

概説

近年、人工知能(AI)の中で、最も注目を集めているのが機械学習の一種である深層学習(Deep Learning)の技術であろう。ここで、機械学習とは、一般に、大量のデータを学習機に与えて学習させることにより、目的の出力を得る技術である。例えば、犬と猫の写真をどちらかに識別する問題の場合、学習機に犬と猫の写真画像を大量に与え、それと同時にその写真が犬か猫かの正解データを与えることにより(このような学習を教師つき学習と呼ぶ)、学習機に犬か猫かを識別する学習をさせる。このようにして構築したモデルに、犬か猫の写真を与えると、そのどちらかに識別してくれるようになる。深層学習は、このような機械学習の1つであり、学習機は多層構造のニューラルネットワーク(人間の神経回路網を模倣したネットワーク)で構成されている。また、深層学習は、現在、医療、自動運転、自然言語処理、音声認識、データ解析など、様々な分野に適用されており、その有効性が示されている。

そこで、本教材では、ニューラルネットワークを画像認識の分野に適用する下記事例について取り上げ、その手法や性能について言及する。

- ① 文字認識における曲線の屈曲度の判定
- ② 楽譜の音符検出における記号判定
- ③ MLP を用いた数字画像の識別
- ④ CNN を用いた数字画像・写真画像の識別

なお、従来のニューラルネットワークの学習では、画像から識別に有効であると考えられる特徴をあらかじめ抽出し、その特徴量をネットワークの入力に与えて学習する手法が用いられてきた。例えば、犬と猫の識別の場合、目の色が識別に大きく関わっていると開発者が判断した場合、目の色の特徴量を抽出して、それを学習に使用する、といった方式である。本教材では、①と②の事例がこの手法に該当する(ここでは、3層型の単純なニューラルネットワークを用いる)。また、③の事例では、機械学習プラットフォームであるTensorFlowを用いた数字画像識別の例を示す(ここでは、簡単のため、特別な特徴量抽出をせず、画像の輝度値そのものを入力として、数字画像の識別を行う例を挙げる)。一方、④の事例では、現在よく使われている深層学習の1つであるCNN(Convolutional Neural Network:畳み込みニューラルネットワーク)を用いた事例を紹介する。ここでは、特徴抽出機能を包含するネットワークが構築できることを示す(例えば、犬と猫の識別の場合、開発者が目の色のような特徴をあらかじめ抽出して与えることなしに、その写真画像そのものをネットワークに与えて学習することができる)。ここで、現在の最新技術だけを紹介せず、旧来の手法①、②、③も紹介するのは、そこで使われているニューラルネットワークの学習技術が、最新技術の手法④にも使われているからである。

3層型ニューラルネットワークによる学習と識別

ここでは、パターン認識分野の数多くの問題に対して、有効な結果を得ている[59]教師つき学習のできる階層型誤差逆伝搬モデルについて解説する。特にここでは、簡単のために3層型のモデルを用いてみよう。以下に3層型ニューラルネットワーク(MLP:Multi-Layer Perceptron)の学習アルゴリズムの概略を示す。

ニューラルネットワークは、ある値の列を入力として与えた場合に、希望する値の列を出力する、ある種の関数であると考えることができる。例えば、犬猫判別ニューラルネットワークは、犬と猫の写真画像を入力として与えたら、犬であるか猫であるかの判別結果を出力として返してくれる関数である。

それでは、犬と猫を判別するニューラルネットワークを例にとって、どのようにネットワークを構成するのかを見ていこう。まず、値の列を入力したいので、入力画像を縦横等分割し、区切られた領域(画素)の輝度値(白黒の濃淡値)を抽出し、それらを1列に並べて入力列としてみる(本来、画像から何らかの特徴量を抽出してから、入力列とする方法を採用するが、ここでは簡単のため、画像そのものを入力列としてみる)。

ニューラルネットワークは、図1に示すように、○印で示したユニットと呼ばれる部分とそれを結合する線で構成される。先ほどの入力画像から得られた数値列は、図1の最下層の○印に1つずつの値が渡される(例えば、入力値の数が10個なら、○の数も10個である)。この入力を与える最下層のことを入力層と呼ぶ。

入力層の値は、線で結合されている1段上のユニット(○印)に伝達される。その際、単純に伝達するわけではなく、その値に重みの値(図1では重みを●印で示した)を乗算して渡す。1つのユニットには、1段下のユニットから多数の値が伝達されるので、それらを足し合わせて、そのユニットの出力とする。これらを下の層から上の層に向かって伝達していく。その際、これらの伝達はユニット値と重みの積和計算のみで行われるため、線形の関数しか表現することができず、複雑な判別を行うことができない。そこで、各ユニットの出力に活性化関数と呼ばれる関数を介して、非線形関数化することが行われる。このようにして、入力層から入力された値は、上の階層のユニットに向かって計算され、最上層で目的の値の列を出力する。ここで、最上層を出力層と呼び、入力層と出力層の間の層のことを隠れ層あるいは中間層と呼ぶ。なお、図1のように上下層の各ユニットが相互に全て結合している層のことを全結合層と呼ぶ。

それでは、次に、どのように出力層の値を目的の値に近づけるかを考えてみよう。今、犬と猫の判別をする場合、出力層のユニット数が2個であり、犬である場合は、1つ目のユニットの値が1、2つ目のユニットの値が0になり、一方、猫である場合は、それとは逆に、0と1の値になるようにしたいと考えよう。ある入力画像の値を入力層に与えた場合、上記のネットワークの伝達の計算を行うと、何らかの出力値が出力層の2つのユニットに得られる。ここで、入力した画像の正解がわかっていると仮定しよう。つまり、

犬であった場合は出力層の値は「1、0」が正解、猫の場合は「0、1」が正解である。このような正解データのことを教師信号と呼ぶ。ニューラルネットワークでは、現在の出力値と教師信号との誤差を計算し(誤差関数)、その値が小さくなる方向に、ネットワークの重み(●印)の値を修正する作業を行う。この計算は、通常のネットワーク計算とは逆順に、つまり出力層から入力層に向かって、重みの値を修正するように行われるため、誤差逆伝播法と呼ばれている。

このような計算を大量の入力画像とそれに対する教師信号のデータを使って行い、犬猫判別ニューラルネットワークを構成する。これにより、構成したネットワークに、犬猫の画像を入力すれば、出力層に、その結果を出力することができる(この例の場合、出力層でより大きな値を出力するユニットに相当するクラスを識別クラスとすれば良い)。

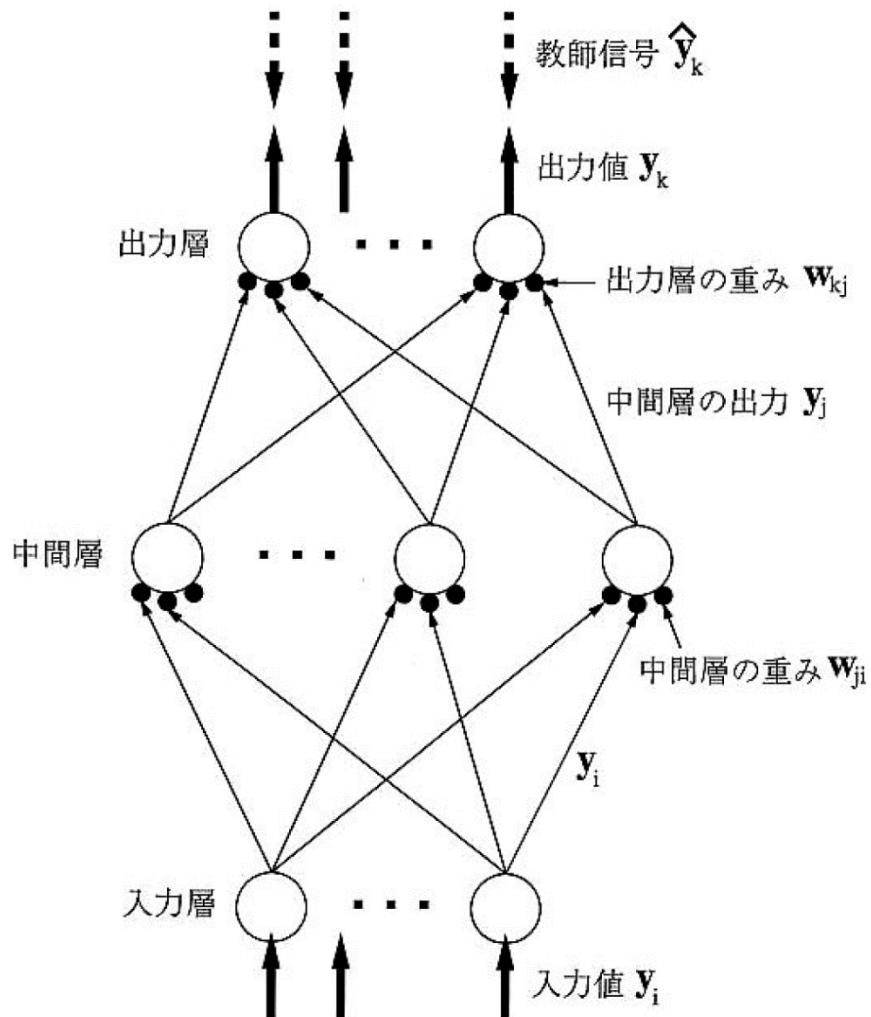


図 1 3層型ニューラルネットワークの構成

それでは、ここまで説明してきた内容を、記号と数式を使って説明してみよう。

今、入力パターン $\{y_i \mid i = 1, 2, \dots, \text{入力値の数}\}$ をネットワークの入力層(図 1を参照)に与えた場合、中間層の出力値 $\{y_j \mid j = 1, 2, \dots, \text{中間層のユニット数}\}$ は、次式で計算される。

$$y_j = f\left(\sum_i w_{ji} \cdot y_i\right) \quad (1)$$

ここで、 $f(x)$ は活性化関数¹と呼ばれ、本手法ではシグモイド関数 $f(x)=1/(1 + \exp(-x))$ を使用する。また、 w_{ji} は中間層の重みの値を示しており、初期値は適当な値を代入しておく。

出力層の出力値 $\{y_k \mid k = 1, 2, \dots, \text{出力層のユニット数 } (k_n)\}$ は、中間層同様、次式で求められる。

$$y_k = f\left(\sum_j w_{kj} \cdot y_j\right) \quad (2)$$

ここで、 w_{kj} は出力層の重みの値であり、中間層の場合と同じく、初期値は適当な値を代入しておく。

次に出力値 y_k と実際に期待する値(教師信号) \hat{y}_k の2乗誤差の総和を暫時小さくするために²、誤差逆伝搬アルゴリズムにより重みの更新を行う(具体的には、誤差関数の値が小さくなる方向に重みを修正する。この計算は誤差関数を重みパラメータで偏微分することによって得られるが、ここではその導出方法は割愛する)。出力層の重みの更新量は、 η を学習定数³とした場合、次式で計算する。

$$\Delta w_{kj} = \eta(\hat{y}_k - y_k)y_k(1 - y_k)y_j \quad (3)$$

中間層の重みの更新量は次式で計算する。

¹ 活性化関数としては、シグモイド関数以外に、tanh、ReLU、Leaky ReLUなどの関数が使われる。

² ここでは誤差関数として二乗和誤差を採用しているが、他にクロスエントロピー誤差を採用する場合もある。

³ 学習定数とは、学習の速度を決定する定数である。小さな値を設定すると学習の進行が遅くなり、局所最適解に収束してしまう可能性が出てくる。一方、大きな値に設定すると学習は速く進むが、最適解に収束しづらくなる。このため、最適な値を設定する必要がある。

$$\Delta w_{ji} = \eta y_j (1 - y_j) y_i \sum_k (\hat{y}_k - y_k) y_k (1 - y_k) w_{kj} \quad (4)$$

多数の入力パターン y_i と教師信号 \hat{y}_k の組をネットワークに与え、上記のアルゴリズムに従い、出力値と教師信号の 2 乗誤差の総和が十分小さくなるまで、重みの更新を繰り返すことにより学習が行われる。

しかし、式(3)、(4)のままでは、重みの更新が極端過ぎて学習が不安定になることがある。そこで、ここでは過去の学習の慣性を利用した次の式を用いることとする⁴。

$$\Delta w_{kj}(t+1) = \eta (\hat{y}_k - y_k) y_k (1 - y_k) y_j + \alpha \Delta w_{kj}(t) \quad (5)$$

$$\Delta w_{ji}(t+1) = \eta y_j (1 - y_j) y_i \sum_k (\hat{y}_k - y_k) y_k (1 - y_k) w_{kj} + \alpha \Delta w_{ji}(t) \quad (6)$$

ここで α は安定化定数と呼ばれ、この値が大きい程過去の学習傾向を重視することになる。 t は学習回数を示す。

また、シグモイド関数 $f = 1/(1+\exp(-x))$ の f 軸の位置も学習の要素とするために、入力層と中間層には常に 1 を出力するユニットを用意した（これは、バイアス項として機能する）。

学習過程に必要となるパラメータをまとめると、次のとおりである。

- 中間層、出力層の重みの初期値 w_{int} （非対称な解も得られるように、通常は小さな乱数値を与える）
- 学習定数 η
- 安定化定数 α
- 中間層のユニット数

これらは、実験を通して最適な値を見つけ出す必要がある。

以上をまとめると、学習アルゴリズムと識別アルゴリズムの概略は次のような形式となる。

〈学習アルゴリズム〉

1. 中間層と出力層の重みを乱数値によって初期化する。
2. 入力パターン（合計 n 個）の各パターン p について、下記の処理を行なう。
 - i. パターン p を入力層に入力し、式(1)に従って中間層の出力値 y_j を求め

⁴ ここで用いた最適化アルゴリズムの方法は、Momentum と呼ばれる手法である。他に確率的勾配降下法(SGD)、AdaGrad、RMSProp、Adam などの方法がある。

る。

- ii. 式(2)に従って出力層の出力値 y_k を求める。
 - iii. 出力層の重みの更新を式(5)を用いて行なう。
 - iv. 中間層の重みの更新を式(6)を用いて行なう。
3. 出力ユニットの平均 2 乗誤差 $\frac{1}{n} \sum_p \left\{ \frac{1}{k_n} \sum_k (\hat{y}_k - y_k)^2 \right\}$ が、ある設定値以上である場合は、処理 2.に戻って繰り返す。設定値より小さい場合は終了。

〈識別アルゴリズム〉

学習アルゴリズムで得られたネットワークを用いて、下記の処理を行なう。

1. 識別したいパターンを入力層に入力し、式(1)に従って中間層の出力値 y_j を求める。
2. 式(2)に従って出力層の出力値 y_k を求める。
3. 出力値に応じて、識別判定を行なう。

なお、識別判定は、そのままの出力値を用いて判定する場合（恒等関数による判別）とソフトマックス関数を用いて判定する場合などがある。

適用事例：文字認識における曲線の屈曲度の判定

序言

手書き文字の中には多くの類似文字が存在する。英数字を例にとれば、「5-S」、「U-V」、「2-Z」などの類似文字の組みが挙げられる。これに対し、多くの商用OCR(Optical Character Reader)では、これらの類似文字を簡単に区別できるように、書き手側に模範的な書き方の指針を与えており、そこでは、類似文字の一方、あるいは両方について慣習的な文字の書き方にわずかな変更を加えている。例として、図2は、「U-V」と「5-S」に対する筆記例を示す。

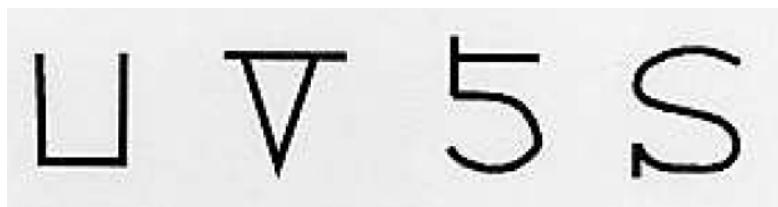


図2 手書き文字の筆記法に関する拘束条件の例

このような筆記法に関する拘束は、「O(アルファベットの O)-0(零)」の組に対しては必要不可欠なものであるかもしれない。しかし、これらの条件を与えても、必ずしも筆記者が、その全てを満足するような書き方をするとは限らず、印刷文字として伝統的に書かれてきた文字の形状に従って書いてしまう傾向があることは否定できない。そこでこれらの類似文字を含む手書き文字に対する従来の認識法では、段階的な手法が広く採用されている。ここでは、ほとんどの文字が第一段階で識別されるが、二つのカテゴリーのどちらにも採用できる文字に対しては、この段階では認識せず、第二段階の処理へと移行する。この第二段階にまわされる文字は上記のような類似文字群であり、第二段階では、これらの類似文字群の識別を行う。このような識別処理をズーミング(zooming)と呼んでいる。

中野[47]は、文字認識のエキスパートシステムに関する提案をしており、そこでズーミングシステムの必要性を指摘している。しかし、類似文字の組は多種多様に存在し、これら全てに対応するズーミングシステムを構築することは難しい。文字認識に精通している技術者であれば、適切な識別ルールを見出してプログラムを作成することは、不可能なことではないかもしれない。しかし、それらのプログラムが全ての組に適用できるとは限らない。たとえ全てに対応するプログラムを作成できたとしても、それらに含まれるパラメータの調整が必要であり、結局、大変な労力と時間を費すことになることは明らかである。このような理由から、ズーミングの自動設計あるいはコンピュータ支援の設計手法が必要となる。

そこで、ここでは、ニューラルネットワークを用いて、上記のズーミング手順を自動化するための事例についての実験を紹介していこう。今回、対象として行ったのは、曲線の尖鋭度に関する識別である。これは、例えば「2-Z」、「5-S」、「U-V」、「D-O」のように、その文字に含まれる曲線の尖鋭度によって識別が可能な類似文字群が数多く存在しているためである。

「5-S」のような類似文字は、複数の曲線セグメントから構成されていると見ることができる。当然、類似文字の識別をするためには、双方の文字中の尖鋭度の異なる曲線セグメント部分を処理対象とするべきであり、それらの曲線セグメントを検出する前処理が必要となる。曲線セグメントの検出は、例えば OPLM(Outmost Point List Method)[50] 手法を用いることにより解決できる。この方法は、入力文字パターンの輪郭を抽出後、それらの曲線を凸と凹に分類することができる。図 3 に類似文字「5-S」の場合の凸と凹セグメントの分割例を示した。このように分割した後、「5-S」の識別には、図 3 のセグメント番号 1 の凸曲線を対象にして尖鋭度の識別処理を行えば良い。

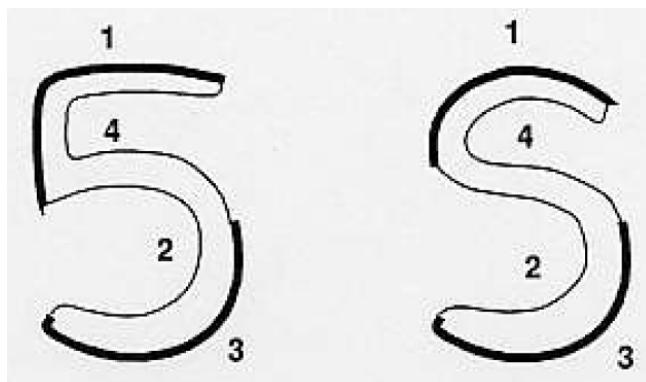


図 3 輪郭線のセグメント分割の例

太線:凸の曲線部分

細線:凹の曲線部分

ここでは、問題を簡単化するために、識別すべき一本の上に凸な曲線セグメントが上記のような方法であらかじめ抽出され、その曲線の両端点と屈曲点(頂点)の位置情報が既知であると仮定して処理を行ってみる。

凸曲線セグメントの曲がり具合の特徴をニューラルネットワークに入力するために、ここでは、次の二つの方法を用いて特徴のコード化を行う。

1. 曲線の輪郭をフリーマンコードを用いてコード化する手法
2. 曲線の外輪郭と内輪郭の速度差をコード化する手法

これらのコード化法を用い、ニューラルネットワークにより曲線の尖鋭度の識別が可能かどうかを調査する。

フリーマンコード化を用いた曲線のコード化

ニューラルネットワークを活用するためには、曲線パターンから曲がり具合についての何らかの特徴量を抽出して、ネットワークの入力とするためにそれらをコード化する必要がある。ここでは、曲線の屈曲部分の方向成分を特徴として採用するためにフリーマンコードを用いたコード化の手法を使用してみる。

フリーマンコードは、線図形を正方格子座標系において折れ線近似表現するコード化手法である。これは図 4 に示すように、各単位方向に図に示すような数値を与える、折れ線を 0 から 7 の数値の列として表現する。例えば、図 5 に示す線図形は「4564356707」の数値列にコード化される。

