Propagacja wsteczna II

Ulepszanie symulatora

W Części 7 opracowałeś symulator wstecznej propagacji błędów. Tu wykorzystasz go na przykładach, a także dodasz kilka nowych funkcji do symulatora: termin zwany pędem oraz możliwość dodawania szumu do sygnałów wejściowych podczas symulacji. Istnieje wiele odmian tego algorytmu, które próbują złagodzić dwa problemy z propagacją wsteczną. Po pierwsze, podobnie jak w przypadku innych sieci neuronowych, istnieje duże prawdopodobieństwo, że rozwiązanie znalezione z propagacją wsteczną nie jest minimalnym globalnym błędem, ale lokalnym. Być może będziesz musiał trochę potrząsnąć ciężarkami, aby wydostać się z lokalnego minimum, i ewentualnie osiągnąć niższe minimum. Drugim problemem związanym z propagacją wsteczną jest szybkość. Algorytm uczy się bardzo wolno. Istnieje wiele propozycji przyspieszenia procesu wyszukiwania. Sieci neuronowe są z natury architekturami przetwarzania równoległego i nadają się do symulacji na sprzęcie do przetwarzania równoległego. Chociaż na rynku dostępnych jest kilka wtykowych sieci neuronowych lub kart do przetwarzania sygnałów cyfrowych, preferowaną niedrogą platformą symulacyjną pozostaje komputer osobisty. Ulepszenia szybkości algorytmu uczącego są zatem bardzo potrzebne.

Kolejny przykład wykorzystania wstecznej propagacji

Zanim zmodyfikujemy symulator w celu dodania funkcji, spójrzmy na ten sam problem, który wykorzystaliśmy do analizy mapy Kohonena w rozdziale 12. Jak pamiętasz, chcielibyśmy być w stanie rozróżniać znaki alfabetyczne, przypisując je do różnych pojemników. W przypadku propagacji wstecznej zastosujemy dane wejściowe i nauczymy sieć za pomocą oczekiwanych odpowiedzi. Oto plik wejściowy, którego użyliśmy do rozróżnienia pięciu różnych znaków, A, X, H, B i I:

```
0 0 1 0 0  0 1 0 1 0  1 0 0 0 1  1 0 0 0 1  1 1 1 1 1  1 0 0 0 1
1 0 0 0 1  0 1 0 1 0  0 0 1 0 0  0 0 1 0 0  0 0 1 0 0  0 1 0 1 0
1 0 0 0 1  1 0 0 0 1  1 0 0 0 1  1 1 1 1 1  1 0 0 0 1  1 0 0 0 1
1 1 1 1 1  1 0 0 0 1  1 0 0 0 1  1 1 1 1 1  1 0 0 0 1  1 0 0 0 1
0 0 1 0 0  0 0 1 0 0  0 0 1 0 0  0 0 1 0 0  0 0 1 0 0  0 0 1 0 0


1 0 0 0 1
1 0 0 0 1
1 0 0 0 1
1 1 1 1 1
0 0 1 0 0
```

Każda linia ma reprezentację 5x7 kropek każdego znaku. Teraz musimy nazwać każdą z kategorii wyjściowych. Prostą reprezentację 3-bitową możemy przypisać w następujący sposób:

A 000

X 010

H 100

B 101

I 111

Wytrenujmy sieć, aby rozpoznawała te postacie. Plik training.dat wygląda następująco.

```
0 0 1 0 0  0 1 0 1 0  1 0 0 0 1  1 0 0 0 1  1 1 1 1 1  1 0 0 0 1
1 0 0 0 1  0 1 0 1 0  0 0 1 0 0  0 0 1 0 0  0 0 1 0 0  0 1 0 1 0
1 0 0 0 1  1 0 0 0 1  1 0 0 0 1  1 1 1 1 1  1 0 0 0 1  1 0 0 0 1
1 1 1 1 1  1 0 0 0 1  1 0 0 0 1  1 1 1 1 1  1 0 0 0 1  1 0 0 0 1
0 0 1 0 0  0 0 1 0 0  0 0 1 0 0  0 0 1 0 0  0 0 1 0 0  0 0 1 0 0

1 0 0 0 1  0 0 0
1 0 0 0 1  0 1 0
1 0 0 0 1  1 0 0
1 1 1 1 1  1 0 1
0 0 1 0 0  1 1 1
```

Teraz możesz uruchomić symulator. Używając parametrów (beta = 0,1, tolerancja = 0,001 i max_cycles = 1000) oraz z trzema warstwami o rozmiarze 35 (wejście), 5 (środek) i 3 (wyjście), otrzymasz typowy wynik podobny do poniższego.

