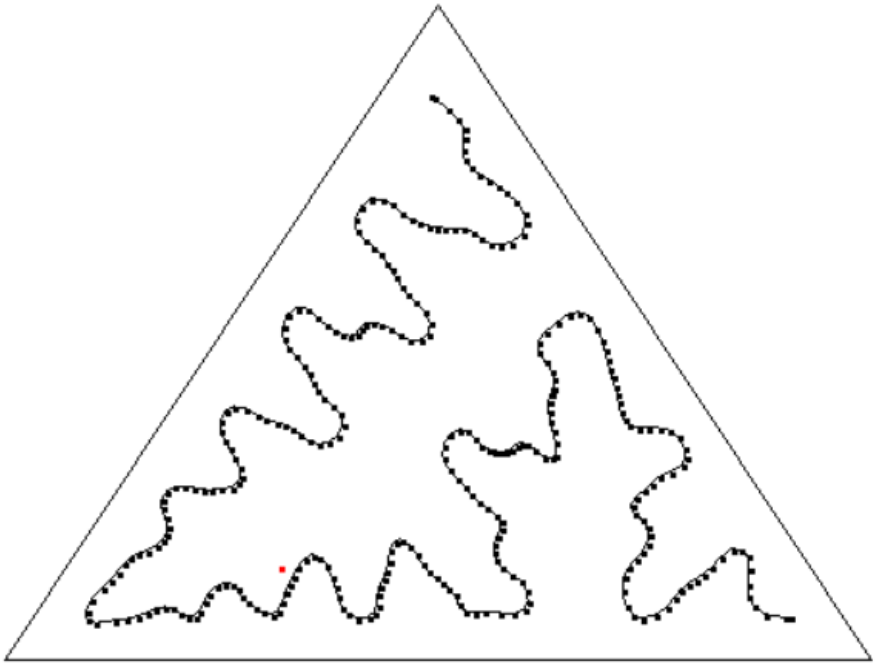
Self Organizing Map (SOM)

Kamil Pudlewski

1. Zadanie realizowane przez aplikację i opis parametrów

Aplikacja którą stworzyłem realizuje zagadnienie map samoorganizujących, które ma za zadanie rozłożyć neurony na siatce wybranej figury. Celem jest uzyskanie efektu krzywych Peano, czyli takich które wypełnia całą figurę. Przykład takiej krzywej został przedstawiony na rysunku 1.1, umieszczonym poniżej. Im lepsze rozłożenie punktów na figurze, tym lepsza jest skuteczność sieci.



Rysunek 1.1 – Przykład krzywych Peano w trójkącie

Algorytm działa tak że sieć startuje z pozycji losowo umieszczonych neuronów w figurze. Następnie zostaje wylosowany punkt znajdujący się wewnątrz figury, na którego podstawie przestawiają się kolejne neurony. Ta czynność jest powtarzana tak

długo, aż jak zostanie wykonana ustalona liczba iteracji. Na początku neurony będą rozkładać się po figurze, aż w końcu powstaną krzywe Peano, o ile parametry startowe sieci zostały odpowiednio dobrane. Więcej o nich zostało napisane w dalszej części raportu.

Aplikacja posiada interfejs graficzny dzięki czemu w łatwy sposób można ustawiać kształty, na których ma pracować sieć oraz dostosować parametry jej działania. Pierwszą opcją do wyboru jest ilość neuronów które mają wypełnić figurę, można wybrać ich dowolną liczbę. Trzeba jednak pamiętać że im więcej się ich ustawi tym uzyskana siatka końcowa będzie gęściejsza. Z drugiej strony za mała ich ilość sprawi że rozciągnięte neurony nie wypełnią się na wybranej figurze zbyt dobrze.

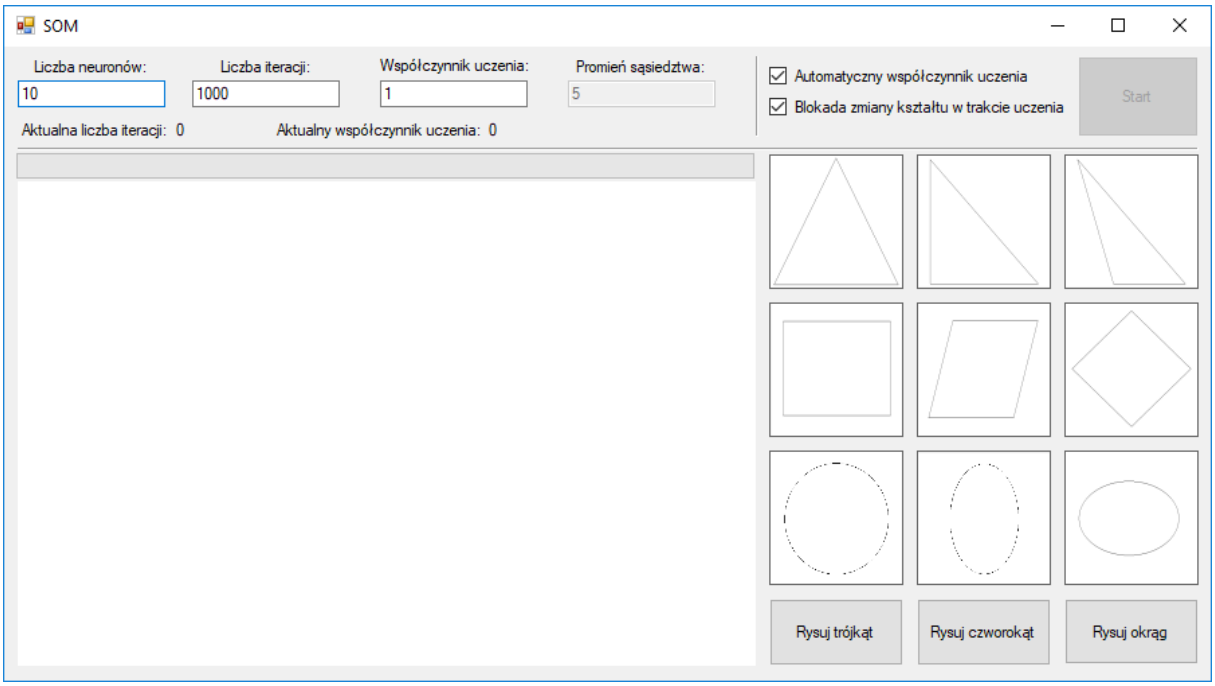
Kolejnym parametrem do ustawienia jest liczba iteracji, określa ona długość trwania układania się sieci neuronów. Duża ilość iteracji pozwala się lepiej dopasować do figury, dzięki czemu można uzyskać wspomniany wcześniej efekt krzywych Peano. Wadą dużej liczby iteracji jest czas trwania wykonywania algorytmu. Z kolei za mała ilość powtórzeń nie doprowadzi neuronów do rozłożenia się po całym kształcie figury.