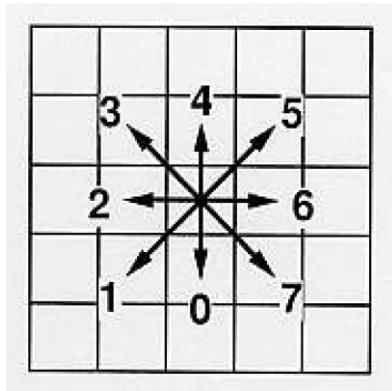


図 4 フリーマンコードの方向と数値

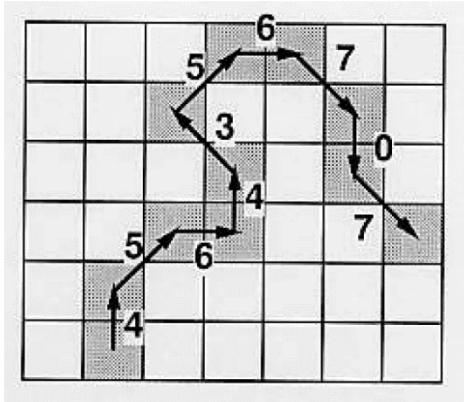


図 5 曲線のフリーマンコード化の例

解像度の高い実際の曲線画像に対して各画素を単位としてフリーマンコード化を行うと、非常に大きな数列になってしまう。よって、ここでは曲線を大まかに水平分割して、フリーマンコード化することを考える。図 6 に示すように、上に凸な曲線セグメントにおいて、左端点($x_{-10}, y(x_{-10})$)、屈曲点($x_0, y(x_0)$)、右端点($x_{10}, y(x_{10})$)の座標が既知であるとした場合のフリーマンコード化は次のように行う。

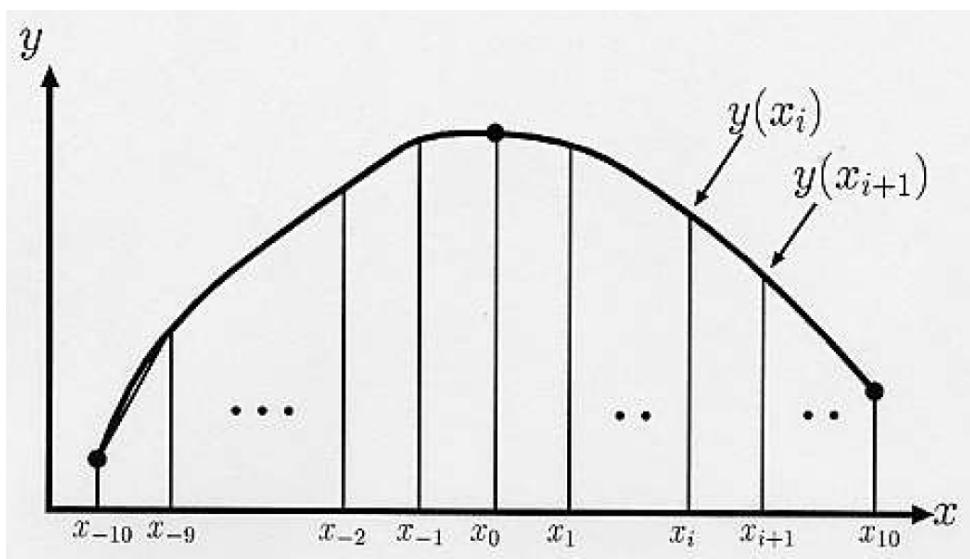


図 6 曲線の分割

まず、軸の区間 $[x_{-10}, x_0]$ と $[x_0, x_{10}]$ をそれぞれ等間隔に 10 分割し、分割した x 軸の各点の値を $x_{-10}, x_{-9}, \dots, x_0, x_1, x_2, \dots, x_i, \dots, x_{10}$ で示す。そこで、各整数 $\{i \mid -10 \leq i \leq 9\}$ について 座標 $(x_i, y(x_i)), (x_{i+1}, y(x_{i+1}))$ を結ぶ区分的な直線を考える。この直線を次のアルゴリズムを用いてフリーマンコードに変換する。

```

for i := -10 to 9 do begin
    loop := (y(x[i+1]) - y(x[i])) / (x[i+1] - x[i]) の絶対値の整数部分
    if (loop = 0) then フリーマンコード 6 を生成
    else begin
        if (i < 0) then begin
            フリーマンコード 5 を生成
            loop-1 回分、フリーマンコード 4 を生成
        end{if}
        else begin {i >= 0 のとき}
            loop-1 回分、フリーマンコード 0 を生成
            フリーマンコード 7 を生成
        end{else}
    end{else}
end{for}

```

ここで、コード化の起点が上に凸な曲線の左端点であるため、上記のアルゴリズムでは 1, 2, 3 の方向コードを持つ直線は存在しないと仮定している。例として、図 7 (a) の

区分的直線は、(b)のようなフリーマンコードに変換される。

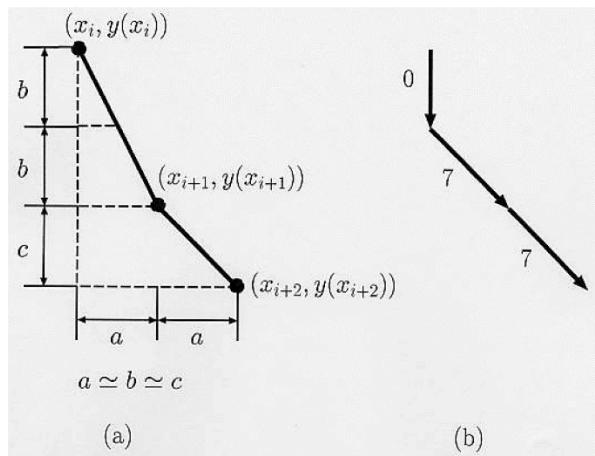


図 7 フリーマンコード化の例

採用するフリーマンコード列は、曲線の屈曲点付近に重要な特徴が含まれることと、ニューラルネットワークに入力するコードの数を全ての曲線について同一にするために、曲線の屈曲点(座標 $(x_0, y(x_0))$)を中心に、図 8 に示すように左右に各 10 個のフリーマンコードを採用する。よって、コード化のアルゴリズムからもわかるように、曲線が鋭く屈曲するほど フリーマンコード列は、その屈曲した頂点付近に集中することになる。

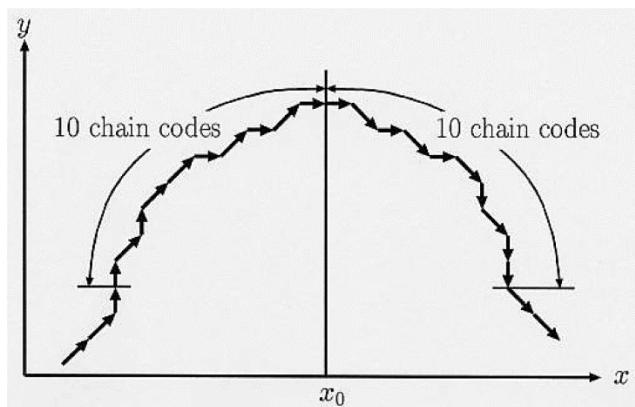


図 8 採用するフリーマンコード列

さらに、フリーマンコード列を回転不变な量とするために、隣合ったコードの一階差分を求め、新たに一階差分化したフリーマンコード列として採用する。隣合ったフリーマンコードの値 a 、 b から、その一階差分 c を求めるアルゴリズムは次のとおりである。

```
if (|b - a| > 4) then c := {8 - |b - a|} * sign(b - a) * (-1);
```

```
if (|b - a| = 4) then c := 4;  
if (|b - a| < 4) then c := b - a;
```

ここで、 $\text{sign}(x)$ は x の符号を求める関数であり、 x が正なら 1、負なら -1、0 なら 0 を返す。一階差分化したフリーマンコードの値を 0 から 7 の値にするために実際には一階差分 $c + 3$ の値を用いる。また、ニューラルネットワークの入力層には、0 から 7 の値を 3 ビットで表現し、各ビットの値を入力することとする。

上記のコード化をビット列で表現した場合、隣り合った方向のコードのビット表現において、必ずしも同じだけの数字の変化があるわけではない。つまり、0 から 7 を 2 進数で 000, 001, 010, 011, 100, ..., 111 と表したとき、000-001 間はビットが 1 つ変化するだけだが、001-010 間は 2 つ変化し、かつ 111-000 間は隣り合った方向コードであるにも関わらず、3 つのビット変化が生じる。そこで、コード変換部分の画像誤差に対する耐性を向上させる目的で、隣り合う方向のフリーマンコードのハミング距離が常に 1 となる Gray 符号化を採用してみる。

Gray 符号化は、0 から 7 までの数値を 000,001,011,010,110,111,101,100 で表現するものである。

曲線の外輪郭と内輪郭の速度差を用いた曲線のコード化

曲線のコード化のための二つ目の特徴として、曲線の外輪郭と内輪郭の速度差に注目してみる。これは、曲線の輪郭を陸上競技のトラックに置き換えると理解しやすい。曲線の内側の輪郭(内輪郭)をトラックの内側のコースとし、外側の輪郭(外輪郭)をトラックの外側のコースとする。そこで、二人の選手が同一ラインから同時にスタートして各々のコースを走り、同じゴールに向かって同タイムで走らなければならないすると、トラック(曲線)が鋭く曲がっていると、外側のコースを走る選手は大きな不利を感じる。反面、なだらかなトラックの場合は、あまりハンディキャップを感じない。これは当然ながらトラックが鋭く曲がっているほど、外側の選手は内側の選手より速く走らねばならないからである。この観点から、内と外の二つのコース(輪郭上)の選手の速度差を曲線の尖鋭度の特徴として採用することにする。

実際の曲線画像上で速度差を表現するために、西田ら[49]が提案した細線化アルゴリズムの手法を応用してみよう。そこでは、図 9 のように、内輪郭と外輪郭上に等間隔に点を定め、各点と、反対側の点列からその点に最も距離が近い点とを対応させる。そして、両側の輪郭線上の点を端から辿り、一方が次の点に移行する間に、他方がどれだけ移動するかを 0, 1 の値によってコード化する。ここで、0 は進度がなく、1 は輪郭線上の点を 1 つ分、先に進むことを意味する。

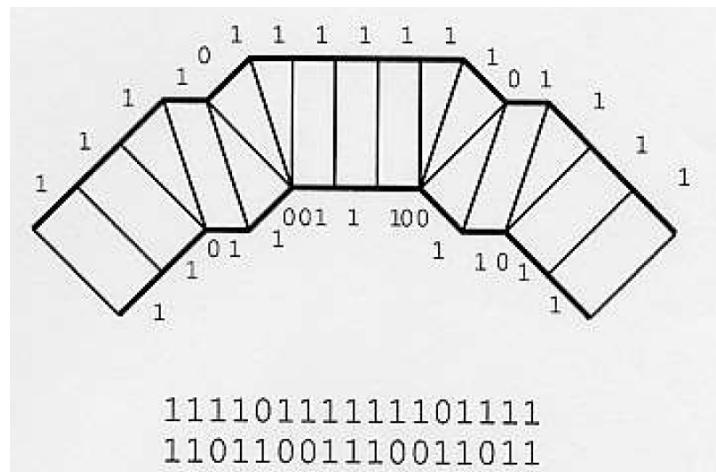


図 9 内輪郭と外輪郭の対応点とそのコード化

曲線の尖鋭度を表現する特徴ベクトルとして、このビット列を使用する。この特徴ベクトルは、両側の輪郭の差分処理によって生成しているため、曲線パターンの回転に不変な量となる。

評価実験

フリーマンコード化による手法では、計算によって求めたモデル曲線と、手書き曲線に対する評価実験を行ってみる。また、外輪郭と内輪郭の速度差を特徴ベクトルとする手法では、手書き曲線の他に実際の手書き文字「U-V」、「（ - く」に対する識別を行ってみよう。

ニューラルネットワークの構成は、3層型のネットワークを採用している。ここで、入力層のユニット数は、特徴ベクトルのビット列分の数を用意し、中間層のユニット数は5、出力層のユニットは対象の曲線が「まるい」とき「0」、「するどい」ときには「1」を指す一つのユニットとする。中間層は種々の実験から最適と思われるユニット数に設定している。また、ネットワークの学習過程での各パラメータの値は、学習定数 $\eta = 0.2$ 、安定化定数 $\alpha = 0.9$ 、中間層と出力層の重みの初期値 $w_{int} = 0.5$ のように設定した。

フリーマンコード化を用いた曲線の識別実験

モデル化曲線の識別

手書き曲線の識別をする前準備としてコンピュータによって生成した曲線(モデル曲線)についての識別を行ってみよう。モデル曲線は、次の関数を用いて生成する。

$$|x|^n + |y|^m = 1 \quad (-1 \leq x \leq 1, n, m > 0) \quad (7)$$

この関数によって生成される曲線は図 10 に示すように $(-1, 0), (1, 0)$ を端点、 $(0, 1)$ を頂点(屈曲点)とし、y 軸に対称な曲線となり、 n, m の値が大きくなるほどなめらかな曲線となり、 n, m の値が小さくなるほど頂点で鋭く屈曲した曲線となる。

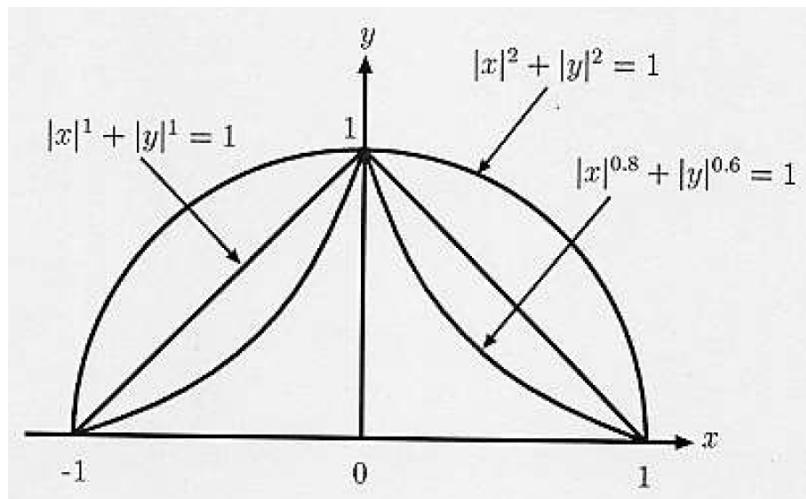


図 10 モデル曲線の例

次に、生成した曲線を図 11 に示すシステムを用いて、被験者に提示する。実験で提示した曲線は、式(7)の n の値を 0.2 刻みで 0.6~2.4、 m の値を 0.2 刻みで 0.2~2.0 まで変化させたときの、合計 100 本の曲線である。被験者はそれらの曲線を見て、それが鋭く曲がっているか、なめらかに曲がっているかを判断して、その判定値を入力した。その結果、鋭く曲がっていると判定された曲線が 49 本、なめらかに曲がっていると判定された曲線が 51 本であった。

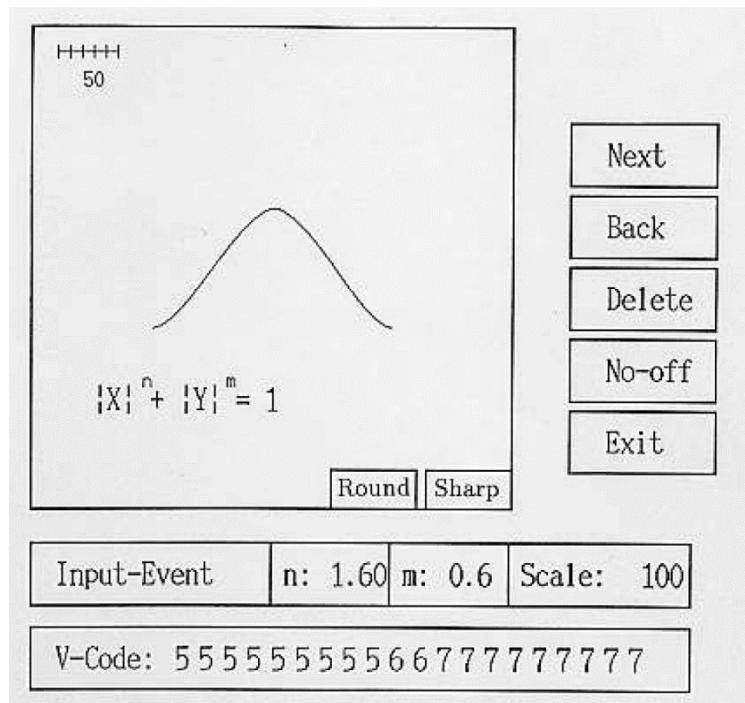


図 11 モデル曲線の提示画面

ニューラルネットワークには、上記の 100 本の曲線とその判定値を与えて学習を行わせた。その際、曲線は「フリーマンコードを用いた曲線のコード化」で述べたコード化手法を用いて、次の 4 つのビットコード列に変換をし、ネットワークの入力としている。

- 通常のフリーマンコード(コード長 60)
- 通常のフリーマンコードを Gray 符号化(コード長 60)
- 一階差分化したフリーマンコード(コード長 57)
- 一階差分化したフリーマンコードを Gray 符号化(コード長 57)

よって、入力層のユニット数は、通常のフリーマンコード化を用いた場合は 60 であり、一階差分化した場合は 57 である。出力層には被験者によって入力された尖鋭度の判定値を教師信号として与え(曲線が「まるい」と判定された場合は「0」、「するどい」と判定された場合は「1」を与える)、「使用するニューラルネットワークの構成と学習法」で述べた誤差逆伝搬アルゴリズムを用いて、ネットワークの重みの学習を行わせた。

学習は 1 サイクルのネットワークの出力値と教師信号の値の平均誤差が 0.0001 に達したところで停止させた。ここで、1 サイクルとは、用いる全ての学習パターン(曲線パターンと、対応する教師信号)をネットワークに与えて、重みの更新をする一回の処理過程のことである。

以上の学習によって得られたネットワークの重みを用いて、学習パターンと未知パターンの識別を行った。学習パターンはネットワークの学習に用いたものと同一の 100

本の曲線である。未知パターンは式(7)の n の値を 0.2 刻みで 0.5~2.3、 m の値を 0.2 刻みで 0.3~2.1 まで変化させたときの、合計 100 本の曲線である。未知パターンの各曲線の尖鋭度の判定値は、学習パターンの判定をしたと同一の被験者に入力してもらった。その結果、未知パターンに関しては鋭く曲がっていると判定された曲線が 50 本、なめらかに曲がっていると判定された曲線が 50 本であった。

識別は学習同様、4 種類の各コードについて行った。識別結果となるニューラルネットワークの実際の出力は実数であり、0.5 をしきい値として、0.5 未満をなめらかな曲線、0.5 以上を鋭い曲線として判定を行っている。この識別結果を表 1 に示す。

表 1 モデル曲線の識別実験の結果

符号化手法	Gray 符号化	データ	学習サイクル [回]	認識率 [%]
通常の フリーマン コード	×	学習パターン(100 本)	554	100
		未知パターン(100 本)	---	99
	○	学習パターン(100 本)	684	100
		未知パターン(100 本)	---	96
一階 差分化 コード	×	学習パターン(100 本)	5445	100
		未知パターン(100 本)	---	97
	○	学習パターン(100 本)	4481	100
		未知パターン(100 本)	---	97

手書き曲線の識別

次に手書き曲線を対象にして識別実験を行ってみよう。 識別までの処理過程は図 12 に示すとおりである。

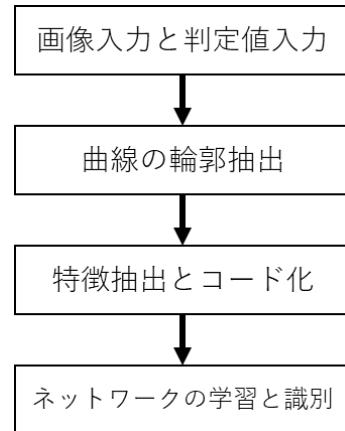


図 12 手書き曲線識別の処理過程

まず、図 13 に示すように曲線の端点 a 、 b と屈曲点 c の位置を被験者に与え、その点を通る、上に凸な曲線を手動で書いてもらう。これは、モデル曲線の認識に用いたネットワークの重みを使って、手書き曲線の尖鋭度の判定が可能かどうかを検証するために、なるべく手書き曲線をモデル曲線に合わせるためにある。また、曲線の大きさや位置の正規化処理を省くことができ、処理を簡略化するため、このような入力手法を採用してみた。

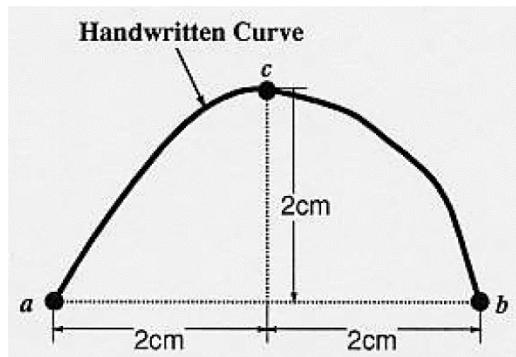


図 13 手書き曲線の書き方

手書き曲線が書かれた画像をイメージスキャナで取り込む。この時のスキャナの解像度は 400dpi であり、20 分割程度のフリーマンコード列を得るには十分な解像度である。 画像濃度は、適当なしきい値によって 2 値化(白黒画像)処理を行う。 また、この画像入力と一緒に、その曲線の尖鋭度の判定値も被験者に入力してもらった。

