第 1 章

フィードフォーワード・ニューラルネットワーク

1.1　アーキテクチャとニューラルネットワークの働き

ニューラルネットワーク全体の構造のことを**アーキテクチャ**（architecture）という。ニューラルネットワークは主に**入力層**（input layer）、**隠れ層**（hidden layer）、**出力層**（output layer）から構成される。それぞれの層は本質的にはベクトルや行列、テンソルであり、その成分を**ユニット**（unit）とよぶ。各層は 1 つ前の層の関数になっており、全体として連鎖的な構造をなす。この連鎖の長さを**深さ**（depth）とよび、深層学習という名前はこの用語に由来している。

入力から様々な計算を経て出力がはき出される。出力がモデル自体に戻すようなフィードバックがないとき、このニューラルネットワークを**フィードフォーワード・ニューラルネットワーク**、**順伝搬型ニューラルネットワーク**（feedforward neural networks）などとよぶ。フィードフォーワード・ニューラルネットワークの目的はある関数 f を近似することである。本稿においては「手書き数字の書かれた画像データからその数字を予測する」ような関数 f をフィードフォーワード・ニューラルネットワークで近似することが目的である。

n 個の入力 $x_1, \ldots, x_n$ に対して、パラメータ $b, w_1, \ldots, w_n$ とある**活性化関数**（activation function）φ を用いて $\varphi(b + w_1x_1 + \cdots + w_nx_n)$ を出力するようなシステムを（単純）**パーセプトロン**（perceptron）あるいは**人工ニューロン**（articicial neuron）という。単純パーセプトロンが複数集まって層をなし、さらにその層が連鎖的に重なったものを**多層パーセプトロン**（multiple perceptrons, MLP）という。フィードフォーワード・ニューラルネットワークは MLP ともみなせる。

図 1.1 の手で書かれた「5」の字を、図 1.2 に示したシンプルなフィードフォーワード・ニューラルネットワークに認識させることを考える。図 1.1 は $28 \times 28 = 784$ ピクセルのグレースケールの画像で、それぞれのピクセルが 0 から 255 までの黒さに対応した値を持っている。この 28×28 の配列を 784 次元のベクトルに変換して、これを入力 $\boldsymbol{x}$ とする。また図 1.1 の画像には $y = 5$ という正解を表すラベルがついている。

隠れ層は図 1.2 の例では 1 層で、これはベクトルである。入力 $\boldsymbol{x}$ と重みのパラメータ $W^{(1)}$ の行列積 $\boldsymbol{u}^{(1)} = W^{(1)}\boldsymbol{x}$ を計算し、そのそれぞれの成分について活性化関数 φ_1 を施したベクトルを隠れ層 $\boldsymbol{h} = \varphi_1\big(\boldsymbol{u}^{(1)}\big) = \varphi_1\big(W^{(1)}\boldsymbol{x}\big)$ としている。図 1.2 のユニット間を結ぶ線分が $W^{(1)}$ の成分を表現している。$\boldsymbol{h}$ にさらにパラメータ $W^{(2)}$ をかけて活性化関数 φ_2 に通すことで、出力のベクトル

