Propagacja wsteczna

Sieć propagacji wstecznej ze sprzężeniem do przodu

Sieć ze sprzężeniem do przodu i propagacją wsteczną jest bardzo popularnym modelem w sieciach neuronowych. Nie ma połączeń sprzężenia zwrotnego, ale błędy są propagowane wstecz podczas uczenia. Stosowany jest błąd najmniejszy średniokwadratowy. Można sformułować wiele aplikacji do korzystania z sieci z propagacją wsteczną ze sprzężeniem do przodu, a metodologia jest modelem dla większości wielowarstwowych sieci neuronowych. Błędy na wyjściu określają miary błędów wyjściowych warstwy ukrytej, które są wykorzystywane jako podstawa do dopasowania wag połączeń między warstwą wejściową a warstwą ukrytą. Dopasowanie dwóch zestawów wag między parami warstw i ponowne obliczenie wyników to proces iteracyjny, który trwa do momentu, gdy błędy spadną poniżej poziomu tolerancji. Parametry szybkości uczenia się skalują korekty do wag. Parametr momentum może być również używany do skalowania korekt z poprzedniej iteracji i dodawania do korekt w bieżącej iteracji.

Mapowanie

Sieć propagacji wstecznej ze sprzężeniem do przodu odwzorowuje wektory wejściowe na wektory wyjściowe. Pary wektorów wejściowych i wyjściowych są wybierane do uczenia sieci w pierwszej kolejności. Po zakończeniu szkolenia wagi są ustawiane i sieć może być używana do znajdowania wyników dla nowych danych wejściowych. Wymiar wektora wejściowego określa liczbę neuronów w warstwie wejściowej, a liczba neuronów w warstwie wyjściowej jest określona przez wymiar wyjść. Jeśli w warstwie wejściowej jest k neuronów i m neuronów w warstwie wyjściowej, ta sieć może wykonać mapowanie z przestrzeni k-wymiarowej do przestrzeni m-wymiarowej. Oczywiście, czym jest to mapowanie, zależy od tego, jaka para wzorców lub wektorów jest używana jako przykłady do trenowania sieci, które określają wagi sieci. Po przeszkoleniu sieć daje obraz nowego wektora wejściowego w ramach tego mapowania. Wiedza o tym, do jakiego mapowania ma być wytrenowana sieć ze sprzężeniem do przodu wstecznej propagacji, implikuje wymiary przestrzeni wejściowej i przestrzeni wyjściowej, dzięki czemu można określić liczbę neuronów, które mają znajdować się w warstwach wejściowych i wyjściowych.

Układ

Architekturę sieci z propagacją wsteczną ze sprzężeniem do przodu pokazano na rysunku. Chociaż może być wiele ukrytych warstw, zilustrujemy tę sieć tylko jedną ukrytą warstwą. Ponadto liczba neuronów w warstwie wejściowej i w warstwie wyjściowej jest określana odpowiednio przez wymiary wzorców wejściowych i wyjściowych. Nie jest łatwo określić, ile neuronów jest potrzebnych do warstwy ukrytej. Aby uniknąć zaśmiecania rysunku, pokażemy układ na rysunku z pięcioma neuronami wejściowymi, trzema neuronami w warstwie ukrytej i czterema neuronami wyjściowymi, z kilkoma reprezentatywnymi

Sieć ma trzy pola neuronów: jedno dla neuronów wejściowych, jedno dla ukrytych elementów przetwarzających i jedno dla neuronów wyjściowych. Jak już wspomniano, połączenia są przeznaczone do aktywności typu feed forward. Istnieją połączenia od każdego neuronu w polu A do każdego w polu B i z kolei od każdego neuronu w polu B do każdego neuronu w polu C. Tak więc istnieją dwa zestawy

wag, które składają się na aktywacje ukrytych neurony warstwy i te, które pomagają określić wyjściowe aktywacje neuronów. Podczas uczenia wszystkie te wagi są dostosowywane przez uwzględnienie tego, co można nazwać funkcją kosztu pod względem błędu w obliczonym wzorze wyjściowym i pożądanym wzorcu wyjściowym.

Trening

Sieć propagacji wstecznej ze sprzężeniem do przodu przechodzi nadzorowane szkolenie ze skończoną liczbą par wzorców składających się z wzorca wejściowego i pożądanego lub docelowego wzorca wyjściowego. W warstwie wejściowej prezentowany jest wzorzec wejściowy. Tutaj neurony przekazują aktywacje wzorców do neuronów następnej warstwy, które znajdują się w warstwie ukrytej. Sygnały wyjściowe neuronów warstwy ukrytej są uzyskiwane przy użyciu być może odchylenia, a także funkcji progowej z aktywacjami określonymi przez wagi i dane wejściowe. Te dane wyjściowe warstwy ukrytej stają się danymi wejściowymi do neuronów wyjściowych, które przetwarzają dane wejściowe za pomocą opcjonalnego odchylenia i funkcji progowej. Ostateczny wynik sieci jest określany przez aktywacje z warstwy wyjściowej. Obliczony wzór i wzór wejściowy są porównywane, funkcja tego błędu jest określana dla każdego składnika wzoru i obliczane jest dopasowanie do wag połączeń między warstwą ukrytą a warstwą wyjściową. Podobne obliczenia, nadal oparte na błędach w danych wyjściowych, są wykonywane dla wag połączeń między warstwą wejściową a ukrytymi. Procedura jest powtarzana z każdą parą wzorców przypisaną do uczenia sieci. Każde przejście przez wszystkie wzorce treningowe nazywane jest cyklem lub epoką. Proces jest następnie powtarzany tyle cykli, ile potrzeba, aż błąd znajdzie się w określonej tolerancji.

W uczeniu w sieci z propagacją wsteczną ze sprzężeniem do przodu może być używanych więcej niż jeden parametr szybkości uczenia się. Możesz użyć jednego z każdym zestawem wag pomiędzy kolejnymi warstwami.

Ilustracja: Regulacja wag połączeń z neuronu w ukrytej warstwie

Będziemy tak konkretni, jak to konieczne, aby obliczenia były jasne. Najpierw przypomnijmy, że aktywacja neuronu w warstwie innej niż warstwa wejściowa jest sumą iloczynów jego wejść i wag odpowiadających połączeniom, które wnoszą te wejścia. Omówmy neuron j-ty w warstwie ukrytej. Bądźmy konkretni i powiedzmy j = 2. Załóżmy, że wzorzec wejściowy to (1.1, 2.4, 3.2, 5.1, 3.9) i docelowy wzorzec wyjściowy to (0,52, 0,25, 0,75, 0,97). Niech wagi dla drugiego neuronu warstwy ukrytej zostaną podane przez wektor (–0,33, 0,07, –0,45, 0,13, 0,37). Aktywacją będzie ilość:

(-0,33 * 1,1) + (0,07 * 2,4) + (-0,45 * 3,2) + (0,13 * 5,1) + (0,37 * 3,9) = 0,471

Teraz dodaj do tego opcjonalne odchylenie, powiedzmy 0,679, aby otrzymać 1,15. Jeśli użyjemy funkcji sigmoidalnej podanej przez:

1 / (1+ exp(-x)),

przy x = 1,15 otrzymujemy wynik tego neuronu warstwy ukrytej jako 0,7595.

Wartości podajemy z dokładnością do kilku miejsc po przecinku tylko dla ilustracji, w przeciwieństwie do precyzji, jaką można uzyskać na komputerze. Potrzebujemy również obliczonego wzorca wyjściowego. Powiedzmy, że okazuje się, że jest rzeczywisty =(0,61, 0,41, 0,57, 0,53), podczas gdy pożądany wzór jest pożądany =(0,52, 0,25, 0,75, 0,97). Oczywiście istnieje rozbieżność między tym co jest pożądane i co jest obliczane. Różnice składników są podane w wektorze, pożądany - rzeczywisty = (-0,09, -0,16, 0,18, 0,44). Używamy ich do utworzenia kolejnego wektora, w którym każdy składnik jest iloczynem składnika błędu, odpowiadającego mu obliczonego składnika wzorca i dopełnienia tego

ostatniego względem 1. Na przykład dla pierwszego składnika błąd wynosi –0,09, obliczona składowa wzorca wynosi 0,61, a dopełnienie wynosi 0,39. Mnożąc je razem (0,61*0,39*-0,09) otrzymujemy -0,02. Obliczając pozostałe składowe w podobny sposób, otrzymujemy wektor (–0,02, –0,04, 0,04, 0,11). Pożądany rzeczywisty wektor, który jest wektorem błędu pomnożonym przez rzeczywisty wektor wyjściowy, daje wartość błędu odzwierciedloną z powrotem na wyjściu warstwy ukrytej. Jest to skalowane wartością (1-wyjściowy wektor), która jest pierwszą pochodną funkcji aktywacji wyjścia dla stabilności numerycznej. W dalszej części zobaczysz wzory tego procesu. Należy kontynuować wsteczną propagację błędów. Potrzebujemy teraz wag połączeń między drugim neuronem w warstwie ukrytej, na której się koncentrujemy, a różnymi neuronami wyjściowymi. Powiedzmy, że te wagi są podane przez wektor (0,85, 0,62, –0,10, 0,21). Błąd drugiego neuronu w warstwie ukrytej jest teraz obliczany jak poniżej, na podstawie jego wyjścia. błąd = 0,7595 * (1 - 0,7595) * ((0,85 * -0,02) + (0,62 * -0,04) + (-0,10 * 0,04) + (0,21 * 0,11)) = -0,0041. Tutaj ponownie mnożymy błąd (np. -0,02) z wyjścia bieżącej warstwy przez wartość wyjściową (0,7595) i wartość (1-0,7595). Używamy wag na połączeniach między neuronami, aby pracować wstecz w sieci. Następnie potrzebujemy parametru szybkości uczenia się dla tej warstwy; ustawmy go na 0,2. Mnożymy to przez wyjście drugiego neuronu w warstwie ukrytej, aby otrzymać 0,1519. Każdy ze składników wektora (–0,02, –0,04, 0,04, 0,11) jest teraz pomnożony przez 0,1519, co dało nasze ostatnie obliczenie. Wynikiem jest wektor, który podaje korekty wag połączeń, które przechodzą od drugiego neuronu w warstwie ukrytej do neuronów wyjściowych. Wartości te są podane w wektorze (–0,003, –0,006, 0,006, 0,017). Po dodaniu tych korekt wagi, które zostaną użyte w następnym cyklu na połączeniach między drugim neuronem w warstwie ukrytej a neuronami wyjściowymi stają się wagami wektora (0,847, 0,614, –0,094, 0,227).