読み込んだ画像において、曲線を含む 800×800 画素の正方領域を手動で切り出し、その領域のごま塩状の雑音を除くために孤立点除去を行った後、曲線の輪郭線抽出を行った。簡単のために輪郭線の両端(図 14 の a, b)の点の位置を手動で入力し、2 点間を結ぶ内側の輪郭線(図 14 の太線)を手書き曲線画像として抽出、採用した。

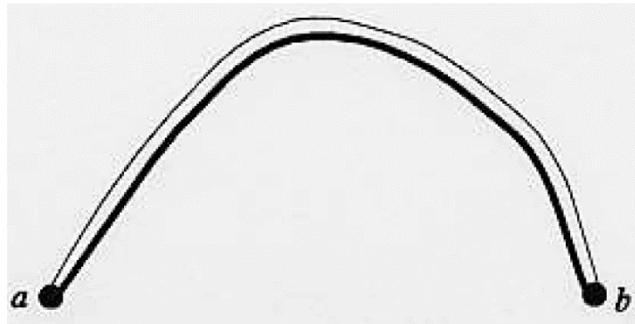


図 14 手書き曲線の輪郭線抽出

抽出した曲線に対して、垂直方向の最上点を検出し、屈曲点として定める。屈曲点の左右の各曲線に対し、水平方向に等間隔に 10 分割し、「フリーマンコード化を用いた曲線のコード化」節で述べた手法を用いて 4 種類のフリーマンコード列を生成する。

上記の方法によって 200 本の手書き曲線を生成し、被験者に判定値を入力してもらったところ、その曲線中、96 本が鋭い曲線と判定され、104 本がまるい曲線として判定された。第一の実験として、モデル曲線の識別で用いたネットワークの重みを用いて、これらの手書き曲線の識別を試みた。その結果を表 2 に示す。

表 2 モデル曲線の識別に用いた重みを使った手書き曲線(200 本)の識別結果

符号化手法	Gray 符号化	認識率[%]
通常のフリーマンコード	×	83.5
	○	87.5
一階差分化コード	×	88.5
	○	91.0

次に第二の実験として、手書き曲線のパターンを再学習させ、その重みを用いて、同様の手書き曲線の識別を行った。ここで、学習用に用いたのは、被験者に新たに書いてもらった 100 本の手書き曲線である。上述と同様の方法で、画像入力、尖鋭度の判定値(教師信号)入力を行い、輪郭線抽出後、曲線を 4 種類のフリーマンコード列に変換した。この曲線中、鋭く曲がっていると判定された曲線は 46 本であり、なめらかに曲がっていると判定された曲線は 54 本であった。これらの手書き曲線の識

別をモデル曲線で用いたニューラルネットワークの学習と同様の方法で学習させた。これによって得られた結果を表 3 に示す。ここで、学習パターンとは、学習に用いた 100 本の手書き曲線であり、未知パターンとは、第一の実験で用意した 200 本の手書き曲線である。

表 3 手書き曲線の識別結果

符号化手法	Gray 符号化	データ	学習サイク ル[回]	認識率 [%]
通常の フリーマン コード	×	学習パターン(100 本)	3148	99.0
		未知パターン(200 本)	--	95.5
	○	学習パターン(100 本)	2692	99.0
		未知パターン(200 本)	--	96.0
一階 差分化 コード	×	学習パターン(100 本)	4262	99.0
		未知パターン(200 本)	--	88.5
	○	学習パターン(100 本)	4416	99.0
		未知パターン(200 本)	--	90.0

識別実験の評価

学習パターンの識別に関して、ニューラルネットワークは、モデル曲線と手書き曲線の両方ともに、ほぼ完全に識別できることがわかる。一方、未知パターンに関しては、被験者が明らかに判定可能な曲線パターンについては、十分満足のいく結果を得た。すなわち、誤認識となった曲線を観察すると、被験者が見ても、まるいか鋭いかの判定がしづらいものが多かったのである。例えば、表 2 の通常のフリーマンコード化したパターンの認識は 200 パターン中、正解が 167 パターンであるが、残りの誤認識パターン 33 本中、27 本の曲線は人間が見ても、鋭く曲がった曲線か、まるい曲線かのどちらともとれるような曲線であった。

通常のフリーマンコードと一階差分化したコードによる認識率の違いに関しては、大きな差異はなかった。よって、曲線の回転に不变な特徴量である 一階差分化したフリーマンコードを用いる方が有利であると言える。

Gray 符号化に関して結果を見ると、Gray 符号化しないパターンより、したパターンの方が全てにおいて、若干、認識率が向上していることがわかる。よって、Gray 符号化を用いることにより、コード変換部分の画像誤差に対する耐性をある程度、向上させることができたと考えられる。

次に認識手法の違いによる結果の違いを検討してみる。モデル曲線に関する認識では、学習、未知パターンともに 96%以上の高い認識率を示した。これに対し、モデル曲線を用いてネットワークの学習をした重みを利用して、手書き曲線の認識を行ったところ、認識率は 83.5%以上となり、かなり低下する。これは、モデル曲線が屈曲点を中心に対称であり、歪みがないのに対して、手書き曲線は非対称で、歪みがあるため、それらが影響していると考えられる。そこで、手書き曲線を学習した重みを使って、上記の手書き曲線を認識した結果は 88.5%以上となった。よって、手書き曲線の学習により、曲線の非対称性や歪みを、ある程度補正する効果が得られたことになる。また、より多くの曲線パターンの学習をすることにより、更に認識率を上げることも可能であると期待できる。

両輪郭の速度差によるコード化手法を用いた識別実験

手書き曲線の識別実験とその評価

最初の実験として 200 本の手書き曲線を用いて識別実験を行った。これらの手書き曲線は、フリーマンコード化手法で被験者に書いてもらった曲線と同一のものである。曲線画像はイメージスキャナでサンプリング、量子化され、白黒の 2 値画像としてコンピュータに取り込む。

各曲線に対して、「曲線の外輪郭と内輪郭の速度差を用いた曲線のコード化」で述べたコード化手法を用いて、特徴ベクトルの抽出を行った。例として、このコード化の処理過程を図 15 に示す。左図は、入力した曲線パターンであり、右図はその外輪郭と内輪郭の対応点を結び、コード化した結果を示している。ここで、屈曲点付近に重要な特徴が含まれることと、ニューラルネットワークに入力するコードの数を同一にするために、採用するコード列は、屈曲点を中心として、両側に 10 ビット分のコードを採用した。つまり、外輪郭、内輪郭の屈曲点の両側に 10 ビット分、合計 40 ビット分のコードとなる。ここで、屈曲点の位置は、簡単のため、手動で入力した。

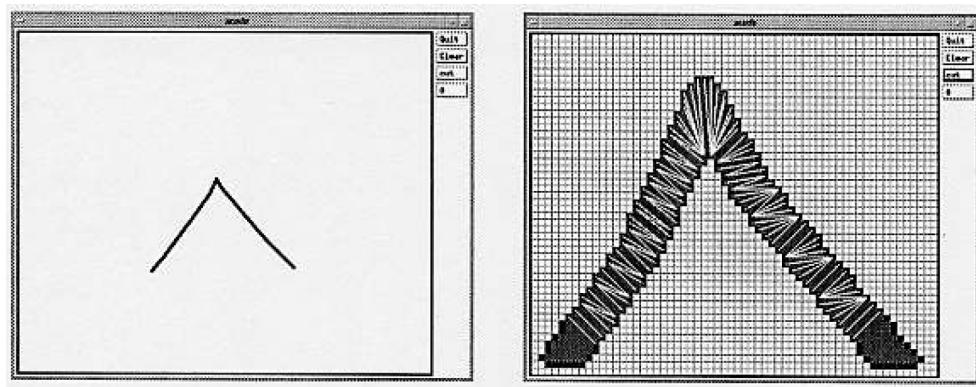


図 15 曲線の外輪郭と内輪郭の対応点の生成過程

左図：入力した曲線パターン(鋭く屈曲している曲線)

右図：輪郭間の対応点の生成。各点の対応線を白線で表示した。

この曲線パターンに対して、生成されたコード列は、

外輪郭 10111111111111111111

内輪郭 11000000001000000001

コード化した 200 本の手書き曲線のうち、100 本をネットワークの学習用パターンとし、残りの 100 本を未知パターンとして使用した。ニューラルネットワークの入力層は、コード化した 40 ビットの値を入力するため、40 個のユニットとしている。ネットワークの学習は、上記の 100 本の学習用曲線パターンのコードを入力層に入力し、出

力層には教師信号として、尖鋭度の判定値(0:まるい、1:鋭い)を与え、前述の誤差逆伝搬学習法を用いて行った。

学習した重みを用いて、100 本の未知パターンの識別を行った。ネットワークの入力には学習時と同様に、曲線をコード化したビット列を与え、出力として得られた値によって、識別を行う。ここで、出力値が 0.5 以上ならば鋭い曲線とし、0.5 未満の時はまるい曲線として識別を行った。つまり、強制的にどちらかに判定する方式を採用した。

識別の結果、学習パターンに対する認識率が 96%に達したところで学習を終了させた場合、未知パターンに対する認識率は 84%であり、100%に達したところで、終了させた場合は 74%の認識率となった。ここで、やや不十分な認識結果となった理由を解析すると、サンプル曲線の境界線が滑らかでなく、雑音を含んでいること、サンプルが少ないことによるそれらの曲線パターンへのオーバーフィッティング（あてはめすぎ）が主な原因として考えられる。また、誤認識となった曲線パターンに対するネットワークの出力値は しきい値の 0.5 付近であることが挙げられ、強制的にどちらかに判定する方式にも問題点があると考えられる。そこで、前処理として曲線の境界部分を滑らかにする処理を施し、更に学習パターンの数を 300 本に増やすとともに、ネットワークの出力値が 0.4~0.6 の場合は識別不能とする範囲を設けて、再実験を試みた。その結果、認識率は、学習パターン、先程の実験と同一の未知パターンに対して、それぞれ 100%、96%となった。

手書き文字「UとV」と「(とく」の識別実験

二つ目の実験として、手書き文字の「 U 」と「 V 」、それぞれ 100 文字を用いて識別実験を行った。これらの文字は、手書き曲線の識別と同様、イメージスキャナでコンピュータに読み込まれ、コード化を行う。

ネットワークの学習には 50 文字の「 U 」と、 50 文字の「 V 」を使用した。手書き曲線の識別と同様の方法で、識別実験を行った結果、認識率は学習パターン、未知パターンともに 100% となった。ここで、未知パターンは、学習で使用した以外の残りの 50 文字の「 U 」と、 50 文字の「 V 」を使用している。 3 つ目の実験は、手書き文字の「 (」と「 く 」、それぞれ 100 文字に対する識別実験である。文字の種類が異なる点を除いては、前述の「 U-V 」の識別と同様の方法で実験を行った。その結果、認識率は学習パターン、未知パターンともに 100% となった。

例として、図 16 と 図 17 に対応点の抽出例を示した。図 16 は鋭い屈曲点を持つ「 < 」に対する例であり、図 17 はまるい曲線を持つ「 (」に対する抽出例である。

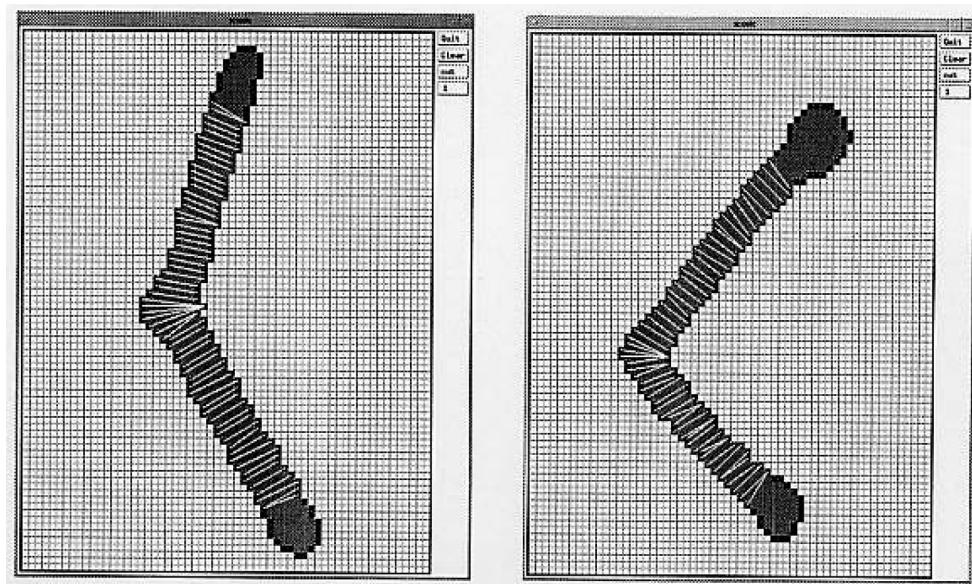


図 16 文字「<」に対する外輪郭と内輪郭の対応点の抽出例

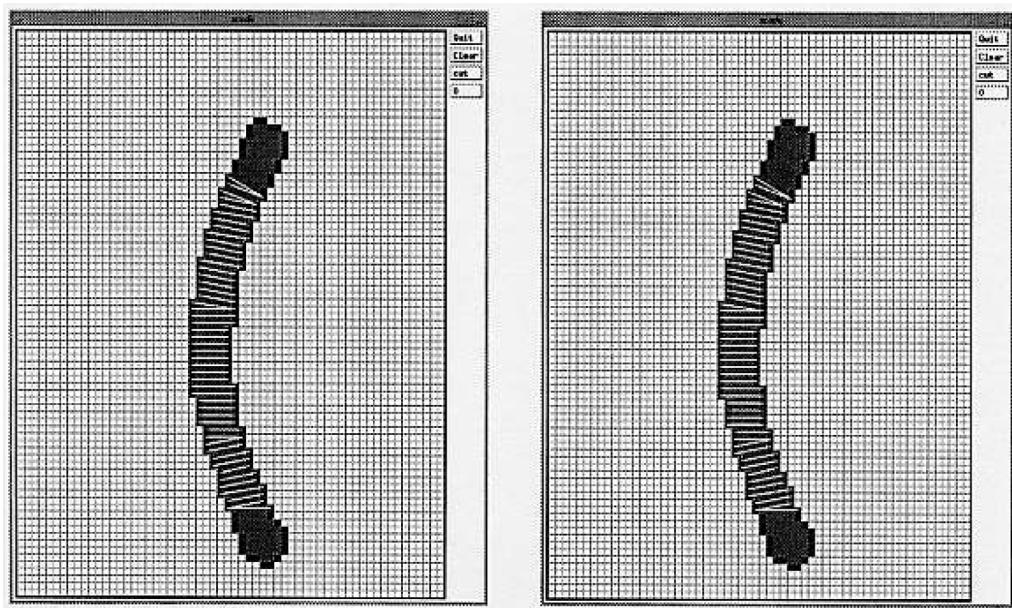


図 17 文字「（」に対する外輪郭と内輪郭の対応点の抽出例

実験のまとめと評価

以上、両輪郭の速度差によるコード化手法を用いた識別実験の結果を表 4 にまとめた。この表から、手書き曲線に対する認識率より、手書き文字に対する認識率の方が良好であることがわかる。これは、カテゴリー分けされていない手書き曲線の尖鋭度の判定では、主観的な判定により強引に二分判定しているために、境界付近での判定が不安定になっていることに起因すると考えられる。すなわち、被験者のその時の気分によって、判定値が異なる場合が生じ、安定しない教師信号が生成される。一方、手書き文字のように最初からカテゴリーが分けられたパターンに対しては、判定が比較的安定しており、これが良い認識結果を導いた要因である。

表 4 識別結果のまとめ

識別対象	サンプルの数			認識率	
	合計	学習パターン	未知パターン	学習パターン	未知パターン
手書き曲線	400	300	100	100%	96%
U-V	200	100	100	100%	100%
(- <	200	100	100	100%	100%

結言

ここでは、文字認識における類似文字の組を識別するためのズーミング手順の自動化のための一つの手段を説明した。ここで着目した類似文字の組は、その文字に含まれる曲線が鋭く屈曲しているか、あるいは、なめらかに曲がっているかで、区別されるものであり、これらを識別することを目的とした。

曲線から特徴ベクトルを抽出する方法として、フリーマンコードによる手法と曲線の外輪郭と内輪郭の速度差を用いる手法の二つを用い、階層型のニューラルネットワークを用いて、曲線の尖銳度の識別を試みた。

その結果、フリーマンコードを用いる手法では、モデル曲線、手書き曲線の未知パターンに対して、それぞれ 96%、88.5%以上の認識率を得た。但し、この判定は強制判定である。これに対し、両輪郭の速度差を用いる手法では、手書き曲線に対し、学習パターン、未知パターンに対して、それぞれ 100%、96%の認識率を得ている。但し、ここでは、ネットワークの出力値に基づき、認識不能という出力も許している。次に、同様の方法により、手書き文字「U-V」と「(- <)」の識別実験を行い、学習パターン、未知パターンの両方に対して 100%の認識率を得た。

これより、曲線から適切な特徴ベクトルを抽出、コード化し、ニューラルネットワークに学習させることにより、自動的に人間の能力に匹敵する識別処理を構築できることがおわかりいただけたと思う。

適用事例：楽譜の音符検出における記号判定

序言

楽譜認識において、最も重要なことは音符を正確に検出することである。なぜなら、音符は楽譜中に最も多く存在し、音の高さ、持続時間、発音のタイミングを決める音楽的に重要な記号であるばかりでなく、音符の位置を基準に描かれる記号が多いため（例えば、シャープ、フラット、アクセントなどは符頭の位置を基準に描かれる）、音符検出精度が他の記号認識の性能に大きな影響を及ぼすからである。よって、強力な音符検出手法が必要となる。

従来の音符検出法について見ると、Prerau[4]、青山ら[6]、Clarke ら[39]は、五線除去後、黒画素の連結領域を切り出し、その外接四角形の大きさと位置で記号の大分類を行い音符を切り出し、認識を行っている。Fujinaga[23]は、水平方向と垂直方向のプロジェクトの形状から音符を切り出し認識を行っている。また、松島ら[16]の手法では、五線位置を基準にして、水平方向に符頭マスクで走査し、パターンマッチングによって符頭を検出後、その周辺探索から符尾、旗を検出する方法を用いている。加藤ら[21]は音符を構成する各記号要素を抽出し、その要素を組み合わせて、音楽的知識に合致したものを楽譜記号とする黒板モデルによる仮説検証処理を行っている。

これらのほとんどの方法は、音符を構成する要素を検出する際、検出過程で得られた種々の特徴量の大小関係や記号要素間の相対的な位置関係を調べ、音符の表記規則に則った要素を検出するために、複雑な if-then ルールによる判別木を使用している。しかし、このような方法は、起こり得る全てのケースを想定して判別木を構築する必要があり、かつその判定中に数多く使用しているパラメータに対して、多くの実験を通して微妙な調整を行う必要がある。

そこで、ここで述べる手法の目的は、従来の音符検出法で必要となる手間のかかる if-then ルールの作成作業とパラメータ調整を、ニューラルネットワークによって代替可能かどうかを検証することである。ここでの検証は、従来法との性能比較や、実際の多くの楽譜に本手法を適用することにより行う。ニューラルネットワークは、前章の結果より、適切な特徴ベクトルのビット列を入力とすることにより、かなり正確に目的とする判定を行うことができる事が示されており、上記の目的に十分適用できると予測される。

以下の節で、ニューラルネットワークを用いた音符検出法の詳細について述べる。

音符検出の流れとその概略

音符検出の処理の流れを図 18 に示す。イメージスキャナで解像度 400dpi、白黒 2 値画像として印刷ピアノ楽譜をコンピュータに読み込んだ後、記号検出の指標となる五線、小節線の位置検出、五線の除去処理を初期段階で行う。これらの具体的な手法は文献[74]を参照されたい。

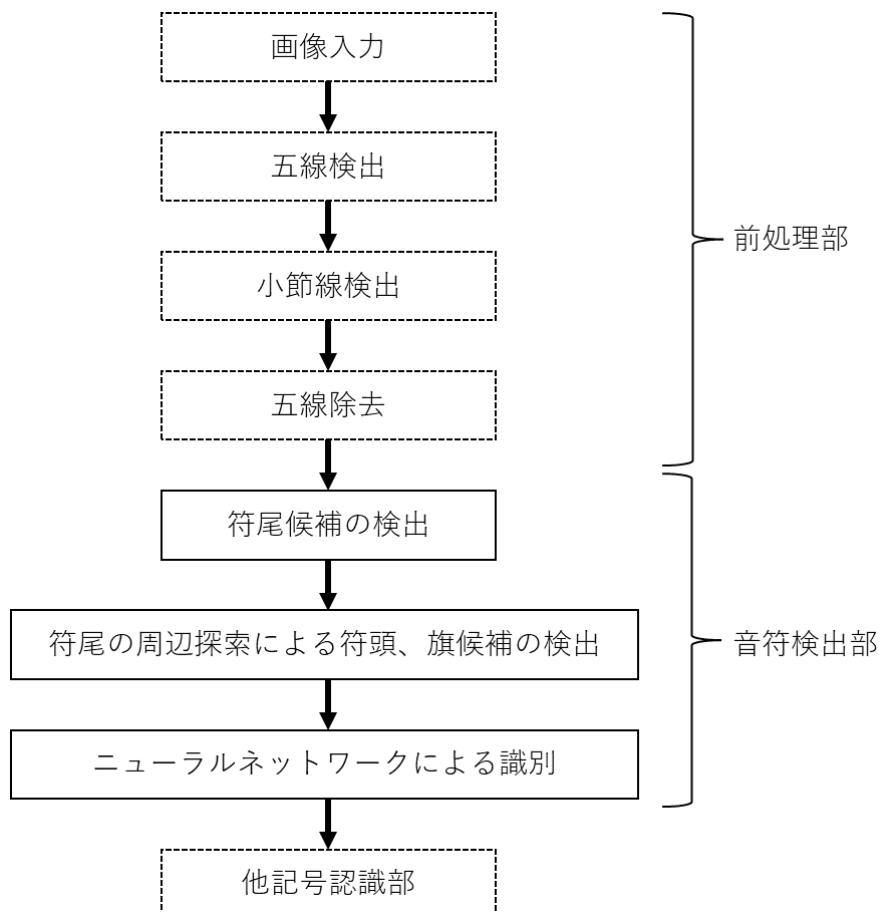


図 18 音符検出の処理過程(本章で扱うのは実線で囲まれた処理)