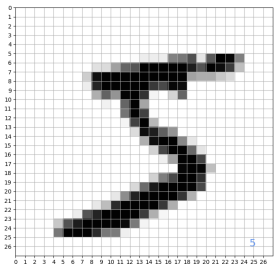

図 1.1　手書き数字の「5」

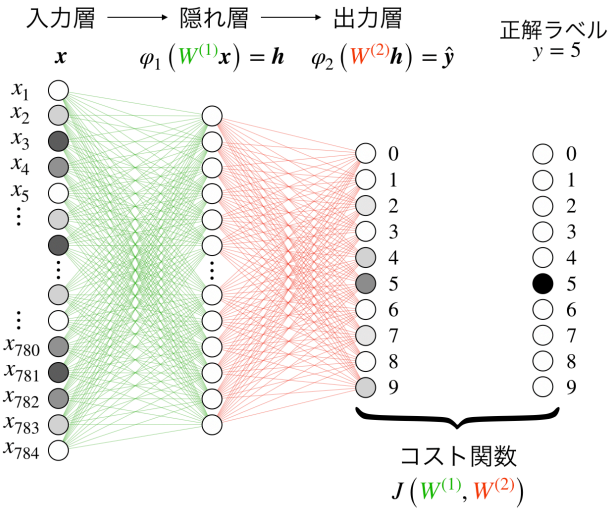

図 1.2　ニューラルネットワークの例

$\hat{\boldsymbol{y}}=\varphi_2\left(\boldsymbol{u}^{(2)}\right)=\varphi_2\left(W^{(2)}\boldsymbol{h}\right)$ を得ている。$\hat{\boldsymbol{y}}$ は 0 から 9 までの数字に対応した 10 成分を持つベクトルで、各成分は入力のラベルがその数字である確率を表している。図 1.2 では確率が最も高いクラス 5 と判定されるが、クラス 4 やクラス 9 である確率もあることを表現した。ニューラルネットワークではこのような流れで多クラス分類を行う。

次にニューラルネットワークを訓練させる方法を説明する。データのラベル y に対応した**ワンホットベクトル**（one-hot vector）と出力 $\hat{\boldsymbol{y}}$ を用いて、それらの違いを表現する**コスト関数**（cost function）を作る。この例では $y=5$ であるから、ワンホットベクトルは第 6 成分のみが 1 でその他の成分が 0 のベクトルになる。出力 $\hat{\boldsymbol{y}}$ は $W^{(1)}$ と $W^{(2)}$ の関数であるから、コスト関数 J もそれらの関数になる。コスト関数としては平均二乗誤差や交差エントロピーを採用することが多い。多数の訓練データを用いて、コスト関数 $J\left(W^{(1)},W^{(2)}\right)$ が最小となるようなパラメータ $W^{(1)}$ と $W^{(2)}$ を学習することで、その汎化性能が向上していく。

コスト関数を計算するまでの流れは、**グラフ**（graph）を用いて図 1.3 のようにも図示することができる。入力層などの各変数を**ノード**（node）として表し、それらの依存構造を矢印の向きで表している。ノードのことを神経細胞を意味する**ニューロン**（neuron）とよぶこともある。ノードの位置関係はさほど重要ではなく、この依存関係が明確であればよい。各変数間の演算はタイプライタ体で書き込んである。`matmul` は行列積の演算を表している。`relu`、`softmax` は活性化関数 φ_1,φ_2 ををそれぞれ ReLU（rectified linear function）、**ソフトマックス関数**（softmax function）としたことを示している。`cross_entropy` でコスト関数として交差エントロピーを採用したことを表した。

コスト関数の最適化に用いるアルゴリズムには様々あるが、本稿ではミニバッチ勾配降下法や Adam など、その勾配を用いて最適化を図るアルゴリズムを使用する。

1.2　入力層

図 1.2 の例では、入力は 1 つの訓練データを変形したベクトルであった。しかし入力は行列やテンソルであってもよい。コスト関数の最適化アルゴリズムとしてミニバッチ勾配降下法を用いるならば、複数の訓練データを用いて作られた計画行列 X が 1 回の学習における入力になる。例えば手書き数字の認識を