---------- -

done: results in file output.dat

training: last vector only

not training: full cycle

weights saved in file weights.dat

-->average error per cycle = 0.035713<--

-->error last cycle = 0.008223 <--

->error last cycle per pattern= 0.00164455 <--

------>total cycles = 1000 <--

------>total patterns = 5000 <--

Symulator zatrzymał się przy 1000 maksymalnych cykli określonych w tym przypadku. Twoje wyniki będą inne, ponieważ wagi zaczynają się od losowego punktu. Zwróć uwagę, że podana tolerancja została prawie spełniona. Zobaczmy, jak blisko wynik zbliżył się do tego, czego chcieliśmy. Spójrz na plik output.dat. Możesz zobaczyć dopasowanie do ostatniego wzoru w następujący sposób:

```
for input vector:
0.000000  0.000000  1.000000  0.000000  0.000000  0.000000  0.000000
1.000000  0.000000  0.000000  0.000000  0.000000  1.000000  0.000000
0.000000  0.000000  0.000000  1.000000  0.000000  0.000000  0.000000
0.000000  1.000000  0.000000  0.000000  0.000000  0.000000  1.000000
0.000000  0.000000  0.000000  0.000000  1.000000  0.000000  0.000000
output vector is:
0.999637  0.998721  0.999330
expected output vector is:
1.000000  1.000000  1.000000
-----------
```

Aby zobaczyć wyniki wszystkich wzorców, musimy skopiować plik training.dat do pliku test.dat i ponownie uruchomić symulator w trybie testowym. Pamiętaj, aby usunąć oczekiwane pole wyjściowe po skopiowaniu pliku. Uruchomienie symulatora w trybie testowym (0) powoduje wyświetlenie następującego wyniku w pliku output.dat:

```
for input vector:
0.000000  0.000000  1.000000  0.000000  0.000000  0.000000  1.000000
0.000000  1.000000  0.000000  1.000000  0.000000  0.000000  0.000000
1.000000  1.000000  0.000000  0.000000  0.000000  1.000000  1.000000
1.000000  1.000000  1.000000  1.000000  1.000000  0.000000  0.000000
0.000000  1.000000  1.000000  0.000000  0.000000  0.000000  1.000000
output vector is:
0.005010  0.002405  0.000141
-----------
for input vector:
1.000000  0.000000  0.000000  0.000000  1.000000  0.000000  1.000000
0.000000  1.000000  0.000000  0.000000  0.000000  1.000000  0.000000
0.000000  0.000000  0.000000  1.000000  0.000000  0.000000  0.000000
0.000000  1.000000  0.000000  0.000000  0.000000  1.000000  0.000000
1.000000  0.000000  1.000000  0.000000  0.000000  0.000000  1.000000
output vector is:
0.001230  0.997844  0.000663
-----------
for input vector:
1.000000  0.000000  0.000000  0.000000  1.000000  1.000000  0.000000
0.000000  0.000000  1.000000  1.000000  0.000000  0.000000  0.000000
1.000000  1.000000  1.000000  1.000000  1.000000  1.000000  1.000000
0.000000  0.000000  0.000000  1.000000  1.000000  0.000000  0.000000
0.000000  1.000000  1.000000  0.000000  0.000000  0.000000  1.000000
output vector is:
0.995348  0.000253  0.002677
-----------
for input vector:


1.000000  1.000000  1.000000  1.000000  1.000000  1.000000  0.000000
0.000000  0.000000  1.000000  1.000000  0.000000  0.000000  0.000000
1.000000  1.000000  1.000000  1.000000  1.000000  1.000000  1.000000
0.000000  0.000000  0.000000  1.000000  1.000000  0.000000  0.000000
0.000000  1.000000  1.000000  1.000000  1.000000  1.000000  1.000000
output vector is:
0.999966  0.000982  0.997594
-----------
for input vector:
0.000000  0.000000  1.000000  0.000000  0.000000  0.000000  0.000000
1.000000  0.000000  0.000000  0.000000  0.000000  1.000000  0.000000
0.000000  0.000000  0.000000  1.000000  0.000000  0.000000  0.000000
0.000000  1.000000  0.000000  0.000000  0.000000  0.000000  1.000000
0.000000  0.000000  0.000000  0.000000  1.000000  0.000000  0.000000
output vector is:
0.999637  0.998721  0.999330
-----------
```

Wzorce treningowe są bardzo dobrze wyuczone. W przypadku zastosowania mniejszej tolerancji możliwe byłoby ukończenie uczenia w mniejszej liczbie cykli. Co się stanie, jeśli przedstawimy sieci obcą postać? Utwórzmy nowy plik test.dat z dwoma wpisami dla liter M i J w następujący sposób:

```
1 0 0 0 1  1 1 0 1 1  1 0 1 0 1  1 0 0 0 1  1 0 0 0 1  1 0 0 0 1
0 0 1 0 0  0 0 1 0 0  0 0 1 0 0  0 0 1 0 0  0 0 1 0 0  0 0 1 0 0

1 0 0 0 1
0 1 1 1 1
```

Wyniki powinny pokazywać każdy obcy znak w najbliższej mu kategorii. Środkowa warstwa sieci działa jak wykrywacz funkcji. Ponieważ określiliśmy pięć neuronów, daliśmy sieci swobodę definiowania pięciu funkcji w zestawie szkoleniowym danych wejściowych, które mają być używane do kategoryzacji danych wejściowych. Wyniki w pliku output.dat są pokazane w następujący sposób.

```
for input vector:
1.000000  0.000000  0.000000  0.000000  1.000000  1.000000  1.000000
0.000000  1.000000  1.000000  1.000000  0.000000  1.000000  0.000000
1.000000  1.000000  0.000000  0.000000  0.000000  1.000000  1.000000
0.000000  0.000000  0.000000  1.000000  1.000000  0.000000  0.000000
0.000000  1.000000  1.000000  0.000000  0.000000  0.000000  1.000000
output vector is:
0.963513  0.000800  0.001231
-----------
for input vector:
0.000000  0.000000  1.000000  0.000000  0.000000  0.000000  0.000000
1.000000  0.000000  0.000000  0.000000  0.000000  1.000000  0.000000


 0.000000  0.000000  0.000000  1.000000  0.000000  0.000000  0.000000
 0.000000  1.000000  0.000000  0.000000  0.000000  0.000000  1.000000
 0.000000  0.000000  0.000000  1.000000  1.000000  1.000000  1.000000
 output vector is:
 0.999469  0.996339  0.999157
 -----------
```

W pierwszym wzorze M jest kategoryzowane jako H, podczas gdy w drugim wzorze J jest kategoryzowane jako I, zgodnie z oczekiwaniami. Przypadek pierwszego wzoru wydaje się rozsądny, ponieważ H i M mają wiele wspólnych pikseli.

