第 1 章

結果

1.1　フィードフォーワード・ニューラルネットワークによる学習

1.1.1　ベースとなるフィードフォーワード・ニューラルネットワークの構築

節??の結果である。付録??のソースコード??を実行したところ、次のような結果を得た。

```
Test loss: 0.259500997513533
Test accuracy: 0.927500003576279
Computation time: 30.111 sec
```

テストデータに対する正解率は 92.75% であることがわかる。エポック数に対するコスト関数の値、正解率のグラフを出力させたところ、次の図 1.1 のようになった。このようにコスト関数の値や汎化誤差の時間発展をグラフにしたものを**学習曲線**（learning curve）という。コスト関数の値はエポック数に対して単調に減少していることから、過学習は起きていないことがわかる。正解率も単調に増加している。また

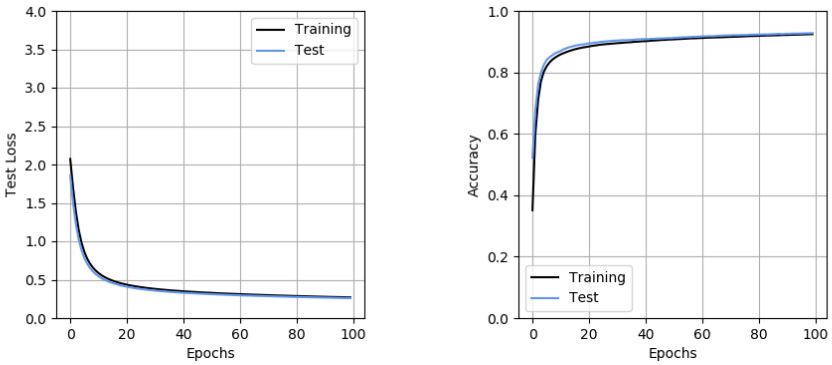

図 1.1　隠れ層が 1 層のニューラルネットワークのコスト関数、正解率の時間発展

パラメータの数を表示させたところ次のようになった。隠れ層（第 0 層）が `dense(Dense)`、出力層（第 1 層）が `dense_1(Dense)` である。パラメータ数の合計は 79,510 で、すべて訓練できていた。

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 100)               78500
```

```
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1010
=================================================================
Total params: 79,510
Trainable params: 79,510
Non-trainable params: 0
_________________________________________________________________
```

さらに初めの 100 個のテストデータに対するモデルの予測を表示させたものを次の図 1.2 に示す。各画像の右下にはネットワークの予測を付し、予測が誤りであったものは横棒で印をつけた。ここでは 100 個のうち 96 個が正解しているため、正解率 92.75% は正しく思われる。しかしながら、人間にとっても判別しづらい字はあるものの、9 を 4 としたり比較的きれいに書かれている 2 を 7 と読み間違えている場合があり、精度は不十分なようである。

図 1.2　先頭 100 個のテストデータに対するモデルの予測

1.1.2　活性化関数の比較

節**??**の結果は次のようになった。活性化関数をシグモイド関数にしただけで、正解率は 4.08% も下がった。また実行時間も 1.26 sec 伸びてしまった。

```
Test loss: 0.45101862847805
Test accuracy: 0.886699998378754
Computation time: 31.371 sec
```

学習曲線を図 1.3 に示す。これと図 1.1 を見比べると、エポック数がおよそ 40 までの範囲での学習が遅くなっていることがわかる。したがって精度の面でも計算速度の面でも、活性化関数はシグモイド関数よりも ReLU の方が優れていることが確かめられた。

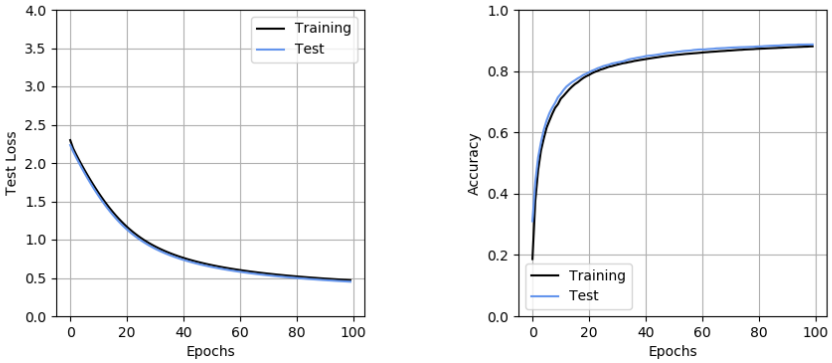

図 1.3　活性化関数をシグモイド関数に変えた場合の学習曲線

1.1.3　コスト関数の比較

節**??**の結果は