Współczynnik uczenia odpowiada za wartość z jaką odległością będą przesuwały się neurony. Z każdą kolejną iteracją liczba ta się zmniejsza. W zależności od ustawionej liczby powtórzeń wartość ta może zmaleć do bardzo małych liczb. Dzięki temu w końcowej fazie rozkładu neurony wyginają się tak, by zająć jak największą powierzchnię, nie robiąc tym samym bardzo dużych skoków. Gdyby współczynnik ten zawsze byłby na takim samym poziomie, neurony nie pozawijałyby się w polu figury.

Za dobre rozłożenie neuronów odpowiedzialny jest również promień sąsiedztwa. W momencie gdy zostanie wylosowany punkt w figurze, neurony przesuwają się w jego kierunku. Jednak nie poruszają się one wszystkie, ilość tą symbolizuje właśnie wspomniany współczynnik sąsiedztwa. Aby wzmocnić efekt lepszego rozłożenia po figurze, jest on zmniejszany w miarę upływu kolejnych iteracji, aż do poziomu gdzie zmianie pozycji podlegać będzie zaledwie jeden neuron.

Ostatnim elementem do wybrania jest kształt, na którym neurony będą się rozkładały. Można go wybrać z listy z gotowymi wzorami znajdującej się po prawej

stronie, pokazanej na rysunku 1.2, lub stworzyć własny. W celu konstrukcji własnej figury, należy kliknąć odpowiedni przycisk oznaczony jako „Rysuj trójkąt”, „Rysuj czworokąt”, lub „Rysuj okrąg” w zależności od figury jaką chcemy wykorzystać.



Rysunek 1.2 – Menu główne aplikacji

W aplikacji domyślne włączone jest automatyczne ustawianie współczynnika uczenia. Polega to na tym, że jego wartość startowa jest ustawiana jako połowa liczby neuronów. Aby wyłączyć tą opcję należy odznaczyć ptaszek w okienku obok opisu funkcji. Wtedy odblokuje się możliwość ręcznego podania współczynnika sąsiedztwa.

Kolejną opcją do modyfikacji jest blokada kształtu w trakcie uczenia. Domyślnie opcja ta jest włączona ale po jej dezaktywacji w dowolnym momencie można zmienić kształt figury. Bawiąc się tą opcją dobrze można dostrzec szybkość adaptacji sieci na nowe warunki. Jednak jest ona zależna od momentu w którym sieć się znajduje. Na początku uczenia szybkość przechodzenia neuronów z jednej figury na drugą jest bardzo duża. Lecz gdy promień sąsiedztwa i współczynnik uczenia są małe może się zdarzyć tak, że sieć nie zdąży się ułożyć albo zrobi to bardzo powoli. Niemiej jednak opcja ta bardzo fajnie pokazuje działanie sieci i pozwala lepiej je zrozumieć.

2. Techniczny opis aplikacji

Aplikacja została napisana w języku C#, za interfejs graficznych odpowiada technologia Windows Forms. To dzięki niemu w łatwy sposób można zmieniać i ustawiać parametry startowe sieci. To co daje każdy z nich zostało już wyjaśnione wcześniej, w tym punkcie opisuję techniczną prezentację zastosowanych zagadnień.

Algorytmy wszystkich figur zostały umieszczone w pliku Figures.cs. To tutaj znajdują się wszystkie klasy i metody stosowane do obsługi konkretnych kształtów. Składową każdej figury jest klasa Point, która reprezentuje punkt. Przechowuje ona wartości x i y, które służą do wyświetlania prostych. Postanowiłem że każdą z figur: trójkąt, czworokąt, koło, czy elipsę opiszę jako osobną klasę. Aby ułatwić sobie późniejsze ich używanie stworzyłem abstrakcyjną klasę „Shape", przedstawioną na rysunku 2.1, która jest pośrednikiem między operacjami wykonywanymi przez sieć a różnymi figurami. Dzięki temu za pomocą jednego wywołania program może wylosować punkt znajdujący się w dowolnej figurze, ponadto ułatwia to ich wyświetlanie na ekranie. Każda z zdefiniowanych figur posiada zestaw metod, takich jak: zwrócenie losowego punktu wewnątrz figury, dodanie współrzędnych krawędzi, resetowanie wprowadzonych danych, sprawdzenie czy wszystkie punkty w figurze są prawidłowo zainicjalizowane, oraz dwie funkcję umożliwiające ich wyświetlanie na ekranie.

```
Odwołania: 8
public abstract class Shape
{
    Odwołania: 8
    public abstract Point RandomPointIn();
    Odwołania: 5
    public abstract void AddPoint(Point p);
    Odwołania: 5
    public abstract void Reset();
    Odwołania: 6
    public abstract bool IsComplete();
    Odwołania: 20
    public abstract void Draw(PictureBox pictureBox);
    Odwołania: 4
    public abstract void Draw(object sender, System.Windows.Forms.PaintEventArgs e);
}
```

Rysunek 2.1 – Abstrakcyjna klasa „Shape"

Gdy użytkownik naciśnie przycisk „Start” znajdujący w prawym górnym rogu menu graficznego. Algorytm wczyta wszystkie podane parametry, oraz kształt wybranej figury. Następnie tworzona jest lista neuronów o zdeklarowanej wielkości i zapełniana losowymi punktami, znajdującymi się wewnątrz określonego kształtu. W każdej iteracji jest dodatkowo losowany jeszcze jeden punkt, według którego następuje przesunięcie neuronów. Dzieje się to w ten sposób że wybierany jest ten neuron który znajduje się najbliżej wyznaczonego punktu, a następnie od niego wybierane są kolejne. Ilość wyznaczonych punktów zależy od aktualnej wartości promienia sąsiedztwa. Potem następuje przesunięcie wszystkich neuronów znajdujących się w promieniu sąsiedztwa z uwzględnieniem wartości stałej uczenia. Fragment z kodem odpowiadający za opisywane czynności został umieszczony na rysunku 2.2, znajdującym się poniżej.

```
// Set the neuron index of the nearest selected point and radius of neighbors
int nearestIndex = distances.IndexOf(distances.Min());
int firstIndex = 0;
int lastIndex = 0;

if (nearestIndex - NeighborhoodRadius < 0)
{
    firstIndex = 0;
}
else
{
    firstIndex = nearestIndex - NeighborhoodRadius;
}

if (nearestIndex + NeighborhoodRadius > NeuronCount)
{
    lastIndex = NeuronCount;
}
else
{
    lastIndex = nearestIndex + NeighborhoodRadius;
}

NeuronsList[nearestIndex] = NeuronsList[nearestIndex] + somLearningRate * (randomPoint - NeuronsList[nearestIndex]);

// Moving neurons
for (int j = firstIndex; j < lastIndex; j++)
{
    if (j == nearestIndex) continue;
    NeuronsList[j] = NeuronsList[j] + somLearningRate / Math.Abs(nearestIndex - j) * (randomPoint - NeuronsList[j]);
}
```