Ilustracja: Regulacja wag połączeń z neuronu w warstwie wejściowej

Przyjrzyjmy się, jak obliczane są korekty wag połączeń wychodzących z neuronu i-tego w warstwie wejściowej do neuronów w warstwie ukrytej. Dla ilustracji weźmy konkretnie i = 3. Wiele informacji, których potrzebujemy, zostało już uzyskanych w poprzednim omówieniu dla drugiego neuronu warstwy ukrytej. Mamy błędy w obliczonym wyjściu jako wektor (–0,09, –0,16, 0,18, 0,44), a błąd drugiego neuronu w warstwie ukrytej otrzymaliśmy jako – 0,0041, którego nie użyliśmy powyżej. Tak jak błąd na wyjściu jest propagowany z powrotem w celu przypisania błędów neuronom w warstwie ukrytej, błędy te mogą być propagowane do neuronów warstwy wejściowej. Aby określić korekty wag na połączeniach między warstwą wejściową i ukrytą, potrzebujemy błędów określonych dla wyjść neuronów warstwy ukrytej, parametru szybkości uczenia się i aktywacji neuronów wejściowych, które są tylko wartościami wejściowymi dla warstwa wejściowa. Przyjmijmy, że parametr szybkości uczenia się wynosi 0,15. Następnie korekty wagi dla połączeń od trzeciego neuronu wejściowego do neuronów warstwy ukrytej uzyskuje się poprzez pomnożenie błędu wyjściowego konkretnego neuronu warstwy ukrytej przez parametr szybkości uczenia się i przez składnik wejściowy z neuronu wejściowego. Korekta wagi na połączeniu z trzeciego neuronu wejściowego do drugiego neuronu warstwy ukrytej wynosi 0,15 * 3,2 * –0,0041, co daje wynik –0,002. Jeśli waga na tym połączeniu wynosi powiedzmy –0,45, to dodając korektę –0,002, otrzymujemy zmodyfikowaną wagę –0,452, do wykorzystania w następnej iteracji działania sieci. Podobne obliczenia są dokonywane w celu modyfikacji wszystkich innych wag.

Korekty wartości progowych lub błędów systematycznych

Odchylenie lub wartość progowa, którą dodaliśmy do aktywacji, przed zastosowaniem funkcji progowej w celu uzyskania wyjścia neuronu, zostanie również dostosowana na podstawie błędu propagowanego z powrotem. Potrzebne do tego wartości zostały omówione w poprzedniej dyskusji. Korekta wartości progowej neuronu w warstwie wyjściowej jest uzyskiwana przez pomnożenie obliczonego błędu (nie tylko różnicy) na wyjściu neuronu wyjściowego i parametru szybkości uczenia

się użytego w obliczeniach korekty wag w tej warstwie. W naszym poprzednim przykładzie mamy parametr szybkości uczenia się jako 0,2, a wektor błędu jako (–0,02, –0,04, 0,04, 0,11), więc korekty wartości progowych czterech neuronów wyjściowych są podane przez wektor (– 0,004, –0,008, 0,008, 0,022). Te korekty są dodawane do bieżących poziomów wartości progowych w neuronach wyjściowych. Dopasowanie do wartości progowej neuronu w warstwie ukrytej uzyskuje się w podobny sposób, mnożąc szybkość uczenia się przez wyliczony błąd na wyjściu neuronu warstwy ukrytej. Dlatego dla drugiego neuronu w warstwie ukrytej dopasowanie do jego wartości progowej jest obliczane jako 0,15 * –0,0041, czyli –0,0006. Dodaj to do bieżącej wartości progowej 0,679, aby uzyskać 0,6784, która ma być użyta dla tego neuronu w następnym wzorcu treningowym dla sieci neuronowej.

Kolejny przykład obliczeń propagacji wstecznej

Widziałeś w poprzednich sekcjach szczegóły obliczeń dla jednego konkretnego neuronu w warstwie ukrytej w sieci ze sprzężeniem do przodu z propagacją wsteczną z pięcioma neuronami wejściowymi i czterema neuronami w warstwie wyjściowej oraz dwoma neuronami w warstwie ukrytej. Zobaczysz wszystkie obliczenia w implementacji C++ w dalszej części tego rozdziału. Jednak teraz przedstawiamy inny przykład i dajemy pełny obraz obliczeń wykonanych w jednej zakończonej iteracji lub cyklu wstecznej propagacji. Rozważmy sieć ze sprzężeniem do przodu z propagacją wsteczną z trzema neuronami wejściowymi, dwoma neuronami w warstwie ukrytej i trzy neurony wyjściowe. Wagi połączeń od neuronów wejściowych do neuronów w warstwie ukrytej są podane w macierzy M-1, a wagi od neuronów w warstwie ukrytej do neuronów wyjściowych są podane w macierzy M-2. Obliczamy wyjście każdego neuronu w warstwach ukrytych i wyjściowych w następujący sposób. Do aktywacji neuronu dodajemy odchylenie lub wartość progową (nazwijmy ten wynik x) i używamy funkcji sigmoid poniżej, aby uzyskać dane wyjściowe.

$f(x) = 1/ (1 + e^{-x})$

Stosowane parametry uczenia to 0,2 dla połączeń między neuronami warstwy ukrytej a neuronami wyjściowymi oraz 0,15 dla połączeń między neuronami wejściowymi a neuronami w warstwie ukrytej. Te wartości, jak pamiętasz, są takie same, jak na poprzedniej ilustracji, aby ułatwić śledzenie obliczeń poprzez porównanie ich z podobnymi obliczeniami w poprzednich sekcjach. Wzorzec wejściowy to (0,52, 0,75, 0,97), a pożądany wzorzec wyjściowy to (0,24, 0,17, 0,65). Początkowe macierze wag są następujące:

M-1 Macierz wag od warstwy wejściowej do warstwy ukrytej

```
   0.6      - 0.4
   0.2        0.8
 - 0.5        0.3
```

Matryca M-2 wag od warstwy ukrytej do warstwy wyjściowej

```
 -0.90        0.43        0.25
  0.11      - 0.67      - 0.75
```

Wartości progowe (lub odchylenie) dla neuronów w warstwie ukrytej wynoszą 0,2 i 0,3, podczas gdy te dla neuronów wyjściowych wynoszą odpowiednio 0,15, 0,25 i 0,05. Tabela przedstawia wszystkie wyniki obliczeń wykonanych w pierwszej iteracji. Zobaczysz zmodyfikowane lub nowe macierze wag i wartości progowe. Użyjesz ich, oryginalnego wektora wejściowego i pożądanego wektora wyjściowego, aby przeprowadzić następną iterację.

Item	I-1	I-2	I-3	H-1	H-2	O-1	O-2	O-3
Input	0.52	0.75	0.97					
Desired Output						0.24	0.17	0.65
M-1 Row 1				0.6	- 0.4			
M-1 Row 2				0.2	0.8			
M-1 Row 3				- 0.5	0.3			
M-2 Row 1						- 0.90	0.43	0.25
M-2 Row 2						0.11	- 0.67	- 0.75
Threshold				0.2	0.3	0.15	0.25	0.05
Activation - H				- 0.023	0.683			
Activation + Threshold -H				0.177	0.983			
Output -H				0.544	0.728			
Complement				0.456	0.272			
Activation -O						- 0.410	- 0.254	- 0.410
Activation + Threshold -O						- 0.260	- 0.004	- 0.360
Output -O						0.435	0.499	0.411
Complement						0.565	0.501	0.589
Diff. from Target						- 0.195	- 0.329	0.239
Computed Error - O						- 0.048	- 0.082	0.058
Computed Error - H				0.0056	0.0012			
Adjustment to Threshold				0.0008	0.0002	-0.0096	-0.0164	0.0116
Adjustment to M-2 Column 1				-0.0005	-0.0070			
Adjustment to M-2 Column 2				0.0007	0.0008			

Adjustment to M-2 Column 3	0.0008	0.0011			
New Matrix M-2 Row 1			- 0.91	0.412	0.262
New Matrix M-2 Row 2			0.096	- 0.694	- 0.734
New Threshold Values -O			0.1404	0.2336	0.0616
Adjustment to M-1 Row 1	0.0004	-0.0001			
Adjustment to M-1 Row 2	0.0006	0.0001			
Adjustment to M-1 Row 3	0.0008	0.0002			
New Matrix M-1 Row 1	0.6004	- 0.4			
New Matrix M-1 Row 2	0.2006	0.8001			
New Matrix M-1 Row 3	-0.4992	0.3002			
New Threshold Values -H	0.2008	0.3002			

