1章

C#で実装するニューラルネットワーク

何事にも訓練が大切だ。
桃も昔は苦いアーモンドだった。
カリフラワーも、大学教育を受けたキャベツに過ぎない。
――マーク・トウェイン――

まず、人工生命の脳に積む予定のニューラルネットワークを、Unity の外で構築する。ここでは Tensorflow などのライブラリを用いずに、Unity の標準言語である C#でゼロから構築するため、ニューラルネットワークの原理を理解する必要がある。[1]

1.1 環境構築

Unity を用いずに C#単体でプログラムを動かすには、コンパイラが必要だ。C/C++言語をコンパイルするには、GCC などのコンパイラや Visual Studio をインストールしなければならないが、C#に関しては、Windows OS に最初からコンパイラが用意されていることがほとんどである。

1.1.1 .NET Framework

Microsoft .NET Framework は、マイクロソフトが開発したアプリケーション開発・実行環境である。Everything[2]などを使って検索するか、「C:」->「Windows」->「Microsoft.NET」->「Framework64」->「v4.0.30319」などと辿っていけば、「csc.exe」という exe ファイルが見つかる。これがコンパイラだ。「csc.exe」は 32bit/64bit、v3.0/v3.5/v4.0.30319…など複数のフォルダから見つかるだろうが、64bit が使えるなら「Framework64」を使えばよいし、バージョンは最も新しいものを使うことを勧める。

---

[1] Python の読み書きができる読者には、参考文献として、斎藤康毅「ゼロから作る Deep Learning ―Python で学ぶディープラーニングの理論と実装」オライリージャパン(2016)を強く推奨する。Python の読み書きができない読者には、Python を覚えることを強く推奨する。

[2] https://forest.watch.impress.co.jp/library/software/everything/download_11365.html

1.1.2 bat ファイルの作成

このコンパイラをコマンドラインから実行し、引数に C#のソースコードを渡すと、コンパイルが実行される。この先、このコンパイラを多用するので、実行コマンドを bat ファイル化しておこう。「csc.exe」までのパスは適宜変更すること。

リスト 1-1 compile.bat

```
@echo off
C:¥Windows¥Microsoft.NET¥Framework64¥v4.0.30319¥csc.exe %1
%~n1.exe
```

1.1.3 Hello world!

適当な作業フォルダを作成し、compile.bat を保存する。そして、以下の hello.cs を作成しよう。

リスト 1-2 hello.cs

```
using System;

class Program
{
	static void Main()
	{
		Console.WriteLine("Hello world!");
	}
}
```

そして以下のようにコマンドを実行する。

リスト 1-3 cmd.exe

```
C:¥C#¥unity-alife¥chap1>compile.bat hello.cs
Microsoft (R) Visual C# Compiler version 4.8.3752.0
for C# 5
Copyright (C) Microsoft Corporation. All rights reserved.

This compiler is provided as part of the Microsoft (R) .NET Framework, but only supports language versions up to C# 5, which is no longer the latest version. For compilers that support newer versions of the C# programming language, see
http://go.microsoft.com/fwlink/?LinkID=533240

Hello world!
```

作業フォルダに hello.exe が作成され、コマンドプロンプトの最終行に「Hello world!」が

表示されたことが確認できる。これが最初の C#プログラムだ。[3]

1.2 ニューラルネットワークの原理

我々がこれから実装するニューラルネットワークは、**図 1**[4]の構造を持つアルゴリズムだ。複数の層が連なって形成され、層から層へと値の受け渡しを行うことで計算を行う。その結果、入力したデータを変換して対応する値を出力（**回帰**, regression）したり、入力したデータをいくつかの種類に区別（**分類**, classification）したりすることができる。層は**入力層・隠れ層・出力層**からなり、**図 1** ではそれぞれ 1 つずつしかないが、隠れ層の数を増やすことで、いわゆる深層学習（Deep Learning）となる。各層は複数の**ニューロン**（**図 1** の○）から形成されるが、このニューロンの数も適宜変更することができる。

ニューラルネットワークによって計算されるべきデータは、入力層に入力される（そのまますぎる表現だが）。次に、入力層のニューロンが持つ値は隠れ層のニューロンへと送られるが、この時、それぞれの値には対応した**重み** w が掛け算される。1 つの隠れ層のニューロンは、入力層のすべてのニューロンとつながっている[5]ため、入力層→隠れ層の伝播には、$i_0 \to h_0$, $i_0 \to h_1$, …, $i_1 \to h_0$, $i_1 \to i_1$, …, $i_{Ni-1} \to h_{Nh-1}$ の計 Ni×Nh 個の重みが関与していることがわかる。同様に、隠れ層→出力層の伝播には Nh×Nu 個の重みが関与する。1 つのニューロンが持つことになる値は、それに投射するニューロンと対応する重みの積の総和であり、例えば

$$h_k = \sum_l i_l w_{i_l h_k} = i_0 w_{i_0 h_k} + i_1 w_{i_1 h_k} + \cdots + i_{Ni-1} w_{i_{Ni-1} h_k}$$

が成り立つ。したがって、絶対値の大きい重みに対応するニューロンの投射ほど、次の層に対する入力として重視されていると考えることができる。

このように伝播を繰り返すと、データは最終的に出力層に到達する。この時、出力層が持っている値がこのニューラルネットワークの計算結果であり、一般に**予測値**（prediction）と呼ばれる。この予測値と実際に出力すべきだった値（真値、教師データなどと呼ばれる）

