Evolving Neural Networks

A tutorial on evolutionary algorithms

For the past decade, deep learning has dominated the machine learning landscape, often to the exclusion of other techniques. As a data scientist, it's important to have a variety of tools at your disposal, and one class of techniques that I feel is too-often overlooked is evolutionary algorithms. It may be a little self-indulgent to say this (myself being an intermediary product of a long-running evolutionary algorithm), but I find the evolutionary technique of design wonderful and in many cases more pragmatic than conventional machine learning techniques. In this tutorial we'll walk through training neural networks using an evolutionary algorithm, and we will use this technique to solve regression, classification, and policy problems. To do this, we will be using Python and the NumPy library.

Algorithm

Evolutionary Algorithms are based on the premise of natural selection, and include a five-step process:

1. Create an initial population of organisms. In our case, these will be neural networks.
2. Evaluate each organism based on some criteria. This is the organism's fitness score.

3. Take the best organisms from step two and have them reproduce. The offspring can either be identical copies of one parent (asexual reproduction) or a mosaic of two or more parents (sexual reproduction).

4. Mutate the offspring.

5. Take the new mutated offspring population and return to step two. Repeat until some condition is met (e.g. a fixed number of generations pass, a target fitness is achieved, etc.)

Step 1: The Organism

We will be using fully-connected feed-forward neural networks as our organisms in this tutorial.

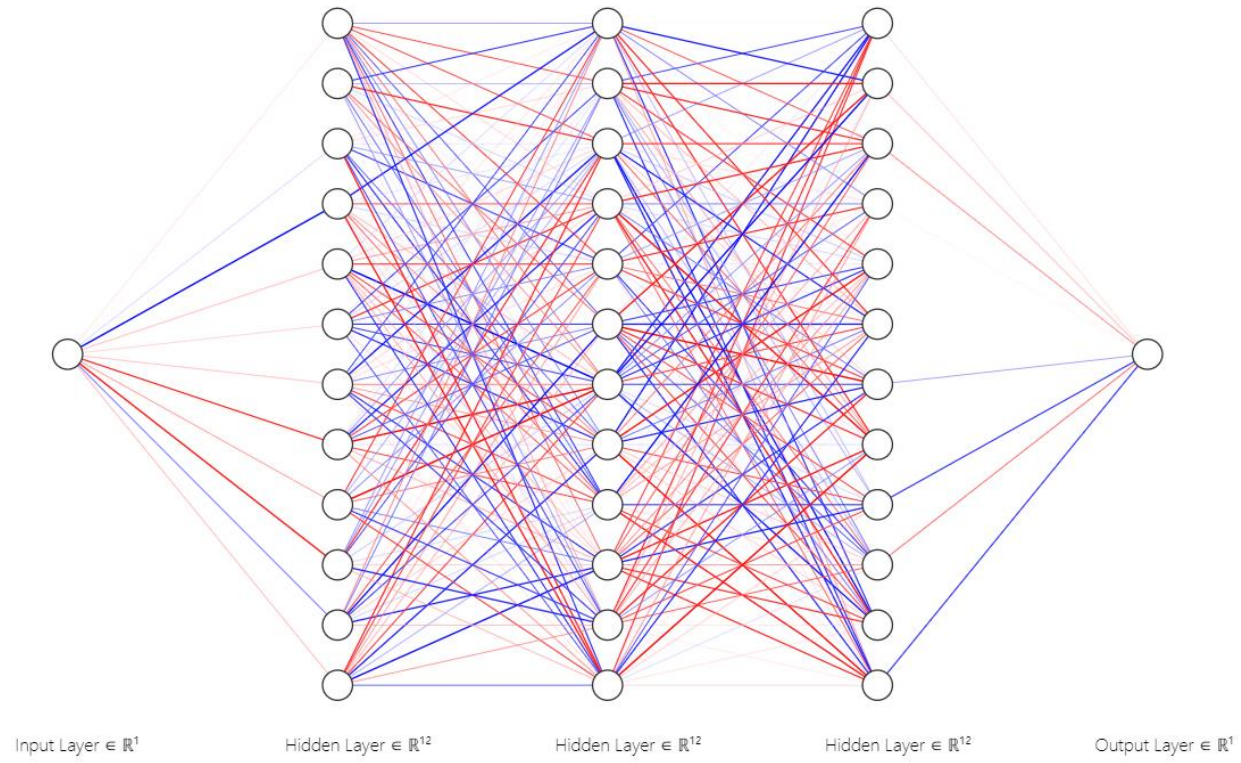


Fig 1: A fully-connected (dense) network with dimensions [1, 12, 12, 12, 1]

In designing our organisms, we have four guiding principles:

1. *We must have control over the input and output dimensions of the organisms*. Fundamentally, we are trying to evolve some function *f* that maps $\mathbb{R}^a \longrightarrow \mathbb{R}^b$, so *a* and *b* should be built into the organism. We accomplish this by parameterizing the input and output dimensions.
2. *We must have control over the output activation function*. The output of the organisms should be appropriate for the problem at hand. We accomplish this by parameterizing the output activation.
3. *We must have control over the complexity of the organisms*. The ideal organism should be just complex enough to evolve the target function and no more. We accomplish this by parameterizing the number of hidden layers and their dimensions. In a more advanced algorithm, this could be achieved by letting the organism evolve its own architecture and penalizing complexity with the fitness function.
4. *Organisms must be compatible for sexual reproduction*. Fortunately, the above principles ensure that this will be the case. All organisms will have the same architecture, so "exchanging genetic material" here means offspring will get some layer weights from momma and some from poppa.