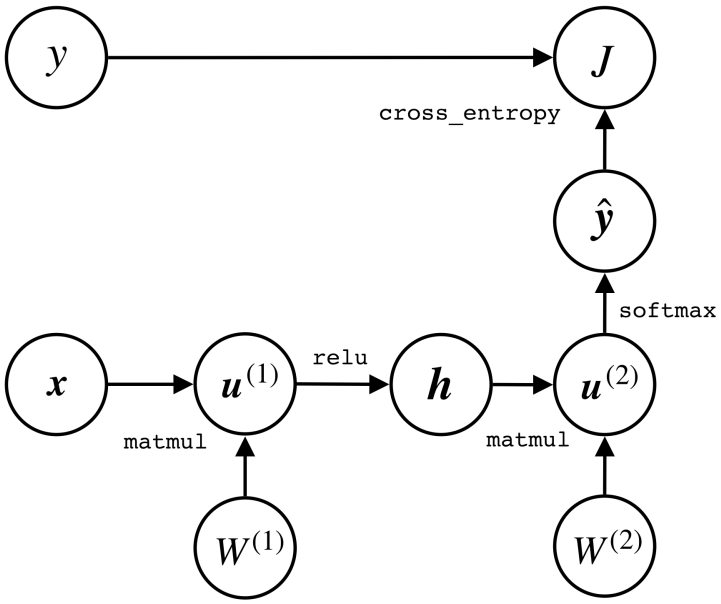

図 1.3　件のニューラルネットワークに対するグラフ

するならば、ミニバッチに選ばれた訓練データ数を m' 個として、計画行列は

$$X = \begin{bmatrix} x_1^{(1)} & \cdots & x_{784}^{(1)} \\ \vdots & & \vdots \\ x_1^{(m')} & \cdots & x_{784}^{(m')} \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}^{(1)\top} \\ \vdots \\ \boldsymbol{x}^{(m')\top} \end{bmatrix} \tag{1.1}$$

のようになる。

1.3　隠れ層

隠れ層の層数を**深さ**（depth）といい、隠れ層のユニット数のことを**幅**（width）という。ニューラルネットワークのアーキテクチャを決めることは、この深さと幅を決めることである。経験的に、隠れ層 1 層であっても目的の関数を十分よく近似できることは知られており、**万能近似定理**（universal approximation theorem）[**?**] が理論的にそのようなニューラルネットワークが存在できることを保証している。しかし存在が認められたニューラルネットワークを実際に学習させることができるとは限らず、またその幅を非現実的に大きくしなければいけない可能性もある。その場合はニューラルネットワークを深くする、つまり隠れ層を増やすことでそれらの問題を解決できることがある。

ニューラルネットワークの人工ニューロンは、入力に線形変換を施した後に非線形関数または恒等関数（引数そのものを返す関数）を適用して出力する。この関数を**活性化関数**（activation function）という。コスト関数の微分を用いてニューラルネットワークの学習を行うため、導関数の性質がよい関数が活性化関数として用いられる。ここでは本稿の手書き数字の認識に用いた ReLU、シグモイド関数、ソフトマックス関数という 3 つの活性化関数を紹介する。

正規化線形関数（rectified linear function）の定義は

$$\varphi(x) = \max\{0, x\} \tag{1.2}$$

である。すなわち $x \leq 0$ ならば 0 を返し、$x > 0$ ならば x 自身を返す関数である。これを使用したユニットのことを正規化線形ユニット（rectified linear unit, **ReLU**）というが、関数そのものも ReLU とよぶことが多い。ReLU の導関数は $x < 0$ ならば 0、$x > 0$ ならば 1 で、$x = 0$ では定義されない。このように導関数が定数であるため計算コストが小さく、正の入力に対しては**勾配消失**（節 1.5.3）が起こりにくいという利点があるため、広く用いられている。しかし負の入力については導関数が 0 であるためによる学習が進まない。

$$\varphi(x) = \frac{1}{1 + \exp(-x)} \tag{1.3}$$

という関数を**標準シグモイド関数**（standard sigmoid function）、**ロジスティック・シグモイド関数**（logistic sigmoid function）、あるいは単に**シグモイド関数**（sigmoid function）などという。本稿では一貫してシグモイド関数とよび、式 (1.3) の右辺を $\sigma(x)$ と表記する。シグモイド関数の導関数は $\sigma'(x) = (1 - \sigma(x))\sigma(x)$ と簡単であるから、勾配降下法と相性がよい。しかし引数の絶対値が大きいとその勾配は小さくなるため、勾配消失が起こりやすい。また導関数の値は最大でも 0.25 しか取らないため、勾配降下法による収束が遅いというデメリットがある。$\tanh(x)$ を代わりに使うことで勾配降下法による収束はいくらか早くなるが、勾配消失問題は解決されていない。このため現在では ReLU がより多く用いられる。

