深層学習day1

入力層～中間層

入力層はm次元の入力値**x**をとりうる。
中間層はm次元の入力を受け取り、j次元の出力値を出力する。この際の出力値は、入力値の線形結合と活性化関数による変換とで生成する。

・入力　　：$\mathbf{x} = (x_1, x_2, \cdots, x_m)$
・重み　　：$\mathbf{W} = (\mathbf{w_1}, \mathbf{w_2}, \cdots, \mathbf{w_j})$
　　ただし$\mathbf{w_i} = (w_{i1}, w_{i2}, \cdots, w_{im})^T$
・バイアス　：$\mathbf{b} = (b_1, b_2, \cdots, b_j)$
・総入力　　：$\mathbf{u} = \mathbf{xW} + \mathbf{b}$
・出力　　：$\mathbf{z} = f(\mathbf{u})$
・活性化関数f　：ReLU関数、シグモイド関数、…など

中間層の出力が次の中間層の入力となることで、中間層は幾重にも連なり得り、この層が深い学習モデルを深層学習と呼ぶ。

■実装演習
　3次元の入力を受け取り、中間層3層にわたって伝播する例。
　中間層では出力を10次元→5次元→4次元　と変化させている。

```
# 順伝播（3層・複数ユニット）

# ウェイトとバイアスを設定
# ネートワークを作成
def init_network():
    print("###### ネットワークの初期化 ######")
    network = {}

    input_layer_size = 3
    hidden_layer_size_1=10
    hidden_layer_size_2=5
    output_layer_size = 4

    #試してみよう
    #_各パラメータのshapeを表示
    #_ネットワークの初期値ランダム生成
    network['W1'] = np.random.rand(input_layer_size, hidden_layer_size_1)
    network['W2'] = np.random.rand(hidden_layer_size_1,hidden_layer_size_2)
    network['W3'] = np.random.rand(hidden_layer_size_2,output_layer_size)

    network['b1'] =  np.random.rand(hidden_layer_size_1)
    network['b2'] =  np.random.rand(hidden_layer_size_2)
    network['b3'] =  np.random.rand(output_layer_size)
```

■考察
単なる線形結合であるが、任意の数のノードと中間層をモデルとして設定することで、出力結果がある程度コントロールできる。深層学習はこの単純な演算の組み合わせで、あらゆる出力を生み出そうという発想なのだと思う。目からウロコであった

活性化関数

ニューラルネットワークにおいて、各層では入力をまず線形結合した値を生成する。その値を活性化関数という任意の非線形関数で変換することで、ニューラルネットワークの表現力が格段に向上する。

なお、学習を進めるためには勾配消失問題が起きにくい活性化関数を選択することが重要である。中間層ではReLU関数が、出力層では回帰問題の場合は恒等関数、分類問題の場合はソフトマックス関数が活性化関数として利用される事が多い。

■実装演習

1層目、2層目の総出力ではReLU関数を、出力層の総出力では恒等変換を利用した例

```
# 1層の総入力
u1 = np.dot(x, W1) + b1

# 1層の総出力
z1 = functions.relu(u1)

# 2層の総入力
u2 = np.dot(z1, W2) + b2

# 2層の総出力
z2 = functions.relu(u2)

# 出力層の総入力
u3 = np.dot(z2, W3) + b3

# 出力層の総出力
y = u3
```

■考察

入力に重みパラメータを掛け合わせた線形結合結果に対し、活性化関数という非線形変換を取り入れることによりモデルの表現力が向上する。これは直感的にも理解しやすいが、数学的には万能近似定理によって、任意の表現力を持つことが証明されている。単純な演算の積み重ねであるが任意の表現力を持つとは驚きである。深層学習は、それ以外の機械学習と比較して、特徴量選択をエンジニアが行う必要がない点が特徴的であると感じた。

出力層

出力層では最後の中間層からの出力を受け取り、任意の次元の出力値に変換する。この際の活性化関数として、分類問題ではソフトマックス関数を使って確率値に変換し、回帰問題は信号の強さを恒等関数でそのまま出力値とする。

また、教師有学習の学習フェーズにおいては、出力値と教師データとの差異の大きさを適切な誤差関数で定義する。学習においては、この誤差関数の出力値が小さくなるように、ニューラルネットワークの各パラメータを更新する。