Here's the implementation:

```
Class Organism():
    def __init__(self, dimensions, use_bias=True, output='softmax'):
        self.layers = []
        self.biases = []
        self.use_bias = use_bias
        self.output = self._activation(output)
        for i in range(len(dimensions)-1):
            shape = (dimensions[i], dimensions[i+1])
            std = np.sqrt(2 / sum(shape))
            layer = np.random.normal(0, std, shape)
            bias = np.random.normal(0, std, (1,  dimensions[i+1])) * use_bias
            self.layers.append(layer)
            self.biases.append(bias)

    def _activation(self, output):
        if output == 'softmax':
            return lambda X : np.exp(X) / np.sum(np.exp(X), axis=1).reshape(-1, 1)
        if output == 'sigmoid':
            return lambda X : (1 / (1 + np.exp(-X)))
        if output == 'linear':
            return lambda X : X

    def predict(self, X):
        if not X.ndim == 2:
            raise ValueError(f'Input has {X.ndim} dimensions, expected 2')
        if not X.shape[1] == self.layers[0].shape[0]:
            raise ValueError(f'Input has {X.shape[1]} features, expected
{self.layers[0].shape[0]}')
        for index, (layer, bias) in enumerate(zip(self.layers, self.biases)):
            X = X @ layer + np.ones((X.shape[0], 1)) @ bias
            if index == len(self.layers) - 1:
                X = self.output(X) # output activation
            else:
                X = np.clip(X, 0, np.inf)  # ReLU
        return X
```

Snippet 1: The partial Organism class and methods for creating and using it

In the _init_ function, we set up the network. The parameter dimensions is a list of layer dimensions, where the first is the width of the input, the last is the width of the output, and all

others are hidden dimensions. The `__init__` function iterates through these *n* dimensions to create *n-1* weight matrices using Glorot Normal initialization, which are stored as `layers`. If bias is enabled, a non-zero bias vector is also stored for each layer. The model uses ReLU activation for all internal layers. The output activation is specified using the `output` parameter, and can be "softmax", "sigmoid", or "linear", as implemented in the `_activation` method.

The `predict` method repeatedly applies ReLU and matrix multiplication to the input matrix. As an example, if the network were made with three hidden layers and softmax output, then the network would apply the function

$$\mathbf{Y} = softmax(ReLU(ReLU(ReLU(\mathbf{X}\mathbf{W_1})\mathbf{W_2})\mathbf{W_3})\mathbf{W_4})$$

Were $\mathbf{X}$ is the input matrix, $\mathbf{W}_i$ is the weight matrix for layer i, and $\mathbf{Y}$ is the output.

We will be creating a population of these organisms, all with the same architecture but each with different, random weights.

Step 2: Organism Fitness

Measuring how well an organism performed is the crux of evolutionary algorithm design, and "good performance" varies from task to task. In regression, the fitness score may be the negative mean squared error. In classification, it may be classification accuracy. In playing Pac-Man, it may be the number of pellets gobbled before dying. Because fitness functions are task-specific, we will wait to explore them until the applications section below. For the time being, it suffices to understand that we require a function `scoring_function` which accepts an organism as input and returns a real number output, where bigger is better.

Step 3: Reproduction

The reproduction step has itself two steps: parent selection and progeny creation. Each new organism needs *k* parents. In asexual reproduction *k* is 1, while in sexual reproduction k is 2 or more. Therefore, to generate a new generation of *n* organisms, we must select *n* sets of *k* organisms from the previous generation; deciding which organism(s) will parent each child should be done on the basis of their fitness score, where the fittest organisms should produce the most offspring. There are many ways to do this, some of which include:

1. Select each parent uniformly from the top 10% of organisms.
2. Order the organisms from best to worst, then select the index of each parent by sampling from the exponential distribution.
3. Apply the softmax function to each organism's score to create a probability of selection for each organism, then sample from that distribution.