---

[3] この手法でコンパイルしたソースコードにエラーがあった場合は、警告とともに**以前の**コンパイル結果が表示される。以前にコンパイルが行われていない場合には、「'hogehoge.exe'は、内部コマンドまたは外部コマンド、操作可能なプログラムまたはバッチ ファイルとして認識されていません。」と表示される（実行ファイルが存在しないため）。また、今後は compile.bat の実行に伴う Compiler の version 情報の出力を省略する。

[4] 入力層・隠れ層のニューロンに与えられた名前は i (input), h (hidden）に対応しているが、出力層のニューロンには output から u を割り当てる。これは、o と 0 の混同を防ぐための対応である。

[5] これを、全結合層と呼ぶ。

の差が**誤差**（error）となるが、この誤差を最小化するように各層の重みを更新することで、ニューラルネットワークを現実に適合させていく。この過程を学習、または**訓練**（training）と呼ぶ。

1.3 何を実装すべきか？

前章の内容がニューラルネットワークの概念であるが、これをゼロから実装するには、より詳細な構造を抑えなければならない。

1.3.1 入力層・隠れ層・出力層

これらの層の関係は、 **1.2** で述べた通りである。層の間を伝播する際には、重みの掛け算が行われるため、各層は独自の重みの配列を保持する必要がある。また、隠れ層の数は任意に増やすことができるため、これをクラスとして実装し、好きな数だけ複製できるようにしよう。

1.3.2 活性化関数

全か無かの法則を知っているだろうか？これはニューラルネットワークの発想の基になった、神経細胞が持つ法則である。生物の感覚・運動機能は神経細胞の活動の連鎖を通して実現される。このとき、1つの神経細胞は活動した複数の神経細胞から入力を受け、細胞に電気的な変化（膜電位の上昇）が生じる。そして、この後の神経細胞の反応は次のうちの2つに1つだ――変化の度合いが**閾値**を超えて**発火**するか、超えずに沈黙するか、である。このように、入力が閾値を超えなければ、どれだけ惜しい所まで行っても全てが無駄になってしまう反応原理のことを、全か無かの法則と呼んでいる。すなわち、神経細胞は入力をアナログ（連続値）で受け取るが、それに対して発火（1）か沈黙（0）のデジタルで反応するのである。

このように、神経細胞（ニューロン）を使って情報をアナログとデジタルの両方式で処理することにより、ネットワークが持つ表現の幅は大きく広がることになる。[6]そのため、人工的なニューラルネットワークでも、これに類似した機能は**活性化関数**として実装されている。

活性化関数は、層と層の間に挟まる形で配置される。すなわち、前層で重みと掛け合わされた総和は、実際には活性化関数を通して処理されてから次層に送られるのだ。活性化関数には、ステップ関数、シグモイド関数、ReLu 関数、Leaky ReLu 関数が存在するが、以下、

---

[6] 原理の詳細は、齋藤 2016 を参照。

これらを 1 つずつ見ていこう。

ステップ関数（図 2 -左上）

$$y(x) = \begin{cases} 1\ (x \geq threshold) \\ 0\ (x < threshold) \end{cases}$$

は、もっとも実際の神経細胞に近い挙動を示す。図では閾値 0 を超えると 1 を返し、超えなければ 0 を返す。

シグモイド関数（図 2 -右上）

$$y(x) = \frac{1}{1 + e^{-x}}$$

は、従来もっともよく使われてきた関数だ。ステップ関数を滑らかにした関数であり、デジタルではなくアナログの連続値を返す。ステップ関数に比べれば、生物学的なもっともらしさはないが、この滑らかさが計算に有利に働くことが多い。また、別の視点からは、負の無限大から正の無限大までの範囲を取る入力を、 0 から 1 の範囲に滑らかに圧縮する関数であると見ることもできる。

ReLu 関数（Rectified Linear Unit, **図 2 -左下**）

$$y(x) = \begin{cases} x\ (y \geq 0) \\ 0\ (y < 0) \end{cases}$$

は、負の値を取る入力をすべて 0 にし、正の入力はそのまま通す。生物学的なもっともらしさはさらに消失したが、隠れ層における計算では、シグモイド関数よりも高いパフォーマンスを誇る。

Leaky ReLU 関数（図 2 -右下）

$$y(x) = \begin{cases} x\ (y \geq 0) \\ \alpha x\ (y < 0) \end{cases}$$

は、ReLu 関数の改良版である。ReLu 関数では負の入力を持つニューロンを一括に無視することになるため、**Dead Neurons 問題**が生ずる。すなわち、使えるニューロンが減るため計算効率が落ちるのだ。そのため Leaky ReLu 関数では、負の値を取る入力は α (=0.01 など)倍に縮小されるが、完全に無視はしないことでニューロンの個性を保持する。

以下、本章で扱うような単純なニューラルネットワークの活性化関数には ReLu 関数を用いるが、Unity に組み込む複雑なネットワークにおいては Leaky ReLU 関数を用いることにする。

1.3.3 Softmax 関数

層間を流れる入力の伝播は、活性化関数を通過した後に次の層で評価される。これは出力層においても同様であり、出力層から出てきた信号も活性化関数を通してから評価されることが多い。**1.3.2** で紹介した活性化関数はすべて x の関数、すなわち 1 つの入力にのみ依

存して値が決まるため、回帰問題の出力を扱うには都合が良い。しかし、分類問題では、分類したいグループの個数と同数のニューロンを持つ出力層の信号から、1 つの解答（予測ラベル）を出力する。したがって、このときの出力の計算には、複数の入力値に依存する関数を用いる。