W górnym wierszu tabeli znajdują się nagłówki kolumn. Są to: Item, I-1, I-2, I-3 (I-k oznacza neuron k warstwy wejściowej); H-1, H-2 (dla neuronów warstwy ukrytej); i O-1, O-2, O-3 (dla neuronów warstwy wyjściowej). W pierwszej kolumnie tabeli M-1 i M-2 odnoszą się do macierzy wag jak powyżej. Tam, gdzie wpis jest dodany z -H, tak jak w Output -H, informacja odnosi się do ukrytej warstwy. Podobnie -O odnosi się do warstwy wyjściowej, jak w Aktywacja + próg -O. Następna iteracja wykorzystuje następujące informacje z poprzedniej iteracji, które można zidentyfikować w tabeli. Wzorzec wejściowy to (0,52, 0,75, 0,97), a pożądany wzorzec wyjściowy to (0,24, 0,17, 0,65). Aktualne macierze wag są następujące:

M-1 Macierz wag od warstwy wejściowej do warstwy ukrytej:

```
  0.6004   - 0.4
  0.2006     0.8001
- 0.4992     0.3002
```

M-2 Macierz wag od warstwy ukrytej do warstwy wyjściowej:

```
-0.910      0.412      0.262
 0.096     -0.694     -0.734
```

Wartości progowe (lub odchylenie) dla neuronów w warstwie ukrytej to 0,2008 i 0,3002, a dla neuronów wyjściowych odpowiednio 0,1404, 0,2336 i 0,0616. Możesz zachować parametry uczenia jako 0,15 dla połączeń między neuronami warstwy wejściowej i ukrytej oraz 0,2 dla połączeń między neuronami warstwy ukrytej a neuronami wyjściowymi lub możesz je nieznacznie zmodyfikować. To, czy zmienić te dwa parametry, czy nie, jest decyzją, którą można podjąć być może w późniejszej iteracji, po uzyskaniu wyobrażenia o tym, jak proces jest zbieżny. Jeśli jesteś zadowolony z tempa, w jakim

obliczony wzorzec wyjściowy zbliża się do docelowego wzorca wyjściowego, nie zmieniałbyś tych współczynników uczenia się. Jeśli czujesz, że konwergencja jest znacznie wolniejsza niż byś chciał, to parametry szybkości uczenia się można nieznacznie zwiększyć. Jest to subiektywna decyzja zarówno pod względem tego, kiedy (jeśli w ogóle), jak i do jakich nowych poziomów parametry te należy zrewidować.

Notacja i równania

Widziałeś właśnie przykład procesu uczenia się w sieci ze sprzężeniem do przodu z propagacją wsteczną, opisany w odniesieniu do jednego neuronu warstwy ukrytej i jednego neuronu wejściowego. Było kilka wektorów, które zostały pokazane i użyte, ale być może nie były łatwe do zidentyfikowania. Dlatego wprowadzamy notację i opisujemy równania, które zostały użyte w tym przykładzie.

Notacja

Porozmawiajmy o dwóch macierzach, których elementami są wagi połączeń. Jedna macierz odnosi się do interfejsu między warstwą wejściową i ukrytą, a druga odnosi się do interfejsu między warstwą ukrytą a warstwą wyjściową. Ponieważ połączenia istnieją od każdego neuronu w jednej warstwie do każdego neuronu w następnej warstwie, istnieje wektor wag połączeń wychodzących z każdego neuronu. Umieszczając ten wektor w wierszu macierzy, otrzymujemy tyle wierszy, ile jest neuronów, z których nawiązywane są połączenia. Niech M_1 i M_2 będą tymi macierzami wag. Co zatem oznacza M_1[i][j]? Jest to waga połączenia od neuronu wejściowego i-tego do neuronu j-tego w warstwie ukrytej. Podobnie M_2[i][j] oznacza wagę połączenia z neuronu i-tego w warstwie ukrytej i neuronu wyjściowego j-tego. Następnie użyjemy x, y, z dla wyjść neuronów odpowiednio w warstwie wejściowej, warstwie ukrytej i warstwie wyjściowej, z dołączonym indeksem dolnym, aby wskazać, do którego neuronu w danej warstwie się odnosimy. Niech P oznacza pożądany wzór wyjściowy, z p_i jako składowymi. Niech m będzie liczbą neuronów wejściowych, aby zgodnie z naszą notacją $(x_1, x_2, \ldots, x_m)$ oznaczało wzorzec wejściowy. Jeśli P ma, powiedzmy, r składników , warstwa wyjściowa potrzebuje r neuronów. Niech liczba neuronów warstwy ukrytej będzie równa n. Niech β_h będzie parametrem szybkości uczenia się dla warstwy ukrytej, a β_o , dla warstwy wyjściowej. Niech θ z odpowiednim indeksem dolnym reprezentuje wartość progową lub odchylenie dla neuronu warstwy ukrytej, a τ z odpowiednim indeksem dolnym odnosi się do wartości progowej neuronu wyjściowego. Niech błędy na wyjściu w warstwie wyjściowej będą oznaczane przez ejs, a te w warstwie ukrytej przez t_i. Jeśli użyjemy prefiksu Δ dowolnego parametru, to patrzymy na zmianę lub dostosowanie tego parametru. Ponadto funkcja progowa, której użyjemy, to funkcja sigmoidalna, $f(x) = 1 / (1 + \exp(-x))$.

Równania

Wyjście neuronu warstwy ukrytej j-tej:

$$y_j = f((\Sigma_i x_i M_1[i][j]) + \theta j)$$

Wyjście neuronu j-tej warstwy wyjściowej:

$$z_j = f((\Sigma_i y_i M_2[i][j]) + \tau_j)$$

I składowa wektora różnic wyjściowych:

wartość żądana - wartość obliczona = $P_i - z_i$

I składnik błędu wyjściowego w warstwie wyjściowej:

$$e_i = (P_i - z_i)$$

I składnik błędu wyjściowego w warstwie ukrytej:

$$t_i = y_i (1 - y_i) (\Sigma j\, M_2[i][j]\, e_j)$$

Korekta wagi między i-tym neuronem w warstwie ukrytej a j-tym neuronem wyjściowym:

$$\Delta M_2[i][j] = \beta_o\, y_i e_j$$

Korekta wagi pomiędzy i-tym neuronem wejściowym a j-tym neuronem w warstwie ukrytej:

$$M_1[i][j] = \beta_h x_i t_j$$

Dopasowanie do wartości progowej lub odchylenia dla j-tego neuronu wyjściowego:

$$\Delta\, \theta_j = \beta_o\, e_j$$

Dopasowanie do wartości progowej lub odchylenia dla neuronu warstwy ukrytej j-tej:

$$\delta\theta_j = \beta_h\, e_j$$

Aby użyć parametru pędu, zamiast równań 5 i 6 użyj:

$$\Delta M_2[i][j] (t) = \beta_o\, y_i e_j + \alpha \Delta M_2[i][j] (t - 1)$$

i

$$\Delta M_1[i][j] (t) = \beta_h\, x_i t_j + \alpha \Delta M_1[i][j] (t - 1)$$

Implementacja w C++ symulatora wstecznej propagacji błędów

Symulator propagacji wstecznej z tego rozdziału ma następujące cele projektowe:

1. Pozwól użytkownikowi określić liczbę i rozmiar wszystkich warstw.

2. Zezwól na użycie jednej lub więcej ukrytych warstw.

3. Być w stanie zapisać i przywrócić stan sieci.

4. Uruchom z dowolnie dużego zestawu danych treningowych lub zestawu danych testowych.

5. Zapytaj użytkownika o kluczowe parametry sieci i symulacji.

6. Wyświetl kluczowe informacje na koniec symulacji.

7. Zademonstruj użycie niektórych funkcji C++.

Krótka wycieczka po tym, jak korzystać z symulatora

Aby zrozumieć kod C++, przyjrzyjmy się działaniu programu. W symulatorze istnieją dwa tryby pracy. Użytkownik jest najpierw pytany o żądany tryb pracy. Dostępne tryby to tryb treningowy i tryb nietreningowy (tryb testowy).

Tryb treningowy

Tutaj użytkownik udostępnia plik szkoleniowy w bieżącym katalogu o nazwie training.dat. Ten plik zawiera przykładowe pary lub wzorce. Każdy wzorzec ma zestaw wejść, po których następuje zestaw wyjść. Każda wartość jest oddzielona co najmniej jedną spacją. Zgodnie z konwencją możesz użyć kilku dodatkowych spacji, aby oddzielić wejścia od wyjść. Oto przykład pliku training.dat zawierającego dwa wzorce:

```
0.4 0.5 0.89           -0.4 -0.8
0.23 0.8 -0.3          0.6 0.34
```

W tym przykładzie pierwszy wzorzec ma wejścia 0,4, 0,5 i 0,89, a oczekiwany wynik wynosi –0,4 i –0,8. Drugi wzór ma wejścia 0,23, 0,8 i –0,3 oraz wyjścia 0,6 i 0,34. Ponieważ istnieją trzy wejścia i dwa wyjścia, rozmiar warstwy wejściowej dla sieci musi wynosić trzy neurony, a rozmiar warstwy wyjściowej musi wynosić dwa neurony. Innym plikiem używanym w szkoleniu jest plik wag. Gdy symulator osiągnie tolerancję błędu określoną przez użytkownika lub maksymalną liczbę iteracji, symulator zapisuje stan sieci, zapisując wszystkie jej wagi w pliku o nazwie weights.dat. Plik ten może być następnie użyty w kolejnym uruchomieniu symulatora w trybie nieszkoleniowym. Aby dać pewne wyobrażenie o tym, jak poradziła sobie sieć, na końcu symulacji przedstawiono informacje o całkowitym i średnim błędzie. Ponadto dane wyjściowe generowane przez sieć dla ostatniego wektora wzorca są udostępniane w pliku wyjściowym o nazwie output.dat.