五線除去後の画像は検出した小節線ごとに区切り、以下の処理はその小節単位に行う。これは、処理範囲を限定し、処理をしやすくするとともに、将来的に小節ごとに並列処理が可能となるため、処理の高速化が期待できるからである。

次に各小節に対して、符尾(音符の棒)の候補を検出し、その符尾の周辺を探索することにより、符頭と旗候補の検出を行う。検出した各符頭、旗の候補に対して、それらの周辺の符尾、符頭、旗候補の相対的な位置関係と、検出時のその記号のもつともらしさの量を特徴量として、ニューラルネットワークにその記号候補が真であるか否かの判定を行わせる。これにより、真の記号要素となった符頭、旗を組み合わせること

とにより、音符の検出ができる。

以下の節では、音符検出部の処理の詳細について述べる。

音符の構成要素の候補検出

音符は図 19 に示すように符頭、符尾、旗、付点から構成されている。符頭には全音符と 2 分音符につく白抜きの楕円形状で描かれる白符頭と、それ以外の音符につき黒塗りの楕円形状で描かれる黒符頭がある。旗は、単独の音符につく単鈎と、音符同士を結んで記述する連鈎からなる。単鈎には図 19 のように符尾の上端につく上単鈎と下端につく下単鈎、連鈎には音符同士を結ぶ長い連鈎と一本の符尾につく短い連鈎が存在する。ここでは、これらの符頭(全音符を除く)、符尾、旗を検出の対象とする。

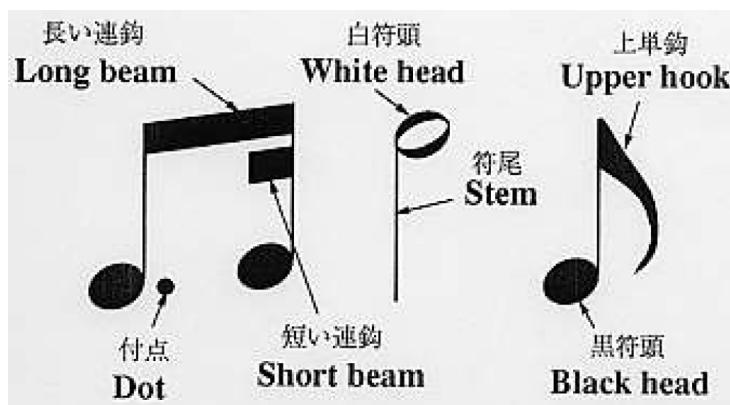


図 19 音符の構成要素とその名称

最初に符頭、符尾、旗であると考えられる候補領域を検出する。これらの記号を全体の画像空間を対象として検出するのは、探索空間が広くなり計算コストがかかるため実用的でない。この問題を克服するために、従来法では五線を基準にして最初に符頭を検出し、その周辺探索により符尾、旗を検出する手法を用いていたが、五線が歪んでいる場合や五線領域外で符頭の位置を予測する場合に誤差が生じやすく、正確な符頭検出ができなかった。

ここでは、全音符以外の音符には必ず符尾が存在し、符頭と旗は必ず符尾に接続している点に注目し、全ての符尾を初期段階で検出できれば、その周辺を探索することにより、符頭と旗を検出できると考えた。また、符尾がこれらの記号群の中で最も単純な形状であるという図形的特徴を考慮しても、符頭や旗より正確に検出できると考えられ、符尾を基準とする周辺探索が望ましい。

候補記号検出で重要なことは、可能性のある領域を全て検出し、未検出領域を極力減らすことである。なぜなら、候補記号検出後のニューラルネットワークの判定では、それらの候補記号が真であるか否かを判定する処理しか行えず、候補記号検出時にとりこぼした未検出領域から新たに記号を再検出することは行えないからである。

符尾の候補検出

符尾はある長さ以上の垂直線分である。この図形的性質を利用し、垂直方向の黒画素のプロジェクション(積算値)から垂直線を抽出することを試みた。プロジェクションは、上部五線と下部五線で囲まれた領域と、その上下に、ある余裕を持たせた領域全てに対して行う。そこで画像中に垂直線分が含まれる位置は、得られたプロジェクションの外形において局所的なピークを形成するので、その位置から符尾の水平方向の位置を予測する。さらに画像に対して、ピーク位置を垂直方向に黒画素を追跡し、ある長さ以上の垂直線成分を検出して、それを符尾の候補とする。

符頭の候補検出

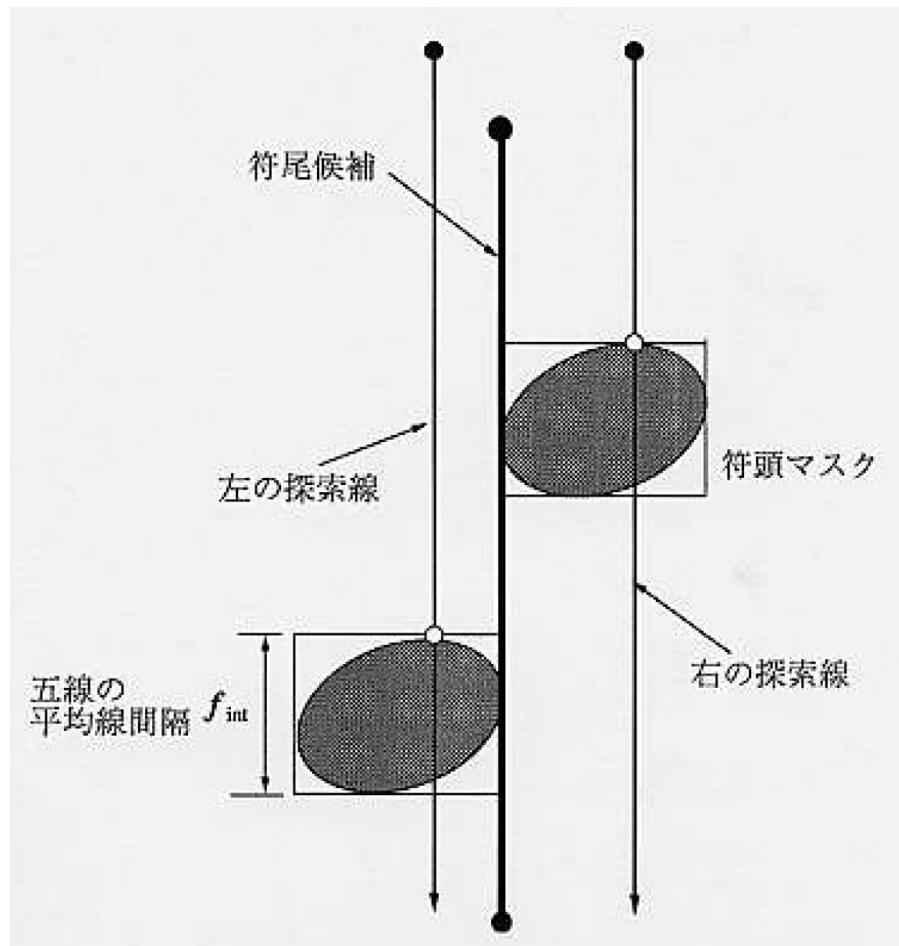


図 20 符頭候補の検出

符頭は、符尾の位置を基準とすると、その両側に存在する可能性がある。よって、図 20 に示すように、検出した各符尾候補の左右を縦方向に黒符頭と白符頭の探索をし、符頭の候補を検出する。探索時には、メッシュ特徴を用いたテンプレートマッチングを行うことにより、あらかじめ用意した数個の標準テンプレートに似通った領域が符頭候補となる。

旗の候補検出

旗は縦方向にある一定の幅をもった帯状の直線あるいは鉤状の曲線分である。したがって、画像中から縦方向の連続した黒ラン成分(連続した黒画素部分)の長さがある範囲に入った部分を抽出する。抽出した部分を黒画素の連結領域でラベリングし、各領域の面積、上輪郭と下輪郭の傾き、縦方向の平均幅の値を基にして、連鉤、単鉤の可能性のある領域を限定する。旗も符頭同様、必ず符尾を伴うので、符尾に接続しない領域はここで候補から除外する。単鉤は更に検出精度を上げるために、テンプレートマッチングを行って、候補領域を限定する。

ニューラルネットワークを用いた候補記号の識別

前節までの手法では、符頭、旗の候補はとりこぼしを少なくするために多めに抽出した。これらの候補から真の記号を識別するために、ニューラルネットワークを用いる方法を説明していこう。

このアプローチでは、まずどのような特徴量をニューラルネットワークに入力するかが重要となる。そこで、音符がどのように描かれるかを考えると、符尾、符頭、旗の間には、例えば、次のような構成規則が存在する。

- 一本の符尾に一つの符頭がつく場合は、符尾の右上端か左下端に符頭がつく。
- 符尾の片方の端に旗がつく場合は、もう一方の端には必ず符頭がつく。

このように、符尾、符頭、旗はある表記規則に従って描かれており、その規則に従った記号の候補を真の記号として抽出すべきである。よって、符尾、符頭、旗の相対的な位置関係の情報を一つ目の特徴量として採用した。そして、二つ目は、候補記号検出時に得られた、その記号であるか否かを示すもっともらしさの量(例えば、標準テンプレートとの類似度)を特徴量とした。以上、二つの特徴量をコード化してニューラルネットワークの入力とする。ネットワークは符頭候補の識別用と旗候補の識別用の二つの構成を考えた。次節で各構成について述べる。

符頭候補識別のための特徴量のコード化

符頭候補の真偽を判定するためのネットワークへの入力を考えると、まず判定したい符頭候補周辺の符尾、符頭、旗候補の存在を調べる必要がある。表記規則を考慮に入れると、図 21 の位置の各記号の存在状況を調べれば十分である。そこで、各位置に番号をつけた。ここで、10 番の符頭(図 21 の灰色で示した符頭)は判定する符頭候補を意味し、11~14 はその符頭に接続する符尾候補、0~9 はその符尾の端点周りと 10 番の符頭周辺の符頭候補、そして 15~17 は 10 番の符頭まわりの旗の候補を意味している。これらの相対位置関係の情報の取得は、言い替えると番号付けされた位置の符尾、符頭、旗の存在を確かめることにより、10 番の符頭候補が真であるかどうかを判定できると仮定したことにはかならない。

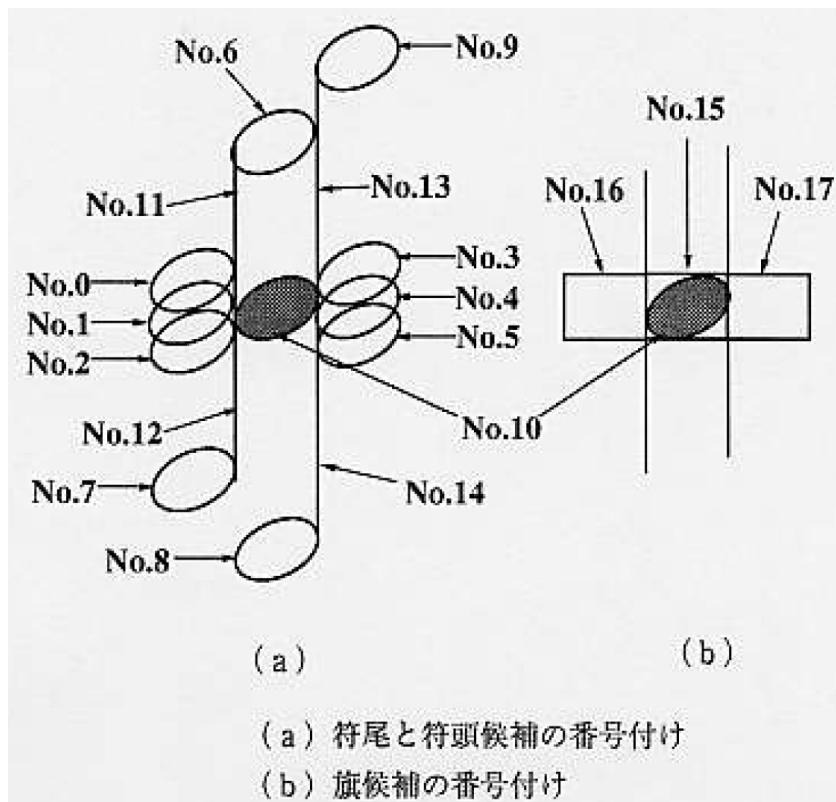


図 21 符頭候補識別のための各記号候補の相対位置の番号付け

次に、各位置の記号のもっともらしさをコード化する。符頭候補のもっともらしさは、その候補がテンプレートマッチングの処理でどれだけ標準テンプレートに適合したかを示す適合度(図 22 の A)による。ここで、眞の符頭となる記号の適合度は 85%以上であることが実験より得られており、そのもっともらしさの量を 2 ビットで表現するため、標準テンプレートへの適合度 85%から 100%の範囲を 3 つの区間に等分割してコード化する。符尾の候補はその線分の長さをもっともらしさの量として採用した。これ

は、一本の符尾に一つの符頭がつく場合、その符尾の長さがおよそ 1 オクターブで描かれるところから、その長さを基準長とした。旗の候補については単鈎の場合は符頭と同様に標準テンプレートとの適合度でコード化を行い、連鈎の場合はその記号の黒画素成分の面積と上輪郭と下輪郭の傾きによってもっともらしさの量のコード化を行う。図 22 に以上のコード化手法をまとめた。これにより、一つの符頭候補(10 番の符頭)の真偽を判定するために 47 ビットのコードを生成し、ネットワークの入力とする。

	Head information										Stem inf.				Flag inf.			
Number	0	1	...	10	11	12	13	14	15	16	17							
Bit code	***	***	...	***	**	**	**	**	**	**	**							
	abc				de				fg									
																	47bits	
a : Head type, [0] for white head and [1] for black one.																		
bc : Head adaptaiton(=A),																		
[00] : does not exist, [01] : 85% <= A < 90%																		
[10] : 90% <= A < 95%, [11] : 95% <= A																		
de : Stem length(=L),																		
[00] : does not exist or L < Len1																		
[01] : Len1 <= L < Len2																		
[10] : Len2 <= L < Len3 (Len3 is almost octave in length)																		
[11] : Len3 <= L																		
fg : Flag adaptation,																		
[00] : does not exist, [01] : low																		
[10] : medium, [11] : high																		

図 22 符頭候補識別のための特徴量コード化ルール

旗候補識別のための特徴量のコード化

旗候補の真偽を判定するためのネットワークに入力する特徴量のコード化は前節の符頭の場合とほぼ同様の手法を用いる。旗に関する表記規則を考慮に入れると、判定したい旗候補の周囲について図 23 の位置の符尾、符頭、旗候補の存在状況を調べれば良い。

図 23 で、9 番で示された灰色の旗が識別をする対象の旗候補を示し、その周辺の記号は、図のように 0～13 に番号付けをする。番号付けした各記号は、図 24 に示すルールに従ってコード化する。ここで、9 番～13 番で番号付けした旗候補は、その位置によって必要とする情報が異なる。これは、該当する位置にその記号を置くことができない場合は、その情報は 9 番の旗候補の判定に影響を与えない、と仮定したのである。例えば、12 番、13 番の位置には単鈎をつけることはできないので、単鈎の存在の情報は無視している。ただし、短い連鈎は長い連鈎が処理過程で切断されて発生する場合があるので、長い連鈎が置ける位置なら、本来短い連鈎が置けなくても、その存在情報を調べることとする。

ここに述べたコード化手法により、一つの旗候補(9 番の旗)を判定するために、合計 43 ビットのコードを生成し、ネットワークの入力とする。

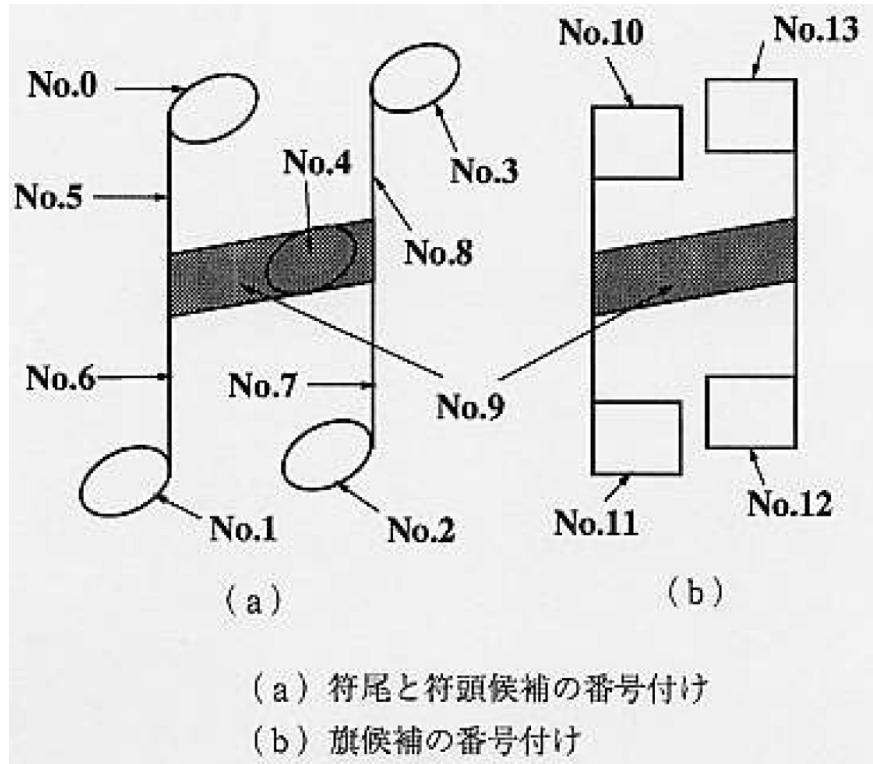


図 23 旗候補識別のための相対位置の番号付け

	Head inf.				Stem inf.				Flag information				
Number	0	...	4	5 6 7 8	9	10	11	12	13				
Bit code	***	***	***	* * * *	*****	*****	*****	****	****				
	abc		d		hijkfg	hjkfg	ijkfg	jkfg	jkfg				

abc : Head information in the same manner as Fig.4.5.
d : Stem information
[0]: does not exist, [1]: exist
h : [1] for upper hook and [0] for otherwise
i : [1] for lower hook and [0] for otherwise
j : [1] for short beam and [0] for otherwise
k : [1] for long beam and [0] for otherwise
fg : Flag adaptation in the same manner as Fig.4.5.

