計算機科学実験 3
画像認識　課題 4

学生番号 1029-28-6051 吉田　聖

2019 年 3 月 25 日

1　作成したプログラム

今回の課題で到達した課題は、発展課題 A4 まで。様々な手法を用いた結果の比較等々を行う。

1.1　活性化関数

1.1.1　ReLU

シグモイド関数の代わりに用いられるもの。この関数を導入する際、はじめ学習率を 0.01 を設定していたが、上手くいかなかった。学習率を 0.0001 に変更することで動作するようになったため、学習率の重要さを理解した。BatchNorm を実装するまで学習率の初期値を意識しなければならない。このクラスでは初期化、順伝播、逆伝播の関数を持つ。

- __init__0 より大きい数かどうかを判定する変数 mask を宣言。
- forward0 以上の場合はそのままの数を、0 未満は 0 を出力するようにする。
- backward0 以上は 1 を、0 未満は 0 を出力する。

```
class ReLU:
    def __init__(self):
        self.mask = None

    def forward(self, x):
        self.mask = (x <= 0)
        out = x.copy()
        out[self.mask] = 0
        return out

    def backward(self, dout):
        dout[self.mask] = 0
        dx = dout
```

```
        return dx
```

1.2 過学習対策

1.2.1 Dropout

いくつかの出力を無視する手法。過学習を回避するために用いられる。初期値設定と順伝播と逆伝播の関数を持つ。

- __init__無視する割合と、無視するかどうかを判定する関数 mask をもつ。
- forward 学習時かテスト時かを判定する flag を用意。学習時はそのまま出力。テスト時は (1-学習率) 倍したものを出力。
- backwardmask によってノードが無視されないときは 1 を出力、無視されたときは 0 を出力。

```
class Dropout:
    def __init__(self, dropout_ratio = 0.5):
        self.dropout_ratio = dropout_ratio
        self.mask = None

    def forward(self, x, train_flg = True):
        if train_flg:
            self.mask = np.random.rand(*x.shape) > self.dropout_ratio
            return x * self.mask
        else:
            return x * (1.0 - self.dropout_ratio)

    def backward(self, dout):
        return dout * self.mask
```

1.2.2 BatchNorm

ノードの出力の分散を 1、平均を 0 となるようにする。

- __init__必要な変数を宣言する。必要な変数が多いので気を付ける。
- forward 学習時とテスト時の flag を保持しておき、レジュメ通りの計算となるように設計する。それぞれの挙動はレジュメ参照
- backward レジュメ通りの計算となるように注意する。特に変数が一次元の場合、numpy の仕様上二次元変数との計算をするとはじめに 1 が挿入されるので適宜適当な型に reshape する。

```
class BatchNorm:
    def __init__(self, gamma, beta, momentum=0.9):
        self.gamma = gamma
        self.beta = beta
```

```
        self.momentum = momentum
        self.input_shape = None

        # テスト時に使用する平均と分散
        self.running_mean = None
        self.running_var = None

        self.batch_size = None
        self.xc = None
        self.std = None
        self.dgamma = None
        self.dbeta = None

    def forward(self, x, train_flg=True):
        if self.running_mean is None:
            n, d = x.shape
            self.running_mean = np.zeros(n)
            self.running_var = np.zeros(n)

        if train_flg:
            mu = x.mean(axis=1)
            mu = np.reshape(mu, (mu.shape[0], 1))
            xc = x - mu
            var = np.mean(xc ** 2, axis=1)
            var = np.reshape(var, (var.shape[0], 1))
            std = np.sqrt(var + 10e-7)
            std = np.reshape(std, (std.shape[0], 1))
            xn = xc / std

            self.batch_size = x.shape[1]
            self.xc = xc
            self.xn = xn
            self.std = std
            self.running_mean = self.momentum * self.running_mean + (1 - self.momentum) * mu
            self.running_var = self.momentum * self.running_var + (1 - self.momentum) * var
        else:
            xc = x - self.running_mean
            xn = xc / (np.sqrt(self.running_var + 10e-7))

        out = self.gamma * xn + self.beta
```

```
        return out

    def backward(self, dout):
        dbeta = dout.sum(axis=1)
        dbeta = np.reshape(dbeta, (dbeta.shape[0], 1))
        dgamma = np.sum(dout * self.xn, axis=1)
        dgamma = np.reshape(dgamma, (dgamma.shape[0], 1))
        dxn = dout * self.gamma
        dxc = dxn / self.std
        dstd = -np.sum((self.xc * dxn) / (self.std * self.std), axis=1)
        dstd = np.reshape(dstd, (dstd.shape[0], 1))
        dvar = 0.5 * dstd / self.std

        dxc += (2.0 / self.batch_size) * dvar * self.xc
        dmu = np.sum(dxc, axis=1)
        dmu = np.reshape(dmu, (dmu.shape[0], 1))
        dx = dxc - dmu / self.batch_size

        self.dgamma = dgamma
        self.dbeta = dbeta

        return dx
```