I chose a compromise between methods one and two, where the top 10% of organisms were selected as the first parent of a child ten times each and the second parent was chosen randomly using the exponential distribution, as above. I also enforced that a clone of the best-performing organism of a given generation be included in the next generation. Here is the relevant code:

```
class
Ecosystem():
                # [Some code removed here]

                def generation(self, repeats=1, keep_best=True):
                    rewards = rewards = [np.mean([self.scoring_function(x) for _ in
            range(repeats)]) for x in self.population]
                    self.population = [self.population[x] for x in np.argsort(rewards)[::-1]]
                    new_population = []
                    for i in range(self.population_size):
                        parent_1_idx = i % self.holdout
                        if self.mating:
                            parent_2_idx = min(self.population_size - 1,
            int(np.random.exponential(self.holdout)))
                        else:
                            parent_2_idx = parent_1_idx
                        offspring =
            self.population[parent_1_idx].mate(self.population[parent_2_idx])
                        new_population.append(offspring)
                    if keep_best:
                        new_population[-1] = self.population[0] # Ensure best organism
            survives
                    self.population = new_population
```

Snippet 2: The partial Ecosystem class and the method for simulating a generation

As you can see, the organisms are scored and sorted in lines 5 and 6. Then, population_size new organisms are created in the loop on line 8. Parent one is selected from the top 10% on line 9 (holdout is the number of organisms that are guaranteed progeny, here being population_size//10). When mating is enabled, parent two is chosen using the exponential distribution with λ=holdout. When mating is disabled, parent one mates with itself and the child is a clone.

Once *n* pairs of parents have been chosen, the progeny can be created by randomly combining traits from each pair. In our case, those traits are weights in the neural network layers. Here is the Organism class's method for progeny creation:

```
class Organism():
# [Some code removed here]
    def mate(self, other, mutate=True):
        if self.use_bias != other.use_bias:
            raise ValueError('Both parents must use bias or not use bias')
        if not len(self.layers) == len(other.layers):
            raise ValueError('Both parents must have same number of layers')
        if not all(self.layers[x].shape == other.layers[x].shape for x in range(len(self.layers))):
            raise ValueError('Both parents must have same shape')

        child = copy.deepcopy(self)
        for i in range(len(child.layers)):
            pass_on = np.random.rand(1, child.layers[i].shape[1]) < 0.5
            child.layers[i] = pass_on * self.layers[i] + ~pass_on * other.layers[i]
            child.biases[i] = pass_on * self.biases[i] + ~pass_on * other.biases[i]
        if mutate:
            child.mutate()
        return child
```

Snippet 3: The partial Organism class and method for mating two organisms together

As you can see, the mate method confirms that both parents are compatible with each other, then randomly selects a parent from whom to inherit each column vector in each weight matrix for the child organism.

Step 4: Mutation

After each child organism is produced, it is subject to mutation (snippet 3, line 18). In this tutorial, the mutation step is realized as the addition of Gaussian noise to each weight in the network. We do not change the activations or architecture of the network here, although a more advanced evolutionary algorithm could certainly do so by adding or removing nodes in the hidden layers. Here is the code:

```
class Organism():
# [Some code removed here]
    def mutate(self, stdev=0.03):
        for i in range(len(self.layers)):
            self.layers[i] += np.random.normal(0, stdev, self.layers[i].shape)
            if self.use_bias:
                self.biases[i] += np.random.normal(0, stdev, self.biases[i].shape)
```

