第7章

VAE

この章では、VAE(Variational Autoencoder) についてその仕組みと実装を解説します。VAEでは、画像などの特徴を潜在変数という変数に圧縮するのですが、この潜在変数を調整することで生成する画像を連続的に変化させることができます。

本章はVEAの概要の解説から始めて、その仕組み、通常のオートエンコーダの実装方法、VAEの実装方法を解説していきます。

なお、本書は実装を重視し、VAEの確率モデルによる解説は行いませんのでご注意ください。確率モデルによる解説は、他の書籍などを参考にしてください。

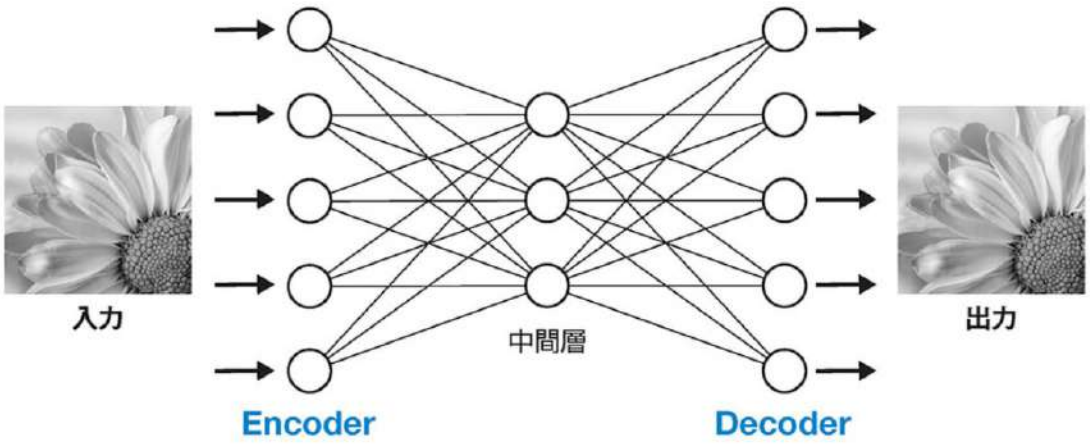
7.1 VAEの概要

通常のオートエンコーダ、そしてその発展形であるVAEについて概要を解説します。

7.1.1 オートエンコーダの概要

VAEは、オートエンコーダ（autoencoder、自己符号化器）と呼ばれるニューラルネットワークの発展形なので、まずはオートエンコーダについて解説します。オートエンコーダは、以下の図に示すようにEncoderとDecoderで構成されています。

オートエンコーダ



こちらの図の例では、入力は画像であり、出力はそれを再現した画像となっています。入力と出力のサイズは同じで、中間層のサイズはそれらよりも小さくなっています。出力が入力を再現する様にネットワークは学習しますが、中間層のサイズは入力よりも小さいのでEncoderによりデータが圧縮されることになります。そして、Decoderは圧縮されたデータから元のデータを復元しようとします。入力データが画像である場合、中間層は元の画像よりも少ないデータ量で画像の特徴を保持できることになります。

このように、オートエンコーダではいわばニューラルネットワークによる入力の圧縮と復元が行われます。教師データが必要ないので、いわゆる教師なし学習に分類されます。

オートエンコーダは、入力と出力の差分をとることで異常な値を検出することがで

きるので、産業における異常検知などに利用されています。

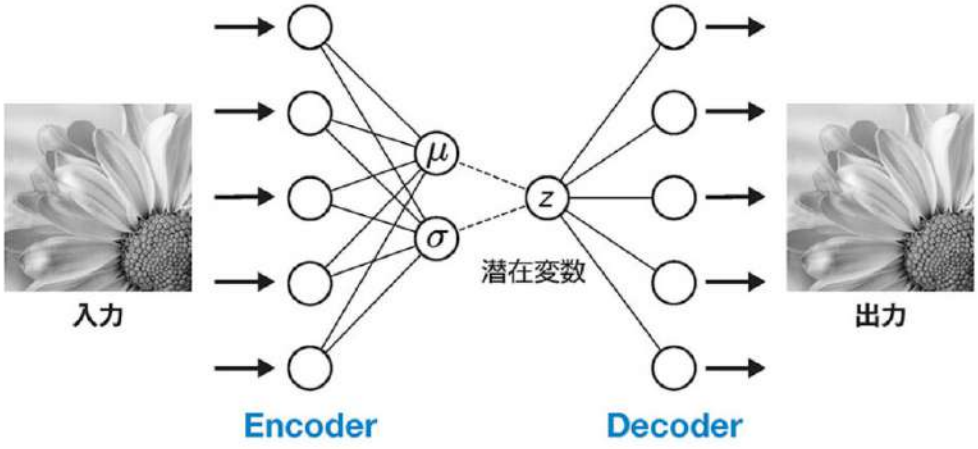
7.1.2 VAEの概要

生成モデル(Generative model)とは、訓練データを学習し、それらのデータと似たような新しいデータを生成するモデルのことです。ディープラーニングの用途は何かの識別だけでありません。生成モデルを用いると、データを生成することができます。

本書では、生成モデルとしてVAEとGANの2種類を扱いますが、本章ではこのうちVAEの方を解説します。 VAE(Variational autoencoder、変分自己符号化器)は、潜在変数と呼ばれる変数を利用することで、訓練データの特徴をうまく捉えて訓練データに似たデータを生成することができます(→参考文献[15])。

