Lecture 9

Neural Networks and Support Vector Machines

Why neural networks?

There are problems that cannot be formulated as algorithms. Therefore the question to be asked is: How do we learn to explore such problems?

Work on neural networks has been motivated by the recognition that the human brain computes in an entirely different way than the digital computer. The brain is highly complex, nonlinear, and parallel information-processing system.

At birth, the brain already has considerable structure and ability to build up its own rules of behavior through experience. A developing nervous system is synonymous with a plastic brain: **plasticity** permits the developing nervous system to adapt to its surrounding environment.

A neural network (NN) is a massively parallel distributed system made up of simple processing units that has a natural propensity for storing experiential knowledge and making it available for use. It resembles the brain in two respects:

1. Knowledge is acquitted by the network from its environment through a learning process.
2. Interneuron connection strengths, known as synaptic weights, are used to store the acquired knowledge.

The procedure used to perform the learning process is called a learning algorithm, the function of which is to modify the synaptic weights of the network in order to achieve a desired objective.

Benefits of Neural Networks

- **Nonlinearity.** A system is nonlinear if it does not follow a linear model, i.e., the output of a nonlinear system is not directly proportional to the input. For example, small changes in the input can produce large and disproportional changes in the output. Nonlinear systems are of interest to computer scientists, physicists and mathematicians because most existing systems are inherently nonlinear in nature.
- **Input-Output Mapping, i. e., Function.** A NN can perform supervised learning, in which the network modifies the synaptic weights according to a labeled set of examples. Each example consists of a unique input signal and the corresponding target response. When the network is presented with an example from the training set, the synaptic weights (the free parameters of the model) are modified to minimize the difference between the target response and the actual response. The training of the network is repeated until it reaches a steady state where there are no further significant changes in the synaptic weights. In this way, the neural network learns from examples by constructing an input-output mapping, i.e., a function.

Benefits of Neural Networks *(cont.)*

- The Input-Output mapping ability of NNs resemble nonparametric statistical inference, which deals with model-free estimation. The term nonparametric is used to signify the fact that no proper assumptions are made on the statistical model for the input data. This is also similar to “tabula rasa” learning in biological systems.
- **Additivity.** NNs have a built-in ability to adapt their synaptic weights to changes in the surrounding environment. For example, a NN trained to operate in a specific environment can be easily retrained to deal with minor changes in the operating environment. Moreover, a NN can operate in a nonstationary environment (one that constantly changes) by changing its synaptic weights in real time.
- **Generalize and associate data**. After successful training a NN can find reasonable solutions for similar problems of the same class that were not explicitly trained.

Benefits of Neural Networks *(cont.)*

- **Evidential response.** In pattern classification, a NN can be designed to provide information not only about which particular pattern to select but also about the confidence on the decision made.
- **Contextual information.** Knowledge is represented by the structure of the NN, where every neuron can be affected by the global activity of all other neurons.
- **Fault tolerance.** The performance of NNs degrades gracefully under adverse operating conditions. For example, the damage of a single neuron might not affect to a large degree the performance of the NN. Another example: about 10^5 neurons can be destroyed while in a drunken state.
- **VLSI implement ability**.
- **Neurobiological analogy.** Neurobiologists look to artificial NNs as a research tool for the interpretation of neurobiological phenomena. On the other hand, engineers look to neurobiology for new ideas to solve problems more complex than those based on the conventional hardwired designed techniques.

The brain vs. the computer

The largest part of the brain is working continuously, while the largest part of the computer is only passive data storage. Thus, the brain is parallel and performing close to its theoretical maximum, from which the computer is orders of magnitude away. Additionally, a computer is static - the brain as a biological neural network can reorganize itself during its "lifespan" and therefore is able to learn, to compensate errors and so forth.

	Brain	Computer
No. of processing units	$\approx 10^{11}$	$\approx 10^{9}$
Type of processing units	Neurons	Transistors
Type of calculation	massively parallel	usually serial
Data storage	associative	address-based
Switching time	$\approx 10^{-3}$s	$\approx 10^{-9}$s
Possible switching operations	$\approx 10^{13}\frac{1}{s}$	$\approx 10^{18}\frac{1}{s}$
Actual switching operations	$\approx 10^{12}\frac{1}{s}$	$\approx 10^{10}\frac{1}{s}$

The 100-step rule

The human brain performs massive and parallel information processing. Experiments showed that a human can recognize the picture of a familiar object or person in 0.1 seconds, which corresponds to 100 discrete time steps of parallel processing, given the neuron switching time of 10^{-3} seconds. A computer following the von Neumann architecture, however, can do practically nothing in 100 time steps of *sequential* processing, which are 100 assembler steps or cycle steps.

History of neural networks

- **1943:** Warren McCulloch and Walter Pitts introduced models of neurological networks, recreated threshold switches based on neurons and showed that even simple networks of this kind are able to calculate nearly any logic or arithmetic function.
- **1949:** Donald O. Hebb formulated the classical Hebbian rule which represents in its more generalized form the basis of nearly all neural learning procedures. The rule implies that the connection between two neurons is strengthened when both neurons are active at the same time. Hebb's theory is often summarized by Siegrid Löwel's phrase: "Neurons that fire together, wire together."
- **1950:** The neuropsychologist Karl Lashley defended the thesis that brain information storage is realized as a *distributed system*. His thesis was based on experiments on rats, where only the extent but not the location of the destroyed nerve tissue influences the rats' performance to find their way out of a labyrinth.

History of neural networks *(cont.)*

- **1951:** For his dissertation Marvin Minsky developed the neurocomputer ***Snark***, which has already been capable to adjust its weights automatically. But it has never been practically implemented, since it is capable to busily calculate, but nobody really knows what it calculates.
- **1957-1958:** At the MIT, Frank Rosenblatt, Charles Wightman and their coworkers developed the first successful neurocomputer, the Mark I perceptron, which was capable to recognize simple numerics by means of a 20 × 20 pixel image sensor and electromechanically worked with 512 motor driven potentiometers – each potentiometer representing one variable weight.
- **1959:** Frank Rosenblatt formulated and verified his *perceptron convergence theorem*.

History of neural networks *(cont.)*

- **1960:** Bernard Widrow and Marcian E. Hoff introduced the ADALINE (ADAptive LInear NEuron), a fast and precise adaptive learning system being the first widely commercially used neural network: It could be found in nearly every analog telephone for real time adaptive echo filtering and was trained by means of the Widrow-Hoff delta rule. Later, Hoff became co-founder of Intel Corporation. One advantage the delta rule had over the original perceptron learning algorithm was its *adaptivity*: If the difference between the actual output and the correct solution was large, the connecting weights also changed in larger steps – the smaller the steps, the closer the target was.
- **1969:** Marvin Minsky and Seymour Papert published a precise mathematical analysis of the perceptron to show that the perceptron model was not capable of representing many important problems (such as the XO*R problem* and *linear separability*), and so put an end to overestimation, popularity and research funds. The entire field would be a *research dead* for the next 15 years.

History of neural networks *(cont.)*

- **1972:** Teuvo Kohonen introduced a model of the ***linear associator***, a model of an associative memory.
- **1973:** Christoph von der Malsburg used a neuron model that was nonlinear and biologically more motivated.
- **1974:** For his dissertation in Harvard Paul Werbos developed a learning procedure called *backpropagation of error* but it was not until one decade later that this procedure reached today's importance.
- **1982:** Teuvo Kohonen described the ***self-organizing feature maps*** (SOM), also known as Kohonen maps. He was looking for the mechanisms involving self-organization in the brain.
- **1982:** John Hopfield also invented the so-called Hopfield networks, which are inspired by the laws of magnetism in physics.
- **1985:** John Hopfield published an article describing a way of finding acceptable solutions for the Travelling Salesman problem by using *Hopfield nets*.
- **1986:**The *backpropagation of error* learning procedure as a generalization of the delta rule was separately developed and widely published by the *Parallel Distributed Processing* Group

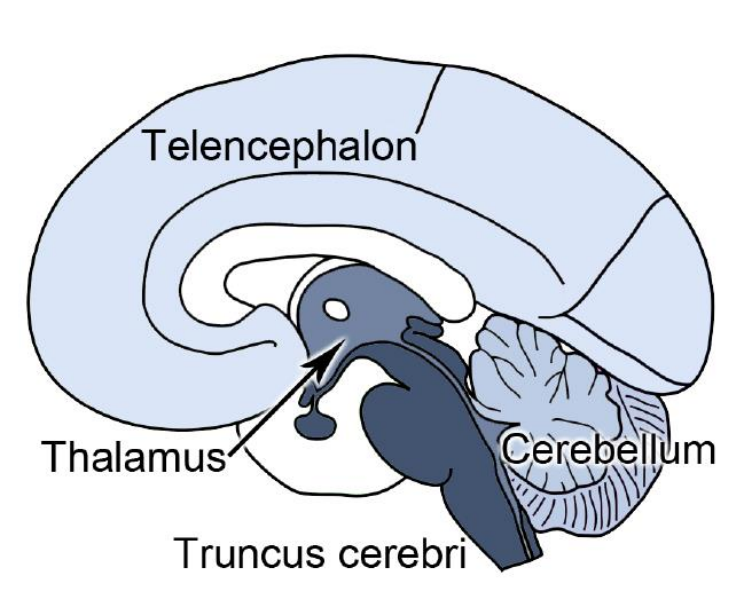
Biological neural networks

- The brain contains about 1.1 trillion cells, including about 100 billion neurons. On average each neuron receives about five thousand connections, called synapses, from other neurons.
- The number of possible combinations of 100 billion neurons firing or not is approximately 10 to the millionth power, or 1 followed by a million zeros. This is the number of possible states of your brain, To put this number in perspective, the number of atoms in the universe is estimated to be about 10 to the power of eighty.
- Although the brain is only 2% of the body's weight, it uses 20-25% of its oxygen and glucose.
- The entire information processing system, i.e. the vertebrate ***nervous system***, consists of the central nervous system and the peripheral nervous system.
- The ***peripheral nervous system*** (***PNS***) comprises the nerves that are situated outside of the brain or the spinal cord. These nerves form a branched and very dense network throughout the whole body.

Biological neural networks *(cont.)*

- The ***central nervous system*** (***CNS***), is the "main-frame“. It is the place where information received by the sense organs are stored and managed. It also controls the inner processes in the body and coordinates the motor functions of the organism. The vertebrate central nervous system consists of the ***brain*** and the ***spinal cord.***
- The ***cerebrum*** (***telencephalon***) is responsible for abstract thinking processes. It is divided into two hemispheres, which are organized in a folded structure. These cerebral hemispheres are connected by one strong nerve cord and several small ones. A large number of neurons are located in the ***cerebral cortex*** which is approx. 2- 4 cm thick and divided into different ***cortical fields***, each having a specific task to fulfill. ***Primary cortical fields*** are responsible for processing qualitative information, such as the management of different perceptions (e.g. the ***visual cortex*** is responsible for the management of vision). ***Association cortical fields***, perform more abstract association and thinking processes. They also contain our memory.