誤差関数…二乗誤差（回帰問題）、クロスエントロピー（分類問題）

■実装演習（多クラス分類の出力例）

ソフトマックス関数で出力値を変換

```
# プロセスを作成
# x：入力値
def forward(network, x):

    print("##### 順伝播開始 #####")
    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']

    # 1層の総入力
    u1 = np.dot(x, W1) + b1

    # 1層の総出力
    z1 = functions.relu(u1)

    # 2層の総入力
    u2 = np.dot(z1, W2) + b2

    # 出力値
    y = functions.softmax(u2)
```

クロスエントロピーを誤差関数として、出力値と教師データとの誤差の大きさを評価

```
# ネットワークの初期化
network =  init_network()

# 出力
y, z1 = forward(network, x)

# 誤差
loss = functions.cross_entropy_error(d, y)
```

■考察

任意のノードから入力を受け取り、エンジニアが望む数だけの出力を出すのが出力層である。この入力数と出力数の関係を任意に設定できる点で、行列演算は非常によくできた変換処理だと感じる。なお上記例のようにソフトマックス関数の損失関数としてクロスエントロピーを用いると、逆伝播がキレイな形になる。それはそうなるようにクロスエントロピーが定義されているからである。

勾配降下法

「学習＝誤差関数が最小となる重みとバイアスを見つけること」であるが、ニューラルネットワークにおいて、これを解析的に行うことは困難である。そこで損失関数Eを各パラメータで更新した勾配∇Eを用いて、以下の数式により損失関数が小さくなる方向にパラメータを少しずつ更新する。

重みパラメータの更新）

$w^{(t+1)} = w^{(t)} - \varepsilon \nabla E$

ただし∇E：Eのwでの偏微分、 ε ：学習率、t：学習回数

バイアスの更新）

$b^{(t+1)} = b^{(t)} - \varepsilon \nabla E$

ただし∇E：Eのbでの偏微分、 ε ：学習率、t：学習回数

なお、学習の際に毎回全データを使って学習を行う必要はない。各学習回ごとにランダムに抽出した標本データを用いて学習を行うことが出来、これを確率的勾配効果法（SGD）と呼ぶ。

SGDのメリット

・計算コスト削減による学習の高速化
・局所最適化解の回避
・オンライン学習が可能

■実装演習

各パラメータにおいて勾配を求め、パラメータ更新を実施。

勾配算出はbackward関数で実施しており、実装として数値微分と誤差逆伝播の2通りが考えられる（口述）

```
# 学習率
learning_rate = 0.01
network =  init_network()
y, z1 = forward(network, x)

# 誤差
loss = functions.cross_entropy_error(d, y)

grad = backward(x, d, z1, y)
for key in ('W1', 'W2', 'b1', 'b2'):
    network[key]  -= learning_rate * grad[key]
```

■考察

構築したモデルが表現する関数の最小値を見つけることは解析的に困難であり、現実的な選択肢ではないのだと思う。そこでコンピュータの演算力を利用した勾配降下法という手法が考え出されたことに対し、まさに妙案だと感じた。誤差関数の最小化というたった一つの目的に向かい、少しずつパラメータを動かせばよいとは、まさにコンピュータだからこそなせる技である。

誤差逆伝播法

誤差関数のパラメータによる微分値を求める際、カンタンな実装としては数値微分の手法がある。つまり、パラメータを微小量だけ変化させたときの誤差関数の変化値を実際に求めることで微分を行う。

　この手法は実装はカンタンであるが、計算コストが高く、結果は近似値であり誤差を含むので、通常は誤差逆伝播法により各パラメータの微分値を求める。誤差逆伝播法では、各計算処理における導関数を解析的に求め、それをかけ合わせることによって、目的のパラメータの微分を行う方法である（微分の連鎖率）。

■実装演習

下記ソース例においては、微分の連鎖率により各パラメータの微分が行われている。

例えば赤枠部、活性化関数における微分値「delta2」がパラメータ「W2」の微分値算出時に掛け合わされている。

```
    # 誤差逆伝播
    def backward(x, d, z1, y):
        print("¥n###### 誤差逆伝播開始 ######")

        grad = {}

        W1, W2 = network['W1'], network['W2']
        b1, b2 = network['b1'], network['b2']
        #  出力層でのデルタ
        delta2 = functions.d_sigmoid_with_loss(d, y)
        #  b2の勾配
        grad['b2'] = np.sum(delta2, axis=0)
        #  W2の勾配
        grad['W2'] = np.dot(z1.T, delta2)
        #  中間層でのデルタ
        delta1 = np.dot(delta2, W2.T) * functions.d_relu(z1)
        # b1の勾配
        grad['b1'] = np.sum(delta1, axis=0)
        #  W1の勾配
        grad['W1'] = np.dot(x.T, delta1)
```