Snippet 4: the partial Organism class and mutation method

Step 5: Repeat

This is hardly a step; all that remains to be done is check if some condition is met, and to return to step two if it has not. I chose to run the algorithm for a fixed number of generations, but stopping when the fitness score hits a desired threshold or after some number of generations pass without improvement are also good choices. Here is the full code for the Organism and Ecosystem classes:

```
import copy
import numpy as np

class Organism():
    def __init__(self, dimensions, use_bias=True, output='softmax'):
        self.layers = []
        self.biases = []
        self.use_bias = use_bias
        self.output = self._activation(output)
        for i in range(len(dimensions)-1):
            shape = (dimensions[i], dimensions[i+1])
            std = np.sqrt(2 / sum(shape))
            layer = np.random.normal(0, std, shape)
            bias = np.random.normal(0, std, (1,  dimensions[i+1])) * use_bias
            self.layers.append(layer)
            self.biases.append(bias)

    def _activation(self, output):
        if output == 'softmax':
            return lambda X : np.exp(X) / np.sum(np.exp(X), axis=1).reshape(-1, 1)
```

```
    if output == 'sigmoid':
        return lambda X : (1 / (1 + np.exp(-X)))
    if output == 'linear':
        return lambda X : X

def predict(self, X):
    if not X.ndim == 2:
        raise ValueError(f'Input has {X.ndim} dimensions, expected 2')
    if not X.shape[1] == self.layers[0].shape[0]:
        raise ValueError(f'Input has {X.shape[1]} features, expected {self.layers[0].shape[0]}')
    for index, (layer, bias) in enumerate(zip(self.layers, self.biases)):
        X = X @ layer + np.ones((X.shape[0], 1)) @ bias
        if index == len(self.layers) - 1:
            X = self.output(X) # output activation
        else:
            X = np.clip(X, 0, np.inf)  # ReLU

    return X

def predict_choice(self, X, deterministic=True):
    probabilities = self.predict(X)
    if deterministic:
        return np.argmax(probabilities, axis=1).reshape((-1, 1))
    if any(np.sum(probabilities, axis=1) != 1):
        raise ValueError(f'Output values must sum to 1 to use deterministic=False')
    if any(probabilities < 0):
        raise ValueError(f'Output values cannot be negative to use deterministic=False')
    choices = np.zeros(X.shape[0])
    for i in range(X.shape[0]):
        U = np.random.rand(X.shape[0])
        c = 0
        while U > probabilities[i, c]:
            U -= probabilities[i, c]
            c += 1
        else:
            choices[i] = c
    return choices.reshape((-1,1))

def mutate(self, stdev=0.03):
    for i in range(len(self.layers)):
        self.layers[i] += np.random.normal(0, stdev, self.layers[i].shape)
        if self.use_bias:
            self.biases[i] += np.random.normal(0, stdev, self.biases[i].shape)
```

```
    def mate(self, other, mutate=True):
        if self.use_bias != other.use_bias:
            raise ValueError('Both parents must use bias or not use bias')
        if not len(self.layers) == len(other.layers):
            raise ValueError('Both parents must have same number of layers')
        if not all(self.layers[x].shape == other.layers[x].shape for x in range(len(self.layers))):
            raise ValueError('Both parents must have same shape')

        child = copy.deepcopy(self)
        for i in range(len(child.layers)):
            pass_on = np.random.rand(1, child.layers[i].shape[1]) < 0.5
            child.layers[i] = pass_on * self.layers[i] + ~pass_on * other.layers[i]
            child.biases[i] = pass_on * self.biases[i] + ~pass_on * other.biases[i]
        if mutate:
            child.mutate()
        return child


class Ecosystem():
    def __init__(self, original_f, scoring_function, population_size=100, holdout='sqrt', mating=True):
        """
        original_f must be a function to produce Organisms, used for the original population
        scoring_function must be a function which accepts an Organism as input and returns a float
        """
        self.population_size = population_size=100
        self.population = [original_f() for _ in range(population_size)]
        self.scoring_function = scoring_function
        if holdout == 'sqrt':
            self.holdout = max(1, int(np.sqrt(population_size)))
        elif holdout == 'log':
            self.holdout = max(1, int(np.log(population_size)))
        elif holdout > 0 and holdout < 1:
            self.holdout = max(1, int(holdout * population_size))
        else:
            self.holdout = max(1, int(holdout))
        self.mating = True

    def generation(self, repeats=1, keep_best=True):
        rewards = [np.mean([self.scoring_function(x) for _ in range(repeats)]) for x in self.population]
        self.population = [self.population[x] for x in np.argsort(rewards)[::-1]]
        new_population = []
        for i in range(self.population_size):
```

```
            parent_1_idx = i % self.holdout
            if self.mating:
                parent_2_idx = min(self.population_size - 1, int(np.random.exponential(self.holdout)))
            else:
                parent_2_idx = parent_1_idx
            offspring = self.population[parent_1_idx].mate(self.population[parent_2_idx])
            new_population.append(offspring)
        if keep_best:
            new_population[-1] = self.population[0] # Ensure best organism survives
        self.population = new_population

    def get_best_organism(self, repeats=1, include_reward=False):
        rewards = [np.mean(self.scoring_function(x)) for _ in range(repeats) for x in self.population]
        if include_reward:
            best = np.argsort(rewards)[-1]
            return self.population[best], rewards[best]
        else:
            return self.population[np.argsort(rewards)[-1]]
```

Snippet 5: The Organism and Ecosystem classes in all their splendor

---

Application

"Cool. Why should I care?" — you, probably

Regression

Let's apply this process to a regression problem. We will be evolving an organism to approximate $sin(\tau x)$ in the domain $x \in [0, 1]$. Obviously this is a trivial problem, and that's why we're using it as an example. So how should we design our organisms?

1. This is a function from $\mathbb{R}^1$ to $\mathbb{R}^1$, so both the input and output dimensions will be 1.
2. The range of sine is [-1, 1], so the output activation will be linear.
3. The fitness function will be the negative mean squared error of the organism's output.
4. We will use three hidden layers with a width of 16, because that is likely complex enough for this simple task.