Softmax 関数は、以下の式で表される関数である。

$$y_k(X) = \frac{e^{x_k}}{\sum_j e^{x_j}}$$

ただし、

$$X = \{x_0, x_1, \dots, x_k, \dots, x_{N-1}\}$$

であり、すなわち y は N 個の入力に依存する関数である。そして入力 x と同様に y も N 個の値を返し、y_k は入力 x_k に対応する値である。そして最後に、この N 個の値のうち、最大の値を取る k に対応したラベルを予測ラベルとして出力する。

Softmax 関数の働きは、N 個の入力の総和を 1 にするよう、滑らかに拡大・縮小することである。実はこのような処理をせずとも、最大の値を取る x_k に対応するラベルを返せば結果は変わらないのだが、Softmax 関数は **1.3.4** で扱う「重みの最適化」を行う際に重要であり、とくに Cross-Entropy 誤差と合わせて使用すると、計算上の大きなメリットがある。

1.3.4 重みの最適化

ニューラルネットワークは、各層が持つ重み w を適宜変化させることによって回帰・分類といった問題に自らを適合させていく。このとき、重みの更新には**勾配降下法**（Gradient descent）という手法が用いられることが多い。

回帰でも分類でも、ニューラルネットワークの予測値と実際の値の間には**誤差**が生じる。理想的には、各層の重みを更新することで、この誤差が小さくなるようにしたい。これを実現するために、とある重み w_a を少し大きくしたときの誤差 L の変化、すなわち L の w_a による偏微分[7]

$$\frac{\partial L}{\partial w_a}$$

を考える。この偏微分の値が負になる方向に w_a を更新すると L は小さくなるので、w_a は

$$w_a \leftarrow w_a - \eta \frac{\partial L}{\partial w_a}$$

と更新して行けば良い。ここで、η は学習率と呼ばれるパラメータであり、0.01 など 1 よりも小さい値が設定されることが多い。このように、パラメータごとに誤差の偏微分（勾配）

---

[7] 誤差 L は w_a 以外にも他の様々な要因（他の重みや活性化関数）によって定まる多変数関数であるため、全微分ではなく偏微分となる。

を求め、その勾配を下（負）に向かうようにある程度（η によって調整される）移動させていく手法を勾配降下法と呼ぶ。

誤差の評価法にも複数種類があるが、もっとも有名なのは**２乗和誤差**（mean squared error）

$$L = \frac{1}{2}\sum_k (y_k - t_k)^2$$

である。ここで y_k は k 番目の予測値であり、t_k は k 番目の真の値である。２乗和誤差では、これらの予測値と真値の差を２乗して正となるようにした後、その和を求める。出力が１つの回帰問題では、単に

$$L = \frac{1}{2}(y - t)^2$$

として用いられる。1/2 を掛けることの意味は、L の y による偏微分を考えるとわかる。

$$L = \frac{1}{2}(y^2 - 2ty + t^2)$$

より、

$$\frac{\partial L}{\partial y} = \frac{1}{2}(2y - 2t) = y - t$$

として値が簡単になる。

分類問題では **Cross-Entropy 誤差**

$$L = -\sum_k t_k \log y_k$$

を用いることが多い。この関数は Softmax 関数とともに使われることが多く、定義式中の y_k も Softmax 関数の出力に対応している。Softmax 関数は複数の入力の総和が 1 となるように変形して出力する関数であることから、出力 y_k は、これに対応するラベルが正解となる**予測確率**と考えることができる。同様に、t_k は対応するラベルが正解となる真の確率であり、すなわち、ラベル k が正解の場合は 1、不正解の場合は 0 となる。

ここで、Softmax 関数と Cross-Entropy 誤差を組み合わせて用いた場合、Softmax 関数に与えられる入力 x_k で誤差 L を偏微分すると、

$$\frac{\partial L}{\partial x_k} = y_k - t_k$$

と単純な値になることが知られている。[8]これこそが、Softmax 関数と Cross-Entropy 誤差を組み合わせて用いることの利点である。

1.3.5　誤差逆伝播法

---

[8] 証明略

さて、勾配降下法では重みごとに誤差 L の偏微分を計算し、それに従って重みを更新して行くのであった。しかし重みとネットワークの関係は複雑であり、とある重みと誤差の変化の関係を調べるためには、実際に重みを動かして誤差の挙動を見ていくしかない。しかし、重みの個数と同じ回数だけ、このシミュレーションを行うのでは非常に時間がかかる。

一般に、入力層に近い層に属する重みほど、誤差に対し大きな影響を与える。なぜならば、その重みを変化させることで、それ以下の層の信号すべてが変化するからだ。すなわち、出力層に近い層に属する重みほどその影響を計算しやすい。

ここで、(偏) 微分の一般的な法則

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial x}$$

を考えてみよう。つまり、y の x による偏微分は、u の x による偏微分と y の u による偏微分の積に分解することができる。任意の層における入力と重みの計算式

$$h_k = \sum_l i_l w_{i_l h_k} = i_0 w_{i_0 h_k} + i_1 w_{i_1 h_k} + \cdots + i_{Ni-1} w_{i_{Ni-1} h_k}$$

によると、この層の出力（次の層または活性化関数の入力）h_k は、例えば重み w_{i0hk} によって制御されていることがわかるが、ここで誤差 L の出力 h_k による偏微分が既に得られている場合、誤差 L の重み w_{i0hk} による偏微分は

$$\frac{\partial L}{\partial w_{i_0 h_k}} = \frac{\partial L}{\partial h_k}\frac{\partial h_k}{\partial w_{i_0 h_k}} = i_0 \frac{\partial L}{\partial h_k}$$