図 24 旗候補識別のための各記号候補の特徴量コード化ルール
(注: 図中の Fig.4.5 は図 22 と置き換えていただきたい)

ニューラルネットワークの学習と候補記号の識別

符頭候補と旗候補の識別をするために、各々に対して一つずつニューラルネットワークを用意する。ニューラルネットワークの構造と学習法は前章でその判定能力が実証されている 3 層型のネットワーク構造とし、誤差逆伝播法によって学習を行うこととする。

最初に符頭候補の識別のためのニューラルネットワークの学習と識別法について述べる。ネットワークの学習のために、まず識別すべき符頭候補(10 番の符頭)の周辺記号を「符頭候補識別のための特徴量のコード化」の手法でコード化し、合計 47 ビットの特徴量をネットワークの入力層(47 ユニットで構成)に入力する。一方、出力層にはその符頭候補に対応する教師信号の値を与える。出力層は一つのユニットから成り、識別すべき符頭候補が真の符頭である場合は教師信号として「1」を与え、偽である場合には「0」を与える。学習は、多くのパターンを対象にして、「使用するニューラルネットワークの構成と学習法」で述べた誤差逆伝搬のアルゴリズムに従い、平均誤差がある値以下になるまで、ネットワークの重みの修正を行う。

次に学習したネットワークの重みを使って、各符頭候補の識別を行う。ここでは、識別すべき符頭候補周辺の特徴を学習過程と同様の手法でコード化して入力層に与え、結果としてネットワークの出力層に出力された値が識別結果を示す。

旗候補の識別をするニューラルネットワークの学習と識別方法も符頭候補のそれと同様の手法で行う。旗候補の場合は、ニューラルネットワークの入力層に図 24 のルールで生成した 43 ビットの特徴量を用いる。

この結果、符頭候補と旗候補の真偽判定が行われる。最終的に、真と判定された符頭または旗に接続する符尾の候補を残し、それを真の符尾とみなすことにより全ての候補記号の識別処理を完了する。

実験結果と評価

実験では、A4版の印刷ピアノ楽譜26枚を用いた。このうち、13枚の楽譜を符頭と旗の候補の判定ネットワークの学習用に使い、残りの13枚をテスト用として用いた。学習用の13枚は初心者向けから上級レベルの楽譜までバラエティに富んでいる。

テスト用の楽譜では、7枚は図25(a)に示すような初心者向けの楽譜であり、残りの6枚は図25(b)に示すような中級もしくは上級レベルの楽譜である。



図 25 テスト用のピアノ楽譜の例(楽譜の一部)

イメージスキャナで楽譜画像を読み取り、五線、小節線検出を行った後、各小節毎に切り出した画像の例を図26に示す。

図27から図31は、この小節画像に対する処理過程の結果を示している。

図27は、五線除去を行った後の画像を示し、図28はその画像に対する垂直方向のプロジェクションを取った結果を示している。プロジェクションの外形において、符尾の存在する可能性のある水平方向の位置では、急峻な山を形成する。そこで、その頂点の検出結果を○印で示した。

図29は図28の○印の水平位置に基づいて符尾候補を検出した結果である。ここで、長方形で囲んだ部分が検出した符尾候補を示す。この結果では、実際の符尾の他に、シャープの縦棒や音部記号部分に余計な垂直線成分が検出されているが、未検出の符尾は発生していない。

図 30 は、検出した符尾候補の周辺探索により、符頭、旗の候補検出をした結果である。この図で、符尾は垂直線分、符頭は楕円(黒塗りは黒符頭、白塗りは白符頭を示す)、旗は四角形で表示した。この時点では、図 29において余計に検出された符尾候補のほとんどが、その周辺に符頭候補が検出できなかつたために除外されているが、調号のシャープ、ト音記号部、黒符頭部に符尾、白符頭、旗の候補が余剰検出されているのがわかる。しかし、ここでも未検出の記号は発生していない。

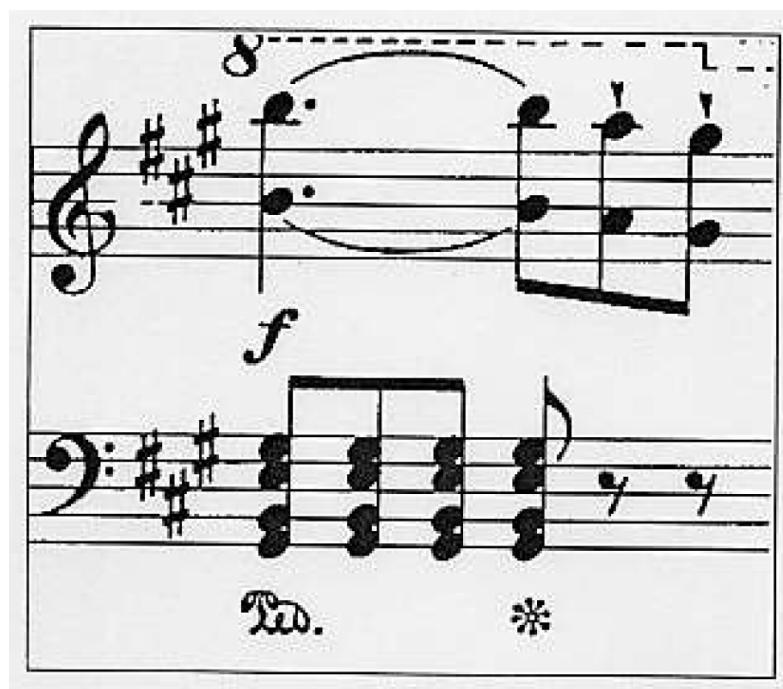


図 26 小節毎に切り出した楽譜の原画像

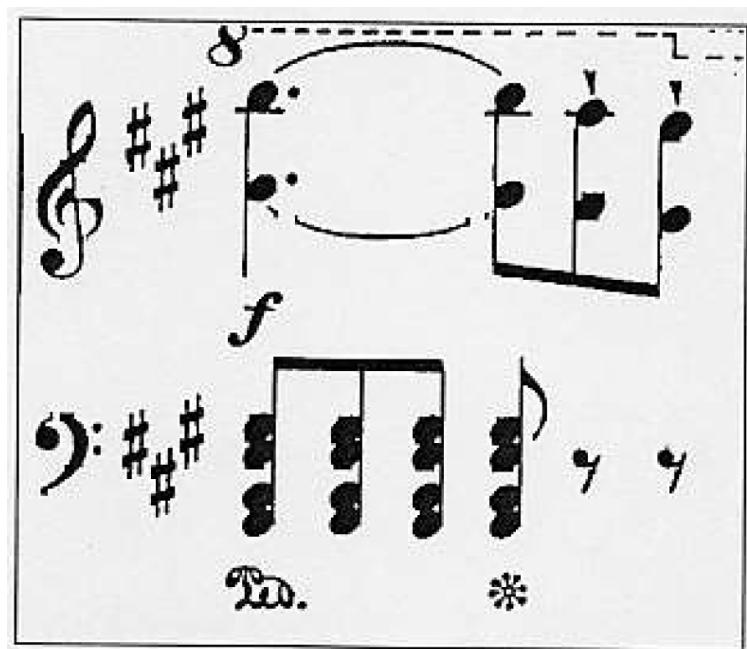


図 27 五線除去後の画像

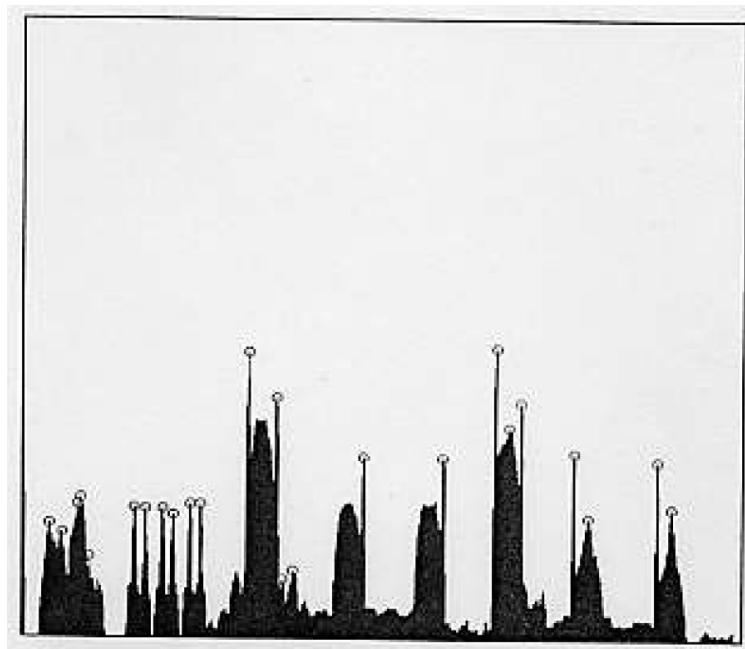


図 28 プロジェクションと符尾候補の水平位置の決定
(○部が符尾候補の水平位置を示す)

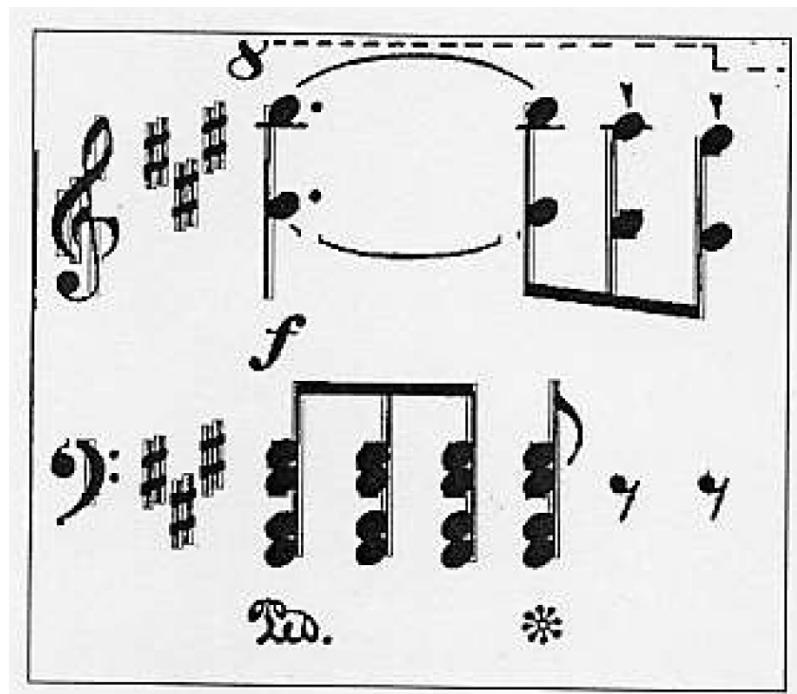


図 29 符尾候補の検出

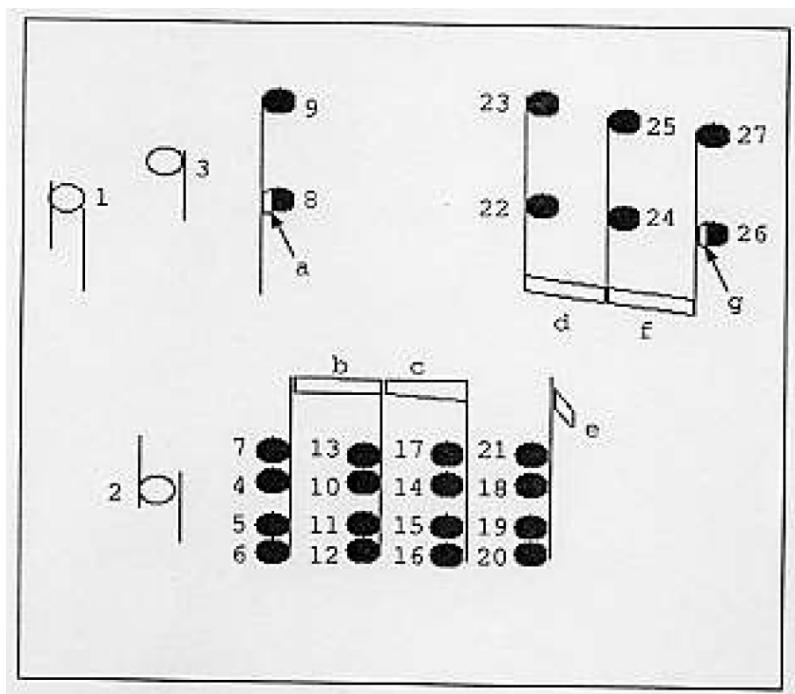


図 30 符頭、旗候補の検出

(各番号とアルファベットは表 6 と表 7 の記号と対応している)

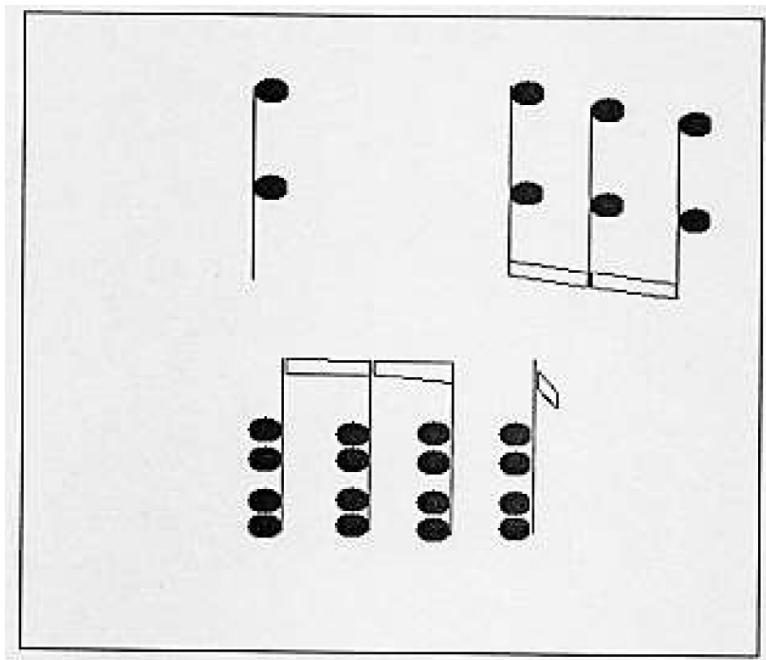


図 31 最終結果

テスト用の楽譜 13 枚に対する候補検出の結果を、表 5 における各項目の上段の数値で示す。ここで、Sample#1 は候補検出が容易であった楽譜サンプルであり、Sample#2 は困難であったサンプルである。これらの楽譜の一部を図 25 (a)と(b) にそれぞれ示した。

表の Average の項目は 1 枚の楽譜に換算した場合の平均値を示し、Total の項目は全 13 枚のテスト用楽譜に対する値を示している。候補検出部での評価対象は、後のニューラルネットワークの識別で処理不可能な未検出記号の数である。そこで、表 5 を見ると、上単鈎と下単鈎に対する未検出率が他の記号に比べて、それぞれ 4.0%(2/50 : 50 記号中 2 個の未検出があった)、2.7%(2/74) であり、多少高い値となつた。これは、単鈎の形状が他の記号の形状に比べて複雑であり、検出しづらいことが原因として挙げられる。しかしながら、その他の記号に対する未検出率は 1%未満であり、候補検出部の性能は十分満足のいくものであると考えられる。未検出を起こしてしまう例としては、図 25 (b)の 3 小節目のように重なりあった連鈎部分で、その記号の検出ができないことがあった。一方、余剰検出の数を見ると、白符頭に対する数が他の記号に比べて大きいのがわかる。これは、図 30 に示すようなト音記号部分や 16 分音符につく連鈎部分に余計な白符頭が多く検出されたことによる。

表 5 音符記号の認識結果

	Stem			Head						Flag								
				Black			White			Upper hook			Lower hook			Beam		
	N	O	E	N	O	E	N	O	E	N	O	E	N	O	E	N	O	E
Sample#1	94	0	11	93	0	0	1	0	15	0	0	0	1	0	0	6	0	5
		0	0		0	0		0	0		0	0		0	0	0	0	0
Rate	100.0%			100.0%						100.0%								
Time	88.6 [s]																	
Sample#2	256	1	23	275	1	9	0	0	52	5	0	0	19	2	0	373	4	8
		1	0		1	3		0	0		0	0		2	0	4	4	0
Rate	99.6%			98.5%						98.5%								
Time	130.5 [s]																	
Average	166	0	15	224	1	2	4	0	15	4	0	0	6	0	0	114	1	15
		1	0		1	1		0	0		0	0		0	0	1	0	0
Time	94.5 [s]																	
Total (13 scores)	2153	4	194	2915	10	22	56	0	195	50	2	0	74	2	2	1481	6	196
		11	5		12	7		2	2		2	0		3	0	8	2	2
Rate	99.3%			99.2%						99.1%								
Time	1228.6 [s]																	

N : 記号の総数、 O : 未検出記号の数、 E : 余剰検出記号の数

* 各項目の上段の数値は候補検出時の結果を示し、下段の太字で示した数値は最終結果を示している。Average と Total の欄はそれぞれ、1枚の楽譜における平均の数値と全13枚のテスト用楽譜に対する数値を示す。

前節で述べたように、抽出した候補の識別は、符頭識別用と旗識別用の2つの3層型ニューラルネットワークによって行う。ここで、各ネットワークの中間層の値は20とした。これは、中間層のユニット数を幾つか変えて実験を行った結果、もっとも良い結果を示した中間層のユニット数である。学習過程では、両ネットワーク共に、重みの初期値 w_{int} を $-0.3 \sim 0.3$ の乱数とし、学習定数 $\eta = 0.75$ 、安定化定数 $\alpha = 0.80$ とした(各パラメータの意味は「使用するニューラルネットワークの構成と学習法」とおりである)。学習用の楽譜13枚から検出した4,274個の符頭候補(その内、真の符頭4,017個)と2,262個の旗候補(その内、真の旗2,075個)を用いて、出力と教師信号の平均誤差がある値以下になるまで重みの学習を行った。その結果、符頭識別ネットワークでは約10分で学習を終え、旗識別ネットでは約8分の学習時間を要した(ワークステーション Sun SPARC Station 10を使用)。

学習によって得られた重みを用いて、テスト用の楽譜13枚の候補識別を行った。

ネットワークの出力層の値は実数であるが、識別は1/0の二値で行いたいので、識別のためにはあるしきい値を設定しなければならない。本実験では、そのしきい値を0.5に設定し、結果の値が0.5より大きい場合には、その候補記号を真の記号とし、0.5以下の場合には偽であると判定させた。

表6と表7に図30の符頭候補と旗候補の特徴量コード化とネットワークの判定の

結果をそれぞれ示す。表 6 の番号は図 30 の符頭候補につけた番号に対応しており、結果の値はニューラルネットワークによって判定された識別結果を示している。値は、「1」が真の符頭であると判定したことを示し、「0」が偽の符頭候補であると判定したことを意味する。この表より、全ての符頭候補が正確に識別できていることがわかる。

同様に、表 7 の記号(アルファベット)は 図 30 の旗候補につけたアルファベットに対応している。この結果より、旗候補についてもニューラルネットワークは正しい判定を行っている。

表 6 図 30 の符頭候補の特徴量コード化とネットワークによる判定結果

番号	コード			結果	正解
	符頭	符尾	旗		
1	000 000 000 000 000 000 000 000 000 000 000 000 000 010	00 01 00 10	00 00 00	0	0
2	000 000 000 000 000 000 000 000 000 000 000 000 000 001	01 00 00 01	00 00 00	0	0
3	000 000 000 000 000 000 000 000 000 000 000 000 000 001	00 00 00 01	00 00 00	0	0
4	000 000 000 000 000 000 000 000 000 000 000 111 000 111	00 00 10 01	00 00 00	1	1
5	000 000 000 000 000 000 000 000 000 000 000 111 000 110	00 00 11 01	00 00 00	1	1
6	000 000 000 000 000 000 000 000 000 000 000 000 000 111	00 00 11 00	00 00 00	1	1
7	000 000 000 000 000 000 000 000 000 000 000 111 000 110	00 00 01 10	00 00 00	1	1
8	000 000 000 000 000 000 000 110 000 000 000 000 111	10 10 00 00	01 00 00	1	1
9	000 000 000 000 000 000 000 000 000 000 000 000 000 110	00 11 00 00	00 00 00	1	1
10	000 000 000 000 000 000 000 000 000 000 000 111 000 111	00 00 10 01	00 00 00	1	1
11	000 000 000 000 000 000 000 000 000 000 000 111 000 110	00 00 11 01	00 00 00	1	1
12	000 000 000 000 000 000 000 000 000 000 000 000 000 111	00 00 11 00	00 00 00	1	1
13	000 000 000 000 000 000 000 000 000 000 000 111 000 110	00 00 01 10	00 00 00	1	1
14	000 000 000 000 000 000 000 000 000 000 000 110 000 110	00 00 10 01	00 00 00	1	1
15	000 000 000 000 000 000 000 000 000 000 000 110 000 101	00 00 11 01	00 00 00	1	1
16	000 000 000 000 000 000 000 000 000 000 000 000 000 110	00 00 11 00	00 00 00	1	1
17	000 000 000 000 000 000 000 000 000 000 000 110 000 110	00 00 01 10	00 00 00	1	1
18	000 000 000 000 000 000 000 000 000 000 000 111 000 111	00 00 10 01	00 00 00	1	1
19	000 000 000 000 000 000 000 000 000 000 000 111 000 110	00 00 11 01	00 00 00	1	1
20	000 000 000 000 000 000 000 000 000 000 000 000 000 111	00 00 11 00	00 00 00	1	1
21	000 000 000 000 000 000 000 000 000 000 000 111 000 110	00 00 01 10	00 00 00	1	1
22	000 000 000 000 000 000 000 110 000 000 000 000 110	10 01 00 00	00 00 00	1	1
23	000 000 000 000 000 000 000 000 000 000 000 000 000 110	00 11 00 00	00 00 00	1	1
24	000 000 000 000 000 000 000 111 000 000 000 000 111	10 01 00 00	00 00 00	1	1
25	000 000 000 000 000 000 000 000 000 000 000 000 000 111	00 11 00 00	00 00 00	1	1
26	000 000 000 000 000 000 000 111 000 000 000 000 111	10 01 00 00	01 00 00	1	1
27	000 000 000 000 000 000 000 000 000 000 000 000 000 111	00 11 00 00	00 00 00	1	1

番号の欄は、図 30 の各符頭候補についての番号に対応している。

結果の欄は、ニューラルネットワークによって判定された識別結果を示し、その符頭を真と判定した場合は「1」、偽であるとした場合は「0」として示してある。

正解の欄は、その符頭が真の符頭である場合は「1」、偽である場合は「0」として示した。

表 7 図 30 の旗候補の特徴量コード化とネットワークによる判定結果

記号	コード			結果	正解
	符頭	符尾	旗		
a	110 000 000 000 111	1 1 0 0	001001 00000 00000 0000 0000	0	0
b	000 111 111 000 000	0 1 1 0	000111 00000 00000 0000 0000	1	1
c	000 111 110 000 000	0 1 1 0	000111 00000 00000 0000 0000	1	1
d	110 000 000 111 000	1 0 0 1	000101 00000 00000 0000 0000	1	1
e	000 111 000 000 000	1 1 0 0	100010 00000 00000 0000 0000	1	1
f	111 000 000 111 000	1 0 0 1	000101 00000 00000 0000 0000	1	1
g	111 000 000 000 111	1 1 0 0	001001 00000 00000 0000 0000	0	0