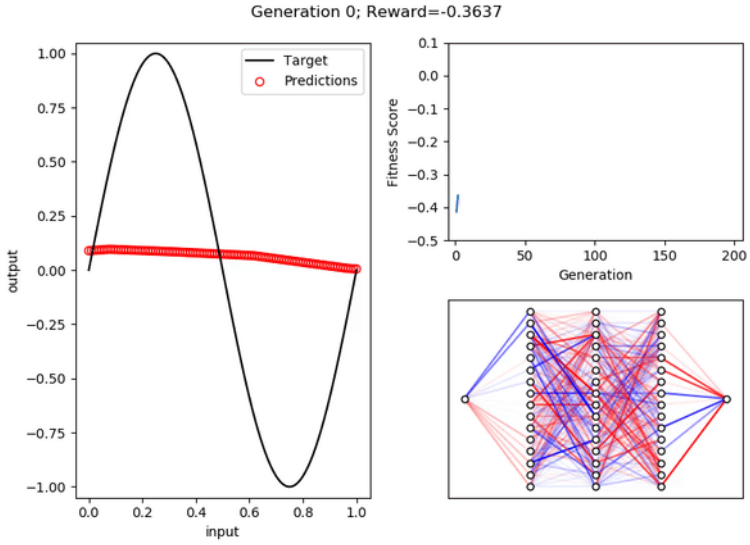
Here's the code:

```
# The function to create the initial population
organism_creator = lambda : Organism([1, 16, 16, 16, 1], output='linear')
# The function we are trying to learn. numpy doesn't have tau...
true_function = lambda x : np.sin(2 * np.pi * x) #
# The loss function, mean squared error, will serve as the negative fitness
loss_function = lambda y_true, y_estimate : np.mean((y_true - y_estimate)**2)

def simulate_and_evaluate(organism, replicates=1):
    """
    Randomly generate `replicates` samples in [0,1],
    use the organism to predict their corresponding value,
    and return the fitness score of the organism
```

```
    """
    X = np.random.random((replicates, 1))
    predictions = organism.predict(X)
    loss = loss_function(true_function(X), predictions)
    return -loss

# Ecosystem requires a function that maps an organism to a real number fitness
scoring_function = lambda organism : simulate_and_evaluate(organism, replicates=100)
# Create the ecosystem
ecosystem = Ecosystem(organism_creator, scoring_function,
                      population_size=100, holdout=0.1, mating=True)
# Save the fitness score of the best organism in each generation
best_organism_scores = [ecosystem.get_best_organism(include_reward=True)[1]]
generations = 201
for i in range(generations):
    ecosystem.generation()
    this_generation_best = ecosystem.get_best_organism(include_reward=True)
    best_organism_scores.append(this_generation_best[1])
    # [Visualization code omitted...]
```

Snippet 6: Using an Ecosystem of Organisms to evolve the sine function

Results:



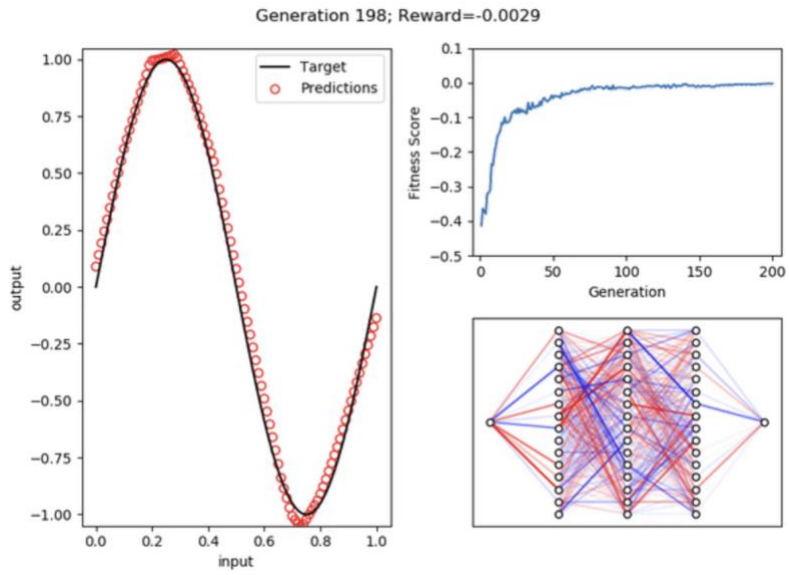


Fig 2: The best organism from each of 200 generations of evolution. Predictions vs. target (left), best fitness per generation (top), and network visualization (bottom).

As you can see, after 200 generations an organism evolved that produces a pretty good approximation of the sine function within the domain *[0, 1]*.

Classification

Let's see how the technique fares against a classification problem. Here we use the iris dataset. For the uninitiated, (welcome!) the iris dataset is a decades-old set of sepal and petal lengths and widths of 150 iris flowers. Each flower belongs to one of three species, and the task is to classify the flower by species based on its four measurements. So what does this mean for our organisms?

1. Each flower has four real-valued measurements, so our input dimension will be four.
2. There are three different classes, so our output dimension will be three.

3. Because we are selecting one class, our output activation will be softmax.
4. We will hold out one third of the data from training for testing. This is to ensure that the organism isn't memorizing the answers.