Tryb nietreningowy (tryb testowy)

W tym trybie użytkownik dostarcza dane testowe do symulatora w pliku o nazwie test.dat. Ten plik zawiera tylko wzorce wejściowe. Po zastosowaniu tego pliku do już przeszkolonej sieci generowany jest plik output.dat, który zawiera dane wyjściowe z sieci dla wszystkich wzorców wejściowych. W tym trybie sieć przechodzi przez jeden cykl działania, obejmując wszystkie wzorce w pliku danych testowych. Aby uruchomić sieć, odczytywany jest plik wag weights.dat, który inicjuje stan sieci. Użytkownik musi podać te same parametry rozmiaru sieci, które są używane do uczenia sieci.

Operacja

Pierwszą rzeczą do zrobienia z symulatorem jest wytrenowanie sieci z wybraną architekturą. Możesz wybrać liczbę warstw i liczbę warstw ukrytych dla swojej sieci. Należy pamiętać, że rozmiary warstwy wejściowej i wyjściowej są podyktowane wzorcami wejściowymi, które przedstawiasz w sieci, oraz wynikami, których szukasz w sieci. Kiedy już zdecydujesz się na architekturę, na przykład prostą trójwarstwową sieć z jedną ukrytą warstwą, przygotowujesz dla niej dane szkoleniowe i zapisujesz je w pliku training.dat. Po tym jesteś gotowy do treningu. Podajesz symulatorowi następujące informacje:

• Tryb (wybierz 1 do treningu)

• Wartości tolerancji błędu i parametru szybkości uczenia, lambda lub beta

• Maksymalna liczba cykli lub przejść przez dane treningowe, które chcesz wypróbować

• Liczba warstw (od trzech do pięciu, trzy oznacza jedną ukrytą warstwę, a pięć oznacza trzy ukryte warstwy)

• Rozmiar każdej warstwy, od wejścia do wyjścia

Symulator następnie rozpoczyna trening i zgłasza aktualny numer cyklu oraz średni błąd dla każdego cyklu. Powinieneś obserwować błąd, aby zobaczyć, że ogólnie maleje z czasem. Jeśli tak nie jest, powinieneś zrestartować symulację, ponieważ rozpocznie się ona z zupełnie nowym zestawem losowych wag i da ci inne, być może lepsze rozwiązanie. Pamiętaj, że będą uzasadnione okresy, w których błąd może przez pewien czas wzrastać. Po zakończeniu symulacji zobaczysz informacje o liczbie użytych cykli i wzorców oraz całkowitym i średnim błędzie, który wynikł. Wagi są zapisywane w pliku weights.dat. Możesz zmienić nazwę tego pliku, aby później użyć tego konkretnego stanu sieci. Rozmiar i liczbę warstw można wywnioskować na podstawie informacji zawartych w tym pliku, jak zostanie to pokazane w następnej sekcji dotyczącej formatu pliku weights.dat. Możesz rzucić okiem na plik output.dat, aby zobaczyć rodzaj osiągniętego wyniku treningu. Aby uzyskać pełne rozliczenie każdego wzorca i dopasowanie do tego wzorca, skopiuj plik szkoleniowy do pliku testowego i usuń z niego informacje wyjściowe. Następnie możesz uruchomić tryb testowy, aby uzyskać pełną listę wszystkich bodźców wejściowych i odpowiedzi w pliku output.dat.

Podsumowanie plików używanych w symulatorze propagacji wstecznej

Oto lista plików w celach informacyjnych, a także do czego są używane.

• weights.dat Możesz spojrzeć na ten plik, aby zobaczyć wagi dla sieci. Pokazuje numer warstwy, a następnie wagi, które są wprowadzane do warstwy. Pierwsza warstwa, czyli warstwa wejściowa, warstwa zero, nie ma żadnych skojarzonych z nią wag. Przykład pliku weights.dat jest pokazany poniżej dla sieci z trzema warstwami o rozmiarach 3, 5 i 2. Zauważ, że szerokość wiersza dla warstwy n odpowiada długości kolumny dla warstwy n + 1:

```
1 -0.199660 -0.859660 -0.339660 -0.25966 0.520340
1  0.292860 -0.487140 0.212860 -0.967140 -0.427140
1  0.542106 -0.177894 0.322106 -0.977894 0.562106
2 -0.175350 -0.835350
2 -0.330167 -0.250167
2  0.503317 0.283317
2 -0.477158 0.222842
2 -0.928322 -0.388322
```

W tym pliku wag szerokość wiersza dla warstwy 1 wynosi 5, co odpowiada wynikowi tej (środkowej) warstwy. Dane wejściowe dla warstwy to długość kolumny, która wynosi 3, tak jak określono. W przypadku warstwy 2 rozmiarem wyjściowym jest szerokość wiersza, która wynosi 2, a rozmiarem wejściowym jest długość kolumny 5, która jest taka sama jak wynikowa dla warstwy środkowej. Możesz przeczytać plik z wagami, aby dowiedzieć się, jak wszystko wygląda.

• training.dat Ten plik zawiera wzorce wejściowe do szkolenia. Możesz mieć tak duży plik, jak chcesz, bez pogarszania wydajności symulatora. Symulator buforuje dane w pamięci do przetworzenia. Ma to na celu zwiększenie szybkości symulacji, ponieważ dostęp do dysku jest czasochłonny. Bufor danych o maksymalnym rozmiarze określonym w instrukcji #define w programie jest wypełniany danymi z pliku

training.dat zawsze, gdy potrzebne są dane. Format pliku training.dat został pokazany w sekcji Tryb treningu.

• test.dat Plik test.dat jest podobny do pliku training.dat, ale nie zawiera oczekiwanych wyników. Używasz tego pliku z wytrenowaną siecią neuronową w trybie testowym, aby zobaczyć, jakie odpowiedzi otrzymasz w przypadku niewytrenowanych danych.

• output.dat Plik output.dat zawiera wyniki symulacji. W trybie testowym wektory wejściowe i wyjściowe są wyświetlane dla wszystkich wektorów wzorców. W trybie symulatora wyświetlany jest również oczekiwany wynik, ale prezentowany jest tylko ostatni wektor w zbiorze uczącym, ponieważ zbiór uczący jest zwykle dość duży. Pokazano tutaj przykład pliku wyjściowego w trybie szkoleniowym:

```
for input vector:
0.400000  -0.400000
output vector is:
0.880095
expected output vector is:
0.900000
```

Klasy C++ i hierarchia klas

Do tej pory dowiedziałeś się, jak realizujemy większość celów nakreślonych w tym programie. Jedyny cel, jaki pozostał, to zademonstrowanie niektórych funkcji C++. W tym programie używamy hierarchii klas z funkcją dziedziczenia. Ponadto stosujemy polimorfizm z wiązaniem dynamicznym i przeciążanie funkcji z wiązaniem statycznym. Najpierw spójrzmy na hierarchię klas używaną w tym programie. Klasa abstrakcyjna to klasa, która nigdy nie ma być tworzona jako obiekt, ale służy jako klasa bazowa, z której inni mogą dziedziczyć funkcje i definicje interfejsów. Taką klasą jest klasa warstwy. Wkrótce zobaczysz, że jedna z jego funkcji jest ustawiona = zero, co wskazuje, że ta klasa jest abstrakcyjną klasą bazową. Z klasy warstwy są dwie gałęzie. Jedna to klasa input_layer, a druga to klasa output_layer. Klasa warstwy środkowej jest bardzo podobna do warstwy wyjściowej w funkcji, a więc dziedziczy po klasie warstwa_wyjściowa. Przeciążanie funkcji można zobaczyć w definicji funkcji calc_error(). Jest używany w warstwie_wejściowej bez parametrów, podczas gdy jest używany w warstwie_wyjściowej (z której dziedziczy warstwa_wejściowa) z jednym parametrem. Używanie tej samej nazwy funkcji nie stanowi problemu i określa się to jako przeciążenie. Oprócz przeciążania funkcji, możesz również mieć przeciążanie operatora, które wykorzystuje operator, który wykonuje jakąś znajomą funkcję, taką jak + dla dodawania, dla innej funkcji, powiedzmy, dodawania wektorów. Jeśli masz przeciążenie z tymi samymi parametrami i słowem kluczowym virtual, masz możliwość dynamicznego wiązania, co oznacza, że określasz, która przeciążona funkcja ma być wykonywana w czasie wykonywania, a nie w czasie kompilacji. Wiązanie w czasie kompilacji jest określane jako wiązanie statyczne. Jeśli umieścisz kilka obiektów C++ w tablicy wskaźników do klasy bazowej, a następnie przejdziesz przez pętlę, która indeksuje każdy wskaźnik i wykonuje przeciążoną funkcję wirtualną, na którą wskazuje wskaźnik, wtedy będziesz używał wiązania dynamicznego. Tak jest dokładnie w przypadku funkcji calc_out(), która jest deklarowana za pomocą słowa kluczowego virtual w klasie bazowej warstwy. Każdy potomek warstwy może dostarczyć wersję funkcji calc_out(), która różni się funkcjonalnością od klasy bazowej, a właściwa funkcja zostanie wybrana w czasie wykonywania na podstawie tożsamości obiektu. W tym przypadku funkcja calc_out(), która jest funkcją obliczania wyników dla każdej warstwy, jest inna dla warstwy wejściowej niż dla pozostałych dwóch typów warstw. Przyjrzyjmy się niektórym szczegółom w pliku nagłówkowym z listingu 1:

Listing 1 Plik nagłówkowy dla symulatora wstecznej propagacji błędów

```
// layer.h V.Rao, H. Rao
// header file for the layer class hierarchy and
// the network class
#define MAX_LAYERS 5
#define MAX_VECTORS 100
class network;
class layer
{
protected:
int num_inputs;
int num_outputs;
float *outputs;// pointer to array of outputs
float *inputs; // pointer to array of inputs, which
// are outputs of some other layer
friend network;
public:
virtual void calc_out()=0;
};
class input_layer: public layer
{
private:
public:
input_layer(int, int);
~input_layer();
virtual void calc_out();
};
class middle_layer;
class output_layer: public layer
{
protected:
```

```
float * weights;
float * output_errors; // array of errors at output
float * back_errors; // array of errors back-propagated
float * expected_values; // to inputs
friend network;
public:
output_layer(int, int);
~output_layer();
virtual void calc_out();
void calc_error(float &);
void randomize_weights();
void update_weights(const float);
void list_weights();
void write_weights(int, FILE *);
void read_weights(int, FILE *);
void list_errors();
void list_outputs();
};
class middle_layer: public output_layer
{
private:
public:
middle_layer(int, int);
~middle_layer();
void calc_error();
};
class network
{
private:
layer *layer_ptr[MAX_LAYERS];
int number_of_layers;
```

```
int layer_size[MAX_LAYERS];
float *buffer;
fpos_t position;
unsigned training;
public:
network();
~network();
void set_training(const unsigned &);
unsigned get_training_value();
void get_layer_info();
void set_up_network();
void randomize_weights();
void update_weights(const float);
void write_weights(FILE *);
void read_weights(FILE *);
void list_weights();
void write_outputs(FILE *);
void list_outputs();
void list_errors();
void forward_prop();
void backward_prop(float &);
int fill_IObuffer(FILE *);
void set_up_pattern(int);
};
```

Szczegóły pliku nagłówka propagacji wstecznej

Na górze pliku znajdują się dwie instrukcje #define, które służą do ustawienia maksymalnej liczby warstw, które można użyć, obecnie pięć, oraz maksymalnej liczby wektorów uczących lub testowych, które można wczytać do we/wy bufor. Obecnie jest to 100. Możesz zwiększyć rozmiar bufora, aby uzyskać większą szybkość, kosztem zwiększonego użycia pamięci. Poniżej znajdują się definicje w klasie bazowej warstwy. Zwróć uwagę, że liczba wejść i wyjść to chronione składowe danych, co oznacza, że potomkowie klasy mogą mieć do nich swobodny dostęp.

```
int num_inputs;
int num_outputs;
```

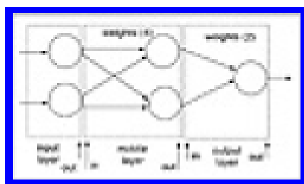
float *outputs; // pointer to array of outputs

float *inputs; // pointer to array of inputs, which

// are outputs of some other layer

friend network;

W tej klasie są również dwa wskaźniki do tablic pływaków. Są wskaźnikami do wyjść w danej warstwie i wejść do danej warstwy. Aby lepiej zorientować się, co obejmuje warstwa, na rysunku 7.3 pokazano małą sieć ze sprzężeniem do przodu z propagacją wsteczną z linią przerywaną, która pokazuje trzy warstwy dla tej sieci.



Warstwa zawiera neurony i wagi. Warstwa odpowiada za obliczenie swojego wyjścia (calc_out()), przechowywanego w tablicy float * outputs oraz błędów (calc_error()) dla każdego z odpowiednich neuronów. Błędy są przechowywane w innej tablicy o nazwie float * output_errors zdefiniowanej w klasie wyjściowej. Zauważ, że z klasą wejściową nie są skojarzone żadne wagi i dlatego jest to przypadek szczególny. Nie musi dostarczać żadnych członków danych ani członków funkcji związanych z błędami lub wsteczną propagacją. Jedynym celem warstwy wejściowej jest przechowywanie danych, które mają być propagowane do następnej warstwy.

W warstwie wyjściowej jest jeszcze kilka tablic. Po pierwsze, do przechowywania błędów wstecznie propagowanych istnieje tablica o nazwie float * back_errors. Istnieje tablica wag o nazwie float * weights, a do przechowywania oczekiwanych wartości, które inicjują proces obliczania błędów, służy tablica o nazwie float * oczekiwane_wartości. Zauważ, że warstwa środkowa potrzebuje prawie wszystkich tych tablic i dziedziczy je jako klasa pochodna klasy output_layer. Poza klasą warstwy i jej potomkami zdefiniowanymi w tym pliku nagłówkowym istnieje jeszcze jedna klasa, a jest to klasa sieci, która służy do konfigurowania kanałów komunikacji między warstwami oraz do dostarczania i usuwania danych z sieci. Klasa sieciowa wykonuje wzajemne połączenie warstw poprzez ustawienie wskaźnika tablicy wejściowej danej warstwy na tablicę wyjściową poprzedniej warstwy. Jest to dość rozszerzalny schemat, który można wykorzystać na przykład do tworzenia wariacji w sieci ze sprzężeniem do przodu z propagacją wsteczną z połączeniami sprzężenia zwrotnego. Innym połączeniem, za które odpowiada klasa sieciowa, jest ustawienie wskaźnika tablicy output_error na tablicę back_error następnej warstwy (pamiętaj, że błędy przepływają w odwrotnym kierunku, a tablica back_error jest błędem wyjściowym warstwy odzwierciedlonym na jej wejściach). Klasa sieciowa przechowuje tablicę wskaźników do warstw oraz tablicę rozmiarów warstw dla wszystkich zdefiniowanych warstw. Te obiekty warstw i tablice są dynamicznie alokowane na stercie za pomocą funkcji New i Delete w C++. Istnieje pewne minimalne sprawdzanie błędów we/wy plików i alokacji pamięci, które można poprawić w razie potrzeby. Jak widać, sieć ze sprzężeniem do przodu z propagacją wsteczną może szybko stać się świnią pamięci i procesora, z dużymi sieciami i dużymi zestawami treningowymi. Rozmiar i topologia sieci lub architektura będą w dużej mierze dyktować obie te cechy.

Szczegóły pliku implementacji wstecznej propagacji błędów

Kolejnym tematem jest implementacja klas i metod. Przyjrzyjmy się plikowi layer.cpp z listingu 2.