1.3 最適化の手法

最適化の手法は初期値設定と更新の二つの関数によって構成される。

1.3.1 Momentum

慣性項。

- __init__ lr は 0.01、momentum は 0.9 が推奨されている。
- update 運動量の計算と同等の計算が行われている。

```
class Momentum:
    def __init__(self, lr, momentum=0.9):
        self.lr =lr
        self.momentum = momentum
        self.v = 0

    def update(self, params, grads):
```

```
        self.v = self.momentum * self.v - self.lr * grads
        return params + self.v
```

1.3.2 Adagrad

学習率を徐々に小さくしていく。アダマール積を用いる。

- __init__lr は 0.001,h は 1e-8 が推奨されている。
- update 分母が小さくなりすぎて、overflow しないように小さい値を入れておく。

```
class  Adagrad:
    def __init__(self, lr=0.001, h=1e-8):
        self.lr = lr
        self.h = h

    def update(self, params, grad):
        self.h += grad * grad
        params -= self.lr * grad / (np.sqrt(self.h) + 1e-7)
        return params
```

1.3.3 RMSprop

変数の h に二乗の指数移動平均を用いる。

- __init__lr は 0.001,p は 0.9 が推奨されている。
- update 分母に 1e-8 程度の小さい数を入れて overflow を防止する。

```
class RMSprop:
    def __init__(self, lr=0.001, p=0.9):
        self.lr = lr
        self.p = p
        self.h = 0

    def update(self, params, grad):
        self.h = self.p * self.h + (1 - self.p) * (grad * grad)
        params -= self.lr * grad / (np.sqrt(self.h) + 1e-8)
        return params
```

1.3.4 Adadelta

学習率の初期設定が不要。RMSProp や AdaGrad の改良版。

- __init__必要な変数の宣言。

- update 同様に設定。

```
class AdaDelta:
    def __init__(self, p=0.95, e=1e-6):
        self.h = 0
        self.s = 0
        self.e = e
        self.p = p
        self.delta = None

    def update(self, params, grad):
        self.h = self.p * self.h + (1 - self.p) * (grad * grad)
        self.delta = - np.sqrt((self.s + self.e)/(self.h + self.e))*grad
        self.s = self.p * self.s + (1 - self.p) * (self.delta * self.delta)
        params += self.delta
        return params
```

1.3.5 Adam

AdaDelta の改良版。二つの慣性項を用いている。

- __init__変数が多いので丁寧に宣言する。
- update 同様に立式する。

```
class Adam:
    def __init__(self, lr=0.001, b1=0.9, b2=0.999):
        self.lr = lr
        self.b1 = b1
        self.b2 = b2
        self.t = 0
        self.m = 0
        self.v = 0
        self.mm = 0
        self.vv = 0

    def update(self, params, grad):
        self.t += 1
        self.m = self.b1 * self.m + (1 - self.b1) * grad
        self.v = self.b2 * self.v + (1 - self.b2) * (grad * grad)
        self.mm = self.m/(1 - self.b1**self.t)
        self.vv = self.v/(1 - self.b2**self.t)
        params -= self.lr * self.mm/(np.sqrt(self.vv) + 1e-8)
```

```
        return params
```