```
Test loss: 0.0511189009994268
Test accuracy: 0.706900000572205
Computation time: 29.949 sec
```

であった。実行時間が 0.162 sec だけ短くなったものの、正解率は 22.06% も下がって 70.69% となってしまった。学習曲線を図 1.4 に示す。コスト関数の値はほとんど横這いでわずかに小さくなっているだけである。また正解率は図 1.1 や図 1.3 と比較して直線的であり、エポック数が足りていないだけで学習を続ければ性能が上がるのではないかと考え、エポック数を 5 倍の 500 にして実行させた。その学習曲線の結果を図 1.5 に示す。正解率は図 1.1 や図 1.3 と同じように上昇している。しかし図 1.1 では正解率 80% に到達するのに 4 か 5 エポックしか必要としないのに対し、図 1.5 では同じ水準に達するのに 20 エポックほども必要であることが学習曲線から読み取れる。最終的な正解率は `Test accuracy: 0.898300009965897`、実行時間は `Computation time: 150.637 sec` であった。コスト関数が交差エントロピーの場合に比べておよそ 5 倍の実行時間を消費したにもかかわらず、正解率はおよそ 2.92% も下がっている。よってコスト関数としては、平均二乗誤差よりも交差エントロピーを用いる方が効率的に学習させることができるとわかった。

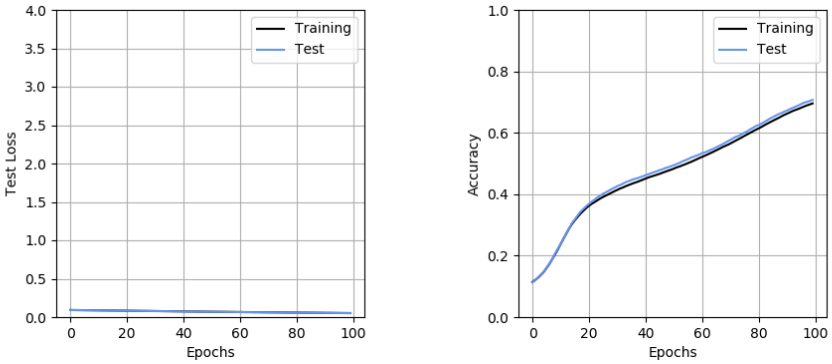

図 1.4　コスト関数を平均二乗誤差に変えた場合の学習曲線

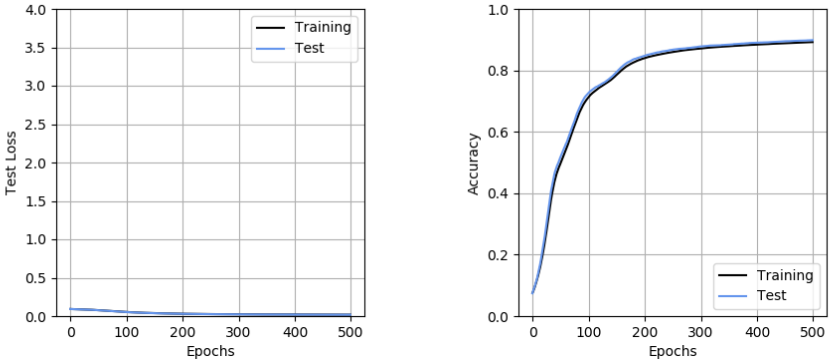

図 1.5　コスト関数を平均二乗誤差とし、エポック数を増やした場合の学習曲線

1.1.4　最適化アルゴリズムの比較

節**??**を実行したところ次の結果を得た。