VAEはオートエンコーダの発展形で、以下の図で示すネットワーク構造をしています。

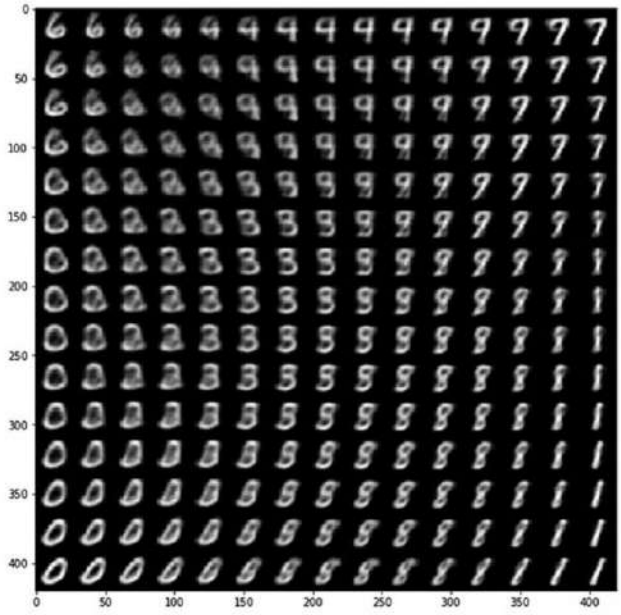
VAEのネットワーク構造



VAEでは、まずEncoderにより入力から平均ベクトルμと分散ベクトルσを求めます。これらを元に潜在変数zが確率的にサンプリングされ、zからDecoderにより元のデータが再現されます。

VAEの特徴の1つは、潜在変数zを調整することで連続的に変化するデータを生成できることです。たとえば、以下の図に示すような連続的に変化する手書き文字の画像を生成することができます。

VAEによって生成された手書き文字画像



この図では、潜在変数zを連続的に変化させることで、6や9、7などの数字が連続的に生成されています。

これを応用することで、たとえば人の表情を連続的に生成することなども可能になります。このようにVAEは柔軟性が高く、連続性を表現できるので注目を集めている生成モデルです。

VAEは、オートエンコーダと異なり、潜在変数の部分が確率分布になる、という特徴があります。これによる利点の1つは、同じ入力でも毎回異なる出力となることです。このためノイズに対して頑強になり、本質的な特徴を抽出する能力が向上します。また、未知の入力に対する挙動の担保にもつながります。

そして、潜在変数が連続的な分布であるために、潜在変数を調整することで出力の特徴を調整することが可能です。実際に、VAEはノイズの除去や、異常検知における異常箇所の特定、潜在変数を利用したクラスタリングなどに有用であり、活用されています。

VAEの実装はここまでの章を学んできた方であれば決して難しくはありませんので、気軽にトライしてみましょう。

7.2 VAEの仕組み

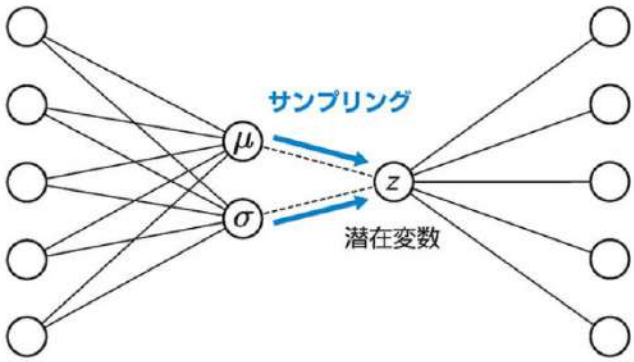
VAEの仕組みについて解説します。潜在変数のサンプリング、バックプロパゲーションに必要なReparametrization Trick、そしてVAEの誤差関数を把握しておきましょう。

7.2.1 潜在変数のサンプリング

VAEでは、潜在変数のサンプリングを行います。潜在変数は、入力の特徴をEncoderを使ってより低い次元に押し込めたものです。この潜在変数をDecoderで処理することにより、入力が再構築されます。

Encoderのニューラルネットワークは、以下の図にあるように平均値μと標準偏差σを出力し、これらを使った正規分布により潜在変数zをサンプリングします。

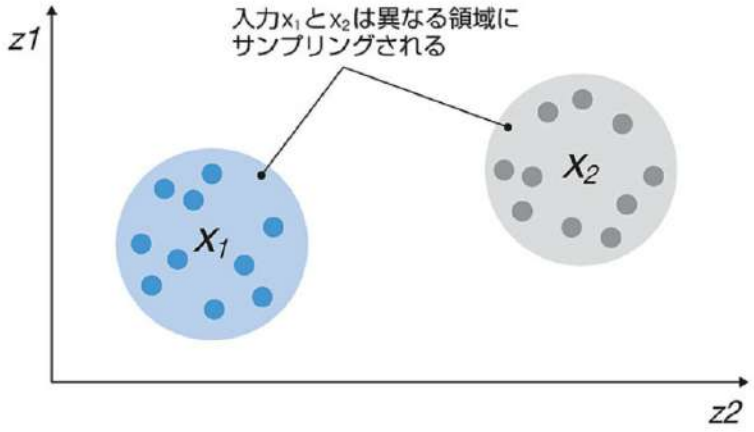
潜在変数のサンプリング



なお、この図からはわかりにくいですが、通常、μ、σ、そして潜在変数zはベクトル、あるいはバッチを考慮した場合は行列になります。

以上により、入力が同じであっても、毎回少し異なる潜在変数が得られることになります。以下の図は入力と潜在変数の関係です。簡単にするために、潜在変数を$z1$、$z2$の2つのみとしています。