■考察

ニューラルネットワークはいくつもの関数からなる巨大な合成関数だと言える。そう考えると、各パラメータでの微分値の算出に、微分の連鎖率を使うという発想は至極当然ではある。しかし、これを実装したプログラムのコード見ると、1つの微分が複数の微分に分解されることで、プログラミングがすごく簡素になり、単純化されていることに驚いた。

深層学習day2

勾配消失問題

各パラメータの勾配を誤差逆伝播法で求めるに当たり、入力層に近いほど、幾重にも微分値が掛け合わされ。そのため活性化関数に微分値の絶対値が1未満の関数を使用した際、入力層側ではそれらが何回も掛け合わされ、ほぼ勾配がゼロとなり（勾配消失）、学習が進まないという問題が起こった。特にシグモイド関数においては微分値の最大値が0.25であり、勾配消失が顕著であった。

勾配消失の回避方法

①活性化関数の選択…0以上の入力に対して勾配が常に1となるLeRU関数が現在の主流

②重みの初期値設定

重みパラメータの初期値によっては各層の出力が0 or 1の2値化してしまい、勾配がほぼ0となる。

そこで用いる活性化関数により、下記の初期値設定手法を用いる

シグモイド関数利用

…Xavierの手法。

1)重みパラメータを標準正規分布よりランダムに設定。

2)前の層のノード数の平方根で除算する

ReLU関数利用

…Heの手法

1)重みパラメータを標準正規分布よりランダムに設定。

2)前の層のノード数の平方根で除算し、さらに2の平方根をかける

③バッチ正規化

各層においてバッチ単位で、入力が標準正規分布に従うように変換（正規化）する。入力値がバリエーションをもつように強制的に変換をするので、勾配消失が起こりにくく、学習を早く進行させることができる。