Inne eksperymenty do wypróbowania

Istnieje wiele innych eksperymentów, które możesz wypróbować, aby lepiej zrozumieć, jak trenować i używać sieci neuronowej z propagacją wsteczną.

• Możesz użyć 8-bitowego kodu ASCII do przedstawienia każdego znaku i spróbować wytrenować sieć. Możesz także zakodować wszystkie znaki alfabetyczne i sprawdzić, czy można je wszystkie odróżnić.

• Możesz zniekształcić znak, aby sprawdzić, czy nadal otrzymujesz poprawny wynik.

• Możesz spróbować zmienić rozmiar warstwy środkowej i zobaczyć wpływ na czas treningu i zdolność uogólniania.

• Można zmienić ustawienie tolerancji, aby zobaczyć różnicę między przetrenowaną a niedostatecznie wytrenowaną siecią pod względem możliwości uogólniania. Oznacza to, że biorąc pod uwagę obcy wzorzec, czy sieć jest w stanie znaleźć najbliższe dopasowanie i użyć tej konkretnej kategorii, czy też dociera do zupełnie nowej kategorii?

Powrócimy do tego samego przykładu po wzbogaceniu symulatora o możliwość dodawania pędu i szumu.

Dodawanie terminu pędu

Prostą zmianą w prawie treningowym, która czasami skutkuje znacznie szybszym treningiem, jest dodanie terminu pędu. Zaimplementowane w symulatorze prawo szkolenia w zakresie wstecznej propagacji jest następujące:

Weight change = Beta * output_error * input

Teraz dodajemy wyraz do równania zmiany masy w następujący sposób:

Weight change = Beta * output_error * input +

Alpha*previous_weight_change

Drugi wyraz w tym równaniu to wyraz pędu. Zmiana wagi, przy braku błędu, byłaby stałą wielokrotnością poprzedniej zmiany wagi. Innymi słowy, zmiana masy postępuje w kierunku, w którym zmierzała. Termin rozpędu to próba utrzymania procesu zmiany wagi w ruchu, a tym samym nie utknięcie w lokalnych minimach.

Zmiany w kodzie

Wprowadzone pliki do wprowadzenia tej zmiany to plik layer.cpp do modyfikacji funkcji członkowskiej update_weights() klasy output_layer oraz główny plik backprop.cpp do wczytania wartości alfa i przekazania jej do funkcji członkowskiej. Do przechowywania poprzednich zmian wagi potrzebna jest dodatkowa pamięć, co ma wpływ na plik layer.h. Termin pędu mógłby zostać wdrożony na dwa sposoby:

1. Wykorzystanie zmiany wagi dla poprzedniego wzoru.

2. Wykorzystując zmianę masy zakumulowaną w poprzednim cyklu.

Chociaż obie te implementacje są prawidłowe, druga jest szczególnie użyteczna, ponieważ dodaje termin, który jest istotny dla wszystkich wzorców, a zatem przyczynia się do globalnej redukcji błędów. Drugi wybór realizujemy, akumulując wartość bieżących zmian wagi cyklu w wektorze zwanym cum_deltas. Przeszłe zmiany wagi cyklu są przechowywane w wektorze o nazwie past_deltas. Są one pokazane w następujący sposób w części pliku layer.h.

```
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

float * cum_deltas; // for momentum

float * past_deltas; // for momentum

friend network;

...
```

Zmiany w pliku layer.cpp

Plik implementacji klasy warstwy zmienia się w procedurze output_layer::update_weights() oraz w konstruktorze i destruktorze dla output_layer. Po pierwsze, oto konstruktor warstwy output_layer.

```
output_layer::output_layer(int ins, int outs)

{

int i, j, k;

num_inputs=ins;

num_outputs=outs;

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];

cum_deltas = new float[num_inputs*num_outputs];

past_deltas = new float[num_inputs*num_outputs];

if ((weights==0)||(output_errors==0)||(back_errors==0)

||(outputs==0)||(expected_values==0)

||(past_deltas==0)||(cum_deltas==0))

{

cout << "not enough memory\n";

cout << "choose a smaller architecture\n";

exit(1);

}

// zero cum_deltas and past_deltas matrix

for (i=0; i< num_inputs; i++)

{
```

```
k=i*num_outputs;
for (j=0; j< num_outputs; j++)
{
cum_deltas[k+j]=0;
past_deltas[k+j=0;
}
}
}
```

Destruktor po prostu usuwa nowe wektory:

```
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;
delete [num_outputs*num_inputs] past_deltas;
delete [num_outputs*num_inputs] cum_deltas;
}
```