により求められる。このように、任意の位置にあるパラメータによる偏微分の値は、それが制御している下位層のパラメータによる偏微分を用いて計算することができる。

誤差逆伝播法（back propagation）は、これを実装するためのアルゴリズムである。各層の重みの更新に際し、出力層から順に誤差の偏微分を計算し、その結果を上層に伝えていく。そして、上位の層では伝わってきた偏微分値を用いてパラメータを更新し、同時にさらに上位の層へ伝えるための偏微分の値を計算する。このとき、誤差の情報が通常のニューラルネットワークの流れを逆流するように伝播していくことが、このアルゴリズムの名称の由来である。[9]

以上の結果をもとに、本章の以下では画像を分類するニューラルネットワークを C#で構築する。その全体像は、**図 3** のようになるだろう。

1.4 MNIST

画像の分類を行うためのデータセットとして、MNIST を用いる。これは 0 から 9 までの

[9] 以下、「偏微分の値」のことは単に「勾配」と呼ぶ。

手書き文字の画像が、正解のラベルとセットになったデータセットである。データセットは、http://yann.lecun.com/exdb/mnist/からダウンロードできるので、「train-images-idx-ubyte.gz」、「train-labels-idx1-ubyte.gz」、「t10k-images-idx3-ubyte.gz」、「t10k-labels-idx1-ubyte.gz」のすべてのファイルを入手して、compile.bat のある作業フォルダに展開[10]しておこう。

1.5 画像認識ニューラルネットワークの実装

ソースコードの全体像は**リスト 1-4** のようになる。

リスト 1-4 network_main.cs

```
using System;
using System.IO;

class Program
{
	const int EPOCH = 100;
	const int SIZE = 28*28;
	const int NUM_TRAIN = 600;
	const int NUM_TEST = 100;
	static float[] img = new float[SIZE];
	static Network network = new Network(SIZE);

	static void Main()
	{
		for (int e = 0; e < EPOCH; e++) {
			core(false, true, true, true, e, ref network);
		}
		core(true, false, true, false, -1, ref network);
	}


	static void core(bool test, bool mut_weights, bool score, bool loss, int epoch, ref Network
network)
	{
		String image_file_name;
		String label_file_name;
		int num_data;
		int num_collect = 0;
		float loss_sum = 0f;

		if (test) {
			image_file_name = "t10k-images.idx3-ubyte";
			label_file_name = "t10k-labels.idx1-ubyte";
			num_data = NUM_TEST;
		} else {
```