1.3.6 chose_optimiser

最適化の手法を簡単に変えるため、作成。フラグがどの手法に該当するかは以下を参照されたい。

```
def chose_optimiser(flag, n):
    if flag == 1:
        return Momentum(n)
    elif flag == 2:
        return Adagrad()
    elif flag == 3:
        return RMSprop()
    elif flag == 4:
        return AdaDelta()
    else :
        return Adam()
```

1.4 実行結果

それぞれ実装した後、対照実験するため基本的に使用するプログラムは以下のとおりである。それぞれの課題で該当する部分だけ変化させて結果を検証する。
活性化関数：ReLU 関数
Dropout:不使用
BatchNormalization:使用
最適化の手法:Adam
学習回数は 1000 回。
エポックは学習 100 回につき一回。
バッチサイズは 100。

1.4.1　発展課題 A1

活性化関数の比較。結果の図は以下。体感的には ReLU 関数のほうが正答率があがる速度が速い気がする。

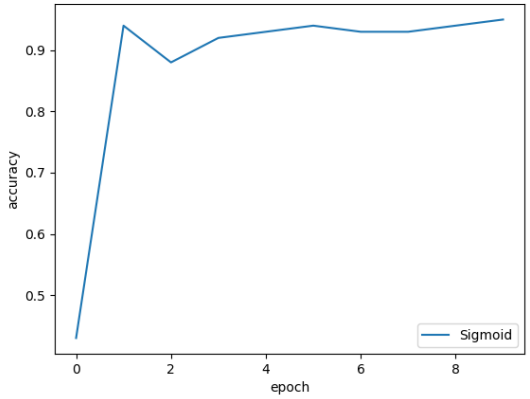

図 1　シグモイド関数

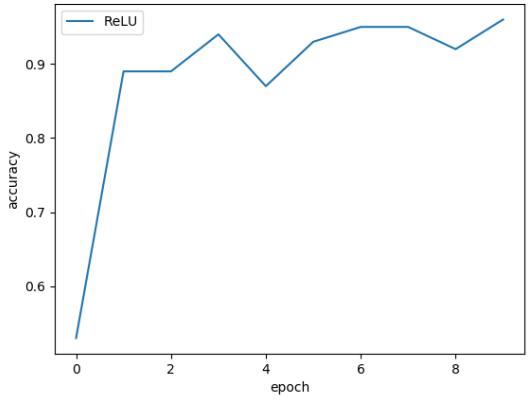

図 2　ReLU 関数

正答率からは有意な差は見受けられないが、エントロピー誤差では大きな差が生じる。ReLU 関数では 0.05293 程で 0.1 を切ることが多く、sigmoid 関数では 0.2 程度に収まる。

1.4.2　発展課題 A2

機械学習では教師データに適応しすぎて汎用データに適応できないのはよくないので、あえて過学習が起こりやすい状況を作り Dropout がうまく動作するかを確かめる。学習するデータを小さくし、テストデータはまた異なるものにして結果を比較する。結果は以下のとおりである。

