Study AI レポート

深層学習 Day1

Day1Session1・入力層~中間層

例えば入力された値から犬、猫、ネズミなどに分類するネットワークがあった場合、入力値が入れられる層が入力層と呼ばれる。体重、体長、ひげの本数、足の長さなどを変数にしてもよい。その入力値に重み w を乗じ、バイアス b が足されて中間層が形成される。中間層は何層あってもよい。そして中間層が何層にも重なっていく。

確認テストに関する考察

・ディープラーニングは何をしようとしているのか？

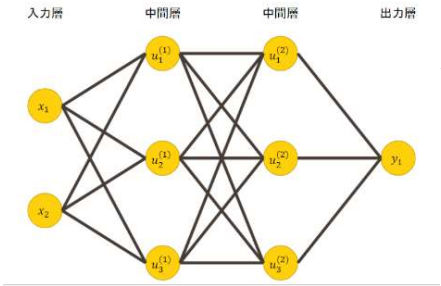
明示的なプログラムの代わりに多数の中間層を持つニューラルネットワークを用いて入力値から目的とする出力地に変換する数学モデルを構築する事

→この”明示的なプログラムを使わない”所にディープラーニングの真髄があり、極端に言えばアルゴリズムを考えずにデータからだけで数学モデルを作れる事が飛躍をもたらしたと考えている。

・最適化の最終目標は何か？

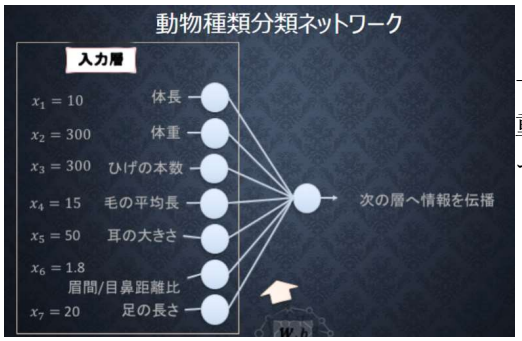
→重みとバイアスの二つを、最適化する事でネットワークが作成される。ハイパーパラメータ以外は自動的に同じ手順で作られる事が特に優れている点だと感じる。

・入力層 2，中間層 2 層、出力層 1 個の出力を図解



→入力層と中間層の全てがつながっている全結合なのがポイント

・動物分類の実例を入れる



→入力自体はなんでもよく、一つのノードに対して体調、体重やひげの本数など、数値で表せる特徴であればなんでも使用できるのが優れたポイントだと考える。

・以下の式を Python で記載する

$$u = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b$$
$$= Wx + b \quad ..(1.2)$$

```
u1 = np.dot(x, W1) + b1
```

→numpy を使用する事でシンプルに記載することができる。

・中間層の出力を定義しているソースコードの抜き出し

```
# プロセスを作成
def forward(network, x):
    print("##### 順伝播開始 #####")

    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']

    # 隠れ層の総入力
    u1 = np.dot(x, W1) + b1
    # 隠れ層の総出力
    z1 = functions.relu(u1)
    # 出力層の総入力
    u2 = np.dot(z1, W2) + b2
    # 出力層の総出力
    y = functions.sigmoid(u2)
```

```
# ReLU関数
def relu(x):
    return np.maximum(0, x)
```

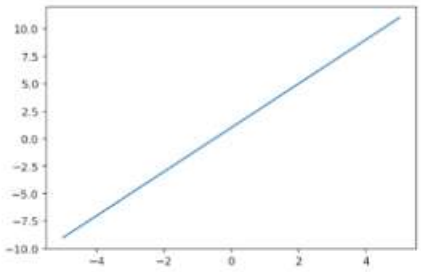
→functions のメソッドに relu 関数を定義している事でシンプルに記載が可能になっている。

Day1Session2・活性化関数

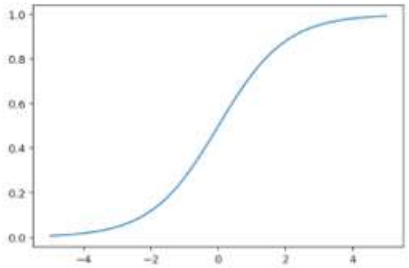
各中間層において入力信号の総和を出力信号に変換する関数を活性化関数と呼ぶ。活性化関数は入力信号の総和がどのように発火するかという事を決定する役割を持つ。ニューラルネットワークの語源である、この中間層のニューロン（ノード）に活性化関数が定義ずけられている。この活性化関数にはシグモイド関数やステップ関数、ReLU 関数などが用いられる。

確認テストに関する考察

・線形と非線形の違いを図に書いて解説



線形な関数



非線形な関数

線形な関数は

- 加法性: $f(x+y)=f(x)+f(y)$
- 斉次性: $f(kx)=kf(x)$

を満たす

非線形な関数は加法性･斉次性を満たさない

→加法性と斉次性を満たす線形関数のほうが、取り扱いが楽であることがこの図から直感的に理解できた。

・配布コードから、活性化関数部分を抜き出せ

```
z1 = functions.sigmoid(u)
```

→functions の下に定義されたシグモイド関数をこのように分かりやすく書くことでコードの可読性が向上する

コード確認 1）単層

```
## 試してみよう_配列の初期化
#W = np.zeros(2)
#W = np.ones(2)
W = np.random.rand(2)
#W = np.random.randint(5, size=(2))

print_vec("重み", W)


# バイアス
b = 0.5

## 試してみよう_数値の初期化
#b = np.random.rand() # 0~1のランダム数値
b = np.random.rand() * 10 -5  # -5~5のランダム数値
```

```
*** 重み ***
[0.68717576 0.37307439]

*** バイアス ***
-4.92541939796057

*** 入力 ***
[2 3]

*** 総入力 ***
-2.431844718889231

*** 中間層出力 ***
0.0
```