Biological neural networks *(cont.)*

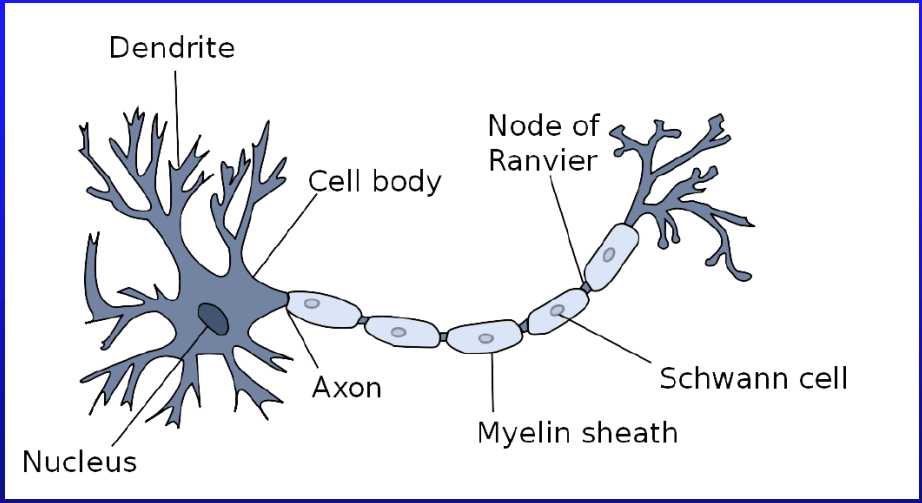


Biological neural networks *(cont.)*

- The **cerebellum** controls and coordinates motor functions. For example, it controls balance and movements and continually corrects errors. For this purpose, the cerebellum has direct sensory information about muscle lengths as well as acoustic and visual information.
- The ***interbrain*** (***diencephalon***) includes many parts. For example, of the ***thalamus*** mediates between sensory and motor signals and the cerebrum. The thalamus decides which part of the information must be transferred to the cerebrum, so that especially less important sensory perceptions can be suppressed at short notice to avoid overloads. It plays a central role in alertness and awareness.
- Another part of the diencephalon is the ***hypothalamus***, which controls a number of processes within the body. The hypothalamus controls many automatic functions, such as hunger, thirst, body temperature, and sexual activity. It also regulates circadian rhythms, odors, stress, blood pressure, perspiration and heart rate.
- The **brainstem** connects the brain with the spinal cord and controls reflexes. For example it controls the blinking reflex and coughing.

Neurons

- A neuron can be thought of as a switch with information input and output. The switch is activated if there are enough stimuli of other neurons hitting the information input. Then, at the information output, a pulse is sent to, for example, other neurons.
- Incoming signals from other neurons or cells are transferred to a neuron by special connections, called ***synapses***. Such connections can usually be found at the dendrites of a neuron and sometimes at the cell nucleus, called soma. There are electrical and chemical synapses.



Neurons *(cont.)*

- In the ***electrical synapse,*** *a*n electrical signal received by the synapse, i.e. coming from the *presynaptic* side, is directly transferred to the *postsynaptic* nucleus of the cell. In other words, there is a direct, strong, and unadjustable connection between the signal transmitter and the signal receiver. This is relevant to shortening reactions that must be "hard coded" within a living organism.
- In the **chemical synapse**, the electrical coupling of source and target is interrupted by the synaptic cleft. This cleft electrically separates the presynaptic side from the postsynaptic one. The information flows as a chemical process. On the presynaptic side of the synaptic cleft the electrical signal is converted into a chemical signal, a process induced by chemical cues released there called neurotransmitters. These neurotransmitters cross the synaptic cleft and transfer the information into the nucleus of the cell where it is reconverted into electrical information. The neurotransmitters are degraded very fast, so that it is possible to release very precise information pulses.

Neurons *(cont.)*

- A chemical synapse is a one-way connection. Due to the fact that there is no direct electrical connection between the pre- and postsynaptic area, electrical pulses in the postsynaptic area cannot be transferred to the presynaptic area.
- The chemical synapse is highly adjustable. There is a large number of different neurotransmitters that can also be released in various quantities in a synaptic cleft. There are neurotransmitters that stimulate the postsynaptic cell nucleus, and others that slow down such stimulation. Some synapses transfer a strongly stimulating signal, some only weakly stimulating ones. The adjustability varies a lot, and synapses are variable, too. That is, over time they can form a stronger or weaker connection.
- In other words, a chemical synapse converts a presynaptic electric signal into a chemical signal and then back into a postsynaptic electrical signal. You can think of a an chemical synapse as a connection that can impose excitation or inhabitation, but not both on the receptive neuron.

Neurons *(cont.)*

- ***Dendrites*** branch like trees from the cell nucleus of the neuron (which is called *soma*) and receive electrical signals from many different sources, which are then transferred into the nucleus of the cell.
- After the cell nucleus (***soma***) has received a plenty of activating and inhibiting signals by synapses or dendrites, the soma accumulates these signals. As soon as the accumulated signal exceeds a certain value (called threshold value), the cell nucleus of the neuron activates an electrical pulse which then is transmitted to the neurons connected to the current one.
- The pulse is transferred to other neurons by means of the ***axon***. In an extreme case, an axon can stretch up to one meter (e.g. within the spinal cord). The axon is electrically isolated in order to achieve a better conduction of the electrical signal and it leads to dendrites, which transfer the information to, for example, other neurons. An axon can also transfer information to other kinds of cells in order to control them.

Neurons *(cont.)*

- We already mentioned that plasticity permits the brain to adapt to its surrounding system. Plasticity can be accounted for by two mechanisms: the creation of new synaptic connections between neurons and the modification of existing synapses.

The amount of neurons in living organisms at different stages of development

- **302 neurons** are required by the nervous system of a *nematode worm*. Nematodes live in the soil and feed on bacteria.
- **10^4 neurons** make an *ant.* Due to the use of different attractants and odors, ants are able to engage in complex social behavior and form huge states with millions of individuals.
- With **10^5 neurons** the nervous system of a *fly* can be constructed.
- With **$0.8 \cdot 10^6$ neurons** we have enough cerebral matter to create a *honeybee.*
- **10^6 neurons** result in a *mouse*

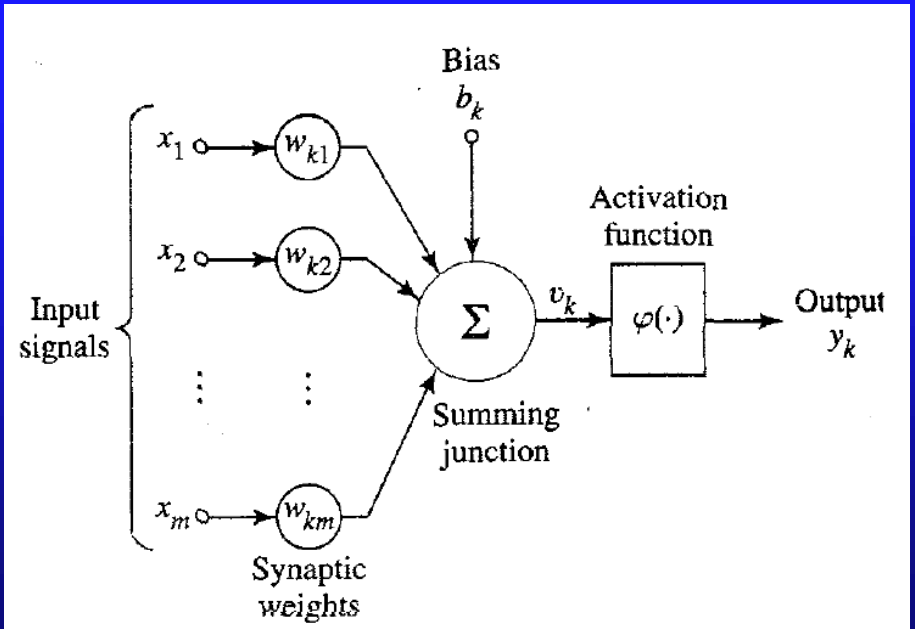
The amount of neurons in living organisms at different stages of development

- **10^7 neurons** are sufficient for a *rat*, an animal which is denounced as being extremely intelligent and are often used to participate in a variety of intelligence tests representative for the animal world.
- **10^8 neurons** are required by the brain of a *dog or a cat.*
- With **$6 \cdot 10^9$ neurons** we get a *chimpanzee.*
- **10^{11} neurons** make a *human.*
- **With $2 \cdot 10^{11}$ neurons** we can get an *elephant* and certain *whale* species.

The Artificial Neuron

An Artificial Neural network, ANN, neural network consists of simple processing units, the *neurons*, and directed, weighted connections between them. The strength of a connection is measured by the synaptic *weigh. The* artificial neuron is an information-processing unit that is fundamental to the operation of artificial NNs. It converts input signals to an output signal. The following diagram presents a mathematical model of the neuron. The neuron has three basic elements:

- a set of synapses or connecting links;
- an adder;
- an activation function.



The Artificial Neuron *(cont.)*

- A signal x_j at the input of synapse j connected to neuron k is multiplied by the synaptic weight w_{kj}. The synaptic weight could be positive or negative.
- The adder performs linear combining, i.e., it sums the input signals, weighted by the respective synaptic weights.
- The activation function determines when the neuron gets activated and the output value to be delivered to other neurons.
- The combined weighted input can be represented as:

$$u_k = \sum_{j=1}^{m} w_{kj} x_j$$

- The activation function converts the combined weighted input u_k to an output y_k:

$$y_k = \varphi(u_k + b_k)$$

- Here, b_k is a term that represents the bias. It could be negative, positive or zero.

ANNs

ANNs can be described in terms of:

- The **activation function.** Different ANNs can use different activation functions.
- The **network topology**, which describes the number of neurons in the model as well as the number of layers and manner in which they are connected
- The **training algorithm** that specifies how connection weights are set in order to inhibit or excite neurons in proportion to the input signal

Types of activation functions

- There are two basic types of activation functions: **threshold** and **sigmoid**.
- Threshold function. The biological neuron uses a threshold activation function or ***threshold potential***, which lies at about −55 mV. As soon as the received stimuli reach this value, the neuron is activated and an electrical signal, an ***action potential***, is initiated. Then, this signal is transmitted to the cells connected to the neuron.
- The biological neuron is activated by changes in the membrane potential, which is controlled by sodium and potassium ions that can diffuse through the membrane, sodium slowly, potassium faster, and change the membrane potential. The ions move through channels within the membrane, the sodium and potassium channels. In addition to these permanently open channels, there also exist channels that are not always open but which react to certain conditions. Since the opening of these channels changes the concentration of ions within and outside of the membrane, it also changes the membrane potential. These controllable channels are opened as soon as the accumulated received stimulus exceeds a certain threshold.