■実装演習

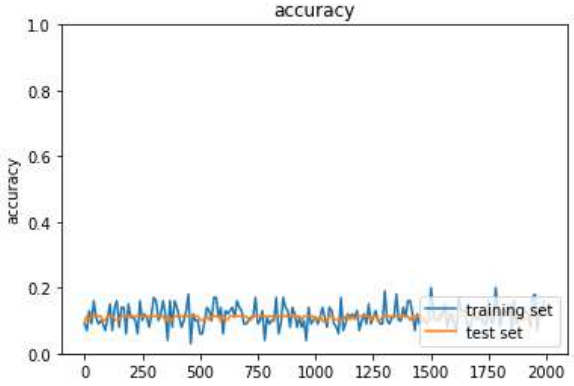
前提：

MNISTデータを中間層2つのニューラルネットワークでクラス分類。

活性化関数と重みパラメータの初期値を変えた際の、学習の進み方（正解率向上）を比較

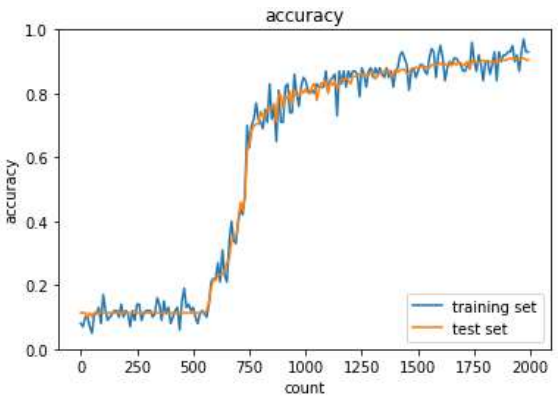
1)活性関数：シグモイド関数、重み：標準正規分布よりランダムサンプリング

正解率は低いまま、学習が進まない。



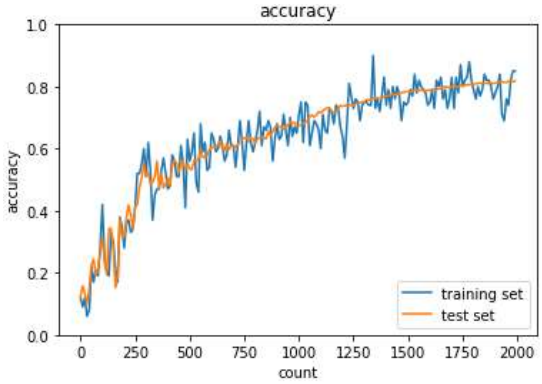
2)活性関数：ReLU関数、重み：標準正規分布よりランダムサンプリング

500回目移行の学習から、正解率が上昇し学習できた。



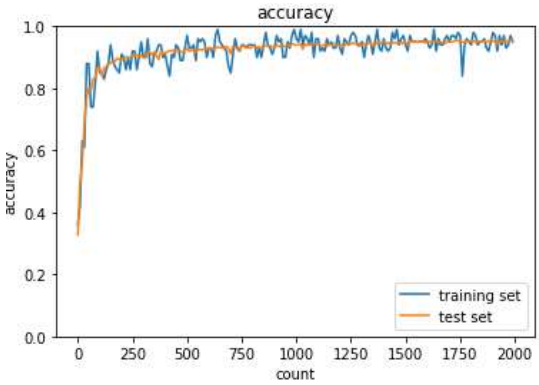
3)活性関数：シグモイド関数、重み：Xavier初期化

勾配消失が起きやすいシグモイド関数でも、学習が徐々に進んだ。



3)活性関数：ReLU関数、重み：He初期化

最も早く学習が進んだ。



■考察

重みの初期値設定について考える。活性化関数の入力値は前層のノードの出力の線形結合である。そのためノード数分だけ分散が大きくなる（絶対値が大きい値をとる確率が高くなる）。そしてシグモイド関数やReLU関数といった活性化関数の入力が0から離れると、シグモイド関数は0 or 1の近似値を出力、ReLUは0 or 大きな正の値を出力、と出力も偏る。出力の偏りは表現力の低下につながり、重みの初期値が不適切だと学習が進まないのだと考える。そこでノード数分だけ大きくなった分散をXavierではノード数の平方根で除すことで、1に正規化しているのだと推察する。

学習率最適化手法

学習率はハイパーパラメータであるが、その設定値が学習の成否に大きく影響する。学習率が適切でない場合、以下のような問題起こる。

学習率が大きすぎる…各パラメータが最適値から遠ざかってしまう（発散）。

学習率が小さすぎる…学習の進みが遅い。最適値ではなく局所最適値で学習が止まる。

以上のような問題を解消するために下記のような学習率最適化手法が考案された

1)Momentum

各回のパラメータ更新量をV_tとすると、今回の更新量に前回の更新量V_{t-1}を一定割合（慣性）で混ぜ合わせる。つまり更新量の移動平均を用いることで、局所最適解で学習が止まることを防ぐ。

2)AdaGrad

各回の学習率を、学習量の二乗の累積の平方根で徐する。学習が進むほど学習率が小さくなるので、発散を防ぐことができる。

3)RMSProp

学習が進むほど学習率を小さくするという点ではAdaGradと同じ。学習率を除する値として、「学習量の二乗の累積の平方根の移動平均」を用いることで、AdaGradに比べ、局所最適解で学習が止まることを防ぐ。

4)Adam

RMSPropとMomentumを合成した手法、つまりごく単純化して説明すると、更新量の移動平均を学習量の累積で除する。結果的にこの手法が最も学習が進みやすく現在よく使われている。

■実装演習

前提：

MNISTデータを中間層2つのニューラルネットワークでクラス分類。

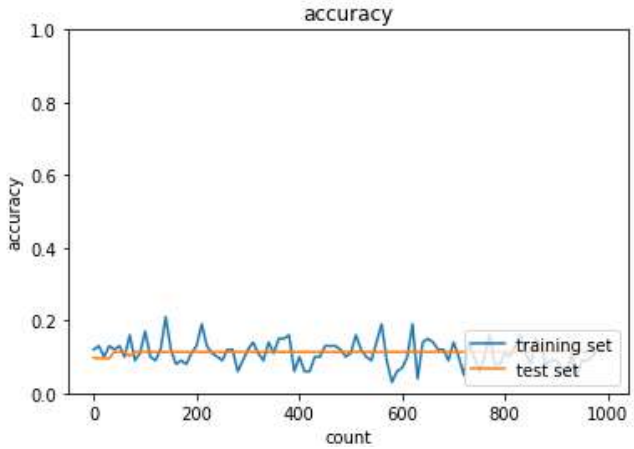
学習率最適化手法を変えた際の、学習の進み方（正解率向上）を比較。活性関数はシグモイド関数。