```
Test loss: 0.106618618778884
Test accuracy: 0.97509999871254
Computation time: 31.240 sec
```

実行時間は 1.129 sec だけ長くなったものの、最適化アルゴリズムを Adam にしただけで、正解率はおよそ 4.76% も上がって 97.51% になった。図 1.6 の学習曲線を見ると、数回のエポックで正解率が 90% に達していることがわかる。エポック数が 40 を超えたあたりから訓練データに関してコスト関数はほどんど 0 に、正解率はほとんど 100% に近い値を保持し続けている。このことから 60000 個の訓練データに関しては 40 回程度のエポックでほとんど学習が完了していることがわかる。試しに先頭の 100 個のテストデータを予測させたところ、すべて正解であった（図 1.7）。この精度であれば人間の認識とほとんど同じである（人間にも読み間違いはあるため、正解率 100% が理想ではない）。

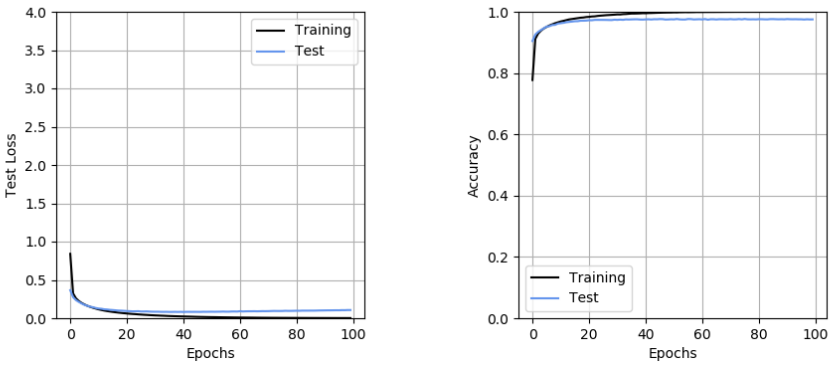

図 1.6　オプティマイザに Adam を採用した場合の学習曲線

図 1.7　Adam を採用した場合の先頭 100 個のテストデータに対するモデルの予測

1.1.5　エポック数、バッチサイズの変更

節**??**のようにエポック数とバッチサイズを変更した場合の実装結果を表 1.1 にまとめて示す。節**??**の基準となるモデルのものは太字で示した。また `Test loss` と `Accuracy` は小数点以下 4 桁までに四捨五入して表示してある。ミニバッチのサイズを 1000 に固定してエポック数を 10 倍ずつ増やしてゆくと、正解率が上がっていくことがわかる。また実行時間もエポック数に比例して増加している。ミニバッチのサイズが共通である場合にパラメータの更新回数はエポック数に比例することが現れている。またエポック数を 100 に固定してミニバッチのサイズを 10 倍ずつ増やしてゆくと、正解率は下がっていくことがわかる。また実行時間はおおよそミニバッチが増えてゆくにつれて減ってゆく。パラメータの更新回数はミニバッチのサイズに反比例することに由来していると考えられる。

表 1.1　エポック数、バッチサイズを変更したときの実装結果

エポック数	ミニバッチのサイズ	コスト関数	正解率	実行時間 / sec
10	1000	0.5729	0.8660	3.394
100	**1000**	**0.2595**	**0.9275**	**30.111**
1000	1000	0.0981	0.9704	310.089
100	100	0.0938	0.9715	76.001
100	**1000**	**0.2595**	**0.9275**	**30.111**
100	10000	0.5691	0.8622	28.952
100	100000	1.5860	0.6639	33.203

1.1.6　隠れ層を 2 層に増やした場合の性能評価

第 1 層目の隠れ層のユニット数は 100 のまま、追加した第 2 層目の隠れ層のユニット数を 10, 25, 50, 100, 200 のように変化させたとき、学習の結果は表 1.2 のようになった。総パラメータの数と実行時間はばらつきがあるものの、第 2 隠れ層のユニット数に比例しているように見える。コスト関数の最終的な値と正解率は第 2 隠れ層のユニット数に関わらずほとんど変わらない。

