深層学習 day2

このレポートでは、深層学習 day1 と同じ用語と記法を用いる。

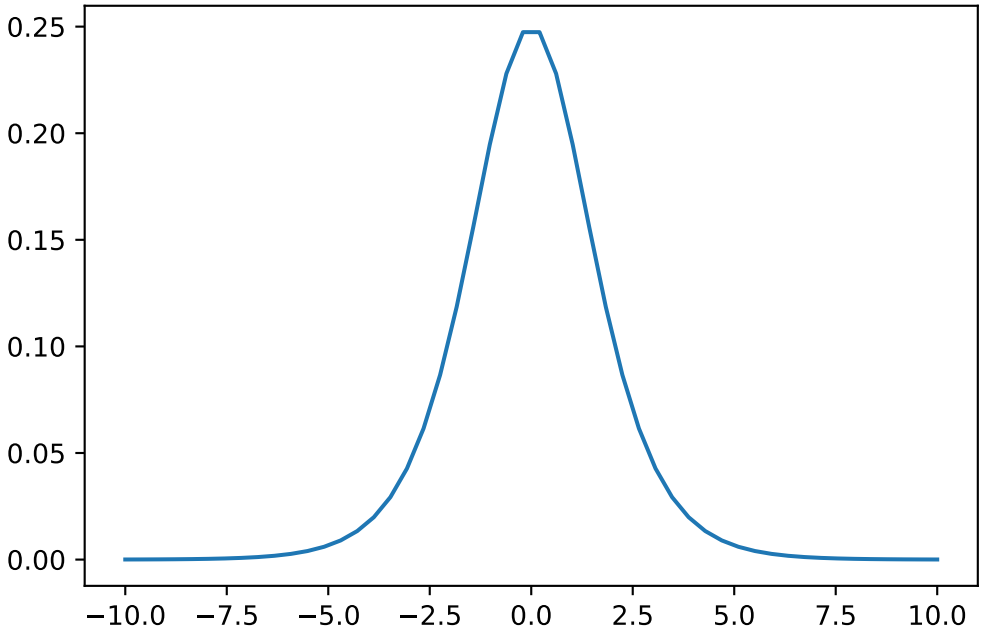
1　勾配消失問題

$\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$ をラベル付けされたデータとし、N をデータの数、$\mathbf{x}_i$ を m 次元特徴ベクトル、$\mathbf{y}_i$ を $\mathbf{x}_i$ のラベルとする。多層順伝搬型ニューラルネットワークを用いて、未知の m 次元特徴ベクトル $\mathbf{x}$ に対して、$\mathbf{x}$ のラベル $\mathbf{y}$ を予測することを試みる。$FN(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n)}, f_1, \dots, f_n) : \mathbb{R}^m \to \mathbb{R}^d$ を多層順伝搬型ニューラルネットワークとする。この多層順伝搬型ニューラルネットワークに出力層を追加することにより、多層順伝搬型ニューラルネットワーク $FN(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n+1)}, f_1, \dots, f_{n+1}) : \mathbb{R}^m \to \mathbb{R}^l$ を構成し、誤差関数 $E(\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n+1)})$ を構成する。記法の簡略化のため、行列のリスト $\mathbf{W}^{(1)}, \dots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \dots, \mathbf{b}^{(n+1)}$ を $\mathbf{W}$ で略記する。表記を簡素化するために、$\mathbf{b}^{(1)} = \cdots = \mathbf{b}^{(n+1)} = 0$ とする。このように仮定しても一般性を失わない（[7], p42 を参照）からである。最適な $\mathbf{W}$ は

$$\min_{\mathbf{W}} E(\mathbf{W}) \tag{1}$$

解くことによって求まる。これを行うために、用いられるアルゴリズムが誤差逆伝搬法である。誤差逆伝搬法では、誤差関数の偏微分を偏微分の連鎖律を用いて計算していく。そのため、ある層に伝搬する誤差は、それまでの全て層の誤差を用いて計算される。従って、ある層に対して、それまでの全て層の誤差が極端に小さいとき、その層に伝搬する誤差も極端に小さくなってしまう。その結果として、誤差関数の偏微分の結果が極端に小さくなり、$\mathbf{W}$ の更新がほとんど行われなくなってしまう。この問題を**勾配消失問題**という。ここでは、勾配消失問題の解決策をいくつか紹介する。

(1) 活性化関数の選択 : 各 f_i に対して、もしも f_i がシグモイド関数 $\sigma(x)$ ならば、f_i を ReLU 関数 $\mathrm{ReLU}(x)$ に変更する。何故ならば、$\sigma'(x)$ のグラフは