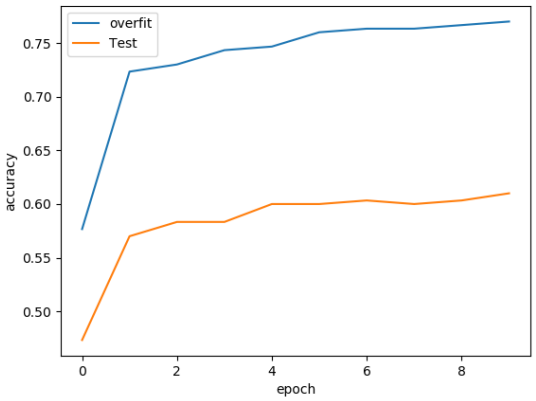

図 3　dropout あり

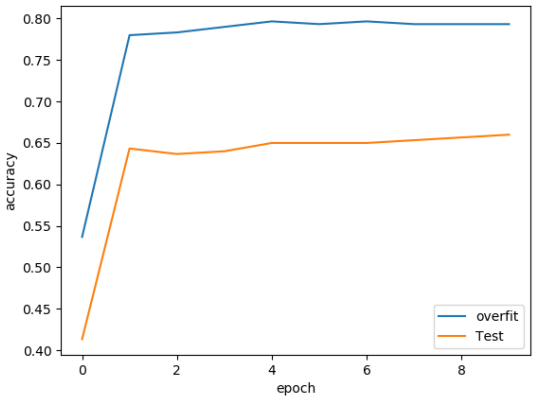

図 4　dropout なし

1.4.3　発展課題 A3

正答率からみると案外変化が無かった。エントロピー誤差の減り方がわずかに早くなったようではあった。

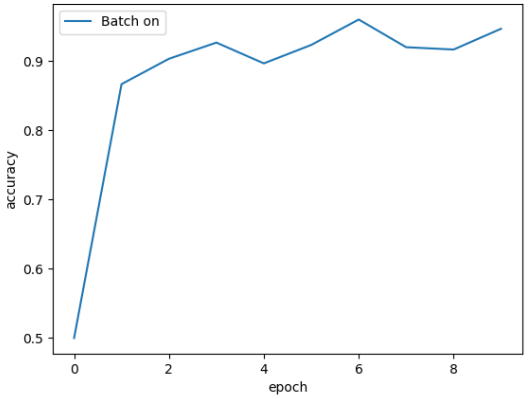

図 5　Batch あり

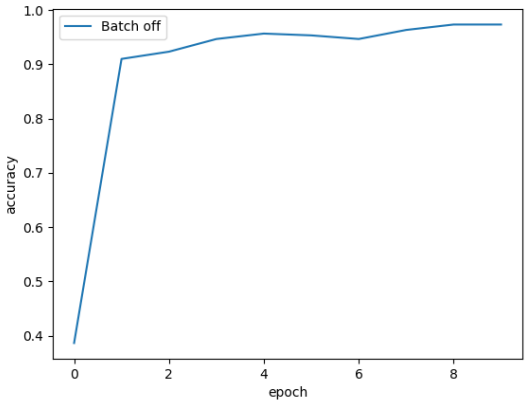

図 6　Batch なし

1.4.4　発展課題 A4

それぞれの正答率の差は以下の図の通り。今回の課題を行う中で当初は Momentum で行っていたが、他の

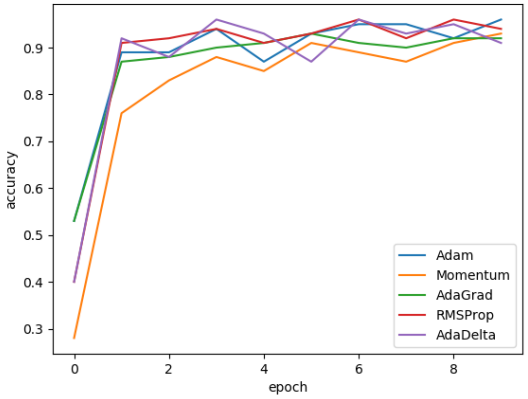

図 7　最適化の手法の比較

関数を取り入れても高々 9 割ほどしか正答しなかったが Adam に変えることで 9.5 割ほどまでに上昇した。

1.5　工夫点

変数をやりとりするディクショナリ関数を用いることで雑多な変数のやりとりが整理されてコーディングするうえでも大きく役に立った。また、関数ごとで class を作ることでニューラルネットワークを作成するときに必要な入力と出力だけを考えるだけで良くなったためデバッグのあたりもつけやすく何よりコンパクトになったため全体を見通しやすくなった。

1.6　問題点

class 化してそれぞれの関数を自由に使えるように work_4 にニューラルネットワークの中身を書き出していったが、そのニューラルネットワークを更に class 化して一つのものとして扱えばそれぞれの関数の比較も容易にできるようになったであろう。今回、一つ一つコメントアウトしたりなど細かな調整が必要であった。flag をある程度用意してそれぞれの関数を flag によって管理すればもっと容易に実験を進めることができたように思う。