Threshold activation of the biological neuron

The biological neuron goes through several states:

- **Resting state:** Only the permanently open sodium and potassium channels are permeable. The membrane potential is at −70 mV

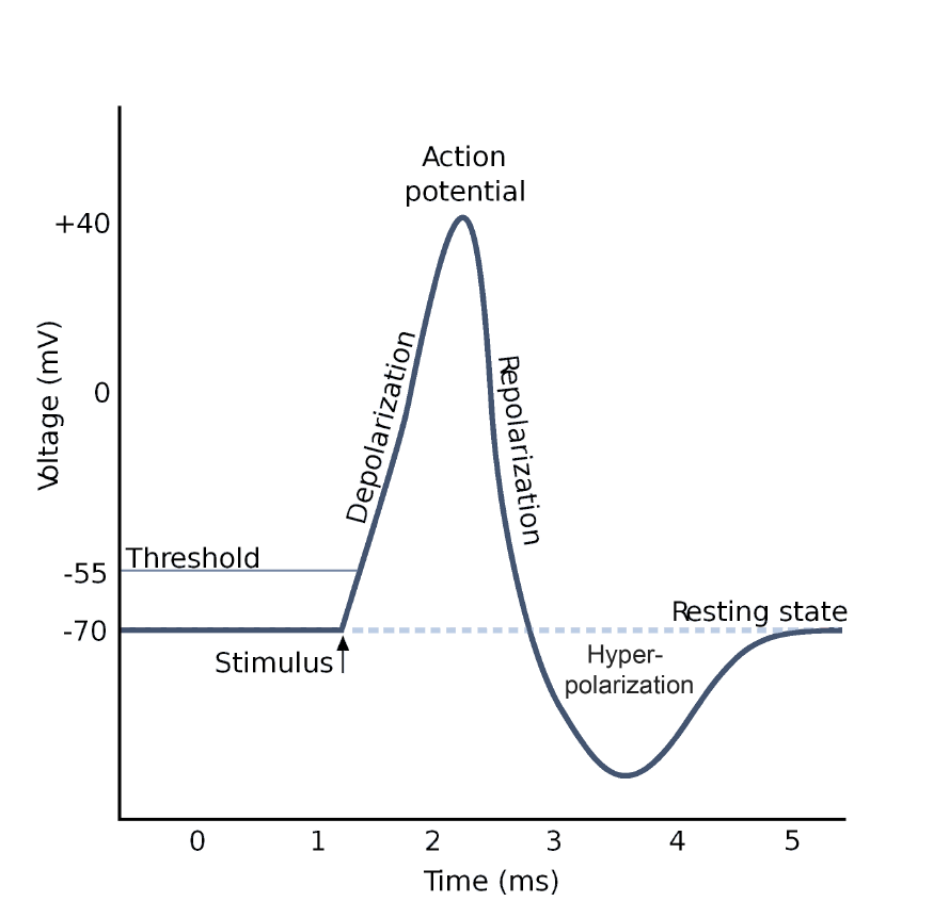
- **Stimulus up to the threshold:** A ***stimulus*** opens channels so that sodium can pour in. The intracellular charge becomes more positive. As soon as the membrane potential exceeds the threshold of −55 mV, the action potential is initiated by the opening of many sodium channels.

- **Depolarization:** Sodium is pouring in. Sodium wants to pour into the cell because there is a lower intracellular than extracellular concentration of sodium. Additionally, the cell is dominated by a negative environment which attracts the positive sodium ions. This massive influx of sodium drastically increases the membrane potential - up to approx. +30 mV - which is the electrical pulse, i.e., the action potential.

Threshold activation of the biological neuron

- **Repolarization:** Now the sodium channels are closed and the potassium channels are opened. The positively charged ions want to leave the positive interior of the cell. Additionally, the intracellular concentration is much higher than the extracellular one, which increases the influx of ions even more. The interior of the cell is once again more negatively charged than the exterior.
- **Hyperpolarization:** Sodium as well as potassium channels are closed again. At first, the membrane potential is slightly more negative than the resting potential. This is due to the fact that the potassium channels close more slowly. As a result, (positively charged) potassium effuses because of its lower extracellular concentration. After a ***refractory period*** of 1 − 2 ms the resting state is reestablished so that the neuron can react to newly applied stimuli with an action potential. In simple terms, the refractory period is a mandatory break a neuron has to take in order to regenerate. The shorter this break is, the more often a neuron can fire per time.

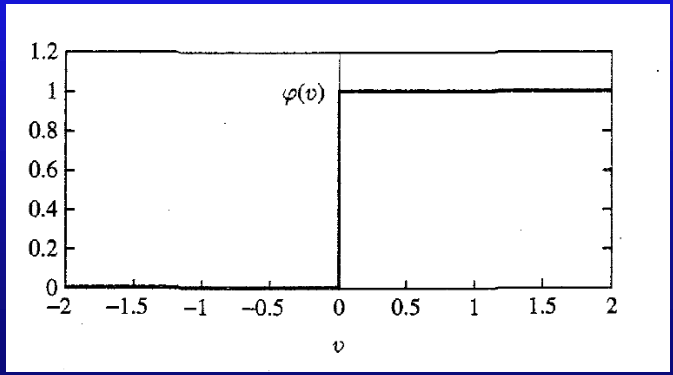
Threshold activation of the biological neuron



Threshold function

From the biological point of view the threshold value represents the threshold at which a neuron starts firing. The simplest activation function is the ***binary threshold function*** which can only take on two values (also referred to as ***Heaviside function***). If the input is above a certain threshold, the function changes from one value to another, but otherwise remains constant. This implies that the function is not differentiable at the threshold and for the rest the derivative is 0. Due to this fact, backpropagation learning (we will learn it later), for example, is impossible.

$$\varphi(v) = \begin{cases} 1 & \text{if } v \geq 0 \\ 0 & \text{if } v < 0 \end{cases}$$



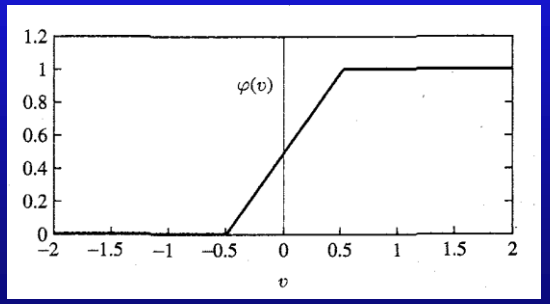
Threshold function *(cont.)*

Such a neuron is referred to as McCulloch-Pitts model in recognition of the pioneering work of McCulloch and Pitts.

Another type of threshold function is the piecewise-linear function, defined by the following equation:

$$\varphi(v) = \begin{cases} 1, & v \geq +\frac{1}{2} \\ v, & +\frac{1}{2} > v > -\frac{1}{2} \\ 0, & v \leq -\frac{1}{2} \end{cases}$$

It is often used as an approximation of a non linear activation function. Here is the graph of the function, which shows that the function is not differentiable at -0.5 and 0.5:

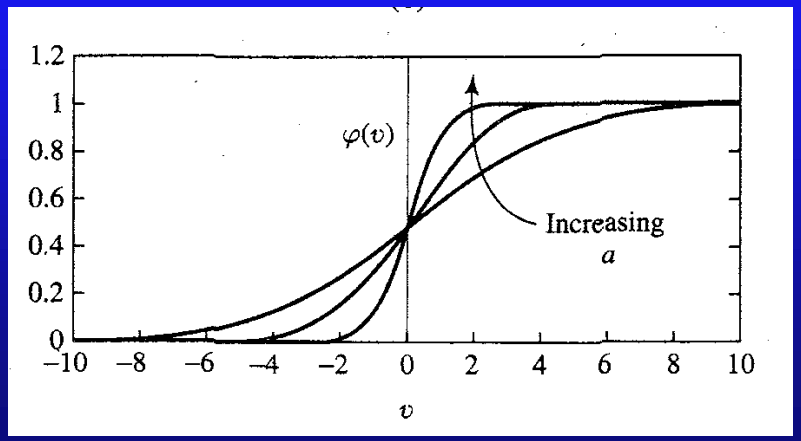


Sigmoid activation function

The sigmoid activation function, whose graph is s-shaped, is the most common form of activation function. It is strictly increasing and exhibits a graceful balance between linear and nonlinear behavior. An example of a sigmoid function is the logistic function:

$$\varphi(v) = \frac{1}{1 + \exp(-av)}$$

a is the slope parameter. By varying the slope parameter we get sigmoid functions of different slopes:



Squashing

Both threshold and sigmoid activation functions produce the so called squashing effect: most of the input values are squeezed into much smaller output interval. The graph of the sigmoid function shows that all input values less than -10 correspond to an output of 0 and all input values greater than 10 correspond to an output of 1. One solution to the squashing problem is to transform all neural network inputs such that their values fall within a small range around 0. This, for example, can be done by standardizing or normalizing the input values.

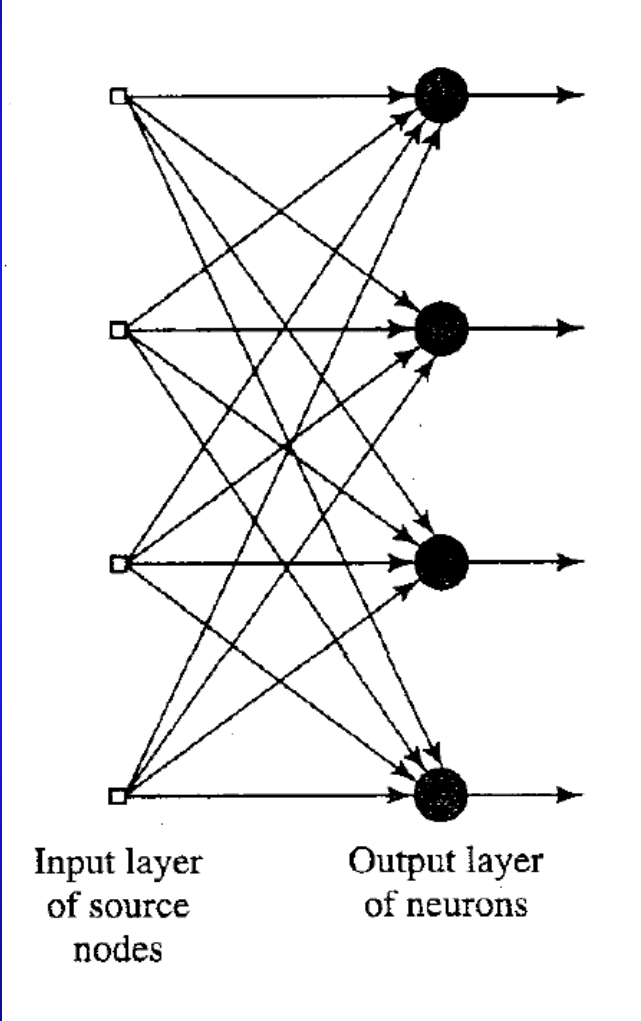
Network topology

The ANNs network topology can be defined in terms of:

- The number of layers
- Whether information in the network is allowed to travel backward
- The number of nodes within each layer of the network

In a layered network, the neurons are organized in the form of layers. In the simplest form of a layered network, we have an input layer of source nodes that projects directly onto an output layer of neurons (computational nodes), but not vice versa. In other words, this network is strictly of a *feedforward* type. Such a network is called a single-layer network, with the term “single-layer” referring to the output layer of computation nodes (neurons). We do not count the input layer of source nodes because no computation is preformed there.

Single-layer feedforward network



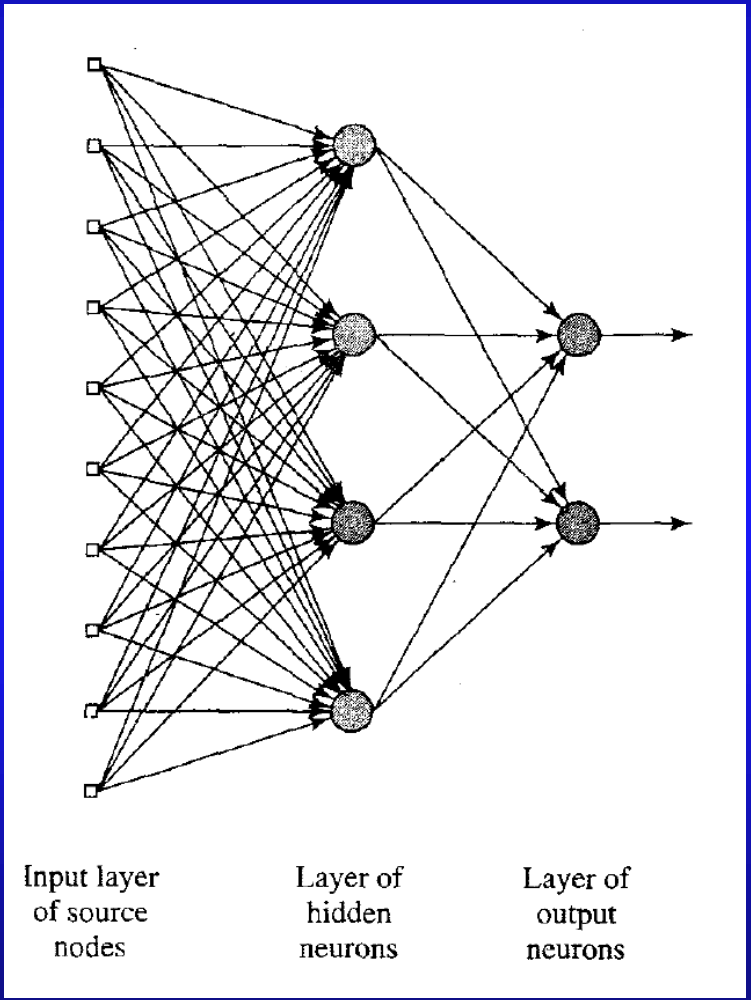
Multi-layer feedforward network

The second class of a feedforward network distinguishes itself by the presence of one or more hidden layers. The term hidden refers to the fact that this part of the network is not seen directly from either the input or output of the network. By adding one or more hidden layers, the network is enabled to extract higher-order statistics from its input. In a sense, the network acquires a global perspective despite its local connectivity due to the extra set of synaptic connections and the extra dimension of neural interactions. Typically, the neurons in each layer of the network have as their inputs the output signals of the preceding layer only.

