Previous topic
Spambase Problem: Strongly Typed GP

Next topic
One Fifth Rule

This Page
Show Source

Quick search
Go

Evolution Strategies Basics

Evolution strategies are special types of evolutionary computation algorithms where the mutation strength is learnt during the evolution. A first type of strategy (endogenous) includes directly the mutation strength for each attribute of an individual inside the individual. This mutation strength is subject to evolution similarly to the individual in a classic genetic algorithm. For more details, [Beyer2002] presents a very good introduction to evolution strategies.

In order to have this kind of evolution we'll need a type of individual that contains a `strategy` attribute. We'll also minimize the objective function, which gives the following classes creation.

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", array.array, typecode="d", fitness=creator.FitnessMin, strategy=N
creator.create("Strategy", array.array, typecode="d")
```

The initialization function for an evolution strategy is not defined by DEAP. The following generation function takes as argument the class of individual to instantiate, *icls*. It also takes the class of strategy to use as strategy, *scls*. The next arguments are the minimum and maximum values for the individual and strategy attributes. The strategy is added in the `strategy` member of the returned individual.

```
def generateES(icls, scls, size, imin, imax, smin, smax):
    ind = icls(random.uniform(imin, imax) for _ in range(size))
    ind.strategy = scls(random.uniform(smin, smax) for _ in range(size))
    return ind
```

This generation function is registered in the toolbox like any other initializer.

```
toolbox.register("individual", generateES, creator.Individual, creator.Strategy,
    IND_SIZE, MIN_VALUE, MAX_VALUE, MIN_STRATEGY, MAX_STRATEGY)
```

The strategy controls the standard deviation of the mutation. It is common to have a lower bound on the values so that the algorithm don't fall in exploitation only. This lower bound is added to the variation operator by the following decorator.

```
def checkStrategy(minstrategy):
    def decorator(func):
        def wrappper(*args, **kargs):
            children = func(*args, **kargs)
            for child in children:
                for i, s in enumerate(child.strategy):
                    if s < minstrategy:
                        child.strategy[i] = minstrategy
            return children
        return wrappper
    return decorator
```

The variation operators are decorated via the `decorate()` method of the toolbox and the evaluation function is taken from the `benchmarks` module.

```
toolbox.register("mate", tools.cxESBlend, alpha=0.1)
toolbox.register("mutate", tools.mutESLogNormal, c=1.0, indpb=0.03)

toolbox.decorate("mate", checkStrategy(MIN_STRATEGY))
toolbox.decorate("mutate", checkStrategy(MIN_STRATEGY))
```

From here, everything left to do is either write the algorithm or use one provided in `algorithms`. Here we will use the `eaMuCommaLambda()` algorithm.

```
def main():
    MU, LAMBDA = 10, 100
    pop = toolbox.population(n=MU)
    hof = tools.HallOfFame(1)
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("avg", numpy.mean)
    stats.register("std", numpy.std)
    stats.register("min", numpy.min)
    stats.register("max", numpy.max)

    pop, logbook = algorithms.eaMuCommaLambda(pop, toolbox, mu=MU, lambda_=LAMBDA,
        cxpb=0.6, mutpb=0.3, ngen=500, stats=stats, halloffame=hof)

    return pop, logbook, hof
```

The complete examples/es/fctmin.

[Beyer2002]	Beyer and Schwefel, 2002, Evolution strategies - A Comprehensive Introduction