さらに第 1 隠れ層と第 2 隠れ層のユニット数を入れ替えて実験した。入れ替える前のものと比較したものを次の表 1.3 に示す。このことから、出力層に近い方の隠れ層のユニット数が、入力層に近い方の隠れ層のユニット数よりも大きい場合、実行時間はかなり短縮されるが、正解率が 1% から 2% ほど落ちてしまうことがわかる。したがって精度の良いニューラルネットワークを構築したいならば、漏斗の形のように、出力層に進むにつれて隠れ層のユニット数を小さくしていくことが必要であるだろう。また出力層の 1 つ前の隠れ層のユニット数が、出力層のそれと等しくても精度が良いとは限らず、数倍程度大きいときに正解率が大きくなっていることがわかる。

表 1.2　第 2 隠れ層のユニット数を変化させたときの学習結果

第 1 層	第 2 層	総パラメータ数	コスト関数	正解率	実行時間 / sec
100	10	79620	0.2084	0.9398	30.981
100	25	81285	0.2073	0.9411	35.246
100	50	83560	0.2120	0.9378	33.610
100	100	89610	0.2173	0.9370	35.841
100	200	100710	0.2250	0.9340	43.940

表 1.3　第 1 隠れ層と第 2 隠れ層のユニット数を入れ替えたときの学習結果の比較

第 1 層	第 2 層	総パラメータ数	コスト関数	正解率	実行時間 / sec
100	10	79620	0.2084	0.9398	30.981
10	**100**	**9960**	**0.2807**	**0.9210**	**19.633**
100	25	81285	0.2073	0.9411	35.246
25	**100**	**23235**	**0.2396**	**0.9310**	**22.033**
100	50	83560	0.2120	0.9378	33.610
50	**100**	**45360**	**0.2322**	**0.9361**	**24.118**
100	200	100710	0.2250	0.9340	43.940
200	**100**	**178110**	**0.2041**	**0.9406**	**56.645**

1.1.7　隠れ層を 3 層以上に増やした場合の性能評価

隠れ層の数 N と自然数 a に対して隠れユニット数を層の順に $10 \times a^N, \ldots, 10 \times a$ のように減らしていった「漏斗型」の多層ニューラルネットワークについて学習を行ったところ、表 1.4 のような結果が得られた。$a = 2,\ N = 6$ の場合に最も高い正解率 96.60% を得ることができた。しかしパラメータ数が最も多いためか、実行時間は 192.090 sec と長いことがわかる。全体的に a の値が小さい、すなわち隠れユニット数が急激に減少しないネットワークで正解率は高くなるようだ。

さらに $a = 2,\ N = 6$ の場合で隠れユニット数を逆転させた「逆漏斗型」ニューラルネットワークと隠れ層は 6 層で各層 210 のユニットを持つ「寸胴型」ニューラルネットワークについても学習を行わせたところ、表 1.5 の結果を得た。「逆漏斗型」の正解率は「漏斗型」に比べて 3% 程度も落ちている。実行時間は「漏斗型」が 192 sec ほどであったのに対し、「逆漏斗型」「寸胴型」はともに 2 分程度になっている。これはどちらもパラメータ数がおよそ半分程度に少なくなっているからだと考えられる。「寸胴型」の正解率 96.37% は「漏斗型」の正解率 96.60% とさほど変わらないが、それでも 0.23% 小さくなっている。したがって多層ニューラルネットワークにおいては、出力層に向かって隠れユニットの数が小さくなってゆけば正解率が高くなることがわかった。

表 1.4　隠れユニット数を $1/a$ 倍ずつ減らしていった多層ニューラルネットワークの学習結果

a	N	総パラメータ数	コスト関数	正解率	実行時間 / sec
2	6	776030	0.1133	0.9660	192.090