であり、$\sigma'(x)$ の値は極端に小さいのに対し、$\mathrm{ReLU}'(x)$ は

$$\mathrm{ReLU}'(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$$

となるからである。

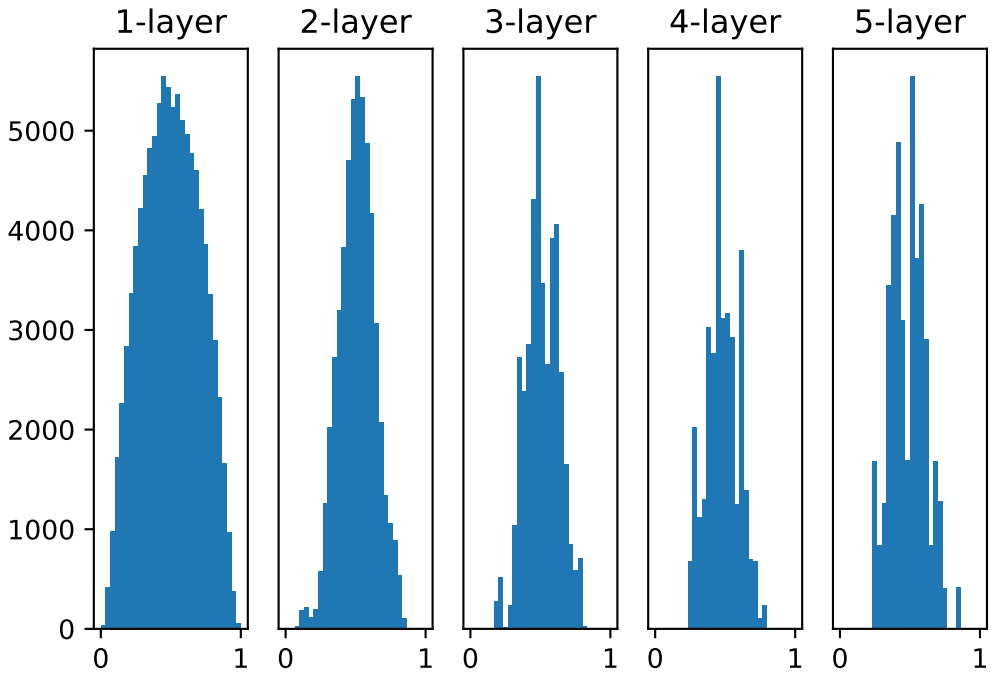
(2) 重みの初期値設定：各層 $FN(\mathbf{W}^{(l)}, f_l)$ の出力の値の分布が多様であれば、勾配消失問題は起きにくくなる。そこで、$\mathbf{W}^{(l)}$ の初期値を f_l に合わせて変える。

(1) f_l がシグモイド関数、双曲線正接関数である場合：n を $FN(\mathbf{W}^{(l-1)}, f_{l-1})$ の値域の次元（$l = 1$ のときは $n = m$ とする）とし、$\mathbf{W}^{(l)}$ の初期値を平均 0、標準偏差 $1/\sqrt{n}$ である正規分布から無作為に決める。この方法を **Xavier の初期値**という。

例 1.1. (コードは勾配消失問題.ipynb) Xavier の初期値によって、多層順伝搬型ニューラルネットワークの各層の出力の値が多様化する様子を観察する。

```
import numpy as np
import matplotlib.pyplot as plt
from google.colab import files

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

input_data = np.random.randn(1000, 100)  # 1000 個のデータ
node_num = 100  # 各隠れ層のノード（ニューロン）の数
```

```
hidden_layer_size = 5  # 隠れ層が 5 層
activations = {}  # ここにアクティベーションの結果を格納する

x = input_data

for i in range(hidden_layer_size):
    if i != 0:
        x = activations[i-1]

    w = np.random.randn(node_num, node_num) * np.sqrt(1.0 / node_num)
    a = np.dot(x, w)
    z = sigmoid(a)
    activations[i] = z

# ヒストグラムを描画
for i, a in activations.items():
    plt.subplot(1, len(activations), i+1)
    plt.title(str(i+1) + "-layer")
    if i != 0: plt.yticks([], [])
    # plt.xlim(0.1, 1)
    # plt.ylim(0, 7000)
    plt.hist(a.flatten(), 30, range=(0,1))
plt.show()
```