2）順伝播

```
# 順伝播（単層・複数ユニット）

# 重み
W = np.array([
    [0.1, 0.2, 0.3],
    [0.2, 0.3, 0.4],
    [0.3, 0.4, 0.5],
    [0.4, 0.5, 0.6]
])

## 試してみよう_配列の初期化
#W = np.zeros((4,3))
#W = np.ones((4,3))
#W = np.random.rand(4,3)
W = np.random.randint(5, size=(4,3))
```

```
*** 重み ***
[[2 3 1]
 [1 3 1]
 [2 3 3]
 [1 0 3]]

*** バイアス ***
[0.1 0.2 0.3]

*** 入力 ***
[ 1.  5.  2. -1.]

*** 総入力 ***
[10.1 24.2  9.3]

*** 中間層出力 ***
[0.99995892 1.         0.99990858]
```

3）多クラス分類　3-5-6

```
# ネートワークを作成
def init_network():
    print("##### ネットワークの初期化 #####")

    #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 = {}

    input_layer_size = 3
    hidden_layer_size=5
    output_layer_size = 6
```

```
##### ネットワークの初期化 #####
*** 重み1 ***
[[0.96435922 0.87718393 0.10590411 0.96364851 0.88982683]
 [0.98543597 0.6359834  0.13505788 0.34614962 0.49913024]
 [0.86382494 0.09451923 0.97245061 0.46351672 0.13100366]]
shape: (3, 5)

*** 重み2 ***
[[0.67087916 0.48141763 0.65603188 0.04588593 0.51557679 0.78059003]
 [0.08960264 0.99562496 0.32629889 0.80996083 0.88153607 0.09200807]
 [0.97125946 0.83733898 0.9262174  0.20639224 0.92459765 0.46342825]
 [0.04851751 0.75201935 0.342094   0.9523487  0.64817616 0.14550226]
 [0.80820571 0.35630611 0.66292031 0.59385379 0.42196434 0.81921215]]
shape: (5, 6)

*** バイアス1 ***
[0.37504933 0.83004589 0.92373469 0.03639516 0.7308522 ]
shape: (5,)

*** バイアス2 ***
[0.67155688 0.63133418 0.84602386 0.6650295  0.92700654 0.50036442]
shape: (6,)

##### 順伝播開始 #####
*** 総入力1 ***
[5.90175531 3.2627543  4.21710638 3.08289308 3.01195049]
shape: (5,)

*** 中間層出力1 ***
[5.90175531 3.2627543  4.21710638 3.08289308 3.01195049]
shape: (5,)

*** 総入力2 ***
[11.6030273  13.64374203 12.73967634  9.17356569 14.01438215 10.27773604]
shape: (6,)


出力合計: 0.9999999999999999

##### 結果表示 #####
*** 出力 ***
[0.04289032 0.33008856 0.13365947 0.00377796 0.47818663 0.01139706]
shape: (6,)

*** 訓練データ ***
[0 0 0 1 0 0]
shape: (6,)

*** 交差エントロピー誤差 ***
5.578544174588969
shape: ()
```

Day1Session3・出力層

出力層では、中間層からの信号を受け取り最終的に出力を出すために準備される層である。ニューラルネットワークは分類にも回帰にも使用する事が可能であるが、一般的に回帰問題では恒等関数が、二値分類ではシグモイド関数が、多クラス分類問題ではソフトマックス関数が用いられる。また回帰の誤差関数としては二乗誤差が、分類問題では交差エントロピーが使用される。ソフトマックス関数の実装ではオーバーフローが発生しないように工夫が必要である。ソフトマックス関数の出力は 0 から 1.0 までの間の実数になり、出力の総和は 1 になるという重要な性質をもつ。この性質のおかげでソフトマックス関数の出力を確率として解釈する事が可能になる。出力層のニューロンの数は解くべき問題に応じて適宜決める必要があり、例えば手書き文字認識のように入力画像が 0 － 9 までのどれかを予想する問題では、10 クラス分類問題になるので出力層のニューロンは 10 個必要になる。

確認テストに関する考察

・なぜ引き算でなく二乗するか？

引き算だけでは、各ラベルでの誤差で正負両方の値が発生し、全体の誤差を正しく表すのに都合が悪い。それぞれのラベルでの誤差を正の値になるようにする

→分散と同じ考え方。ある値からどの位離れているか？の評価には二乗する事が都合がよい

・式中の 1/2 はどのような意味を持つか？

ネットワークを学習する時に行う誤差逆伝搬の計算時に誤差関数の微分を用いるが、その際の計算式を簡単にするためで、本質的な意味はない

→機械学習の計算では、この例のように計算式を簡単にする等の工夫が多く用いられている。そのような工夫が積み重なって、現実的な時間で PC の計算能力で解を出せる仕組みが出来ている。

```python
# プロセスを作成
def forward(network, x):
    print("##### 順伝播開始 #####")

    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']

    # 隠れ層の総入力
    u1 = np.dot(x, W1) + b1
    # 隠れ層の総出力
    z1 = functions.relu(u1)
    # 出力層の総入力
    u2 = np.dot(z1, W2) + b2
    # 出力層の総出力
    y = functions.sigmoid(u2)
```

```
##### 結果表示 #####
*** 中間層出力 ***
[0.6 1.3 2. ]

*** 出力 ***
[0.87435214]

*** 訓練データ ***
[1]

*** 誤差 ***
0.13427195993720972
```

出力層にシグモイド関数を使用している。誤差評価としてクロスエントロピーを定義していて、この

```python
# クロスエントロピー
def cross_entropy_error(d, y):
    if y.ndim == 1:
        d = d.reshape(1, d.size)
        y = y.reshape(1, y.size)

    # 教師データがone-hot-vectorの場合、正解ラベルのインデックスに変換
    if d.size == y.size:
        d = d.argmax(axis=1)

    batch_size = y.shape[0]
    return -np.sum(np.log(y[np.arange(batch_size), d] + 1e-7)) / batch_size
```

