang et.al.] 2016.
- Other succesful applications, besides machine vision and image recognition, CNNs have succefully been used in natural language processing, e.g., [Zhang, et.al, 2016].

Recurrent Neural Networks (RNN)

- Recall that at the heart of the success of CNNs lies the concept of weight sharing.
- Without having to assign specific weights to each individual pixel, scaling to different sizes of images can be readily done.
- The concept of weight sharing will now be applied to the case of sequential data. That is, the input data a) are not independent, b) they occur in sequence and more important c) the specific order in which they occur carries important information.
- Some typical examples of such applications are speech recognition, language modeling, machine translation, where the order in which phonemes or words occur carry important information.

Recurrent Neural Networks (RNN)

- Recall that at the heart of the success of CNNs lies the concept of weight sharing.
- Without having to assign specific weights to each individual pixel, scaling to different sizes of images can be readily done.
- The concept of weight sharing will now be applied to the case of sequential data. That is, the input data a) are not independent, b) they occur in sequence and more important c) the specific order in which they occur carries important information.
- Some typical examples of such applications are speech recognition, language modeling, machine translation, where the order in which phonemes or words occur carry important information.

Recurrent Neural Networks (RNN)

- Recall that at the heart of the success of CNNs lies the concept of weight sharing.
- Without having to assign specific weights to each individual pixel, scaling to different sizes of images can be readily done.
- The concept of weight sharing will now be applied to the case of sequential data. That is, the input data a) are not independent, b) they occur in sequence and more important c) the specific order in which they occur carries important information.
- Some typical examples of such applications are speech recognition, language modeling, machine translation, where the order in which phonemes or words occur carry important information.

Recurrent Neural Networks (RNN)

- Recall that at the heart of the success of CNNs lies the concept of weight sharing.
- Without having to assign specific weights to each individual pixel, scaling to different sizes of images can be readily done.
- The concept of weight sharing will now be applied to the case of sequential data. That is, the input data a) are not independent, b) they occur in sequence and more important c) the specific order in which they occur carries important information.
- Some typical examples of such applications are speech recognition, language modeling, machine translation, where the order in which phonemes or words occur carry important information.

Recurrent Neural Networks (RNN)

- Recall that at the heart of the success of CNNs lies the concept of weight sharing.
- Without having to assign specific weights to each individual pixel, scaling to different sizes of images can be readily done.
- The concept of weight sharing will now be applied to the case of sequential data. That is, the input data a) are not independent, b) they occur in sequence and more important c) the specific order in which they occur carries important information.
- Some typical examples of such applications are speech recognition, language modeling, machine translation, where the order in which phonemes or words occur carry important information.

Recurrent Neural Networks (RNN)

- Recall that at the heart of the success of CNNs lies the concept of weight sharing.
- Without having to assign specific weights to each individual pixel, scaling to different sizes of images can be readily done.
- The concept of weight sharing will now be applied to the case of sequential data. That is, the input data a) are not independent, b) they occur in sequence and more important c) the specific order in which they occur carries important information.
- Some typical examples of such applications are speech recognition, language modeling, machine translation, where the order in which phonemes or words occur carry important information.

Recurrent Neural Networks (RNN)

- Recall that at the heart of the success of CNNs lies the concept of weight sharing.
- Without having to assign specific weights to each individual pixel, scaling to different sizes of images can be readily done.
- The concept of weight sharing will now be applied to the case of sequential data. That is, the input data a) are not independent, b) they occur in sequence and more important c) the specific order in which they occur carries important information.
- Some typical examples of such applications are speech recognition, language modeling, machine translation, where the order in which phonemes or words occur carry important information.

Recurrent Neural Networks (RNN)

- Recall that at the heart of the success of CNNs lies the concept of weight sharing.
- Without having to assign specific weights to each individual pixel, scaling to different sizes of images can be readily done.
- The concept of weight sharing will now be applied to the case of sequential data. That is, the input data a) are not independent, b) they occur in sequence and more important c) the specific order in which they occur carries important information.
- Some typical examples of such applications are speech recognition, language modeling, machine translation, where the order in which phonemes or words occur carry important information.

Recurrent Neural Networks (RNN)

- RNNs are built around the concept of the **state**.
- The concept of a state vector is at the heart of many dynamical systems, such as hidden Markov models (HMM) and Kalman filters. The state vector comprises the **memory** of the system up to time n. That is, it encodes the history of the system. The response-output of the system at time n, depends on the state vector as well as the input at time n.
- The idea in RNNs is to apply the same type of operations (weight sharing) at each time instant (reccurency) by involving the state as well as the currently available input vectors.

Recurrent Neural Networks (RNN)

- RNNs are built around the concept of the **state**.
- The concept of a state vector is at the heart of many dynamical systems, such as hidden Markov models (HMM) and Kalman filters. The state vector comprises the **memory** of the system up to time n. That is, it encodes the history of the system. The response-output of the system at time n, depends on the state vector as well as the input at time n.
- The idea in RNNs is to apply the same type of operations (weight sharing) at each time instant (reccurency) by involving the state as well as the currently available input vectors.

Recurrent Neural Networks (RNN)

- RNNs are built around the concept of the **state**.
- The concept of a state vector is at the heart of many dynamical systems, such as hidden Markov models (HMM) and Kalman filters. The state vector comprises the **memory** of the system up to time n. That is, it encodes the history of the system. The response-output of the system at time n, depends on the state vector as well as the input at time n.
- The idea in RNNs is to apply the same type of operations (weight sharing) at each time instant (reccurency) by involving the state as well as the currently available input vectors.

Recurrent Neural Networks (RNN)

The variables involved in an RNN are:

- The state vector at time n, denoted as $\boldsymbol{h}_n$. The symbol reminds us that the state variables correspond to the hidden layer, in a NN terminology.
- The input vector, $\boldsymbol{x}_n$.
- The output vector, $\hat{\boldsymbol{y}}_n$, and the desired output vector, $\boldsymbol{y}_n$, used during training.
- The RNN model is described in term of a set of parameters, to be learned during training; namely, the matrices U, W, V and the vectors $\boldsymbol{b}, \boldsymbol{c}$.
- The basic RNN model is described by:

$$\begin{aligned} \boldsymbol{h}_n &= f(U\boldsymbol{x}_n + W\boldsymbol{h}_{n-1} + \boldsymbol{b}) \\ \hat{\boldsymbol{y}}_n &= g(V\boldsymbol{h}_n + \boldsymbol{c}). \end{aligned}$$

Recurrent Neural Networks (RNN)

The variables involved in an RNN are:

- The state vector at time n, denoted as $\boldsymbol{h}_n$. The symbol reminds us that the state variables correspond to the hidden layer, in a NN terminology.
- The input vector, $\boldsymbol{x}_n$.
- The output vector, $\hat{\boldsymbol{y}}_n$, and the desired output vector, $\boldsymbol{y}_n$, used during training.
- The RNN model is described in term of a set of parameters, to be learned during training; namely, the matrices U, W, V and the vectors $\boldsymbol{b}, \boldsymbol{c}$.
- The basic RNN model is described by:

$$\begin{aligned} \boldsymbol{h}_n &= f(U\boldsymbol{x}_n + W\boldsymbol{h}_{n-1} + \boldsymbol{b}) \\ \hat{\boldsymbol{y}}_n &= g(V\boldsymbol{h}_n + \boldsymbol{c}). \end{aligned}$$

Recurrent Neural Networks (RNN)

The variables involved in an RNN are:

- The state vector at time n, denoted as $\boldsymbol{h}_n$. The symbol reminds us that the state variables correspond to the hidden layer, in a NN terminology.
- The input vector, $\boldsymbol{x}_n$.
- The output vector, $\hat{\boldsymbol{y}}_n$, and the desired output vector, $\boldsymbol{y}_n$, used during training.
- The RNN model is described in term of a set of parameters, to be learned during training; namely, the matrices U, W, V and the vectors $\boldsymbol{b}, \boldsymbol{c}$.
- The basic RNN model is described by:

$$\begin{aligned} \boldsymbol{h}_n &= f(U\boldsymbol{x}_n + W\boldsymbol{h}_{n-1} + \boldsymbol{b}) \\ \hat{\boldsymbol{y}}_n &= g(V\boldsymbol{h}_n + \boldsymbol{c}). \end{aligned}$$

Recurrent Neural Networks (RNN)

The variables involved in an RNN are:

- The state vector at time n, denoted as $\boldsymbol{h}_n$. The symbol reminds us that the state variables correspond to the hidden layer, in a NN terminology.
- The input vector, $\boldsymbol{x}_n$.
- The output vector, $\hat{\boldsymbol{y}}_n$, and the desired output vector, $\boldsymbol{y}_n$, used during training.
- The RNN model is described in term of a set of parameters, to be learned during training; namely, the matrices U, W, V and the vectors $\boldsymbol{b}, \boldsymbol{c}$.
- The basic RNN model is described by:

$$\begin{aligned} \boldsymbol{h}_n &= f(U\boldsymbol{x}_n + W\boldsymbol{h}_{n-1} + \boldsymbol{b}) \\ \hat{\boldsymbol{y}}_n &= g(V\boldsymbol{h}_n + \boldsymbol{c}). \end{aligned}$$

Recurrent Neural Networks (RNN)

The variables involved in an RNN are:

- The state vector at time n, denoted as $\boldsymbol{h}_n$. The symbol reminds us that the state variables correspond to the hidden layer, in a NN terminology.
- The input vector, $\boldsymbol{x}_n$.
- The output vector, $\hat{\boldsymbol{y}}_n$, and the desired output vector, $\boldsymbol{y}_n$, used during training.
- The RNN model is described in term of a set of parameters, to be learned during training; namely, the matrices U, W, V and the vectors $\boldsymbol{b}, \boldsymbol{c}$.
- The basic RNN model is described by:

$$\begin{aligned} \boldsymbol{h}_n &= f(U\boldsymbol{x}_n + W\boldsymbol{h}_{n-1} + \boldsymbol{b}) \\ \hat{\boldsymbol{y}}_n &= g(V\boldsymbol{h}_n + \boldsymbol{c}). \end{aligned}$$

Recurrent Neural Networks (RNN)

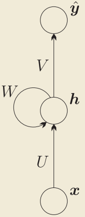
- Typical choices of the nonlinear functions are: for the state equation $f = \tanh$ or $f = \text{ReLU}$ and for the output one $g = \text{softmax}$.
- Note that the parameters are shared across all time instants; thus scaling to different sequence lengths can be readily done.

Recurrent Neural Networks (RNN)

- Typical choices of the nonlinear functions are: for the state equation $f = \text{tanh}$ or $f = \text{ReLU}$ and for the output one $g = \text{softmax}$.
- Note that the parameters are shared across all time instants; thus scaling to different sequence lengths can be readily done.

Recurrent Neural Networks (RNN)

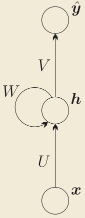
- Typical choices of the nonlinear functions are: for the state equation $f = \text{tanh}$ or $f = \text{ReLU}$ and for the output one $g = \text{softmax}$.
- Note that the parameters are shared across all time instants; thus scaling to different sequence lengths can be readily done.



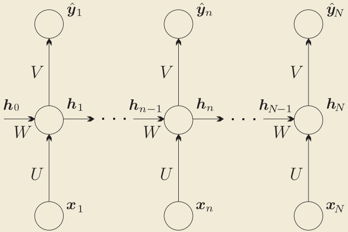
Graphical RNN model

Recurrent Neural Networks (RNN)

- Typical choices of the nonlinear functions are: for the state equation $f = \text{tanh}$ or $f = \text{ReLU}$ and for the output one $g = \text{softmax}$.
- Note that the parameters are shared across all time instants; thus scaling to different sequence lengths can be readily done.



Graphical RNN model



Unfolded RNN graphical model

Backpropagation Through Time (BPTT)

- Training an RNN is similar to training feedforward NNs. It turns out that the required gradients of the cost function, w.r. to the unknown parameters, takes place recursively, by starting at the latest time instant, say N, and go backwards in time, $n = N-1, N-2, \ldots$. This is the reason that the algorithm is known as Backpropagation Through Time (BPTT).
- At the heart of the algorithm lies the computation of the gradient of the cost function w.r. to the state vectors. This computation follows a backpropagation rationale.

Backpropagation Through Time (BPTT)

- Training an RNN is similar to training feedforward NNs. It turns out that the required gradients of the cost function, w.r. to the unknown parameters, takes place recursively, by starting at the latest time instant, say N, and go backwards in time, $n = N-1, N-2, \ldots$. This is the reason that the algorithm is known as Backpropagation Through Time (BPTT).
- At the heart of the algorithm lies the computation of the gradient of the cost function w.r. to the state vectors. This computation follows a backpropagation rationale.