$$\varphi(z_k) = \frac{\exp(z_k)}{\sum_{k=0}^{K} \exp(z_k)} \tag{1.4}$$

という関数を**ソフトマックス関数**（softmax function）という。これは手書き数字の認識などの多クラス分類の出力層で用いられる。分類するクラスが $k = 0, 1, \ldots, K$ の $(K + 1)$ 種類であるとき、それらに対応した成分を持つベクトル $\boldsymbol{z} = [z_0, \ldots, z_K]^\top$ が入力である。式 (1.4) の右辺を $\mathrm{softmax}(\boldsymbol{z})_k$ のように表記する。すべてのクラスについて $\mathrm{softmax}(\boldsymbol{z})_k$ の総和を取れば 1 になることから、確率として取り扱えることが利点である。しかし入力値の間の差が大きいと、勾配が消失して学習が進まない。このことを回避するためには、定数を引数から引いても変わらないという性質を使って、引数からあらかじめ $\max_k z_k$ を引いてからソフトマックス関数を使えばよい。導関数は

$$\frac{\partial}{\partial z_j}\mathrm{softmax}(\boldsymbol{z})_i = \Big(\delta_{ij} - \mathrm{softmax}(\boldsymbol{z})_j\Big)\mathrm{softmax}(\boldsymbol{z})_i$$

となる。δ_{ij} はクロネッカーのデルタである。

ReLU、シグモイド関数、ソフトマックス関数のグラフと、ReLU、シグモイド関数の導関数のグラフを図 1.4 に示す。ただしソフトマックス関数のグラフは表示の都合上 100 倍したものを載せてある。

1.4　出力層

節 1.1 の図 1.2 での手書き数字の認識をするニューラルネットワークにおいては、出力は 0 から 9 までの数字に対応した成分を持つ、10 次元のベクトルになる。多クラス分類であるから、出力層の活性化関数はソフトマックス関数を用いる。すなわち出力層に入力されるベクトルを $\boldsymbol{z}$ とすれば

