# Evolutionary Strategies

Chromosom/Individuum -> realwertiger Vektor  
Gen -> Element des Vektors

**(1+1)-ES:**

Ein Elternteil -> (klonen) -> Ein Kind -> (Mutation) -> mutiertes Kind -> (Selektion) -> neue Population

P={p} c={c} c‘={c‘}

Weiterlaufen bis entweder vorgegebene Anzahl Generationen oder gewünschte Fitness erreicht wurde.

Mutation: zufällig normalverteilt mit Mittelwert = 0 und einer spezifischen Standardabweichung. Es gibt eine ideale Abweichung, die eine maximale Evolutionsgeschwindigkeit bringt. Zu wenig Varianz: Zu wenig Veränderungen. Zu viel Varianz: Zu große Änderungen führen zu nicht sinnvollen Mutationen.

**(µ+λ)-ES**

Nutzung von µ Eltern und λ Kindern, Auswahl aus der Union von Kindern und Eltern  
-> Eltern können mitgeschleppt werden (Unsterblichkeit), anfällig für lokale Maxima/Minima

**(µ, λ)-ES**

Wie oben, aber Auswahl der neuen Generation NUR aus Kindern, weniger anfällig für lokale Minima/Maxima

Selektionsdruck immer über das Verhältnis von µ und λ: ratio r=µ/ λ, 0 < r <= 1

**(µ/ρ # λ)-ES**

# steht für entweder + oder , und beschreibt, aus welcher Menge die neue Generation ausgewählt wird (und später auch, welche Generationen als neue Elterngenerationen gewählt werden können.

Kein Klonen mehr, Bildung von Paaren mit ρ Eltern und Rekombination der Gene via Mittelwert oder Crossover.  
Px=[3,7,2.8,9,5]  
Py=[7,7,2.0,5,1]  
r= [1,1,0 ,1,0] (Zufällig generiert bei Crossover)  
c1=[7,7,2.8,5,5]  
c2=[3,7,2.0,9,1]

**(m/r # l (µ/ρ # λ)/n)-ES**

Mehrere Parent-Populationen, Bildung von l Kinderpopulationen aus jeweils r gemischten Parent-Populationen. Auswahl der neuen Elternpopulationen aus den besten Kinder-Populationen (Im Falle `,`) oder aus Eltern- und Kind-Generationen.

# Genetische Algorithmen

Grundsätzlich:

* Alle können sich vermehren, nur Unterschied in der Wahrscheinlichkeit
* (Meist) Binäre Codierung der Gene
* Gene können entweder einzelne Bestandteile des Vektors oder auch mehrere Bestandteile zusammen sein.

Codierung:

Problem mit den Sprüngen im ASCII, deshalb verwendet man Gray-Code

### Schema Theorem und Building Block Hypothesis

Schema: Zeichenkette, die neben 1 und 0 noch beliebige Platzhalter enthält (-> Templatefunktion). Definiert das Set von Chromosomen, bei dem alle definierten Bits übereinstimmen.

Order of Scheme: Anzahl der definierten Bits

Definierte Länge: Abstand zwischen dem ersten und letzten definierten Bit

Anzahl der möglichen Schemata mit Länge L: 3L-1  
Anzahl der möglichen Chromosomen mit L: 2L

Auswirkungen der Selektion:

f(xi,t) Fitness von Chromosom xi zum Zeitpunkt t  
m(xi,t) Anzahl der Kopien des Chromosoms zum Zeitpunkt t  
p(xi)=Fitness/durchschnittliche\_Fitness Wahrscheinlichkeit, dass das Chromosom gewählt wird

Fitness eines Schemas: Nach der Anzahl der Kopien gewichteter Mittelwert der Fitness aller Chromosomen eines Sets