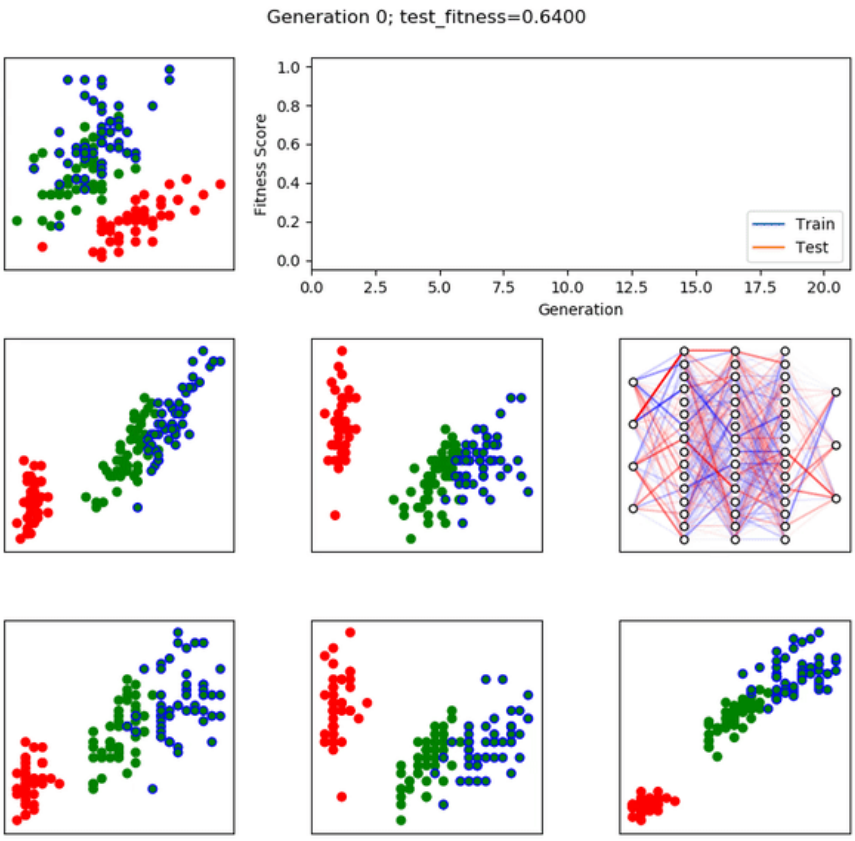
Here's the code:

```
#
Load
the
data
    df = pd.read_csv('iris.csv')
    # Enumerate the classes
    unique_classes = sorted(list(set(df['variety'])))
    class_number = {y : x for x,y in enumerate(unique_classes)}
    df['variety'] = [class_number[x] for x in df['variety']]
    # Convert to numpy array and standardize the features
    data_X = df[['sepal.length', 'sepal.width', 'petal.length', 'petal.width']].values
    data_Y = df[['variety']].values
    data_X = data_X - np.min(data_X, axis=0)
    data_X = data_X / np.max(data_X, axis=0)

    # The function to create the initial population
    organism_creator = lambda : Organism([4, 16, 16, 16, 3], output='softmax')
    # The fitness function will be the classification accuracy
    fitness_function = lambda y_true, y_estimate : (
            np.mean(y_true.flatten() == np.argmax(y_estimate, axis=1)))

    def simulate_and_evaluate(organism, indices):
        """
        Predict the probabilities of each class and return the fitness
        Indices is the list of indices to use in evaluation
        """
        predictions = organism.predict(data_X[indices])
        return fitness_function(data_Y[indices], predictions)

    # Separate training and test data using random indices
    indices = np.arange(len(df))
    np.random.shuffle(indices)
    indices_train, indices_test = indices[:100], indices[100:]
```

```
# Create the scoring function and build the ecosystem
scoring_function = lambda organism : simulate_and_evaluate(organism,
indices=indices_train)
ecosystem = Ecosystem(organism_creator, scoring_function,
                      population_size=100, holdout=0.1, mating=True)

# Run 20 generations
generations = 20
for i in range(generations):
    ecosystem.generation()
    this_generation_best = ecosystem.get_best_organism(include_reward=True)
    # [Visualization code omitted...]
```

Snippet 7: Using an Ecosystem of Organisms to evolve the a flower classifier

Results:



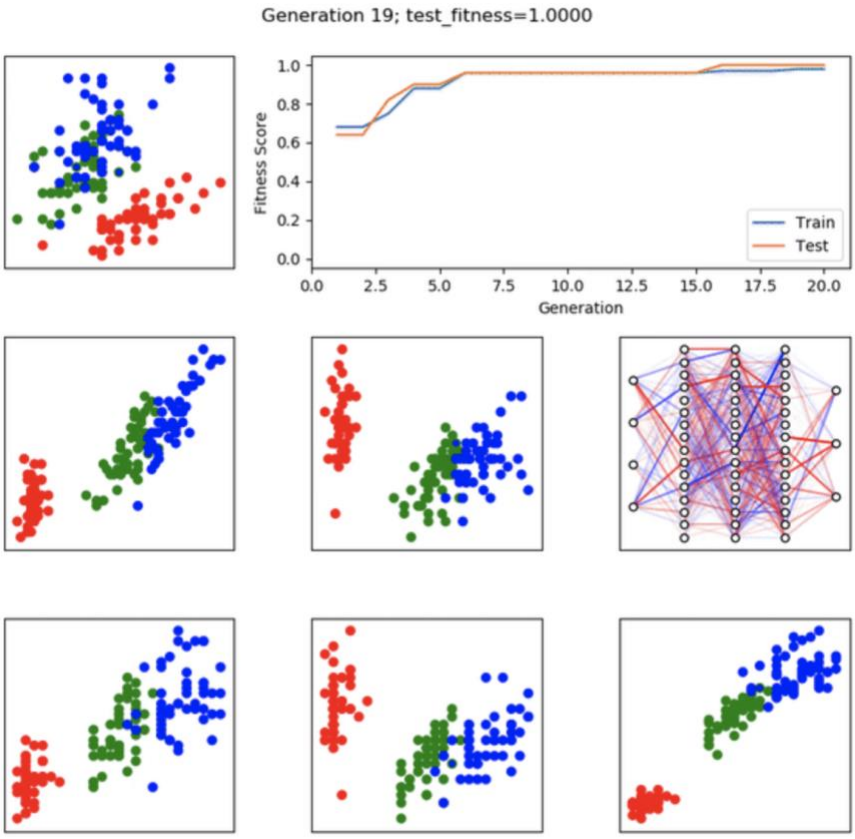


Fig 3: The best organism from each of 20 generations.