Backpropagation Through Time (BPTT)

- The cost function is the sum over time, n, of the loss function contributions, that depend on the corresponding values of $\boldsymbol{h}_n$, $\boldsymbol{x}_n$.
- That is,

$$J = \sum_{n=1}^{N} J_n(\boldsymbol{y}_n, \hat{\boldsymbol{y}}_n), \ \hat{\boldsymbol{y}}_n = g(\boldsymbol{h}_n, V, \boldsymbol{c}), \text{ and } \boldsymbol{h}_n = f(\boldsymbol{h}_{n-1}, U, W, \boldsymbol{b}, \boldsymbol{x}_n).$$

- In words, each $\boldsymbol{h}_n$ affects J in two ways:
  - Directly, through J_n.
  - Indirectly, via the chain that is imposed by the RNN structure:

$$\boldsymbol{h}_n \rightarrow \boldsymbol{h}_{n+1} \rightarrow \ldots \rightarrow \boldsymbol{h}_N, \ n = 1, 2, \ldots, N-1.$$

- The above leads to the following recursive computation, that propagates backwards:

$$\frac{\partial J}{\partial \boldsymbol{h}_n} = \frac{\partial \boldsymbol{h}_{n+1}}{\partial \boldsymbol{h}_n} \frac{\partial J}{\partial \boldsymbol{h}_{n+1}} + \left(\frac{\partial \hat{\boldsymbol{y}}_n}{\partial \boldsymbol{h}_n}\right)^T \frac{\partial J}{\partial \hat{\boldsymbol{y}}_n}.$$

Backpropagation Through Time (BPTT)

- The cost function is the sum over time, n, of the loss function contributions, that depend on the corresponding values of $\boldsymbol{h}_n$, $\boldsymbol{x}_n$.
- That is,

$$J = \sum_{n=1}^{N} J_n(\boldsymbol{y}_n, \hat{\boldsymbol{y}}_n), \ \hat{\boldsymbol{y}}_n = g(\boldsymbol{h}_n, V, \boldsymbol{c}), \text{ and } \boldsymbol{h}_n = f(\boldsymbol{h}_{n-1}, U, W, \boldsymbol{b}, \boldsymbol{x}_n).$$

- In words, each $\boldsymbol{h}_n$ affects J in two ways:
  - Directly, through J_n.
  - Indirectly, via the chain that is imposed by the RNN structure:

$$\boldsymbol{h}_n \rightarrow \boldsymbol{h}_{n+1} \rightarrow \ldots \rightarrow \boldsymbol{h}_N, \ n = 1, 2, \ldots, N-1.$$

- The above leads to the following recursive computation, that propagates backwards:

$$\frac{\partial J}{\partial \boldsymbol{h}_n} = \frac{\partial \boldsymbol{h}_{n+1}}{\partial \boldsymbol{h}_n} \frac{\partial J}{\partial \boldsymbol{h}_{n+1}} + \left(\frac{\partial \hat{\boldsymbol{y}}_n}{\partial \boldsymbol{h}_n}\right)^T \frac{\partial J}{\partial \hat{\boldsymbol{y}}_n}.$$

Backpropagation Through Time (BPTT)

- The cost function is the sum over time, n, of the loss function contributions, that depend on the corresponding values of $\boldsymbol{h}_n$, $\boldsymbol{x}_n$.
- That is,

$$J = \sum_{n=1}^{N} J_n(\boldsymbol{y}_n, \hat{\boldsymbol{y}}_n), \; \hat{\boldsymbol{y}}_n = g(\boldsymbol{h}_n, V, \boldsymbol{c}), \text{ and } \boldsymbol{h}_n = f(\boldsymbol{h}_{n-1}, U, W, \boldsymbol{b}, \boldsymbol{x}_n).$$

- In words, each $\boldsymbol{h}_n$ affects J in two ways:
  - Directly, through J_n.
  - Indirectly, via the chain that is imposed by the RNN structure:

$$\boldsymbol{h}_n \rightarrow \boldsymbol{h}_{n+1} \rightarrow \ldots \rightarrow \boldsymbol{h}_N, \; n = 1, 2, \ldots, N-1.$$

- The above leads to the following recursive computation, that propagates backwards:

$$\frac{\partial J}{\partial \boldsymbol{h}_n} = \frac{\partial \boldsymbol{h}_{n+1}}{\partial \boldsymbol{h}_n} \frac{\partial J}{\partial \boldsymbol{h}_{n+1}} + \left(\frac{\partial \hat{\boldsymbol{y}}_n}{\partial \boldsymbol{h}_n} \right)^T \frac{\partial J}{\partial \hat{\boldsymbol{y}}_n}.$$

Recurrent Neural Networks (RNN)

Forward pass:

- Starting at $n = 1$, compute in sequence,

$$(\boldsymbol{h}_1, \hat{\boldsymbol{y}}_1) \rightarrow (\boldsymbol{h}_2\ \hat{\boldsymbol{y}}_2) \rightarrow \ldots \rightarrow (\boldsymbol{h}_N, \hat{\boldsymbol{y}}_N).$$

Backward pass:

- Starting at $n = N$, compute in sequence,

$$\frac{\partial J}{\partial \boldsymbol{h}_N} \rightarrow \frac{\partial J}{\partial \boldsymbol{h}_{N-1}} \rightarrow \ldots \rightarrow \frac{\partial J}{\partial \boldsymbol{h}_1}.$$

Recurrent Neural Networks (RNN)

Forward pass:

- Starting at $n = 1$, compute in sequence,

$$(\boldsymbol{h}_1, \hat{\boldsymbol{y}}_1) \rightarrow (\boldsymbol{h}_2\ \hat{\boldsymbol{y}}_2) \rightarrow \dots \rightarrow (\boldsymbol{h}_N, \hat{\boldsymbol{y}}_N).$$

Backward pass:

- Starting at $n = N$, compute in sequence,

$$\frac{\partial J}{\partial \boldsymbol{h}_N} \rightarrow \frac{\partial J}{\partial \boldsymbol{h}_{N-1}} \rightarrow \dots \rightarrow \frac{\partial J}{\partial \boldsymbol{h}_1}.$$

Diminishing And Expanding Gradients

- For the same reasons as for the backpropagation for the feedforward NNs, the BPTT version also suffers from the diminishing/expanding values of the gradients, due a) to the products introduced by the chain differentiation and b) the saturating nature of the gradient of the tanh function.
- Moreover, in the case of RNN, this is usually more serious, since for long enough sequences, the number of involved backward steps can be quite large.
- There are two paths to bypass this problem. One is to involve the ReLU nonlinearity. The other one is more clever and it comprises in modifying the basic structure of the RNN.

Diminishing And Expanding Gradients

- For the same reasons as for the backpropagation for the feedforward NNs, the BPTT version also suffers from the diminishing/expanding values of the gradients, due a) to the products introduced by the chain differentiation and b) the saturating nature of the gradient of the tanh function.
- Moreover, in the case of RNN, this is usually more serious, since for long enough sequences, the number of involved backward steps can be quite large.
- There are two paths to bypass this problem. One is to involve the ReLU nonlinearity. The other one is more clever and it comprises in modifying the basic structure of the RNN.

Diminishing And Expanding Gradients

- For the same reasons as for the backpropagation for the feedforward NNs, the BPTT version also suffers from the diminishing/expanding values of the gradients, due a) to the products introduced by the chain differentiation and b) the saturating nature of the gradient of the tanh function.
- Moreover, in the case of RNN, this is usually more serious, since for long enough sequences, the number of involved backward steps can be quite large.
- There are two paths to bypass this problem. One is to involve the ReLU nonlinearity. The other one is more clever and it comprises in modifying the basic structure of the RNN.

Diminishing And Expanding Gradients

- For the same reasons as for the backpropagation for the feedforward NNs, the BPTT version also suffers from the diminishing/expanding values of the gradients, due a) to the products introduced by the chain differentiation and b) the saturating nature of the gradient of the tanh function.
- Moreover, in the case of RNN, this is usually more serious, since for long enough sequences, the number of involved backward steps can be quite large.
- There are two paths to bypass this problem. One is to involve the ReLU nonlinearity. The other one is more clever and it comprises in modifying the basic structure of the RNN.

Diminishing And Expanding Gradients

- For the same reasons as for the backpropagation for the feedforward NNs, the BPTT version also suffers from the diminishing/expanding values of the gradients, due a) to the products introduced by the chain differentiation and b) the saturating nature of the gradient of the tanh function.
- Moreover, in the case of RNN, this is usually more serious, since for long enough sequences, the number of involved backward steps can be quite large.
- There are two paths to bypass this problem. One is to involve the ReLU nonlinearity. The other one is more clever and it comprises in modifying the basic structure of the RNN.

The Long Short-Term Memory Network (LSTM)

- The key idea behind the LSTM networks, proposed by [Hochreiter and Scmidhuber, 91] is the so called cell state; they have been explicitly designed to overcome the diminishing/expanding gradient problem in RNNs.
- The LSTM networks have the build in ability to **control** the information flow into and out the system's memory, via nonlinear elements known as gates.
- The gates are implemented via the logistic sigmoid nonlineariy, whose output varies between zero and one. Their imposed control is equivalent with a respective weighting on the involved time updates. More important, this weighting (large or small) is dictated, in **context**; that is, it depends on the current input as well as the state (memory) vectors.

The Long Short-Term Memory Network (LSTM)

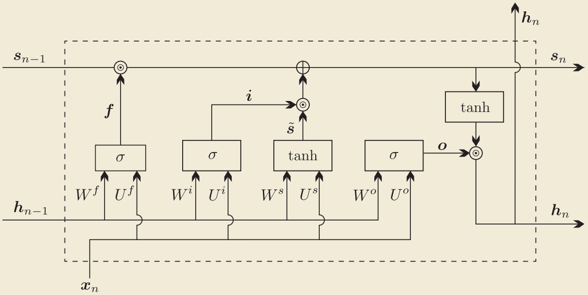
- The key idea behind the LSTM networks, proposed by [Hochreiter and Scmidhuber, 91] is the so called cell state; they have been explicitly designed to overcome the diminishing/expanding gradient problem in RNNs.
- The LSTM networks have the build in ability to **control** the information flow into and out the system's memory, via nonlinear elements known as gates.
- The gates are implemented via the logistic sigmoid nonlineariy, whose output varies between zero and one. Their imposed control is equivalent with a respective weighting on the involved time updates. More important, this weighting (large or small) is dictated, in **context**; that is, it depends on the current input as well as the state (memory) vectors.

The Long Short-Term Memory Network (LSTM)

- The key idea behind the LSTM networks, proposed by [Hochreiter and Scmidhuber, 91] is the so called cell state; they have been explicitly designed to overcome the diminishing/expanding gradient problem in RNNs.
- The LSTM networks have the build in ability to **control** the information flow into and out the system's memory, via nonlinear elements known as gates.
- The gates are implemented via the logistic sigmoid nonlineariy, whose output varies between zero and one. Their imposed control is equivalent with a respective weighting on the involved time updates. More important, this weighting (large or small) is dictated, in **context**; that is, it depends on the current input as well as the state (memory) vectors.

The Long Short-Term Memory Network (LSTM)

- In LSTM networks, besides the state vector $\boldsymbol{h}_n$, the cell state vector is propagated through time. The basic cell/module that comprises an LSTM is shown below:



$$\boldsymbol{f} = \sigma(U^f \boldsymbol{x}_n + W^f \boldsymbol{h}_{n-1} + \boldsymbol{b}^f)$$

$$\boldsymbol{i} = \sigma(U^i \boldsymbol{x}_n + W^i \boldsymbol{h}_{n-1} + \boldsymbol{b}^i)$$

$$\boldsymbol{o} = \sigma(U^o \boldsymbol{x}_n + W^o \boldsymbol{h}_{n-1} + \boldsymbol{b}^o)$$

$$\tilde{\boldsymbol{s}} = \tanh(U^s \boldsymbol{x}_n + W^s \boldsymbol{h}_{n-1} + \boldsymbol{b}^s)$$

$$\boldsymbol{s}_n = \boldsymbol{s}_{n-1} \odot \boldsymbol{f} + \boldsymbol{i} \odot \tilde{\boldsymbol{s}}$$

$$\boldsymbol{h}_n = \boldsymbol{o} \odot \tanh(\boldsymbol{s}_n)$$

The Long Short-Term Memory Network (LSTM)

- RNNs, mainly via the LSTM implementation, have been applied and successfully used in a number of areas, such as language processing, machine translation, speech processing, visual semantic alignment for generating image descriptions in machine vision.
- For example in language modeling, the input is typically a sequence of words. Each word is represented as a number (one-hot vectors). This is basically a pointer to the available vocabulary of words. The output is the sequence of predicted words. During training, we set $\boldsymbol{y}_n = \boldsymbol{x}_{n+1}$. That is, the RNN is trained as a nonlinear predictor.

The Long Short-Term Memory Network (LSTM)

- RNNs, mainly via the LSTM implementation, have been applied and successfully used in a number of areas, such as language processing, machine translation, speech processing, visual semantic alignment for generating image descriptions in machine vision.
- For example in language modeling, the input is typically a sequence of words. Each word is represented as a number (one-hot vectors). This is basically a pointer to the available vocabulary of words. The output is the sequence of predicted words. During training, we set $\boldsymbol{y}_n = \boldsymbol{x}_{n+1}$. That is, the RNN is trained as a nonlinear predictor.

Deep and Bi-directional RNNs

- Deep RNNs: Instead of computing one layer of states, one can stack a number of RNNs, one on top of the other and form deep RNNs. In a deep RNN, the state vector of the previous layer becomes the input to the next layer. The output of the network is provided in terms of the states of last layer.
- Bi-directional RNNs: As the term reveals, in a bidirectional RNN, there are two sets of state vectors, namely $\boldsymbol{h}_n^f$ that propagates in the forward direction and $\boldsymbol{h}_n^b$ that runs in the backward direction. Thus, the output vector, $\boldsymbol{y}_n$, at time n is made to depend both on the past as well as on the future.