$$\hat{\boldsymbol{y}} = \mathrm{softmax}(\boldsymbol{z})$$

である。したがって出力されたベクトルの各成分は「その成分に対応した数字がデータに書かれている確率」として取り扱うことができる。

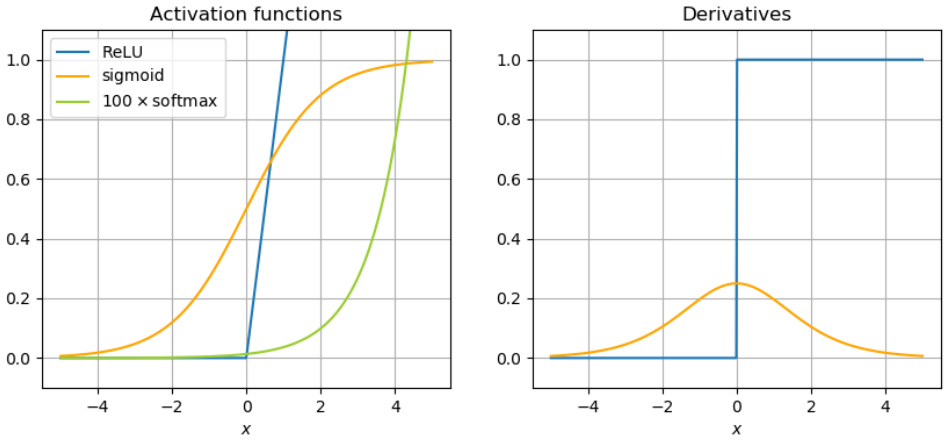

図 1.4　活性化関数とそれらの導関数のグラフ

ミニバッチ確率勾配法で多クラス分類を行うことを考える。ミニバッチに選ばれた訓練データを $\mathbb{X} = \left\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m')}\right\}$ とする。計画行列は式 (1.1) を用いる。i 番目の訓練データ $\boldsymbol{x}^{(i)}$ を用いたときの出力のベクトルを $\hat{\boldsymbol{y}}^{(i)} = \left[\hat{y}_0^{(i)}, \ldots, \hat{y}_K^{(i)}\right]^\top$ とし、$\boldsymbol{x}^{(i)}$ のラベルに対応したワンホットベクトルを $\boldsymbol{y}^{(i)}$ とする。この出力とワンホットベクトルの近さを小さくする方向にパラメータを変更していくのが、ニューラルネットワークの訓練の目的である。ニューラルネットワークで用いているパラメータはベクトル、行列、あるいはテンソルであるかもしれず、複数ある場合もあるので、それらをまとめて集合 $\mathbb{W}$ と書くことにする。ここでは「近さ」を表す**コスト関数**（cost function）をどのように表現するか、その微分はどうなるかを説明する。$\hat{\boldsymbol{y}}^{(i)}$ はパラメータ $\mathbb{W}$ の関数であるから、コスト関数も $\mathbb{W}$ の関数 $J(\mathbb{W})$ である。

1 つ目はコスト関数を**平均二乗誤差**（mean squared error, MSE）とする方法である。平均二乗誤差は $\hat{\boldsymbol{y}}^{(i)}$ と $\boldsymbol{y}^{(i)}$ の差の L^2 ノルムの 2 乗を平均したものである。

$$J(\mathbb{W}) = \frac{1}{m'} \sum_{i=1}^{m'} \left\| \hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)} \right\|_2^2. \tag{1.5}$$

このコスト関数の出力 $\hat{\boldsymbol{y}}^{(i)}$ に関する勾配 $\boldsymbol{\nabla}_{\hat{\boldsymbol{y}}^{(i)}} J(\mathbb{W})$ は

$$\boldsymbol{\nabla}_{\hat{\boldsymbol{y}}^{(i)}} J(\mathbb{W}) = \frac{2}{m'} \sum_{i=1}^{m'} \left(\hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)} \right)$$

となる。もう 1 つはコスト関数として交差エントロピーを用いる方法がある。

$$J(\mathbb{W}) = -\frac{1}{m'} \sum_{i=1}^{m'} \sum_{k=0}^{K} y_k^{(i)} \log \hat{y}_k^{(i)}. \tag{1.6}$$

ただし $\boldsymbol{y}^{(i)} = \left[y_0^{(i)}, \ldots, y_K^{(i)}\right]^\top$ はワンホットベクトルであるから、k に関する総和はほとんどゼロになって、i 番目のラベル $y^{(i)}$ に対応した項だけが残ることになる。このコスト関数の出力 $\hat{\boldsymbol{y}}^{(i)}$ に関する勾配の j 成分 $\left(\boldsymbol{\nabla}_{\hat{\boldsymbol{y}}^{(i)}} J(\mathbb{W})\right)_j$ は

$$\left(\boldsymbol{\nabla}_{\hat{\boldsymbol{y}}^{(i)}} J(\mathbb{W})\right)_j = -\frac{1}{m'} \frac{y_j^{(i)}}{\hat{y}_j^{(i)}}$$

となる。

1.5　誤差逆伝搬法

ニューラルネットワークはコスト関数が最小になるようにパラメータを更新しながら訓練する。その際にコスト関数のパラメータに関する勾配が必要となる。ここでは TensorFlow が実装している**誤差逆伝搬法**または**バックプロパゲーション**（backpropagation）の説明をする。

1.5.1　微分の連鎖律

誤差逆伝搬法には微分の連鎖律を用いる。ある関数 f がベクトル $\boldsymbol{u}$ の関数であり、ベクトル $\boldsymbol{u}$ はベクトル $\boldsymbol{v}$ の関数であるとする。このとき f の $\boldsymbol{v}$ に関する勾配 $\boldsymbol{\nabla}_{\boldsymbol{v}} f$ は微分の**連鎖律**（chain rule）を用いて次のように計算できる。

$$(\boldsymbol{\nabla}_{\boldsymbol{v}} f)_i = \frac{\partial f}{\partial v_i} = \frac{\partial f}{\partial u_j}\frac{\partial u_j}{\partial v_i}.$$