■入力と潜在変数の関係



この図にはX_1とX_2の2つの入力がありますが、それぞれ異なる潜在変数の領域に正規分布に従ってサンプリングされます。これは入力が変化するとμとσが変化するためです。VAEを訓練すれば、入力の特徴ごとに異なる潜在変数の領域に、マッピングが行われるようになります。

7.2.2 Reparametrization Trick

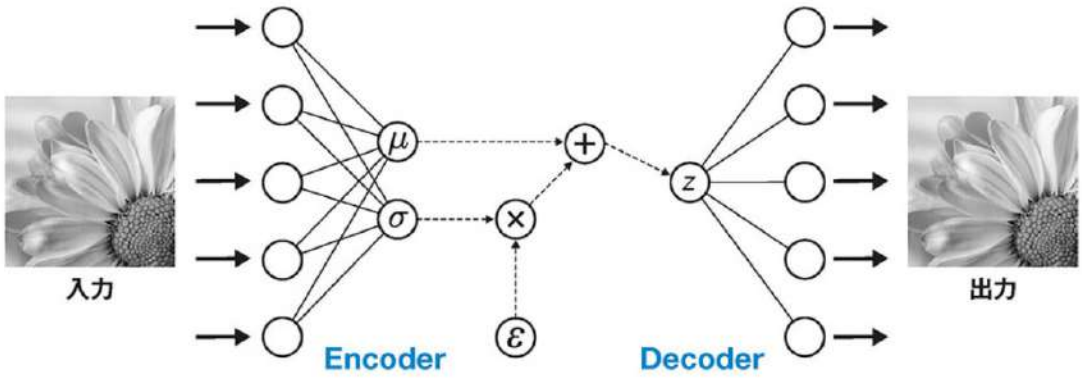
VAEは入力を再現するように学習するのですが、確率分布によるサンプリングが間に挟まっており、偏微分ができないのでそのままではバックプロパゲーションを適用できません。そこで、VAEではReparametrization Trickという方法が使われます。Reparametrization Trickでは、平均値0標準偏差1の正規分布からサンプリングされた値ϵを使って以下のように潜在変数を表します。

$$z = \mu + \epsilon\sigma$$

この式では、ϵに標準偏差σをかけて、平均値μに足して潜在変数としています。潜在変数が和と積の形で表されるので、偏微分が可能になりバックプロパゲーションを適用できるようになります。実際は潜在変数の数は1つではなく、バッチも考慮する必要があるので、上記の式の各変数は行列として扱います。ϵは潜在変数やサンプルごとに異なる値をとります。

以上を以下の図で表します。

Reparametrization Trick



μとσは、Encoderのニューラルネットワークの出力になります。そして、ϵを使って得られた潜在変数zは、Decoderの入力になります。

ϵは順伝播の際にサンプリングされますが、その値を逆伝播で使用するためその間保持しておく必要があります。

7.2.3 誤差の定義

VAEにバックプロパゲーションを適用するためには、誤差の定義が必要です。VAEの誤差は、「入力を再構築したものがどれだけ入力からずれているか」に影響を受ける必要がありますが、同時に「潜在変数がどれだけ発散しているか」に影響を受ける必要があります。

潜在変数には、学習が進むと0から離れ散らばってしまう傾向があります。このような発散が起きると、潜在変数の見通しが悪くなり扱いにくくなるので望ましくはありません。そこで、発散を防ぐためにVAEの誤差には正則化項E_{reg}が加えられます。これは、大まかに言って潜在変数がどれだけ発散しているか、を表します。これと、出力が入力からどれだけずれているかを表す再構成誤差E_{rec}を合わせて、本書ではVAEの誤差を以下のように表します。

$$E = E_{rec} + E_{reg} \qquad \text{式07-01}$$

この誤差Eを最小化するようにVAEは学習することになりますが、右辺の2つの項がうまく均衡できるかどうかで学習の成否が決まります。それでは、この後右辺の各誤差について解説していきます。

7.2.4 再構成誤差

式07-01におけるVAEの再構成誤差E_{rec}は、以下の式でよく表されます。

$$E_{rec} = \frac{1}{h}\sum_{i=1}^{h}\sum_{j=1}^{m}(-x_{ij}\log y_{ij} - (1-x_{ij})log(1-y_{ij})) \qquad 式07\text{-}02$$

この式において、x_{ij}はVAEの入力、y_{ij}はVAEの出力、hはバッチサイズ、mは入力層、出力層のニューロン数になります。すべての入出力で総和をとり、バッチ内で平均をとります。

ここで、∑∑内を添字を省略して以下の通りに表します。

$$e_{rec} = -x\log y - (1-x)log(1-y)$$

これは、二値の「交差エントロピー」と呼ばれ、2つの値が離れている度合いを表します。この場合e_{rec}はxとyの値の隔たりの大きさを表しますが、xがyと等しいときに最小値をとります。