Listing 2 plik implementacji warstwy.cpp dla symulatora wstecznej propagacji błędów

```
// layer.cpp V.Rao, H.Rao
// compile for floating point hardware if available
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "layer.h"
inline float squash(float input)
// squashing function
// use sigmoid -- can customize to something
// else if desired; can add a bias term too
//
{
if (input < -50)
return 0.0;
else if (input > 50)
return 1.0;
else return (float)(1/(1+exp(-(double)input)));
}
inline float randomweight(unsigned init)
{
int num;
// random number generator
// will return a floating point
// value between -1 and 1
if (init==1) // seed the generator
srand ((unsigned)time(NULL));
num=rand() % 100;
```

```
return 2*(float(num/100.00))-1;
}
// the next function is needed for Turbo C++
// and Borland C++ to link in the appropriate
// functions for fscanf floating point formats:
static void force_fpf()
{
float x, *y;
y=&x;
x=*y;
}
// --------------------
// input layer
//---------------------
input_layer::input_layer(int i, int o)
{
num_inputs=i;
num_outputs=o;
outputs = new float[num_outputs];
if (outputs==0)
{
cout << "not enough memory\n";
cout << "choose a smaller architecture\n";
exit(1);
}
}
input_layer::~input_layer()
{
delete [num_outputs] outputs;
}
void input_layer::calc_out()
```

```
{
//nothing to do, yet
}
// --------------------
// output layer
//---------------------
output_layer::output_layer(int i, int o)
{
num_inputs =i;
num_outputs =o;
weights = new float[num_inputs*num_outputs];
output_errors = new float[num_outputs];
back_errors = new float[num_inputs];
outputs = new float[num_outputs];
expected_values = new float[num_outputs];
if ((weights==0)||(output_errors==0)||(back_errors==0)
||(outputs==0)||(expected_values==0))
{
cout << "not enough memory\n";
cout << "choose a smaller architecture\n";
exit(1);
}
}
output_layer::~output_layer()
{
// some compilers may require the array
// size in the delete statement; those
// conforming to Ansi C++ will not
delete [num_outputs*num_inputs] weights;
delete [num_outputs] output_errors;
delete [num_inputs] back_errors;
```

```
delete [num_outputs] outputs;
}
void output_layer::calc_out()
{
int i,j,k;
float accumulator=0.0;
for (j=0; j<num_outputs; j++)
{
for (i=0; i<num_inputs; i++)
{
k=i*num_outputs;
if (weights[k+j]*weights[k+j] > 1000000.0)
{
cout << "weights are blowing up\n";
cout << "try a smaller learning constant\n";
cout << "e.g. beta=0.02 aborting...\n";
exit(1);
}
outputs[j]=weights[k+j]*(*(inputs+i));
accumulator+=outputs[j];
}
// use the sigmoid squash function
outputs[j]=squash(accumulator);
accumulator=0;
}
}
void output_layer::calc_error(float & error)
{
int i, j, k;
float accumulator=0;
float total_error=0;
```

```
for (j=0; j<num_outputs; j++)
{
output_errors[j] = expected_values[j]-outputs[j];
total_error+=output_errors[j];
}
error=total_error;
for (i=0; i<num_inputs; i++)
{
k=i*num_outputs;
for (j=0; j<num_outputs; j++)
{
back_errors[i]=
weights[k+j]*output_errors[j];
accumulator+=back_errors[i];
}
back_errors[i]=accumulator;
accumulator=0;
// now multiply by derivative of
// sigmoid squashing function, which is
// just the input*(1-input)
back_errors[i]*=(*(inputs+i))*(1-(*(inputs+i)));
}
}
void output_layer::randomize_weights()
{
int i, j, k;
const unsigned first_time=1;
const unsigned not_first_time=0;
float discard;
discard=randomweight(first_time);
for (i=0; i< num_inputs; i++)
```

```
{
k=i*num_outputs;
for (j=0; j< num_outputs; j++)
weights[k+j]=randomweight(not_first_time);
}
}
void output_layer::update_weights(const float beta)
{
int i, j, k;
// learning law: weight_change =
// beta*output_error*input
for (i=0; i< num_inputs; i++)
{
k=i*num_outputs;
for (j=0; j< num_outputs; j++)
weights[k+j] +=
beta*output_errors[i]*(*(inputs+i));
}
}
void output_layer::list_weights()
{
int i, j, k;
for (i=0; i< num_inputs; i++)
{
k=i*num_outputs;
for (j=0; j< num_outputs; j++)
cout << "weight["<<i<<","<<
j<<"] is: "<<weights[k+j];
}
}
void output_layer::list_errors()
```

```
{
int i, j;
for (i=0; i< num_inputs; i++)
cout << "backerror["<<i<<
"] is : "<<back_errors[i]<<"\n";
for (j=0; j< num_outputs; j++)
cout << "outputerrors["<<j<<
"] is: "<<output_errors[j]<<"\n";
}
void output_layer::write_weights(int layer_no,
FILE * weights_file_ptr)
{
int i, j, k;
// assume file is already open and ready for
// writing
// prepend the layer_no to all lines of data
// format:
// layer_no weight[0,0] weight[0,1] ...
// layer_no weight[1,0] weight[1,1] ...
// ...
for (i=0; i< num_inputs; i++)
{
fprintf(weights_file_ptr,"%i ",layer_no);
k=i*num_outputs;
for (j=0; j< num_outputs; j++)
{
fprintf(weights_file_ptr,"%f",
weights[k+j]);
}
fprintf(weights_file_ptr,"\n");
}
```

```
}
void output_layer::read_weights(int layer_no,
FILE * weights_file_ptr)
{
int i, j, k;
// assume file is already open and ready for
// reading
// look for the prepended layer_no
// format:
// layer_no weight[0,0] weight[0,1] ...
// layer_no weight[1,0] weight[1,1] ...
// ...
while (1)
{
fscanf(weights_file_ptr,"%i",&j);
if ((j==layer_no)|| (feof(weights_file_ptr)))
break;
else
{
while (fgetc(weights_file_ptr) != `\n')
{;}// get rest of line
}
}
if (!(feof(weights_file_ptr)))
{
// continue getting first line
i=0;
for (j=0; j< num_outputs; j++)
{
fscanf(weights_file_ptr,"%f",
&weights[j]); // i*num_outputs = 0
```

```
}
fscanf(weights_file_ptr,"\n");
// now get the other lines
for (i=1; i< num_inputs; i++)
{
fscanf(weights_file_ptr,"%i",&layer_no);
k=i*num_outputs;
for (j=0; j< num_outputs; j++)
{
fscanf(weights_file_ptr,"%f",
&weights[k+j]);
}
}
fscanf(weights_file_ptr,"\n");
}
else cout << "end of file reached\n";
}
void output_layer::list_outputs()
{
int j;
for (j=0; j< num_outputs; j++)
{
cout << "outputs["<<j
<<"] is: "<<outputs[j]<<"\n";
}
}
// ---------------------
// middle layer
//----------------------
middle_layer::middle_layer(int i, int o):
output_layer(i,o)
```

```
{
}
middle_layer::~middle_layer()
{
delete [num_outputs*num_inputs] weights;
delete [num_outputs] output_errors;
delete [num_inputs] back_errors;
delete [num_outputs] outputs;
}
void middle_layer::calc_error()
{
int i, j, k;
float accumulator=0;
for (i=0; i<num_inputs; i++)
{
k=i*num_outputs;
for (j=0; j<num_outputs; j++)
{
back_errors[i]=
weights[k+j]*(*(output_errors+j));
accumulator+=back_errors[i];
}
back_errors[i]=accumulator;
accumulator=0;
// now multiply by derivative of
// sigmoid squashing function, which is
// just the input*(1-input)
back_errors[i]*=(*(inputs+i))*(1-(*(inputs+i)));
}
}
network::network()
```

```
{
position=0L;
}
network::~network()
{
int i,j,k;
i=layer_ptr[0]->num_outputs;// inputs
j=layer_ptr[number_of_layers-1]->num_outputs; //outputs
k=MAX_VECTORS;
delete [(i+j)*k]buffer;
}
void network::set_training(const unsigned & value)
{
training=value;
}
unsigned network::get_training_value()
{
return training;
}
void network::get_layer_info()
{
int i;
//---------------------
//
// Get layer sizes for the network
//
// --------------------
cout << " Please enter in the number of layers for your network.\n";
cout << " You can have a minimum of 3 to a maximum of 5. \n";
cout << " 3 implies 1 hidden layer; 5 implies 3 hidden layers : \n\n";
cin >> number_of_layers;
```

```
cout << " Enter in the layer sizes separated by spaces.\n";
cout << " For a network with 3 neurons in the input layer,\n";
cout << " 2 neurons in a hidden layer, and 4 neurons in the\n";
cout << " output layer, you would enter: 3 2 4 .\n";
cout << " You can have up to 3 hidden layers,for five maximum entries
:\n\n";
for (i=0; i<number_of_layers; i++)
{
cin >> layer_size[i];
}
// --------------------------
// size of layers:
// input_layer layer_size[0]
// output_layer layer_size[number_of_layers-1]
// middle_layers layer_size[1]
// optional: layer_size[number_of_layers-3]
// optional: layer_size[number_of_layers-2]
//---------------------------
}
void network::set_up_network()
{
int i,j,k;
//---------------------------
// Construct the layers
//
//---------------------------
layer_ptr[0] = new input_layer(0,layer_size[0]);
for (i=0;i<(number_of_layers-1);i++)
{
layer_ptr[i+1y] =
new middle_layer(layer_size[i],layer_size[i+1]);
```

```
}
layer_ptr[number_of_layers-1] = new
output_layer(layer_size[number_of_layers-2],layer_size[number_of_layers-
1]);
for (i=0;i<(number_of_layers-1);i++)
{
if (layer_ptr[i] == 0)
{
cout << "insufficient memory\n";
cout << "use a smaller architecture\n";
exit(1);
}
}
//--------------------------
// Connect the layers
//
//--------------------------
// set inputs to previous layer outputs for all layers,
// except the input layer
for (i=1; i< number_of_layers; i++)
layer_ptr[i]->inputs = layer_ptr[i-1y]->outputs;
// for back_propagation, set output_errors to next layer
// back_errors for all layers except the output
// layer and input layer
for (i=1; i< number_of_layers -1; i++)
((output_layer *)layer_ptr[i])->output_errors =
((output_layer *)layer_ptr[i+1])->back_errors;
// define the IObuffer that caches data from
// the datafile
i=layer_ptr[0]->num_outputs;// inputs
j=layer_ptr[number_of_layers-1]->num_outputs; //outputs
```

```
k=MAX_VECTORS;
buffer=new
float[(i+j)*k];
if (buffer==0)
cout << "insufficient memory for buffer\n";
}
void network::randomize_weights()
{
int i;
for (i=1; i<number_of_layers; i++)
((output_layer *)layer_ptr[i])
->randomize_weights();
}
void network::update_weights(const float beta)
{
int i;
for (i=1; i<number_of_layers; i++)
((output_layer *)layer_ptr[i])
->update_weights(beta);
}
void network::write_weights(FILE * weights_file_ptr)
{
int i;
for (i=1; i<number_of_layers; i++)
((output_layer *)layer_ptr[i])
->write_weights(i,weights_file_ptr);
}
void network::read_weights(FILE * weights_file_ptr)
{
int i;
for (i=1; i<number_of_layers; i++)
```

```
((output_layer *)layer_ptr[i])
->read_weights(i,weights_file_ptr);
}
void network::list_weights()
{
int i;
for (i=1; i<number_of_layers; i++)
{
cout << "layer number : " <<i<< "\n";
((output_layer *)layer_ptr[i])
->list_weights();
}
}
void network::list_outputs()
{
int i;
for (i=1; i<number_of_layers; i++)
{
cout << "layer number : " <<i<< "\n";
((output_layer *)layer_ptr[i])
->list_outputs();
}
}
void network::write_outputs(FILE *outfile)
{
int i, ins, outs;
ins=layer_ptr[0]->num_outputs;
outs=layer_ptr[number_of_layers-1]->num_outputs;
float temp;
fprintf(outfile,"for input vector:\n");
for (i=0; i<ins; i++)
```

```
{
temp=layer_ptr[0]->outputs[i];
fprintf(outfile,"%f ",temp);
}
fprintf(outfile,"\noutput vector is:\n");
for (i=0; i<outs; i++)
{
temp=layer_ptr[number_of_layers-1]->
outputs[i];
fprintf(outfile,"%f ",temp);
}
if (training==1)
{
fprintf(outfile,"\nexpected output vector is:\n");
for (i=0; i<outs; i++)
{
temp=((output_layer *)(layer_ptr[number_of_layers-1]))->
expected_values[i];
fprintf(outfile,"%f ",temp);
}
}
fprintf(outfile,"\n----------\n");
}
void network::list_errors()
{
int i;
for (i=1; i<number_of_layers; i++)
{
cout << "layer number : " <<i<< "\n";
((output_layer *)layer_ptr[i])
->list_errors();
```

```
}
}
int network::fill_IObuffer(FILE * inputfile)
{
// this routine fills memory with
// an array of input, output vectors
// up to a maximum capacity of
// MAX_INPUT_VECTORS_IN_ARRAY
// the return value is the number of read
// vectors
int i, k, count, veclength;
int ins, outs;
ins=layer_ptr[0]->num_outputs;
outs=layer_ptr[number_of_layers-1]->num_outputs;
if (training==1)
veclength=ins+outs;
else
veclength=ins;
count=0;
while ((count<MAX_VECTORS)&&
(!feof(inputfile)))
{
k=count*(veclength);
for (i=0; i<veclength; i++)
{
fscanf(inputfile,"%f",&buffer[k+i]);
}
fscanf(inputfile,"\n");
count++;
}
if (!(ferror(inputfile)))
```

```
return count;
else return -1; // error condition
}
void network::set_up_pattern(int buffer_index)
{
// read one vector into the network
int i, k;
int ins, outs;
ins=layer_ptr[0]->num_outputs;
outs=layer_ptr[number_of_layers-1]->num_outputs;
if (training==1)
k=buffer_index*(ins+outs);
else
k=buffer_index*ins;
for (i=0; i<ins; i++)
layer_ptr[0]->outputs[i]=buffer[k+i];
if (training==1)
{
for (i=0; i<outs; i++)
((output_layer *)layer_ptr[number_of_layers-1])->
expected_values[i]=buffer[k+i+ins];
}
}
void network::forward_prop()
{
int i;
for (i=0; i<number_of_layers; i++)
{
layer_ptr[i]->calc_out(); //polymorphic
// function
}
```

```
}
void network::backward_prop(float & toterror)
{
int i;
// error for the output layer
((output_layer*)layer_ptr[number_of_layers-1])->
calc_error(toterror);
// error for the middle layer(s)
for (i=number_of_layers-2; i>0; i--)
{
((middle_layer*)layer_ptr[i])->
calc_error();
}
```

Spójrz na funkcje w pliku layer.cpp

Poniżej znajduje się lista funkcji w pliku layer.cpp wraz z krótkim opisem celu każdej z nich.