Hidden layers allow a network to learn non-linear functions.

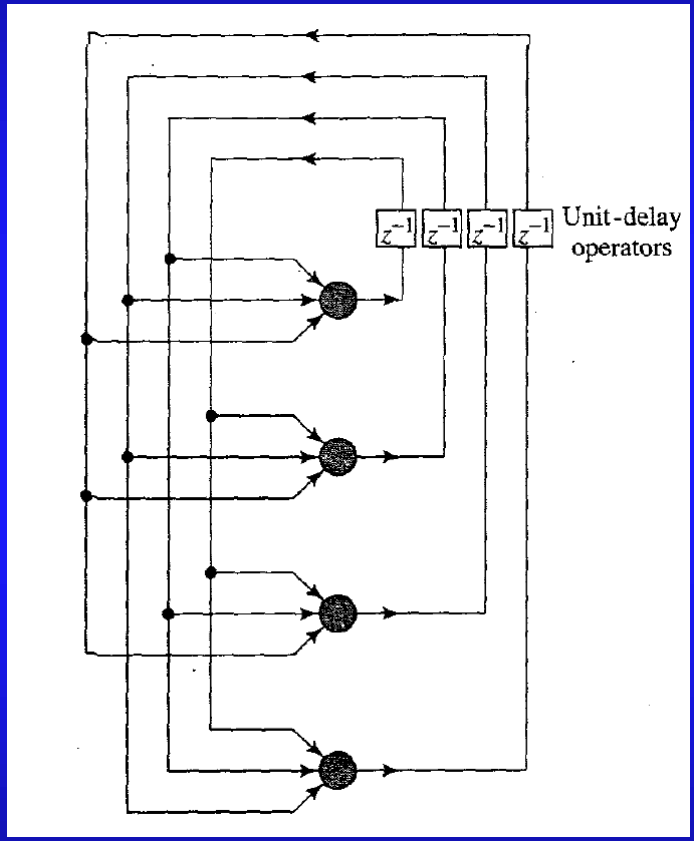
The network on the next slide is usually referred as 10-4-2 network because it has 10 source nodes, 4 hidden neurons, and 2 output neurons. This network is said to be *fully connected* in the sense that every node in each layer is connected to every node in the adjacent forward layer. If some communication links are missing, the network is said to be *partially connected.*

Multi-layer feedforward network



Recurrent networks

A recurrent network distinguishes itself from a feedforward network in that it has al least one *feedback loop*. For example, a recurrent network may consist of a single layer with each neuron feeding its output signal back to the inputs of all other neurons, as illustrated on the right side. It is also possible for a feedback loop to be a self feedback loop. This refers to the situation where the output of a neuron is fed back into its own input. The presence of unit-time delay elements allows the network to exhibit nonlinear behavior.



The number of nodes in each layer

The number of input nodes is predetermined by the number of features in the input data. The number of output nodes is predetermined by the number of outcomes to be modeled or the number of class levels in the outcome. There is no reliable rule to determine the number of neurons in hidden layers.

More complex network topologies with a greater number of network connections allow the learning of more complex problems. However, there is always a risk of overfitting. In addition, large and complex networks can be computationally expensive and slow to train.

It has been proved that any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].

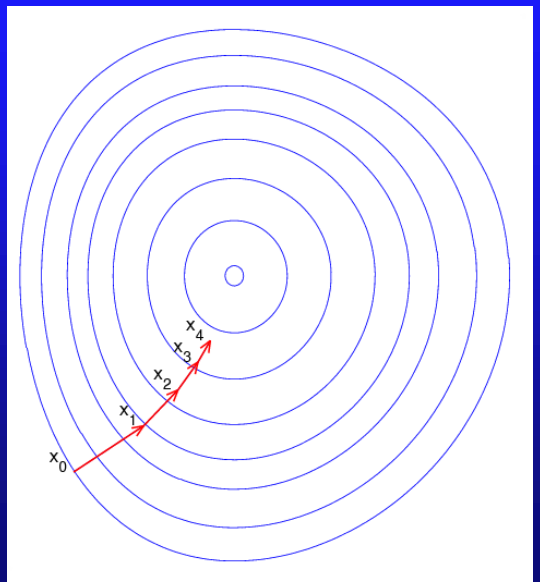
Every bounded continuous function can be approximated with arbitrarily small error by network with single hidden layer [Cybenko 1989; Hornik et al. 1989].

Learning algorithm: Backpropagation

A feed-forward neural network implements a function. It takes its inputs from the input layer and generates the outputs on the output layer. Such functions are called vector functions because the set of inputs and the set of outputs are vectors. The weights on the edges in the neural network serve as parameters that determine the function. When we change the weights on the edges, the function also changes. In general, our goal is to generate a set of weights that will enable the neural network to closely simulate a desired function. The training of a neural network consist in continuously adjusting the weights until a desirable error rate is achieved. The backpropagation algorithm is one of the earliest algorithms for training ANNs. It uses a training and a test sets of examples. The neural network is trained by feeding the input vector into the neurons in the input layer, seeing what output vector is generated by the network in the feed forward phase, comparing the desired output vector with the generated output vector and then adjusting the weights in the network in an attempt to move the generated output vector closer to the desired output vector. The back-propagation algorithm incrementally adjust the weights using a a technique called gradient descent.

Gradient descent

This is a simple optimization algorithm. To find a local minimum of a function using gradient descent, one takes steps proportional to the *negative* of the gradient of the function at the current point. If one wants to find a local maximum, then one must work in the opposite direction by taking steps proportional to the gradient descent. In general, the gradient shows the direction of the function's growth. For example, one can start at an arbitrary point, x_0, compute the gradient at x_0, take a small step to x_1 in a direction proportional to the negative of the gradient, compute the gradient at x_1, make another small step to x_2, compute the gradient and so on.



Gradient descent

Formally:

1. Choose x_0
2. While $k<k_{max}$, $x_k = x_{k-1} - \lambda \nabla F(x_{k-1})$

Consider the function, $f(x; y) = w_1 x^2 + w_2 y$, where w_1 and w_2 are parameters that we can choose. We want to find values of w_1 and w_2 that enable this function to simulate our training examples. For simplicity, we start with $w_1 = w_2 = 0.5$. Our fist training example says that the input vector (2; 3) should go with the output 9. In the feed-forward phase, we plug-and-chug:

$$f(2; 3) = 0.5 * 22 + 0.5 * 3 = 2 + 1.5 = 3.5$$

3.5 is not very close to our desired value of 9. How should we adjust the weights of our function to get closer to the desired output value? In general, we want to minimize the difference between the desired output value and the generated output value. Mathematically, it turns out to be easier to minimize the square of this difference, which leads to the same result. In our case, the square of the difference is $E = (9 - 3.5)^2 = 30.25$. The question is: how should we incrementally adjust the weights, w_1 and w_2, to make the function f generate an output closer to the desired output?

Gradient descent

To adjust the weights, we need first to compute the gradient of the squared error, which requires computing the partial derivatives of E with respect to the parameters w_1 and w_2. If D is the desired output:

$$\frac{\partial}{\partial w_1}(D - f(x;y))^2 = \frac{\partial}{\partial w_1}\left(D - (w_1 x^2 + w_2 y)\right)^2$$
$$= 2\left(D - (w_1 x^2 + w_2 y)\right)(-x^2)$$

Evaluating this partial derivative at $x = 2$ and $y = 3$, and $w_1 = w_2 = 0.5$ yields:

$$2(9 - (0.5 * 4 + 0.5 * 3))(-4) = -44.$$

Similarly,

$$\frac{\partial}{\partial w_2}(D - f(x;y))^2 = \frac{\partial}{\partial w_2}\left(D - (w_1 x^2 + w_2 y)\right)^2$$
$$= 2\left(D - (w_1 x^2 + w_2 y)\right)(-y)$$

Gradient descent

Evaluating this partial derivative at $x = 2$ and $y = 3$, and $w_1 = w_2 = 0.5$ yields:

$$2(9 - (0.5 * 4 + 0.5 * 3))(-3) = -33$$

In other words, the gradient at point (2, 3) is (-44, -33). This vector points in the direction of maximum instantaneous increase of E. The negative of this vector (i.e., (44; 33)) points in the direction of maximum instantaneous decrease of E. Since we want to minimize the squared error, E, we want to move the values of w_1 and w_2 a tiny bit in the direction of (44; 33). This can be done by adding 0.044 to w_1 and 0.033 to w_2. Here, we have multiplied each component of the negative gradient by the same fraction, 1/100 . The choice of fraction is arbitrary; however, it should be small. The new weights are:

$$w_1 = 0.5 + 0.044 = 0.544\,, w_2 = 0.5 + 0.033 = 0.533$$

Gradient descent

Now, we can verify that the output generated using these new weights is slightly closer to the desired output that is the output generated using the original weights:

Original: $f(2; 3) = 0.5*2^2 + 0.5*3 = 3.5$

New: $f(2; 3) = 0.544*2^2 + 0.533*3 = 3.775$

Although it doesn't seem like a huge progress; keep in mind that this is only the result of one training example.

In general, the backpropagation algorithm iterates through cycles called **epochs**. The starting weights are typically set at random. Then, the algorithm iterates through the processes, until a stopping criterion is reached. Each epoch in the backpropagation algorithm includes:

- A **forward phase** in which a training example is given to the neural network and the output is recorded.
- A **backward phase** in which the output is compared to the true target value and the difference between the output and the target value is propagated backwards in the network to modify the connection weights between neurons and reduce future errors.