Rysunek 2.2 – Fragment kodu odpowiadający za przesuwanie neuronów

Po przesunięciu neuronów zmniejszana jest stała uczenia. Etap ten zaprezentowany jest na rysunku 2.3, widocznym poniżej. We wzorze występuje zmienna o nazwie lambda która jest opisywana wzorem:

```
lambda = Ilość iteracji / Log(sigma)
sigma = Ilość neuronów / 2;
```

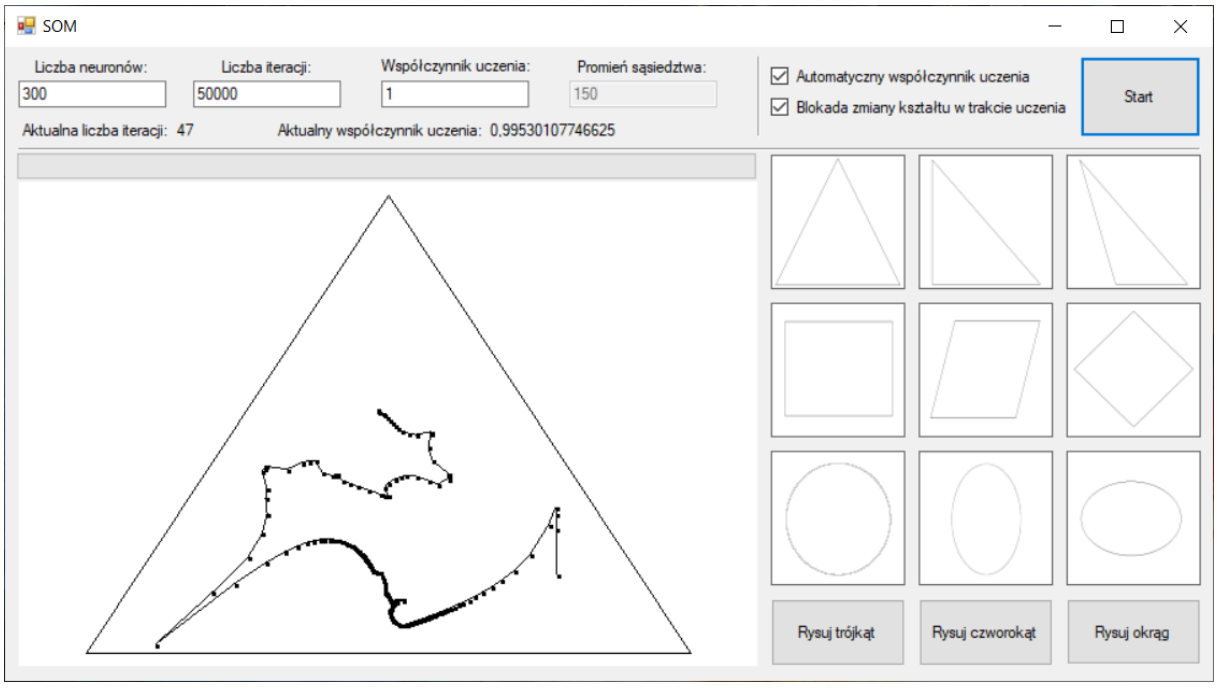
```
// Decreasing learning rate
double x = Math.Exp(-(i + 1) / lambda);
somLearningRate = tmpLearningRate * x;
LearningRate = somLearningRate;
```

Rysunek 2.3 – Fragment kodu odpowiadający za zmniejszanie stałej uczenia

Ostatnim elementem jest zaktualizowanie stanu sieci i paska postępu. Całość jest powtarzana tak długo na ile została ustawiona liczba iteracji.