ここで、二値の交差エントロピーと以下の式で表される二乗誤差を比較します。

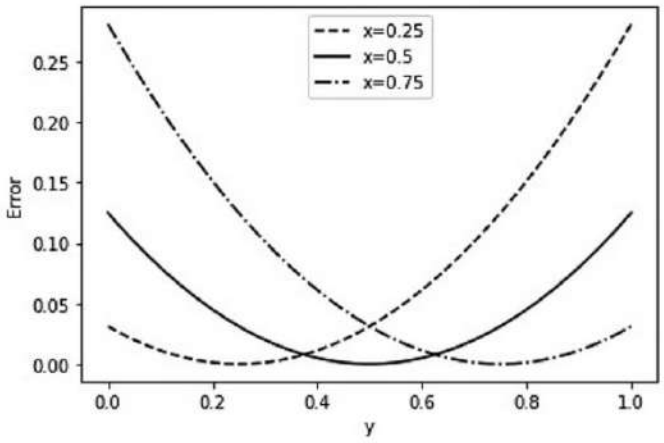
$$\frac{1}{2}(x-y)^2$$

直感的に把握するために両者をグラフで描画しますが、まずは二乗誤差を以下のコードで描画します。xの値が0.25、0.5、0.75のとき、yの値とともに二乗誤差がどう変化するのかを確認します。

二乗誤差の変化

```
import numpy as np
import matplotlib.pyplot as plt

def square_error(x, y):
    return (x - y)**2/2  # 二乗誤差

y = np.linspace(0, 1)
xs = [0.25, 0.5, 0.75]
for x in xs:
    plt.plot(y, square_error(x, y), label="x="+str(x))
```

```
plt.legend()
plt.xlabel("y")
plt.ylabel("Error")
plt.show()
```



※わかりやすいように一部線種を変更しています。

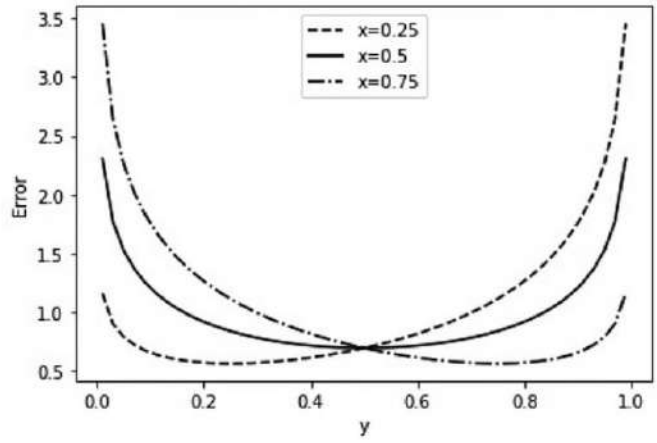
二乗誤差の場合、最小値の両側でなだらかに誤差が立ち上がることが確認できます。

次に、二値の交差エントロピー誤差を以下のコードで描画します。xの値が0.25、0.5、0.75のとき、yの値とともに二値の交差エントロピー誤差がどう変化するのかを確認します。

二値の交差エントロピー誤差の変化

```
import numpy as np
import matplotlib.pyplot as plt

# 二値の交差エントロピーを返す
def binary_crossentropy(x, y):
    return -x*np.log(y) - (1-x)*np.log(1-y)

y = np.linspace(0.01, 0.99)
xs = [0.25, 0.5, 0.75]
for x in xs:
    plt.plot(y, binary_crossentropy(x, y), label="x="+str(x))

plt.legend()
plt.xlabel("y")
```

```
plt.ylabel("Error")
plt.show()
```



※わかりやすいように一部線種を変更しています。

最小値付近は誤差がなだらかに変化しますが、0と1の付近で誤差は急激に立ち上がります。xとyに大きな隔たりが生じたとき誤差が急激に増大するので、誤差を小さくしようとする働きも大きくなります。

二値の交差エントロピーが適用可能なのは、xとyの範囲が0から1の範囲の場合に限られます。しかしながら、二乗誤差と比較して誤差の変化の緩急差が大きいため、誤差を収束させやすくなっています。このため、VAEの再構成誤差には二値の交差エントロピーがよく使われます。

式07-02ではすべての入出力でこのような二値の交差エントロピーの総和をとり、バッチ内で平均をとることで、入力の再現度合いを表す再構成誤差としています。

7.2.5 正則化項

式07-01におけるVAEの正則化項E_{reg}は、以下の式のようによく表されます。

$$E_{reg} = \frac{1}{h}\sum_{i=1}^{h}\sum_{k=1}^{n} -\frac{1}{2}(1 + \log\sigma_{ik}^2 - \mu_{ik}^2 - \sigma_{ik}^2) \qquad \text{式07-03}$$

この式において、hはバッチサイズ、nは潜在変数の数、σ_{ij}は標準偏差、μ_{ij}は平均値です。すべての潜在変数で総和をとり、バッチ内で平均をとります。

この項の$\sum\sum$の内部について考えましょう。添字を省略して以下の通りに表します。

$$e_{reg} = -\frac{1}{2}(1 + \log\sigma^2 - \mu^2 - \sigma^2)$$

e_{reg}は、標準偏差σが1、平均値μ_{ik}が0のとき最小値の0をとります。そして、e_{reg}はσが1を離れるか、μが0を離れると大きくなります。σとμで潜在変数がサンプリングされるので、e_{reg}は「潜在変数がどれだけ標準偏差1、平均値0から離れているか」を表すことになります。