出力値は閾値次第になるが目的 1 と分類が正しく行われる結果がでていると思われる。

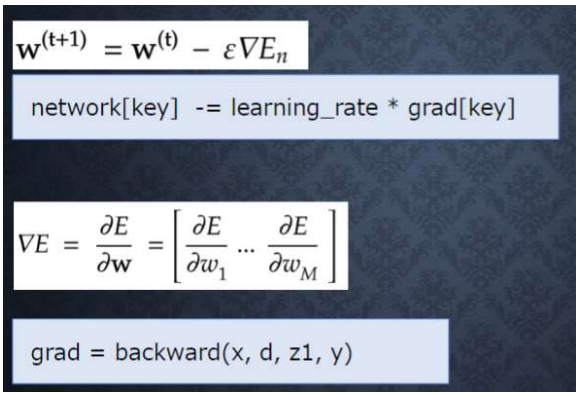
Day1Session4・勾配降下法

機械学習では学習の際に最適なパラメータを探索しなければならない。ニューラルネットワークの場合は最適な重みとバイアスを見つける必要があり、損失関数が最小値となる最適パラメータを見つけなければならない。しかし一般に損失関数は複雑で、パラメータ空間は広大でどこに最小値を取る場所があるかを探すのは難しい。そこで最小値を見つけるために現在の場所から勾配方向に一定の距離だけ進め、移動した先でも勾配を求めてその手法を繰り返す事で関数の値を徐々に減らすのが勾配降下法である。

学習率とは一回の学習でどれだけパラメータを更新するか、という値であり一般的に大きすぎても小さすぎても良い場所にたどり着くことが出来ない。この学習率のようはパラメータはハイパーパラメータと呼ばれ、学習の際に自動で獲得される重みやバイアスと異なり、人の手で設定する必要がある。
勾配法の中で、ミニバッチとして無作為に選ばれたデータを使用する方法を、確率的勾配降下法SGD（Stochastic gradient descent）という。

確認テストに関する考察

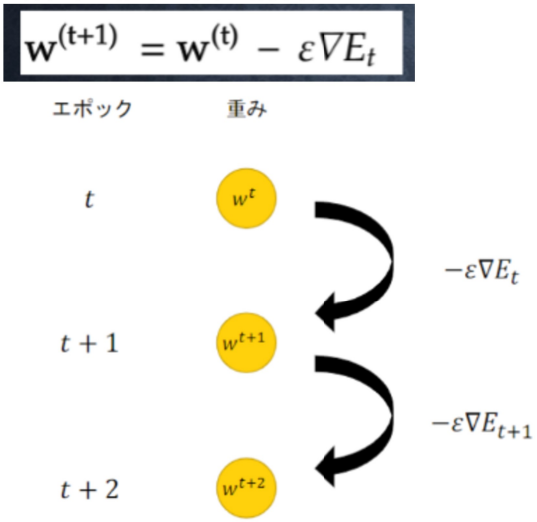
・該当するソースコード部分



・オンライン学習とは何か？

学習データが入ってくるたびに都度パラメータを更新し学習を進めていく方法。一方バッチ学習では一度にすべての学習データを使ってパラメータ更新を行う。
→一見オンライン学習のほうが、都度パラメータを更新してくので優れた方法に見えるかもしれないが、基準が都度変化してしまうデメリットもあるので、どちらの学習方法を使ってネットワークを作るか？は使用者が対称に応じて見極めなければならない。

・数式の意味を図で説明する



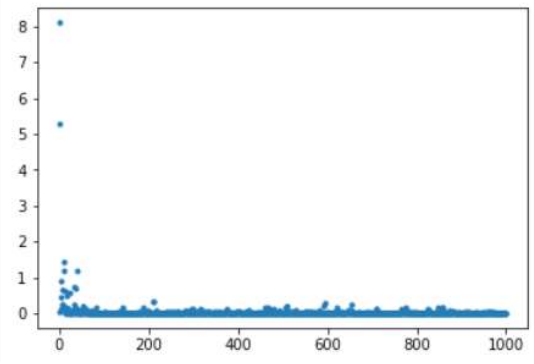
→漸化式的に、一個前の値を用いてエポックが更新されると毎にΔEの値を変えて引いていく所がポイントである。高校数学の考え方に慣れておく必要がある。

```
# 勾配降下の繰り返し
for dataset in random_datasets:
    x, d = dataset['x'], dataset['d']
    z1, y = forward(network, x)
    grad = backward(x, d, z1, y)
    # パラメータに勾配適用
    for key in ('W1', 'W2', 'b1', 'b2'):
        network[key]  -= learning_rate * grad[key]

    # 誤差
    loss = functions.mean_squared_error(d, y)
    losses.append(loss)

print("##### 結果表示 #####")
lists = range(epoch)
```

```
##### 結果表示 #####
```



コード確認
勾配降下法でロス関数がエポック事に低下していき、ある回数からほとんど変化しなくなる様子がグラフに図示されている。

Day1Session5・誤差逆伝搬

誤差逆伝搬とは、算出された誤差を出力層側から順に微分し、前の層前の層へと伝播させていく。最小限の計算で各パラメータでの微分値を解析的に計算する方法である。誤差から微分を逆算することで、不要な再帰的計算を避けて微分を算出できる。誤差逆伝搬のポイントは、局所的な微分を順方向から逆向きに右から左に伝達していく。この局所的微分を伝達する原理は連鎖律によるものである。連鎖率とは、ある関数が合成関数で表せる場合、その合成関数の微分は合成関数を構成するそれぞれの関数の微分の積によって表す事ができる事である。

確認テストに関する考察

・誤差逆伝搬法で既に行った計算結果を保持しているソースコードを抽出する

```
#  出力層でのデルタ
delta2 = functions.d_sigmoid_with_loss(d, y)
```

```
#  中間層でのデルタ
delta1 = np.dot(delta2, W2.T) * functions.d_relu(z1)
```