Deep and Bi-directional RNNs

- Deep RNNs: Instead of computing one layer of states, one can stack a number of RNNs, one on top of the other and form deep RNNs. In a deep RNN, the state vector of the previous layer becomes the input to the next layer. The output of the network is provided in terms of the states of last layer.
- Bi-directional RNNs: As the term reveals, in a bidirectional RNN, there are two sets of state vectors, namely $\boldsymbol{h}_n^f$ that propagates in the forward direction and $\boldsymbol{h}_n^b$ that runs in the backward direction. Thus, the output vector, $\boldsymbol{y}_n$, at time n is made to depend both on the past as well as on the future.

Attention

- As the name suggests, the concept of attention draws heavily on the attention mechanism found in humans.
- For example in our visual system, the “attention” provides us with the ability to focus on the most important information that resides in the scene; important information is always in **context**, that is, in relation to what we are looking for.
- In machine learning, one of the most popular ways to implement attention is via a weighting (linear transformation) on variables that the output depends on. The weights of such a transformation are learned during training (various algorithms have been suggested).

Attention

- As the name suggests, the concept of attention draws heavily on the attention mechanism found in humans.
- For example in our visual system, the “attention” provides us with the ability to focus on the most important information that resides in the scene; important information is always in **context**, that is, in relation to what we are looking for.
- In machine learning, one of the most popular ways to implement attention is via a weighting (linear transformation) on variables that the output depends on. The weights of such a transformation are learned during training (various algorithms have been suggested).

Attention

- As the name suggests, the concept of attention draws heavily on the attention mechanism found in humans.
- For example in our visual system, the “attention” provides us with the ability to focus on the most important information that resides in the scene; important information is always in **context**, that is, in relation to what we are looking for.
- In machine learning, one of the most popular ways to implement attention is via a weighting (linear transformation) on variables that the output depends on. The weights of such a transformation are learned during training (various algorithms have been suggested).

Attention

- In the framework of RNNs, recall that the output, $\boldsymbol{y}_n$, at time n is given in terms of the state values at the respective time, i.e., $\boldsymbol{h}_n$. In other words, the most recent information related to the whole past history (as this is encoded in the state vector) is used.
- However, it is rather unreasonable to assume that the long history of a sequence is sufficiently represented in the last state vector. For example in a sentence (and depending on the structure of a language), the next word may depend mostly on the meaning of a word that appeared some time earlier in the sequence of the words, or on a combination of words.

Attention

- In the framework of RNNs, recall that the output, $\boldsymbol{y}_n$, at time n is given in terms of the state values at the respective time, i.e., $\boldsymbol{h}_n$. In other words, the most recent information related to the whole past history (as this is encoded in the state vector) is used.
- However, it is rather unreasonable to assume that the long history of a sequence is sufficiently represented in the last state vector. For example in a sentence (and depending on the structure of a language), the next word may depend mostly on the meaning of a word that appeared some time earlier in the sequence of the words, or on a combination of words.

Attention

- In this context, one path to implement attention is to employ (in place of $\hat{\boldsymbol{y}}_n = g(V\boldsymbol{h}_n + \boldsymbol{c})$) the weighted sum

$$\boldsymbol{y}_n = g\left(\sum_{i=1}^{n} \alpha_{ni}\boldsymbol{h}_i\right),$$

  where g is a nonlinearity.

- In words, the output is left to be computed as a function of all the previous states; the weighting coefficients are learned during training. α_{ni} expresses the degree that the available information at time i affects the word at time n. In this way, the system learns to **attend** on the most important information in the available history.

Attention

- In this context, one path to implement attention is to employ (in place of $\hat{\boldsymbol{y}}_n = g(V\boldsymbol{h}_n + \boldsymbol{c})$) the weighted sum

$$\boldsymbol{y}_n = g\left(\sum_{i=1}^{n} \alpha_{ni}\boldsymbol{h}_i\right),$$

  where g is a nonlinearity.
- In words, the output is left to be computed as a function of all the previous states; the weighting coefficients are learned during training. α_{ni} expresses the degree that the available information at time i affects the word at time n. In this way, the system learns to **attend** on the most important information in the available history.

Attention

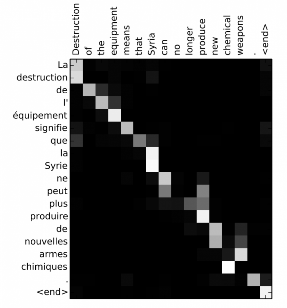
Example 1:

- An interesting aspect of the previous model is that one can follow what the model in doing and, thus, exploit it accordingly.

Attention

Example 1:

- An interesting aspect of the previous model is that one can follow what the model in doing and, thus, exploit it accordingly.
- An example from [Badhanau D., et.al., 2016]. While translating from French to English, the network attends sequentially to each input state; however from time to time, the network attends to two words at time to produce an output.



Attention

Example 2:

- In [Xu K., et. al., 2016], the attention mechanism is applied to generate image descriptions. First, a variant of a CNN is used to encode the original image in terms of a set of feature vectors, $\boldsymbol{a}_i, \ i = 1, 2, \ldots, L$. Each feature vector corresponds to a different region in the original image.
- Each one of these feature vectors is associated with an attention related weight, α_i, and it is provided as an input to an RNN.
- The network is trained so that to compute the output word probability given the state of the LSTM.

Attention

Example 2:

- In [Xu K., et. al., 2016], the attention mechanism is applied to generate image descriptions. First, a variant of a CNN is used to encode the original image in terms of a set of feature vectors, $\boldsymbol{a}_i,\ i = 1, 2, \ldots, L$. Each feature vector corresponds to a different region in the original image.
- Each one of these feature vectors is associated with an attention related weight, α_i, and it is provided as an input to an RNN.
- The network is trained so that to compute the output word probability given the state of the LSTM.

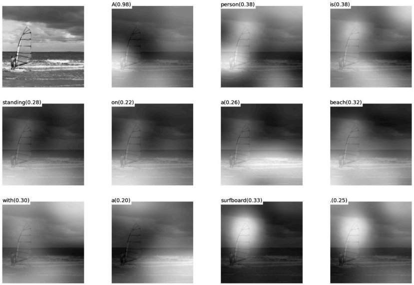
Attention

Example 2:

- In [Xu K., et. al., 2016], the attention mechanism is applied to generate image descriptions. First, a variant of a CNN is used to encode the original image in terms of a set of feature vectors, $\boldsymbol{a}_i,\ i = 1, 2, \ldots, L$. Each feature vector corresponds to a different region in the original image.
- Each one of these feature vectors is associated with an attention related weight, α_i, and it is provided as an input to an RNN.
- The network is trained so that to compute the output word probability given the state of the LSTM.

Attention

- By visualizing the attention weights, as in the previous example, one can interpret what the model is focusing at while generating a word:



(b) A person is standing on a beach with a surfboard.

Adversarial Examples

- Having built and trained networks that achieve accuracies, sometimes, close to what humans achieve, does it mean that the networks truly **understand** what they have learned?
- The answer is NO, in spite of the fact that they can predict with very high accuracies data in the test set.
- It turns out that, one can easily construct examples, by slightly perturbing input data in a specific way, which can consistently fool the network with high probability. Such examples are known as adversarial examples.
- Moreover, the difference between the adversarial examples and the original patterns, from which they are generated, is hardly perceptible. No human could ever classify them as being different.

Adversarial Examples

- Having built and trained networks that achieve accuracies, sometimes, close to what humans achieve, does it mean that the networks truly **understand** what they have learned?
- The answer is NO, in spite of the fact that they can predict with very high accuracies data in the test set.
- It turns out that, one can easily construct examples, by slightly perturbing input data in a specific way, which can consistently fool the network with high probability. Such examples are known as adversarial examples.
- Moreover, the difference between the adversarial examples and the original patterns, from which they are generated, is hardly perceptible. No human could ever classify them as being different.

Adversarial Examples

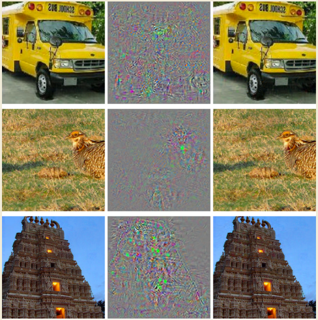
- Having built and trained networks that achieve accuracies, sometimes, close to what humans achieve, does it mean that the networks truly **understand** what they have learned?
- The answer is NO, in spite of the fact that they can predict with very high accuracies data in the test set.
- It turns out that, one can easily construct examples, by slightly perturbing input data in a specific way, which can consistently fool the network with high probability. Such examples are known as adversarial examples.
- Moreover, the difference between the adversarial examples and the original patterns, from which they are generated, is hardly perceptible. No human could ever classify them as being different.

Adversarial Examples

- Having built and trained networks that achieve accuracies, sometimes, close to what humans achieve, does it mean that the networks truly **understand** what they have learned?
- The answer is NO, in spite of the fact that they can predict with very high accuracies data in the test set.
- It turns out that, one can easily construct examples, by slightly perturbing input data in a specific way, which can consistently fool the network with high probability. Such examples are known as adversarial examples.
- Moreover, the difference between the adversarial examples and the original patterns, from which they are generated, is hardly perceptible. No human could ever classify them as being different.

Adversarial Examples

- Examples taken from [Szegedy, et. al., 2014]. Adversarial examples generated for AlexNet. All images in the right column are predicted to be an "ostrich, Struthio camelus"!!!



Adversarial Examples

How adversarial examples are generated?: Some examples.

- In [Szegedy, et. al., 2014], for a given input data point $\boldsymbol{x}$, one solves an optimization task, that finds the minimum norm perturbation, $\boldsymbol{v}$, such as the label of $\boldsymbol{x}$ and the label of $\boldsymbol{x} + \boldsymbol{v}$ to be different.
- In [Goodfellow, et.al., 2015], the perturbation is made in the direction of the sign of the gradient of the cost function w.r. to an input point, i.e., $\boldsymbol{v} = \epsilon \text{sign}\left\{\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y})\right\}$, $\epsilon > 0$.
- In [Moosavi-Dezfooli, et.al. 2016], an optimization method is used to compute a single-universal minimum norm perturbation that fools all images in the input data base with high probability.
- Adversarial examples are highly uncommon to be found in the input data bases, and it seems that they reside in regions of very low probability and are, somehow, hard to find by randomly sampling around an input point.

Adversarial Examples

How adversarial examples are generated?: Some examples.

- In [Szegedy, et. al., 2014], for a given input data point $\boldsymbol{x}$, one solves an optimization task, that finds the minimum norm perturbation, $\boldsymbol{v}$, such as the label of $\boldsymbol{x}$ and the label of $\boldsymbol{x}+\boldsymbol{v}$ to be different.
- In [Goodfellow, et.al., 2015], the perturbation is made in the direction of the sign of the gradient of the cost function w.r. to an input point, i.e., $\boldsymbol{v} = \epsilon \text{sign}\left\{\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y})\right\}$, $\epsilon > 0$.
- In [Moosavi-Dezfooli, et.al. 2016], an optimization method is used to compute a single-universal minimum norm perturbation that fools all images in the input data base with high probability.
- Adversarial examples are highly uncommon to be found in the input data bases, and it seems that they reside in regions of very low probability and are, somehow, hard to find by randomly sampling around an input point.

Adversarial Examples

How adversarial examples are generated?: Some examples.

- In [Szegedy, et. al., 2014], for a given input data point $\boldsymbol{x}$, one solves an optimization task, that finds the minimum norm perturbation, $\boldsymbol{v}$, such as the label of $\boldsymbol{x}$ and the label of $\boldsymbol{x}+\boldsymbol{v}$ to be different.
- In [Goodfellow, et.al., 2015], the perturbation is made in the direction of the sign of the gradient of the cost function w.r. to an input point, i.e., $\boldsymbol{v} = \epsilon \text{sign}\left\{\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y})\right\}$, $\epsilon > 0$.
- In [Moosavi-Dezfooli, et.al. 2016], an optimization method is used to compute a single-universal minimum norm perturbation that fools all images in the input data base with high probability.
- Adversarial examples are highly uncommon to be found in the input data bases, and it seems that they reside in regions of very low probability and are, somehow, hard to find by randomly sampling around an input point.

Adversarial Examples

How adversarial examples are generated?: Some examples.

- In [Szegedy, et. al., 2014], for a given input data point $\boldsymbol{x}$, one solves an optimization task, that finds the minimum norm perturbation, $\boldsymbol{v}$, such as the label of $\boldsymbol{x}$ and the label of $\boldsymbol{x}+\boldsymbol{v}$ to be different.
- In [Goodfellow, et.al., 2015], the perturbation is made in the direction of the sign of the gradient of the cost function w.r. to an input point, i.e., $\boldsymbol{v} = \epsilon \text{sign}\left\{\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y})\right\}$, $\epsilon > 0$.
- In [Moosavi-Dezfooli, et.al. 2016], an optimization method is used to compute a single-universal minimum norm perturbation that fools all images in the input data base with high probability.
- Adversarial examples are highly uncommon to be found in the input data bases, and it seems that they reside in regions of very low probability and are, somehow, hard to find by randomly sampling around an input point.

Adversarial Examples

How adversarial examples phenomenon is justified?: Some attempts.

- It seems that at the heart of the problem lies the high dimensionality of the input space.

Adversarial Examples

How adversarial examples phenomenon is justified?: Some attempts.