式07-03ではすべての入出力でこのようなe_{reg}の総和をとり、バッチ内で平均をとることで、潜在変数の発散度合いを表す正則化項としています。

7.3 オートエンコーダの実装

VAEの実装に入る前に、通常のオートエンコーダの実装について解説します。Encoderで中間層に画像を圧縮した後に、Decoderで元の画像を再構築します。

訓練データには、以前に扱ったscikit-learnの手書き数字画像を使用します。

7.3.1 構築するネットワーク

以下の図は、今回構築するオートエンコーダのネットワークです。画像の幅と高さが8ピクセルなので、入力層には8×8=64のニューロンが必要になります。

構築するオートエンコーダのネットワーク

nはニューロンの数

また、出力が入力を再現するように学習するので、出力層のニューロン数は入力層と同じになります。

中間層にはこれらよりも少ない16のニューロンを配置します。これは画像のピクセル数より少ないので、画像が圧縮されることになります。

コードでは、上記を以下の通りに設定します。

```
img_size = 8  # 画像の高さと幅
n_in_out = img_size * img_size  # 入出力層のニューロン数
n_mid = 16    # 中間層のニューロン数
```

7.3.2 各層の実装

中間層と出力層を実装します。中間層の活性化関数にはReLUを使用します。

入力の値は0から1の範囲なのですが、出力の範囲はこれに合わせる必要があります。このため、出力層の活性化関数には出力範囲が0から1の間であるシグモイド関数を使用します。

各層のクラス定義

```
# -- 各層の継承元 --
class BaseLayer:
    def update(self, eta):
        self.w -= eta * self.grad_w
        self.b -= eta * self.grad_b

# -- 中間層 --
class MiddleLayer(BaseLayer):
    def __init__(self, n_upper, n):
        # Heの初期値
        self.w = np.random.randn(n_upper, n) * np.sqrt(2/n_upper)
        self.b = np.zeros(n)

    def forward(self, x):
        self.x = x
        self.u = np.dot(x, self.w) + self.b
        self.y = np.where(self.u <= 0, 0, self.u) # ReLU

    def backward(self, grad_y):
```

```
        delta = grad_y * np.where(self.u <= 0, 0, 1)

        self.grad_w = np.dot(self.x.T, delta)
        self.grad_b = np.sum(delta, axis=0)
        self.grad_x = np.dot(delta, self.w.T)

# -- 出力層 --
class OutputLayer(BaseLayer):
    def __init__(self, n_upper, n):
        # Xavierの初期値
        self.w = np.random.randn(n_upper, n) / np.sqrt(n_upper)
        self.b = np.zeros(n)

    def forward(self, x):
        self.x = x
        u = np.dot(x, self.w) + self.b
        self.y = 1/(1+np.exp(-u))  # シグモイド関数

    def backward(self, t):
        delta = (self.y-t) * self.y * (1-self.y)

        self.grad_w = np.dot(self.x.T, delta)
        self.grad_b = np.sum(delta, axis=0)
        self.grad_x = np.dot(delta, self.w.T)
```

7.3.3 順伝播と逆伝播

各層を初期化し、順伝播と逆伝播の関数、およびパラメータ更新用の関数を定義します。middle_layerがEncoder、output_layerがDecoderです。

オートエンコーダなので、入力と出力のニューロン数は同じになります。

順伝播と逆伝播の実装コード

```
# -- 各層の初期化 --
middle_layer = MiddleLayer(n_in_out, n_mid)  # Encoder
output_layer = OutputLayer(n_mid, n_in_out)  # Decoder

# -- 順伝播 --
def forward_propagation(x_mb):
```

```
    middle_layer.forward(x_mb)
    output_layer.forward(middle_layer.y)

# -- 逆伝播 --
def backpropagation(t_mb):
    output_layer.backward(t_mb)
    middle_layer.backward(output_layer.grad_x)

# -- パラメータの更新 --
def update_params():
    middle_layer.update(eta)
    output_layer.update(eta)
```

7.3.4 ミニバッチ法の実装

ミニバッチ法により学習します。オートエンコーダなので、順伝播のforward_propagation関数に渡す入力と、逆伝播のbackpropagationに渡す正解は同じになります。

ミニバッチ法による学習

```
n_batch = len(x_train) // batch_size  # 1エポックあたりのバッチ数
for i in range(epochs):

    # -- 学習 --
    index_random = np.arange(len(x_train))
    np.random.shuffle(index_random)  # インデックスをシャッフルする
    for j in range(n_batch):

        # ミニバッチを取り出す
        mb_index = index_random[j*batch_size : (j+1)*batch_size]
        x_mb = x_train[mb_index, :]

        # 順伝播と逆伝播
        forward_propagation(x_mb)
        backpropagation(x_mb)

        # 重みとバイアスの更新
        update_params()
```