→Δ 1， 2として定義している、reluとシグモイドとクロスエントロピーの複合導関数を使用している。

```
##### 誤差逆伝播開始 #####
*** 偏微分_dE/du2 ***
[[ 0.08706577 -0.08706577]]

*** 偏微分_dE/du2 ***
[[-0.02611973 -0.02611973 -0.02611973]]

*** 偏微分_重み1 ***
[[-0.02611973 -0.02611973 -0.02611973]
 [-0.13059866 -0.13059866 -0.13059866]]

*** 偏微分_重み2 ***
[[ 0.10447893 -0.10447893]
 [ 0.21766443 -0.21766443]
 [ 0.33084994 -0.33084994]]

*** 偏微分_バイアス1 ***
[-0.02611973 -0.02611973 -0.02611973]

*** 偏微分_バイアス2 ***
[ 0.08706577 -0.08706577]
```

```
##### 結果表示 #####
##### 更新後パラメータ #####
*** 重み1 ***
[[0.1002612  0.3002612  0.5002612 ]
 [0.20130599 0.40130599 0.60130599]]

*** 重み2 ***
[[0.09895521 0.40104479]
 [0.19782336 0.50217664]
 [0.2966915  0.6033085 ]]

*** バイアス1 ***
[0.1002612 0.2002612 0.3002612]

*** バイアス2 ***
[0.09912934 0.20087066]
```

コード確認　偏微分により重みとバイアスの変化を伝播させている事がよく分かる。

Day2Session1・勾配消失問題

誤差逆伝搬法が下位層に進んでいくに連れて、勾配がどんどん緩やかになっていく。そのため勾配降下法による更新では下位層のパラメータはほとんど変わらず、訓練は最適地に収束しなくなる問題が発生する。これが勾配消失問題である。活性化関数として用いられるシグモイド関数は、0 から 1 の間を緩やかに変化する関数で、信号の強弱を伝えられるようになり、ニューラルネットワーク普及のきっかけとなった、しかし大きな値では出力の変化が微小なため、勾配消失問題を引き起こす事があった。勾配消失問題への対策としては、１．活性化関数の選択、２．重みの初期値の設定方法　３．バッチ正規化の 3 手法がある。

まず活性化関数の選択としては、シグモイド関数の微分の最大値が 0.25 であり、繰り返すごとに値が勾配が小さくなる要因である。これに対して ReLU 関数の微分が 0 と 1 に別れる事でデータをスパースにする効果と、値が小さくならないので活性化関数としてよく使用されている。

重みの初期値の設定方法としては、ガウス分布、Xavier の初期値、He の初期値などが知られている。ガウス分布では勾配消失が発生してしまう場合も Xavier や He の初期値を使用する事で防止する事ができる。活性化関数に ReLU を使う場合は He,シグモイドや Tanh などの S 字カーブの場合は Xavier を使うのがよい。

バッチ正規化とは、ミニバッチ単位で入力値のデータの偏りを抑制する手法である。バッチ正規化は、活性化関数に値を渡す前後にバッチ正規化の処理をはらんだ層を加える事である。バッチ正規化の効果としては、学習を速く進行させる事ができる、初期値依存が減る、過学習を抑制する等である。

確認テストに関する考察

・連鎖率の原理を使い、dz/dx を求めよ　z = t2 t=x+y

dz/dx=dz/dt*dt/dx　dz/dt=2t　dt/dx = 1　よって dz/dx=2t*1 =2(x+y)

→偏微分の有効活用である事が分かる

・シグモイド関数を微分した時、入力値が 0 の時に最大値をとる。その値として正しい物

0.25

→値が小さい。元の 1/4 になるので連鎖していくと小さくなる事がわかる

・重みの初期値に 0 を設定すると、どのような問題が発生するか？完結に説明せよ

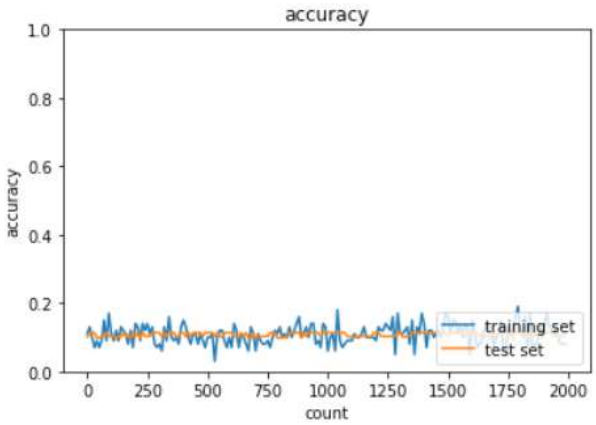
全ての重みが均一に更新されるため、多数の重みをもつ意味がなくなる。

→人の個性に例える所がとても分かりやすかった。複雑な反応が出せるネットワークにはやはり複雑なシステム構築が必要で、それには様々な個性（異なる値のパラメータ変数）がいるという事を理解できた。