記号の欄は、図 30 の各旗候補についての記号（アルファベット）に対応している。

結果の欄は、ニューラルネットワークによって判定された識別結果を示し、その旗を真と判定した場合は「1」、偽であるとした場合は「0」として示してある。

正解の欄は、その旗が真の旗である場合は「1」、偽である場合は「0」として示した。

図 31 は上記の判定に基づいた最終結果を示す。ここで、候補検出時に余計に検出された記号が、ニューラルネットワークの識別処理により完全に除去されているのがわかる。

表 8 には、学習用楽譜 13 枚、テスト用楽譜 13 枚の中で検出した全ての符頭と旗候補に対するネットワークの識別結果を示す。これより、符頭と旗候補に対する識別率はテスト用パターンに対しても 99.5% を越え、かなり正確に識別できたことがわかる。

ニューラルネットワークを用いた識別手法の有効性を確かめるために、従来法との比較を行った。ここで比較対象に用いたのは、音符の構成規則を人手によって if-then ルール化し、パラメータ調整を行う手法である。本章の手法で検出したと同様の符頭、旗候補に対して、上記手法を適用した結果を表 8 の Traditional の欄に示す。両者の識別率を比較すると、旗候補の識別に関しては同等であること、符頭候補識別に関しては従来法よりニューラルネットワークによる手法の方が優れていることがわかる。

表 8 ニューラルネットワークを用いた符頭と旗の識別結果

Pattern	Decision Method	Head Candidates	Flag Candidates
Training	This work	(4,272/4,274) 99.95%	(2,261/2,262) 99.96%
Test	This work	(3,165/3,178) 99.59%	(1,788/1,793) 99.72%
	Traditional	(3,137/3,178) 98.71%	(1,785/1,793) 99.55%

(a/b) a:正しい判定の総数、b:候補の総数

Traditional: if-then ルールを用いた手法

ニューラルネットワークの識別に誤りが生じたパターンを解析してみると、複数の音符が連続で継れて描かれている部分で、全体の音符の構成としては誤っているのだが、それらを規則に合致した記号として判定してしまう場合があった。これは、コード化方法が、識別する符頭や旗の周辺の局所的な部分のみに着目しているために起こる誤りであり、より大域的な視点で音符の構成規則に従わせる必要もあったと考えられる。よって、さらに判定精度を上げるには、より大きな範囲での各記号の相対位置関係の情報が必要であると思われる。

この手法の特徴は、音楽記号の形状を単に認識するだけでなく、それらの空間的な位置関係を認識しているところにある。しかしながら、ネットワークは本当に音符の構成規則を学習できたのだろうかという疑問が生じる。

そこで、学習済みのネットワークに人工的に生成した音符のパターンを入力して、その結果を調査したところ、ネットワークは完全には規則を学習していないことが判明した。このような場合までカバーしなくてはならないならば、従来手法で用いられたif-then ルールによる判定の方がより正確に行えるかもしれない。

しかし、これらのネットワークの不備は、学習パターン中に全ての音符の構成パターンの組み合わせが出現しないことによるものと考えられ、より正確にルールを学習させるためには、人工的に生成した多くの学習パターンを使った集中学習が有効であると思われる。しかし、実用面から考えると、ネットワークの判定は、if-then ルールによる判定と同等あるいはそれ以上の能力を持つことが実証され、十分に適用可能である。これは、学習時に実際の楽譜から抽出した多数のパターンを用いていることから、人間が想定したルールでは補えなかった部分も学習している可能性があり、それが良

好な結果を導き出しているとも言える。

一方、実際の楽譜に対する適用を考えると、音符の構成規則の拡張のしやすさも重要な要素となる。なぜなら、例えば図 32 (b)のように、1つの符尾に黒符頭と白符頭が同時につくような、一般的な音符の構成規則に従わないで描かれている楽譜も存在するため、それらに適用できるように、簡単にルールの拡張ができなければならないのである。そこで、上記のルールを音符検出部に付加することを実験的に行った。まず if-then ルールを使用した従来手法の場合、そのルールの書き換えと多くの実験を通じたパラメータ調整を行った結果、約 3 時間の時間を必要とした。これに対して、ニューラルネットワークによる手法では、いくつかのサンプルを実際の画像から取り出し、重みの再学習をさせるのに、わずか 30 分程度しかかからなかった。このような実際の楽譜に対するルール修正の必要性は大きく、それに費される時間の削減は現実問題として大きな利点となる。ニューラルネットワークによる手法は、このようなルールの拡張性の面からみても 従来手法より優れていると言える。

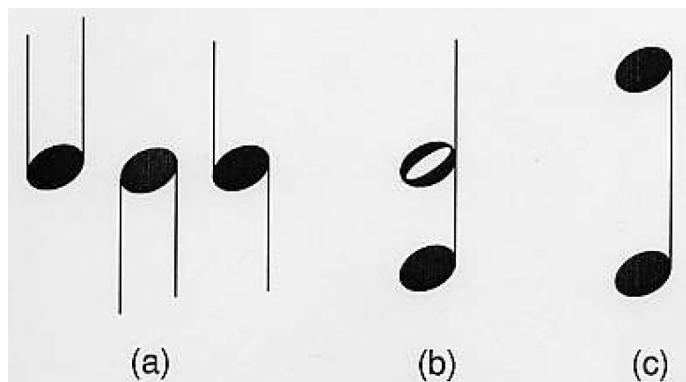


図 32 存在しない音符の構成

表 5 の各項目の下段に最終的な記号の抽出結果を示す。ここで、認識率(Rate)の値は、全記号数に対する修正が必要な記号数(余剰検出記号数と未検出記号数の和)の割合を 100%から差し引いて求めた値である。この結果より、各記号の認識率は難易度の異なる多くの楽譜について 99%以上の値を示し、かなり正確に検出が行えることがわかる。

表 5 には ワークステーション(SUN SPARC Station 10)を用いた場合の処理時間の結果も合わせて示した。ここでの時間は、符尾、符頭および旗の検出にかかる時間のみを示しており、実際には、この時間に画像入力約 40 秒と前処理部約 10 秒の時間を加える必要がある。結果より、図 25 (a)Sample#1 のような初級者用の楽譜に対しては、音符検出に 60 秒から 100 秒程度の時間がかかり、図 25 (b)Sample#2 のような中上級者用については、130 秒程度、平均では 90 秒程度の時間を要した。処理時間は、おおよそその楽譜に含まれる符頭と旗の数に比例している。これは、候補記号検出時のテンプレートマッチングの実行回数と、ニューラルネットワークの計算回

数が、符頭と旗の数に比例して増えることに起因していると考えられる。本手法の有効性を確かめるために、手動入力との比較を行った。その結果、マウスを使用して実験に用いた 13 枚のテスト用楽譜に現れる音符記号の位置情報を得るのに約 14,500 秒の時間を費した。これに対して本手法では、1228.6 秒で同様の位置情報を得ることができる。修正確率が 1%以下と低いため、修正時間もそれほどかかりない。20 分程度の修正時間がかかるとしても、人手による入力に比べると、5 倍以上高速に情報取得ができる。

以上をまとめると、ニューラルネットワークを用いた候補記号の識別は従来の if-then ルールで構築された判定部分を十分に置き換えることが可能であると言える。

結言

本章では、音符を構成する符頭、符尾、旗の位置を高速かつ正確に抽出するためには、符尾の位置に基づいた周辺探索による符頭、旗候補の検出法とニューラルネットワークによる候補識別法について述べた。

これらの手法をネットワークが未学習の難易度の異なる 13 枚の印刷ピアノ楽譜に適用した結果、符尾、符頭、旗のそれぞれに対して、99.3%、99.2%、99.1%の高い認識率を得ることができた。ネットワークは音符の構成規則を完全に学習したわけではないが、実際の楽譜中の符頭、旗を識別するには、十分な能力を持つことが実証された。処理時間に関しては、A4 版のピアノ楽譜に対して、ワークステーションで約 60 秒から 130 秒の時間がかかった。これより、手動で入力する手法に比べ、5 倍以上高速に音符の位置情報を取得することができる。

また、ニューラルネットワークは、従来法での音符の構成規則に基づく複雑な if-then ルールの構築や手間のかかるパラメータの調整をすることなしに、その処理の代替ができるることを識別率、拡張性、処理時間の側面から検証して示した。

適用例：3層型ニューラルネットワークを用いた数字認識

ここでは、Google 社が開発した機械学習用のプラットフォーム TensorFlow⁵を用いて、数字画像認識を行うプログラムを作成する。作成には、Web 上で Python を記述・実行することができる Google Colaboratory⁶を用いる。Web の教材ページに、サンプルプログラムをノートブック形式で掲載するので、そのプログラムを実際に実行しながら、動作確認をすると良い。なお、プログラムを完全に理解するには、Python や Numpy (Python 用の数値計算ライブラリ) の使い方を知る必要があるので、必要に応じて学習されたい。

画像データと対応する教師データの読み込み

最初に TensorFlow と Numpy のモジュールをインポートし、数字の画像データベースである MNIST を読み込む。

```
import tensorflow as tf #TensorFlow モジュールのインポート
import numpy as np #NumPy モジュールのインポート
mnist = tf.keras.datasets.mnist #数字画像データベースの参照変数の設定
(x_train, y_train), (x_test, y_test) = mnist.load_data() #画像の読み込み
print(type(x_train)) #x_train のデータ型の確認
print(x_train.dtype) #x_train(多次元配列) の各要素のデータ型の確認
print(x_train.ndim) #x_train 配列の次元数の確認
print(x_train.shape) #x_train 配列の大きさの確認
```

ここで、x_train, y_train, x_test, y_test のデータ型は NumPy の多次元配列 (numpy.ndarray)となる。x_train は学習用データで数字画像を表現している 3 次元配列である。各文字画像は 28×28 画素であり、各画素は 0 から 255 の間の整数(uint8)で表現され(グレースケール画像)、このような文字画像が 60,000 枚、含まれている。一方、y_train は学習用データの教師データを示している 1 次元配列である。各要素は 0 から 9 の間の整数値(uint8)で表現され、全体で 60,000 個が含まれている。x_test, y_test はテスト用データの文字画像とその教師データをそれぞれ示しており、データ構造は、x_train と y_train と同様である。ただし、文字画像と教師データの個数はそれぞれ 10,000 個となっている。(上記プログラムのように多次元配列の各属性値は print 文で確認可能)。

⁵ <https://www.tensorflow.org/tutorials?hl=ja>

⁶ <https://colab.research.google.com/notebooks/intro.ipynb?hl=ja>

```
x_train, x_test = x_train / 255.0, x_test / 255.0
```

次に、上記の計算をすることにより、x_train と x_test の各画素値を 0.0 から 1.0 の浮動小数点数(float64)に変換する。このように、ニューラルネットワークへの入力値は 0 から 1 に正規化することが多い。以下、x_train, y_train の学習データを図示すると次のような構成になっている。なお、x_train[i] (x_train の i 番目のデータ) の画像に対する教師データ(正解データ)は y_train[i] に格納されている。

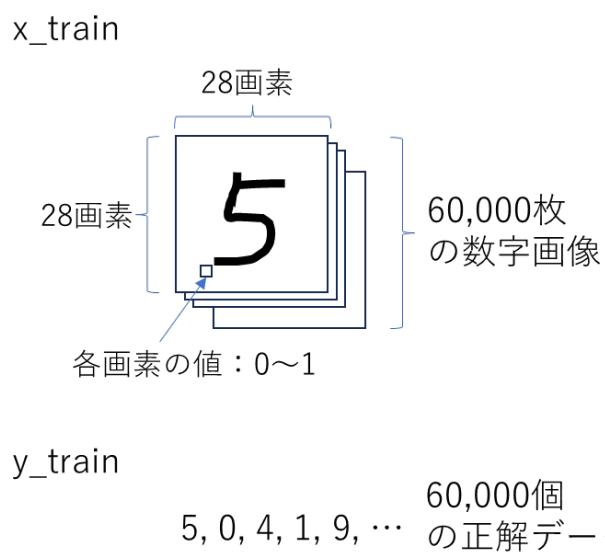


図 33 学習データの構成

ニューラルネットワークの構成と学習法の設定

次に、ニューラルネットワークの構成と学習法を設定する。ここでは、次のように設定してみる。

- 入力層へは、28 画素 × 28 画素を 1 列に並べた 784 次元のベクトル(各画素は 0～1 の値)を入力する。よって、入力層のユニット数は 784 個となる。
 - 中間層を 64 個とし、活性化関数はシグモイド関数を採用。
 - 出力層は、数字のクラス数 10 個に合わせて、ユニット数は 10 個とし、ソフトマックス関数で各クラスの確率値を出力。
 - 最適化アルゴリズムとして SGD(確率的勾配降下法)を採用。
 - 誤差関数としてクラス出力に関するクロスエントロピーを採用
 - 学習途中の評価のための尺度として識別精度(accuracy)を指定
- 以上を実現するために次のように記述する。

```

model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(64, activation='sigmoid'),
    tf.keras.layers.Dense(10)
])
model.compile(optimizer='sgd',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(
                  from_logits=True),
              metrics=['accuracy'])

```

上記の構造を図示すると次のとおりである。

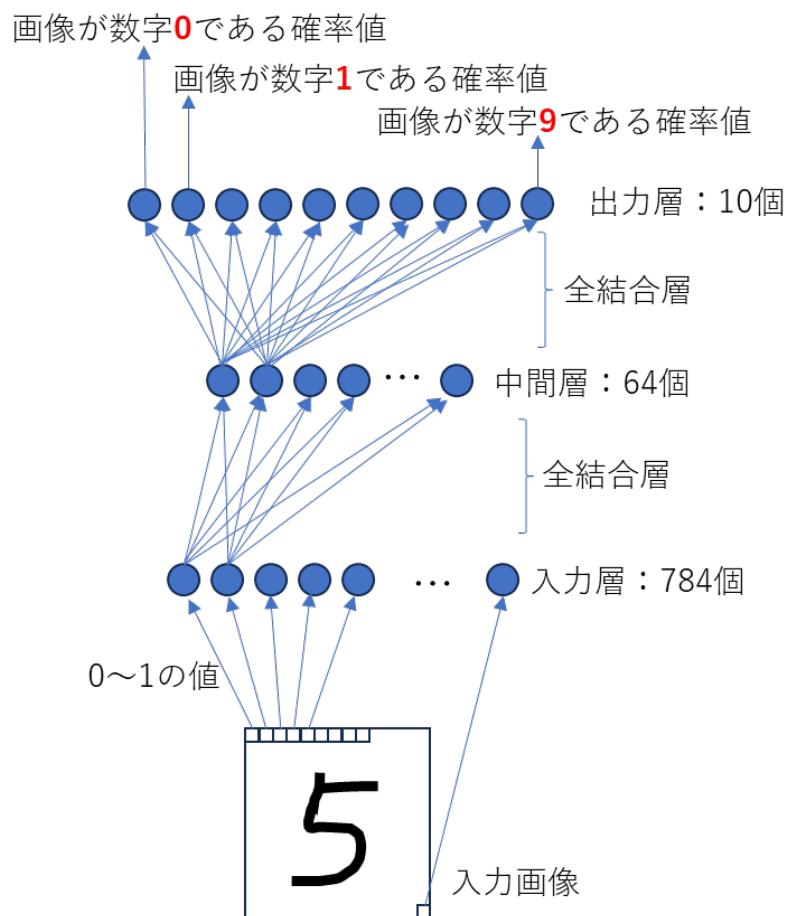


図 34 ニューラルネットワークの構成

ニューラルネットワークの学習と評価

次に、学習用データを用いて構成したニューラルネットワークを使って学習を行う。ここでは、学習用データとして、先ほど読み込んだ `x_train` と `y_train` のデータを用いる。`x_train` は 60,000 枚の数字画像(グレースケール画像)であり、`y_train` はそれらの各画像に対応する正解データ(0 から 9 の数字)である。学習を実施するには、下記プログラムを実行する。

```
model.fit(x_train, y_train, epochs=5)
```

ここで、エポック(epoch)とは、数字画像全体(60,000 枚)を一通り学習すると 1 回とカウントする数値なので、この場合は 5 回繰り返し学習していることになる。学習結果は次のとおりである。なお、学習時のネットワークの重みの初期値が乱数によって設定されるため、学習ごとに微妙に結果の値が異なる点を留意されたい。

```
Epoch 1/5
1875/1875 [=====] - 5s 2ms/step - loss: 1.5168 - accuracy: 0.6697
Epoch 2/5
1875/1875 [=====] - 5s 3ms/step - loss: 0.7756 - accuracy: 0.8357
Epoch 3/5
1875/1875 [=====] - 4s 2ms/step - loss: 0.5684 - accuracy: 0.8687
Epoch 4/5
1875/1875 [=====] - 4s 2ms/step - loss: 0.4774 - accuracy: 0.8822
Epoch 5/5
1875/1875 [=====] - 5s 3ms/step - loss: 0.4267 - accuracy: 0.8903
```

これより、学習データについては、89.03%の認識率が得られていることがわかる。次に、テストデータについての認識率を評価してみよう。

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print(test_acc)
```

```
313/313 [=====] - 1s 2ms/step - loss: 0.3898 - accuracy: 0.9004
0.9003999829292297
```

この結果から、テストデータの認識率は 90.04%となっていることがわかる。

ニューラルネットワークの構成・学習法の変更による効果

次に、学習に用いるニューラルネットワークの構造や各種パラメータを変更して、さ

らにテストデータの認識率を向上させることができるかを検証してみよう。

最初に中間層の活性化関数⁷をシグモイド関数から ReLU に変更してみる。プログラムで、sigmoid の部分を relu に変更する(下記赤字部分)。

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10)
])
```

他のプログラムは前回と同様である。このモデルを用いた結果、学習データに対する認識率は 93.04%、テストデータに対する認識率は 93.36% に向上する。

次に、中間層のユニット数を 64 個から 128 個に増やしてみる。プログラムで、下記赤字部分を修正する。

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])
```

結果、学習データとテストデータに対する認識率は、それぞれ 93.47%、93.81% と若干ではあるが向上がみられた。その後、さらにユニット数を 256 個とする実験も行ったが、テストデータに対する認識率は、ほとんど変わらなかつたため、ここでは 128 個に固定する。

次に、最適化アルゴリズム⁸を SGD から Adam に変更してみる。プログラムでは、下記赤字部分を修正する。

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(
                  from_logits=True),
              metrics=['accuracy'])
```

⁷ 利用可能な活性化関数は次の URL を参照:<https://keras.io/ja/activations/>

⁸ 利用可能な最適化アルゴリズムは次の URL を参照:
<https://keras.io/ja/optimizers/>

この結果、学習データとテストデータに対する認識率は、それぞれ 98.62%、97.74%と向上した(以後、Adam に固定)。

次に、中間層の数を 1 つ増やして、全体で 4 層構造のニューラルネットワークにしてみる。ここでは、中間層の第 1 層は 128 個のままでし、第 2 層として 64 個のものを追加する。使用する活性化関数はともに ReLU である。

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10)
])
```

この結果、学習データとテストデータに対する認識率は、それぞれ 98.68%、97.56%となった。これより、学習データについては若干向上したが、テストデータについては若干下がってしまった。この傾向は、学習データに過剰に適応した過学習(overfitting)の状態を示しているため、以後は、中間層を 1 層(64 個)として固定することにする。

次に、学習回数(エポック)を 5 から 10 に変更してみよう。

```
model.fit(x_train, y_train, epochs=10)
```

結果、学習データとテストデータに対する認識率は、それぞれ 99.51%、97.78%となり、ここでも、過学習傾向となつたため、エポック数は 5 程度で十分と考えられる。

最後に、過学習の傾向を抑制するために、ドロップアウトの機能を導入してみよう。ここでは、出力層以外のユニットのうち、2 割をランダムに無効化して学習することにする。

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dropout(0.2),
    tf.keras.layers.Dense(10)
])
```

これにより、学習データとテストデータに対する認識率は、それぞれ 97.69%、97.85%と

なり、学習データについては認識率は落ちるが(過学習傾向の抑制)、テストデータについての認識率は向上していることがわかる。同じ条件で、ドロップアウトの確率を4割にした場合は、学習・テストデータ両方で、若干、認識率が下がってしまった。これより、あまり多くのユニットを削減してしまうと、認識率の低下につながってしまうため、適切な値を見つける必要がある。

以上、今回の構造やパラメータ変更によって得られた結果を下表にまとめる。このように、ニューラルネットワークの学習を行う際には、様々なパラメータを試して、最適なモデルを探る必要がある。

表 9 各種パラメータを変更した場合の認識率

中間層1		中間層2		最適化 Alg.	Dropout	エポック	認識率[%]	
ユニット数	活性化	ユニット数	活性化				学習	テスト
64	sigmoid			sgd		5	89.03	90.04
64	relu			sgd		5	93.04	93.36
128	relu			sgd		5	93.47	93.81
128	relu			adam		5	98.62	97.74
128	relu	64	relu	adam		5	98.68	97.56
128	relu			adam		10	99.51	97.78
128	relu			adam	0.2	5	97.69	97.85
128	relu			adam	0.4	5	96.34	97.43