各層の出力の値の分布が多様であることが確認できる。

(2) f_l が ReLU 関数である場合：n を $FN(\mathbf{W}^{(l-1)}, f_{l-1})$ の値域の次元（$l=1$ のときは $n=m$ とする）とし、$\mathbf{W}^{(l)}$ の初期値を平均 0、標準偏差 $\sqrt{2/n}$ である正規分布から無作為に決める。この方法を **He の初期値**という。Xavier の初期値の場合と同様に、He の初期値を用いると各層の出力の値の分布が多様になる。

(3) バッチ正規化：ミニバッチの単位で入力値のデータの偏りを抑制することにより、各層の出力の値の分布を多様にする方法を**バッチ正規化**という。アルゴリズムは以下である。

(1) 学習係数 ϵ をとる。
(2) データ集合 $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$ をいくつかの**ミニバッチ** $B_1, \ldots, B_k$ に分割する。
(3) $\mathbf{W}^{(0)}$ を適当に決める。
(4) 無作為に B_j をとる。
(5) B_j の平均 μ と分散 σ^2 を計算し、任意の $\mathbf{x} \in B_j$ を $(\mathbf{x}-\mu)/\sqrt{\sigma^2+\alpha}$（$\alpha$ は小さな値とする）で置き換える（α を足すのは、0 による除算を防止するため）。
(6) $\nabla \frac{1}{|B_j|} \sum_{\mathbf{x}_i \in B_j} E_i(\mathbf{W}^{(t)})$ を計算する。
(7) $\mathbf{W}^{(t)}$ を $\mathbf{W}^{(t)} - \epsilon \nabla \frac{1}{|B_j|} \sum_{\mathbf{x}_i \in B_j} E_i(\mathbf{W}^{(t)})$ で置き換える。
(8) (4) に戻る。

2　学習率最適化手法

$\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$ をラベル付けされたデータとし、N をデータの数、$\mathbf{x}_i$ を m 次元特徴ベクトル、$\mathbf{y}_i$ を $\mathbf{x}_i$ のラベルとする。多層順伝搬型ニューラルネットワークを用いて、未知の m 次元特徴ベクトル $\mathbf{x}$ に対して、$\mathbf{x}$ のラベ

ル $\mathbf{y}$ を予測することを試みる。$FN(\mathbf{W}^{(1)},\dots,\mathbf{W}^{(n)},\mathbf{b}^{(1)},\dots,\mathbf{b}^{(n)},f_1,\dots,f_n):\mathbb{R}^m\to\mathbb{R}^d$ を多層順伝搬型ニューラルネットワークとする。この多層順伝搬型ニューラルネットワークに出力層を追加することにより、多層順伝搬型ニューラルネットワーク $FN(\mathbf{W}^{(1)},\dots,\mathbf{W}^{(n+1)},\mathbf{b}^{(1)},\dots,\mathbf{b}^{(n+1)},f_1,\dots,f_{n+1}):\mathbb{R}^m\to\mathbb{R}^l$ を構成し、誤差関数 $E(\mathbf{W}^{(1)},\dots,\mathbf{W}^{(n+1)},\mathbf{b}^{(1)},\dots,\mathbf{b}^{(n+1)})$ を構成する。記法の簡略化のため、行列のリスト $\mathbf{W}^{(1)},\dots,\mathbf{W}^{(n+1)},\mathbf{b}^{(1)},\dots,\mathbf{b}^{(n+1)}$ を $\mathbf{W}$ で略記する。表記を簡素化するために、$\mathbf{b}^{(1)}=\cdots=\mathbf{b}^{(n+1)}=0$ とする。このように仮定しても一般性を失わない（[7], p42 を参照）からである。最適な $\mathbf{W}$ は

$$\min_{\mathbf{W}} E(\mathbf{W}) \tag{2}$$

解くことによって求まる。勾配降下法は、(2) を解くためのアルゴリズムであり、以下のように定義される。

(1) ある正の正数 ϵ をとる。この ϵ を**学習率**という。
(2) $\mathbf{W}$ を適当に決める。
(3) $\nabla E(\mathbf{W})$ を計算する。
(4) $\mathbf{W}$ を $\mathbf{W}-\epsilon\nabla E(\mathbf{W})$ で置き換える。
(5) (3) に戻る。

このように始めに学習率 ϵ を決めて計算を行うことには以下の問題がある。

(1) 学習係数の値が大きい場合：最適値にいつまでもたどり着かず発散してしまう。
(2) 学習係数の値が小さい場合：
  (a) 発散することはないが、小さすぎると収束するまでに時間がかかってしまう。
  (b) 大局的最適値に収束しづらくなる。

これらの問題を解決するために考案されたのが、**学習率最適化手法**というアルゴリズムであり、このアルゴリズムでは、ϵ とは異なる学習率を導入し、それを学習状況に応じて変化させる。ここでは、いくつかの学習率最適化手法を紹介する。