# Neuronale Netzwerke

* Algorithmen zur Verarbeitung von Daten, deren Prinzipien von biologischen Nervenzellen abgeschaut wurden.
* Ein mathematisches Neuron ist charakterisiert durch eine Eingabedimension n, einen Gewichtsvektor w und eine Aktivierungsfunktion f sowie einen Schwellwert/Bias Θ
* Man unterscheidet (im Wesentlichen) zwischen 2 Neuronenarten:
  + Skalarproduktbasierte Neuronen, bei denen die Funktion eine Summation aller Inputs und deren Gewichte ist (Also praktisch das Skalarprodukt aus w und Eingabevektor x transponiert)
  + Metrik-/Normbasiert: Funktion ist die Summe der Differenzen von Eingabe x und Wichtung w
* Wichtige Typen:
  + Perzeptron: x -> H(<x,w> - Θ) [Pyramidenzelle, skalarproduktbasiert)
  + Sigmoidaler Neuron:
  + Radiale-Basis-Funktion-Neuron:
* Ein Neuronales Netz ist ein Tupel (N,->,w, Θ,f,I,O)
  + N… Neuronen (Index der …)
  + -> … NxN – Verbindungen zw. Den Neuronen
  + w… Gewichtsvektor (wi,j) der Verbindungen
  + Θ… Schwellwerte
  + f… Vektor der Aktivierungsfunktionen
  + I… Eingabeneuronen (Inputs)
  + O… Ausgabeneutronen (Outputs)
* Neuronale Architektur: Ein neuronales Netz, bei dem Gewichte und Schwellenwerte noch nicht festgelegt sind, sondern Variablen darstellen
* Vernetzungsstrukturen:
  + Feed-Forward Network (FFN): (N,->) ist ein azyklischer Graph, bei denen zwischen I und O klar getrennt wird
    - Trennung der Neuronen N in verschiedene Layer (1 (Input) bis h (Output)), innerhalb dieser Layer kann nicht vernetzt werden, nur zu anderen Layern.
    - Alle Layer zwischen I und O als hidden Layer zusammengefasst, hidden Layer nicht zwingend
    - Verbindungen über mehrere Layer hinweg werden shortcuts genannt
  + Voll Vernetztes Multilayer FFN:
    - Sonderfall des FFN, Jede Schicht ist vollständig mit den Neuronen der nächsten Schicht vernetzt, ohne shortcuts
  + Rekurente Netze (vollst. Vernetzt)
    - -> NxN, also I=O=N
* Aktivierungsfunktionen
  + F=H, d.h. f= Summation der Gewichte mal der Inputwerte (Hearisidefunktion)
  + Bipolare Aktivierung: 1 für x>=0, 0 für x<0
  + Id(z)=z (Identische Funktion)
  + Semi-linear (=Input, Cutoff für Input > 1 oder < 0)
  + Sigmoid: sgd(z)=1/(1+exp(-z))
  + tanh(z)=(ez-e-z)/(ez+e-z)
  + Relu: Rectified linear with funktion (=Input, aber Cutoff für Input < 0)

Lernen in neuronalen Architekturen bedeutet Anpassung der Gewichte und Schwellwerte, um eine Lernaufgabe zu erfüllen.

Lernalgorithmus:

* Initialisierung mit zufälligen Gewichten und Schwellenwerten
* Solange mindestens ein Datenpunkt falsch klassifiziert wird: Anpassung der Gewichte in Richtung des Fehlers
* Der PLA konvergiert in endlich vielen Schritten, falls die Lernaufgabe lösbar ist.
* Eine Lernaufgabe heißt lösbar für ein Perzeptron, solange die Datenpunkte der beiden Klassen linear trennbar sind.
* Problem: Lösbarkeit des Problems ist nicht vorher bekannt und kann zu unendlich langen Durchläufen führen.
  + Heuristischer Ansatz: Abbruch zu einer Zeit k1
  + Dann Teilung der Datenmenge in richtig und falsch klassifiziert
  + Training eines neuen Perzeptrons mit der falsch klassifizierten Datenmenge
  + Wiederholung nach Bedarf
  + Schwierigkeit: Wann soll abgebrochen werden? Wie kann bestimmt werden, ob das erste Perzeptron schon gut gelernt hat?
  + Perzeptron-Pocket-Alg.: Für optimales Perzeptron im nicht-trennbaren Fall
    - Nach jeder Änderung der Gewichte Prüfung, wie viele Datenpunkte richtig klassifiziert wurden bzw. wie wenig Fehler gemacht worden und Speicherung der besten Gewichte
* Ansatz für Mehrklassenproblem:
  + Ebenfalls Heuristik
  + Auswahl einer Klasse gegen alle anderen Klassen und lernen der einzelnen Klasse
  + Dann neues Perzeptron für nächste Klasse
  + -> Kaskade von Perzeptrons

Lineare Perzeptrons

Aktivierungsfunktion linear, nicht mehr Heariside/Stufenfunktion

* Fehlerfunktion: SSE (Sum Squared Error): Mittelwert der quadrierten Abweichungen der Klassenlabel (y) von dem erhaltenen Funktionswert
* Bestimmung der Gewichte so, dass SSE minimal ist
  + -> Ableitung der Fehlerfunktion (SSE) nach den Gewichten = 0 an diesem Minimalpunkt
  + Trick: Bestimmung der lokalen Fehler aller Punkte und Summation dieser.
  + Le(xp,w)=(wTxp-yp)xp
  + (Updateschritt)
* SGDL für SSE: mit random(xP, yP)

## Feed-Forward-Netze

* Nutzung von sigmoiden Perzeptrons
  + Kann an Heariside-Funktion angenähert werden
  + Unendlich ableitbar
* Wie viele Hidden Layer? Wie viele Neuronen pro Layer?
  + Für lineare Perzeptronen nur eine hidden layer, da sich aus der Kombination von linearen Funktionen nur lineare Funktionen ergeben können
  + Theorem P. Werbos: Das Multilayer Perzeptron kann eine beliebige Funktionen f im reelen Raum beliebig genau approximieren, falls f mindestens 2 mal differenzierbar ist. Dafür reicht eine hidden layer. (Nur gültig, wenn Funktion nicht linear, differenzbar und beschränkt ist.
  + Anzahl der benötigten Neuronen im hidden Layer unbekannt und nicht vorher bestimmbar
  + Ausweg: Trainieren eines sehr großen Netzes, Berechnung der SSE und pruning von Neuronen, solange sich dadurch der SSE noch verbessert
  + Problem bei Netzen mit vielen Schichten: Nur die letzten layer vor dem Output werden ordentlich trainiert, frühere Gewichte kaum betroffen (Vanishing gradient)
    - Anstelle der sigmoidalen Funktion vielleicht lieber als Annäherung mit Relu arbeiten, leichter (schneller) ableitbar
    - Blockprinzip nach Hinton: Zerlegung des Netzwerkes in viele einzelblöcke aus input und „bottle neck“ (den verbundenen Neuronen), das einen Output der gleichen Dimension erzeugt, die einzeln optimiert werden. Genannt Autoencoder-Netzwerke

## Deep Networks

* Multilayerperzeptron mit vielen Hidden Layern
* Üblicherweise nach dem Blockprinzip von Hinton (s. ein paar Zeile weiter oben) trainiert
* Beispiel: **Bilderkennung/Klassifikation**
* Jede Schicht ist eine Art Filter, die die Daten transformieren, z. B. Faltung oder Pooling der Daten
* Am Ende entstehen Datenblöcke, deren räumlicher Zusammenhang (Reihenfolge der Eingabe der Inputs) absichtlich ignoriert wird
* Voraussetzung für gutes Deep Network:
  + Gut vortrainiertes Netz (z.B. google, Amazon)
  + Oder Extreeeem viele Daten