- It seems that at the heart of the problem lies the high dimensionality of the input space.

Adversarial Examples

How adversarial examples phenomenon is justified?: Some attempts.

- It seems that at the heart of the problem lies the high dimensionality of the input space.
- In [Goodfellow, et.al., 2015], the phenomenon is highlighted around the involved linear operations. Take as an example a linear classifier, $\boldsymbol{\theta}$, and two points, the original one, $\boldsymbol{x}$ and its perturbed adversarial version $\boldsymbol{x}^{'} = \boldsymbol{x} + \boldsymbol{v}$. Let $\boldsymbol{v} = \epsilon \text{sign}\{\boldsymbol{\theta}\},\ \epsilon > 0$.

Adversarial Examples

How adversarial examples phenomenon is justified?: Some attempts.

- It seems that at the heart of the problem lies the high dimensionality of the input space.
- In [Goodfellow, et.al., 2015], the phenomenon is highlighted around the involved linear operations. Take as an example a linear classifier, $\boldsymbol{\theta}$, and two points, the original one, $\boldsymbol{x}$ and its perturbed adversarial version $\boldsymbol{x}^{'} = \boldsymbol{x} + \boldsymbol{v}$. Let $\boldsymbol{v} = \epsilon \text{sign}\left\{\boldsymbol{\theta}\right\}, \ \epsilon > 0$.
- Then,

$$\boldsymbol{\theta}^T \boldsymbol{x}^{'} = \boldsymbol{\theta}^T \boldsymbol{x} + \epsilon \sum_{i=1}^{l} |\theta_i|.$$

Adversarial Examples

How adversarial examples phenomenon is justified?: Some attempts.

- It seems that at the heart of the problem lies the high dimensionality of the input space.
- In [Goodfellow, et.al., 2015], the phenomenon is highlighted around the involved linear operations. Take as an example a linear classifier, $\boldsymbol{\theta}$, and two points, the original one, $\boldsymbol{x}$ and its perturbed adversarial version $\boldsymbol{x}^{'} = \boldsymbol{x} + \boldsymbol{v}$. Let $\boldsymbol{v} = \epsilon \text{sign}\left\{\boldsymbol{\theta}\right\}, \ \epsilon > 0$.
- Then,

$$\boldsymbol{\theta}^T \boldsymbol{x}^{'} = \boldsymbol{\theta}^T \boldsymbol{x} + \epsilon \sum_{i=1}^{l} |\theta_i|.$$

- Note that even if ϵ can be very small, if l is very large then the terms $\boldsymbol{\theta}^T \boldsymbol{x}^{'}$ and $\boldsymbol{\theta}^T \boldsymbol{x}$ can be very different.

Adversarial Examples

How adversarial examples phenomenon is justified?: Some attempts.

- It seems that at the heart of the problem lies the high dimensionality of the input space.
- In [Goodfellow, et.al., 2015], the phenomenon is highlighted around the involved linear operations. Take as an example a linear classifier, $\boldsymbol{\theta}$, and two points, the original one, $\boldsymbol{x}$ and its perturbed adversarial version $\boldsymbol{x}^{'} = \boldsymbol{x} + \boldsymbol{v}$. Let $\boldsymbol{v} = \epsilon \text{sign}\{\boldsymbol{\theta}\}, \ \epsilon > 0$.
- Then,

$$\boldsymbol{\theta}^T \boldsymbol{x}^{'} = \boldsymbol{\theta}^T \boldsymbol{x} + \epsilon \sum_{i=1}^{l} |\theta_i|.$$

- Note that even if ϵ can be very small, if l is very large then the terms $\boldsymbol{\theta}^T \boldsymbol{x}^{'}$ and $\boldsymbol{\theta}^T \boldsymbol{x}$ can be very different.
- Of course, NN are not linear classifiers but linear operations are involved as, for example, in ReLU. Also, if sigmoid activations are involved, an effort is made to operate in their linear regions.

Adversarial Examples

How the adversarial examples phenomenon is justified?: Some attempts.

- In [Fawzi, et. al., 2016], a more theoretically pleasing explanation is provided by showing that adversarial examples have distinct geometric characteristics, in the input space, compared to completely random perturbations. The former, in connection with some underlying geometric properties of the decision surface, are causing the problem.
- This is a hot topic and more results are expecting to appear in the near future.

Adversarial Examples

How the adversarial examples phenomenon is justified?: Some attempts.

- In [Fawzi, et. al., 2016], a more theoretically pleasing explanation is provided by showing that adversarial examples have distinct geometric characteristics, in the input space, compared to completely random perturbations. The former, in connection with some underlying geometric properties of the decision surface, are causing the problem.
- This is a hot topic and more results are expecting to appear in the near future.

Adversarial Examples

Adversarial training: Some techniques.

- To robustify a network against adversarial examples a number of techniques have already been suggested.
- One way is to involve adversarial examples in the training set during the training phase. This is equivalent to regularization via artificially extending the data set ([Szegedy, et. al., 2014])
- Another path is to modify the loss function for taking special care of the adversarial examples. For example, in [Goodfellow, et.al. 2015], it is suggest to use

$$J^{'}(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y}) = \alpha J(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y}) + (1-\alpha) J(\boldsymbol{\theta}, \boldsymbol{x} + \Delta_{\boldsymbol{x}}, \boldsymbol{y}),$$

where

$$\Delta_{\boldsymbol{x}} := \epsilon \text{sign} \left\{ \nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y}) \right\}.$$

Adversarial Examples

Adversarial training: Some techniques.

- To robustify a network against adversarial examples a number of techniques have already been suggested.
- One way is to involve adversarial examples in the training set during the training phase. This is equivalent to regularization via artificially extending the data set ([Szegedy, et. al., 2014])
- Another path is to modify the loss function for taking special care of the adversarial examples. For example, in [Goodfellow, et.al. 2015], it is suggest to use

$$J^{'}(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y}) = \alpha J(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y}) + (1-\alpha) J(\boldsymbol{\theta}, \boldsymbol{x} + \Delta_{\boldsymbol{x}}, \boldsymbol{y}),$$

where

$$\Delta_x := \epsilon \text{sign} \left\{ \nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y}) \right\}.$$

Adversarial Examples

Adversarial training: Some techniques.

- To robustify a network against adversarial examples a number of techniques have already been suggested.
- One way is to involve adversarial examples in the training set during the training phase. This is equivalent to regularization via artificially extending the data set ([Szegedy, et. al., 2014])
- Another path is to modify the loss function for taking special care of the adversarial examples. For example, in [Goodfellow, et.al. 2015], it is suggest to use

$$J^{'}(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y}) = \alpha J(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y}) + (1-\alpha) J(\boldsymbol{\theta}, \boldsymbol{x} + \Delta_{\boldsymbol{x}}, \boldsymbol{y}),$$

where

$$\Delta_{\boldsymbol{x}} := \epsilon \text{sign} \left\{ \nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, \boldsymbol{y}) \right\}.$$

Adversarial Examples

Adversarial training: Some techniques.

- Another way to look at adversarial examples is that they violate the smoothness condition. That is, we expect that, for small enough $\epsilon > 0$, the input patterns $\boldsymbol{x}$ and $\boldsymbol{x} + \boldsymbol{v}$, for any $\boldsymbol{v} : \ \|\boldsymbol{v}\| \leq \epsilon$, to have the same label, with high probability.
- In this vein, in [Miyato, et.al., 2016], a regularizer is used that rewards smoothness of the model distribution w.r. to the input around every input data point.
- In [Shaham, et. al. 2016] a robust optimization method is proposed , which builds around a min-max formulation, where the cost function is optimized w.r. to a worst-case realization of a perturbation.
- This is also a new field and many more results is anticipated to come up in the near future.

Adversarial Examples

Adversarial training: Some techniques.

- Another way to look at adversarial examples is that they violate the smoothness condition. That is, we expect that, for small enough $\epsilon > 0$, the input patterns $\boldsymbol{x}$ and $\boldsymbol{x} + \boldsymbol{v}$, for any $\boldsymbol{v}: \ \|\boldsymbol{v}\| \leq \epsilon$, to have the same label, with high probability.
- In this vein, in [Miyato, et.al., 2016], a regularizer is used that rewards smoothness of the model distribution w.r. to the input around every input data point.
- In [Shaham, et. al. 2016] a robust optimization method is proposed , which builds around a min-max formulation, where the cost function is optimized w.r. to a worst-case realization of a perturbation.
- This is also a new field and many more results is anticipated to come up in the near future.

Adversarial Examples

Adversarial training: Some techniques.

- Another way to look at adversarial examples is that they violate the smoothness condition. That is, we expect that, for small enough $\epsilon > 0$, the input patterns $\boldsymbol{x}$ and $\boldsymbol{x} + \boldsymbol{v}$, for any $\boldsymbol{v} : \|\boldsymbol{v}\| \leq \epsilon$, to have the same label, with high probability.
- In this vein, in [Miyato, et.al., 2016], a regularizer is used that rewards smoothness of the model distribution w.r. to the input around every input data point.
- In [Shaham, et. al. 2016] a robust optimization method is proposed , which builds around a min-max formulation, where the cost function is optimized w.r. to a worst-case realization of a perturbation.
- This is also a new field and many more results is anticipated to come up in the near future.

Adversarial Examples

Adversarial training: Some techniques.

- Another way to look at adversarial examples is that they violate the smoothness condition. That is, we expect that, for small enough $\epsilon > 0$, the input patterns $\boldsymbol{x}$ and $\boldsymbol{x} + \boldsymbol{v}$, for any $\boldsymbol{v}: \ \|\boldsymbol{v}\| \leq \epsilon$, to have the same label, with high probability.
- In this vein, in [Miyato, et.al., 2016], a regularizer is used that rewards smoothness of the model distribution w.r. to the input around every input data point.
- In [Shaham, et. al. 2016] a robust optimization method is proposed , which builds around a min-max formulation, where the cost function is optimized w.r. to a worst-case realization of a perturbation.
- This is also a new field and many more results is anticipated to come up in the near future.

Deep Generative Models

- So far, we have been involved with supervised learning techniques of various types of multilayer networks. Our focus now turns on unsupervised learning of deep networks.
- The goal of such networks is to grasp regularities and structure hidden in the input data. In this way, one can achieve their efficient representation in terms of a layer-wise **feature generation** via the use of unlabelled data only.
- In this way, one can use the learned representation of the input in various subsequent supervised learning tasks. Such techniques can be useful in what is known as transfer learning or multitask learning.
- Furthermore, supervised learning techniques of deep networks require a large number of labelled data. In some cases, this may not be possible and the use of unlabelled data can facilitate the learning of the input representation, which can then be exploited in a subsequent supervised learning tasks.

Deep Generative Models

- So far, we have been involved with supervised learning techniques of various types of multilayer networks. Our focus now turns on unsupervised learning of deep networks.
- The goal of such networks is to grasp regularities and structure hidden in the input data. In this way, one can achieve their efficient representation in terms of a layer-wise **feature generation** via the use of unlabelled data only.
- In this way, one can use the learned representation of the input in various subsequent supervised learning tasks. Such techniques can be useful in what is known as transfer learning or multitask learning.
- Furthermore, supervised learning techniques of deep networks require a large number of labelled data. In some cases, this may not be possible and the use of unlabelled data can facilitate the learning of the input representation, which can then be exploited in a subsequent supervised learning tasks.

Deep Generative Models

- So far, we have been involved with supervised learning techniques of various types of multilayer networks. Our focus now turns on unsupervised learning of deep networks.
- The goal of such networks is to grasp regularities and structure hidden in the input data. In this way, one can achieve their efficient representation in terms of a layer-wise **feature generation** via the use of unlabelled data only.
- In this way, one can use the learned representation of the input in various subsequent supervised learning tasks. Such techniques can be useful in what is known as transfer learning or multitask learning.
- Furthermore, supervised learning techniques of deep networks require a large number of labelled data. In some cases, this may not be possible and the use of unlabelled data can facilitate the learning of the input representation, which can then be exploited in a subsequent supervised learning tasks.

Deep Generative Models

- So far, we have been involved with supervised learning techniques of various types of multilayer networks. Our focus now turns on unsupervised learning of deep networks.
- The goal of such networks is to grasp regularities and structure hidden in the input data. In this way, one can achieve their efficient representation in terms of a layer-wise **feature generation** via the use of unlabelled data only.
- In this way, one can use the learned representation of the input in various subsequent supervised learning tasks. Such techniques can be useful in what is known as transfer learning or multitask learning.
- Furthermore, supervised learning techniques of deep networks require a large number of labelled data. In some cases, this may not be possible and the use of unlabelled data can facilitate the learning of the input representation, which can then be exploited in a subsequent supervised learning tasks.

Deep Generative Models

- Deep generative models can be used to artificially generate input data, which is a form of "regularization" by expanding the training data set. This leads to a reduction of overfitting by artificially increasing the number of training points w.r to the number of unknown parameters to be estimated.
- Deep generative networks are inspired by the notion of probabilistic graphical models, which have been dealt in Chapters 15 and 16.

Deep Generative Models

- Deep generative models can be used to artificially generate input data, which is a form of "regularization" by expanding the training data set. This leads to a reduction of overfitting by artificially increasing the number of training points w.r to the number of unknown parameters to be estimated.
- Deep generative networks are inspired by the notion of probabilistic graphical models, which have been dealt in Chapters 15 and 16.