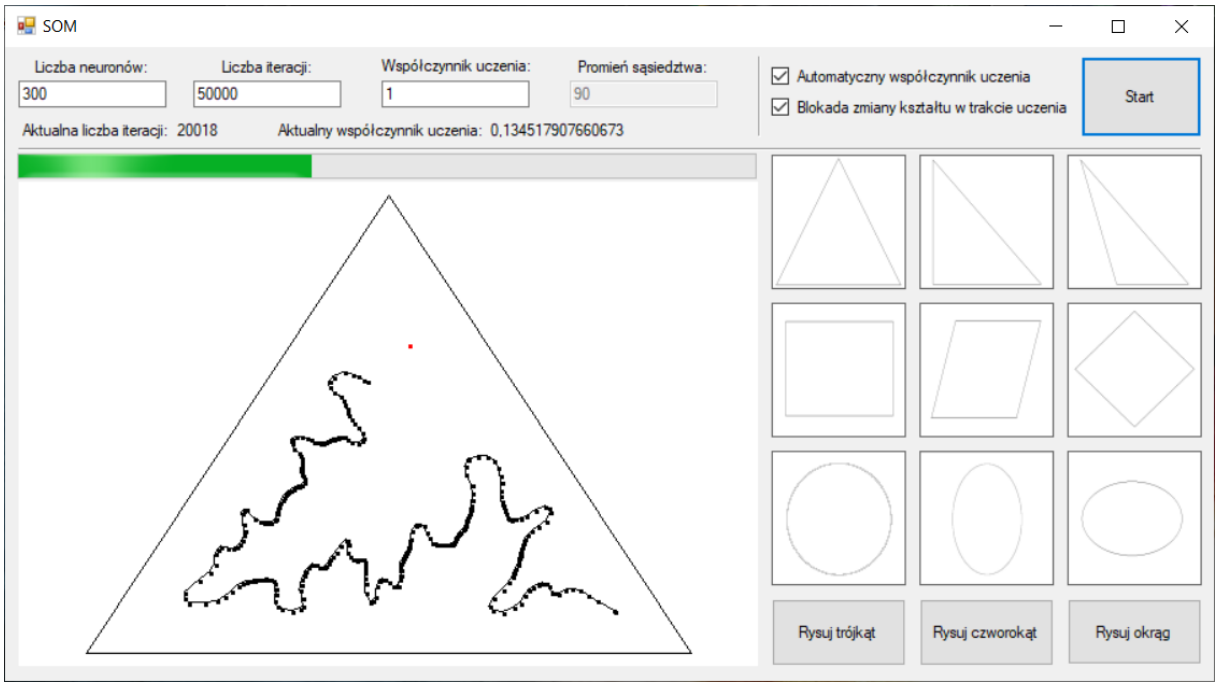
3. Przykładowe wyniki

Na początek do przetestowania sieci neuronowej użyłem kształtu trójkąta. Ustawiłem ilość neuronów na trzysta, liczbę iteracji na pięćdziesiąt tysięcy, współczynnik uczenia na 1. Zostawiłem włączoną automatyczną opcję wyboru współczynnika sąsiedztwa, co przy tej liczbie neuronów dało 150. Podczas działania sieci nie zmieniałem kształtu figury na inny. Uwieczniłem kilka stanów rozłożenia siatki neuronów na figurze. Rysunek 3.1, pokazuje stan po pierwszych sekundach wystartowania algorytmu. Widać dokładnie że, większość z neuronów jest zbita w grupie i dopiero zaczyna rozprzestrzeniać się po figurze. Promień sąsiedztwa jeszcze nie zdążył się zmniejszyć, ale za to współczynnik ucznia już nieznacznie zmalał.



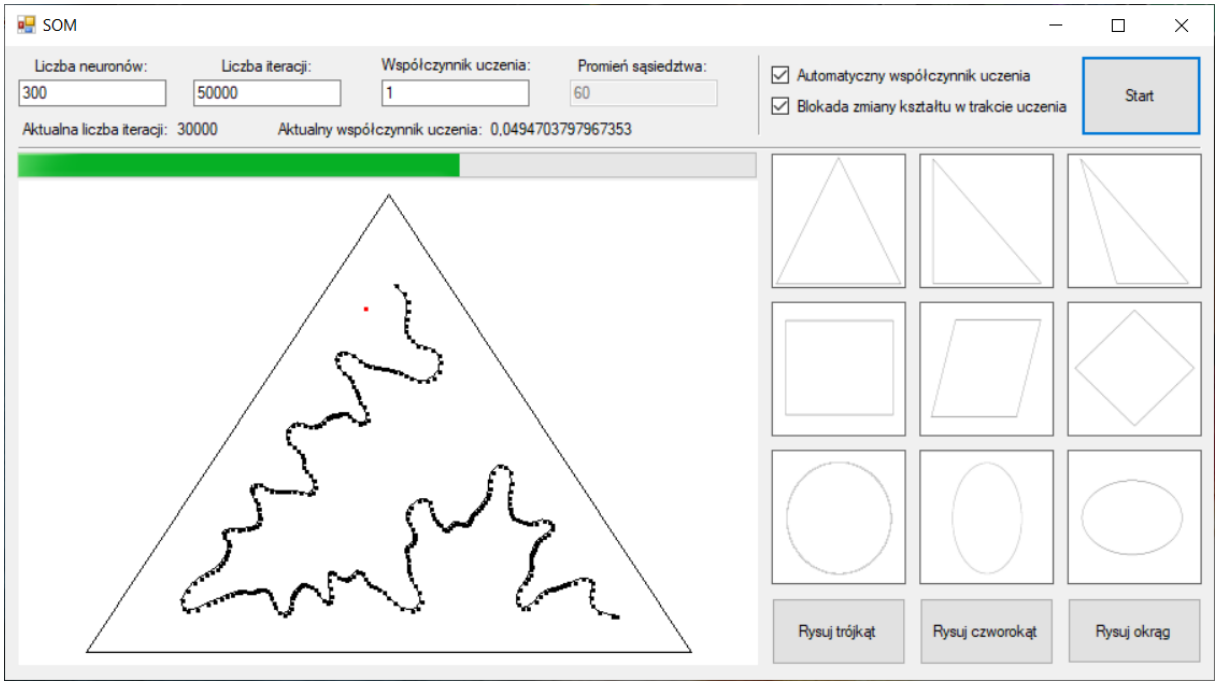
Rysunek 3.1 – Stan sieci przy czterdziestej siódmej iteracji

Kolejny stan uchwyciłem po dwudziestu tysiącach iteracji, co widoczne jest na rysunku 3.2, znajdującym się poniżej. Tutaj już można dostrzec rozłożenie po figurze wszystkich neuronów, jednak nie są one jeszcze w pełni ułożone, gdyż sieć nie rozprzestrzeniła się jeszcze przy górnym wierzchołku trójkąta. Promień sąsiedztwa zmalał do 90 neuronów, stała uczenia wyniosła okolice 0.134.



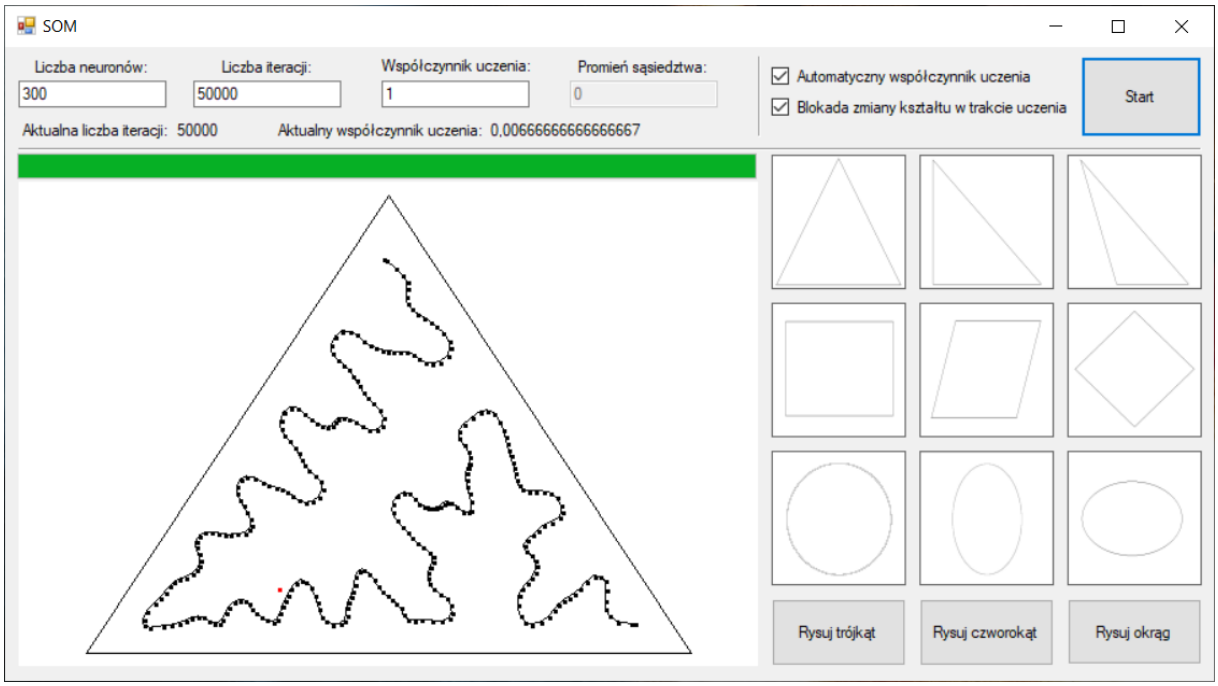
Rysunek 3.2 – Stan sieci po dwudziestu tysiącach iteracji