3	3	239380	0.1674	0.9504	67.056
4	3	611810	0.1454	0.9558	134.113
5	2	209310	0.1980	0.9422	58.341
6	2	304870	0.1895	0.9451	72.679
7	2	419730	0.1842	0.9472	94.051
8	2	554490	0.1849	0.9465	116.425

表 1.5　「逆漏斗型」と「寸胴型」ネットワークによる学習結果

アーキテクチャ	総パラメータ数	コスト関数	正解率	実行時間 / sec
「逆漏斗型」	296150	0.2044	0.9379	121.451
「寸胴型」	388510	0.1178	0.9637	124.980

1.1.8　ドロップアウトによる効果

節**??**のドロップアウト についての検証を行う。まず　ドロップアウト率 $p = 0.25$ で試したところ、図 1.8 の学習曲線を得た。しかしテストデータによる正解率が訓練データによるそれよりも上回っており、テストデータの方が性能が良く現れている。このことは学習があまり進んでおらず、数の少ないテストデータの学習の方が先に学習が進んでいること考えられる。学習を早めるため、オプティマイザに Adam を選ぶと図 1.9 のようになって適切に学習ができていることがわかる。ただし見やすさのため図 1.9 は縦軸の表示範囲を、左側のコスト関数のグラフについては 0 から 1，右側の正解率のグラフについては 0.8 から 1 のように変えてある。

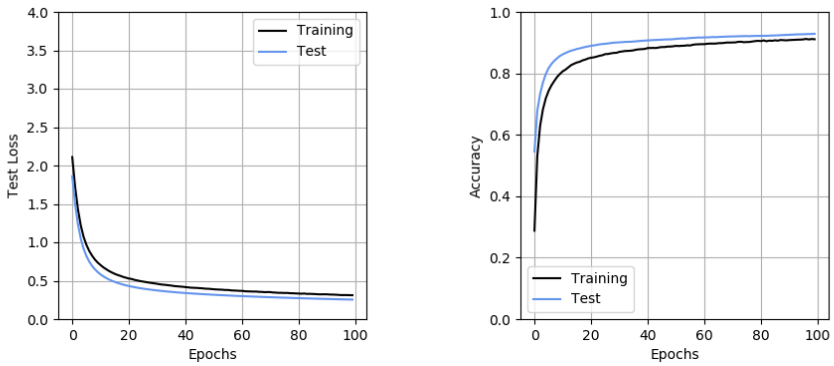

図 1.8　学習が遅い場合の学習曲線

このようにオプティマイザをすべて Adam に変更して、ドロップアウト率を $p = 0$（ドロップアウトをしない），0.25，0.5，0.75 と変化させて学習させると次の表 1.6 の結果を得た。ドロップアウトを入れないよりも、$p = 0.25$ で入れた方がわずかに性能が良くなっている。$p = 0.25$ のときの性能が最も良いことがわかる。なお $p = 0.75$ のとき学習曲線は図 1.10 のようになり、学習が完了していないことがわか

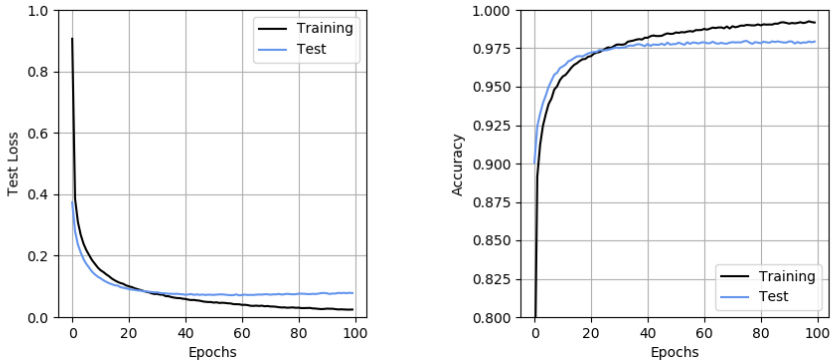

図 1.9　オプティマイザを Adam にして学習させたときの学習曲線

る。エポック数を大きくするなど対処しても、訓練データの誤差は小さくなってゆくが、テストデータの誤差が増加に転じてしまった（図 1.11）。これは訓練データのみに適合しすぎて過学習が起きているためである。ドロップアウト率が大きい時はうまく学習させることができなかった。したがってドロップアウト率は $p = 0.25,\ 0.5$ 程度で良いことがわかった。

表 1.6　ドロップアウト率に対する性能の実験結果

p	エポック数	ミニバッチのサイズ	コスト関数	正答率	実行時間 / sec
0	100	1000	0.0939	0.9778	31.083
0.25	100	1000	0.0781	0.9794	37.676
0.50	100	100	0.1069	0.9776	95.312
0.75	100	100	0.1403	0.9639	93.350

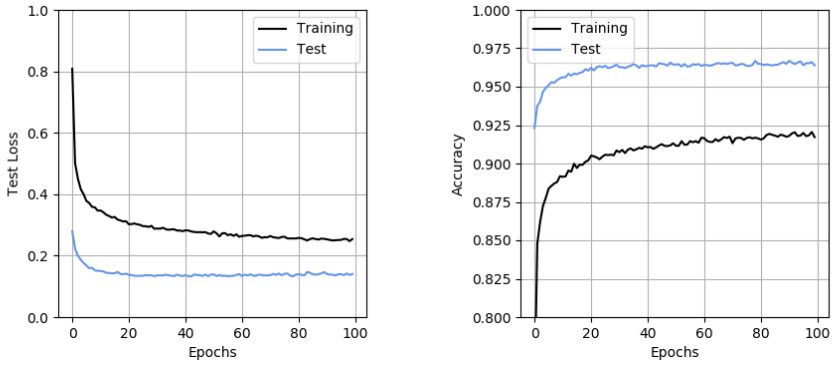

図 1.10　$p = 0.75$,　エポック数 100 のときの学習曲線

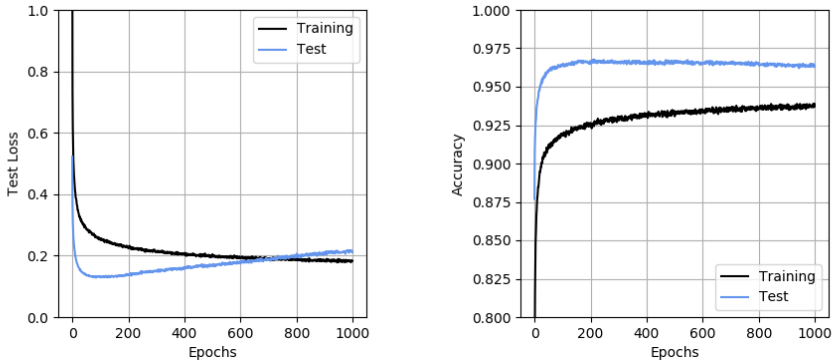

図 1.11　$p = 0.75$,　エポック数 1000 のときの学習曲線

1.2　畳み込みニューラルネットワークによる学習

1.2.1　ベースとなる畳み込みニューラルネットワークの構築

節**??**の結果を記す。学習をさせたところ、次の結果を得た。