Przyjrzyjmy się teraz zmianom w procedurze update_weights():

```
void output_layer::update_weights(const float beta,
                                  const float alpha)
{
int i, j, k;
float delta;
// learning law: weight_change =
//              beta*output_error*input + alpha*past_delta
for (i=0; i< num_inputs; i++)
        {
        k=i*num_outputs;
        for (j=0; j< num_outputs; j++)
                {
                delta=beta*output_errors[j]*(*(inputs+i))
                +alpha*past_deltas[k+j];
                weights[k+j] += delta;
                cum_deltas[k+j]+=delta; // current cycle
                }

        }

}
```

Zmiana w prawie treningu sprowadza się do obliczenia delty i dodania jej do skumulowanej sumy zmian wagi w cum_deltas. W pewnym momencie (na początku nowego cyklu) musisz ustawić wektor past_deltas na wektor cum_delta. Gdzie to się dzieje? Ponieważ warstwa nie ma pojęcia o cyklu, należy to zrobić na poziomie sieci. Na początku każdego cyklu znajduje się funkcja na poziomie sieci o nazwie update_momentum, która z kolei wywołuje funkcję poziomu warstwy o tej samej nazwie. Funkcja poziomu warstwy zamienia wektor past_deltas i wektor cum_deltas i ponownie inicjuje wektor cum_deltas do zera. Musimy wrócić do pliku layer.h, aby zobaczyć zmiany, które są potrzebne do zdefiniowania dwóch wspomnianych funkcji.

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

float * cum_deltas; // for momentum

float * past_deltas; // for momentum

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, const float);
void update_momentum();
void list_weights();
void write_weights(int, FILE *);
void read_weights(int, FILE *);
void list_errors();
void list_outputs();
};
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, const float);
void update_momentum();
```

...

Zarówno na poziomie sieci, jak i klasy output_layer prototyp funkcji dla funkcji składowych update_momentum jest podświetlony. Implementacja tych funkcji jest pokazana w następujący sposób z klasy layer.cpp.

```
void output_layer::update_momentum()
{
// This function is called when a
// new cycle begins; the past_deltas
// pointer is swapped with the
// cum_deltas pointer. Then the contents
// pointed to by the cum_deltas pointer
// is zeroed out.
int i, j, k;
float * temp;

// swap
temp = past_deltas;
past_deltas=cum_deltas;
cum_deltas=temp;

// zero cum_deltas matrix
// for new cycle
for (i=0; i< num_inputs; i++)
        {
        k=i*num_outputs;
        for (j=0; j< num_outputs; j++)
                cum_deltas[k+j]=0;
        }
}

void network::update_momentum()
{
int i;

for (i=1; i<number_of_layers; i++)
        ((output_layer *)layer_ptr[i])
                ->update_momentum();
```

Dodawanie hałasu podczas treningu

Innym podejściem do wyłamywania się z lokalnych minimów, a także zwiększania zdolności uogólniania, jest wprowadzenie pewnego szumu do sygnałów wejściowych podczas treningu. Liczba losowa jest dodawana do każdego składnika wejściowego wektora wejściowego, gdy jest on stosowany do sieci. Jest to skalowane przez ogólny współczynnik szumów, NF, który ma zakres od 0 do 1. Możesz dodać tyle szumu do symulacji, ile chcesz, lub wcale, wybierając NF = 0. Gdy jesteś blisko rozwiązania i osiągnąłeś satysfakcjonujące minimum, nie chcesz, aby szum w tym czasie przeszkadzał ze zbieżnością do minimum. Wprowadzamy współczynnik szumu, który zmniejsza się wraz z liczbą cykli, jak pokazano w poniższym fragmencie z pliku backprop.cpp.

```
// update NF
// gradually reduce noise to zero
if (total_cycles>0.7*max_cycles)
                    new_NF = 0;
else if (total_cycles>0.5*max_cycles)
                    new_NF = 0.25*NF;
else if (total_cycles>0.3*max_cycles)
                    new_NF = 0.50*NF;
else if (total_cycles>0.1*max_cycles)
                    new_NF = 0.75*NF;

backp.set_NF(new_NF);
```

Współczynnik hałasu jest redukowany w regularnych odstępach czasu. Nowy współczynnik szumów jest aktualizowany za pomocą funkcji klasy sieci o nazwie set_NF(float). W klasie sieci znajduje się zmienna składowa o nazwie NF, która przechowuje aktualną wartość współczynnika szumu. Szum jest dodawany do danych wejściowych w funkcji składowej input_layer calc_out().

Innym powodem używania szumu jest zapobieganie zapamiętywaniu przez sieć. W każdym cyklu efektywnie prezentujesz inny wzorzec wprowadzania, więc sieci może być trudno zapamiętać wzorce.

Jeszcze jedna zmiana — rozpoczęcie treningu z pliku zapisanej wagi Wkrótce przyjrzymy się kompletnym wykazom symulatora propagacji wstecznej. Jest jeszcze jedno ulepszenie do omówienia. Często w długich symulacjach przydatna jest możliwość rozpoczęcia od znanego punktu, który jest już zapisanym zestawem wag. Jest to prosta zmiana w programie backprop.cpp, która jest warta wysiłku. Dodatkową korzyścią jest to, że ta funkcja umożliwi uruchomienie symulacji z dużą wartością beta przez, powiedzmy, 500 cykli, zapisanie wag, a następnie rozpoczęcie nowej symulacji z mniejszą wartością beta przez kolejne 500 lub więcej cykli. Możesz robić gotowe przerwy w długich symulacjach, które napotkasz w Części 14. W tym miejscu przyjrzyjmy się kompletnym wykazom zaktualizowanych plików layer.hi layer.cpp w listingach 1 i 2:

Listing 1 plik layer.h zaktualizowany o szum i pęd