Na następnym rysunku 3.3 widoczna jest sytuacja po trzydziestu tysiącach iteracji. W porównaniu do poprzednio opisywanego stanu, neurony doszły w okolice górnego wierzchołka trójkąta. Liczba przesuwanych jednocześnie neuronów spadła do sześćdziesięciu, a współczynnik uczenia osiągnął okolice 0.05.



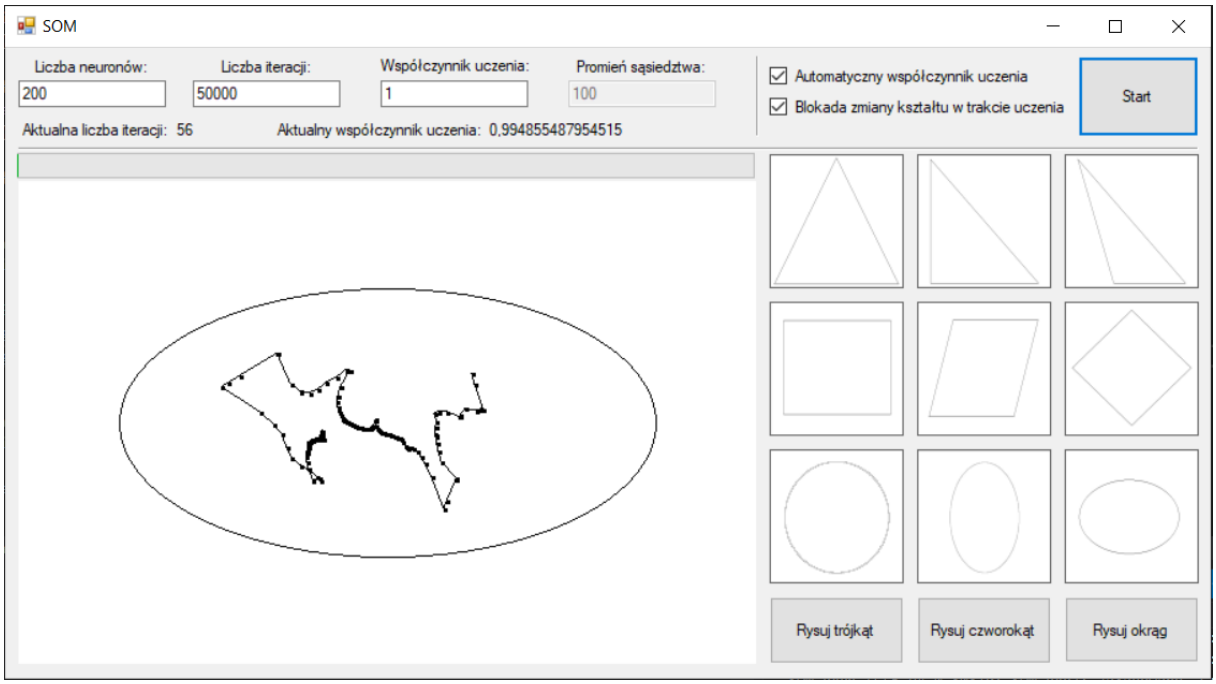
Rysunek 3.3 – Stan sieci po trzydziestu tysiącach iteracji

Na ostatnim rysunku oznaczonym numerem 3.4 pokazany jest stan końcowy sieci. Widać że neurony rozciągnęły się po całej figurze, tworząc krzywe Peano. Współczynnik uczenia zatrzymał się osiągając wartość 0.0066(6). Promień sąsiedztwa zmalał do zera po zatrzymaniu działania sieci.



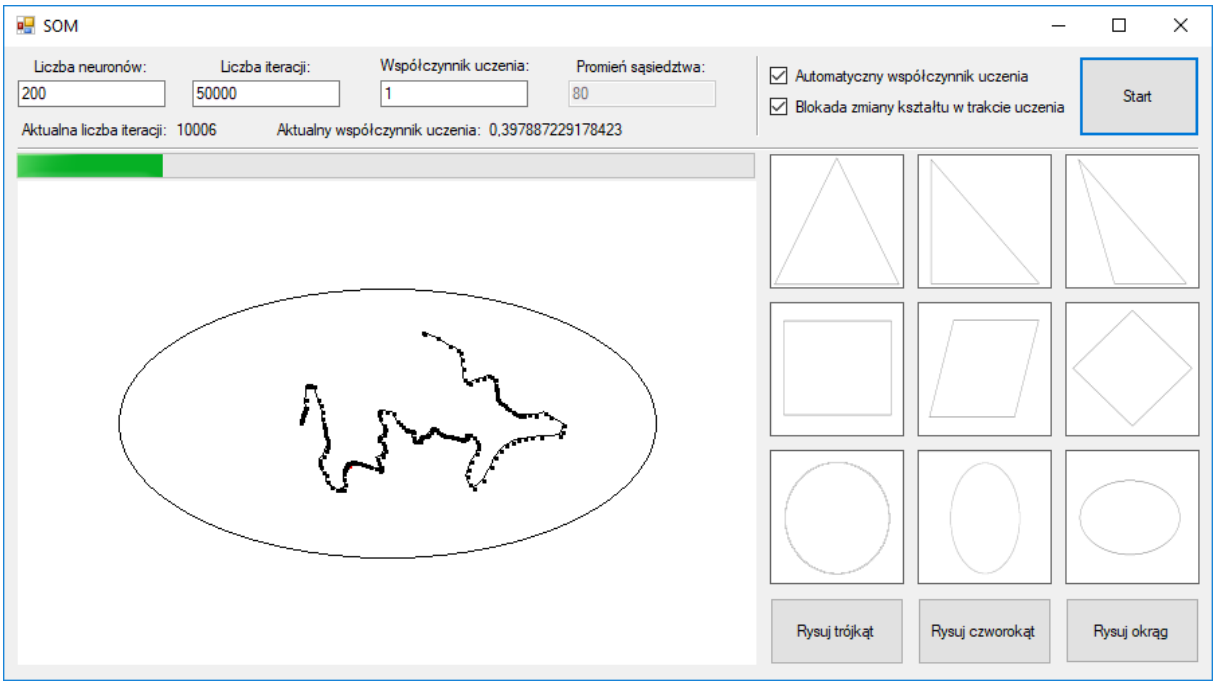
Rysunek 3.4 – Stan sieci po pięćdziesięciu tysiącach iteracji