Deep Generative Models

- When the training set is very small and supervised learning techniques cannot be employed, the hidden layers can be thought of as part of a deep generative model, which builds the equivalent representation of the input. Unsupervised training in a layer-wise greedy-type approach can then be used as a **pre-training** phase. The obtained values of the parameters can be used as initial values of a subsequent supervised fine tuning of the parameters

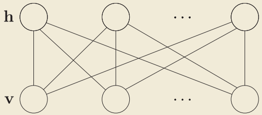
- As a matter of fact, such techniques led to the revival of deep networks, although they are rarely used these days.

Deep Generative Models

- When the training set is very small and supervised learning techniques cannot be employed, the hidden layers can be thought of as part of a deep generative model, which builds the equivalent representation of the input. Unsupervised training in a layer-wise greedy-type approach can then be used as a **pre-training** phase. The obtained values of the parameters can be used as initial values of a subsequent supervised fine tuning of the parameters
- As a matter of fact, such techniques led to the revival of deep networks, although they are rarely used these days.

Restricted Boltzmann Machines

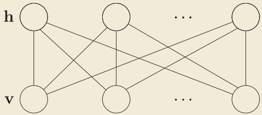
- A Restricted Boltzmann Machine (RBM) is a special type of the more general class of Boltzmann Machines (BM). The figure below shows the probabilistic graphical model corresponding to an RBM.



- It is an undirected graphical model with no connections among nodes of the same layer. Moreover, the upper level comprises nodes corresponding to hidden variables and the lower level consists of visible nodes. That is, observations are applied to the nodes of the lower layer only.

Restricted Boltzmann Machines

- A Restricted Boltzmann Machine (RBM) is a special type of the more general class of Boltzmann Machines (BM). The figure below shows the probabilistic graphical model corresponding to an RBM.



- It is an undirected graphical model with no connections among nodes of the same layer. Moreover, the upper level comprises nodes corresponding to hidden variables and the lower level consists of visible nodes. That is, observations are applied to the nodes of the lower layer only.

Restricted Boltzmann Machines

- Following the general definition of a Boltzmann machine, the joint distribution of the involved random variables is of the form,

$$P(v_1, \ldots, v_J, h_1, \ldots, h_I) = \frac{1}{Z} \exp\big(-E(\boldsymbol{v}, \boldsymbol{h})\big),$$

where different symbols for the J visible $(\mathrm{v}_j,\ j = 1, \ldots, J)$ and the I hidden variables $(\mathrm{h}_i,\ i = 1, \ldots, I)$ have been used. E is the energy of the system, which is defined next.

- The energy is defined in terms of a set of unknown parameters, θ,

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\sum_{i=1}^{I}\sum_{j=1}^{J} \theta_{ij} h_i v_j - \sum_{i=1}^{I} b_i h_i - \sum_{j=1}^{J} c_j v_j, \qquad (9)$$

where b_i and c_j are the bias terms for the hidden and visible nodes, respectively. The normalizing constant is obtained as,

$$Z = \sum_{\boldsymbol{v}}\sum_{\boldsymbol{h}} \exp\big(-E(\boldsymbol{v}, \boldsymbol{h})\big).$$

Restricted Boltzmann Machines

- Following the general definition of a Boltzmann machine, the joint distribution of the involved random variables is of the form,

$$P(v_1, \dots, v_J, h_1, \dots, h_I) = \frac{1}{Z} \exp\big(-E(\boldsymbol{v}, \boldsymbol{h})\big),$$

where different symbols for the J visible $(\mathrm{v}_j,\ j = 1, \dots, J)$ and the I hidden variables $(\mathrm{h}_i,\ i = 1, \dots, I)$ have been used. E is the energy of the system, which is defined next.

- The energy is defined in terms of a set of unknown parameters, θ,

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\sum_{i=1}^{I} \sum_{j=1}^{J} \theta_{ij} h_i v_j - \sum_{i=1}^{I} b_i h_i - \sum_{j=1}^{J} c_j v_j, \tag{9}$$

where b_i and c_j are the bias terms for the hidden and visible nodes, respectively. The normalizing constant is obtained as,

$$Z = \sum_{\boldsymbol{v}} \sum_{\boldsymbol{h}} \exp\big(-E(\boldsymbol{v}, \boldsymbol{h})\big).$$

Restricted Boltzmann Machines

- The goal now is to derive a scheme for training an RBM; that is, to learn the set of the unknown parameters, $\theta_{ij},\ b_i,\ c_j$, which will be collectively denoted as Θ, $\boldsymbol{b}$ and $\boldsymbol{c}$, respectively.
- We will focus on discrete variables, hence the involved distributions are probabilities. More specifically, we will focus on variables of a binary nature, i.e., $\mathrm{v}_j,\ \mathrm{h}_i \in \{0,1\},\ \ j = 1,\ldots,J$, $i = 1,\ldots,I$.
- The goal of learning is to maximize the log-likelihood, using N observations of the visible variables, denoted as $\boldsymbol{v}_n$, $n = 1,\ldots,N$, where

$$\boldsymbol{v}_n := [v_{1n},\ldots,v_{Jn}]^T,$$

is the vector of the corresponding observations at time n. We will say that the visible nodes are clamped on the respective observations.

Restricted Boltzmann Machines

- The goal now is to derive a scheme for training an RBM; that is, to learn the set of the unknown parameters, $\theta_{ij},\ b_i,\ c_j$, which will be collectively denoted as Θ, $\boldsymbol{b}$ and $\boldsymbol{c}$, respectively.
- We will focus on discrete variables, hence the involved distributions are probabilities. More specifically, we will focus on variables of a binary nature, i.e., $\mathrm{v}_j,\ \mathrm{h}_i \in \{0,1\},\ \ j=1,\ldots,J$, $i=1,\ldots,I$.
- The goal of learning is to maximize the log-likelihood, using N observations of the visible variables, denoted as $\boldsymbol{v}_n$, $n=1,\ldots,N$, where

$$\boldsymbol{v}_n := [v_{1n},\ldots,v_{Jn}]^T,$$

is the vector of the corresponding observations at time n. We will say that the visible nodes are clamped on the respective observations.

Restricted Boltzmann Machines

- The goal now is to derive a scheme for training an RBM; that is, to learn the set of the unknown parameters, $\theta_{ij},\ b_i,\ c_j$, which will be collectively denoted as Θ, $\boldsymbol{b}$ and $\boldsymbol{c}$, respectively.
- We will focus on discrete variables, hence the involved distributions are probabilities. More specifically, we will focus on variables of a binary nature, i.e., $\mathrm{v}_j,\ \mathrm{h}_i \in \{0,1\},\ \ j=1,\ldots,J$, $i=1,\ldots,I$.
- The goal of learning is to maximize the log-likelihood, using N observations of the visible variables, denoted as $\boldsymbol{v}_n$, $n=1,\ldots,N$, where

$$\boldsymbol{v}_n := [v_{1n},\ldots,v_{Jn}]^T,$$

  is the vector of the corresponding observations at time n. We will say that the visible nodes are clamped on the respective observations.

Contrastive Divergence

- In order to train an RBM, one has to compute the normalizing constant, Z, which turns out to be a computationally intractable task. A way to bypass it to approach it is via Gibbs sampling techniques.
- Contrastive divergence (CD): The idea behind this method is to generate the missing samples of the hidden variables via Gibbs sampling, starting the chain from the observations available for the visible nodes. The most important feature is that, in practice, only a few iterations of the chain are sufficient.
- Theoretically, the CD method can be justified by relying on an approximation of the the maximum likelihood loss function as a difference of two Kullback-Leibler divergences. It can also be conceived as a stochastic approximation attempt, where expectations are replaced by samples, which are generated by the Gibbs sampling.

Contrastive Divergence

- In order to train an RBM, one has to compute the normalizing constant, Z, which turns out to be a computationally intractable task. A way to bypass it to approach it is via Gibbs sampling techniques.
- Contrastive divergence (CD): The idea behind this method is to generate the missing samples of the hidden variables via Gibbs sampling, starting the chain from the observations available for the visible nodes. The most important feature is that, in practice, only a few iterations of the chain are sufficient.
- Theoretically, the CD method can be justified by relying on an approximation of the the maximum likelihood loss function as a difference of two Kullback-Leibler divergences. It can also be conceived as a stochastic approximation attempt, where expectations are replaced by samples, which are generated by the Gibbs sampling.

Contrastive Divergence

- In order to train an RBM, one has to compute the normalizing constant, Z, which turns out to be a computationally intractable task. A way to bypass it to approach it is via Gibbs sampling techniques.
- Contrastive divergence (CD): The idea behind this method is to generate the missing samples of the hidden variables via Gibbs sampling, starting the chain from the observations available for the visible nodes. The most important feature is that, in practice, only a few iterations of the chain are sufficient.
- Theoretically, the CD method can be justified by relying on an approximation of the the maximum likelihood loss function as a difference of two Kullback-Leibler divergences. It can also be conceived as a stochastic approximation attempt, where expectations are replaced by samples, which are generated by the Gibbs sampling.

- Following this rationale, a first primitive version of this algorithmic scheme can be cast as:
  - Step 1: Start the Gibbs sampler at $\boldsymbol{v}^{(1)} := \boldsymbol{v}_n$, i.e., observations, and generate samples for the hidden variables as,
  $$\boldsymbol{h}^{(1)} \sim P(\boldsymbol{h}|\boldsymbol{v}^{(1)}).$$
  - Step 2: Use $\boldsymbol{h}^{(1)}$ to generate samples for the visible nodes,
  $$\boldsymbol{v}^{(2)} \sim P(\boldsymbol{v}|\boldsymbol{h}^{(1)}).$$
  These are known as fantasy data.
  - Step 3: Use $\boldsymbol{v}^{(2)}$ to generate the next set of hidden variables,
  $$\boldsymbol{h}^{(2)} \sim P(\boldsymbol{h}|\boldsymbol{v}^{(2)}).$$
- The scheme based on these steps is known as CD-1, since only one up-down-up Gibbs sweep is used. If k such steps are employed, the resulting scheme is referred to as CD-k. Once the samples have been generated, the parameter update can be written as
$$\theta_{ij}(n) = \theta_{ij}(n-1) + \mu\left(h_i^{(1)} v_{jn} - h_i^{(2)} v_j^{(2)}\right).$$

- Following this rationale, a first primitive version of this algorithmic scheme can be cast as:
  - Step 1: Start the Gibbs sampler at $\boldsymbol{v}^{(1)} := \boldsymbol{v}_n$, i.e., observations, and generate samples for the hidden variables as,
  $$\boldsymbol{h}^{(1)} \sim P(\boldsymbol{h}|\boldsymbol{v}^{(1)}).$$
  - Step 2: Use $\boldsymbol{h}^{(1)}$ to generate samples for the visible nodes,
  $$\boldsymbol{v}^{(2)} \sim P(\boldsymbol{v}|\boldsymbol{h}^{(1)}).$$
  These are known as fantasy data.
  - Step 3: Use $\boldsymbol{v}^{(2)}$ to generate the next set of hidden variables,
  $$\boldsymbol{h}^{(2)} \sim P(\boldsymbol{h}|\boldsymbol{v}^{(2)}).$$
- The scheme based on these steps is known as CD-1, since only one up-down-up Gibbs sweep is used. If k such steps are employed, the resulting scheme is referred to as CD-k. Once the samples have been generated, the parameter update can be written as
$$\theta_{ij}(n) = \theta_{ij}(n-1) + \mu\left(h_i^{(1)}v_{jn} - h_i^{(2)}v_j^{(2)}\right).$$

Contrastive Divergence-The Algorithmic Steps

- Following this rationale, a first primitive version of this algorithmic scheme can be cast as:
  - Step 1: Start the Gibbs sampler at $\boldsymbol{v}^{(1)} := \boldsymbol{v}_n$, i.e., observations, and generate samples for the hidden variables as,

$$\boldsymbol{h}^{(1)} \sim P(\boldsymbol{h}|\boldsymbol{v}^{(1)}).$$

  - Step 2: Use $\boldsymbol{h}^{(1)}$ to generate samples for the visible nodes,

$$\boldsymbol{v}^{(2)} \sim P(\boldsymbol{v}|\boldsymbol{h}^{(1)}).$$

    These are known as fantasy data.
  - Step 3: Use $\boldsymbol{v}^{(2)}$ to generate the next set of hidden variables,

$$\boldsymbol{h}^{(2)} \sim P(\boldsymbol{h}|\boldsymbol{v}^{(2)}).$$

- The scheme based on these steps is known as CD-1, since only one up-down-up Gibbs sweep is used. If k such steps are employed, the resulting scheme is referred to as CD-k. Once the samples have been generated, the parameter update can be written as

$$\theta_{ij}(n) = \theta_{ij}(n-1) + \mu \left(h_i^{(1)} v_{jn} - h_i^{(2)} v_j^{(2)} \right).$$

- Following this rationale, a first primitive version of this algorithmic scheme can be cast as:
  - Step 1: Start the Gibbs sampler at $\boldsymbol{v}^{(1)} := \boldsymbol{v}_n$, i.e., observations, and generate samples for the hidden variables as,
  $$\boldsymbol{h}^{(1)} \sim P(\boldsymbol{h}|\boldsymbol{v}^{(1)}).$$
  - Step 2: Use $\boldsymbol{h}^{(1)}$ to generate samples for the visible nodes,
  $$\boldsymbol{v}^{(2)} \sim P(\boldsymbol{v}|\boldsymbol{h}^{(1)}).$$
  These are known as fantasy data.
  - Step 3: Use $\boldsymbol{v}^{(2)}$ to generate the next set of hidden variables,
  $$\boldsymbol{h}^{(2)} \sim P(\boldsymbol{h}|\boldsymbol{v}^{(2)}).$$
- The scheme based on these steps is known as CD-1, since only one up-down-up Gibbs sweep is used. If k such steps are employed, the resulting scheme is referred to as CD-k. Once the samples have been generated, the parameter update can be written as
$$\theta_{ij}(n) = \theta_{ij}(n-1) + \mu \left(h_i^{(1)} v_{jn} - h_i^{(2)} v_j^{(2)} \right).$$

Contrastive Divergence-The Algorithmic Steps

- A more refined scheme results if the estimates of the gradients are not obtained via a single observation sample, but they are instead averaged over a number of observations. The training input examples are presented in terms of mini-batches of length L. The previous reported steps are performed for each observation, but now the update is carried out only once per block of L samples, by averaging out the obtained estimates of the gradient, i.e.,

$$\theta_{ij}^{(t)} = \theta_{ij}^{(t-1)} + \frac{\mu}{L}\sum_{l=1}^{L} g_{ij}^{(l)}, \; i = 1, \ldots, I, \; j = 1, \ldots J, \qquad (10)$$

where

$$g_{ij}^{(l)} := h_i^{(1)} v_{j(l)} - h_i^{(2)} v_j^{(2)},$$

denotes the gradient approximation associated with the corresponding observation $v_{j(l)}$, $(l) \in \{1, 2, \ldots, N\}$, which is currently considered by the algorithm (and gives birth to the associated Gibbs samples).