結果

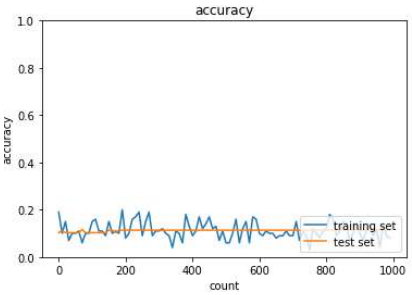
学習率固定では全く進まなかった学習が、RMSPropとAdamを利用することで進むようになった。

なお、活性化関数としてReLUを使うと、MementumとAdaGradでも学習が進んだ。

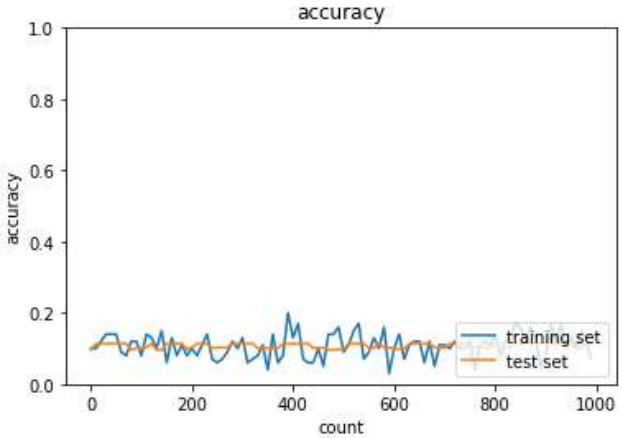
1)学習率固定



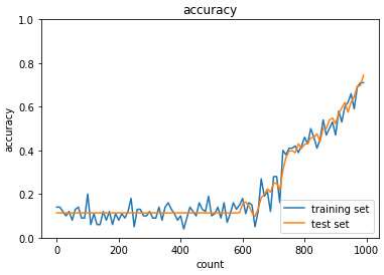
※↓活性関数：ReLU



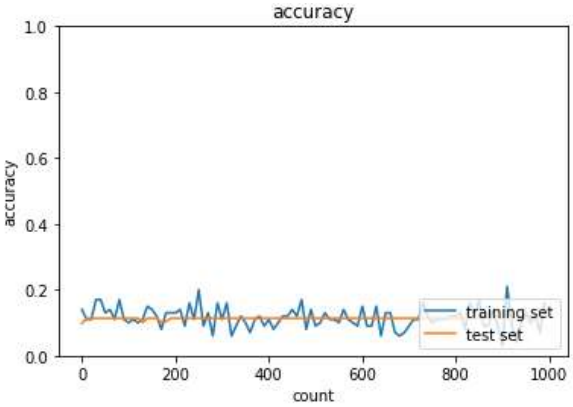
2)Momentum



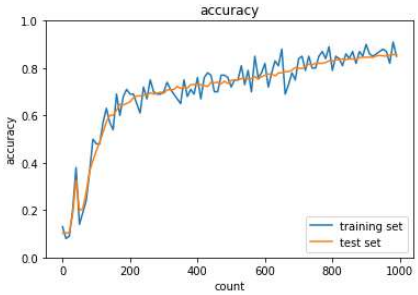
※↓活性関数：ReLU



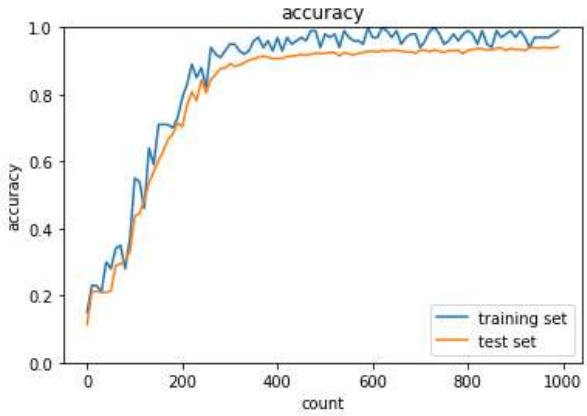
3)AdaGrad



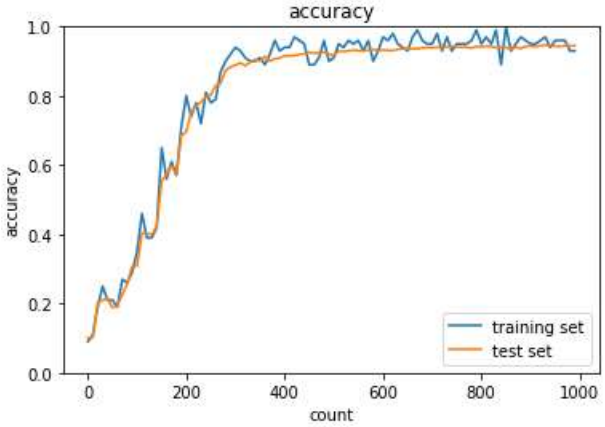
※↓活性関数：ReLU



4)RMSProp



5)Adam

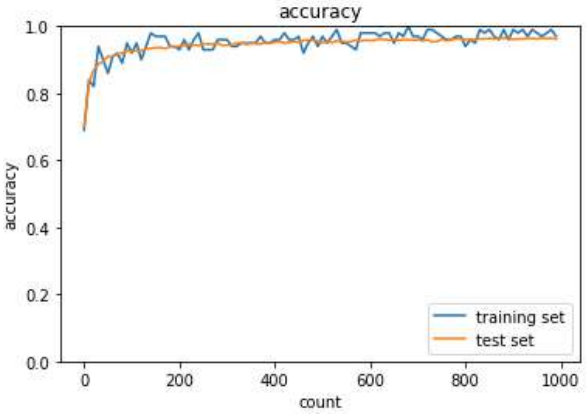


■考察

活性化関数がシグモイド関数の時、Mementumでは進まなかった学習が、ReLU関数を使用することでMomentumでも進むようになった。学習を上手く成功させるためには、学習率最適化手法に加え、活性化関数、正則化手法、パラメータの初期値の設定など、適切なものを選択する必要がある。この選択がエンジニアの腕の見せ所であり、そのためには各手法の理解が必要だと分かった。

▼Adam、ReLU関数、He初期化、バッチ正規化を全て組み合わせ

上図に比べ、非常に速く学習が進んだ。



過学習

予測モデルが教師データに過剰に適合したため、未知のテストデータへの予測精度が十分に上がらないことを過学習という。これはモデルの説明力がデータに対して柔軟すぎるため、教師データの誤差項まで説明しようと学習してしまうことに起因する。

過学習への対策

1)正則化…重みパラメータのノルムを誤差関数に足し合わせる。重みパラメータが大きいと誤差関数も大きくなる。重みパラメータの過大化、すなわちモデルの表現力が抑えられる。

2)ドロップアウト

各学習回ごとに、ランダムにノードを削除して学習させること。ノード削除により、モデルの表現力が過剰になることが抑えられる。学習回ごとに異なるモデルで学習していると言え、アンサンブル学習の効果があると考えられている。