[10] 「解凍」は死語らしい。つらい。

```
            image_file_name = "train-images.idx3-ubyte";
            label_file_name = "train-labels.idx1-ubyte";
            num_data = NUM_TRAIN;
        }

        using (FileStream image_stream = File.OpenRead(image_file_name))
        using (var image_reader = new BinaryReader(image_stream))
        using (FileStream label_stream = File.OpenRead(label_file_name))
        using (var label_reader = new BinaryReader(label_stream))
        {
            int trash;
            for (int i = 0; i < 4; i++) {
                trash = image_reader.ReadInt32();
            }
            for (int i = 0; i < 2; i++) {
                trash = label_reader.ReadInt32();
            }

            for (int i = 0; i < num_data; i++) {

                float pix_max = 0f;
                for (int s = 0; s < SIZE; s++) {
                    img[s] = Convert.ToSingle(image_reader.ReadByte());
                    if (pix_max < img[s]) pix_max = img[s];
                }

                for (int s = 0; s < SIZE; s++) {
                    network.input_layer.x[s] = img[s] / pix_max;
                }

                int label = (int)label_reader.ReadByte();

                network.input_layer.forward(ref network.input_relu.x);
                network.input_relu.forward(ref network.hidden_layer.x);
                network.hidden_layer.forward(ref network.hidden_relu.x);
                network.hidden_relu.forward(ref network.softmax_layer.x);
                network.softmax_layer.forward();

                if (mut_weights) {
                    network.softmax_layer.backward(ref network.hidden_relu.dy, label);
                    network.hidden_relu.backward(ref network.hidden_layer.dy);
                    network.hidden_layer.backward(ref network.input_relu.dy);
                    network.input_relu.backward(ref network.input_layer.dy);

                    network.input_layer.update();
                    network.hidden_layer.update();
                }
                if (score) {
                    if (network.softmax_layer.predict == label) num_collect += 1;
                }

                if (loss) loss_sum += network.softmax_layer.loss;
            }
            Console.WriteLine("====================================");
```

```
                if (test) Console.WriteLine("TEST");
                else Console.WriteLine("Epoch: {0}", epoch);
                if (score) Console.WriteLine("score: {0}", (float)num_collect/(float)num_data);
                if (loss) Console.WriteLine("loss: {0}", loss_sum/(float)num_data);
                Console.WriteLine("==================================");
            }
        }

}

class Layer
{
        public float[] x;
        public int input_size;
        public int output_size;
}

class Affine : Layer
{
        private float alpha;
        private float learning_rate;
        private float w_max;
        private float w_min;
        public float[,] w;
        public float[] dy;
        public float[,] dw;
        public float[,] v;

        public Affine(int n_inputs, int n_outputs, float a, float lr, float w_ma, float w_mi, ref Random
rnd)
        {
            input_size = n_inputs;
            output_size = n_outputs;
            alpha = a;
            learning_rate = lr;
            w_max = w_ma;
            w_min = w_mi;
            x = new float[input_size];
            w = new float[input_size, output_size];
            dy = new float[output_size];
            dw = new float[input_size, output_size];
            v = new float[input_size, output_size];

            for (int i = 0; i < input_size; i++) for (int u = 0; u < n_outputs; u++)
                w[i, u] = (float)rnd.NextDouble();
        }

        public void forward(ref float[] y)
        {
            for (int u = 0; u < output_size; u++) {
                float output_sum = 0f;
                for (int i = 0; i < input_size; i++) output_sum += x[i] * w[i, u];
                y[u] = output_sum;
            }
```

```
    }

    public void backward(ref float[] dx)
    {
        for (int i = 0; i < input_size; i++) {
            float dx_sum = 0f;
            for (int u = 0; u < output_size; u++) dx_sum += dy[u] * w[i, u];
            dx[i] = dx_sum;
        }
    }

    public void update()
    {
        for (int i = 0; i < input_size; i++) for (int u = 0; u < output_size; u++) {
            v[i, u] = alpha * v[i, u] - learning_rate * x[i] * dy[u];
            w[i, u] += v[i, u];
            if (w[i, u] < w_min) w[i, u] = w_min;
            else if (w[i, u] > w_max) w[i, u] = w_max;
        }
    }
}

class ReLu : Layer
{
    public float[] dy;

    public ReLu(int size)
    {
        input_size = size;
        output_size = size;
        x = new float[size];
        dy = new float[size];
    }

    public void forward(ref float[] y)
    {
        for (int i = 0; i < input_size; i++) {
            if (x[i] >= 0) y[i] = x[i];
            else y[i] = 0f;
        }
    }

    public void backward(ref float[] dx)
    {
        for (int i = 0; i < input_size; i++) {
            if (x[i] >= 0) dx[i] = dy[i];
            else dx[i] = 0f;
        }
    }
}

class SoftmaxWithLoss : Layer
{
    public float loss;
```

```
	public int predict;
	public float[] y;

	public SoftmaxWithLoss(int size)
	{
		input_size = size;
		output_size = size;
		x = new float[input_size];
		y = new float[output_size];
	}

	public void forward()
	{
		float input_max = 0f;
		predict = 0;
		for (int i = 0; i < input_size; i++) {
			if (input_max < x[i]) {
				input_max = x[i];
				predict = i;
			}
		}

		float soft_sum = 0f;
		for (int i = 0; i < input_size; i++) soft_sum += (float)Math.Exp(x[i] - input_max);
		for (int u = 0; u < output_size; u++) y[u] = (float)Math.Exp(x[u] - input_max) / soft_sum;
	}

	public void backward(ref float[] dx, int label)
	{
		if (y[label] < 1e-7f) loss = -(float)Math.Log(1e-7f);
		else loss = -(float)Math.Log(y[label]);

		for (int i = 0; i < input_size; i++) {
			if (i == label) dx[i] = y[i] - 1f;
			else dx[i] = y[i] - 0f;
		}
	}
}

class Network
{
	const int NUM_HIDDEN = 500;
	const int NUM_LABEL = 10;
	const float LEARNING_RATE = 0.001f;
	const float ALPHA = 0.9f;
	const float W_MAX = 1e7f;
	const float W_MIN = 1e-7f;
	private Random rnd = new Random();

	public Affine input_layer;
	public ReLu input_relu;
	public Affine hidden_layer;
	public ReLu hidden_relu;
	public SoftmaxWithLoss softmax_layer;
```

```
    public Network(int size)
    {
        input_layer = new Affine(size, NUM_HIDDEN, ALPHA, LEARNING_RATE, W_MAX,
W_MIN, ref rnd);
        input_relu = new ReLu(NUM_HIDDEN);
        hidden_layer = new Affine(NUM_HIDDEN, NUM_LABEL, ALPHA, LEARNING_RATE,
W_MAX, W_MIN, ref rnd);
        hidden_relu = new ReLu(NUM_LABEL);
        softmax_layer = new SoftmaxWithLoss(NUM_LABEL);
    }
}
```

では、順に見ていこう。

リスト 1-5 Program（フィールド定義）

```
const int EPOCH = 100;
const int SIZE = 28*28;
const int NUM_TRAIN = 600;
const int NUM_TEST = 100;
static float[] img = new float[SIZE];
static Network network = new Network(SIZE);
```

ここでは、Program クラス（メインのクラス）で使用する定数とフィールドを定義している。EPOCH は訓練を繰り返す回数、NUM_TRAIN は訓練に用いる画像の数、NUM_TEST は訓練後の性能テストに用いる画像の枚数である。MNIST には訓練用に 6 万枚、テスト用に 1 万枚の画像が用意されているが、すべて使用すると非常に時間がかかるため、ごく一部に限定する。MNIST の画像データは 28x28 の行列で格納され、0 から 255 の数値によってモノクロ画像の濃淡を表現している。この大きさを定数 SIZE に与える。

Float 型の配列である img はこの画像データを受け取って処理を行うためのフィールドである。実際のデータは縦 28、横 28 の行列になっているが、これをニューロンが 1 列に並んだ層で処理するため、1 次元に変形して受け取る。

最後の network は独自に実装した Network 型のオブジェクトである。この中に、計算に用いるニューラルネットワークが構築されている。では、少し飛んで、先に Network の中身を見てみよう。

リスト 1-6 Network

```
class Network
{
    const int NUM_HIDDEN = 500;
    const int NUM_LABEL = 10;
    const float LEARNING_RATE = 0.001f;
    const float ALPHA = 0.9f;
    const float W_MAX = 1e7f;
    const float W_MIN = 1e-7f;
```

```
    private Random rnd = new Random();

    public Affine input_layer;
    public ReLu input_relu;
    public Affine hidden_layer;
    public ReLu hidden_relu;
    public SoftmaxWithLoss softmax_layer;

    public Network(int size)
    {
        input_layer = new Affine(size, NUM_HIDDEN, ALPHA, LEARNING_RATE, W_MAX, W_MIN,
ref rnd);
        input_relu = new ReLu(NUM_HIDDEN);
        hidden_layer = new Affine(NUM_HIDDEN, NUM_LABEL, ALPHA, LEARNING_RATE, W_MA
X, W_MIN, ref rnd);
        hidden_relu = new ReLu(NUM_LABEL);
        softmax_layer = new SoftmaxWithLoss(NUM_LABEL);
    }
}
```