Each scatterplot represents a pair of different features plotted against each other. Each point is a flower, where the outline colour represents the true class and the fill colour represents the predicted class. The training and testing fitness of the best organism in each generation are shown in the line plot, and the best organism is illustrated in the centre-right panel. In only 15 generations, the ecosystem evolves a network that correctly classifies the entire testing set!

Policy

Let's try evolving an organism capable of playing OpenAI's CartPole "game", included in the AI Gym package. Here's an example from YouTube:

Source: Morvan on youtube: https://youtu.be/46wjA6dqxOM

From the official documentation:

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

At each timestep of the game, the gamestate is represented as a vector of four real numbers, corresponding to the horizontal position of the cart, the horizontal velocity of the cart, the angle of the pole, and the angular velocity of the pole. The player must then apply an action (applying a force of +1 or -1) and the game advances one timestep. The player has "won" the game if they are able to avoid losing for 500 timesteps.

So what does this mean for our ecosystem and organisms?

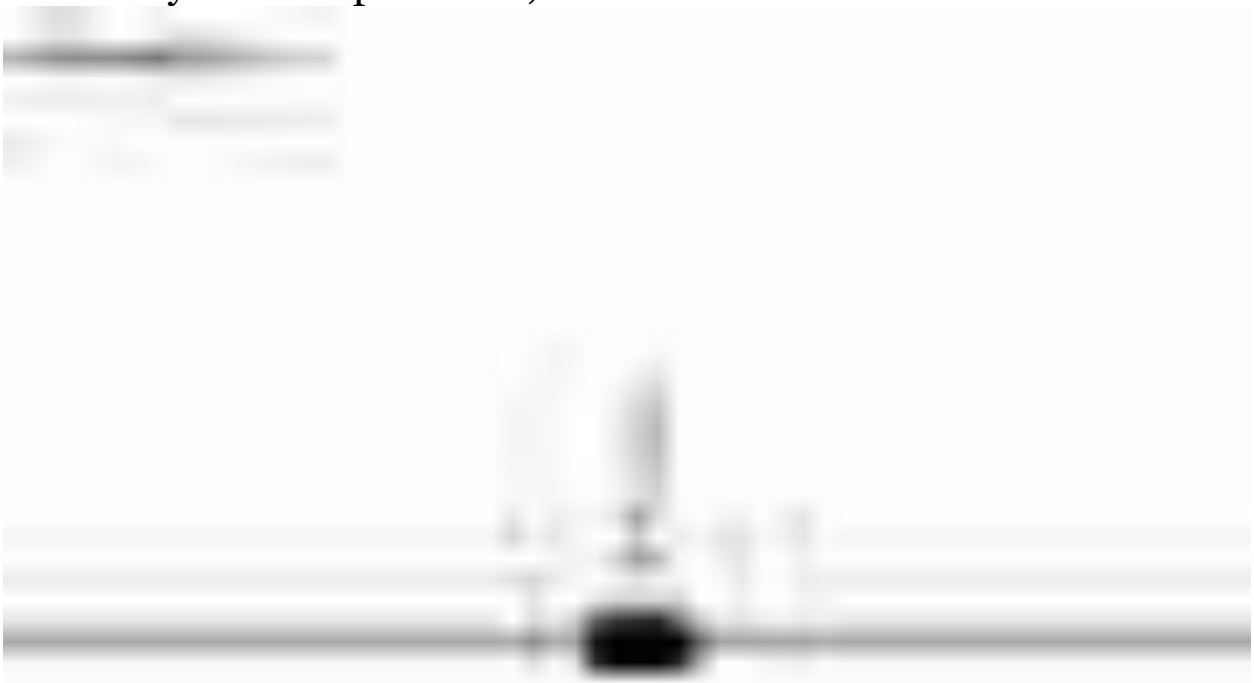
1. Each timestep has four real-valued measurements, so our input dimension will be four.
2. There are two possible actions, so our output dimension will be two.
3. Because we are selecting one action, our output activation will be softmax.
4. The fitness function is the number of timesteps survived. For robustness, we will run the simulation 5 times and take the average number of timesteps survived.