■実装演習

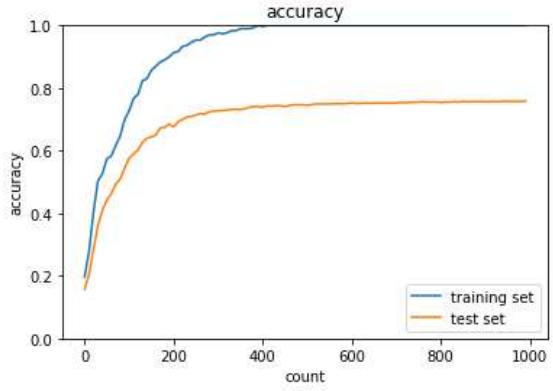
前提：

MNISTデータを中間層6つ、各ノード数100のニューラルネットワークでクラス分類。

目的に対して、モデルが過剰な表現力を有している状態。

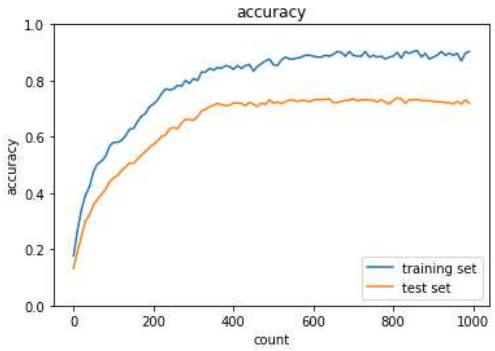
1)過学習を起こした例

教師データの正解率100%に対し、テストデータの正解率は60%強で止まっている。



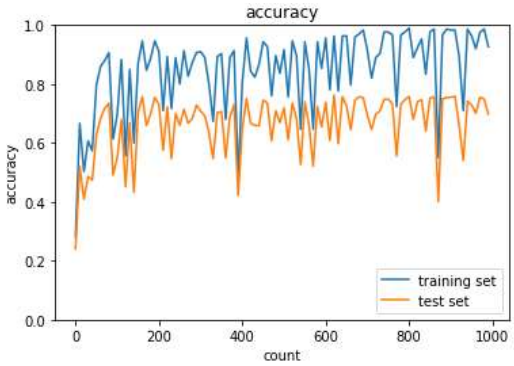
2)L2正則化、正則化の強さ λ =0.1

教師データとテストデータとで正解率の乖離は抑えられた。



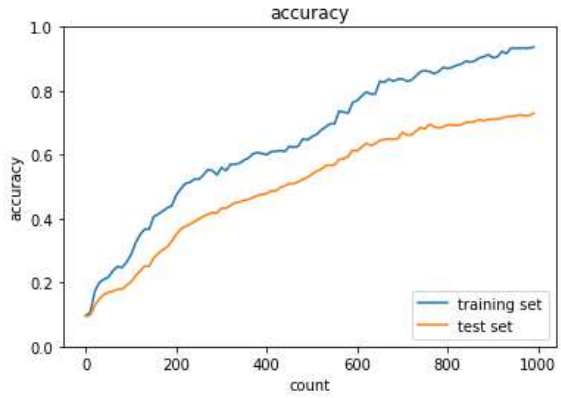
3)L1正則化

教師データとテストデータとで正解率の乖離は抑えられた。学習の進み方が不安定



4)ドロップアウト

過学習が低減



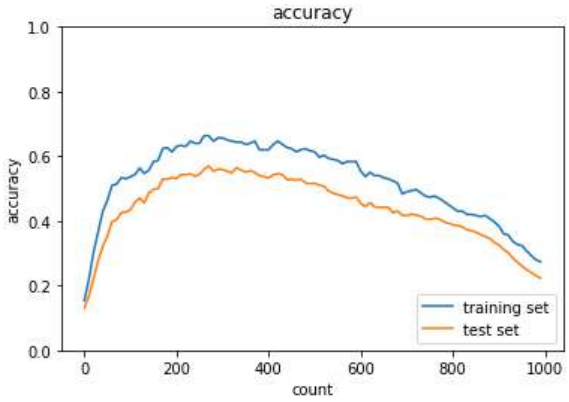
■考察

上記実装例では、テストデータと教師データとの正解率の乖離を抑えることはできたが、テストデータに対する正解率はいずれも60%強で頭打ちしており、汎化性能向上はできていない。

これを解消すべく、正則化のハイパーパラメータを色々と変え、さらにバッチ正規化の手法も組み合わせたが、解消することは出来なかった。中間層の数とノード数を過剰にした場合、そもそもその値を適正化しないことには、汎化性能向上は難しいことがわかった。

また下記の通り、正則化の強さを強くし過ぎると、学習回毎に逆に正解率が下がる現象も確認され、興味深かった。

▼L2正則化、正則化の強さ λ =0.2



畳み込みニューラルネットワークの概念

これまでは全結合によるニューラルネットワークを学んできたが、畳み込みニューラルネットワークではそれに加え、畳み込み層とプーリング層が前段階で存在する。

　入力→畳み込み層→プーリング層→全結合層→出力

入力データの連続性を考慮することで、画像データの扱いに長けている。画像データが畳み込み層、プーリング層を通過すると、画像のエッジなどのデータの連続性による特徴量が抽出される。

・畳み込み層

　全結合層はデータの連続性を全く無視しているが、畳み込み層ではそれを考慮する。各チャネルごとにフィルタと重みパラメータを用意して、データの連続性を加味しながら入力値を変換する。

・プーリング層

入力に対し、任意の指定領域ごとに最大値、もしくは平均値を取得する。学習するパラメータをは存在しない。

■実装演習

畳み込み演算は定義通りに行うと幾重にもfor文を重ねたような実装になる。そこで実装ではim2colというフィルタの適用先ごとに展開処理をする関数を定義する。処理が行列演算となり簡素化される。