```
// layer.h V.Rao, H. Rao

// header file for the layer class hierarchy and

// the network class

// added noise and momentum

#define MAX_LAYERS 5

#define MAX_VECTORS 100

class network;

class Kohonen_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;
friend Kohonen_network; // update for Kohonen model
public:
virtual void calc_out()=0;
};
class input_layer: public layer
{
private:
float noise_factor;
float * orig_outputs;
public:
input_layer(int, int);
~input_layer();
virtual void calc_out();
void set_NF(float);
friend network;
};
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
float * cum_deltas; // for momentum
float * past_deltas; // for momentum
```

```
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, const float);
void update_momentum();
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, const float);
void update_momentum();
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);
void set_NF(float);
};
```

Listing 2 Plik Layer.cpp zaktualizowany o szum i pęd

```
// layer.cpp V.Rao, H.Rao
// added momentum and noise
// 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
//
{i
f (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)
{i
nt 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];
orig_outputs = new float[num_outputs];
if ((outputs==0)||(orig_outputs==0))
{
cout << "not enough memory\n";
cout << "choose a smaller architecture\n";
exit(1);
}
noise_factor=0;
}
input_layer::~input_layer()
{
delete [num_outputs] outputs;
delete [num_outputs] orig_outputs;
}
void input_layer::calc_out()
{
//add noise to inputs
// randomweight returns a random number
// between -1 and 1
int i;
```

```
for (i=0; i<num_outputs; i++)
outputs[i] =orig_outputs[i]*
(1+noise_factor*randomweight(0));
}
void input_layer::set_NF(float noise_fact)
{
noise_factor=noise_fact;
}
// ----------------------
// output layer
//----------------------
output_layer::output_layer(int ins, int outs)
{
int i, j, k;
num_inputs=ins;
num_outputs=outs;
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];
cum_deltas = new float[num_inputs*num_outputs];
past_deltas = new float[num_inputs*num_outputs];
if ((weights==0)||(output_errors==0)||(back_errors==0)
||(outputs==0)||(expected_values==0)
||(past_deltas==0)||(cum_deltas==0))
{
cout << "not enough memory\n";
cout << "choose a smaller architecture\n";
exit(1);
}
```

```
// zero cum_deltas and past_deltas matrix
for (i=0; i< num_inputs; i++)
{
k=i*num_outputs;
for (j=0; j< num_outputs; j++)
{
cum_deltas[k+j]=0;
past_deltas[k+j]=0;
}
}
}
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;
delete [num_outputs*num_inputs] past_deltas;
delete [num_outputs*num_inputs] cum_deltas;
}
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,
const float alpha)
{
int i, j, k;
```

```
float delta;
// learning law: weight_change =
// beta*output_error*input + alpha*past_delta
for (i=0; i< num_inputs; i++)
{
k=i*num_outputs;
for (j=0; j< num_outputs; j++)
{
delta=beta*output_errors[j]*(*(inputs+i))
+alpha*past_deltas[k+j];
weights[k+j] += delta;
cum_deltas[k+j]+=delta; // current cycle
}
}
}
void output_layer::update_momentum()
{
// This function is called when a
// new cycle begins; the past_deltas
// pointer is swapped with the
// cum_deltas pointer. Then the contents
// pointed to by the cum_deltas pointer
// is zeroed out.
int i, j, k;
float * temp;
// swap
temp = past_deltas;
past_deltas=cum_deltas;
cum_deltas=temp;
// zero cum_deltas matrix
// for new cycle
```

```
for (i=0; i< num_inputs; i++)
{
k=i*num_outputs;
for (j=0; j< num_outputs; j++)
cum_deltas[k+j]=0;
}
}
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 net work.\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+1] =
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-1]->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”;
exit(1);
}
}
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, const float alpha)
{
int i;
for (i=1; i<number_of_layers; i++)
((output_layer *)layer_ptr[i])
->update_weights(beta,alpha);
}
void network::update_momentum()
{
int i;
for (i=1; i<number_of_layers; i++)
((output_layer *)layer_ptr[i])
->update_momentum();
}
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++)
((input_layer*)layer_ptr[0])
->orig_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();
}
}
void network::set_NF(float noise_fact)
{
((input_layer*)layer_ptr[0])->set_NF(noise_fact);
}
```

Nowy i ostateczny plik backprop.cpp

Ostatnim plikiem do prezentacji jest plik backprop.cpp. Jest to pokazane na listingu 13.3.

Listing 3 Plik implementacyjny dla symulatora propagacji wstecznej, z szumem i pędem backprop.cpp

```
// backprop.cpp V. Rao, H. Rao
#include „warstwa.cpp”
#define TRAINING_FILE „training.dat”
#define WEIGHTS_FILE „weights.dat”
#define OUTPUT_FILE „output.dat”
#define TEST_FILE „test.dat”
nieważne główne ()
{
zmiennoprzecinkowa tolerancja_błędu=0,1;
pływak całkowity_błąd=0.0;
```