畠み込みニューラルネットワークによる学習とクラス分類

ここでは、画像認識分野でよく使われている畠み込みニューラルネットワーク(CNN: Convolutional Neural Network)について、その構造と学習・クラス分類の方法について概略を説明する。

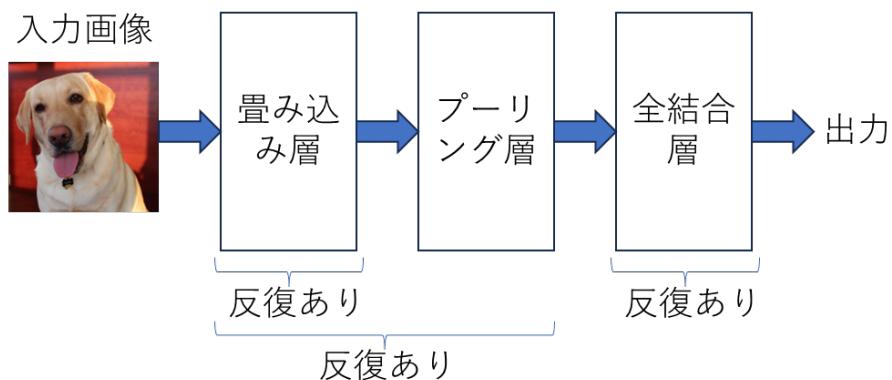


図 35 畠み込みニューラルネットワークの構造

図 35 に一般的な畠み込みニューラルネットワークの構造を示す。ここで、画像をクラス分類するネットワークの構築を想定してみよう(例えば、犬と猫の画像を入力したら、犬の画像の確率と猫の画像の確率を出力してくれるようなネットワーク)。まず、入力された画像の情報は、畠み込み層と呼ばれる部分に入力され、その出力がプーリング層と呼ばれる部分に入力される。ここで、畠み込み層は複数回反復して適用する場合があり、畠み込み層とプーリング層のセットも複数回反復する場合がある。プーリング層からの出力は全結合層に入力され(この部分も通常複数の層で構成される)、その結果として画像をクラス分けした結果を出力する。ここで、全結合層の部分は、「3 層型ニューラルネットワークによる学習と識別」で述べたような、隣合う層の全てのユニットが相互に全結合されている構造となっている。

以下、各部の構造や働きについて述べる。

畠み込み層

畠み込み層は、画像の局所的な特徴を抽出する部分と考えることができる。これを実現するために、畠み込み層では、フィルタと呼ばれる行列を利用する。

図 36 に一般的な畠み込み層の構造を示す。まず、入力された画像がカラー画像であると想定すると R、G、B で構成される 3 枚の画像と見ることができるので、これら 3 枚を入力画像と考えよう(このような入力枚数をチャンネルと呼ぶ。ここでは、チャンネル数が 3 となる)。

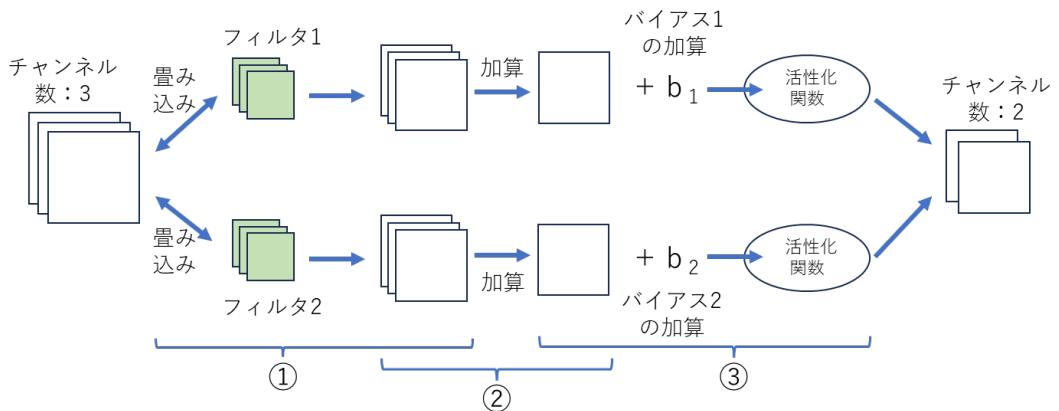


図 36 畳み込み層の構造(文献[75]p.240 上図を参考に作成)

次に図 36①の部分に示すように、各チャンネル画像についてフィルタを畳み込んでいる。この畳み込みの処理は図 37 のように行われる。まず、フィルタが図 37 の左側の 3×3 の行列で表現されている場合、これを画像の左上に重ね合わせ(クリーム色のハッチング部分)、同じ位置の要素同士を掛け算してから全体の総和を計算して一つの値を求める操作を行う(左上の場合は 3 の値が得られる)。この操作を図 38 のように画像上の左上から右下に向かってフィルタを重ね合わせて計算した結果が図 37 の右側の 4×4 の行列である。このような操作を画像へのフィルタの畳み込み処理と呼ぶ。

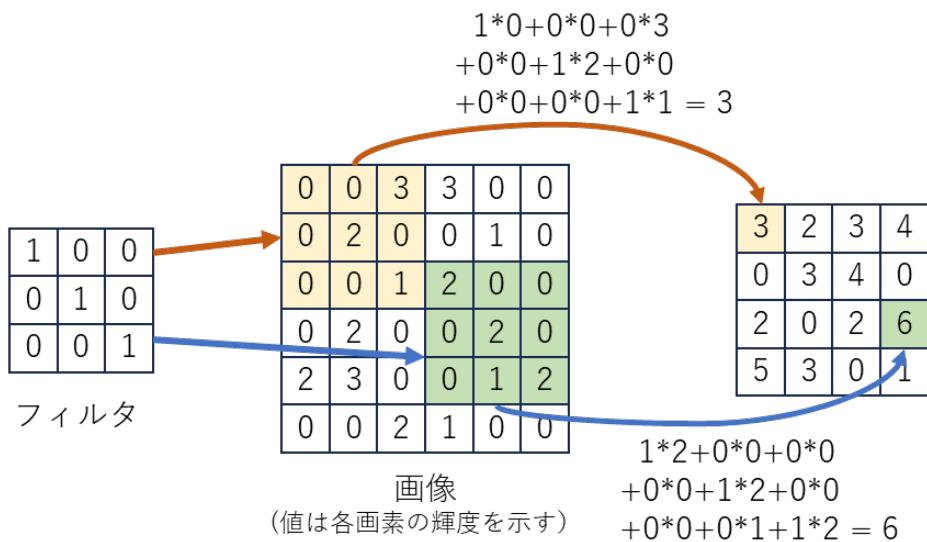


図 37 画像へのフィルタの畳み込み(文献[75]p.238 上図を参考に作成)

0	0	3	3	0	0
0	2	0	0	1	0
0	0	1	2	0	0
0	2	0	0	2	0
2	3	0	0	1	2
0	0	2	1	0	0

0	0	3	3	0	0
0	2	0	0	1	0
0	0	1	2	0	0
0	2	0	0	2	0
2	3	0	0	1	2
0	0	2	1	0	0

0	0	3	3	0	0
0	2	0	0	1	0
0	0	1	2	0	0
0	2	0	0	2	0
2	3	0	0	1	2
0	0	2	1	0	0

0	0	3	3	0	0
0	2	0	0	1	0
0	0	1	2	0	0
0	2	0	0	2	0
2	3	0	0	1	2
0	0	2	1	0	0

0	0	3	3	0	0
0	2	0	0	1	0
0	0	1	2	0	0
0	2	0	0	2	0
2	3	0	0	1	2
0	0	2	1	0	0

図 38 画像へのフィルタの重ね合わせ順序(ストライドが 1 の場合)

ここで、フィルタの畳み込みが何を意味しているのか考えてみよう。図 37 のフィルタのように左斜めに 1 の要素、他の要素が 0 のようなフィルタの場合、画像に重ね合わせて要素の積和計算を行うと、その重ね合わせた画像部分に左斜め成分がある場合は、大きな値を取ることになる。このようにフィルタは、重ね合わせた局所的な画像の特徴を検出していることに相当する。もちろん、異なるフィルタを適用した場合は、別の特徴を抽出することができる。

なお、画像にフィルタを畳み込んだ結果の行列は元の画像より小さくなる。図 37 の例の場合、元の画像が 6×6 行列だったが、 3×3 のフィルタを畳み込んだ結果、 4×4 行列になっている。このように、フィルタの畳み込みや後述するプーリングを繰り返すと画像が小さくなってしまうため、元画像を取り囲むように 0 の値を付加してから、フィルタを畳み込み、画像の縮小を防ぐ場合がある。このような処理をパディングと呼ぶ(0 の値を付加する方法をゼロパディングと呼ぶ)。畳み込みの際、パディングは元画像の端の画素の影響が減少してしまうという欠点を改善する効果もある。

また、図 38 では、フィルタを画像上で 1 画素ずつ移動しながら走査していたが、N 画素($N > 1$)ずつ移動して走査する場合もある。ただし、移動幅が大きいと画像の局所特徴を取り漏らしてしまう恐れがあるため、画像認識では、1 画素ずつ移動させる場合が多い。なお、この移動幅のことをストライドと呼んでいる。

それでは次に図 36②の部分について説明する。上記の方法で RGB の各画像についてフィルタ 1(各チャンネル用に 3 種類ある)とフィルタ 2(3 種類)をそれぞれ畳み込んだ場合、フィルタ 1 の結果として 3 つの行列、フィルタ 2 の結果として 3 つの行列がそれぞれ得られる。ここで、それぞれの 3 つの画像特微量(行列)を統合するため、3 つの行列の要素ごとに総和を求め、1 つの行列にまとめる。

次に図 36③に示すように、得られた行列の各要素にバイアス値(図の b_1 と b_2)を加算した後、活性化関数を適用し、最終的な要素値を決定する。この結果、フィルタ数と同じ数の行列(図の場合、行列の数は 2)が得られ、各行列の大きさは畳み込み後の行列の大きさと一致したものが得られる。

プーリング層

畳み込み層で得られたチャンネル数分の行列に対して、通常はプーリングと呼ばれる処理を適用する。プーリングは、画像特徴の局所的な位置ずれを許容し、行列のサイズを小さくする効果がある。

それでは、プーリングの具体的な処理手順を見ていこう。プーリングには、局所領域の平均値を求めるアベレージプーリングと最大値を求める MAX プーリングがあるが、ここでは CNN でよく使われている MAX プーリングについて説明する。

3, 2, 0, 3 のうちの最大値は 3

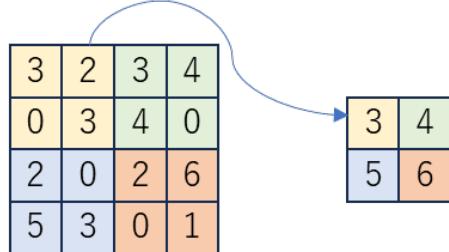


図 39 MAX プーリングの例

畠み込み層の結果、図 39 左の行列が得られた場合、これに 2×2 の領域で MAX プーリングを適用した結果が右の行列である(ストライドは 2)。このように、図のハッチングされた 2×2 の小領域を見て、その中の最大値を抽出して、新たな行列を生成する。

プーリングは小領域の代表値を計算しているため、前述のように、画像内で特徴を示す部分の位置が多少変動したとしても、その位置ずれを吸収できる効果がある。また、図 39 のように、プーリングを適用すると適用前の行列より結果の行列サイズが小さくなる効果があり、計算時間の削減にも効力を発揮する。

全結合層と出力

図 35 に示すように、プーリング層の結果は、最終的に全結合層への入力となり、全結合層の出力が最終結果となる。ここでは、図 40 を使って、その様子を説明する。まずチャンネル数分あるプーリングの結果(図ではチャンネル数が 2)を 1 列の数値列(ベクトル)に変換し、この数値を全結合層の入力層の入力とする。前述のように、全結合層は、「3 層型ニューラルネットワークによる学習と識別」で説明した構成と同じであり(つまり隣り合う層のユニット間が全て結合されており、そこに重みが付加されている)、各層の出力の計算方法も同じである。ただし、最終結果として、画像のクラス分類をする場合(例えば、画像が犬のクラスか、猫のクラスかを分類するような場合)、各クラスの画像である確率を出力させたい。このような場合は、全結合層の出力層の値にソフトマックス関数と呼ばれる活性化関数を適用することが多い。ソフトマックス関数 $f(x)$ は、識別クラスの数が n (出力ユニットの数が n) だった場合、ある出力ユニットの値 x について、下記の値を返す。

$$f(x) = \frac{\exp(x)}{\sum_{i=1}^n \exp(x_i)}$$

ここで、 x_i は出力ユニットの*i*番目の値を示す。これにより、出力ユニットの各値を0から1の値に正規化され、かつ出力ユニット全体の総和を1とすることができます。よって、各ユニットの出力値を各クラス画像となる確率値とみることができる(値1に近づくほど、そのクラス画像である確率が高いとみる)。

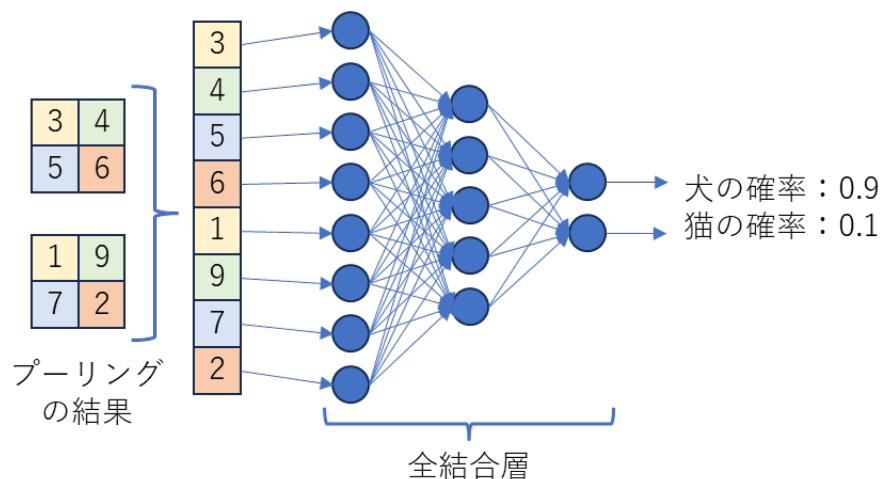


図 40 全結合層とその出力

学習と識別

ネットワークの学習では、上記の手法で得られた出力値について、教師信号を与えて誤差を計算し、ネットワークの各種パラメータの値を更新する。

それでは、まず教師信号について考えてみよう。教師信号は、入力画像に該当するユニットの確率値を1として、他のユニット値を0に設定すれば良い。図40の例では、犬の画像を入力した場合は、最終結果を示す出力ユニットに教師信号として「1, 0」(上のユニットに1、下のユニットに0)を与え、猫の場合は、「0, 1」を与えることになる。

次に出力値と教師信号の誤差について説明する。画像のクラス分類を行う場合は、誤差としてクロスエントロピー誤差を採用する場合が多い。クロスエントロピー誤差は、2つの分布の差(この場合、ネットワークが計算して出力する値の分布と教師信号が与える値の分布との差)を計算することができる。いま、*k*番目の出力ユニットの出力値を y_k 、対応する教師信号を \hat{y}_k とした場合、クロスエントロピー誤差は、下記の値で計算できる。

$$-\sum_k \hat{y}_k \log(y_k)$$

学習では、この誤差の値を最小にする方向に、畳み込み層のフィルタの値・バイアスの値、全結合層の重みの値を誤差逆伝播法によって修正する。そして、大量の教師付き画像データで学習を行ったネットワークは、画像を入力すると、その画像がどのクラスにどれだけの確率で属しているかを出力するようになる。

適用例:畳み込みニューラルネットワーク(CNN)を用いた数字画像認識

前述の適用例では、単純な3層型ニューラルネットワークを用いて数字画像認識を行ってみたが、ここでは、同様の画像データについて、畳み込みニューラルネットワークを用いた学習と認識を行ってみよう(本プログラムは、TensorFlowのチュートリアルページ⁹の内容を参考としている)。

各種モジュールの読み込みと数字画像データの読み込み

```
1 import tensorflow as tf
2 from keras import datasets, layers, models
3 (train_images, train_labels), (test_images, test_labels) = datasets.mnist.load_data()
4 train_images = train_images.reshape((60000, 28, 28, 1))
5 test_images = test_images.reshape((10000, 28, 28, 1))
6 train_images, test_images = train_images / 255.0, test_images / 255.0
```

1行目では、TensorFlowモジュールをインポートし、2行目では、Kerasモジュール内の各種関数を直接呼び出せるように設定をしている。3行目は、使用する数字画像データベース(MNIST)のデータを読み込んでいる。ここで、train_imagesは学習用のデータ、train_labelsはその教師データ(対応する画像のクラス、0から9の整数で表現)、test_imagesは評価用のデータ、test_labelsはその教師データをそれぞれ示している。4行目は行列形状を設定しており、train_imagesは画像枚数が60,000枚、各画像は28×28画素、各画素はグレースケールであることを意味している。5行目は、test_imagesの形状であり、画像枚数が10,000枚、各画像は28×28画素、各画素はグレースケールである。そして、最後の6行目では、各画像の各画素が0から255のグレースケールとなっているので、この値を0から1に正規化している。

⁹ https://www.tensorflow.org/tutorials/images/intro_to_cnns?hl=ja

CNN の構成

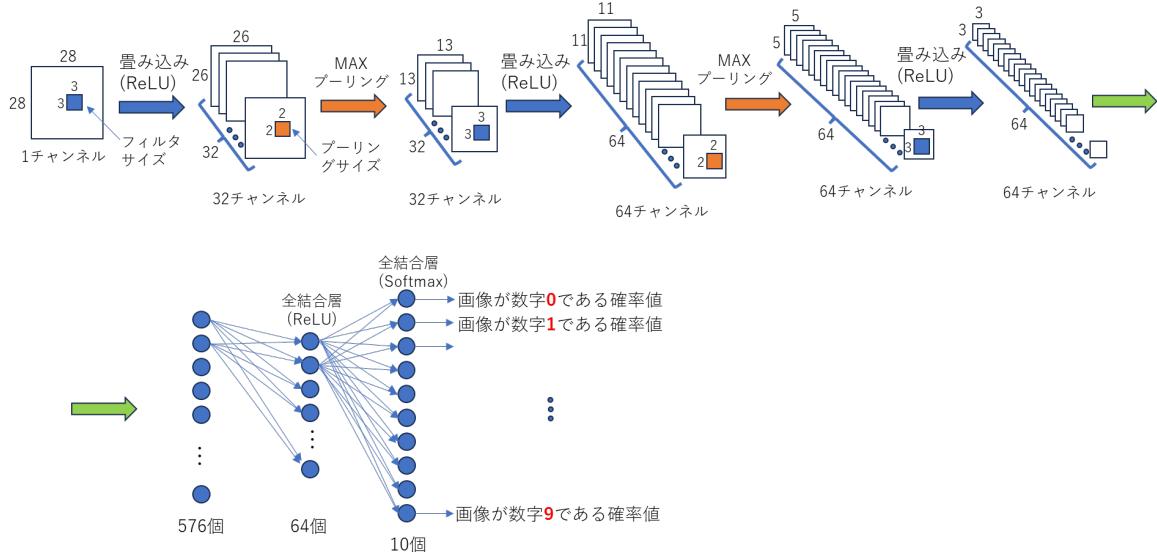


図 41 CNN の構造

```

1 model = models.Sequential()
2 model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28,
1)))
3 model.add(layers.MaxPooling2D((2, 2)))
4 model.add(layers.Conv2D(64, (3, 3), activation='relu'))
5 model.add(layers.MaxPooling2D((2, 2)))
6 model.add(layers.Conv2D(64, (3, 3), activation='relu'))
7 model.add(layers.Flatten())
8 model.add(layers.Dense(64, activation='relu'))
9 model.add(layers.Dense(10, activation='softmax'))
10 model.compile(optimizer='adam',
                 loss='sparse_categorical_crossentropy',
                 metrics=['accuracy'])

```

図 41 に、数字画像認識に使用する畳み込みニューラルネットワークの構造を示し、その下に、この構造を実現するためのプログラムを示す。以下、プログラム行と図を対照しながら、各部の説明をする。

プログラムの `model.add` メソッドは、ネットワークに各層を追加している。まず、2行目では、`layers.Conv2D` を使って、畳み込み層を追加している。32 は、フィルタ数を示し、(3, 3)はフィルタのサイズ、`activation` は活性化関数の指定(ここでは、ReLU を指定)、`input_shape` は入力画像のサイズが 28×28 画素でグレースケール画像であるこ

とを意味している。図 41 で考えると、入力画像について、 3×3 のフィルタを適用して畳み込みを行い、結果として、 26×26 の大きさの 32 チャンネルの画像(行列)を生成していることになる。

3 行目では、`layers.MaxPooling2D` を使って、MAX プーリング層を追加している。引数の(2, 2)はプーリングサイズを意味している。その結果、 26×26 画素の画像は半分のサイズの 13×13 画素の画像となる。チャンネル数は 32 のままである。