Modeling the strength of concrete with ANNs

In this example, we will show how to use an ANN to estimate the strength of concrete. The concrete dataset contains 1,030 examples of concrete with eight features describing the components used in the mixture. These features are related to the concrete strength and they include the amount (in kilograms per cubic meter) of cement, slag, ash, water, superplasticizer, coarse aggregate, and fine aggregate used in the product in addition to the aging time (measured in days).

```
> concrete <- read.csv("concrete.csv")
> str(concrete)
'data.frame':   1030 obs. of  9 variables:
 $ cement      : num  141 169 250 266 155 ...
 $ slag        : num  212 42.2 0 114 183.4 ...
 $ ash         : num  0 124.3 95.7 0 0 ...
 $ water       : num  204 158 187 228 193 ...
 $ superplastic: num  0 10.8 5.5 0 9.1 0 0 6.4 0 9 ...
 $ coarseagg   : num  972 1081 957 932 1047 ...
 $ fineagg     : num  748 796 861 670 697 ...
 $ age         : int  28 14 28 28 28 90 7 56 28 28 ...
 $ strength    : num  29.9 23.5 29.2 45.9 18.3 ...
```

Exploring and preparing the data

In order to avoid the squashing effect, we will scale the data to a narrow range around zero. We can scale the data either with the built-in *scale()* function we used before or with the *normalize()* function we defined before. The *scale()* function works better with data that follow a normal distribution, whereas the *normalize()* function works better with data that follow uniform distribution. In this case, we will use the *normalize()* function.

```
> normalize <- function(x) {
+ return((x - min(x)) / (max(x) - min(x)))
+ }
```

As we did before, we will use lapply() to apply the normalize() function to every column in the concrete data frame:

To confirm that the normalization worked, we can print a summary, showing that all values lie between 0 and 1:

```
> summary(concrete_norm$strength)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.0000  0.2664  0.4001  0.4172  0.5457  1.0000
```

Training a model on the data

We can assume that the CSV file is already sorted in random order. We can divide it into two portions: a training set with 75 percent of the examples and a testing set with 25 percent.

```
> concrete_train <- concrete_norm[1:773, ]
> concrete_test <- concrete_norm[774:1030, ]
```

There are several R packages for training ANNs. We will use the neuralnet package developed by Stefan Fritsch and Frauke Guenther.

```
> install.packages("neuralnet")
> library(neuralnet)
```

Neural network syntax

using the `neuralnet()` function in the `neuralnet` package

Building the model:

```
m <- neuralnet(target ~ predictors, data = mydata,
               hidden = 1)
```

- `target` is the outcome in the `mydata` data frame to be modeled
- `predictors` is an R formula specifying the features in the `mydata` data frame to use for prediction
- `data` specifies the data frame in which the `target` and `predictors` variables can be found
- `hidden` specifies the number of neurons in the hidden layer (by default, 1)

The function will return a neural network object that can be used to make predictions.

Making predictions:

```
p <- compute(m, test)
```

- `m` is a model trained by the `neuralnet()` function
- `test` is a data frame containing test data with the same features as the training data used to build the classifier

The function will return a list with two components: `$neurons`, which stores the neurons for each layer in the network, and `$net.result`, which stores the model's predicted values.

Example:

```
concrete_model <- neuralnet(strength ~ cement + slag
  + ash, data = concrete)
model_results <- compute(concrete_model,
  concrete_data)
strength_predictions <- model_results$net.result
```

Training a model on the data

First, we will train a simple multilayer feedforward network with only a single hidden node:

```
> concrete_model <- neuralnet(strength ~ cement + slag+ ash +
water + superplastic + coarseagg + fineagg + age,data =
 + concrete_train)
```

We can plot the network using the *plot()* command:

```
> plot(concrete_model)
```

You can see the nodes, the connections and the weights. The bias terms are denoted as nodes labeled with the number 1. By definition the bias b_k was defined as:

$$\sum_{j=1}^{m} w_{kj}\, x_j + b_k$$

We can account for b_k if we introduce one more "dummy" node x_{m+1}, which is set to 1, with $w_{m+1} = b_k$:

$$\sum_{j=1}^{m} w_{kj}\, x_j + b_k = \sum_{j=1}^{m+1} w_{kj}\, x_j$$

Evaluating model performance

At the bottom of the figure, you will find the number of training steps and an error measure, **Sum of Squared Errors** (**SSE**).

Mathematically, a neural network with a single hidden node is very similar to linear regression. The weights of the neural network correspond to the regression coefficients and the bias correspond to the intercept.

We use the *compute()* to generate predictions:

```
> model_results <- compute(concrete_model, concrete_test[1:8])
```

The compute() function returns a list with two components: $neurons, which stores the neurons for each layer in the network, and $net.result, which stores the predicted values:

```
>  predicted_strength <- model_results$net.result
```

Because this is a numeric model we cannot use a confusion matrix to examine model accuracy. Instead, we will use the correlation between our predicted concrete strength and the true value:

```
> cor(predicted_strength, concrete_test$strength)
             [,1]
[1,] 0.8063692675
```

Improving model performance

The correlation of about 0.806 indicates a fairly strong relationship between the predicted and the test values. One way to improve the performance of the model is ti increase the number of hidden nodes to five.

```
> concrete_model2 <- neuralnet(strength ~ cement + slag + ash
+ + water + superplastic + coarseagg + fineagg + age, data =
+ concrete_train, hidden = 5)
```

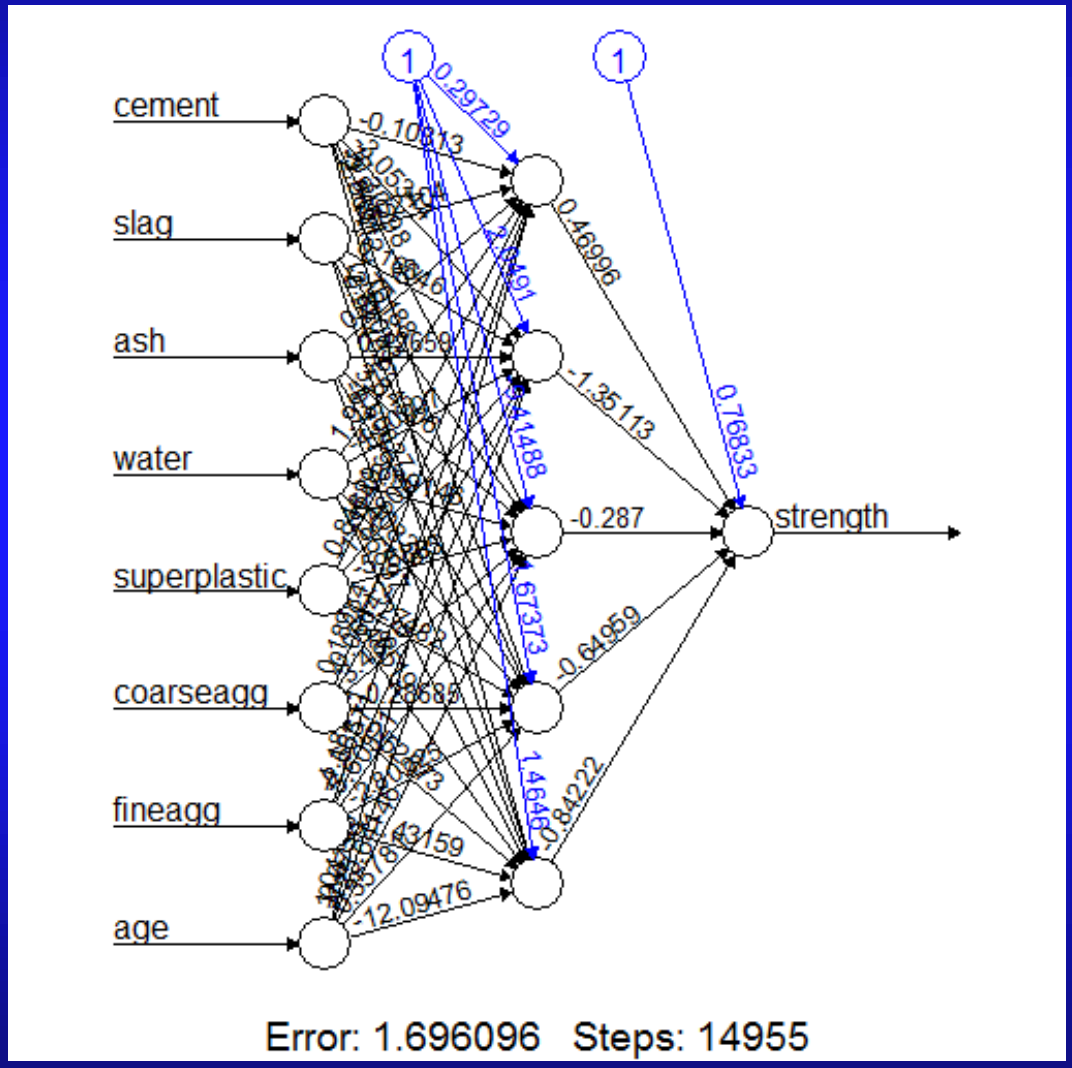
As you can see, it takes some time to train the network.

```
> plot(concrete_model2)
```

The network is shown on the next slide. It shows that the reported error has been reduced from 5.08 in the previous model to 1.63 in this model. As usual, we will compute the predicted values and will compare them with the test values. The correlation increases from 0.806 to 0.93

```
> predicted_strength2 <- model_results2$net.result
> cor(predicted_strength2, concrete_test$strength)
             [,1]
[1,] 0.9334346491
```

Improving model performance



Support Vector Machines

Hyperplane: In a *p*-dimensional space, a hyperplane is a flat affine subspace of hyperplane dimension $p - 1$. For instance, in two dimensions, a hyperplane is a flat one-dimensional subspace, i.e., a line. In three dimensions, a hyperplane is a flat two-dimensional subspace, i.e., a plane. In two dimensions, a hyperplane (a line) is defined by the equation:

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 = 0$$

That is, every point (x_1, x_2) on the hyperplane satisfies the equation. The equation for a p-dimensional hyperplane is:

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p = 0$$

Each hyperplane divides the space into two half spaces. All points in the first half space satisfy the inequality:

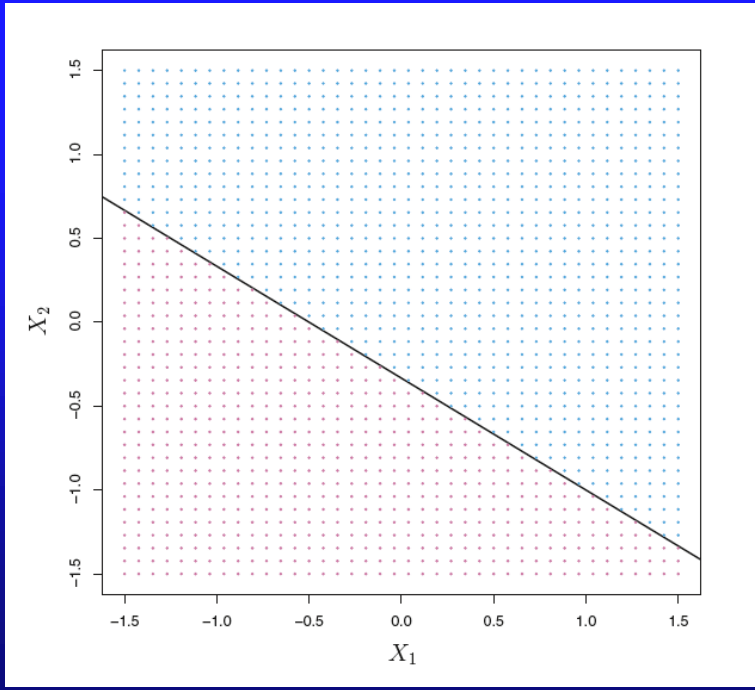
$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p > 0.$$

Maximal Margin Classifier

All points in the second half space satisfy the inequality:

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p < 0,$$

The hyperplane $1 + 2X_1 + 3X_2 = 0$ is shown below. The blue region is the set of points for which $1 + 2X_1 + 3X_2 > 0$, and the purple region is the set of points for which $1 + 2X_1 + 3X_2 < 0$.