・一般的に考えられるバッチ正規化の効果を 2 点挙げよ

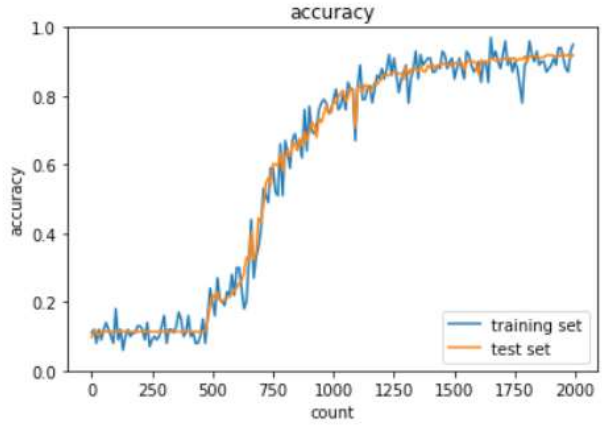
学習を早く収束させる事ができる。過学習を抑制する

→別の専門書では、初期値にそれほど依存しない、という文言があったが、これは過学習を抑制する要件の中に一部含まれると考える。

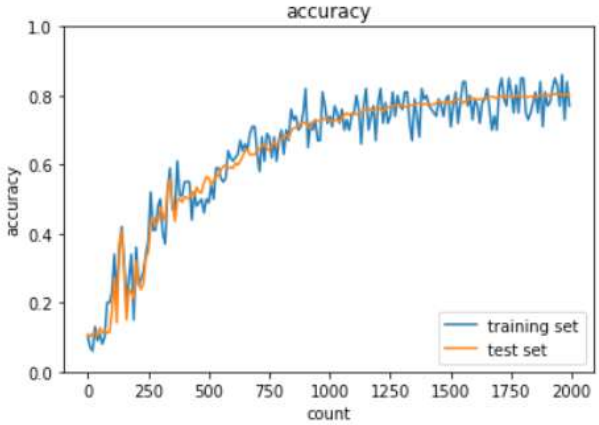


勾配消失が起った場合。テストセットもトレーニングセットも正確性が上昇していかない。

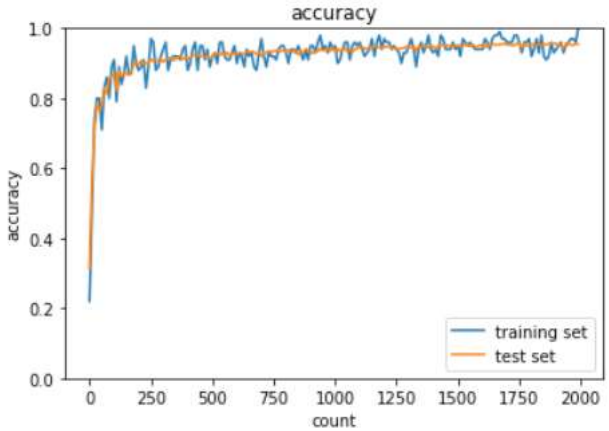
活性化関数にシグモイド関数を摘要しており、その問題が現れている



活性化関数に Relu を使用した場合。テストセットもトレーニングセットも正確性が向上しているのがわかる。初期の重みにはガウス分布を使用している。



活性化関数に Relu を使用し、重みの初期化に Xavier の初期化を摘要した場合。学習の進み方が明らかに向上している。



活性化関数に Relu を使用し、重みの初期化に He の初期化を摘要した場合。Xavier の初期化に比較しても学習の進み方が特に顕著に向上している事がわかる。

Day2Session2・学習率最適化手法

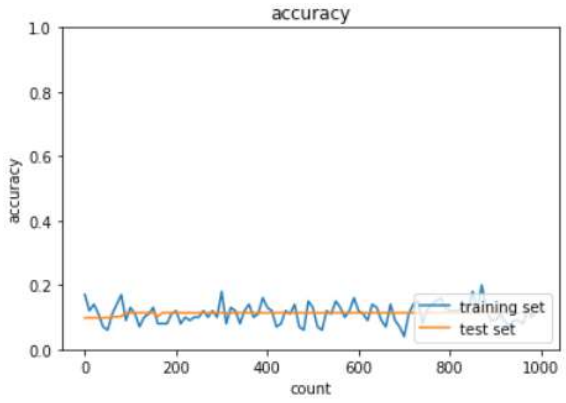
学習率の値が大きい場合は、最適地にいつまでもたどり着かず発散してしまう。また学習率の値が小さい場合は、発散する事はないが小さすぎると収束するまでに時間がかかってしまう、また大局的最適地に収束しづらくなる問題がある。これに対して初期の学習率を大きく設定し、徐々に小さくしたり、パラメータ毎に学習率を可変させる方法がある。学習率最適化方法としては、SGD、モメンタム、AdaGrad,RMSProp,Adam などがある。モメンタムは誤差をパラメータで微分したものと学習率の積を減算した後、現在の重みに前回の重みを減算した値と慣性の積を加算する方法で、局所最適解ではなく、大局的最適解になり、谷間についてから最適値に行くまでの時間が早い。AdaGrad は誤差をパラメータで微分したものと再定義した学習率の積を減算する方法である。メリットとしては勾配の緩やかな斜面に対して最適値に近づける事ができるが、学習率が徐々に小さくなるので鞍点問題を引き起こすことがある。 RMSProp は誤差をパラメータで微分したものと再定義した学習率の積を減算する方法で、大局的最適解となり、ハイパーパラメータの調整が必要な場合が少ない。Adam とは、モメンタムの過去の勾配の指数関数的減衰平均と RMSProp の、過去の勾配の二乗指数巻子的減衰平均をそれぞれはらんだ最適化アルゴリズムである。つまりモーメンタム、RMSProp との合成が Adam になっており、それぞれのメリットを持つ特徴がある。様々な手法があり、それぞれ特徴があって全てに優れた方法はなく、使用者が選んで使う必要がある。アンサンブル学習を美術館での絵の感想を複数人でいいあう例に例えるのが分かりやすかった。この学習率最適化に関して様々なアルゴリズムがあり、とても面白いと感じた。