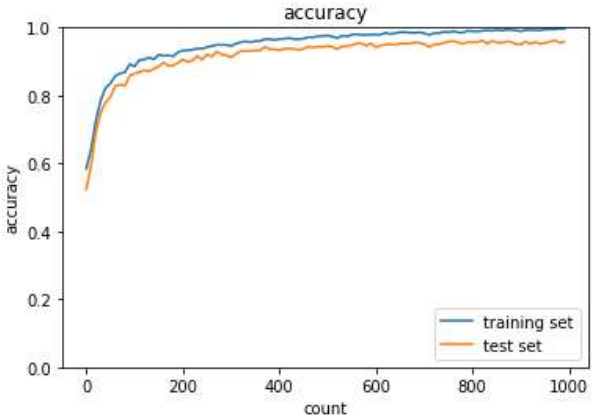
```
========== input_data ===========
 [[[[ 1. 12. 75.  6.]
    [19. 34. 80. 67.]
    [66. 83. 41. 91.]
    [82. 18. 24. 74.]]]


 [[[61.  6. 38.  5.]
    [62. 71. 75.  2.]
    [19.  5. 26. 61.]
    [80. 96. 84.  5.]]]]
==============================
============= col ==============
 [[ 1. 12. 75. 19. 34. 80. 66. 83. 41.]
 [12. 75.  6. 34. 80. 67. 83. 41. 91.]
 [19. 34. 80. 66. 83. 41. 82. 18. 24.]
 [34. 80. 67. 83. 41. 91. 18. 24. 74.]
 [61.  6. 38. 62. 71. 75. 19.  5. 26.]
 [ 6. 38.  5. 71. 75.  2.  5. 26. 61.]
 [62. 71. 75. 19.  5. 26. 80. 96. 84.]
 [71. 75.  2.  5. 26. 61. 96. 84.  5.]]
==============================
```

実施例

前提：

　MNISTデータを畳み込みニューラルネットワークでクラス分類。

　畳み込み層：1層、プーリング層：1層、全結合層：2層



■考察

今までの全結合層ニューラルネットワークでもMNISTの分類は行えていたが、畳み込み処理を加えることでも上手く分類できることが確認できた。正解率の上昇の仕方が安定している点で、学習成否の判断がしやすいと感じた。しかしどうしても演算コストは相対的に大きいため、1回の学習にかかる時間は数倍になったと感じた。MNISTのデータに対しては畳み込みニューラルネットワークはやや過剰な実装なのかもしれない。

最新のCNN

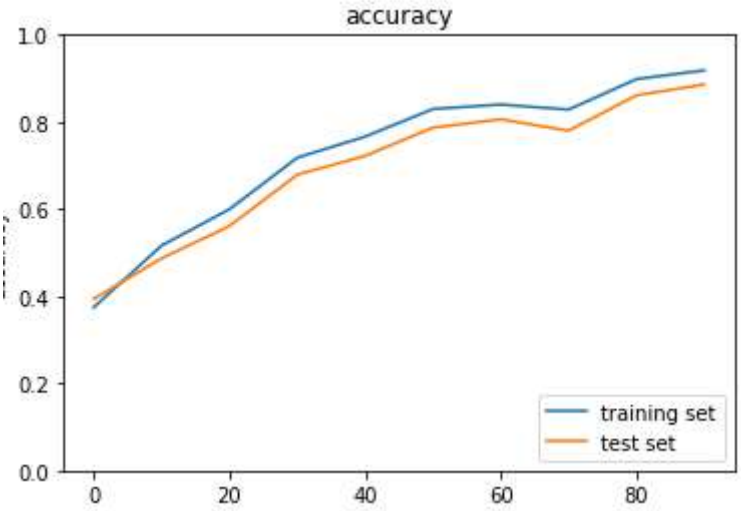
ディープラーニングブームの火付け役になったのがAlexNetである。畳み込み層とプーリング層を重ねて、最後に全結合層を経由して予測値を出力する、という点では従来の畳み込みニューラルネットワークと同様。しかし計算コストの高い畳み込みニューラルネットワークに耐えられるビックデータとGPUが利用できるようになったことで、AlexNetが真価を発揮し、飛躍的な精度向上が達成された。

■実装演習

ＣＮＮで階層を深くした例。

「畳み込み層6層→プーリング層1層→全結合層」

計算コストが大きいため、学習回数はこれまでの1000回から100回に減らして図式化。



■考察

計算コストが高いため100回で打ち切ったが、順調に学習が進んでいることが分かる。わずか30分ほどの学習で、画像を約90%近い精度で識別できていることは素晴らしいと感じる。ネットワーク全体では膨大な計算量であるが、それは単なる部品の組み合わせであることにプログラムを見て気づき、感動する。機能の部品化はプログラムの強みであり、それによって人間が理解し、チューニングできるモデルを作り上げることができる、深層学習とプログラミングの相性の良さを強く感じた。