(a) モメンタム：モメンタムは以下のアルゴリズムである。
  (1) ある正の正数 ϵ,α をとる。
  (2) $\mathbf{W}$ を適当に決める。
  (3) あるベクトル $\mathbf{v}$ を適当に決める。
  (4) $\nabla E(\mathbf{W})$ を計算する。
  (5) $\mathbf{v}$ を $\alpha\mathbf{v}-\epsilon\nabla E(\mathbf{W})$ で置き換える。
  (6) $\mathbf{W}$ を $\mathbf{W}+\mathbf{v}$ で置き換える。

例 2.1. (コードは学習率最適化手法.ipynb)

```
class Momentum:

    """Momentum SGD"""

    def __init__(self, lr=0.01, momentum=0.9):
        self.lr = lr
        self.momentum = momentum
```

```
        self.v = None

    def update(self, params, grads):
        if self.v is None:
            self.v = {}
            for key, val in params.items():
                self.v[key] = np.zeros_like(val)

        for key in params.keys():
            self.v[key] = self.momentum*self.v[key] - self.lr*grads[key]
            params[key] += self.v[key]
```

(b) AdaGrad：AdaGrad は以下のアルゴリズムである。

(1) ある正の正数 ϵ をとる。

(2) $\mathbf{W}$ を適当に決める。

(3) ある実数 $\mathbf{h}$ を適当に決める。

(4) $\nabla E(\mathbf{W})$ を計算する。

(5) $\mathbf{h}$ を $\mathbf{h} + (\nabla E(\mathbf{W}))^2$ で置き換える。

(6) $\mathbf{W}$ を $\mathbf{W} - \epsilon \frac{1}{\sqrt{\mathbf{h}}} \nabla E(\mathbf{W})$ で置き換える。

例 **2.2.** (コードは学習率最適化手法.ipynb)

```
class AdaGrad:

    """AdaGrad"""

    def __init__(self, lr=0.01):
        self.lr = lr
        self.h = None

    def update(self, params, grads):
        if self.h is None:
            self.h = {}
            for key, val in params.items():
                self.h[key] = np.zeros_like(val)

        for key in params.keys():
            self.h[key] += grads[key] * grads[key]
            params[key] -= self.lr * grads[key] / (np.sqrt(self.h[key]) + 1e-7)
```

(c) RMSProp：RMSProp は以下のアルゴリズムである。

(1) ある正の正数 ϵ, α をとる。

(2) $\mathbf{W}$ を適当に決める。
(3) ある実数 $\mathbf{h}$ を適当に決める。
(4) $\nabla E(\mathbf{W})$ を計算する。
(5) $\mathbf{h}$ を $\alpha\mathbf{h} + (1-\alpha)(\nabla E(\mathbf{W}))^2$ で置き換える。
(6) $\mathbf{W}$ を $\mathbf{W} - \epsilon\frac{1}{\sqrt{\mathbf{h}}}\nabla E(\mathbf{W})$ で置き換える。

例 **2.3.** (コードは学習率最適化手法.ipynb)

```
class RMSprop:

    """RMSprop"""

    def __init__(self, lr=0.01, decay_rate = 0.99):
        self.lr = lr
        self.decay_rate = decay_rate
        self.h = None

    def update(self, params, grads):
        if self.h is None:
            self.h = {}
            for key, val in params.items():
                self.h[key] = np.zeros_like(val)

        for key in params.keys():
            self.h[key] *= self.decay_rate
            self.h[key] += (1 - self.decay_rate) * grads[key] * grads[key]
            params[key] -= self.lr * grads[key] / (np.sqrt(self.h[key]) + 1e-7)
```

(d) Adam : Adam は以下のアルゴリズムである。
(1) ある正の正数 $\epsilon, \beta_1, \beta_2$ をとる。
(2) $\mathbf{W}$ を適当に決める。
(3) ある実数 $\mathbf{v}, \mathbf{s}$ を適当に決める。
(4) $\nabla E(\mathbf{W})$ を計算する。
(5) $\mathbf{v}$ を $\beta_1\mathbf{v} + (1-\beta_1)\nabla E(\mathbf{W})$ で置き換える。
(6) $\mathbf{s}$ を $\beta_2\mathbf{s} + (1-\beta_2)\nabla E(\mathbf{W})^2$ で置き換える。
(6) $\mathbf{W}$ を $\mathbf{W} - \epsilon\frac{\mathbf{v}}{\sqrt{\mathbf{s}}}$ で置き換える。