Maximal Margin Classifier

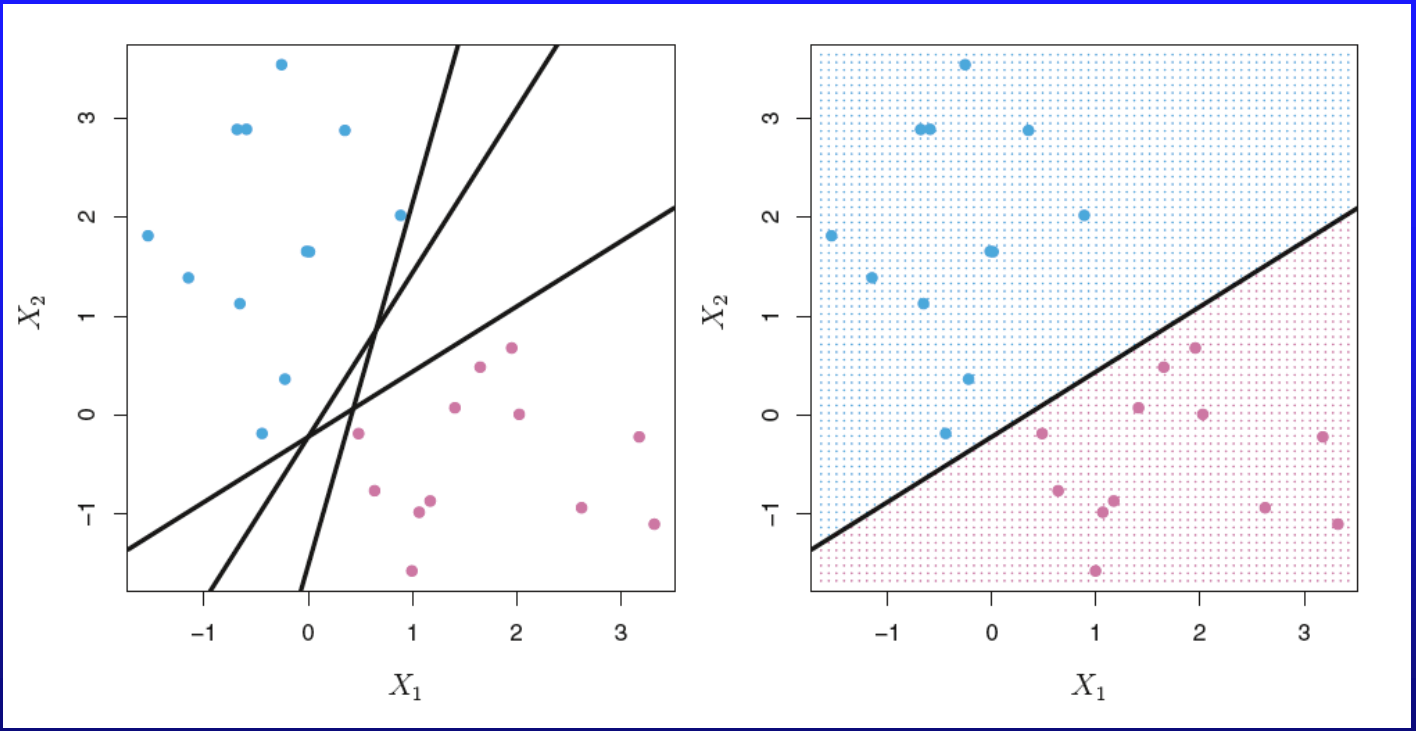
Suppose that we have a $n \times p$ data matrix $\mathbf{X}$ that consists of n training observations in p-dimensional space:

$$x_1 = \begin{pmatrix} x_{11} \\ \vdots \\ x_{1p} \end{pmatrix}, \ldots, x_n = \begin{pmatrix} x_{n1} \\ \vdots \\ x_{np} \end{pmatrix},$$

Suppose that these observations fall into two classes. That is, $x_1, \ldots, x_n \in \{-1, 1\}$ where -1 represents one class and 1 the other class. We also have a test observation, a p-vector of observed features $x^* = (x_1^*,..,x_p^*)^T$. Our goal is to develop a classifier based on the training data that will correctly classify the test observation using its feature measurements. We will use the *separating hyperplane* approach. Suppose that it is possible to construct a hyperplane that separates the hyperplane training observations perfectly according to their class labels. Examples of three such *separating hyperplanes* are shown in the left-hand panel on the next slide.

Maximal Margin Classifier

There are two classes of observations, shown in blue and in purple, each of which has measurements on two variables. Three separating hyperplanes, out of many possible, are shown in black on left. On right, there is a separating hyperplane shown in black. The blue and purple grid indicates the decision rule made by a classifier based on this separating hyperplane: a test observation that falls in the blue portion of the grid will be assigned to the blue class, and a test observation that falls into the purple portion of the grid will be assigned to the purple class.



Maximal Margin Classifier

We can label the observations from the blue class as $y_i = 1$ and those from the purple class as $y_i = -1$. Then a separating hyperplane has the property that:

$$\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip} > 0 \text{ if } y_i = 1,$$

and

$$\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip} < 0 \text{ if } y_i = -1.$$

We, can combine these two cases into one:

$$y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip}) > 0$$

for every point in 1 or -1. If a separating hyperplane exists, we can use it to construct a very natural classifier: a test observation is assigned a class depending on which side of the hyperplane it is located. That is, we classify the test observation *x** based on the sign of the hyperplane function:

$$\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip}$$

Maximal Margin Classifier

If it is positive, then we assign the test observation to class 1, and if it is negative, then we assign it to class −1. We can also make use of the magnitude of the hyperplane function at x*. If the magnitude is far from zero, then this means that x*∗ lies far from the hyperplane, and so we can be confident about our class assignment for x*. On the other hand, if the magnitude is close to zero, then *x** is located near the hyperplane, and so we are less certain about the class assignment for *x**.

In general, if our data can be perfectly separated using a hyperplane, then there will in fact exist an infinite number of such hyperplanes. This is because a given separating hyperplane can usually be shifted a tiny bit up or down, or rotated, without coming into contact with any of the observations. Then, which hyperplane to choose? A natural choice **is the maximal margin hyperplane** (also known as the maximal margin hyperplane optimal separating hyperplane), which is the separating hyperplane that is farthest from the training observations.

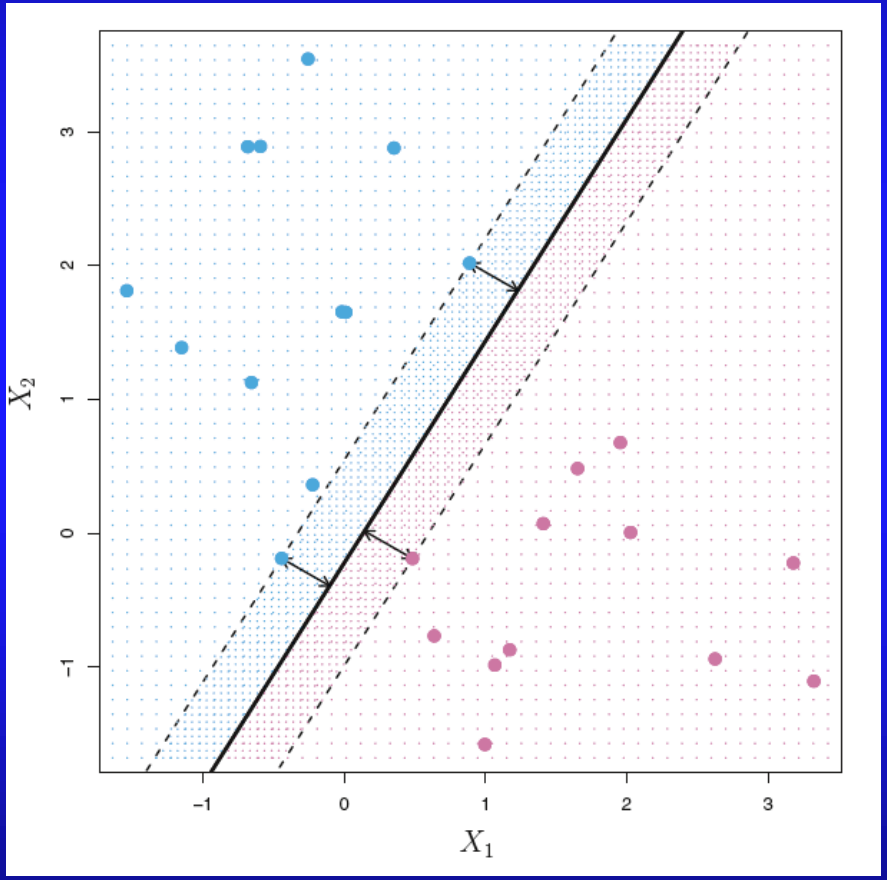
Maximal Margin Classifier

We can compute the (perpendicular) distance from each training observation to a given separating hyperplane; the smallest such distance is the minimal distance from the observations to the hyperplane, and is known as the *margin*. The maximal margin hyperplane is the separating hyperplane for which the margin is largest. That is, it is the hyperplane that has the farthest minimum distance to the training observations. We can then classify a test observation based on which side of the maximal margin hyperplane it lies. This is known as the **maximal margin classifier**.

There are three training observations on the next slide which are equidistant from the maximal margin hyperplane and lie along the dashed lines indicating the width of the margin. These three observations are known as **support vectors,** since they are vectors in *p*-dimensional space. The support vectors “support” the maximal margin hyperplane in the sense that if these points were moved slightly then the maximal margin hyperplane would move as well.

Maximal Margin Classifier

There are two classes of observations, shown in blue and in purple. The maximal margin hyperplane is shown as a solid line. The margin is the distance from the solid line to either of the dashed lines. The two blue points and the purple point that lie on the dashed lines are the support vectors, and the distance from those points to the margin is indicated by arrows. The purple and blue grid indicates the decision rule made by a classifier based on this separating hyperplane.



Maximal Margin Classifier

The maximal margin hyperplane depends directly on the support vectors, but not on the other observations: a movement to any of the other observations would not affect the separating hyperplane, provided that the observation's movement does not cause it to cross the boundary set by the margin. This is an important feature of support vectors: the support vectors provide a very compact way to store a classification model, even if the number of features is extremely large. The maximal margin hyperplane is the solution to the optimization problem:

$$\underset{\beta_0,\beta_1,\ldots,\beta_p}{\text{maximize}}\, M$$

$$\text{subject to } \sum_{j=1}^{p} \beta_j^2 = 1,$$

$$y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip}) \geq M \;\; \forall\, i = 1, \ldots, n.$$