Here's the code:

```
# Set up the environment and collect the observation space and action space sizes
observation_space = env.observation_space.shape[0]
action_space = env.action_space.n

# The function for creating the initial population
organism_creator = lambda : Organism([observation_space, 16, 16, 16, action_space],
output='softmax')

def simulate_and_evaluate(organism, trials=1):
    """
    Run the environment `trials` times, using the organism as the agent
    Return the average number of timesteps survived
    """
    fitness = 0
    for i in range(trials):
        state = env.reset() # Get the initial state
        while True:
            fitness += 1
            action = organism.predict(state.reshape((1,-1)))
            action = np.argmax(action.flatten())
            state, reward, terminal, info = env.step(action)
            if terminal: # break if the agent wins or loses
                break
    return fitness / trials

# Create the scoring function and build the ecosystem
scoring_function = lambda organism : simulate_and_evaluate(organism, trials=5)
ecosystem = Ecosystem(organism_creator, scoring_function,
                      population_size=100, holdout=0.1, mating=True)

generations = 200
for i in range(generations):
    ecosystem.generation()
    # [Visualization code omitted]
    if this_generation_best[1] == 500: # Stop when an organism achieves a perfect score
        break
```

Snippet 8: Using an Ecosystem of Organisms to evolve a CartPole player

Is it really that simple? Well, here are the results:



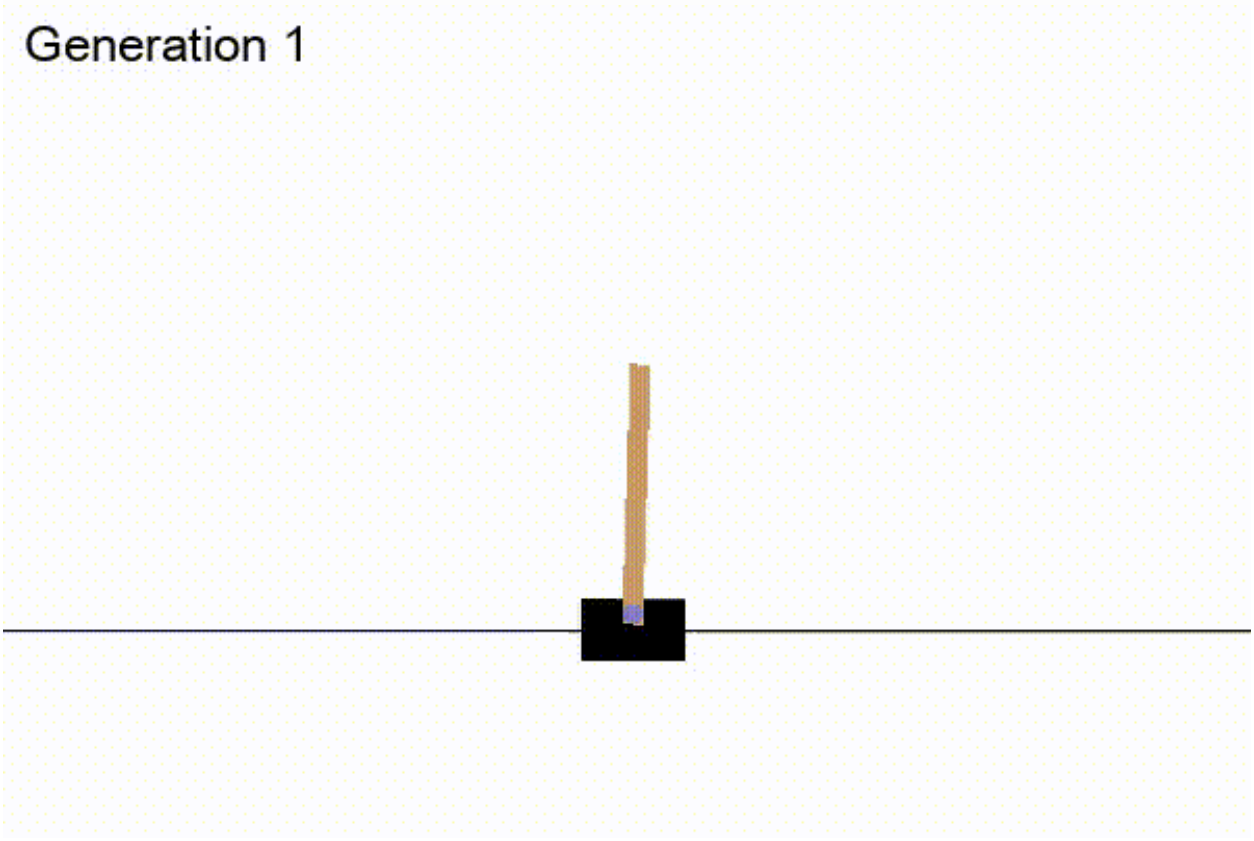


Fig 4: The best organism in each generation playing CartPole

That's right, the ecosystem evolved an organism that beat the game in only 6 generations. I was a little shocked myself; you'll notice on line 31 that I expected it would take a lot more than that. I had intended to show the fitness over time just like in the other two problems, but I won't bother you with it since six data points don't make for an interesting figure.

Conclusion

Evolutionary algorithms are intuitive and effective. Although the above examples were fairly simple, they demonstrate the evolutionary algorithms are applicable to a broad class of problems. Evolutionary algorithms are particularly useful in situations where the fitness of the solution is measurable, but not in a way that easily allows for backward propagation of the gradient, as in the CartPole game. For these reasons, it's well-worth adding evolutionary algorithms to your machine learning toolbox. Stay tuned for my next article, in which I'll explore fitness functions at length and how reasonable assumptions can lead to hilariously disastrous results.