例 **2.4.** (コードは学習率最適化手法.ipynb)

```
class Adam:

    """Adam (http://arxiv.org/abs/1412.6980v8)"""
```

```
def __init__(self, lr=0.001, beta1=0.9, beta2=0.999):
    self.lr = lr
    self.beta1 = beta1
    self.beta2 = beta2
    self.iter = 0
    self.m = None
    self.v = None

def update(self, params, grads):
    if self.m is None:
        self.m, self.v = {}, {}
        for key, val in params.items():
            self.m[key] = np.zeros_like(val)
            self.v[key] = np.zeros_like(val)

    self.iter += 1
    lr_t  = self.lr * np.sqrt(1.0 - self.beta2**self.iter) / (1.0 - self.beta1**self.iter)

    for key in params.keys():
        #self.m[key] = self.beta1*self.m[key] + (1-self.beta1)*grads[key]
        #self.v[key] = self.beta2*self.v[key] + (1-self.beta2)*(grads[key]**2)
        self.m[key] += (1 - self.beta1) * (grads[key] - self.m[key])
        self.v[key] += (1 - self.beta2) * (grads[key]**2 - self.v[key])

        params[key] -= lr_t * self.m[key] / (np.sqrt(self.v[key]) + 1e-7)

        #unbias_m += (1 - self.beta1) * (grads[key] - self.m[key]) # correct bias
        #unbisa_b += (1 - self.beta2) * (grads[key]*grads[key] - self.v[key]) # correct bia
        #params[key] += self.lr * unbias_m / (np.sqrt(unbisa_b) + 1e-7)
```

3 過学習

$D = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$ をラベル付けされたデータとし、N をデータの数、$\mathbf{x}_i$ を m 次元特徴ベクトル、$\mathbf{y}_i$ を $\mathbf{x}_i$ のラベルとする。多層順伝搬型ニューラルネットワークを用いて、未知の m 次元特徴ベクトル $\mathbf{x}$ に対して、$\mathbf{x}$ のラベル $\mathbf{y}$ を予測することを試みる。$FN(\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(n)}, \mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(n)}, f_1, \ldots, f_n) : \mathbb{R}^m \to \mathbb{R}^d$ を多層順伝搬型ニューラルネットワークとする。この多層順伝搬型ニューラルネットワークに出力層を追加することにより、多層順伝搬型ニューラルネットワーク $FN(\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(n+1)}, f_1, \ldots, f_{n+1}) : \mathbb{R}^m \to \mathbb{R}^l$ を構成し、誤差関数 $E(\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(n+1)})$ を構成する。記法の簡略化のため、行列のリスト

$\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(n+1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{b}^{(n+1)}$ を $\mathbf{W}$ で略記する。表記を簡素化するために、$\mathbf{b}^{(1)} = \cdots = \mathbf{b}^{(n+1)} = 0$ とする。このように仮定しても一般性を失わない（[7], p42 を参照）からである。また、関数のリスト $f_1, \ldots, f_{n+1}$ を f で略記する。D を 2 つのデータ D_{train}, D_{test} へと（$|D_{train}| > |D_{test}|$ であるように）分割する。前者を**訓練データ**、後者を**テストデータ**という。D_{train} のデータに適合するように多層順伝搬型ニューラルネットワーク $FN(\mathbf{W}, f)$ を設計し、勾配降下法を D_{train}, D_{test} それぞれで行ったとする。D_{train} での誤差関数 $E(\mathbf{W})$ の値が学習が進むごとに減少するのに対し、D_{test} での誤差関数 $E(\mathbf{W})$ の値がある地点から減少しなくなったとき、$FN(\mathbf{W}, f)$ は**過学習**を起こしているという。ここでは、過学習を抑制する方法を紹介する。

(1) 正則化：勾配降下法を用いた結果、$\mathbf{W}$ の各パラメーターが大きな値をとってしまい、その結果として過学習が起こっている場合がある。この問題を解決する方法が**正則化**である。正則化では、誤差関数 $E(\mathbf{W})$ に正則化項 $\lambda \frac{1}{p} \|\mathbf{W}\|_p$（ただし、$\lambda$ は正の実数であり、$\|\|_p$ は p ノルム）を加算することで、$\mathbf{W}$ の各パラメーターの値の大きさを抑制する。$p = 1$ であるとき、$E(\mathbf{W}) + \lambda \frac{1}{p} \|\mathbf{W}\|_p$ を **L1 正則化**といい、$p = 2$ であるとき、**L2 正則化**という。