7.3.5 全体のコード

以下は全体のコードです。訓練データの用意、各層のクラス、順伝播と逆伝播の関数、ミニバッチ法が順次実装されています。

誤差は、各エポックの終了後に測定して表示します。また、学習の終了後に誤差の推移を表示します。

全体コードと実行結果（誤差の推移が表示される）

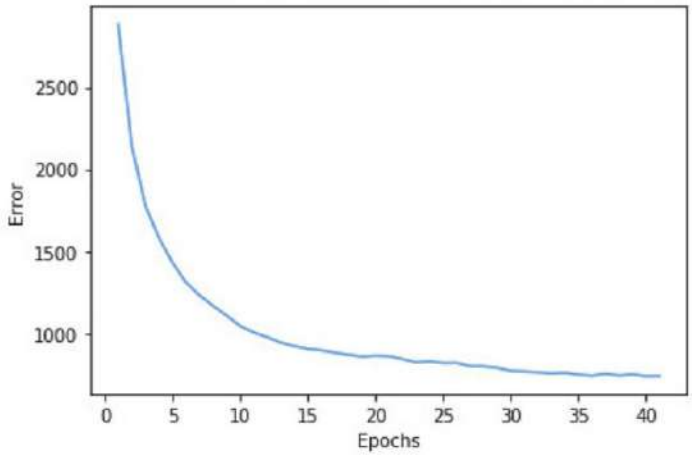
```
import numpy as np
# import cupy as np  # GPUの場合
import matplotlib.pyplot as plt
from sklearn import datasets

# -- 各設定値 --
img_size = 8   # 画像の高さと幅
n_in_out = img_size * img_size  # 入出力層のニューロン数
n_mid = 16     # 中間層のニューロン数

eta = 0.01     # 学習係数
epochs = 41
batch_size = 32
interval = 4   # 経過の表示間隔

# -- 訓練データ --
digits_data = datasets.load_digits()
x_train = np.asarray(digits_data.data)
x_train /= 15  # 0～1の範囲に

# -- 各層の継承元 --
class BaseLayer:
    def update(self, eta):
        self.w -= eta * self.grad_w
        self.b -= eta * self.grad_b

# -- 中間層 --
class MiddleLayer(BaseLayer):
    def __init__(self, n_upper, n):
        # Heの初期値
        self.w = np.random.randn(n_upper, n) * \
                 np.sqrt(2/n_upper)
```

```
        self.b = np.zeros(n)

    def forward(self, x):
        self.x = x
        self.u = np.dot(x, self.w) + self.b
        self.y = np.where(self.u <= 0, 0, self.u) # ReLU

    def backward(self, grad_y):
        delta = grad_y * np.where(self.u <= 0, 0, 1)

        self.grad_w = np.dot(self.x.T, delta)
        self.grad_b = np.sum(delta, axis=0)
        self.grad_x = np.dot(delta, self.w.T)

# -- 出力層 --
class OutputLayer(BaseLayer):
    def __init__(self, n_upper, n):
        # Xavierの初期値
        self.w = np.random.randn(n_upper, n) / np.sqrt(n_upper)
        self.b = np.zeros(n)

    def forward(self, x):
        self.x = x
        u = np.dot(x, self.w) + self.b
        self.y = 1/(1+np.exp(-u))  # シグモイド関数

    def backward(self, t):
        delta = (self.y-t) * self.y * (1-self.y)

        self.grad_w = np.dot(self.x.T, delta)
        self.grad_b = np.sum(delta, axis=0)
        self.grad_x = np.dot(delta, self.w.T)

# -- 各層の初期化 --
middle_layer = MiddleLayer(n_in_out, n_mid)  # Encoder
output_layer = OutputLayer(n_mid, n_in_out)  # Decoder

# -- 順伝播 --
def forward_propagation(x_mb):
    middle_layer.forward(x_mb)
    output_layer.forward(middle_layer.y)
```

```
# -- 逆伝播 --
def backpropagation(t_mb):
    output_layer.backward(t_mb)
    middle_layer.backward(output_layer.grad_x)

# -- パラメータの更新 --
def update_params():
    middle_layer.update(eta)
    output_layer.update(eta)

# -- 誤差を計算 --
def get_error(y, t):
    return 1.0/2.0*np.sum(np.square(y - t))  # 二乗和誤差

error_record = []
n_batch = len(x_train) // batch_size  # 1エポックあたりのバッチ数
for i in range(epochs):

    # -- 学習 --
    index_random = np.arange(len(x_train))
    np.random.shuffle(index_random)  # インデックスをシャッフルする
    for j in range(n_batch):

        # ミニバッチを取り出す
        mb_index = index_random[j*batch_size : (j+1)*batch_size]
        x_mb = x_train[mb_index, :]

        # 順伝播と逆伝播
        forward_propagation(x_mb)
        backpropagation(x_mb)

        # 重みとバイアスの更新
        update_params()

    # -- 誤差を求める --
    forward_propagation(x_train)
    error = get_error(output_layer.y, x_train)
    error_record.append(error)

    # -- 経過の表示 --
    if i%interval == 0:
        print("Epoch:"+str(i+1)+"/"+str(epochs),
```

```
                "Error:"+str(error))

    plt.plot(range(1, len(error_record)+1), error_record)
    plt.xlabel("Epochs")
    plt.ylabel("Error")
    plt.show()
```

```
Epoch:1/41 Error:2884.696765146869
Epoch:5/41 Error:1430.3853401275119
Epoch:9/41 Error:1106.5350345578545
Epoch:13/41 Error:941.9110134597945
Epoch:17/41 Error:879.0944965043436
Epoch:21/41 Error:858.8966310805197
Epoch:25/41 Error:816.7074772962773
Epoch:29/41 Error:787.5007695329515
Epoch:33/41 Error:752.9997137820637
Epoch:37/41 Error:751.1997352034167
Epoch:41/41 Error:738.4983322508115
```