ネットワークの構造は、public Network(int size){}の中身からわかる通り、

1. 入力層
2. ReLu 関数
3. 隠れ層
4. ReLu 関数
5. Softmax 関数＋Cross-Entropy 誤差

の構成になっている。入力層・隠れ層のように、入力と重みを掛け合わせ、総和を次の層に伝播する層を Affine 層と呼び、独自のクラスとして実装している。ReLu 関数についても同様であり、Softmax 関数は Cross-Entropy 誤差の計算とまとめて 1 つのクラスとして実装した。なお、入力層のニューロンの数は定数 SIZE と同数であり、隠れ層は 500 個（= NUM_HIDDEN)、出力層は 0 から 9 の数字に対応する 10 個（= NUM_LABEL）のニューロンによって構成されている。定数 W_MAX と W_MIN はそれぞれ、重みの最大値と最小値であり、ALPHA については後述する。

リスト 1-7 Layer

```
class Layer
{
    public float[] x;
    public int input_size;
    public int output_size;
}
```

Affine, Relu, SoftmaxWithLoss の各クラスは、Layer クラスを継承する。Layer には、全クラスが共通して持つフィールド x（入力を受け取る配列）、input_size（受け取る入力の数）、output_size（出力の数）を格納する。

リスト 1-8 Affine

```
class Affine : Layer
{
    private float alpha;
    private float learning_rate;
    private float w_max;
    private float w_min;
    public float[,] w;
    public float[] dy;
    public float[,] dw;
    public float[,] v;

    public Affine(int n_inputs, int n_outputs, float a, float lr, float w_ma, float w_mi, ref Random
rnd)
    {
        input_size = n_inputs;
        output_size = n_outputs;
        alpha = a;
        learning_rate = lr;
        w_max = w_ma;
        w_min = w_mi;
        x = new float[input_size];
        w = new float[input_size, output_size];
        dy = new float[output_size];
        dw = new float[input_size, output_size];
        v = new float[input_size, output_size];

        for (int i = 0; i < input_size; i++) for (int u = 0; u < n_outputs; u++)
            w[i, u] = (float)rnd.NextDouble();
    }

    public void forward(ref float[] y)
    {
        for (int u = 0; u < output_size; u++) {
            float output_sum = 0f;
            for (int i = 0; i < input_size; i++) output_sum += x[i] * w[i, u];
            y[u] = output_sum;
        }
    }

    public void backward(ref float[] dx)
    {
        for (int i = 0; i < input_size; i++) {
            float dx_sum = 0f;
            for (int u = 0; u < output_size; u++) dx_sum += dy[u] * w[i, u];
            dx[i] = dx_sum;
        }
    }
```

```
    public void update()
    {
        for (int i = 0; i < input_size; i++) for (int u = 0; u < output_size; u++) {
            v[i, u] = alpha * v[i, u] - learning_rate * x[i] * dy[u];
            w[i, u] += v[i, u];
            if (w[i, u] < w_min) w[i, u] = w_min;
            else if (w[i, u] > w_max) w[i, u] = w_max;
        }
    }
}
```

Affine は Layer に追加して、w（重みを格納する配列）、dy（下位層からの勾配を受け取る配列）、dw（重みごとの勾配を格納する配列）、v（各重みに対応したモーメントを格納する配列：後述）のほか、重みの更新に必要な係数のフィールドを持つ。配列 w は 0 から 1 のランダムな値、v は 0 で初期化しておく。

forward 関数は順伝播（通常の計算）を制御する。引数に float 型の配列として次の層の入力配列を取り、この層の入力と重みから、その内容を計算して更新する。

backward 関数は逆伝播を制御し、誤差逆伝播法の実現に関わる。引数に float 型の配列として上位層が受け取る勾配の配列をとり、この層の入力 x に対応する

$$\frac{\partial L}{\partial x_i}$$

を計算して引数の配列を更新する。

update 関数は Affine 層だけが持つ関数であり、重みの更新を制御する。ここで、更新には単純な勾配降下法

$$w_a \leftarrow w_a - \eta \frac{\partial L}{\partial w_{kl}}$$

ではなく、**Momentum 法**を用いる。Momentum 法では、各重みに対応するモーメント v を使用する。重み w_{kl} に対応するモーメント v_{kl} は以下のように更新され、そのモーメントを用いて対応する重みを更新する。

$$v_{kl} \leftarrow \alpha\, v_{kl} - \eta \frac{\partial L}{\partial w_{kl}}$$

$$w_{kl} \leftarrow w_{kl} + v_{kl}$$

単純な勾配降下法では、重みの更新方向は極めて場当たり的、すなわち「一度正に倒したと思ったら今度は負に、そしてまた正に…」という一貫性のない更新が起こりうるが、モーメント v を挟むことで、前回の更新方向が保存される。これによって、重みの更新に**慣性**が生まれ、更新ルールが一貫性を持つようになる。このような一貫性は、訓練の安定化に寄与する。先述の定数 ALPHA は、上式の α の値である。

リスト 1-9 ReLu

```
class ReLu : Layer
{
    public float[] dy;

    public ReLu(int size)
    {
        input_size = size;
        output_size = size;
        x = new float[size];
        dy = new float[size];
    }

    public void forward(ref float[] y)
    {
        for (int i = 0; i < input_size; i++) {
            if (x[i] >= 0) y[i] = x[i];
            else y[i] = 0f;
        }
    }

    public void backward(ref float[] dx)
    {
        for (int i = 0; i < input_size; i++) {
            if (x[i] >= 0) dx[i] = dy[i];
            else dx[i] = 0f;
        }
    }
}
```