例 3.1. (コードは過学習.ipynb) L2 正則化が行われたニューラルネットを実装する。

```
import sys
sys.path.append("/content/drive/MyDrive/Colab Notebooks")
import numpy as np
from collections import OrderedDict
from common.layers import *
from common.gradient import numerical_gradient



class MultiLayerNet:
    """全結合による多層ニューラルネットワーク
    Parameters
    ----------
    input_size : 入力サイズ（MNIST の場合は 784）
    hidden_size_list : 隠れ層のニューロンの数のリスト（e.g. [100, 100, 100]）
    output_size : 出力サイズ（MNIST の場合は 10）
    activation : 'relu' or 'sigmoid'
    weight_init_std : 重みの標準偏差を指定（e.g. 0.01）
        'relu' または'he' を指定した場合は「He の初期値」を設定
        'sigmoid' または'xavier' を指定した場合は「Xavier の初期値」を設定
    weight_decay_lambda : Weight Decay（L2 ノルム）の強さ
    """
    def __init__(self, input_size, hidden_size_list, output_size,
                 activation='relu', weight_init_std='relu', weight_decay_lambda=0):
        self.input_size = input_size
```

```
        self.output_size = output_size
        self.hidden_size_list = hidden_size_list
        self.hidden_layer_num = len(hidden_size_list)
        self.weight_decay_lambda = weight_decay_lambda
        self.params = {}

        # 重みの初期化
        self.__init_weight(weight_init_std)

        # レイヤの生成
        activation_layer = {'sigmoid': Sigmoid, 'relu': Relu}
        self.layers = OrderedDict()
        for idx in range(1, self.hidden_layer_num+1):
            self.layers['Affine' + str(idx)] = Affine(self.params['W' + str(idx)],
                                                      self.params['b' + str(idx)])
            self.layers['Activation_function' + str(idx)] = activation_layer[activation]()

        idx = self.hidden_layer_num + 1
        self.layers['Affine' + str(idx)] = Affine(self.params['W' + str(idx)],
            self.params['b' + str(idx)])

        self.last_layer = SoftmaxWithLoss()

    def __init_weight(self, weight_init_std):
        """重みの初期値設定
        Parameters
        ----------
        weight_init_std : 重みの標準偏差を指定（e.g. 0.01）
            'relu'または'he'を指定した場合は「Heの初期値」を設定
            'sigmoid'または'xavier'を指定した場合は「Xavierの初期値」を設定
        """
        all_size_list = [self.input_size] + self.hidden_size_list + [self.output_size]
        for idx in range(1, len(all_size_list)):
            scale = weight_init_std
            if str(weight_init_std).lower() in ('relu', 'he'):
                scale = np.sqrt(2.0 / all_size_list[idx - 1])  # ReLUを使う場合に
推奨される初期値
            elif str(weight_init_std).lower() in ('sigmoid', 'xavier'):
                scale = np.sqrt(1.0 / all_size_list[idx - 1])  # sigmoidを使う場
合に推奨される初期値
```

```
            self.params['W' + str(idx)] = scale * np.random.randn(all_size_list[idx-1], all_siz
            self.params['b' + str(idx)] = np.zeros(all_size_list[idx])

    def predict(self, x):
        for layer in self.layers.values():
            x = layer.forward(x)

        return x

    def loss(self, x, t):
        """損失関数を求める
        Parameters
        ----------
        x : 入力データ
        t : 教師ラベル
        Returns
        -------
        損失関数の値
        """
        y = self.predict(x)

        weight_decay = 0
        for idx in range(1, self.hidden_layer_num + 2):
            W = self.params['W' + str(idx)]
            weight_decay += 0.5 * self.weight_decay_lambda * np.sum(W ** 2)

        return self.last_layer.forward(y, t) + weight_decay

    def accuracy(self, x, t):
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        if t.ndim != 1 : t = np.argmax(t, axis=1)

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

    def numerical_gradient(self, x, t):
        """勾配を求める（数値微分）
        Parameters
```

```
        ----------
        x : 入力データ
        t : 教師ラベル
        Returns
        -------
        各層の勾配を持ったディクショナリ変数
            grads['W1']、grads['W2']、...は各層の重み
            grads['b1']、grads['b2']、...は各層のバイアス
        """
        loss_W = lambda W: self.loss(x, t)

        grads = {}
        for idx in range(1, self.hidden_layer_num+2):
            grads['W' + str(idx)] = numerical_gradient(loss_W, self.params['W' + str(idx)])
            grads['b' + str(idx)] = numerical_gradient(loss_W, self.params['b' + str(idx)])

        return grads

    def gradient(self, x, t):
        """勾配を求める（誤差逆伝搬法）
        Parameters
        ----------
        x : 入力データ
        t : 教師ラベル
        Returns
        -------
        各層の勾配を持ったディクショナリ変数
            grads['W1']、grads['W2']、...は各層の重み
            grads['b1']、grads['b2']、...は各層のバイアス
        """
        # forward
        self.loss(x, t)

        # backward
        dout = 1
        dout = self.last_layer.backward(dout)

        layers = list(self.layers.values())
        layers.reverse()
        for layer in layers:
```

```
        dout = layer.backward(dout)

    # 設定
    grads = {}
    for idx in range(1, self.hidden_layer_num+2):
        grads['W' + str(idx)] = self.layers['Affine' + str(idx)].dW + self.weight_decay_lam
        grads['b' + str(idx)] = self.layers['Affine' + str(idx)].db

    return grads
```