Contrastive Divergence-The Algorithmic Steps

- A more refined scheme results if the estimates of the gradients are not obtained via a single observation sample, but they are instead averaged over a number of observations. The training input examples are presented in terms of mini-batches of length L. The previous reported steps are performed for each observation, but now the update is carried out only once per block of L samples, by averaging out the obtained estimates of the gradient, i.e.,

$$\theta_{ij}^{(t)} = \theta_{ij}^{(t-1)} + \frac{\mu}{L} \sum_{l=1}^{L} g_{ij}^{(l)}, \; i = 1, \ldots, I, \; j = 1, \ldots J, \tag{10}$$

where

$$g_{ij}^{(l)} := h_i^{(1)} v_{j(l)} - h_i^{(2)} v_j^{(2)},$$

denotes the gradient approximation associated with the corresponding observation $v_{j(l)}, \; (l) \in \{1, 2, \ldots, N\}$, which is currently considered by the algorithm (and gives birth to the associated Gibbs samples).

- Recursion (10) can be written in a more compact form as

$$\Theta^{(t)} = \Theta^{(t-1)} + \frac{\mu}{L} \sum_{l=1}^{L} G_{ij}^{(l)},$$

where

$$G_{ij}^{(l)} := \boldsymbol{h}^{(1)} \boldsymbol{v}_{(l)}^{T} - \boldsymbol{h}^{(2)} \boldsymbol{v}^{(2)T}.$$

Once all blocks have been considered, this corresponds to one epoch of training. The process continues for a number of successive epochs until a convergence criterion is met.

- Another version of the scheme results if we replace the obtained samples of the hidden variables with their respective mean values. This turns out to lead to estimates with lower variance. In our current context, where the variables are of a binary nature, it is readily seen that

$$\mathbb{E}[\mathrm{h}_i^{(1)}] = P\Big(h_i = 1 | v_{j(l)}\Big) = \mathrm{sigm}\Big(\sum_{j=1}^{J} \theta_{ij}(t-1) v_{j(l)} + b_i(t-1)\Big),$$

$$\mathbb{E}[\mathrm{h}_i^{(2)}] = P\Big(h_i = 1 | v_j^{(2)}\Big) = \mathrm{sigm}\Big(\sum_{j=1}^{J} \theta_{ij}(t-1) v_j^{(2)} + b_i(t-1)\Big).$$

Contrastive Divergence-The Algorithmic Steps

- Recursion (10) can be written in a more compact form as

$$\Theta^{(t)} = \Theta^{(t-1)} + \frac{\mu}{L}\sum_{l=1}^{L} G_{ij}^{(l)},$$

  where

$$G_{ij}^{(l)} := \boldsymbol{h}^{(1)}\boldsymbol{v}_{(l)}^T - \boldsymbol{h}^{(2)}\boldsymbol{v}^{(2)T}.$$

  Once all blocks have been considered, this corresponds to one epoch of training. The process continues for a number of successive epochs until a convergence criterion is met.

- Another version of the scheme results if we replace the obtained samples of the hidden variables with their respective mean values. This turns out to lead to estimates with lower variance. In our current context, where the variables are of a binary nature, it is readily seen that

$$\mathbb{E}[\mathrm{h}_i^{(1)}] = P\big(h_i = 1|v_{j(l)}\big) = \mathrm{sigm}\Big(\sum_{j=1}^{J}\theta_{ij}(t-1)v_{j(l)} + b_i(t-1)\Big),$$

$$\mathbb{E}[\mathrm{h}_i^{(2)}] = P\big(h_i = 1|v_j^{(2)}\big) = \mathrm{sigm}\Big(\sum_{j=1}^{J}\theta_{ij}(t-1)v_j^{(2)} + b_i(t-1)\Big).$$

Contrastive Divergence-The Algorithmic Steps

- In this case, the updates become

$$\Theta^{(t)} = \Theta^{(t-1)} + \frac{\mu}{L}\sum_{l=1}^{L} G_{ij}^{(l)},$$

$$G_{ij}^{(l)} := \mathbb{E}[\boldsymbol{h}^{(1)}]\boldsymbol{v}_{(l)}^T - \mathbb{E}[\boldsymbol{h}^{(2)}]\boldsymbol{v}^{(2)T}.$$

- The updates of the bias terms are derived in a similar way (one can also assume that there are fictitious extra nodes of a fixed value $+1$, and incorporate the bias terms in θ_{ij}) and we get

$$\boldsymbol{b}^{(t)} = \boldsymbol{b}^{(t-1)} + \frac{\mu}{L}\sum_{l=1}^{L} \boldsymbol{g}_b^{(l)},$$

$$\boldsymbol{g}_b^{(l)} := \mathbb{E}[\boldsymbol{h}^{(1)}] - \mathbb{E}[\boldsymbol{h}^{(2)}],$$

and

$$\boldsymbol{c}^{(t)} = \boldsymbol{c}^{(t-1)} + \frac{\mu}{L}\sum_{l=1}^{L} \boldsymbol{g}_c^{(l)},$$

$$\boldsymbol{g}_c^{(l)} := \boldsymbol{v}_{(l)} - \boldsymbol{v}^{(2)}.$$

- In this case, the updates become

$$\Theta^{(t)} = \Theta^{(t-1)} + \frac{\mu}{L}\sum_{l=1}^{L} G_{ij}^{(l)},$$

$$G_{ij}^{(l)} := \mathbb{E}[\boldsymbol{h}^{(1)}]\boldsymbol{v}_{(l)}^{T} - \mathbb{E}[\boldsymbol{h}^{(2)}]\boldsymbol{v}^{(2)T}.$$

- The updates of the bias terms are derived in a similar way (one can also assume that there are fictitious extra nodes of a fixed value $+1$, and incorporate the bias terms in θ_{ij}) and we get

$$\boldsymbol{b}^{(t)} = \boldsymbol{b}^{(t-1)} + \frac{\mu}{L}\sum_{l=1}^{L} \boldsymbol{g}_b^{(l)},$$

$$\boldsymbol{g}_b^{(l)} := \mathbb{E}[\boldsymbol{h}^{(1)}] - \mathbb{E}[\boldsymbol{h}^{(2)}],$$

  and

$$\boldsymbol{c}^{(t)} = \boldsymbol{c}^{(t-1)} + \frac{\mu}{L}\sum_{l=1}^{L} \boldsymbol{g}_c^{(l)},$$

$$\boldsymbol{g}_c^{(l)} := \boldsymbol{v}_{(l)} - \boldsymbol{v}^{(2)}.$$

The RBM Learning Algorithm via CD-1 for Binary Variables

- The resulting algorithm is summarized below:
  - Initialization
    - Initialize $\Theta(0)$, $\boldsymbol{b}(0)$, $\boldsymbol{c}(0)$, randomly.
  - **For** each epoch **DO**
    - **For** each block of size L **Do**
      - $G = O$, $\boldsymbol{g}_b = \boldsymbol{0}$, $\boldsymbol{g}_c = \boldsymbol{0}$; set gradients to zero.
      - **For** each $\boldsymbol{v}_n$ in the block **Do**
        * $\boldsymbol{h}^{(1)} \sim P(\boldsymbol{h}|\boldsymbol{v}_n)$
        * $\boldsymbol{v}^{(2)} \sim P(\boldsymbol{v}|\boldsymbol{h}^{(1)})$
        * $\boldsymbol{h}^{(2)} \sim P(\boldsymbol{h}|\boldsymbol{v}^{(2)})$
        * $G = G + \mathbb{E}[\boldsymbol{h}^{(1)}]\boldsymbol{v}_n^T - \mathbb{E}[\boldsymbol{h}^{(2)}]\boldsymbol{v}^{(2)}$
        * $\boldsymbol{g}_b = \boldsymbol{g}_b + \mathbb{E}[\boldsymbol{h}^{(1)}] - \mathbb{E}[\boldsymbol{h}^{(2)}]$
        * $\boldsymbol{g}_c = \boldsymbol{g}_c + \boldsymbol{v}_n - \boldsymbol{v}^{(2)}$
      - **End For**
      - $\Theta = \Theta + \frac{\mu}{L} G$
      - $\boldsymbol{b} = \boldsymbol{b} + \frac{\mu}{L} \boldsymbol{g}_b$
      - $\boldsymbol{c} = \boldsymbol{c} + \frac{\mu}{L} \boldsymbol{g}_c$
    - **End for**
    - If a convergence criterion is met, Stop
  - **End For**

An Example of Pre-training Deep Feedforward Networks

- We provide an example of how unsupervised pre-training can be used in the context of a supervised learning task.
- Such an approach enjoys a historical symbolism, since such a pretraing was first used and made it possible to train deep networks. This revived the subsequent interest in neural networks and led to their current "reign".
- In this context, given a set of training examples, $(y_n, \boldsymbol{x}_n),\ n = 1, 2, \ldots, N$, training a deep multilayer NN involves two major phases:
  1. pre-training
  2. supervised fine tuning.

An Example of Pre-training Deep Feedforward Networks

- We provide an example of how unsupervised pre-training can be used in the context of a supervised learning task.
- Such an approach enjoys a historical symbolism, since such a pretraing was first used and made it possible to train deep networks. This revived the subsequent interest in neural networks and led to their current "reign".
- In this context, given a set of training examples, $(y_n, \boldsymbol{x}_n),\ n = 1, 2, \ldots, N$, training a deep multilayer NN involves two major phases:
  1. pre-training
  2. supervised fine tuning.

An Example of Pre-training Deep Feedforward Networks

- We provide an example of how unsupervised pre-training can be used in the context of a supervised learning task.
- Such an approach enjoys a historical symbolism, since such a pretraing was first used and made it possible to train deep networks. This revived the subsequent interest in neural networks and led to their current "reign".
- In this context, given a set of training examples, $(y_n, \boldsymbol{x}_n),\ n = 1, 2, \ldots, N$, training a deep multilayer NN involves two major phases:
  1. pre-training
  2. supervised fine tuning.

An Example of Pre-training Deep Feedforward Networks

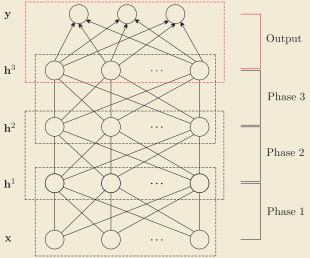
- Pre-training the weights, associated with hidden nodes, involves unsupervised learning via the RBM rationale. Assuming K hidden layers, $\mathbf{h}^k,\ k = 1, 2, \ldots, K$, we look at them in pairs, i.e., $(\mathbf{h}^{k-1}, \boldsymbol{h}^k),\ k = 1, 2, \ldots, K$, with $\mathbf{h}^0 := \mathbf{x}$, being the input layer.
- Each pair is treated as an RBM, in a hierarchical manner, with the outputs of the previous one becoming the inputs to the next (black boxes in the following figure):

An Example of Pre-training Deep Feedforward Networks

- Pre-training the weights, associated with hidden nodes, involves unsupervised learning via the RBM rationale. Assuming K hidden layers, $\mathbf{h}^k, \ k = 1, 2, \ldots, K$, we look at them in pairs, i.e., $(\mathbf{h}^{k-1}, \boldsymbol{h}^k), \ k = 1, 2, \ldots, K$, with $\mathbf{h}^0 := \mathbf{x}$, being the input layer.
- Each pair is treated as an RBM, in a hierarchical manner, with the outputs of the previous one becoming the inputs to the next (black boxes in the following figure):

An Example of Pre-training Deep Feedforward Networks

- Pre-training the weights, associated with hidden nodes, involves unsupervised learning via the RBM rationale. Assuming K hidden layers, $\mathbf{h}^k,\ k = 1, 2, \ldots, K$, we look at them in pairs, i.e., $(\mathbf{h}^{k-1}, \boldsymbol{h}^k),\ k = 1, 2, \ldots, K$, with $\mathbf{h}^0 := \mathbf{x}$, being the input layer.
- Each pair is treated as an RBM, in a hierarchical manner, with the outputs of the previous one becoming the inputs to the next (black boxes in the following figure):



Deep Feedforward Networks Pre-raining

- Pre-training of the weights leading to the output nodes (red box in previous figure) is performed via a supervised learning algorithm.
- To stress it out, the last hidden layer together with the output layer are **NOT** treated as an RBM, but as an one layer feed-forward network. In other words, the input to this supervised learning task are the features formed in the last hidden layer.