ただしアインシュタインの縮約規則を用いている。また (j,i) 成分に $\partial u_j/\partial v_i$ を持つ行列を、$\boldsymbol{v}$ に関する $\boldsymbol{u}$ の**ヤコビ行列**（Jacobian matrix）という。このヤコビ行列を $\partial \boldsymbol{u}/\partial \boldsymbol{v}$ で表してさらに計算を進めると、

$$(\boldsymbol{\nabla}_{\boldsymbol{v}} f)_i = \frac{\partial u_j}{\partial v_i}\frac{\partial f}{\partial u_j} = \left(\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{v}}\right)_{ji} (\boldsymbol{\nabla}_{\boldsymbol{u}} f)_j = \left[\left(\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{v}}\right)^{\top}\right]_{ij} (\boldsymbol{\nabla}_{\boldsymbol{u}} f)_j = \boldsymbol{a}_i^{\top} (\boldsymbol{\nabla}_{\boldsymbol{u}} f)$$

となる。ただし $(\partial \boldsymbol{u}/\partial \boldsymbol{v})^{\top}$ の i 列目の行ベクトルを $\boldsymbol{a}_i^{\top}$ とした。ゆえに求めていた勾配は

$$\boldsymbol{\nabla}_{\boldsymbol{v}} f = \left(\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{v}}\right)^{\top} (\boldsymbol{\nabla}_{\boldsymbol{u}} f)$$

となることがわかる。

より一般に、ある関数 f がテンソル $\mathbf{U}$ の関数であり、テンソル $\mathbf{U}$ はテンソル $\mathbf{V}$ の関数であるとする。このとき f の $\mathbf{V}$ に関する勾配 $\boldsymbol{\nabla}_{\mathbf{V}} f$ がどうなるかを考える。$\boldsymbol{\nabla}_{\mathbf{V}} f$ はテンソル $\mathbf{V}$ と同じサイズのテンソルであるが、その成分を一つの添字 i で指定できる。すなわちテンソル $\boldsymbol{\nabla}_{\mathbf{V}} f$ をベクトルの配列に変換したと考える。すると $(\boldsymbol{\nabla}_{\mathbf{V}} f)_i = \partial f/\partial V_i$ であるから、先ほどと同じように勾配を計算することができる。

$$\boldsymbol{\nabla}_{\mathbf{V}} f = \sum_i (\boldsymbol{\nabla}_{\mathbf{V}} U_i)\frac{\partial f}{\partial U_i}. \tag{1.7}$$

このように $\boldsymbol{\nabla}_{\mathbf{V}} f$ は $\boldsymbol{\nabla}_{\mathbf{V}} \mathbf{U}$ と $\boldsymbol{\nabla}_{\mathbf{U}} f$ から作り出すことができる。

1.5.2　誤差逆伝搬法を用いた学習

コスト関数を計算するのには入力層から隠れ層を通って出力を出していたが、出力側から入力側に向かって勾配を計算してゆくので誤差逆伝搬法という。ノード $\mathbf{A}$ から関数 f を使ってノード $\mathbf{B}$ の値を計算するときに、同時に勾配 $\boldsymbol{\nabla}_{\mathbf{A}} \mathbf{B}$ を計算して保存しておく。出力層までこれを計算したら、これまで計算していた複数の勾配から逆向きに微分の連鎖律 (1.7) の計算を行う。こうすることで最終的に必要だったコスト関数のパラメータに関する勾配が得られるのだ。