(2) ドロップアウト：$\mathbf{W}$ のパラメーターの個数が多すぎることが原因で過学習が起こっている場合がある。この問題を解決する方法が**ドロップアウト**である。ドロップアウトでは、勾配降下法を行う際、パラメーターを無作為に削除する。これ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。

4　畳み込みニューラルネットの概念

$D = \{(\mathbf{X}_i, \mathbf{y}_i)\}_{i=1}^N$ をラベル付けされたデータとし、N をデータの数、$\mathbf{X}_i$ を画像、$\mathbf{y}_i$ を $\mathbf{X}_i$ のラベルとする。**畳み込みニューラルネット**は未知の画像 $\mathbf{X}$ に対して、$\mathbf{X}$ のラベル $\mathbf{y}$ を予測するための深層学習モデルである。任意の $\mathbf{X} \in D$ に対して、画像 $\mathbf{X}$ の縦横の画素数を $W \times W$ とし、$\mathbf{X}$ のチャンネル数を K とする。$\mathbf{X}$ の k 番目のチャンネルにおける (i, j) 成分を x_{ijk} で表すことにする。各 i, j, k の値はそれぞれ 0 から $W-1, W-1, K-1$ までの値を取るとする。以後、縦横の画素数が $W \times W$ であり、チャンネル数が K である画像全体の集合を $I(W, K)$ で表すことにする。$H_1, \ldots, H_M$ を縦横の画素数が $H \times H$ であり、チャンネル数が K である画像、$b_1, \ldots, b_M$ を実数、$f : \mathbb{R} \to \mathbb{R}$ を関数とする。H_m の k 番目のチャンネルにおける (p, q) 成分を h_{pqkm} で表すことにする。このとき、$CN(H_1, \ldots, H_M, b_1 \ldots, b_M, f) : I(W, K) \to I(W, M)$ を

$$CN(H_1, \ldots, H_M, b_1 \ldots, b_M, f)(\mathbf{X}) = (f(u_{ijm})), \quad u_{ijm} = \sum_{k=0}^{K-1} \sum_{p=0}^{H-1} \sum_{q=0}^{H-1} x_{i+p,j+q,k} h_{pqkm} + b_m$$

で定義する。$CN(H_1, \ldots, H_M, b_1 \ldots, b_M, f)$ を**畳み込み層**という。畳み込み層が途中で用いられている多層順伝搬型ニューラルネットワークを**畳み込みニューラルネットワーク**という。次に、**最大プーリング層** $P(H) : I(W, K) \to I(W/H, K)$（$H$ は W を割り切る自然数）を

$P(H)(\mathbf{X}) = (u_{ijk}), \quad u_{ijk} = \max_{(p,q) \in P_{ij}} x_{pqk}$（$P_{ij}$ は成分 (i, j) を中心とする $H \times H$ 領域に含まれる成分全体の集合）

で定義し、**平均プーリング層** $P(H) : I(W, K) \to I(W/H, K)$（$H$ は W を割り切る自然数）を

$P(H)(\mathbf{X}) = (u_{ijk}), \quad u_{ijk} = \dfrac{1}{H^2} \displaystyle\sum_{(p,q) \in P_{ij}} x_{pqk}$（$P_{ij}$ は成分 (i, j) を中心とする $H \times H$ 領域に含まれる成分全体の集合）