Na następny test postanowiłem zmienić figurę, tym razem wybrałem elipsę. Liczbę neuronów zmniejszyłem do dwustu, liczbę iteracji zostawiłem na poziomie pięćdziesięciu tysięcy oraz ustawiłem stałą uczenia na jeden. Tutaj podobnie jak wcześniej po pięćdziesięciu iteracjach sieć dopiero zaczyna się rozkładać co przedstawione jest na rysunku 3.5 zamieszczonym poniżej.



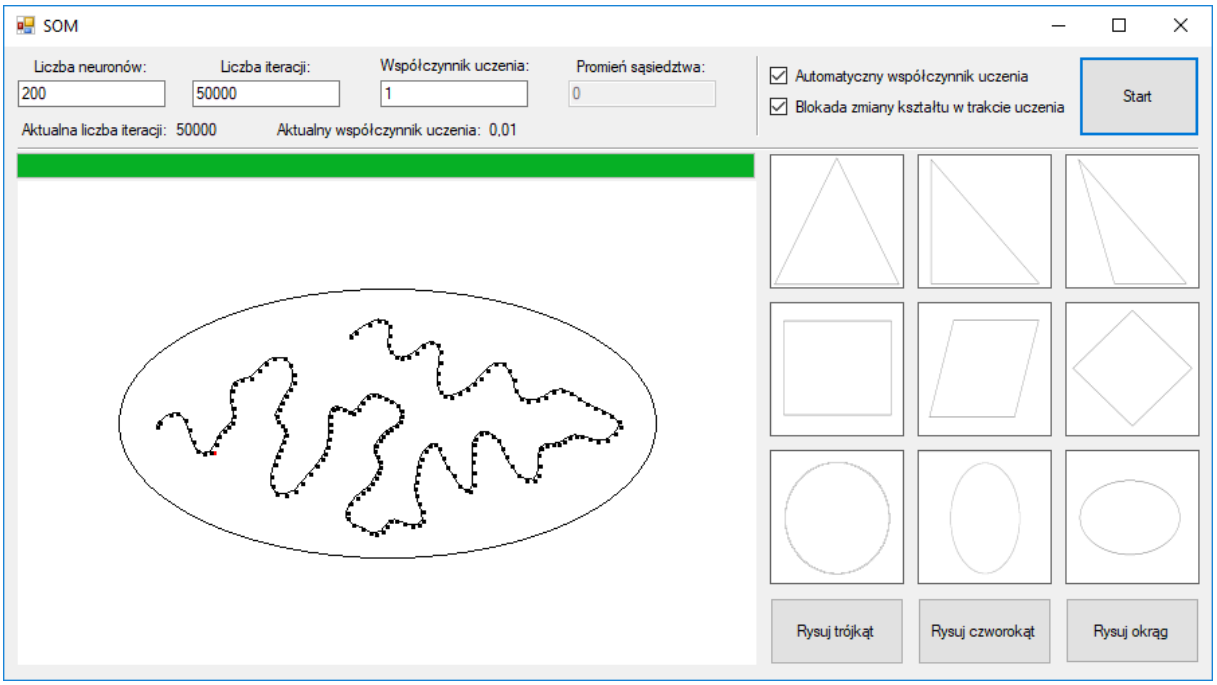
Rysunek 3.5 – Stan sieci po pięćdziesięciu iteracjach

Na kolejnym rysunku 3.6 zamieszczonym poniżej, przedstawiony jest stan neuronów po dziesięciu tysiącach iteracji. Jak można zauważyć neurony ułożyły się wstępnie w środku figury. Promień sąsiedztwa zmniejszył się do osiemdziesięciu, a współczynnik uczenia osiągnął wartość 0.397.



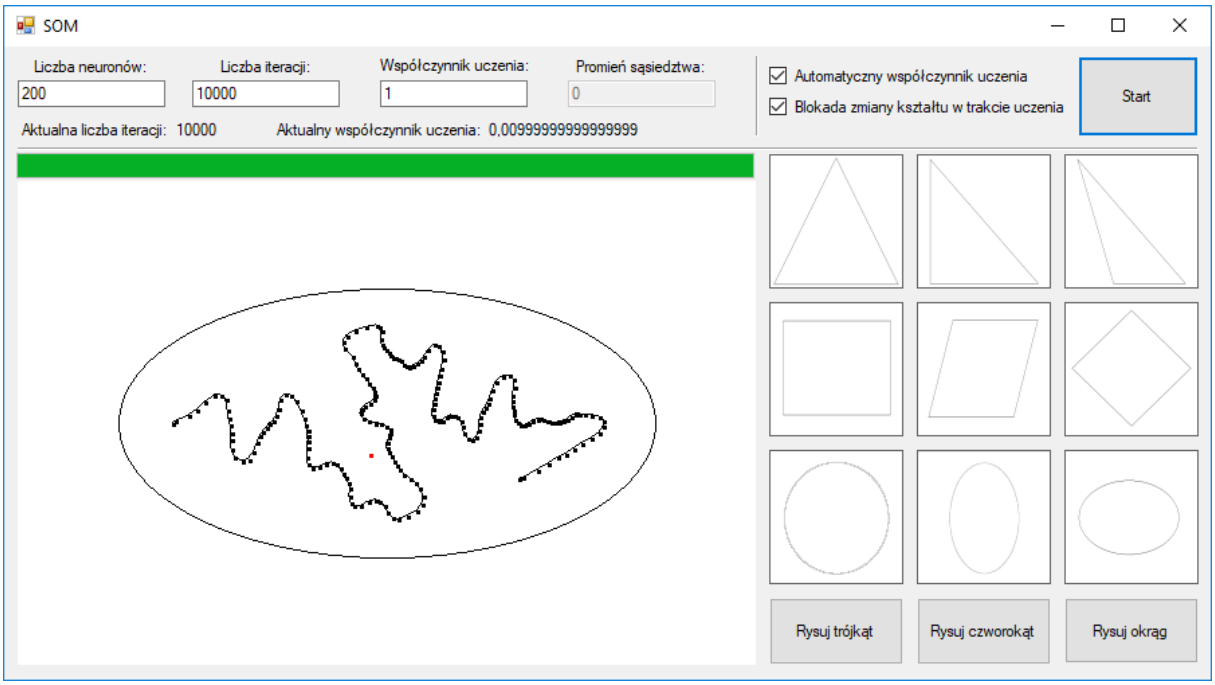
Rysunek 3.5 – Stan sieci po dziesięciu tysiącach iteracji