```
Test loss: 0.327157819122076
Test accuracy: 0.909
Computation time:56.841 sec
```

単純なモデルであるが実行時間に 56.841 sec かかった。また正解率は 90.9% であり、これは節**??**のフィードフォーワード・ニューラルネットワークの正解率 92.75% よりも低い。パラメータ数についての結果は以下のようになった。

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 1)         10
_________________________________________________________________
flatten (Flatten)            (None, 784)               0
_________________________________________________________________

dense (Dense)                (None, 10)                7850
=================================================================
Total params: 7,860
Trainable params: 7,860
Non-trainable params: 0
_________________________________________________________________
```

総パラメータ数は節**??**のフィードフォーワード・ニューラルネットワークのそれよりに比べておよそ 1/10 に少なくなっているが、正解率が低いのはこの影響があると考えられる。学習曲線は次の図 1.12 の

ようになった。コスト関数の値、正解率はそれぞれ単調に減少、増加しており、過学習はしていないことが確認できる。試しに初めの 100 個のテストデータに対するモデルの予測を表示させたものを次の図 1.13 に示す。ここでは 100 個のうち 95 個が正解している。図 1.2 と共通して間違えているものが 3 つあるのが確認できる。1 を 3 と間違えていたり、9 を 4 と答えている例がある。これらの数字は字形大まかに似ていて、たった 1 枚のフィルターで学習させたため仕方ないが、まだ精度は不十分である。しかし 3×3 のフィルター 1 枚だけでも正解率は 9 割りを超えており、フィルター数を増やしたりプーリング層を加えたりすることでさらに改善されることが期待できる。