ReLu は非常に単純だ。forward 関数で入力が負になるものを 0 に変更し、backward では、下位層から伝播してきた勾配を、ReLu によって 0 が出力されたものは 0 にし、ReLu を素通りしたものはそのまま上位の層に伝播する。

リスト 1-10 SoftmaxWithLoss

```
class SoftmaxWithLoss : Layer
{
    public float loss;
    public int predict;
    public float[] y;

    public SoftmaxWithLoss(int size)
    {
        input_size = size;
        output_size = size;
        x = new float[input_size];
        y = new float[output_size];
    }

    public void forward()
    {
        float input_max = 0f;
```

```
            predict = 0;
            for (int i = 0; i < input_size; i++) {
                if (input_max < x[i]) {
                    input_max = x[i];
                    predict = i;
                }
            }

            float soft_sum = 0f;
            for (int i = 0; i < input_size; i++) soft_sum += (float)Math.Exp(x[i] - input_max);
            for (int u = 0; u < output_size; u++) y[u] = (float)Math.Exp(x[u] - input_max) / soft_sum;
        }

        public void backward(ref float[] dx, int label)
        {
            if (y[label] < 1e-7f) loss = -(float)Math.Log(1e-7f);
            else loss = -(float)Math.Log(y[label]);

            for (int i = 0; i < input_size; i++) {
                if (i == label) dx[i] = y[i] - 1f;
                else dx[i] = y[i] - 0f;
            }
        }
    }
```

SoftmaxWithLoss は先述の通り、Softmax 関数の処理と Cross-Entropy 誤差の計算を合体させたクラスである。Layer の標準フィールドに加えて、誤差を格納する loss 変数、予測ラベルを格納する predict 変数、Softmax 関数の出力を格納する float 型の配列 y を持つ。

このクラス自体が出力先を格納する配列 y を持つので、forward 関数は配列を引数に取らない。forward 関数が実行されると Softmax 関数の計算結果が y に格納され、その中での最大値に対応する分類ラベル（数字）を予測ラベルとして predict に格納する。

ここで、Softmax 関数の計算式には以下のものを用いている。

$$y_k(x) = \frac{\exp(x_k - C)}{\sum_j \exp(x_j - C)}$$

C は定数であるが、ここでは入力 x の最大値 x_{MAX} を用いている。これが **1.3.3** における定義式と等しいことは、以下のように示される。

$$y_k(x) = \frac{\exp(x_k)}{\sum_j \exp(x_j)}$$

$$= \frac{\exp(x_k)}{\exp(x_0) + \exp(x_1) + \cdots + \exp(x_{N-1})}$$

$$= \frac{\exp(-C) \cdot \exp(x_k)}{\exp(-C)\{\exp(x_0) + \exp(x_1) + \cdots + \exp(x_{N-1})\}}$$

$$= \frac{\exp(x_k - C)}{\exp(x_0 - C) + \exp(x_1 - C) + \cdots + \exp(x_{N-1} - C)}$$

$$= \frac{\exp(x_k - C)}{\sum_j \exp(x_j - C)}$$

もとの関数をそのまま使うと、soft_sum(分母)のような変数は容易にオーバーフローする。実際に例を示すと、exp(100)は約 2.69×10^{43} となる。この指数部分から入力 x の最大値を引くことで、指数部分は最大のもので 0、それ以外は負の数を取る。exp の演算は、指数が負の領域ではオーバーフローに対し安全なので、このような変形を用いる。

上記の理由から、２種類の定義式は同値なので、backward 関数で上位の層に伝播する勾配も **1.3.4** で示したものと同様、y_k-1（k が正解ラベルのとき）または y_k-0（k が正解ラベルではないとき）である。

以上で各層の説明は終了である。これらを組み合わせて Network クラスを作り、それを Program クラスの core 関数に渡すことで、訓練とテストが行われる。

リスト 1-11 Program クラス―core

```
static void core(bool test, bool mut_weights, bool score, bool loss, int epoch, ref Network network)
{
	String image_file_name;
	String label_file_name;
	int num_data;
	int num_collect = 0;
	float loss_sum = 0f;

	if (test) {	// ①
		image_file_name = "t10k-images.idx3-ubyte";
		label_file_name = "t10k-labels.idx1-ubyte";
		num_data = NUM_TEST;
	} else {
		image_file_name = "train-images.idx3-ubyte";
		label_file_name = "train-labels.idx1-ubyte";
		num_data = NUM_TRAIN;
	}

	// ②
	using (FileStream image_stream = File.OpenRead(image_file_name))
	using (var image_reader = new BinaryReader(image_stream))
	using (FileStream label_stream = File.OpenRead(label_file_name))
	using (var label_reader = new BinaryReader(label_stream))
	{
		// ③
		int trash;
		for (int i = 0; i < 4; i++) {
			trash = image_reader.ReadInt32();
		}
		for (int i = 0; i < 2; i++) {
			trash = label_reader.ReadInt32();
		}
```

```
            // ④
            for (int i = 0; i < num_data; i++) {

                float pix_max = 0f;
                for (int s = 0; s < SIZE; s++) {
                    img[s] = Convert.ToSingle(image_reader.ReadByte());
                    if (pix_max < img[s]) pix_max = img[s];
                }

                for (int s = 0; s < SIZE; s++) {
                    network.input_layer.x[s] = img[s] / pix_max;
                }

                int label = (int)label_reader.ReadByte();

                network.input_layer.forward(ref network.input_relu.x);
                network.input_relu.forward(ref network.hidden_layer.x);
                network.hidden_layer.forward(ref network.hidden_relu.x);
                network.hidden_relu.forward(ref network.softmax_layer.x);
                network.softmax_layer.forward();

                // ⑤
                if (mut_weights) {
                    network.softmax_layer.backward(ref network.hidden_relu.dy, label);
                    network.hidden_relu.backward(ref network.hidden_layer.dy);
                    network.hidden_layer.backward(ref network.input_relu.dy);
                    network.input_relu.backward(ref network.input_layer.dy);

                    network.input_layer.update();
                    network.hidden_layer.update();
                }
                // ⑥-1
                if (score) {
                    if (network.softmax_layer.predict == label) num_collect += 1;
                }
                // ⑦-1
                if (loss) loss_sum += network.softmax_layer.loss;
            }
            Console.WriteLine("====================================");
            if (test) Console.WriteLine("TEST");
            else Console.WriteLine("Epoch: {0}", epoch);
            // ⑥-2
            if (score) Console.WriteLine("score: {0}", (float)num_collect/(float)num_data);
            // ⑦-2
            if (loss) Console.WriteLine("loss: {0}", loss_sum/(float)num_data);
            Console.WriteLine("====================================");
        }
    }
```