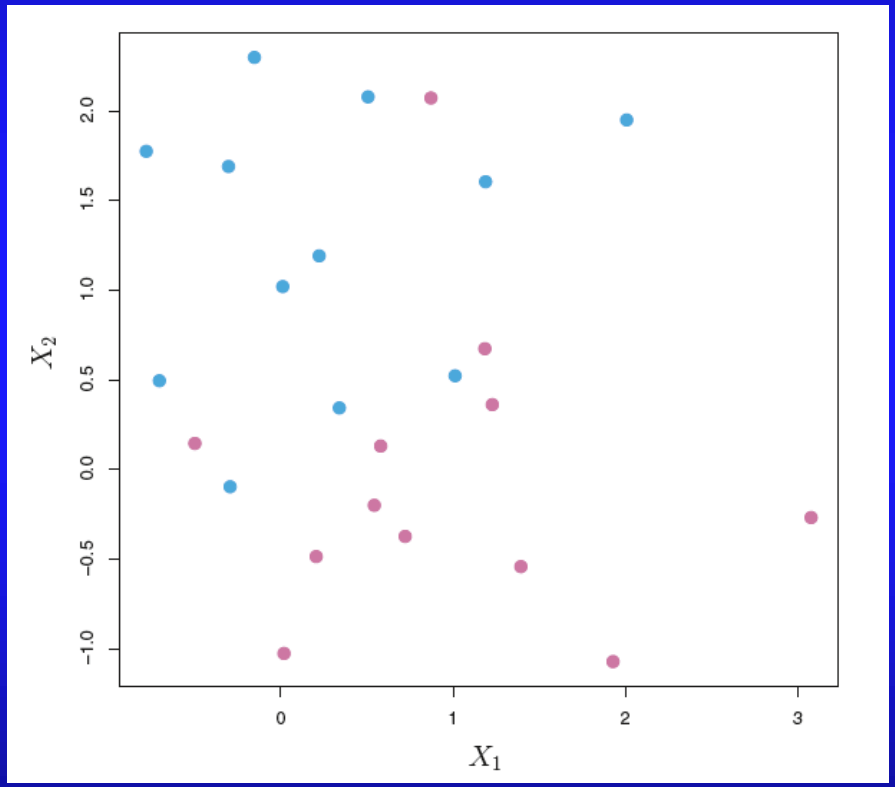
It is equivalent to the problem listed in your textbook:

$$\min \frac{1}{2} \|\vec{w}\|^2$$

$$s.t. \;\; y_i(\vec{w} \cdot \vec{x}_i - b) \geq 1, \forall \vec{x}_i$$

The non linearly separable case

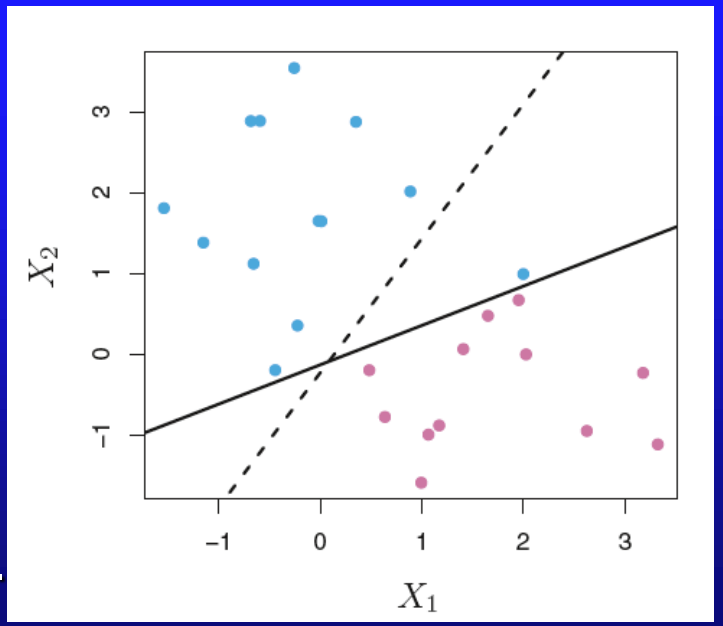
The maximal margin classifier is a very natural way to perform classification, if a *separating hyperplane exists*. However, in many cases no separating hyperplane exists, and so there is no maximal margin classifier. In this case, the optimization problem on the previous slide has no solution with $M > 0$. There is an example (shown on the right) of a non linearly separable case. There are two classes of observations, shown in blue and in purple. The two classes are not separable by a hyperplane, and so the maximal margin classifier cannot be used.



The non linearly separable case

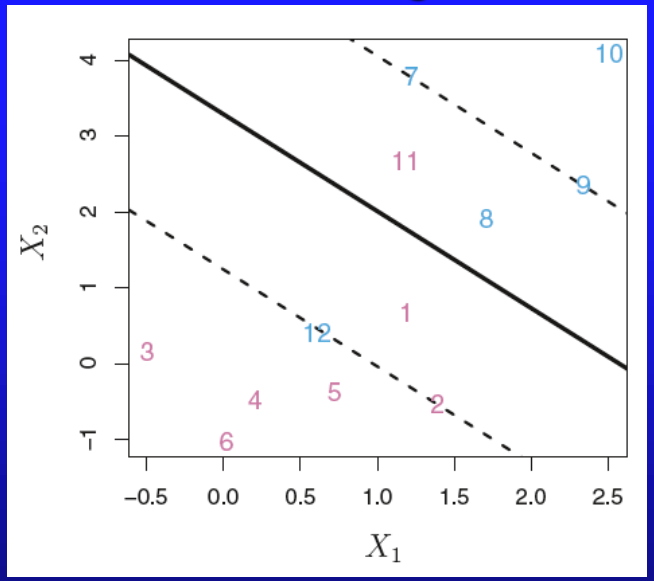
In the non linearly separable case, we can extend the concept of a separating hyperplane in order to develop a hyperplane that *almost* separates the classes, using a so-called *soft margin*.

There are cases in which a classifier based on a separating hyperplane might not be desirable even if a separating hyperplane exist. For example, if the margin is tiny, the classifier might overfit data or might me sensitive to small changes in data. Consider the example shown on the right. The addition of a single observation in the right-hand panel leads to a dramatic change in the maximal margin hyperplane. The dashed line indicates the maximal margin hyperplane that was obtained in the absence of this additional point The resulting maximal margin hyperplane is not satisfactory because it has a tiny margin. In this case, we might be willing to consider a classifier based on a hyperplane that does *not* perfectly separate the two classes in the interest of achieving greater robustness to individual observations, and obtaining better classification of most of the training observations.



The support vector classifier

The **support vector classifier**, sometimes called a **soft margin classifier**, misclassifies a few training observations in order to do a better job in classifying the remaining observation. It allows some observations to be on the incorrect side of the margin, or even the incorrect side of the hyperplane. The margin is *soft* because it can be violated by some of the training observations. On the figure, the purple observations 3, 4, 5 and 6 are on the right side of the margin. Observation 2 is on the margin, and observation 1 is on the wrong side of the margin. Observation 11 is on the wrong side of the plane. The blue observation 10 is on the right side of the margin, observations 7 and 9 are right on the margin, observation 8 is on the wrong side of the margin, and observation 12 is on the wrong side of the plane.



The support vector classifier

In the support vector classifier, the hyperplane is chosen to correctly separate most of the training observations into the two classes, but may misclassify a few observations. It is the solution to the optimization problem:

$$\begin{aligned}
&\underset{\beta_0,\beta_1,\ldots,\beta_p,\epsilon_1,\ldots,\epsilon_n}{\text{maximize}} \quad M \\
&\text{subject to} \quad \sum_{j=1}^{p} \beta_j^2 = 1, \\
&y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_p x_{ip}) \geq M(1 - \epsilon_i), \\
&\epsilon_i \geq 0, \quad \sum_{i=1}^{n} \epsilon_i \leq C,
\end{aligned}$$

M is the width of the margin; we seek to make this quantity as large as possible. $\varepsilon_1, \ldots, \varepsilon_n$ are *slack variables* that allow individual observations to be on slack, the wrong side of the margin or the hyperplane. *C* is a parameter which can be tuned. It represents the total slack. After solving the optimization problem, we find the hyperplane equation, which can be used to classify a test observation x* based on the sign of $\beta_0 + \beta_1 x_1^* + \ldots + \beta_p x_p^*$

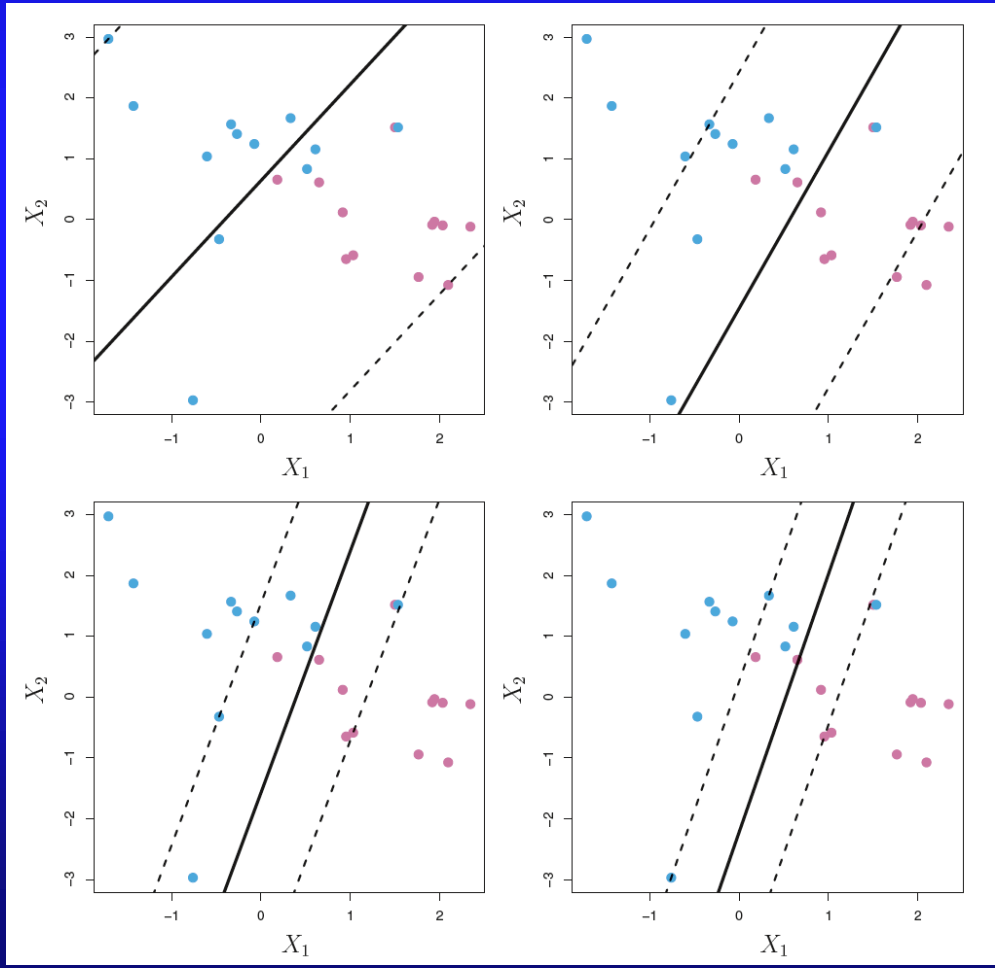
The support vector classifier

The soft margin hyperplane has a interesting property: it turns out that only observations that either lie on the margin or that violate the margin will affect the hyperplane, and hence the classifier obtained. In other words, an observation that lies strictly on the correct side of the margin does not affect the support vector classifier! Changing the position of that observation would not change the classifier at all, provided that its position remains on the correct side of the margin. Observations that lie directly on the margin, on the wrong side of the margin for their class, or on the wrong side of the hyperplane are known as *support vectors*. These observations do affect the support vector classifier.

When the tuning parameter C is large, then the margin is wide, many observations violate the margin, and so there are many support vectors. if C is small, then there will be fewer support vectors and hence the resulting classifier will have low bias but high variance. Therefore, C controls the bias-variance trade-off of the support vector classifier.

The support vector classifier

A support vector classifier was fit using four different values of the tuning parameter C. The largest value of C was used in the top left panel, and smaller values were used in the top right, bottom left, and bottom right panels.