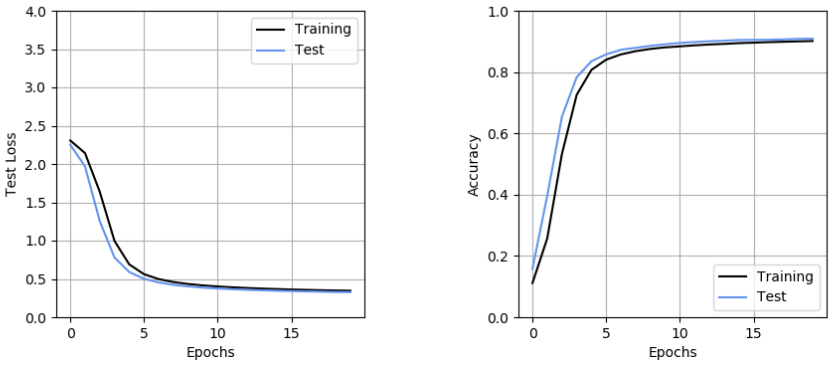

図 1.12　3×3 のフィルター 1 枚による畳み込みニューラルネットワークの学習曲線

図 1.13　先頭 100 個のテストデータに対する畳み込みニューラルネットワークの予測

1.2.2　フィルターのサイズと枚数の比較

節**??**の結果を記す。まずフィルターのサイズはすべて 3×3 のまま、フィルター数を 2, 3, 4, 5, 6 のように変えて学習させたときの結果を次の表 1.7 に示す。パラメータ数はフィルター数におおよそ比例している。また正解率はこの状況においてはわずかに増加していることがわかる。

表 1.7　フィルター数を増加させたときの実行結果

フィルター数	総パラメータ数	コスト関数	正解率	実行時間 / sec
2	15710	0.3385	0.9029	65.392
3	23560	0.3044	0.9145	67.026
4	31410	0.3069	0.9144	72.644
5	39260	0.3074	0.9159	77.334
6	47110	0.3026	0.9144	78.914

またフィルター数は 1 のまま、フィルターのサイズを 4×4, 5×5, 6×6 と変化させたとき、実行結果は表 1.8 のようになった。フィルターサイズを大きくさせても、正解率はほとんど同じで、実行時間だけが延びることがわかる。したがってフィルターはサイズが小さいものを用いれば、効率が良いことが実験

的にわかった。

表 1.8　フィルターサイズを変化させたときの実行結果

フィルターサイズ	総パラメータ数	コスト関数	正解率	実行時間 / sec
4×4	7867	0.3218	0.9069	97.402
5×5	7876	0.3554	0.8984	139.740
6×6	7887	0.3272	0.9084	191.790

1.2.3　プーリング層の追加

節**??**の検証結果を記す。まずプーリングカーネルのサイズが 2×2 の最大プーリングを行ったときの結果は次のようになった。これを見ると、プーリングを行うだけでは精度が悪いことがわかる。

```
Test loss: 0.823955835819244
Test accuracy: 0.8323
Computation time:8.764 sec
```

平均プーリングを行った結果は次のようになった。実行時間は変わらないものの、正解率がわずかに下がっていることがわかる。またコスト関数の値も最大プーリングのときよりも大きい。これらのことから最大プーリングの方が効率がよいと考えられる。

```
Test loss: 1.0449662100791932
Test accuracy: 0.8018
Computation time:8.782 sec
```

