```python
"""Script for computing the one-step error probability."""

import numpy as np
from hopfield import DeterministicHopfieldNetwork
from pattern_utilities import generate_n_random_patterns, print_pattern


def main():
    n_neurons = 120
    n_patterns_vector = [12, 24, 48, 70, 100, 120]
    # Set diagonal_weights_rule to "non-zero" in second task
    diagonal_weights_rule = "zero"
    n_iterations = int(1e5)
    one_step_error_probability = np.zeros(len(n_patterns_vector))

    for n_patterns_i, n_patterns in enumerate(n_patterns_vector):
        n_errors = 0
        for _ in range(n_iterations):
            network = DeterministicHopfieldNetwork()

            patterns = generate_n_random_patterns(n_patterns, n_neurons)
            network.set_patterns(patterns)
            network.set_diagonal_weights_rule(diagonal_weights_rule)
            network.generate_weights()

            pattern_to_feed_index = np.random.randint(n_patterns)
            original_pattern = patterns[pattern_to_feed_index, :]
            neuron_to_update_index = np.random.randint(n_neurons)

            updated_pattern = network.update_neuron(
                original_pattern,
                neuron_to_update_index
                )
            updated_neuron = updated_pattern[neuron_to_update_index]
            original_neuron = original_pattern[neuron_to_update_index]
            updated_neuron = updated_pattern[neuron_to_update_index]

            if original_neuron != updated_neuron:
                n_errors += 1

        one_step_error_probability[n_patterns_i] = n_errors/n_iterations

    print_pattern(one_step_error_probability)


if __name__ == "__main__":
    main()
```

```python
"""Implementation of a Hopfield network using Hebb's rule."""

from abc import ABC, abstractmethod
import numpy as np
from scipy import stats


class HopfieldNetwork(ABC):
    """Abstract Hopfield net using Hebb's rule to compute weights."""

    def set_diagonal_weights_rule(self, diagonal_weights_rule):
        """Specify if the diagonal weights should be zero or not.

        diagonal_weights_rule must equal "zero" or "non-zero".
        """
        if diagonal_weights_rule == "zero":
            self.diagonal_weights_equal_zero = True
        elif diagonal_weights_rule == "non-zero":
            self.diagonal_weights_equal_zero = False
        else:
            raise KeyError(
                diagonal_weights_rule
                + " not a valid diagonal_weights_rule"
                )

    def set_patterns(self, patterns):
        """Specify the stored patterns.

        patterns must be a 1D- or 2D-array where each row is a pattern.
        """
        self.patterns = patterns

    def generate_weights(self):
        """Generate the weights for the network."""
        _, n_neurons = self.patterns.shape
        self.weights = np.zeros((n_neurons, n_neurons))
        for pattern in self.patterns:
            self.weights += np.outer(pattern, pattern)
        self.weights /= n_neurons

        if self.diagonal_weights_equal_zero:
            np.fill_diagonal(self.weights, 0)

    def update_random_neuron(self, pattern):
        n_neurons = pattern.shape
        neuron_index = np.random.randint(n_neurons)
        return self.update_neuron(pattern, neuron_index)
```

```python

    def update_neuron(self, pattern, neuron_index):
        """Returns updated pattern after update of neuron_index."""
        weights_i = self.weights[neuron_index, :]
        local_field = np.inner(weights_i, pattern)
        updated_neuron = self.get_state_of_local_field(local_field)
        updated_pattern = pattern.copy()
        updated_pattern[neuron_index] = updated_neuron
        return updated_pattern

    @abstractmethod
    def get_state_of_local_field(self, local_field):
        pass


class DeterministicHopfieldNetwork(HopfieldNetwork):
    """Concrete Hopfield net with deterministic updating."""

    def get_state_of_local_field(self, local_field):
        """"Update rule when updating a neuron."""
        return 1 if local_field >= 0 else -1


class StochasticHopfieldNetwork(HopfieldNetwork):
    """Concrete Hopfield net with stochastic updating."""

    def get_state_of_local_field(self, local_field):
        """"Update rule when updating a neuron.

        Returns 1 with probability
        1 / (1+exp(-2*noise_parameter*local_field),
        else -1.
        """
        p = 1/(1 + np.exp(-2*self.noise_parameter*local_field))
        rand = stats.bernoulli.rvs(p)
        return 1 if rand else -1

    def set_noise_parameter(self, noise_parameter):
        self.noise_parameter = noise_parameter
```

```python
"""Functions for working with patterns.

A pattern is defined as a 1D-array and several patterns are stored as
a 2D-array, where each row corresponds to one pattern. Each element can
take the values -1 or +1 only.
"""

import numpy as np
```

```python
from scipy import stats


def generate_n_random_patterns(n_patterns, n_neurons):
    """Returns n_patterns random patterns, each of length n_neurons.

    If several patterns are generated, each row corresponds to one
    pattern.
    """
    random_0s_and_1s = stats.bernoulli.rvs(0.5, size=(n_patterns, n_neurons))
    random_minus_1s_and_1s = 2*random_0s_and_1s - 1
    return random_minus_1s_and_1s


def get_index_of_equal_pattern(pattern_to_match, patterns):
    """Returns the row index of patterns corresponding to the pattern
    that is equal to pattern_to_match.

    Returns 1 if no matching pattern is found.
    """
    for index, pattern in enumerate(patterns):
        n_different_neurons = get_n_different_neurons(pattern_to_match, pattern)
        if n_different_neurons == 0:
            return index
    return -1


def get_n_different_neurons(pattern1, pattern2):
    return sum(pattern1 != pattern2)


def vector_to_typewriter(vector, n_columns):
    """Returns 2D-array of vector.

    The first row in the returned array consists of the first n_columns
    elements in vector and so on.
    """
    return np.reshape(vector, (-1, n_columns))


def print_typewriter_pattern(pattern, n_columns):
    """Prints a pattern in a typewriter scheme."""
    print_pattern(vector_to_typewriter(pattern, n_columns))


def print_pattern(pattern):
    np.set_printoptions(formatter={"float_kind": lambda x: "%.4f" % x})
    print(repr(pattern), sep=", ")
```