Deep Feedforward Networks Pre-raining

- Pre-training of the weights leading to the output nodes (red box in previous figure) is performed via a supervised learning algorithm.
- To stress it out, the last hidden layer together with the output layer are **NOT** treated as an RBM, but as an one layer feed-forward network. In other words, the input to this supervised learning task are the features formed in the last hidden layer.

Deep Feedforward Networks With Pre-training

- Finally, the fine tuning involves retraining in a typical backpropagation algorithm rationale, using the values obtained during pre-training for initialization. This is very important in getting a better feeling and understanding on how deep learning works. The label information is used in the hidden layers only at the fine tuning stage.
- During pre-training, the feature values in each layer grasp information related to the input distribution and the underlying regularities. The label information modifies the features at fine tuning stage. It does not participate in the process of discovering the features. Most of this part is left to the unsupervised phase, during pre-training.

Deep Feedforward Networks With Pre-training

- Finally, the fine tuning involves retraining in a typical backpropagation algorithm rationale, using the values obtained during pre-training for initialization. This is very important in getting a better feeling and understanding on how deep learning works. The label information is used in the hidden layers only at the fine tuning stage.
- During pre-training, the feature values in each layer grasp information related to the input distribution and the underlying regularities. The label information modifies the features at fine tuning stage. It does not participate in the process of discovering the features. Most of this part is left to the unsupervised phase, during pre-training.

Algorithm For Training Deep Feedforward Networks

- The methodology is summarized in the Algorithm given below:
  - Initialization.
    - Initialize randomly all the weights for the hidden nodes, Θ^k, $\boldsymbol{b}^k$, $\boldsymbol{c}^k$, $k = 1, 2, \ldots, K$.
    - Initialize randomly the weights leading to the output nodes.
    - Set $\boldsymbol{h}^0(n) := \boldsymbol{x}_n, \; n = 1, 2, \ldots, N$.

Phase I: Unsupervised Pre-training of Hidden Units

- **For** $k = 1, 2, \ldots, K$, **Do**;
  - Treat $\boldsymbol{h}^{k-1}$ as visible nodes and $\boldsymbol{h}^k$ as hidden nodes to an RBM.
  - Train the RBM with respect to Θ^k, $\boldsymbol{b}^k$, $\boldsymbol{c}^k$, via the RBM Algorithm.
  - Use the obtained values of the parameters to generate for each node in the layer, $\boldsymbol{h}^k$, N values, corresponding to the N observations.
    - Option 1:
      * $\boldsymbol{h}^k(n) \sim P(\boldsymbol{h}|\boldsymbol{h}^{k-1}(n)), \; n = 1, 2, \ldots, N$; Sample from the distribution.
    - Option 2:
      * $\boldsymbol{h}^k(n) = [P(h_1^k|\boldsymbol{h}^{k-1}(n), \ldots, P(h_{I_k}^k|\boldsymbol{h}^{k-1}(n)]^T, \; n = 1, 2, \ldots, N;$

    Propagate probabilities. I_k is the number of nodes in the layer.
- **End For**

Phase II: Supervised Pre-Training of Output Nodes

- Train the parameters of the pair $(\boldsymbol{h}^K, \boldsymbol{y})$, associated with the output layer, via any *supervised* learning algorithm. Treat $(y_n, \boldsymbol{h}^K(n)), n = 1, 2, \ldots, N$, as the training data.

Phase III: Fine-Tuning of All Nodes via Supervised Training

- Use the obtained values for all the parameters as initial values and train the whole network via the backpropagation, using $(y_n, \boldsymbol{x}_n), \; n = 1, 2, \ldots, N$ as training examples.

Deep Belief Networks

- So far, our focus was on an information flow in the feedforward or bottom-up direction. However, this is only part of the whole story.
- The other part concerns training generative models. The goal of such learning tasks is to "teach" the model to generate data. This is basically equivalent with learning probabilistic models that relate a set of variables, which can be observed, with another set of hidden ones.
- Deep networks have so far been viewed as models that form in layer by layer rationale features of features, i.e., more and more abstract representations of the input data are produced.
- The issue now becomes whether one can start from the last layer, and follow a top-down path with the new goal being that of generating data.

Deep Belief Networks

- So far, our focus was on an information flow in the feedforward or bottom-up direction. However, this is only part of the whole story.
- The other part concerns training generative models. The goal of such learning tasks is to "teach" the model to generate data. This is basically equivalent with learning probabilistic models that relate a set of variables, which can be observed, with another set of hidden ones.
- Deep networks have so far been viewed as models that form in layer by layer rationale features of features, i.e., more and more abstract representations of the input data are produced.
- The issue now becomes whether one can start from the last layer, and follow a top-down path with the new goal being that of generating data.

Deep Belief Networks

- So far, our focus was on an information flow in the feedforward or bottom-up direction. However, this is only part of the whole story.
- The other part concerns training generative models. The goal of such learning tasks is to "teach" the model to generate data. This is basically equivalent with learning probabilistic models that relate a set of variables, which can be observed, with another set of hidden ones.
- Deep networks have so far been viewed as models that form in layer by layer rationale features of features, i.e., more and more abstract representations of the input data are produced.
- The issue now becomes whether one can start from the last layer, and follow a top-down path with the new goal being that of generating data.

Deep Belief Networks

- So far, our focus was on an information flow in the feedforward or bottom-up direction. However, this is only part of the whole story.
- The other part concerns training generative models. The goal of such learning tasks is to "teach" the model to generate data. This is basically equivalent with learning probabilistic models that relate a set of variables, which can be observed, with another set of hidden ones.
- Deep networks have so far been viewed as models that form in layer by layer rationale features of features, i.e., more and more abstract representations of the input data are produced.
- The issue now becomes whether one can start from the last layer, and follow a top-down path with the new goal being that of generating data.

Deep Belief Networks

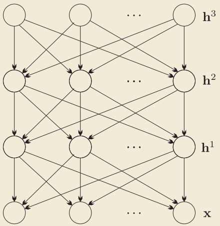
- Besides the need for artificially generating data in some practical applications, there is a further urge to look at this reverse direction of information flow.
- There are studies which suggest that such top-down connections exist in our visual system in order to generate lower level features of images starting from higher level representations. Such a mechanism can explain the creation of vivid imagery, dreaming as well as the disambiguating effect on the interpretation of local image regions by providing contextual prior information from previous frames.

Deep Belief Networks

- Besides the need for artificially generating data in some practical applications, there is a further urge to look at this reverse direction of information flow.
- There are studies which suggest that such top-down connections exist in our visual system in order to generate lower level features of images starting from higher level representations. Such a mechanism can explain the creation of vivid imagery, dreaming as well as the disambiguating effect on the interpretation of local image regions by providing contextual prior information from previous frames.

Deep Belief Networks

- A popular way to represent statistical generative models is via the use of probabilistic graphical models. A typical example of a generative model is that of sigmoidal networks. A sigmoidal network is illustrated in the figure below. It is a directed acyclic graph (Bayesian).



Deep Belief Networks

- Following the theory, the joint probability of the observed ($\boldsymbol{x}$) and hidden variables, distributed in K layers, is given by,

$$P(\boldsymbol{x}, \boldsymbol{h}^1, \ldots, \boldsymbol{h}^K) = P(\boldsymbol{x}|\boldsymbol{h}^1) \left(\prod_{k=1}^{K-1} P\big(\boldsymbol{h}^k|\boldsymbol{h}^{k+1}\big) \right) P(\boldsymbol{h}^K),$$

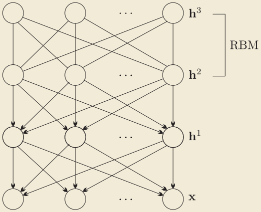
where the conditionals for each one of the I_k nodes of the kth layer are defined as,

$$P(h_i^k|\boldsymbol{h}^{k+1}) = \sigma\Big(\sum_{j=1}^{I_{k+1}} \theta_{ij}^{k+1} h_j^{k+1} \Big), \qquad k = 1, 2, \ldots, K-1,$$

$$i = 1, 2, \ldots, I_k.$$

Deep Belief Networks

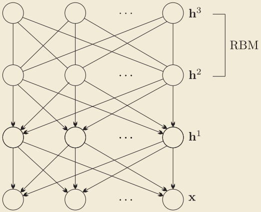
- A variant of the sigmoidal network was proposed is known as Deep Belief Network. The difference with a sigmoidal one is that the top two layers comprise an RBM. Thus, it is a mixed type of network consisting of both, directed as well as undirected edges, as shown below:



- The respective joint probability of all the involved variables is given by

$$P(\boldsymbol{x}, \boldsymbol{h}^1, \ldots, \boldsymbol{h}^K) = P(\boldsymbol{x}|\boldsymbol{h}^1) \left(\prod_{k=1}^{K-2} P(\boldsymbol{h}^k|\boldsymbol{h}^{k+1}) \right) P(\boldsymbol{h}^{K-1}, \boldsymbol{h}^K).$$

- A variant of the sigmoidal network was proposed is known as Deep Belief Network. The difference with a sigmoidal one is that the top two layers comprise an RBM. Thus, it is a mixed type of network consisting of both, directed as well as undirected edges, as shown below:



- The respective joint probability of all the involved variables is given by

$$P(\boldsymbol{x}, \boldsymbol{h}^1, \ldots, \boldsymbol{h}^K) = P(\boldsymbol{x}|\boldsymbol{h}^1) \left(\prod_{k=1}^{K-2} P\big(\boldsymbol{h}^k|\boldsymbol{h}^{k+1}\big) \right) P\big(\boldsymbol{h}^{K-1}, \boldsymbol{h}^K\big).$$

Deep Belief Networks

- It is known that learning Bayesian networks of relatively large size is intractable, due to the presence of converging edges.
- A way out is to employ the Algorithm for training RBMs. In other words, all hidden layers, starting form the input one, are treated as RBMs, and a greedy layer by layer pre-training bottom-up philosophy is adopted.

Deep Belief Networks

- It is known that learning Bayesian networks of relatively large size is intractable, due to the presence of converging edges.
- A way out is to employ the Algorithm for training RBMs. In other words, all hidden layers, starting form the input one, are treated as RBMs, and a greedy layer by layer pre-training bottom-up philosophy is adopted.

Deep Belief Networks

- Once the bottom-up pass has been completed, the estimated values of the unknown parameters are used for initializing another fine-tuning training algorithm, in place of the Phase III step of the Algorithm for training deep networks; however, this time the fine-tuning algorithm is an unsupervised one, since no labels are available.

- Such a scheme has been developed for training sigmoidal networks and it is known as wake-sleep algorithm. The scheme has a variational approximation flavor, and if initialized randomly takes a long time to converge. The objective behind the wake-sleep scheme is to adjust the weights during the top-down pass, so as to maximize the probability of the network to generate the observed data.

Deep Belief Networks

- Once the bottom-up pass has been completed, the estimated values of the unknown parameters are used for initializing another fine-tuning training algorithm, in place of the Phase III step of the Algorithm for training deep networks; however, this time the fine-tuning algorithm is an unsupervised one, since no labels are available.
- Such a scheme has been developed for training sigmoidal networks and it is known as wake-sleep algorithm. The scheme has a variational approximation flavor, and if initialized randomly takes a long time to converge. The objective behind the wake-sleep scheme is to adjust the weights during the top-down pass, so as to maximize the probability of the network to generate the observed data.

Generating Samples via a DBN

- Once training of the weights has been completed, data generation is achieved by the scheme summarized below:
  - Obtain samples $\boldsymbol{h}^{K-1}$, for the nodes at level $K-1$. This can be done via running a Gibbs chain, by alternating samples, $\boldsymbol{h}^{K} \sim P(\boldsymbol{h}|\boldsymbol{h}^{K-1})$ and $\boldsymbol{h}^{K-1} \sim P(\boldsymbol{h}|\boldsymbol{h}^{K})$. The convergence of the Gibbs chain can be speeded up by initializing the chain with a feature vector formed at the $K-1$ layer by one of the input patterns; this can be done by following a bottom-up pass to generate features in the hidden layers, as the one used during pre-training.
  - **For** $k = K-2, \ldots, 1$, **Do**; Top-down pass.
    - **For** $i = 1, 2, \ldots, I_k$, **Do**
      - $h_i^{k-1} \sim P\big(h_i|\boldsymbol{h}^k\big)$; Sample for each one of the nodes.
    - **End For**
  - **End For**
  - $\boldsymbol{x} = \boldsymbol{h}^0$; Generated pattern.

Example For Optical Character Recognition (OCR)

- This examples demonstrates the use of a deep network as a classifier in an OCR application. The characters (classes) which are involved are the Greek letters α, ν, o and τ, extracted from old historical documents. The respective class volumes are 1735, 1850, 2391 and 2264.