誤差はなめらかに減少しているようです。それでは、オートエンコーダより再構築された画像を確認しましょう。

7.3.6 生成された画像の表示

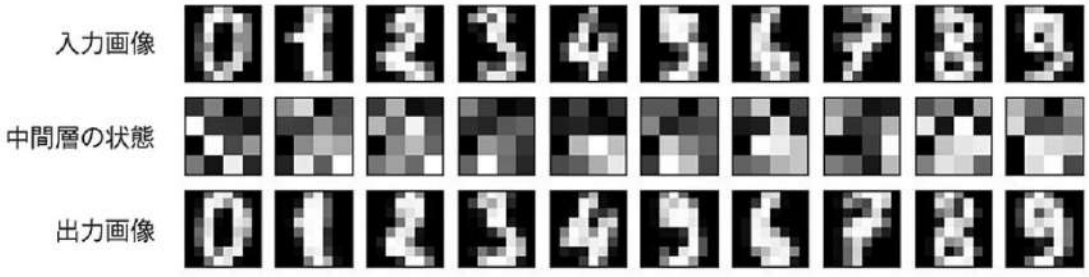
以下のコードにより、入力画像および再構築された画像を並べて表示します。また、中間層の出力も4×4の画像にして並べます。

画像が正しく再構築されているか、そしてその際に中間層がどのような状態にある

のかを確認します。

オートエンコーダによる画像の圧縮と再構築

```
n_img = 10  # 表示する画像の数
middle_layer.forward(x_train[:n_img])
output_layer.forward(middle_layer.y)

plt.figure(figsize=(10, 3))
for i in range(n_img):
    # 入力画像
    ax = plt.subplot(3, n_img, i+1)
    plt.imshow(x_train[i].reshape(img_size, -1).tolist(),
                                    cmap="Greys_r")
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # 中間層の出力
    ax = plt.subplot(3, n_img, i+1+n_img)
    plt.imshow(middle_layer.y[i].reshape(4, -1).tolist(),
               cmap="Greys_r")
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # 出力画像
    ax = plt.subplot(3, n_img, i+1+2*n_img)
    plt.imshow(output_layer.y[i].reshape(img_size,
            -1).tolist(), cmap="Greys_r")
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

plt.show()
```



多少の違いはありますが、出力は入力を再現した画像になっています。また、中間層は数字画像ごとに異なる状態にあることも確認できます。64ピクセルの画像を特徴付ける情報を、16の状態に圧縮できたことになります。

中間層のニューロン数がこれより少なくなるにつれ、次第にうまく画像を再構築できなくなります。興味のある方は、中間層におけるニューロン数の削減にトライしてみましょう。

中間層の状態から画像を復元することはできましたが、中間層と出力画像の対応関係を直感的に把握したり、中間層の状態を変化させて生成画像を調整することは難しそうです。これらを実現するためには、人間に意味が理解できて、制御が可能な少数の変数に画像を圧縮するのが望ましいです。

以降は、それを可能にするVAEの実装方法を解説していきます。

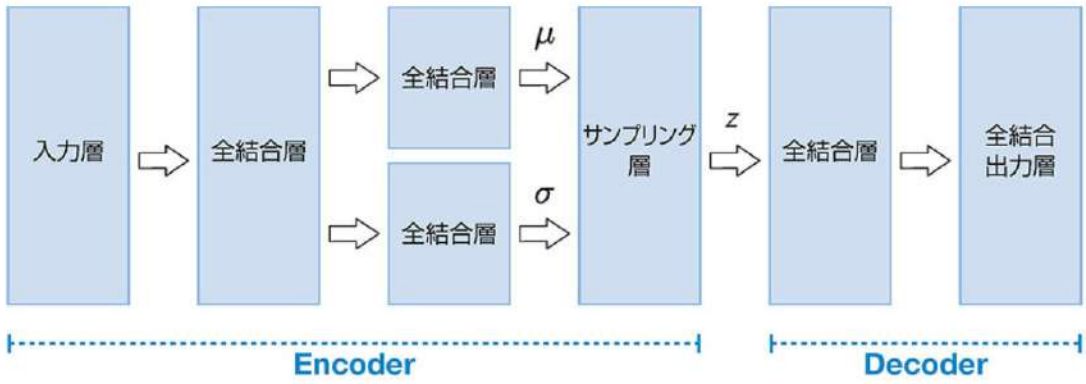
7.4 VAEに必要な層

VAEの実装に必要な層について解説します。

7.4.1 VAEの構成

以下の図は、VAEの構成の例です。

VAEの構成例



Encoderでは、ニューラルネットワークが2つに分岐します。そして、それぞれのニューラルネットワークが、それぞれ平均値と標準偏差を出力します。これらの標準

偏差と平均値から潜在変数がサンプリングされますが、このサンプリングを層として実装します。本章では、この層を「サンプリング層」と呼びます。サンプリングされた潜在変数は、Decoderのニューラルネットワークの入力となります。今回は、サンプリング層以外の層はすべて全結合層とします。