• void set_training(const unsigned &) Ustawia wartość prywatnego członka danych, szkolenia; użyj 1 dla trybu treningowego i 0 dla trybu testowego.

• unsigned get_training_value() Pobiera wartość stałej szkoleniowej określającej używany tryb.

• void get_layer_info() Pobiera od użytkownika informacje o liczbie warstw i ich rozmiarach.

• void set_up_network() Ta procedura ustawia połączenia między warstwami poprzez odpowiednie przypisanie wskaźników.

• void randomize_weights() Na początku procesu uczenia ta procedura jest używana do losowania wszystkich wag w sieci.

• void update_weights(const float) W ramach treningu wagi są aktualizowane zgodnie z prawem uczenia się stosowanym w propagacji wstecznej.

• void write_weights(FILE *) Ta procedura służy do zapisywania wag do pliku.

• void read_weights(FILE *) Ta procedura służy do odczytywania wag do sieci z pliku.

• void list_weights() Ta procedura może być użyta do wylistowania wag podczas trwania symulacji.

• void write_outputs(FILE*) Ta procedura zapisuje dane wyjściowe sieci do pliku.

• void list_outputs() Ta procedura może być użyta do wylistowania wyjść sieci podczas trwania symulacji.

• void list_errors() Wyświetla listę błędów dla wszystkich warstw podczas trwania symulacji.

• void forward_prop() Wykonuje propagację w przód.

• void reverse_prop(float &) Wykonuje wsteczną propagację błędów.

• int fill_IObuffer(FILE *) Ta procedura wypełnia wewnętrzny bufor IO danymi z zestawów danych treningowych lub testowych.

• void set_up_pattern(int) Ta procedura jest używana do ustawienia jednego wzorca z bufora IO do treningu.

• inline float squash (wejście float) Ta funkcja wykonuje funkcję sigmoid.

• inline float randomweight (jednostka bez znaku) Ta procedura zwraca losową wagę z zakresu od -1 do 1; użyj 1, aby zainicjować generator, a 0 dla wszystkich kolejnych wywołań.

Ostatnim plikiem do przejrzenia jest plik backprop.cpp przedstawiony na listingu 3.

Listing 3 Plik backprop.cpp dla symulatora wstecznej propagacji błędów

```
// backprop.cpp V. Rao, H. Rao

#include "layer.cpp"

#define TRAINING_FILE "training.dat"

#define WEIGHTS_FILE "weights.dat"

#define OUTPUT_FILE "output.dat"

#define TEST_FILE "test.dat"

void main()

{

float error_tolerance =0.1;

float total_error =0.0;

float avg_error_per_cycle =0.0;

float error_last_cycle =0.0;

float avgerr_per_pattern =0.0; // for the latest cycle

float error_last_pattern =0.0;

float learning_parameter =0.02;

unsigned temp, startup;

long int vectors_in_buffer;

long int max_cycles;

long int patterns_per_cycle =0;

long int total_cycles, total_patterns;

int i;

// create a network object
```

```
network backp;
FILE * training_file_ptr, * weights_file_ptr, * output_file_ptr;
FILE * test_file_ptr, * data_file_ptr;
// open output file for writing
if ((output_file_ptr=fopen(OUTPUT_FILE,"w"))==NULL)
{
cout << "problem opening output file\n";
exit(1);
}
// enter the training mode : 1=training on 0=training off
cout << "------------------------\n";
cout << " C++ Neural Networks and Fuzzy Logic \n";
cout << " Backpropagation simulator \n";
cout << " version 1 \n";
cout << "------------------------\n";
cout << "Please enter 1 for TRAINING on, or 0 for off: \n\n";
cout << "Use training to change weights according to your\n";
cout << "expected outputs. Your training.dat file should contain\n";
cout << "a set of inputs and expected outputs. The number of\n";
cout << "inputs determines the size of the first (input) layer\n";
cout << "while the number of outputs determines the size of the\n";
cout << "last (output) layer :\n\n";
cin >> temp;
backp.set_training(temp);
if (backp.get_training_value() == 1)
{
cout << "--> Training mode is *ON*. weights will be saved\n";
cout << "in the file weights.dat at the end of the\n";
cout << "current set of input (training) data\n";
}
else
```

```
{
cout << "--> Training mode is *OFF*. weights will be loaded\n";
cout << "from the file weights.dat and the current\n";
cout << "(test) data set will be used. For the test\n";
cout << "data set, the test.dat file should contain\n";
cout << "only inputs, and no expected outputs.\n";
}
if (backp.get_training_value()==1)
{
// --------------------
// Read in values for the error_tolerance,
// and the learning_parameter
// --------------------
cout << " Please enter in the error_tolerance\n";
cout << " --- between 0.001 to 100.0, try 0.1 to start \n";
cout << "\n";
cout << "and the learning_parameter, beta\n";
cout << " --- between 0.01 to 1.0, try 0.5 to start -- \n\n";
cout << " separate entries by a space\n";
cout << " example: 0.1 0.5 sets defaults mentioned :\n\n";
cin >> error_tolerance >> learning_parameter;
//---------------------
// open training file for reading
//--------------------
if ((training_file_ptr=fopen(TRAINING_FILE,"r"))==NULL)
{
cout << "problem opening training file\n";
exit(1);
}
data_file_ptr=training_file_ptr; // training on
// Read in the maximum number of cycles
```

```
// each pass through the input data file is a cycle
cout << "Please enter the maximum cycles for the simula-\
tion\n";
cout << "A cycle is one pass through the data set.\n";
cout << "Try a value of 10 to start with\n";
cin >> max_cycles;
}
else
{
if ((test_file_ptr=fopen(TEST_FILE,"r"))==NULL)
{
cout << "problem opening test file\n";
exit(1);
}
data_file_ptr=test_file_ptr; // training off
}
//
// training: continue looping until the total error is less than
// the tolerance specified, or the maximum number of
// cycles is exceeded; use both the forward signal propaga
tion
// and the backward error propagation phases. If the error
// tolerance criteria is satisfied, save the weights in a
file.
// no training: just proceed through the input data set once in the
// forward signal propagation phase only. Read the starting
// weights from a file.
// in both cases report the outputs on the screen
// initialize counters
total_cycles=0; // a cycle is once through all the input data
total_patterns=0; // a pattern is one entry in the input data
```

```
// get layer information
backp.get_layer_info();
// set up the network connections
backp.set_up_network();
// initialize the weights
if (backp.get_training_value()==1)
{
// randomize weights for all layers; there is no
// weight matrix associated with the input layer
// weight file will be written after processing
// so open for writing
if ((weights_file_ptr=fopen(WEIGHTS_FILE,"w"))
==NULL)
{
cout << "problem opening weights file\n";
exit(1);
}
backp.randomize_weights();
}
else
{
// read in the weight matrix defined by a
// prior run of the backpropagation simulator
// with training on
if ((weights_file_ptr=fopen(WEIGHTS_FILE,"r"))
==NULL)
{
cout << "problem opening weights file\n";
exit(1);
}
backp.read_weights(weights_file_ptr);
```

```
}
// main loop
// if training is on, keep going through the input data
// until the error is acceptable or the maximum number of
// cycles
// is exceeded.
// if training is off, go through the input data once. report // outputs
// with inputs to file output.dat
startup=1;
vectors_in_buffer = MAX_VECTORS; // startup condition
total_error = 0;
while ( ((backp.get_training_value()==1)
&& (avgerr_per_pattern
> error_tolerance)
&& (total_cycles < max_cycles)
&& (vectors_in_buffer !=0))
|| ((backp.get_training_value()==0)
&& (total_cycles < 1))
|| ((backp.get_training_value()==1)
&& (startup==1))
)
{
startup=0;
error_last_cycle=0; // reset for each cycle
patterns_per_cycle=0;
// process all the vectors in the datafile
// going through one buffer at a time
// pattern by pattern
while ((vectors_in_buffer==MAX_VECTORS))
{
vectors_in_buffer=
```

```
backp.fill_IObuffer(data_file_ptr); // fill buffer
if (vectors_in_buffer < 0)
{
cout << "error in reading in vectors, aborting\n";
cout << "check that there are no extra
linefeeds\n";
cout << "in your data file, and that the
number\n";
cout << "of layers and size of layers match
the\n";
cout << "the parameters provided.\n";
exit(1);
}
// process vectors
for (i=0; i<vectors_in_buffer; i++)
{
// get next pattern
backp.set_up_pattern(i);
total_patterns++;
patterns_per_cycle++;
// forward propagate
backp.forward_prop();
if (backp.get_training_value()==0)
backp.write_outputs(output_file_ptr);
// back_propagate, if appropriate
if (backp.get_training_value()==1)
{
backp.backward_prop(error_last_pattern);
error_last_cycle += error_last_pattern
z *error_last_pattern;
backp.update_weights(learning_parameter);
```

```
// backp.list_weights();
// can
// see change in weights by
// using list_weights before and
// after back_propagation
}
}
error_last_pattern = 0;
}
avgerr_per_pattern=((float)sqrt((double)error_last_cycle
/patterns_per_cycle));
total_error += error_last_cycle;
total_cycles++;
// most character displays are 26 lines
// user will see a corner display of the cycle count
// as it changes
cout << "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n";
cout << total_cycles << "\t" << avgerr_per_pattern << "\n";
fseek(data_file_ptr, 0L, SEEK_SET); // reset the file pointer
// to the beginning of
// the file
vectors_in_buffer = MAX_VECTORS; // reset
} // end main loop
cout << "\n\n\n\n\n\n\n\n\n\n\n";
cout << "-----------------------\n";
cout << " done: results in file output.dat\n";
cout << " training: last vector only\n";
cout << " not training: full cycle\n\n";
if (backp.get_training_value()==1)
{
backp.write_weights(weights_file_ptr);
```

```
backp.write_outputs(output_file_ptr);
avg_error_per_cycle = (float)sqrt((double)total_error/
total_cycles);
error_last_cycle = (float)sqrt((double)error_last_cycle);
cout << " weights saved in file weights.dat\n";
cout << "\n";
cout << "-->average error per cycle = " <<
avg_error_per_cycle << " <-\n";
cout << "-->error last cycle= " << error_last_cycle << " <-\n";
cout << "->error last cycle per pattern= " << avgerr_per_pattern << " <-
\n";
}
cout << "------>total cycles = " << total_cycles << " <--\n";
cout << "------>total patterns = " << total_patterns << " <---\n";
cout << "-------------------------\n";
// close all files
fclose(data_file_ptr);
fclose(weights_file_ptr);
fclose(output_file_ptr);
}
```

