2 層ニューラルネットワークのクラス

mono

ニューラルネットワークの学習の手順

ニューラルネットワークには**重み**と**バイアス**がある

このパラメータを適した値に調整することを**学習**とよぶ

学習の手順(1~3 の繰り返し)

1. ミニバッチ
2. 勾配の算出
3. パラメータの更新

ニューラルネットワークの学習の手順

- **ミニバッチ**
  - 訓練データの中からランダムに一部のデータを選ぶ（選ばれたデータをミニバッチと呼ぶ）
- **勾配の算出**
  - ミニバッチの損出関数を減らすために各重みパラメータの勾配を求める（勾配は損出関数の値を最も減らす方向を示す）
- **パラメータの更新**
  - 重みパラメータを勾配方向に微少量だけ更新する。

確率的勾配降下法（SGD）

SGD: Stochastic Gradient descent

- **確率的**に無作為に選び出したデータを用いて
- **勾配**を求めることで
- 損失関数の**最小化**を図る

2 層ニューラルネットワークのクラス

```
class TwoLayerNet:

    def __init__(self, input_size, hidden_size, output_size, weight_init_std=0.01):
        # 重みの初期化
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.randn(input_size, hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = weight_init_std * np.random.randn(hidden_size, output_size)
        self.params['b2'] = np.zeros(output_size)
.
.
.
```

__init__

- params['W1'],params['W2']：それぞれ 1,2 層目の重み
- params['b1'],params['b2']：それぞれ 1,2 層目のバイアス
- 重みはランダムな値で初期化
  - np.random.randn:平均 0、分散 1 の乱数
- バイアスは 0 で初期化

ちょっと動かしてみる

- 初期化

```
net= TwoLayerNet(input_size=784,hidden_size=100,output_size=10)

print(net.params['W1'].shape) #(784,100)
print(net.params['b1'].shape) #(100,)
print(net.params['W2'].shape) #(100,10)
print(net.params['b2'].shape) #(10,)
```

predict

- 推論処理をする

loss

- 損失関数を求める
- 交差エントロピー誤差(t は正解ラベル、y は NN の出力)

$$E = -\sum_k t_k \log y_k$$

ちょっと動かしてみる

- 推論処理

```
x=np.random.rand(100,784)
y=net.predict(x)
```

accuracy

- 認識精度を求める

```
def accuracy(self, x, t):
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        #正解ラベル
        t = np.argmax(t, axis=1)

        accuracy = np.sum(y == t) / float(x.shape[0])
        return accuracy
```

numerical_gradient

- 勾配を求める

```
def numerical_gradient(self, x, t):
        #損失関数
        loss_W = lambda W: self.loss(x, t)
        #勾配
        grads = {}
        grads['W1'] = numerical_gradient(loss_W, self.params['W1'])
        grads['b1'] = numerical_gradient(loss_W, self.params['b1'])
        grads['W2'] = numerical_gradient(loss_W, self.params['W2'])
        grads['b2'] = numerical_gradient(loss_W, self.params['b2'])
        return grads
```

gradient

- 勾配を求める（高速版）
- 誤差逆伝搬法（次の章で学ぶ）を用いて計算
- 今回は中身には触れないけど学習ではこっちを使います

ちょっと動かしてみる

- 勾配の計算(めちゃくちゃ遅い)

```
x=np.random.rand(100,784)
t=np.random.rand(100,10)
grads=net.numerical_gradient(x,t)
print(grads['W1'].shape) #(784,100)
print(grads['b1'].shape) #(100,)
print(grads['W2'].shape) #(100,10)
print(grads['b2'].shape) #(10,)
```

ちょっと動かしてみる

- 勾配の計算(高速版)

```
x=np.random.rand(100,784)
t=np.random.rand(100,10)
grads=net.gradient(x,t)
print(grads['W1'].shape) #(784,100)
print(grads['b1'].shape) #(100,)
print(grads['W2'].shape) #(100,10)
print(grads['b2'].shape) #(10,)
```