core 関数は引数に test, mut_weights, score, loss の 4 種類のモードと、現在の繰り返し回数を受け取る epoch 変数と、訓練・テストに用いる Network を参照する network 変数を取る。

４種類のモードは以下のように動作する。

1. test
   true のとき、テスト用の MNIST データを読み込む。false のときは訓練用の MNIST データを読み込む。（コメント：①）
2. mut_weights
   true のとき、Network の Affine 層の重みを更新する。test=true のときは false に設定すると良い。（コメント：⑤）
3. score
   true のとき、正解の予測を出した割合を記録し、core の処理が終わったときに表示する。（コメント：⑥-1,2）
4. loss
   true のとき、Network の SoftmaxWithLoss で計算された誤差を記録し、core の処理が終わったときにその平均値を表示する。（コメント：⑦-1,2）

以上のパラメータにより設定されたモードに従って、core 関数は訓練またはテストを実行する。コメント：②の部分では、画像データとラベルデータをそれぞれ読み込む用の変数を作る。その後、test 変数の内容に合わせて画像・ラベルデータを読み込むが、それぞれの冒頭部分にはデータ数・画像サイズなどのメタデータが含まれているため、この不要な情報を trash 変数に読み込んで捨てる（コメント：③）。

この後、次の処理を NUM_TRAIN または NUM_TEST の回数、すなわち指定された画像の枚数だけ繰り返す（コメント：④）。

まず画像データを img に読み込み、全体を img 内の最大値で割ることで、画像の明暗をスケールする。正解ラベルデータは label に読み込む。そして、input_layer → input_relu → hidden_layer → hidden_relu → softmax_layer の順に forward 関数を実行して信号を伝播させていき、予測ラベルを算出する。mut_weights が true のときは、backward 関数を softmax_layer → hidden_relu → hidden_layer → input_relu の順に実行し、勾配を逆伝播させた後、Affine 層の update を行う（コメント：⑤）。score や loss が true のときはそれぞれの値を記録し（コメント：⑥-1, ⑦-1）、最後に表示する（コメント：⑥-2, ⑦-2）。

リスト 1-12 Program クラス―Main

```
static void Main()
{
      for (int e = 0; e < EPOCH; e++) {
            core(false, true, true, true, e, ref network);
      }
      core(true, false, true, false, -1, ref network);
}
```

Main 関数では、core 関数を定数 EPOCH 回実行して Network を訓練したあと、test を true にして未知のデータに対して性能のテストを行う。以上のコードを実行すると、以下のような結果が得られるだろう。

リスト 1-13 network_main.cs の実行

```
C:¥C#¥unity-alife¥chap1>compile.bat network_main.cs

====================================
Epoch: 0
score: 0.1533333
loss: 13.61919
====================================
====================================
Epoch: 1
score: 0.4616667
loss: 8.582331
====================================

（...中略...）

====================================
Epoch: 98
score: 0.9433333
loss: 0.9089858
====================================
====================================
Epoch: 99
score: 0.9333333
loss: 1.058493
====================================
====================================
TEST
score: 0.83
====================================
```

訓練を繰り返すごとに score は上昇し、loss は減少する。100 回訓練を行った後にテストデータを判別させると、そこそこの成績を出すことができた。テストデータに対する成績が訓練データと比較すると悪くなるのは、これが未知のデータだからであるが、それでも適当に答えた場合（期待正答率 10%）よりは有意に高い score を出している。

（**2章**につづく）

書誌情報

「Unity と C#で創る人工生命」見本誌

【発行団体】

USHITORA Lab. https://ushitora.net

【発行履歴】

見本誌 v0 発行　2019 年 11 月 23 日　10:42

見本誌 v1 発行　2019 年 11 月 23 日　15:51

著者情報

大野 駿太郎

【略歴】

富山県高岡市出身。平成 31 年 3 月富山大学医学部医学科卒業。現同大学院博士課程 1 年。研究内容は神経科学。Python, Rust, C/C++, C#, Julia, Common Lisp, Unity などを使い、コンピュータ上で脳のシミュレーションを行っている。

Special Thanks

つくる UOZU プロジェクト　https://detail.uozugame.com

【概要】

富山県魚津市が行っている、ゲームクリエイター養成プロジェクト。開発メンタリングやハッカソン、各種ゲームイベントを通し、街を挙げてゲーム産業を盛り上げる。著者もスタッフとして参加している。