- Each binary image is converted to a binary feature vector by scanning it row-wise and concatenating the rows to form a $28 \times 28 = 784$ dimensional binary representation. In the sequel, 80% of the resulting patterns, per class, are randomly chosen to form the training set and the remaining patterns serve testing purposes. The class labels are represented by 4-digit binary codewords. For example, the first class (letter α) is given the binary code $[1\ 0\ 0\ 0\ 0]$, the second class is given the codeword $[0\ 1\ 0\ 0\ 0]$ and so on.
- Due to the binary nature of the patterns, the use of RBMs with binary stochastic units as the building blocks of a deep network is a natural choice.

Example For Optical Character Recognition (OCR)

- This examples demonstrates the use of a deep network as a classifier in an OCR application. The characters (classes) which are involved are the Greek letters α, ν, o and τ, extracted from old historical documents. The respective class volumes are 1735, 1850, 2391 and 2264.

- Each binary image is converted to a binary feature vector by scanning it row-wise and concatenating the rows to form a $28 \times 28 = 784$ dimensional binary representation. In the sequel, 80% of the resulting patterns, per class, are randomly chosen to form the training set and the remaining patterns serve testing purposes. The class labels are represented by 4-digit binary codewords. For example, the first class (letter α) is given the binary code $[1\ 0\ 0\ 0\ 0]$, the second class is given the codeword $[0\ 1\ 0\ 0\ 0]$ and so on.
- Due to the binary nature of the patterns, the use of RBMs with binary stochastic units as the building blocks of a deep network is a natural choice.

Example For Optical Character Recognition (OCR)

- This examples demonstrates the use of a deep network as a classifier in an OCR application. The characters (classes) which are involved are the Greek letters α, ν, o and τ, extracted from old historical documents. The respective class volumes are 1735, 1850, 2391 and 2264.

- Each binary image is converted to a binary feature vector by scanning it row-wise and concatenating the rows to form a $28 \times 28 = 784$ dimensional binary representation. In the sequel, 80% of the resulting patterns, per class, are randomly chosen to form the training set and the remaining patterns serve testing purposes. The class labels are represented by 4-digit binary codewords. For example, the first class (letter α) is given the binary code $[1\ 0\ 0\ 0\ 0]$, the second class is given the codeword $[0\ 1\ 0\ 0\ 0]$ and so on.
- Due to the binary nature of the patterns, the use of RBMs with binary stochastic units as the building blocks of a deep network is a natural choice.

Example For Optical Character Recognition (OCR)

- The chosen deep network consists of five layers in total: an input layer, $\mathbf{x}$, of 784 binary visible units, 3 layers, namely $\mathbf{h}^1$, $\mathbf{h}^2$ and $\mathbf{h}^3$, of hidden binary units (consisting of 500, 500 and 2000 nodes respectively) and, finally, an output layer, $\mathbf{y}$. The activation function of the four neurons of the output layer is the so called softmax. The output of the kth output neuron, $k = 1, 2, \ldots, M$, is given by:

$$\hat{y}_k = \frac{\exp(z_k)}{\sum_{m=1}^{M} \exp(z_m)}, \; k = 1, 2, \ldots, M$$

  where z_m denotes the input to the activation function of the mth neuron. This can easily be shown to provide the posterior probability estimates of the patterns for each one of the classes.

- During the testing stage, each unknown pattern is "clamped" on the visible nodes of the input layer, $\mathbf{x}$, and the network operates in a feed-forward mode to propagate the results until the output layer, $\mathbf{y}$, has been reached. During this feed-forward operation, the nodes of the hidden layers propagate activation outputs, i.e, the probabilities at the output of their logistic functions. For each input pattern, the softmax node corresponding to the maximum value is chosen as the winner and the pattern is assigned to the respective class.

Example For Optical Character Recognition (OCR)

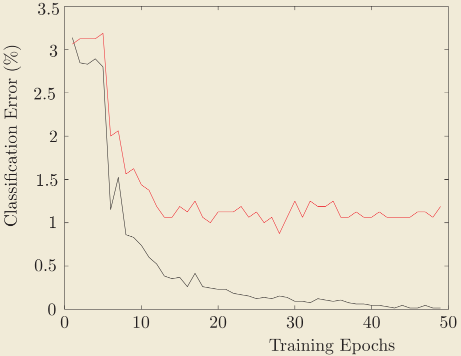
- The chosen deep network consists of five layers in total: an input layer, $\mathbf{x}$, of 784 binary visible units, 3 layers, namely $\mathbf{h}^1$, $\mathbf{h}^2$ and $\mathbf{h}^3$, of hidden binary units (consisting of 500, 500 and 2000 nodes respectively) and, finally, an output layer, $\mathbf{y}$. The activation function of the four neurons of the output layer is the so called softmax. The output of the kth output neuron, $k = 1, 2, \ldots, M$, is given by:

$$\hat{y}_k = \frac{\exp(z_k)}{\sum_{m=1}^{M} \exp(z_m)}, \; k = 1, 2, \ldots, M$$

  where z_m denotes the input to the activation function of the mth neuron. This can easily be shown to provide the posterior probability estimates of the patterns for each one of the classes.
- During the testing stage, each unknown pattern is "clamped" on the visible nodes of the input layer, $\mathbf{x}$, and the network operates in a feed-forward mode to propagate the results until the output layer, $\mathbf{y}$, has been reached. During this feed-forward operation, the nodes of the hidden layers propagate activation outputs, i.e, the probabilities at the output of their logistic functions. For each input pattern, the softmax node corresponding to the maximum value is chosen as the winner and the pattern is assigned to the respective class.

Example For Optical Character Recognition (OCR)

- The figure below presents the training and testing error curves, at the end of each training epoch. Note that, due to the small number of classes and network size, the resulting errors become really small after just a few epochs. In this case, the errors are mainly due to seriously distorted characters. Furthermore, observe that the training error (as a general tend) decreases monotonically. In contrast, the test error curve, settles at around 1% of error probability.



Stacked Autoencoders

- Instead of building a deep network architecture by hierarchically training layers of, say, RBMs, in order to capture a representtation of the input data, one can alternatively employ autoencoders.
- The latter have been proposed as methods for dimensionality reduction. An autoencoder consists of two parts, the encoder and the decoder.

Stacked Autoencoders

- Instead of building a deep network architecture by hierarchically training layers of, say, RBMs, in order to capture a representtation of the input data, one can alternatively employ autoencoders.
- The latter have been proposed as methods for dimensionality reduction. An autoencoder consists of two parts, the encoder and the decoder.

Stacked Autoencoders

- Encoder: The output of the encoder is the reduced representation of the input pattern, and it is defined in terms of a vector function,

$$\boldsymbol{f} : \boldsymbol{x} \in \mathbb{R}^l \longmapsto \boldsymbol{h} \in \mathbb{R}^m, \quad h_i := f_i(\boldsymbol{x}) = \phi_e(\boldsymbol{\theta}_i^T \boldsymbol{x} + b_i^e), \ i = 1, 2, \ldots, m,$$

  with $\phi_e(\cdot)$ being the activation function; the latter is usually taken to be the logistic sigmoid function, $\phi_e(\cdot) = \sigma(\cdot)$.

- Decoder: The decoder is another function $\boldsymbol{g}$,

$$\boldsymbol{g} : \boldsymbol{h} \in \mathbb{R}^m \longmapsto \hat{\boldsymbol{x}} \in \mathbb{R}^l, \quad \hat{x}_j = g_j(\boldsymbol{h}) = \phi_d(\boldsymbol{\theta}_j^{'T} \boldsymbol{h} + \beta_j^d), \ j = 1, 2, \ldots, l.$$

  The activation $\phi_d(\cdot)$ is, usually, taken to be either the identity (linear reconstruction) or the logistic sigmoid one.

Stacked Autoencoders

- Encoder: The output of the encoder is the reduced representation of the input pattern, and it is defined in terms of a vector function,

  $$\boldsymbol{f} : \boldsymbol{x} \in \mathbb{R}^l \longmapsto \boldsymbol{h} \in \mathbb{R}^m, \quad h_i := f_i(\boldsymbol{x}) = \phi_e(\boldsymbol{\theta}_i^T \boldsymbol{x} + b_i^e), \; i = 1, 2, \ldots, m,$$

  with $\phi_e(\cdot)$ being the activation function; the latter is usually taken to be the logistic sigmoid function, $\phi_e(\cdot) = \sigma(\cdot)$.

- Decoder: The decoder is another function $\boldsymbol{g}$,

  $$\boldsymbol{g} : \boldsymbol{h} \in \mathbb{R}^m \longmapsto \hat{\boldsymbol{x}} \in \mathbb{R}^l, \quad \hat{x}_j = g_j(\boldsymbol{h}) = \phi_d(\boldsymbol{\theta}_j^{\prime T} \boldsymbol{h} + \beta_j^d), \; j = 1, 2, \ldots, l.$$

  The activation $\phi_d(\cdot)$ is, usually, taken to be either the identity (linear reconstruction) or the logistic sigmoid one.

Stacked Autoencoders

- The task of training is to estimate the parameters,

$$\Theta := [\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_m,], \ \boldsymbol{b}^e, \ \Theta^{'} := [\boldsymbol{\theta}_1^{'}, \ldots, \boldsymbol{\theta}_l^{'}], \ \boldsymbol{b}^d.$$

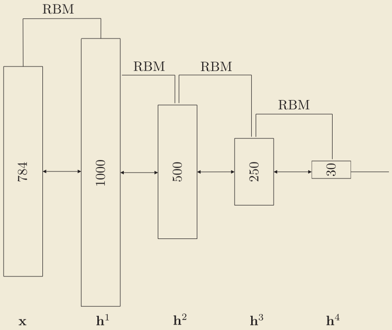
  It is common to assume that $\Theta^{'} = \Theta^T$. The parameters are estimated so as the reconstruction error, $\boldsymbol{e} = \boldsymbol{x} - \hat{\boldsymbol{x}}$, over the available input samples, to be minimum in some sense, e.g., least squares.

Example: A Deep Autoencoder

- The goal of the encoder is to gradually reduce the dimensionality of the input vectors and this will be achieved by using a multilayer neural network, where the hidden layers decrease in size. We are going to demonstrate the method via an example, using the database of the Greek letters discussed before and following the same procedure concerning the partition in training and test data.
- The block diagram of the encoder part of the autoencoder is shown below:

Example: A Deep Autoencoder

- The goal of the encoder is to gradually reduce the dimensionality of the input vectors and this will be achieved by using a multilayer neural network, where the hidden layers decrease in size. We are going to demonstrate the method via an example, using the database of the Greek letters discussed before and following the same procedure concerning the partition in training and test data.
- The block diagram of the encoder part of the autoencoder is shown below:

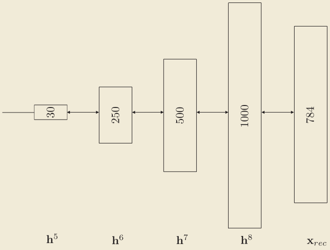


Example: A Deep Autoencoder

- The first three hidden layers consist of binary units, whereas the last layer consists of linear (Gaussian) units. We then proceed by pre-training the weights connecting every pair of successive layers using the contrastive divergence algorithm (for 20 epochs), starting from $(\mathbf{x}, \mathbf{h}^1)$ and proceeding with $(\mathbf{h}^1, \mathbf{h}^2)$ and so on. This is in line with what we discussed so far for training deep networks. For the RBM training stage, we use the whole training data set and divide it into mini-batches (consisting of 100 patterns), as it is common practice.
- The decoder is the reverse structure, i.e., its input layer receives the 30-dimensional representation at the output of $\mathbf{h}^4$ and consists of four hidden layers of increasing size, whose dimensions reflect exactly the hidden layers of the encoder, plus an output layer.

Example: A Deep Autoencoder

- The first three hidden layers consist of binary units, whereas the last layer consists of linear (Gaussian) units. We then proceed by pre-training the weights connecting every pair of successive layers using the contrastive divergence algorithm (for 20 epochs), starting from $(\mathbf{x}, \mathbf{h}^1)$ and proceeding with $(\mathbf{h}^1, \mathbf{h}^2)$ and so on. This is in line with what we discussed so far for training deep networks. For the RBM training stage, we use the whole training data set and divide it into mini-batches (consisting of 100 patterns), as it is common practice.
- The decoder is the reverse structure, i.e., its input layer receives the 30-dimensional representation at the output of $\mathbf{h}^4$ and consists of four hidden layers of increasing size, whose dimensions reflect exactly the hidden layers of the encoder, plus an output layer.



Example: A Deep Autoencoder

- After all the weights have been initialized as described before, the whole encoder-decoder network is treated as amultilayer feed-forward network and its weights are fine-tuned via the backpropagation algorithm (for 200 epochs); for each input pattern, the desired output is the pattern itself. In this way, the backpropagation algorithm tries to minimize the reconstruction error.

Example: A Deep Autoencoder

- After all the weights have been initialized as described before, the whole encoder-decoder network is treated as amultilayer feed-forward network and its weights are fine-tuned via the backpropagation algorithm (for 200 epochs); for each input pattern, the desired output is the pattern itself. In this way, the backpropagation algorithm tries to minimize the reconstruction error.

Input patterns and respective reconstructions. The top row shows the original patterns. The bottom row shows the corresponding reconstructed patterns a) Left block, prior to the application of the backpropagation algorithm for fine-tuning and b) Right block, after the fine tuning.