で定義する。これらは通常畳み込み層の後に用いられる。

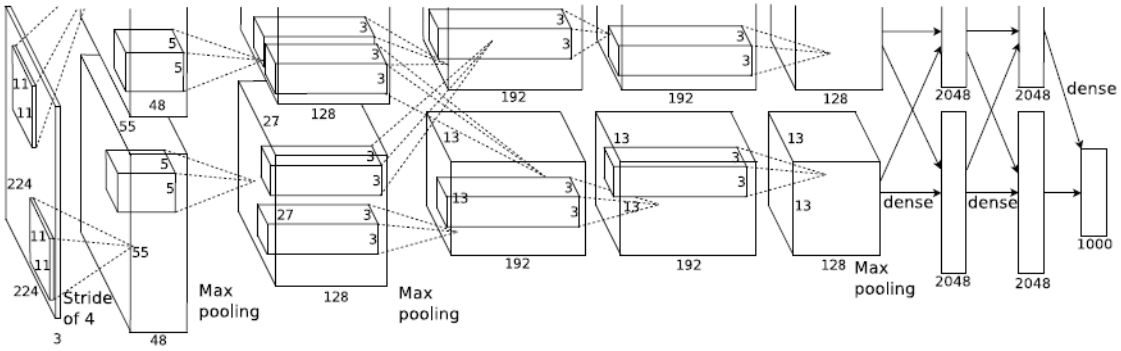
例 4.1. (コードは畳み込みニューラルネットの概念.ipynb) keras を用いて、畳み込みニューラルネットを実装する。

```
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.summary()
>Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d_3 (Conv2D)            (None, 148, 148, 32)      896

 max_pooling2d_2 (MaxPooling  (None, 74, 74, 32)       0
 2D)

 conv2d_4 (Conv2D)            (None, 72, 72, 64)        18496

 max_pooling2d_3 (MaxPooling  (None, 36, 36, 64)       0
 2D)

 conv2d_5 (Conv2D)            (None, 34, 34, 128)       73856

 max_pooling2d_4 (MaxPooling  (None, 17, 17, 128)      0
 2D)

 conv2d_6 (Conv2D)            (None, 15, 15, 128)       147584

 max_pooling2d_5 (MaxPooling  (None, 7, 7, 128)        0
```

```
 2D)

 flatten (Flatten)           (None, 6272)              0

 dense (Dense)               (None, 512)               3211776

 dense_1 (Dense)             (None, 1)                 513

=================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
_________________________________________________________________
```

5　最新の CNN

この章では、最新の CNN である AlexNet[5] について紹介する。AlexNet は 5 層の畳み込み層およびプーリング層など、それに続く 3 層の全結合層から構成される。



構成は以下のようになっている。最初の畳み込み層は $224 \times 224 \times 3$ の画像を 96 個の $11 \times 11 \times 3$ のフィルターをストライド 4 で用いて畳み込みを行う。2 番目の畳み込み層は、最初の畳み込み層の出力を最大プーリングかつ正規化した画像を 256 個の $5 \times 5 \times 48$ のフィルターを用いて畳み込みを行う。残りの 3、4、5 番目の畳み込み層では、プーリングと正規化は行われない。3 番目の畳み込み層は 2 番目の畳み込み層の出力を最大プーリングかつ正規化した画像を 384 個の $3 \times 3 \times 256$ のフィルターを用いて畳み込みを行い、4 番目の畳み込み層は 384 個の $3 \times 3 \times 192$ のフィルターを用いて畳み込みを行い、5 番目の畳み込み層は 256 個の $3 \times 3 \times 192$ のフィルターを用いて畳み込みを行う。全結合層はそれぞれにユニットに対して、4096 個のニューロンを持つ。過学習を避けるために、data augmentation を用いて、画像データの個数を割増し、学習の際はドロップアウトを用いている。

参考文献

[1] Andriy Burkov. (2019). The hundred-page machine learning book.

[2] Francois Chollet. (2018). Deep learning with python. Manning Publications Co.

[3] Marc Peter Deisenroth., A. Aldo Faisal., Cheng Soon Ong. (2020). Mathematics for machine learning. Cambridge University Press.

[4] Aurëlien Gëron. (2019). Hands-on machine learning with Scikit-Learn, Keras & TensorFlow. 2nd Edition. Oreilly.

[5] Alex Krizhevsky., Ilya Sutskever., Geoffrey E. Hinton. (2012). ImageNet Classification with Deep Convolutional Neural Networks. Advances in Neural Information Processing Systems 25.

[6] 小縣信也., 斎藤翔汰., 溝口聡., 若杉一幸. (2021). ディープラーニング E 資格エンジニア問題集. インプレス.

[7] 岡谷貴之. (2015). 深層学習. 講談社.

[8] Sebastian Raschka., Vahid Mirjalili. (2019). Python machine learning. Third Edition. Packt.

[9] 斎藤康毅. (2016). ゼロから作る deep learning. オライリージャパン.

[10] 斎藤康毅. (2018). ゼロから作る deep learning 2. オライリージャパン.

[11] 瀧雅人. (2017). これならわかる深層学習. 講談社.