```
float avg_error_per_cycle=0.0;
pływak błąd_ostatni_cykl=0.0;
float avgerr_per_pattern=0.0; // dla najnowszego cyklu
pływak błąd_ostatni_wzorzec=0.0;
zmiennoprzecinkowy parametr_nauczania=0,02;
pływak alfa; // parametr pędu
pływak NF; // współczynnik szumów
pływak nowy_NF;
temp. bez znaku, uruchamianie, start_weights;
długie int vectors_in_buffer;
długie int max_cycles;
długi int pattern_per_cycle=0;
long int total_cycles, total_patterns;
wew;
// utwórz obiekt sieciowy
kopia zapasowa sieci;
PLIK * plik_szkoleniowy_ptr, * plik_wagi_ptr, * plik_wyjściowy_ptr;
PLIK * test_file_ptr, * data_file_ptr;
// otwórz plik wyjściowy do zapisu
if ((plik_wyjściowy_ptr=fopen(PLIK_WYJŚCIOWY,”w”))==NULL)
{
cout << „problem z otwarciem pliku wyjściowego\n”;
wyjście (1);
}
// wejdź w tryb treningu : 1=trening włączony 0=trening wyłączony
cout << „————————————————————————————-\n”;
cout << “ Sieci neuronowe C++ i logika rozmyta \n”;
cout << „Symulator propagacji wstecznej \n”;
cout << “ wersja 2 \n”;
cout << „————————————————————————————-\n”;
cout << „Proszę wpisać 1 dla włączenia TRENINGU lub 0 dla wyłączenia: \n\n”;
```

```
cout << „Użyj treningu, aby zmienić wagę zgodnie ze swoim\n”;
cout << “oczekiwane wyjścia. Twój plik training.dat powinien zawierać\n”;
cout << “zestaw danych wejściowych i oczekiwanych wyników. Liczba\n”;
cout << „wejścia określają rozmiar pierwszej (wejściowej) warstwy\n”;
cout << „podczas gdy liczba wyjść określa rozmiar\n”;
cout << „ostatnia (wyjściowa) warstwa :\n\n”;
cin >> temp;
backp.set_training(temp);
if (backp.get_training_value() == 1)
{
cout << „—> Tryb treningu jest *WŁĄCZONY*. wagi zostaną zapisane\n”;
cout << „w pliku weights.dat na końcu \n”;
cout << „bieżący zestaw danych wejściowych (treningowych)\n”;
}
w przeciwnym razie
{
cout << „—> Tryb treningu jest *OFF*. ciężary zostaną załadowane\n”;
cout << „z pliku weights.dat i bieżącego\n”;
cout << “(test) zostanie użyty zestaw danych. Do testu\n”;
cout << „zestaw danych, plik test.dat powinien zawierać\n”;
cout << „tylko wejścia, bez oczekiwanych wyjść.\n”;
}
if (backp.get_training_value()==1)
{
// ————————————————————————-
// Wczytaj wartości dla error_tolerance,
// i parametr_uczenia
// ————————————————————————-
cout << „Proszę wprowadzić tolerancję błędu\n”;
cout << “ —- od 0,001 do 100,0, spróbuj 0,1, aby rozpocząć - \n”;
Cout << „\n”;
```

```
cout << „i parametr_uczenia, beta\n”;
cout << “ —- od 0,01 do 1,0, spróbuj 0,5, aby rozpocząć - \n\n”;
cout << „oddziel wpisy spacją\n”;
cout << “ przykład: 0.1 0.5 ustawia wspomniane domyślne :\n\n”;
cin >> tolerancja_błędu >> parametr_nauczenia;
// ——————————————————————-
// Wczytaj wartości pędu
// parametr, alfa (0-1.0)
// i współczynnik szumów, NF (0-1,0)
// ——————————————————————-
cout << „Wprowadź teraz wartości pędu \n”;
cout << „parametr, alfa (0-1.0)\n”;
cout << “ i współczynnik szumów, NF (0-1.0)\n”;
cout << „Możesz wpisać zero dla każdego z nich\n”;
cout << „parametry, aby wyłączyć pęd lub\n”;
cout << „funkcje hałasu.\n”;
cout << „Jeśli używana jest funkcja szumu, losowo\n”;
cout << „składnik szumu jest dodawany do wejść\n”;
cout << „To jest zmniejszone do 0 ponad maksimum\n”;
cout << „określona liczba cykli.\n”;
cout << „wpisz alfa, a następnie NF, np. 0.3 0.5\n”;
cin >> alfa >> NF;
//———————————————————————-
// otwórz plik szkoleniowy do czytania
//———————————————————————-
if ((training_file_ptr=fopen(TRAINING_FILE”,r”))==NULL)
{
cout << „problem z otwieraniem pliku szkoleniowego\n”;
wyjście (1);
}
data_file_ptr=trening_file_ptr; // Trenuj
```

```
// Odczytaj maksymalną liczbę cykli
// każde przejście przez plik danych wejściowych to cykl
cout << „Proszę wprowadzić maksymalne cykle symulacji\n”;
cout << „Cykl to jedno przejście przez zbiór danych.\n”;
cout << „Wypróbuj wartość 10 na początek\n”;
cin >> max_cykle;
cout << „Czy chcesz odczytać wagi z weights.dat do
rozpocząć?\n”;
cout << „Wpisz 1, aby odczytać z pliku, 0, aby rozpocząć losowo
wagi\n”;
cin >> start_weights;
}
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
propagation
// 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
new_NF=NF;
// 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)&&(start_weights!=1))
{
// randomize weights for all layers; there is no
// weight matrix associated with the input layer
// weight file will be written after processing
backp.randomize_weights();
// set up the noise factor value
backp.set_NF(new_NF);
}
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);
```