節 1.1 の例で、コスト関数のパラメータ $W^{(2)}$ に関する勾配 $\boldsymbol{\nabla}_{W^{(2)}} J$ を誤差逆伝搬法を用いて計算するためのパスを図 1.2 に描き加えたものを図 1.5 に示す。$W^{(2)} \to \boldsymbol{u}^{(2)}$, $\boldsymbol{u}^{(2)} \to \hat{\boldsymbol{y}}$, $\hat{\boldsymbol{y}} \to J$ の計算をする際にそれぞれ $\boldsymbol{\nabla}_{W^{(2)}} \boldsymbol{u}^{(2)}$, $\boldsymbol{\nabla}_{\boldsymbol{u}^{(2)}} \hat{\boldsymbol{y}}$, $\boldsymbol{\nabla}_{\hat{\boldsymbol{y}}} J$ を計算しておく。そして $\boldsymbol{\nabla}_{\hat{\boldsymbol{y}}} J$ と $\boldsymbol{\nabla}_{\boldsymbol{u}^{(2)}} \hat{\boldsymbol{y}}$ から $\boldsymbol{\nabla}_{\boldsymbol{u}^{(2)}} J$ を、$\boldsymbol{\nabla}_{\boldsymbol{u}^{(2)}} \hat{\boldsymbol{y}}$ と $\boldsymbol{\nabla}_{\boldsymbol{u}^{(2)}} J$ から $\boldsymbol{\nabla}_{W^{(2)}} J$ を計算する。

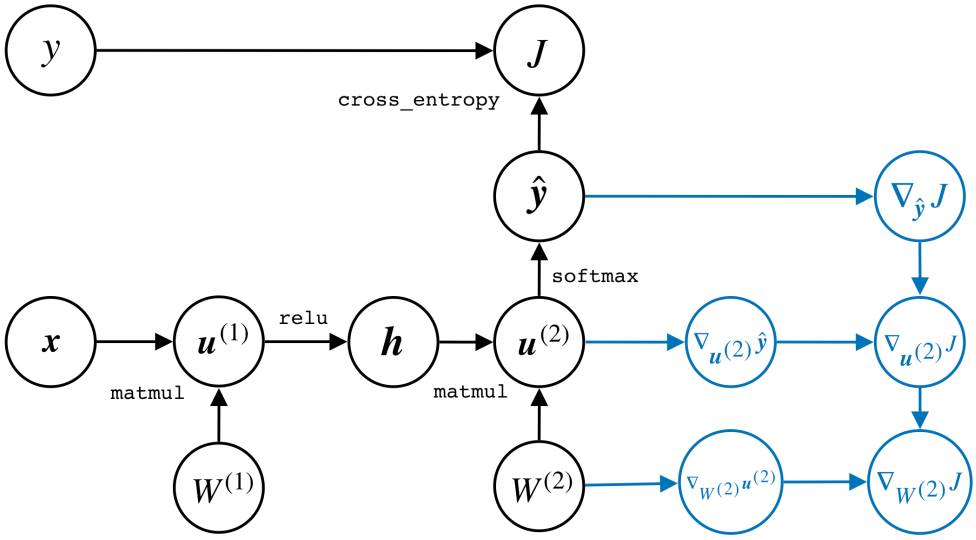

図 1.5　逆伝搬のパスを図 1.2 のグラフに書き加えたもの

1.5.3　勾配消失問題

図 1.4 を見れば分かるように、シグモイド関数 $\sigma(x)$ は $|x|$ が大きい領域で定数値に漸近し、傾きが小さくなっている。このように関数が非常に平坦になることを、自動詞の用法で**飽和している**（saturate）という。もし隠れ層で飽和するような活性化関数を用いていた場合、その傾きは非常に小さくなるため、微分の連鎖律を用いて勾配を掛け合わせてゆくと、入力層まで逆伝搬したときにコスト関数の勾配は非常に小さくなってしまい、勾配降下法によるパラメータの更新が進まなくなる。このように活性化関数が飽和するせいでコスト関数の勾配が非常に小さくなる問題を**勾配消失問題**（vanishing gradient problem）という。

勾配消失問題を回避するためには活性化関数をタスクに適切なものにする、最適化アルゴリズムを変えるなどの対処が必要である。