確認テストに関する考察

・モメンタム、AdaGrad、RMSProp の特徴を説明せよ

→上記に説明済み。相互に関係しあっている所が面白い。アルゴリズムに名前がついている程、有名で確率された手法であることが理解できる。

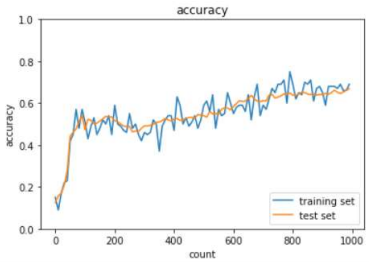
S GD



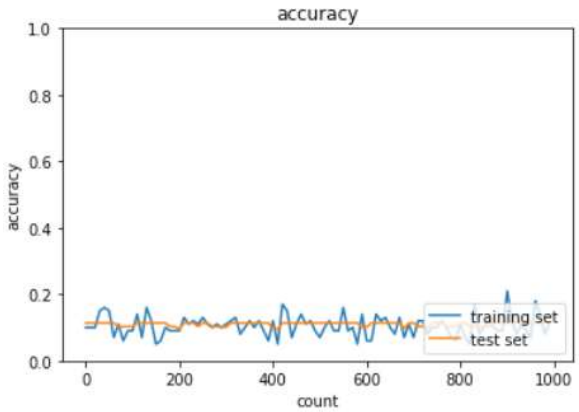
全く学習が進まない

バッチノーマライゼーションなし

バッチノーマライゼーション有だと少し改善する

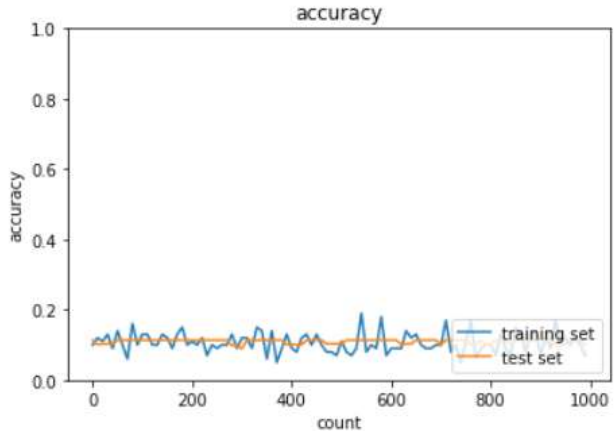


モーメンタム



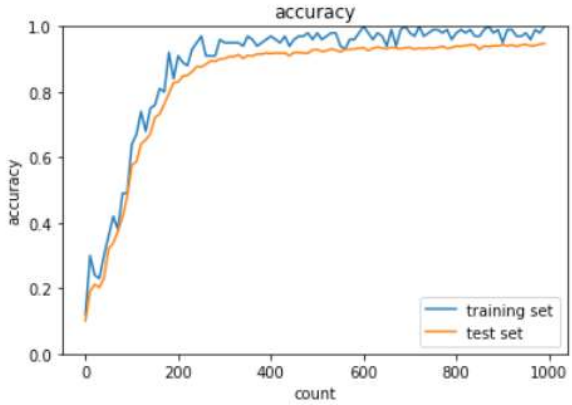
SGD と同様に学習ができていない

AdaGrad



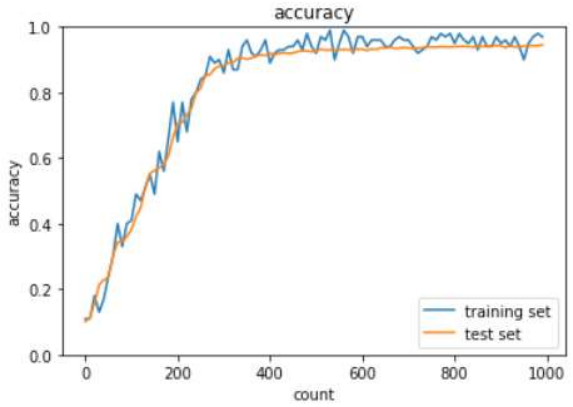
学習されていない

RMSProp

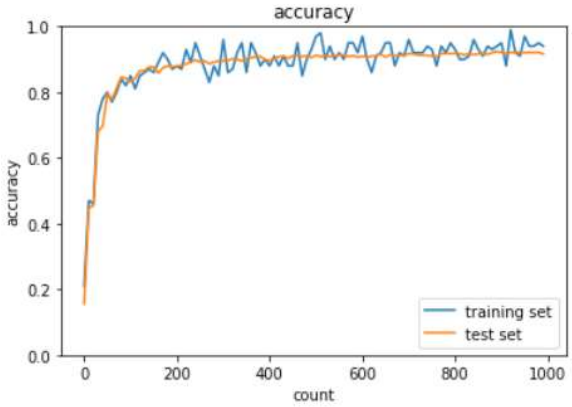


学習が進むようになった

Adam



学習の進みは少し遅いが、**RMSPROP** に比較してテストセットの学習結果も向上している



Adam に対してバッチノーマライゼーションを追加すると学習速度が劇的に向上する。

Day2Session3・過学習

過学習とは、テスト誤差と訓練誤差とで学習曲線が乖離する事である。特定の訓練サンプルに対して特化して学習してしまう事で、本当に推測しなければいけない未知のデータに対して性能が悪くなる避けなければならない現象である。過学習が起きる原因は、パラメータの数が多い、パラメータの値が適切でなく、ノードが多い事などからネットワークの自由度が高い状態になっている事があげられる。この問題に対して正則化といわれる、ネットワークの自由度（層数、ノード数、パラメータの値）を制約する事で過学習を抑制する方法がある。正則化手法としてはＬ１正則化、Ｌ２正則化、ドロップアウトなどがあげられる。Weight decay という手法は、重みが大きい値を取ることで過学習が発生する事があるので、大きな重みを持つことに対してペナルティを課す事で抑制する方法である。これは誤差関数にＰノルムを加える事でペナルティを課す方法である。Ｐ＝１の場合Ｌ１正則化、Ｐ＝２の場合Ｌ２正則化と呼ばれる。

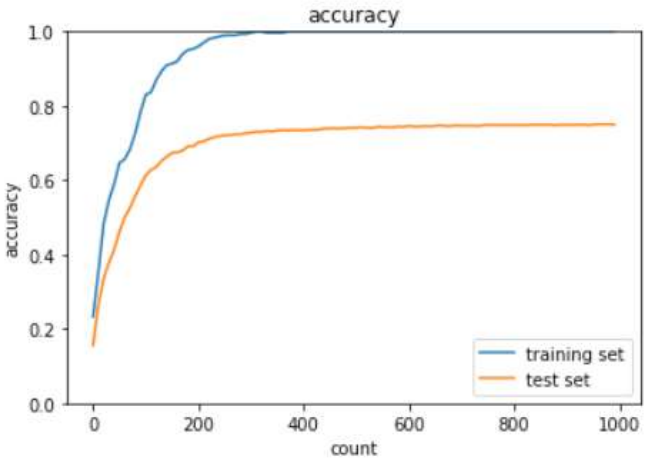
ニューラルネットワークのモデルが複雑になってくると Weight decay だけでは対応が困難になる場合がある。ドロップアウトとは、過学習の課題としてノードの数が多い事によって起きやすい事に目を付け、ランダムにノードを除去して学習させる事で過学習を防ぐ方法がある。メリットとしてはデータ量を変化させずに、異なるモデルを学習させていると解釈できる。