```
fclose(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;
backp.update_momentum(); // added to reset
// momentum matrices
// each cycle
// 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*error_last_pattern;
```

```
avgerr_per_pattern=
((float)sqrt((double)error_last_cycle/patterns_per_cycle));
// if it's not the last cycle, update weights
if ((avgerr_per_pattern
> error_tolerance)
&& (total_cycles+1 < max_cycles))
backp.update_weights(learning_
parameter, alpha);
// backp.list_weights(); // can
// see change in weights by
// using list_weights before and
// after back_propagation
}
}
error_last_pattern = 0;
}
total_error += error_last_cycle;
total_cycles++;
// update NF
// gradually reduce noise to zero
if (total_cycles>0.7*max_cycles)
new_NF = 0;
else if (total_cycles>0.5*max_cycles)
new_NF = 0.25*NF;
else if (total_cycles>0.3*max_cycles)
new_NF = 0.50*NF;
else if (total_cycles>0.1*max_cycles)
new_NF = 0.75*NF;
backp.set_NF(new_NF);
// most character displays are 25 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\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
if (backp.get_training_value()==1)
{
if ((weights_file_ptr=fopen(WEIGHTS_FILE,"w"))
==NULL)
{
cout << "problem opening weights file\n";
exit(1);
}
}
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);
fclose(weights_file_ptr);
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(output_file_ptr);
}
```

Wypróbowanie funkcji Noise i Momentum

Możesz przetestować symulator wersji 2, który właśnie skompilowałeś z przykładem, który widziałeś na początku rozdziału. Przekonasz się, że znalezienie optymalnych wartości alfa, współczynnika szumu i beta wymaga wielu prób i błędów. Dotyczy to również rozmiaru warstwy środkowej i liczby warstw środkowych. W przypadku niektórych problemów dodanie pędu znacznie przyspiesza zbieżność. W przypadku innych problemów możesz nie znaleźć żadnej zauważalnej różnicy. Przykładowy przebieg problemu rozpoznawania pięciu znaków, omówiony na początku tego rozdziału, dał następujące wyniki: beta = 0,1, tolerancja = 0,001, alfa = 0,25, NF = 0,1 i rozmiary warstw utrzymywane na poziomie 35 5 3.

```
-----------------------------
        done:   results in file output.dat
                training: last vector only
                not training: full cycle