以下同様に、4 行目で畳み込み層、5 行目で MAX プーリング、6 行目で畳み込み層をそれぞれ追加している。図 41 に示すように、結果として、 3×3 画素の画像が 64 枚(64 チャンネル)生成されることになる。

7 行目の `layers.Flatten` では、上記で生成したテンソルの要素値を 1 列に並べてベクトル化する操作を行う。具体的には、 3×3 の画像が 64 枚あるので、その画像の各画素を 1 列に並べることになるので、 $3 \times 3 \times 64 = 576$ の数値列(ベクトル)となる。これが、全結合層の入力ベクトルとなる。

8 行目の `layers.Dense` は、全結合層の追加を意味する。引数の 64 は、ユニットの個数、`activation` は活性化関数の種類を示す。よって、図 41 の下段に示すように、入力層 576 個のユニットから全結合された 64 個のユニットで構成され、このユニットからの出力には活性化関数として ReLU が適用される。

9 行目も全結合層の追加であり、これが出力層となる。ここでは、数字クラス(数字の 0 から 9)を表現するために 10 個のユニットを用意し、活性化関数はソフトマックス関数を指定している。これにより、図 41 に示すように各ユニットからは、各数字クラスである確率値が出力されることになる(例えば、一番上の出力ユニットからは、入力画像が数字の 0 である確率値が出力される)。

最後に 10 行目の `model.compile` で、最適化アルゴリズムを Adam、誤差関数としてクラス出力に関するクロスエントロピー誤差を採用、学習途中の評価のための尺度として識別精度(accuracy)を指定している。

CNN の学習と評価

上記の手順で作成した CNN を用いて、学習データについて学習を行うためには、次のようにプログラムを記述する。

```
model.fit(train_images, train_labels, epochs=5)
```

ここで、`epochs` を 5 としているので、数字画像全体(60,000 枚)を一通り学習する計算を 5 回繰り返していることになる。学習結果は次のとおりである。なお、学習時のフィルタの値、バイアス値、重みの値の初期値は乱数によって設定されるため、学習ごとに微妙に結果の値が異なる点に留意されたい。

```
Epoch 1/5
1875/1875 [=====] - 19s 4ms/step - loss: 0.1599 - accuracy: 0.9500
Epoch 2/5
1875/1875 [=====] - 7s 4ms/step - loss: 0.0494 - accuracy: 0.9844
Epoch 3/5
1875/1875 [=====] - 8s 4ms/step - loss: 0.0364 - accuracy: 0.9887
Epoch 4/5
1875/1875 [=====] - 8s 4ms/step - loss: 0.0283 - accuracy: 0.9919
Epoch 5/5
1875/1875 [=====] - 7s 4ms/step - loss: 0.0213 - accuracy: 0.9932
<keras.callbacks.History at 0x7e535a0bbbb0>
```

これより、学習データについては、99.32%の認識率が得られていることがわかる。次にテストデータについての認識率を評価してみよう。

```
test_loss, test_acc = model.evaluate(test_images, test_labels)
```

```
313/313 [=====] - 1s 3ms/step - loss: 0.0269 - accuracy: 0.9923
```

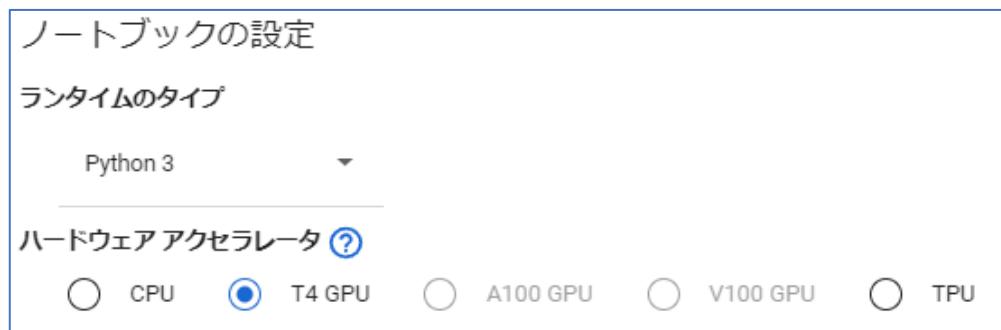
この結果から、テストデータの認識率は 99.23% となっていることがわかる。

以上の結果を前述の「3 層型ニューラルネットワークを用いた数字認識」の結果と比較してみよう。前回の結果は表 9 より、各種パラメータを調整しても、テストデータに対する認識率は 97.85% にとどまっていた。それに比べると、CNN を用いた手法では、99% 以上の認識率を達成している。このように、CNN では、畳み込み層とプーリング層を組み合わせて適用することにより、画像から適切な特徴量を抽出することができ、それを全結合層のニューラルネットワークに入力することで、全結合層のみで構成される単純なニューラルネットワークに比べて、画像認識の精度を上げることができることがわかる。

GPU の使用

CNN では、学習データが多くなると、学習に膨大な時間がかかる場合がある。そこで、Google Colaboratory では、必要に応じて、GPU を使用して、処理の高速化をすることが可能である。設定は、次のとおりである。

メニューの「編集」から「ノートブックの設定」を選択、「ハードウェアアクセラレータ」から「T4 GPU」などを選択すればよい。



適用例:畳み込みニューラルネットワーク(CNN)を用いたカラー画像認識

前述の適用例では、白黒のグレースケール画像(濃淡画像)として数字画像の認識を行ってみたが、ここでは、カラー画像を対象としてCNNを用いた学習と認識を行つてみる。なお、ここで使用するCNNの構造は、画像の入力部分がRGBの3チャンネル画像となる他は、数字認識に用いたCNNの構造とほぼ同じである(本プログラムは、TensorFlowのチュートリアルページ¹⁰の内容を参考としている)。

各種モジュールの読み込みとカラー画像データの読み込み

```
1 import tensorflow as tf
2 from keras import datasets, layers, models
3 import matplotlib.pyplot as plt
4 (train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
5 train_images, test_images = train_images / 255.0, test_images / 255.0
```

1行目では、TensorFlowモジュールをインポートし、2行目では、Kerasモジュール内の各種関数を直接呼び出せるように設定、3行目では、描画モジュールであるMatplotlibをインポートしている。4行目は、使用するカラー写真画像データを読み込んでいる。ここで使用するのは、CIFAR-10と呼ばれる画像データベースであり、各画像は10種類の物体(飛行機、車、鳥、猫、鹿、犬、カエル、馬、船、トラック)を撮影した32×32画素のカラー画像である。読み込んだtrain_imagesは学習用データであり、50,000枚の画像からなり、train_labelsはそれらの各画像に対応する正解データ(0から9の数字、0が飛行機、1が車、…、9がトラックに対応する)である。一方、test_imagesは評価用データであり、10,000枚の画像からなり、test_labelsはそれらの各画像に対応する正解データである。そして、最後の5行目では、RGBの各画像の各画素が0から255の輝度値を持っているので、この値を0から1に正規化している。

画像データと正解データの表示

```
1 class_names = ['飛行機', '車', '鳥', '猫', '鹿',
                 '犬', 'カエル', '馬', '船', 'トラック']
2 plt.imshow(train_images[0])
3 plt.show()
```

¹⁰ <https://www.tensorflow.org/tutorials/images/cnn?hl=ja>

```
4     print(class_names[train_labels[0][0]])
```

先ほど読み込んだ学習用の画像データとそれに対応する正解データを表示してみよう。まず、1行目では、10種類の画像クラスに名前をつけて、class_namesというリストに格納している。2, 3行目では、学習用画像データの最初の画像 train_images[0]（ここで、添え字 0 が画像番号を示す。2番目の画像は train_images[1] である。）を画面に表示している（図 42 参照）。

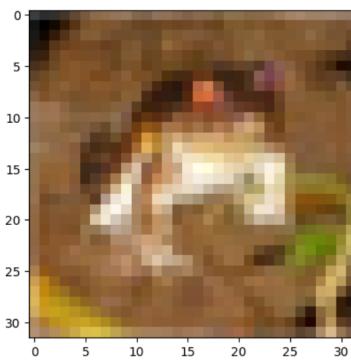


図 42 画像データの表示例

最後に 4 行目で、上記画像データに対応する正解データ train_labels[0][0] のクラス名を表示している。ここで、train_labels は 2 次元配列であり、最初の添え字は画像番号を示し、2 つ目の添え字は全て 0 となっている。この画像では「カエル」と表示される。

CNN の構成

```
1 model = models.Sequential()
2 model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
3 model.add(layers.MaxPooling2D((2, 2)))
4 model.add(layers.Conv2D(64, (3, 3), activation='relu'))
5 model.add(layers.MaxPooling2D((2, 2)))
6 model.add(layers.Conv2D(64, (3, 3), activation='relu'))
7 model.add(layers.Flatten())
8 model.add(layers.Dense(64, activation='relu'))
9 model.add(layers.Dense(10, activation='softmax'))
10 model.compile(optimizer='adam',
                 loss='sparse_categorical_crossentropy',
                 metrics=['accuracy'])
```

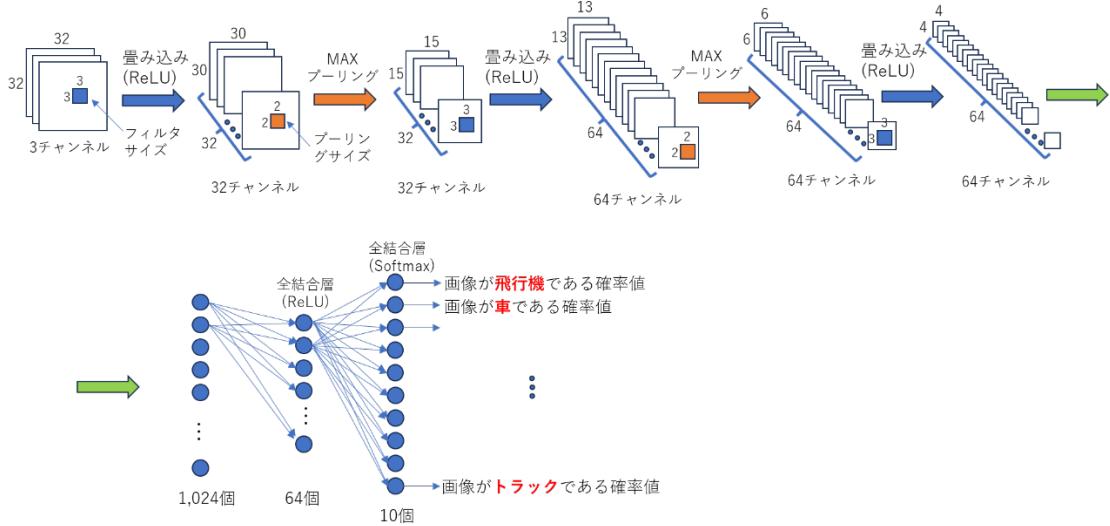


図 43 CNN の構造

図 43 に、カラー画像認識に使用する畳み込みニューラルネットワークの構造を示し、その上に、この構造を実現するためのプログラムを示す。以下、プログラム行と図を対照しながら、各部の説明をする。

プログラムの `model.add` メソッドは、ネットワークに各層を追加している。まず、2 行目では、`layers.Conv2D` を使って、畳み込み層を追加している。32 は、フィルタ数を示し、(3, 3)はフィルタのサイズ、`activation` は活性化関数の指定(ここでは、ReLU を指定)、`input_shape` は入力画像のサイズが 32×32 画素でカラー画像であることを意味している(カラー画像は R, G, B の 3 枚の画像で構成されているので、3 チャンネルの画像であると考えられる)。図 43 で考えると、入力画像について、 3×3 のフィルタを適用して畳み込みを行い、結果として、 30×30 の大きさの 32 チャンネルの画像(行列)を生成することになる。

3 行目では、`layers.MaxPooling2D` を使って、MAX プーリング層を追加している。引数の(2, 2)はプーリングサイズを意味している。その結果、 30×30 画素の画像は半分のサイズの 15×15 画素の画像となる。チャンネル数は 32 のままである。

以下同様に、4 行目で畳み込み層、5 行目で MAX プーリング、6 行目で畳み込み層をそれぞれ追加している。図 43 に示すように、結果として、 4×4 画素の画像が 64 枚(64 チャンネル)生成されることになる。

7 行目の `layers.Flatten` では、上記で生成したテンソルの要素値を 1 列に並べてベクトル化する操作を行う。具体的には、 4×4 の画像が 64 枚あるので、その画像の各画素を 1 列に並べることになるので、 $4 \times 4 \times 64 = 1,024$ の数値列(ベクトル)となる。これが、全結合層の入力ベクトルとなる。

8 行目の `layers.Dense` は、全結合層の追加を意味する。引数の 64 は、ユニットの個数、`activation` は活性化関数の種類を示す。よって、図 43 の下段に示すように、入

力層 1,024 個のユニットから全結合された 64 個のユニットで構成され、このユニットからの出力には活性化関数としてReLUが適用される。

9 行目も全結合層の追加であり、これが出力層となる。ここでは、画像クラスを表現するために 10 個のユニットを用意し、活性化関数はソフトマックス関数を指定している。これにより、図 43 に示すように各ユニットからは、各画像クラスである確率値が出力されることになる（例えば、一番上の出力ユニットからは、入力画像が「飛行機」である確率値が出力される）。

最後に 10 行目の model.compile で、最適化アルゴリズムを Adam、誤差関数としてクラス出力に関するクロスエントロピー誤差を採用、学習途中の評価のための尺度として識別精度(accuracy)を指定している。

CNN の学習と評価

上記の手順で作成した CNN を用いて、学習データについて学習を行うためには、次のようにプログラムを記述する。なお、ここでは、各エポックでの学習済み CNN を用いた評価用データに対する識別率も合わせて表示するため、validation_data の項目を付加している。

```
model.fit(train_images, train_labels, epochs=10,
          validation_data=(test_images, test_labels))
```

ここで、epochs を 10 としているので、カラー画像全体(50,000 枚)を一通り学習する計算を 10 回繰り返すことになる。各エポックでの学習データ(50,000 枚)に対する認識率と評価データ(10,000 枚)に対する認識率の経過は次のとおりである。なお、学習時のフィルタの値、バイアス値、重みの値の初期値は乱数によって設定されるため、学習ごとに微妙に結果の値が異なる点に留意されたい。

```
Epoch 1/10
1563/1563 [=====] - 11s 5ms/step - loss: 1.4920 - accuracy: 0.4553 - val_loss: 1.3478 - val_accuracy: 0.5307
Epoch 2/10
1563/1563 [=====] - 9s 6ms/step - loss: 1.1275 - accuracy: 0.6017 - val_loss: 1.0339 - val_accuracy: 0.6382
Epoch 3/10
1563/1563 [=====] - 9s 6ms/step - loss: 0.9737 - accuracy: 0.6586 - val_loss: 0.9300 - val_accuracy: 0.6738
Epoch 4/10
1563/1563 [=====] - 9s 6ms/step - loss: 0.8748 - accuracy: 0.6934 - val_loss: 0.9304 - val_accuracy: 0.6749
Epoch 5/10
1563/1563 [=====] - 9s 6ms/step - loss: 0.7975 - accuracy: 0.7207 - val_loss: 0.9057 - val_accuracy: 0.6888
Epoch 6/10
1563/1563 [=====] - 8s 5ms/step - loss: 0.7397 - accuracy: 0.7427 - val_loss: 0.8813 - val_accuracy: 0.6927
Epoch 7/10
1563/1563 [=====] - 9s 6ms/step - loss: 0.6917 - accuracy: 0.7566 - val_loss: 0.8335 - val_accuracy: 0.7133
Epoch 8/10
1563/1563 [=====] - 9s 6ms/step - loss: 0.6493 - accuracy: 0.7719 - val_loss: 0.8856 - val_accuracy: 0.7027
Epoch 9/10
1563/1563 [=====] - 9s 6ms/step - loss: 0.6019 - accuracy: 0.7884 - val_loss: 0.8800 - val_accuracy: 0.7030
Epoch 10/10
1563/1563 [=====] - 9s 6ms/step - loss: 0.5622 - accuracy: 0.8023 - val_loss: 0.8741 - val_accuracy: 0.7180
<keras.callbacks.History at 0x7d25c22a7220>
```

ここで、accuracy の部分が学習データに対する認識率、val_accuracy の部分が評価データに対する認識率を示す。この結果から、カラー画像について、71%以上の認識率が得られていることがわかる。前述の数字画像認識に比べると、大幅に認識率が低下しているのは、この分類問題が数字認識に比べて難しいことを示している。より精度を上げるためにには、CNN の構造や学習方法をさらに見直す必要があるだろう。

单一画像の認識

ここでは、評価用の画像データから 1 枚の画像を選択して、その画像について、学習済みの CNN を使って認識を行ってみよう。

評価データの 2 番目の画像(船の画像)について、認識を行うプログラムを下記に示す。

```
1 import numpy as np
2 image = test_images[1]
3 plt.imshow(image)
4 plt.show()
5 ex_image = image[np.newaxis, ...]
6 predict_image = model.predict(ex_image)
7 print(predict_image[0])
8 pred_class = predict_image[0].argmax()
9 print(class_names[pred_class])
```

2 行目で評価用画像データ `test_images` の 2 番目の画像(添え字「1」が 2 番目の画像を指す)の情報を `image` に代入し、3・4 行目で、その画像を表示している。この CNN モデルで認識する場合、画像群の形式が $(?, 32, 32, 3)$ の 4 次元配列になっている必要がある($?$ 部分は画像の枚数を示す)。現在の `image` の画像形式は $(32, 32, 3)$ の 3 次元配列になっているため、5 行目で次元を増やした `ex_image` の画像を生成している。これにより、`ex_image` の形式は $(1, 32, 32, 3)$ となる。学習したモデルによる認識(推定)には、`model.predict` を用いる。6 行目では、引数に認識したい画像 `ex_image` を渡して、その画像の認識を行い、結果を `predict_image` に代入している。`predict_image` は 1 行 10 列の 2 次元配列になっており、その 1 行目を取り出して表示しているのが 7 行目である。表示結果を見ると、10 個の 0 から 1 の実数値が並んでいく。これは、それぞれ、飛行機、車、…、トラックに対する確率値を示している。8 行目は、この値の中で、最も大きな値を持つ画像クラスに対応する添え字を取り出し、`pred_class` 変数に代入している。そして、最後の 9 行目で、その添え字に該当するクラス名を表示している。

ここでは、認識画像が「船」の画像なので、predict_image[0]の9番目の値が最も大きくなり(つまり、船の画像である確率値が最も高くなる)、そのクラス名として「船」が表示されるはずである。

任意画像の認識

前述までの画像認識では、CIFAR-10で用意された評価用画像データについての認識を説明してきたが、ここでは、自分で用意した画像を、学習済みのCNNを使って認識してみよう。

まず、認識したい画像(この例では、dog_image.jpgのファイル名で保存されたJPEG画像を対象とする)を個人のGoogle ドライブの領域にアップロードしておく。通常、Google Colaboratory を用いると、そのプログラムファイルは、Google ドライブの「Colab Notebooks」フォルダの下に保存されるので、画像ファイルもこのフォルダの下にアップロードしておくことにする。

Google Colaboratory から Google ドライブのファイルを読み込むためには、Google ドライブをマウントする必要がある。この操作は、次のプログラムを実行すればよい(最初に実行するとドライブへのアクセスを許可してよいか聞かれるので、「許可」をする)。

```
from google.colab import drive  
drive.mount('/content/drive')
```

Google ドライブからファイル(dog_image.jpg)を読み込んで、その画像について認識を行うプログラムは次のとおりである。

```
1 from PIL import Image  
2 orgimg = Image.open('/content/drive/MyDrive/Colab  
Notebooks/dog_image.jpg').resize((32, 32))  
3 image = np.array(orgimg)/255.0  
4 plt.imshow(image)  
5 plt.show()  
6 ex_image = image[np.newaxis, ...]  
7 predict_image = model.predict(ex_image)  
8 pred_class = predict_image[0].argmax()  
9 print(class_names[pred_class])
```

このプログラムでは、画像処理用のモジュールとしてPillowを用いる。1行目はそのPillowモジュールのインポートを行っている。2行目は、先ほどアップロードした画像フ

ファイル(dog_image.jpg)を読み込み、画像サイズを 32×32 画素にリサイズし、orgimg に代入している。ここで、orgimg は Pillow モジュールの画像形式となっているため、これをNumPyの形式に変換し、さらに各要素を 0 から 1 の値に正規化しているのが 3 行目である。これにより $(32,32,3)$ 形式の 3 次元配列が生成される。4 行目以降の処理は前節で説明した処理と同じである。ここでは、「犬」の画像を読み込ませ、それを認識した結果、「犬」と推定できたことを示している。

以上、任意の画像を読み込んで CNN で認識することが可能となった。教材を学習している方も、是非、認識したい画像を読み込ませて、その認識精度を確かめていただきたい。

参考文献(続き)

- 75 我妻 幸長 “はじめてのディープラーニング”, SB クリエイティブ (2018)
- 76 山下 隆義 “イラストで学ぶディープラーニング 改訂第 2 版”, 講談社 (2018)
- 77 岡野原 大輔 “ディープラーニングを支える技術”, 技術評論社 (2022)