The Support Vector Machine

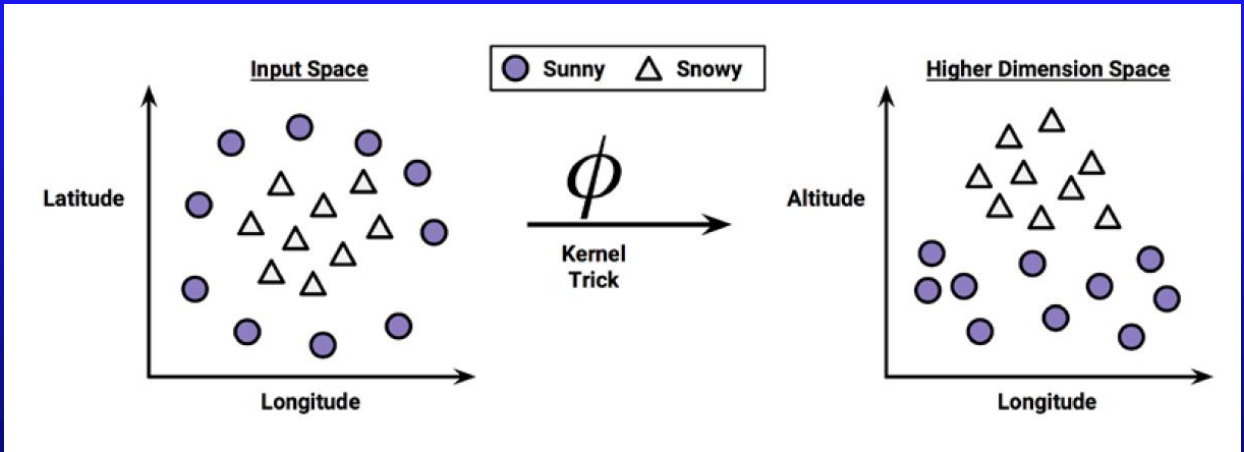
The maximal margin classifier and the soft margin classifier perform poorly if the boundaries between the sets are not close to linear. One solution for non linear cases is to enlarge the feature space using functions of the predictors, such as quadratic and cubic terms, in order to address this non-linearity. For instance, rather than fitting a support vector classifier using p features

$$X_1, X_2, \ldots, X_p,$$

we could instead fit a support vector classifier using 2*p* features

$$X_1, X_1^2 X_1{}^2, X_2, X_2^2, \ldots, X_p, X_p^2$$

This trick produces an interesting result: in the enlarged feature space, the decision boundary is linear and we can use a support vector classifier.



The Support Vector Machine

The enlargement of the feature space to $X_1, X_1^2 X_1{}^2, X_2, X_2^2, \ldots, Xp, X_p^2$ results in another mathematical optimization problem:

$$\underset{\beta_0,\beta_{11},\beta_{12}\ldots,\beta_{p1},\beta_{p2},\epsilon_1,\ldots,\epsilon_n}{\text{maximize}} \quad M$$

$$\text{subject to } y_i \left(\beta_0 + \sum_{j=1}^{p} \beta_{j1} x_{ij} + \sum_{j=1}^{p} \beta_{j2} x_{ij}^2 \right) \geq M(1 - \epsilon_i)$$

$$\sum_{i=1}^{n} \epsilon_i \leq C, \quad \epsilon_i \geq 0, \quad \sum_{j=1}^{p} \sum_{k=1}^{2} \beta_{jk}^2 = 1.$$

The constraint on the second row is a quadratic polynomial and it accounts for the nonlinear boundary in the original feature space. Interestingly enough, other functions of the predictors could be considered rather than polynomials. These functions are called *kernels*. The Support Vector Machine is an extension of the support vector classifier that results from enlarging the feature space in a specific way, using kernels.

The Support Vector Machine

Operating on the enlarged feature space is often computationally ineffective. Instead, most operations are done on the kernels in order to speed up computations. It turns out that the solution to the optimization problem from the previous slide involves only the *inner products* of the observations as opposed to the observations themselves. Thus the inner product of two p-dimensional vectors $x_i, x_{i'}$ is given by:

$$\langle x_i, x_{i'} \rangle = \sum_{j=1}^{p} x_{ij} x_{i'j}$$

In other words, the linear support vector classifier can be represented as:

$$f(x) = \beta_0 + \sum_{i=1}^{n} \alpha_i \langle x, x_i \rangle,$$

As we mentioned before, the equation of the support vector classifier depends only on the support vectors.

The Support Vector Machine

That is, α_i is nonzero only for the support vectors in the solution. In other words, if a training observation is not a support vector, then its α_i equals zero. So if *S* is the collection of indices of the support vectors, we can rewrite the equation for the support vector classifier as:

$$f(x) = \beta_0 + \sum_{i \in \mathcal{S}} \alpha_i \langle x, x_i \rangle$$

A kernel function $K(x_i, x_{i'})$ is a generalization of the inner product. In other words, the simplest kernel function is the inner product:

$$K(x_i, x_{i'}) = \sum_{j=1}^{p} x_{ij} x_{i'j},$$

The equation of the support vector classifier can be rewritten using kernel functions:

$$f(x) = \beta_0 + \sum_{i \in \mathcal{S}} \alpha_i K(x, x_i).$$

Kernel functions are symmetric, i.e., K(x,y) = K(y,x). In a sense, they measure the similarity between the points in the feature space.

The Support Vector Machine

As we discussed before, the Pearson correlation coefficient is based on the inner product of two vectors. The inner product is known as linear kernel because the support vector classifier is linear in the features. Examples of other kernel functions include polynomial kernel:

$$K(x_i, x_{i'}) = (1 + \sum_{j=1}^{p} x_{ij} x_{i'j})^d.$$

Radial kernel:

$$K(x_i, x_{i'}) = \exp(-\gamma \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2).$$

Gaussian kernel:

$$\mathrm{K}(\vec{x_i}, \vec{x_j}) = e^{\frac{-||\vec{x_i} - \vec{x_j}||^2}{2\sigma^2}}$$

Performing OCR with SVMs

As an example, we will develop a simple model for Optical Character Recognition (OCR) using a SVM. When OCR software first processes a document, it divides the paper into a matrix such that each cell in the grid contains a single glyph, which is a term referring to a letter, symbol, or number. We'll assume that we have already developed the algorithm to partition the document into rectangular regions each consisting of a single character. We will also assume that the document contains only alphabetic characters in English. Therefore, we'll simulate a process that involves matching glyphs to one of the 26 letters, A through Z.

We will read and explore the structure of the dataset (it is provided in the Course Materials section on Blackboard). The character images were based on 20 different fonts and each letter within these 20 fonts was randomly distorted to produce a file of 20,000 unique stimuli. Each stimulus was converted into 16 primitive numerical attributes (statistical moments and edge counts) which were then scaled to fit into a range of integer values from 0 through 15

```
> letters <- read.csv("letterdata.csv")
> str(letters)
```

The data frame includes 20000 observations of 17 variables

Attribute Information

1. lettr capital letter (26 values from A to Z)
2. x-box horizontal position of box (integer)
3. y-box vertical position of box (integer)
4. width width of box (integer)
5. high height of box (integer)
6. onpix total # on pixels (integer)
7. x-bar mean x of on pixels in box (integer)
8. y-bar mean y of on pixels in box (integer)
9. x2bar mean x variance (integer)
10. y2bar mean y variance (integer)
11. xybar mean x y correlation (integer)
12. x2ybr mean of x * x * y (integer)
13. xy2br mean of x * y * y (integer)
14. x-ege mean edge count left to right (integer)
15. xegvy correlation of x-ege with y (integer)
16. y-ege mean edge count bottom to top (integer)
17. yegvx correlation of y-ege with x (integer

Exploring and preparing the data

SVMs require all features to be numeric, and moreover, that each feature is scaled to a fairly small interval. In this case, the first requirement is met: every feature is an integer and we do not need to convert any factors into numbers. On the other hand, the ranges of some features are wide and require rescaling. Luckily for us, the SVM model will build will perform the rescaling automatically. The dataset has been already randomized and we will use the first 16,000 records (80 percent) for the training set and the and the next 4,000 records (20 percent) for the test set.

```
> letters_train <- letters[1:16000, ]
> letters_test <- letters[16001:20000, ]
```

Training a model on the data

There are several R packages for SVMs. We will use the *kernlab* package, which was developed natively in R and which can be used with the *caret* package that allows SVM models to be trained and evaluated using a variety of automated methods.

By default, the *ksvm()* function uses the Gaussian RBF kernel. Here is its syntax:

Support vector machine syntax

using the `ksvm()` function in the `kernlab` package

Building the model:

```
m <- ksvm(target ~ predictors, data = mydata,
          kernel = "rbfdot", C = 1)
```

- `target` is the outcome in the `mydata` data frame to be modeled
- `predictors` is an R formula specifying the features in the `mydata` data frame to use for prediction
- `data` specifies the data frame in which the `target` and `predictors` variables can be found
- `kernel` specifies a nonlinear mapping such as `"rbfdot"` (radial basis), `"polydot"` (polynomial), `"tanhdot"` (hyperbolic tangent sigmoid), or `"vanilladot"` (linear)
- `C` is a number that specifies the cost of violating the constraints, i.e., how big of a penalty there is for the "soft margin." Larger values will result in narrower margins

The function will return a SVM object that can be used to make predictions.

Making predictions:

```
p <- predict(m, test, type = "response")
```

- `m` is a model trained by the `ksvm()` function
- `test` is a data frame containing test data with the same features as the training data used to build the classifier
- `type` specifies whether the predictions should be `"response"` (the predicted class) or `"probabilities"` (the predicted probability, one column per class level).

The function will return a vector (or matrix) of predicted classes (or probabilities) depending on the value of the type parameter.

Example:

```
letter_classifier <- ksvm(letter ~ ., data =
  letters_train, kernel = "vanilladot")
letter_prediction <- predict(letter_classifier,
  letters_test)
```

Training a model on the data

```
> install.packages("kernlab")
> library(kernlab)
```

We call the *ksvm()* function on the training data and specify the linear (that is, vanilla) kernel using the vanilladot option:

```
> letter_classifier <- ksvm(letter ~ ., data = letters_train,
+ kernel = "vanilladot")
```

The equation “letter ~ .” means that we are using all features in the dataset to classify *letter*.

We run the *predict()* function to make predictions on the testing dataset:

```
> letter_predictions <- predict(letter_classifier,
 + letters_test)
```

letter_predictions is a factor with 26 levels, "A","B","C","D",..., containing a predicted letter for each observation

We can see the first 6 predictions using the *head()* function:

```
> head(letter_predictions)
[1] U N V X N H
Levels: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Evaluating model performance

Since the results are categorical we use the *table()* function to compare the test labels with the predicted labels:

```
> table(letter_predictions, letters_test$letter)
```

The table is 26 by 26 with all correct predictions on the main diagonal. Counts above and below the diagonal indicate errors. In order to count the number of the errors and the correct predictions, we cerate a vector of length 4000 with FALSE and TRUE values:

```
>  agreement <- letter_predictions == letters_test$letter
```

Then, we can count the number of TRUE and FALSE values in agreement:

```
> table(agreement)
agreement
FALSE  TRUE
  643  3357
```

This is about 84 percent:

```
> prop.table(table(agreement))
agreement
  FALSE    TRUE
0.16075 0.83925
```

Improving model performance

One way to improve the performance is to use a Gaussian RBF kernel:

It takes a while to produce the results:

```
> letter_classifier_rbf <- ksvm(letter ~ ., data =
+ letters_train, kernel = "rbfdot")
```

We can use the trained model to make predictions:

```
> letter_predictions_rbf <- predict(letter_classifier_rbf,
 + letters_test)
```

Finally, you can measure the model performance:

```
> agreement_rbf <- letter_predictions_rbf ==
letters_test$letter
> table(agreement_rbf)
agreement_rbf
FALSE  TRUE
  275  3725
```

Improving model performance

```
> prop.table(table(agreement_rbf))
agreement_rbf
   FALSE    TRUE
0.06875 0.93125
```

The accuracy of our character recognition model increased from 84 percent to 93 percent.

Another way to improve the model performance is to tune the parameter C in order to modify the width of the decision boundary.