確認テストに関する考察

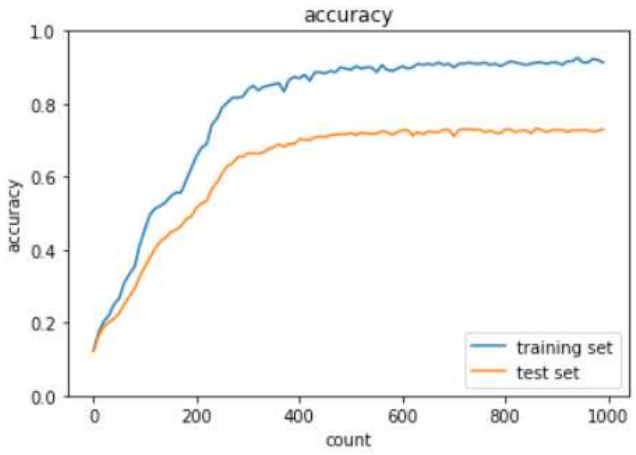
・機械学習で使われる線形モデルの正則化は、モデルの重みを制限する事で可能となる。リッジ回帰という手法があり、その特徴として正しい物は？

リッジ回帰の場合、隠れ層に対して正則化項を加える
→ネットワークの自由度を抑制すること、という事が理解できた

・Ｌ１正則化を表しているのはどちらか？
距離の定義が異なる　Ridge 推定量 Lasso 推定量　パラメータの一つがゼロの値の推定値になるので、右側の Lasso 推定量がＬ１正則化になる
→図を重ね合わせて説明されることで理解がすすんだ。P1 ノルム　ユーグリッド距離を使うのが L1 正則化で P2 ノルム　マンハッタン距離を使うのが L2 正則化

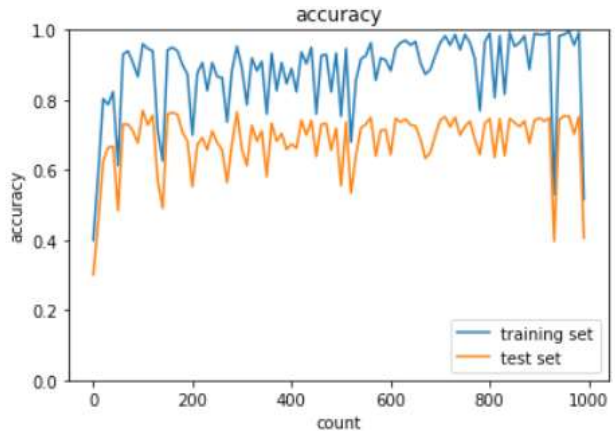


トレーニングセットに対して良好な結果に対し、テストセットは学習回数を増やしても 80%にも正解率が上がらなくなった過学習が発生している状態。起りやすくするために学習データは少なくしている。



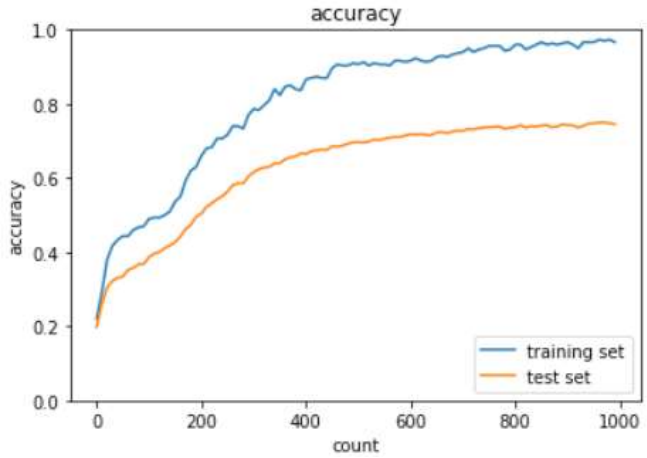
L2 正則化を入れた場合。トレーニングセットの正解率も下がっているが、テストセットとトレーニングセットの差は少し改善している。