さらにプーリングカーネルのサイズを変えて学習を行うと、表 1.9 のような結果を得た。プーリングカーネルのサイズが大きくなると出力の特徴マップが縮小するのでパラメータ数が減少している。また正解率は大きく減少することがわかる。これも特徴マップが小さくなるためである。したがってプーリングを行うときは、小さいプーリングカーネルで最大プーリングを行うのがよいことがわかった。

表 1.9　プーリングカーネルのサイズと対する実行結果

プーリングカーネル	総パラメータ数	コスト関数	正解率	実行時間 / sec
2×2	1970	0.8240	0.8323	8.764
3×3	820	1.1241	0.7773	8.079
4×4	500	1.4857	0.6395	8.815

1.2.4　LeNet-5 の実装

LeNet-5 を実装したところ、次の結果を得た。

```
Test loss: 0.186557103292644
```

```
Test accuracy: 0.9436
Computation time:256.474 sec
```

実行時間は 4 分半ほどかかったものの、畳み込み層、プーリング層が 1 層のときに比べてかなり精度が良くなっている。学習曲線は図 1.14 のようになり、過学習は起きていないことがわかる。

また畳み込み層 C5 と F6 の間、F6 と出力層の間、そのどちらにも $p = 0.25$ のドロップアウトを挿入した場合について性能を出力させると次の表 1.10 のようになった。両方に挿入すると性能が 0.63% 落ちた。また片方のみに挿入すれば、性能が若干上がっている。これは出力層から遠い層のものほどユニットが多いため、効率的にドロップアウトを用いられているためと考えられる。しかし多くの層の間でドロップアウトを行うと学習に時間がかかり、効率が悪くなる。

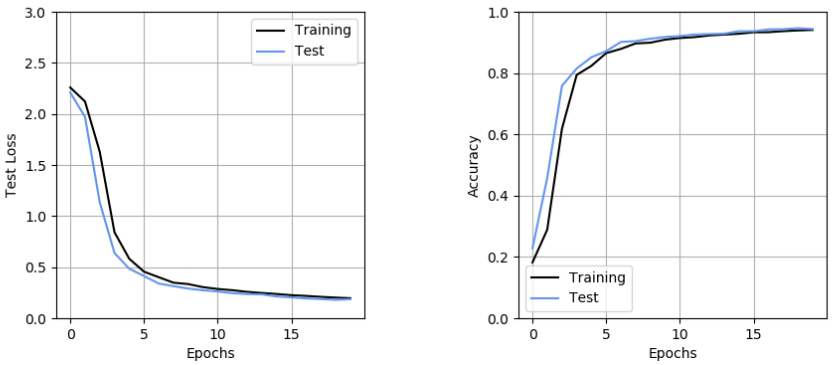

図 1.14　LeNet-5 の学習曲線

表 1.10　ドロップアウトを挿入したときの結果

ドロップアウトを挿入した場所	コスト関数	正解率	実行時間 / sec
(挿入しない)	0.1866	0.9436	256.474
C5—F6 間	0.1744	0.9473	258.315
F6—出力層間	0.1876	0.9437	264.223
両方	0.2120	0.9373	256.262

1.2.5　畳み込みニューラルネットワークのアーキテクチャ

節**??**の 2 つのネットワーク A と B について学習を行うと表 1.11 の結果を得た。どちらの畳み込みネットワークでもさほど変わらない正解率を得たが、ネットワーク B の実行には A の約 6.5 倍にあたる 34 分もかかってしまった。したがって畳み込みニューラルネットワークでは出力層に向かってフィルター数の増加していく「末広がり」の形をしたアーキテクチャで効率よく学習させることができるとわかった。

表 1.11　ネットワーク A と B についての実行結果

ネットワーク	総パラメータ数	コスト関数	正解率	実行時間 / sec
A	74122	0.2199	0.9353	311.363
B	260122	0.2121	0.9396	2041.787