Plik backprop.cpp implementuje kontrolki symulatora. Najpierw pobierane są dane od użytkownika dla parametrów sieci. Zakładając, że używany jest tryb treningowy, plik treningowy jest otwierany i dane są odczytywane z pliku w celu wypełnienia bufora we/wy. Następnie wykonywana jest główna pętla, w której sieć przetwarza wzorzec po wzorcu, aby zakończyć cykl, czyli jedno przejście przez cały zestaw danych uczących. (Bufor IO jest uzupełniany zgodnie z wymaganiami podczas tego procesu.) Po wykonaniu jednego cyklu wskaźnik pliku jest resetowany na początek pliku i rozpoczyna się kolejny cykl. Symulator kontynuuje cykle aż do spełnienia jednego z dwóch podstawowych kryteriów:

1. Przekroczono maksymalną liczbę cykli określoną przez użytkownika.

2. Średni błąd na wzór dla ostatniego cyklu jest mniejszy niż tolerancja błędu określona przez użytkownika.

W przypadku wystąpienia jednej z tych sytuacji symulator zatrzymuje się i zgłasza osiągnięty błąd oraz zapisuje wagi w pliku weights.dat i jeden wektor wyjściowy w pliku output.dat. W trybie testowym sieć przetwarza dokładnie jeden cykl, a dane wyjściowe są zapisywane w pliku output.dat. Na początku symulacji w trybie testowym sieć jest konfigurowana z wagami z pliku weights.dat. Aby uprościć

program, użytkownik jest proszony o wprowadzenie liczby warstw i rozmiaru warstw, chociaż program może to wywnioskować z pliku wag.

Kompilowanie i uruchamianie symulatora propagacji wstecznej

Kompilacja pliku backprop.cpp spowoduje skompilowanie symulatora, ponieważ layer.cpp jest zawarty w backprop.cpp. Aby uruchomić symulator, po utworzeniu pliku wykonywalnego (przy użyciu sprzętu zmiennoprzecinkowego 80X87, jeśli jest dostępny), wpisz backprop i zobacz następujący ekran:

C++ Neural Networks and Fuzzy Logic

Backpropagation simulator

version 1

Please enter 1 for TRAINING on, or 0 for off:

Use training to change weights according to your

expected outputs. Your training.dat file should contain

a set of inputs and expected outputs. The number of

inputs determines the size of the first (input) layer

while the number of outputs determines the size of the

last (output) layer :

1

-> Training mode is *ON*. weights will be saved

in the file weights.dat at the end of the

current set of input (training) data

Please enter in the error_tolerance

-- between 0.001 to 100.0, try 0.1 to start --

and the learning_parameter, beta

-- between 0.01 to 1.0, try 0.5 to start --

separate entries by a space

example: 0.1 0.5 sets defaults mentioned :

0.2 0.25

Please enter the maximum cycles for the simulation

A cycle is one pass through the data set.

Try a value of 10 to start with

Please enter in the number of layers for your network.

You can have a minimum of three to a maximum of five.

three implies one hidden layer; five implies three hidden layers:

3

Enter in the layer sizes separated by spaces.

For a network with three neurons in the input layer,

two neurons in a hidden layer, and four neurons in the

output layer, you would enter: 3 2 4.

You can have up to three hidden layers for five maximum entries :

2 2 1

1 0.353248

2 0.352684

3 0.352113

4 0.351536

5 0.350954

...

299 0.0582381

300 0.0577085

----

done: results in file output.dat

training: last vector only

not training: full cycle

weights saved in file weights.dat

-->average error per cycle = 0.20268 <--

-->error last cycle = 0.0577085 <--

->error last cycle per pattern= 0.0577085 <--

------>total cycles = 300 <--

------>total patterns = 300 <--

Numer cyklu i średni błąd na wzór są wyświetlane w miarę postępu symulacji (nie wszystkie wartości są pokazane). Możesz to monitorować, aby upewnić się, że symulator zbliża się do rozwiązania. Jeśli błąd nie wydaje się zmniejszać poza pewien punkt, ale dryfuje lub wybucha, należy ponownie uruchomić symulator z nowym punktem początkowym zdefiniowanym przez inicjator losowych wag. Możesz także spróbować zmniejszyć rozmiar parametru szybkości uczenia się. Nauka może przebiegać wolniej, ale może to pozwolić na znalezienie lepszego minimum. Ten przykład pokazuje tylko jeden wzorzec w zestawie uczącym z dwoma danymi wejściowymi i jednym wyjściem. Wyniki wraz z (jednym) ostatnim wzorcem są pokazane w następujący sposób z pliku output.dat: dla wektora wejściowego:

0.400000 -0.400000

output vector is:

0.842291

expected output vector is:

0.900000

Dopasowanie jest całkiem dobre, jak można się spodziewać, ponieważ optymalizacja jest łatwa dla sieci; jest tylko jeden wzór, o który trzeba się martwić. Spójrzmy na ostateczny zestaw wag dla tej symulacji w weights.dat. Wagi te zostały uzyskane poprzez aktualizację wag dla 300 cykli zgodnie z prawem uczenia:

1 0,175039 0,435039

1 -1,319244 -0,559244

2 0,358281

2 2.421172

Na razie opuścimy symulator wstecznej propagacji i wrócimy do niego w kolejnym rozdziale w celu dalszej eksploracji. Z symulatorem możesz eksperymentować na wiele różnych sposobów:

• Wypróbuj inną liczbę warstw i rozmiary warstw dla danego problemu.

• Wypróbuj różne parametry szybkości uczenia się i zobacz ich wpływ na zbieżność i czas treningu.

• Wypróbuj bardzo duży parametr szybkości uczenia się (powinien wynosić od 0 do 1); spróbuj liczby powyżej 1 i zanotuj wynik.

Posumowanie

Poznałeś jeden z najpotężniejszych algorytmów sieci neuronowych, zwany propagacją wsteczną. Nie mając połączeń sprzężenia zwrotnego, propagując tylko błędy odpowiednio do połączeń warstwy ukrytej i warstwy wejściowej, algorytm wykorzystuje tzw. uogólnioną regułę delta i trenuje sieć przykładowymi parami wzorców. Trudno jest określić, ile neuronów warstwy ukrytej ma być dostarczonych. Liczba ukrytych warstw może być większa niż jedna. Ogólnie rzecz biorąc, rozmiar ukrytych warstw jest związany z cechami lub cechami wyróżniającymi, które należy odróżnić od danych. Nasz przykład tu odnosi się do prostego przypadku, w którym istnieje jedna ukryta warstwa. Wyjścia neuronów wyjściowych, a zatem i sieci, są wektorami o składowych od 0 do 1, ponieważ funkcja progowa jest funkcją sigmoidalną. W razie potrzeby wartości te można skalować, aby uzyskać wartości w innym przedziale. Nasz przykład nie odnosi się do żadnej konkretnej funkcji, która ma być obliczana przez sieć, ale wejścia i wyjścia zostały wybrane losowo. To może ci powiedzieć, że jeśli nie znasz równania funkcyjnego między dwoma zestawami wektorów, sieć sprzężenia zwrotnego propagacji wstecznej może znaleźć mapowanie dla dowolnego wektora w domenie, nawet jeśli równanie funkcjonalne nie zostanie znalezione. Z tego co wiemy, ta funkcja może być również nieliniowa. Jest jeden ważny fakt, o którym należy pamiętać o algorytmie wstecznej propagacji błędów. Jego najbardziej stroma procedura zejścia w treningu nie gwarantuje znalezienia globalnego lub ogólnego minimum, może znaleźć tylko lokalne minimum powierzchni energetycznej.