weight_decay_lambda = 0.1



L1 正則化を入れた場合。正解率の回数に対する変動が大きな結果となっている。

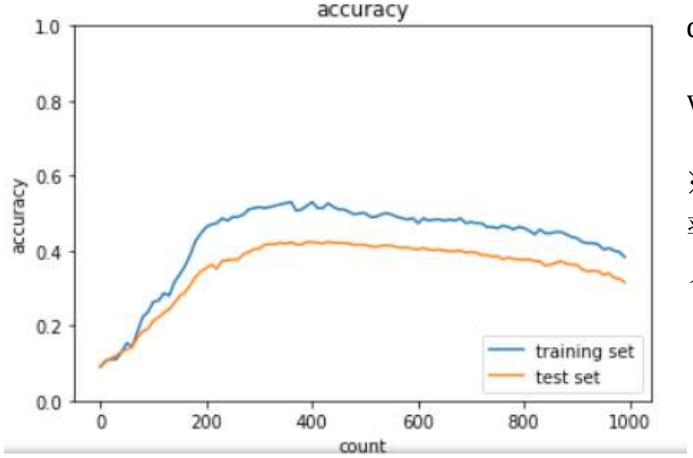
weight_decay_lambda = 0.005



ドロップアウト＋L1 正則化の混合テストセットにおける正解率の頭打ちが改善している。

dropout_ratio = 0.08

weight_decay_lambda=0.004



dropout_ratio = 0.08

weight_decay_lambda=0.01

※パイパーパラメータの調整は難しく、正解率と過学習抑制のどちらも改善させるパラメータの選択、探索はなかなか難しい。

Day2Session4・畳み込みニューラルネットワークの概念

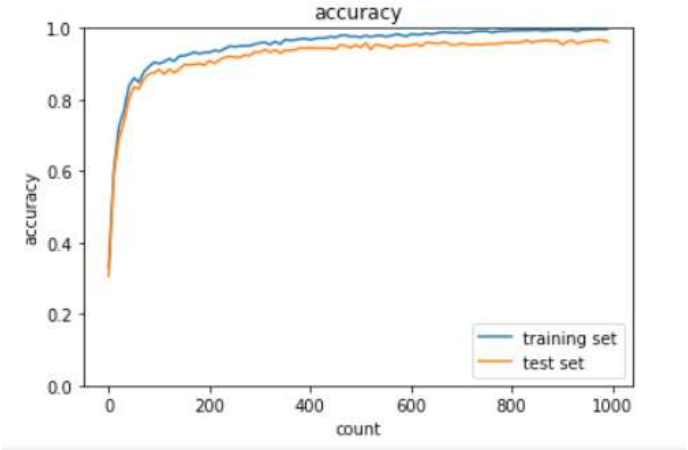
畳み込みニューラルネットワークとはConvolutional Neural Network:CNNの事である。これまでは全結合（fully-connected）という方法であったが、この方法が持つ問題は”データの形状が無視されてしまう事”である。例えば入力信号が画像だった場合、縦・横・チャンネル方向に本来意味のある情報が含まれているが、それを1次元データにして入力することでその情報を活かせていない事である。これに対しCNNは、畳み込み層を持つことにより、形状を維持する事ができる。ＣＮＮの構造は、入力層→畳み込み層→プーリング層→（畳み込みとプーリングは繰り返したりする）→全結合層→出力層という形をとる。畳み込み演算では、パディング（周囲の穴埋め）やストライド（カーネルをどれだけ動かすか）といった概念が重要になる。プーリングは、縦横方向の空間を小さくする演算である。例えば3×3のカーネル内で最大の値を取って代表値として置換える事でデータを小さくする。プーリング層は学習するパラメータを持たず、チャンネル層は変化させない特徴があり、このプーリング層を入れる事で微小な変化に対してロバストになるという利点がある。

確認テストに関する考察

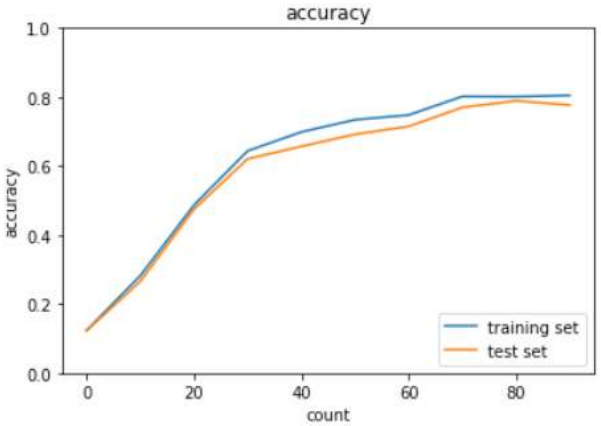
・サイズ6×6の入力画像をサイズ2×2のフィルタで畳みこんだ時の出力画像のサイズを答えよ。ストライドとパディングは1とする

OH＝（6+2*1-2)/1 + 1 = 7　※OWも同じ

→理屈は分かりやすいが、計算式を覚えておいたほうが機械的にできて楽と判断した。

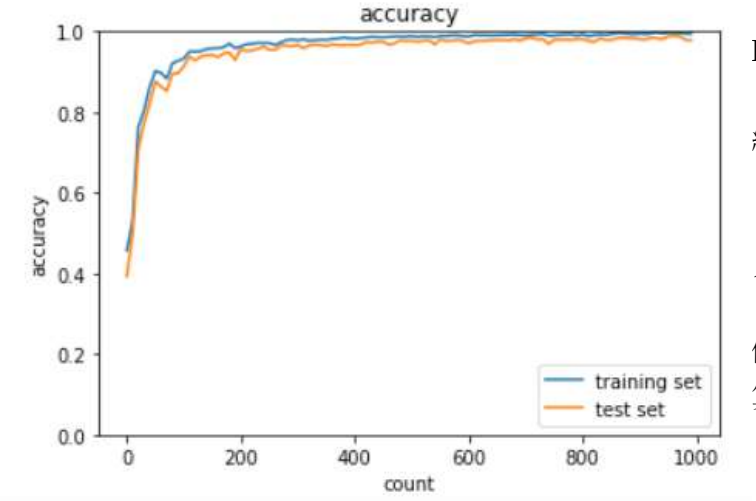


シンプルニューラルネットワークの結果



ダルブルニューラルネットワークの結果

100回までの学習速度部分の表示



He の初期化を用い、層の深いコンボリューショナルニューラルネットワークの結果

トレーニングセットに対し 99.4%以上、テストセットも約 98%の結果を出している。

使用したノートパソコンでは 1000 回の計算を行うのに 2 時間以上時間がかかった。

```
Generation: 970. 正答率(トレーニング) = 0.9966
                : 970. 正答率(テスト) = 0.989
Generation: 980. 正答率(トレーニング) = 0.9958
                : 980. 正答率(テスト) = 0.988
Generation: 990. 正答率(トレーニング) = 0.9944
                : 990. 正答率(テスト) = 0.979
Generation: 1000. 正答率(トレーニング) = 0.9942
                : 1000. 正答率(テスト) = 0.978
```

Day2Sess5・最新の CNN

LeNet は手書き数字認識用に 1989 年に提案された。LeNet では活性化関数にシグモイド関数が使用されている。

AlexNet は 2012 年にヒントンらによって提唱され、ILSVRC で優勝している。LeNet との違いとしては、活性化関数に ReLU を使用し、LRN（Local Response Normalization）という局所的正規化層をいれ、過学習を防ぐ施策として、サイズ 4096 の全結合層の出力にドロップアウトを使用している事である。その他に VGG、GoogleNet、ResNet などが提唱されている。