Końcowy wynik rozłożenia neuronów na elipsie został zaprezentowany na rysunku 3.6, umieszczonym poniżej. Jak można zauważyć punkty rozłożyły się dość dobrze po całej figurze, jak w przypadku trójkąta. Współczynnik uczenia zatrzymał się na wartości 0.01.



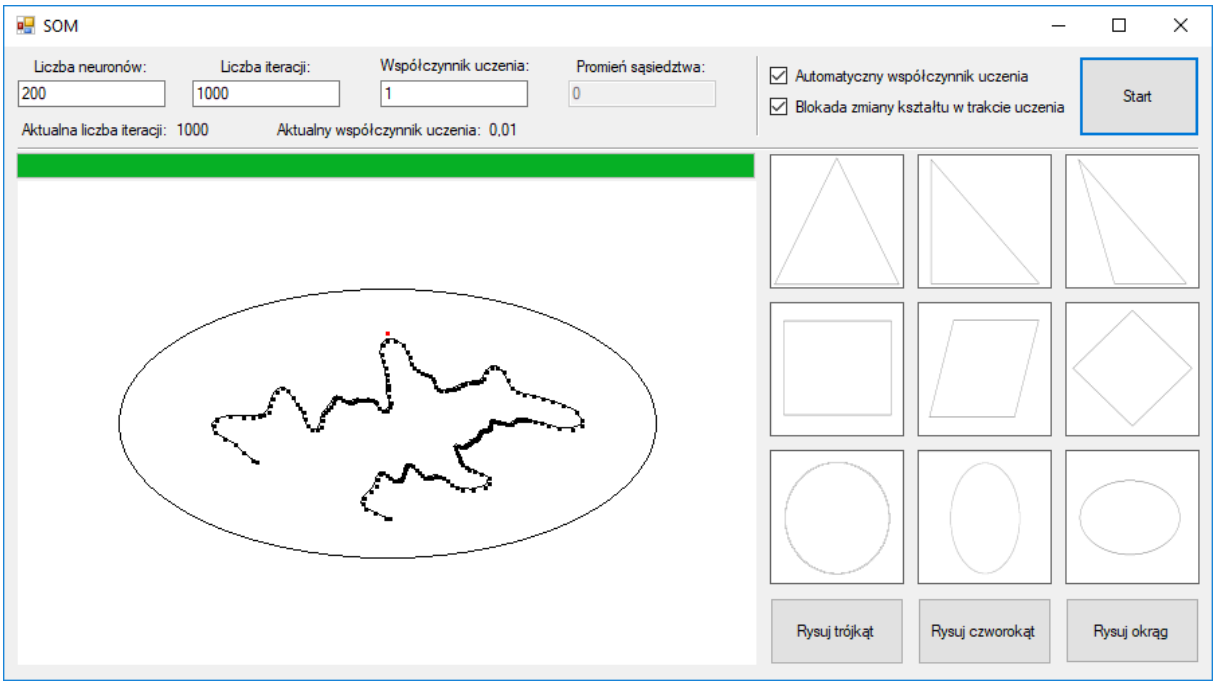
Rysunek 3.6 – Stan sieci po pięćdziesięciu tysiącach iteracji

W kolejnym teście postanowiłem wykorzystać podobne wartości co wcześniej, lecz zmniejszyłem liczbę iteracji do dziesięciu tysięcy. Wynik tej operacji został przedstawiony na rysunku 3.6, umieszczonym poniżej. W porównaniu do poprzedniego rozkładu, tutaj widać że neurony wypełniły figurę w znacznie mniejszym stopniu, co świadczy o tym, że ograniczenie liczby iteracji mocno wpływa na zachowanie sieci.



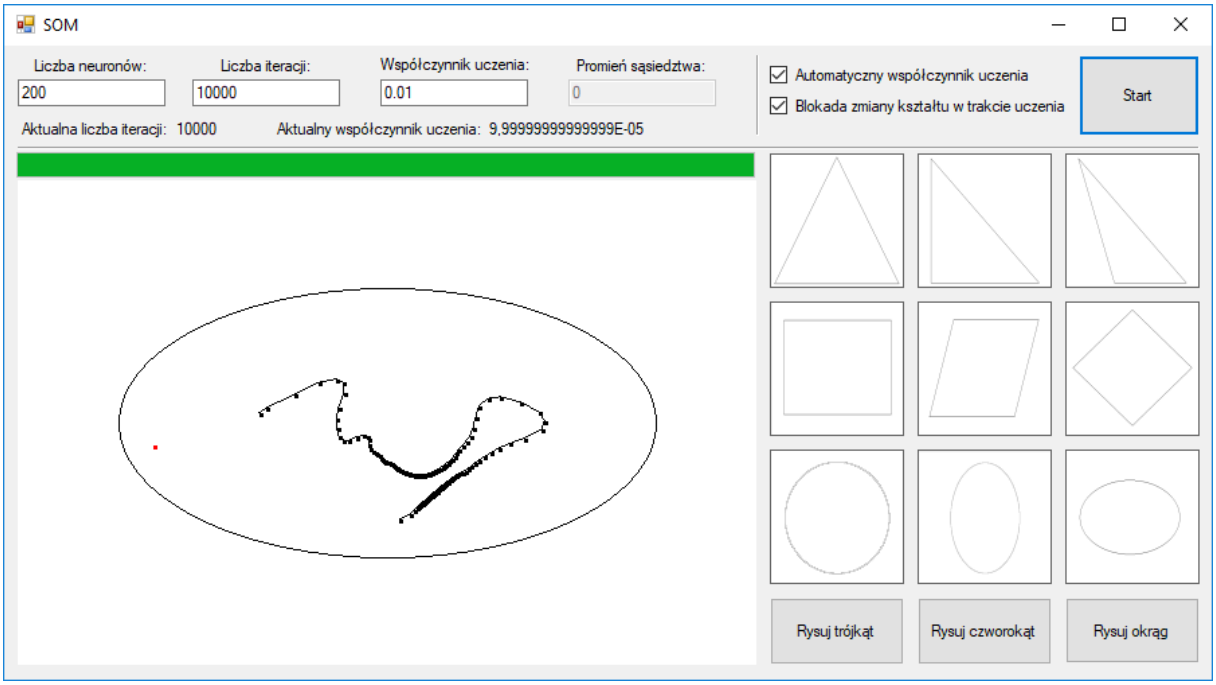
Rysunek 3.6 – Stan sieci po dziesięciu tysiącach iteracji