                weights saved in file weights.dat

-->average error per cycle = 0.02993<--
-->error last cycle = 0.00498<--
->error last cycle per pattern= 0.000996 <--
------->total cycles = 242 <--
------->total patterns = 1210 <--
-----------------------------
```

Sieć była w stanie osiągnąć lepsze rozwiązanie (pod względem pomiaru błędów) w jednej czwartej liczby cykli. Możesz spróbować zróżnicowania alfa i NF, aby zobaczyć wpływ na ogólny czas symulacji. Możesz teraz zacząć od tych samych początkowych wag początkowych, podając wartość 1 dla pytania o początkowe wagi. W przypadku dużych wartości alfa i beta sieć zwykle nie będzie zbieżna, a wagi staną się niedopuszczalnie duże (otrzymasz komunikat o tym fakcie).

Odmiany algorytmu propagacji wstecznej

Propagacja wsteczna to wszechstronny algorytm sieci neuronowej, który bardzo często prowadzi do sukcesu. Jego piętą achillesową jest powolność, z jaką zbiega się w przypadku pewnych problemów. W literaturze istnieje wiele odmian algorytmu, które mają na celu poprawę szybkości zbieżności i odporności. Zaproponowano warianty w następujących fragmentach algorytmu:

• Parametry adaptacyjne. Możesz ustawić reguły, które modyfikują parametr alfa (parametr momentum) i beta (parametr uczenia się) w miarę postępu symulacji. Na przykład możesz zmniejszyć beta, gdy zmiana wagi nie zmniejsza błędu. Możesz rozważyć cofnięcie konkretnej zmiany wagi, ustawienie alfa na zero i powtórzenie zmiany wagi z nową wartością beta.

• Używaj innych minimalnych procedur wyszukiwania oprócz najbardziej stromego zjazdu. Na przykład możesz użyć metody Newtona do znalezienia minimum, chociaż byłby to dość powolny proces. Inne przykłady obejmują zastosowanie metod gradientu sprzężonego lub optymalizacji Levenberga-Marquardta, które skutkowałyby bardzo szybkim treningiem.

• Użyj różnych funkcji kosztów. Zamiast obliczać błąd (zgodnie z oczekiwaniami - rzeczywista wydajność), możesz określić inną funkcję kosztu, którą chcesz zminimalizować.

• Zmodyfikuj architekturę. Możesz użyć częściowo połączonych warstw zamiast w pełni połączonych warstw. Możesz także użyć sieci rekurencyjnej, to znaczy takiej, w której niektóre wyjścia są zwrotne jako wejścia.

Aplikacje

Propagacja wsteczna pozostaje królem architektur sieci neuronowych ze względu na łatwość użycia i szerokie zastosowanie. Kilka godnych uwagi zastosowań w literaturze zostanie przytoczonych jako przykłady.

• NETTalk. W 1987 roku Sejnowski i Rosenberg opracowali sieć połączoną z syntezatorem mowy, który był w stanie wypowiadać angielskie słowa, przyuczony do tworzenia fonemów z tekstu angielskiego. Architektura składała się z siedmioznakowego okna warstwy wejściowej. Znaki były częścią

przewijanego tekstu angielskiego. Sieć została przeszkolona, aby wymawiać literę pośrodku okna. Warstwa środkowa miała 80 neuronów, natomiast warstwa wyjściowa składała się z 26 neuronów. Dzięki 1024 schematom treningowym i 10 cyklom sieć zaczęła wytwarzać zrozumiałą mowę, podobnie do procesu uczenia się mowy przez dziecko. Po 50 cyklach sieć była około 95% dokładna. Można celowo uszkodzić sieć poprzez usunięcie neuronów, ale nie spowodowało to spadku wydajności; zamiast tego wydajność uległa wdzięcznemu pogorszeniu. Nastąpił szybki powrót do zdrowia po ponownym treningu przy użyciu mniejszej liczby neuronów. Pokazuje to odporność sieci neuronowych na uszkodzenia.

• Rozpoznawanie celów sonaru. Sieci neuronowe wykorzystujące propagację wsteczną zostały wykorzystane do identyfikacji różnych typów celów przy użyciu sygnatury częstotliwości (z szybką transformacją Fouriera) odbitego sygnału.

• Nawigacja samochodowa. Pomerleau opracował sieć neuronową, która jest w stanie nawigować samochodem w oparciu o obrazy uzyskane z kamery zamontowanej na dachu samochodu oraz dalmierz kodujący odległości w skali szarości. Obraz 30×32 pikseli i obraz z dalmierza 8×32 zostały wprowadzone do ukrytej warstwy o rozmiarze 29, zasilającej warstwę wyjściową 45 neuronów. Neurony wyjściowe zostały ułożone w linii prostej, z każdą stroną reprezentującą skręt w określonym kierunku (w prawo lub w lewo), podczas gdy neurony środkowe reprezentowały „jazdę na wprost". Po przeszkoleniu w sieci 1200 obrazów dróg, kierowca sieci neuronowej był w stanie pokonać część kampusu Carnegie-Mellon z prędkością około 3 mil na godzinę, ograniczoną jedynie prędkością obliczeń wykonywanych w czasie rzeczywistym na przeszkolona sieć w komputerze Sun-3 w samochodzie.

• Kompresja obrazu. G.W. Cottrell, P. Munro i D. Zipser zastosowali propagację wsteczną do kompresji obrazów, uzyskując współczynnik kompresji 8:1. Wykorzystali standardową propagację wsteczną z 64 neuronami wejściowymi (8×8 pikseli), 16 neuronami ukrytymi i 64 neuronami wyjściowymi równymi wartościom wejściowym. Nazywa się to samonadzorowaną propagacją wsteczną i reprezentuje sieć autoasocjacyjną. Skompresowany sygnał jest pobierany z warstwy ukrytej. Warstwa wejściowa do warstwy ukrytej składała się z kompresora, podczas gdy warstwa ukryta do wyjścia tworzy dekompresor.

• Rozpoznawanie obrazu. Le Cun poinformował o sieci wstecznej propagacji z trzema ukrytymi warstwami, które potrafiły rozpoznawać odręcznie napisane pocztowe kody pocztowe. Użył tablicy pikseli 16×16 do reprezentowania każdej odręcznie napisanej cyfry i musiał zakodować 10 wyjść, z których każde reprezentowało cyfrę od 0 do 9. Interesującym aspektem tej pracy jest to, że ukryte warstwy nie były w pełni połączone. Sieć została utworzona z bloków neuronów w pierwszych dwóch warstwach ukrytych ustawionych jako detektory cech dla różnych części poprzedniej warstwy. Wszystkie neurony w bloku zostały ustawione tak, aby miały takie same wagi jak te z poprzedniej warstwy. Nazywa się to podziałem wagi. Każdy blok próbkowałby inną część obrazu z poprzedniej warstwy. Pierwsza warstwa ukryta miała 12 bloków po 8×8 neuronów, natomiast druga warstwa ukryta miała 12 bloków po 4×4 neurony. Trzecia warstwa ukryta była w pełni połączona i składała się z 30 neuronów. Było 1256 neuronów. Sieć została przeszkolona na 7300 przykładach i przetestowana na 2000 przypadkach ze wskaźnikami błędów wynoszącymi 1% na zestawie uczącym i 5% na zestawie testowym.

Podsumowanie

W tym rozdziale dokładniej zbadałeś algorytm wstecznej propagacji błędów, kontynuując dyskusję z Części 7.

• Do prawa szkoleniowego dodano termin określający dynamikę, który w niektórych przypadkach prowadził do znacznie szybszej konwergencji.

• Do danych wejściowych dodano termin szumu, aby umożliwić odbycie szkolenia z zastosowanym szumem losowym. Hałas ten zmniejszał się wraz z liczbą cykli, tak aby nauka ostatniego etapu mogła odbywać się w bezgłośnym środowisku.

• Ostateczna wersja symulatora wstecznej propagacji błędów została skonstruowana i wykorzystana na przykładzie z Części 12. Dalsze zastosowanie symulatora zostanie omówione w Części 14.

• Przedstawiono kilka zastosowań z algorytmem wstecznej propagacji błędów, pokazując szerokie zastosowanie tego algorytmu.