Uzyskując gorszy wynik w poprzednim przebiegu, sprawdziłem jak rozłożyłby się neurony gdybym zmniejszył ilość iteracji do tysiąca. Wynik tej operacji przedstawiony jest na rysunku 3.7, umieszczonym poniżej. Analogicznie do poprzedniej sytuacji otrzymałem jeszcze gorsze rozłożenie neuronów.



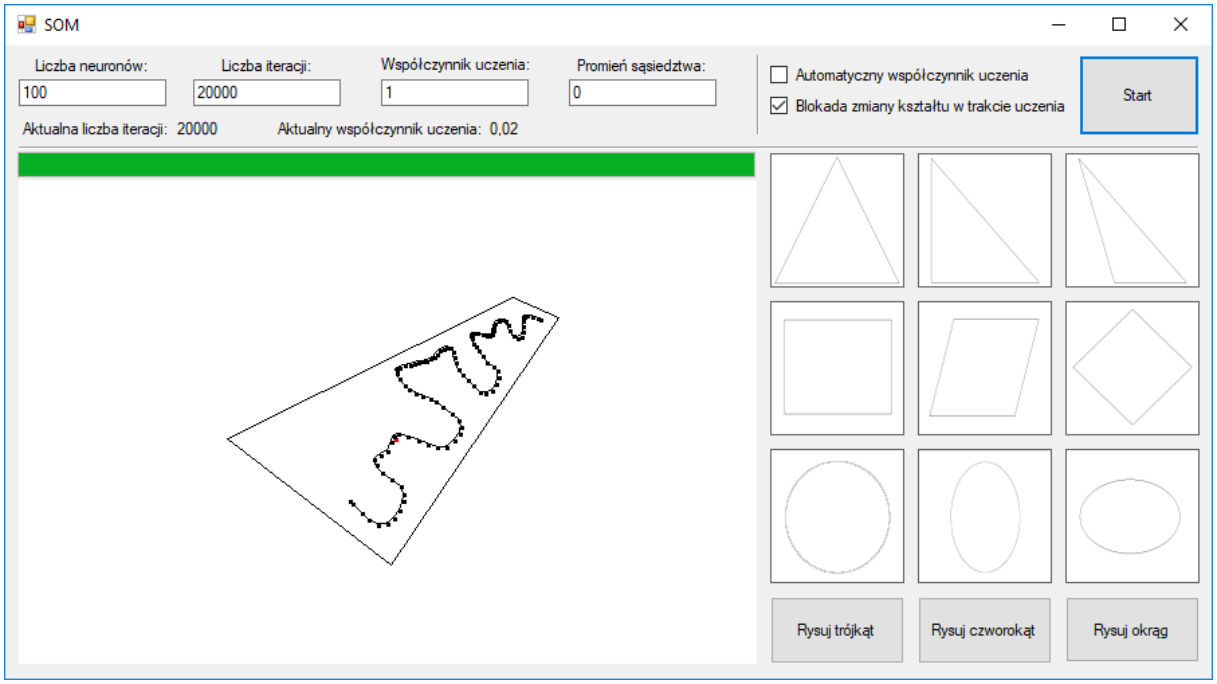
Rysunek 3.7 – Stan sieci po tysiącu iteracji

Postanowiłem jeszcze przetestować wpływ modyfikacji współczynnika uczenia na rozkład neuronów w figurze. Powróciłem do ustawienia ilości iteracji na dziesięć tysięcy, przy dwustu neuronach i zmniejszyłem stałą uczenia do 0.01. Tym razem wynik okazał się totalną katastrofą. Rozłożenie siatki wypadło najgorzej jak dotąd testowanych przypadków. Spowodowane jest to tym, że kolejne przemieszczenia neuronów były bardzo małe. Dodatkowo efekt ten spotęgowało to, że współczynnik uczenia jeszcze malał z każdą kolejną iteracją. Jak można zauważyć na rysunku 3.8, umieszczonym poniżej, współczynnik uczenia spadł aż do czterech zer po przecinku. W efekcie osłabiło to tak przemieszczenie neuronów, że ledwo zmieniały swoje miejsce.



Rysunek 3.8 – Stan sieci po dziesięciu tysiącach iteracji

Na ostatnim teście zdecydowałem się na użycie opcji rysowania własnej figury. Stworzyłem zatem czworokąt, ustawiłem liczbę neuronów na sto, liczbę iteracji na dwadzieścia tysięcy, współczynnik uczenia 1. Wyłączyłem też opcję automatycznego wyboru współczynnika uczenia i ustawiłem go na maksymalną wartość dla podanych parametrów, czyli sto. Oznacza to że sieć rozpocznie działanie od zmienienia pozycji większej ilości punktów niż zazwyczaj. Wynik eksperymentu został przedstawiony na rysunku 3.9, widocznym poniżej.



Rysunek 3.9 – Stan sieci po dwudziestu tysiącach iteracji

4. Podsumowanie

Podsumowując używanie map samoorganizujących do tworzenia krzywych Peano wypełniających dowolne kształty, działa dość dobrze. Jednak aby neurony rozłożyły się na jak największej powierzchni, trzeba dobrze dobrać liczbę iteracji, stałą uczenia oraz promień sąsiedztwa. Złe dobranie parametrów może skutkować słabym dopasowaniem do figury, albo nawet nierozłożeniem się punktów na siatce. Dobre zapełnienie figury wymaga dużej ilości iteracji, a co za tym idzie jest bardzo długi czas oczekiwania aż sieć zakończy swoje działanie. Jednak gdy wybierze się dobre parametry i odczeka dłuższą chwilę, otrzymane efekty potrafią usatysfakcjonować.