それでは、平均値と標準偏差を出力する層、およびサンプリング層、出力層の実装を見ていきましょう。

7.4.2 平均値、標準偏差を出力する層

平均値、標準偏差を出力する層を以下の通りに実装します。平均値、標準偏差ともに共通のクラスを使って実装します。この層の活性化関数には、恒等関数を使用します。

ParamsLayerクラスの定義

```
# -- 正規分布のパラメータを求める層 --
class ParamsLayer(BaseLayer):
    def __init__(self, n_upper, n):
        # Xavierの初期値
        self.w = np.random.randn(n_upper, n) / np.sqrt(n_upper)
        self.b = np.zeros(n)

    def forward(self, x):
        self.x = x
        u = np.dot(x, self.w) + self.b
        self.y = u  # 恒等関数

    def backward(self, grad_y):
        delta = grad_y

        self.grad_w = np.dot(self.x.T, delta)
        self.grad_b = np.sum(delta, axis=0)
        self.grad_x = np.dot(delta, self.w.T)
```

標準偏差を出力する層に関してですが、実装の都合上、層の出力は標準偏差そのものではなく標準偏差の2乗の対数、すなわち分散の対数を表すことにします(→参考文献[11])。これを以下の式で表します。

$$\phi = \log \sigma^2 \qquad \text{式07-04}$$

このφは対数なので、負の値もとることができて値の範囲に制限がありません。従って、活性化関数に恒等関数を使用することが可能になります。また、この形にすることで後述するサンプリング層の逆伝播がシンプルに表記できるようになります。

7.4.3 サンプリング層

サンプリング層はニューラルネットワークの層とは異なります。μとφを入力として潜在変数zを出力しますが、順伝播と逆伝播を実装する必要があります。この層に学習するパラメータはありません。

サンプリング層の順伝播は、以下のReparametrization Trickの式に基づいて行われます。

$$z = \mu + \epsilon\sigma$$

これは、**式07-04**のφを使うと以下のように表すことができます。

$$z = \mu + \epsilon \exp \frac{\phi}{2}$$

サンプリング層の逆伝播ですが、上の層に伝播させるために入力であるμとφの勾配を求める必要があります。逆伝播に使用する誤差関数ですが、誤差を小さくする目的に対してバッチサイズで割ることは不要なので以下の形で表します。

$$E = E_{rec} + E_{reg} \qquad \text{式07-05}$$

$$E_{rec} = \sum_{i=1}^{h}\sum_{j=1}^{m}(-x_{ij}\log y_{ij} - (1-x_{ij})log(1-y_{ij})) \qquad \text{式07-06}$$

$$\begin{aligned} E_{reg} &= \sum_{i=1}^{h}\sum_{k=1}^{n} -\frac{1}{2}(1 + \log\sigma_{ik}^2 - \mu_{ik}^2 - \sigma_{ik}^2) \\ &= \sum_{i=1}^{h}\sum_{k=1}^{n} -\frac{1}{2}(1 + \phi_{ik} - \mu_{ik}^2 - \exp\phi_{ik}) \end{aligned} \qquad \text{式07-07}$$

上記を踏まえて、μの勾配を以下のようにして求めます。簡単にするために添字は省略します。

$$\begin{aligned}\frac{\partial E}{\partial \mu} &= \frac{\partial}{\partial \mu}(E_{rec} + E_{reg}) \\ &= \frac{\partial E_{rec}}{\partial z}\frac{\partial z}{\partial \mu} + \frac{\partial E_{reg}}{\partial \mu} \\ &= \frac{\partial E_{rec}}{\partial z} + \mu\end{aligned}$$

ここで、$\frac{\partial E_{rec}}{\partial z}$はDecoderへの入力の勾配なので、Decoderからの逆伝播により得ることができます。

次にϕの勾配ですが、次のようにして求めます。同じく添字は省略します。

$$\begin{aligned}\frac{\partial E}{\partial \phi} &= \frac{\partial}{\partial \phi}(E_{rec} + E_{reg}) \\ &= \frac{\partial E_{rec}}{\partial z}\frac{\partial z}{\partial \phi} + \frac{\partial E_{reg}}{\partial \phi} \\ &= \frac{\partial E_{rec}}{\partial z}\frac{\epsilon}{2}\exp\frac{\phi}{2} - \frac{1}{2}(1 - \exp\phi)\end{aligned}$$

順伝播、逆伝播とにも行列積は必要ないので、行列による表記は省きます。

以上を踏まえて、以下の通りにサンプリング層をコードで実装します。

LatentLayerクラス

```
# -- 潜在変数をサンプリングする層 --
class LatentLayer:
    def forward(self, mu, log_var):
        self.mu = mu              # 平均値
        self.log_var = log_var    # 分散の対数

        self.epsilon = np.random.randn(*log_var.shape)
        self.z = mu + self.epsilon*np.exp(log_var/2)

    def backward(self, grad_z):
        self.grad_mu = grad_z + self.mu
        self.grad_log_var = grad_z*self.epsilon/2*np.exp( \
                self.log_var/2) - 0.5*(1-np.exp(self.log_var))
```

7.4.4 出力層

出力層の逆伝播では、以下のようにδが表されます。

$$\begin{aligned}\delta &= \frac{\partial E}{\partial u} \\ &= \frac{\partial E}{\partial y}\frac{\partial y}{\partial u}\end{aligned} \qquad \text{式07-08}$$

出力層の活性化関数にはシグモイド関数を使うので、ここで以下の式を代入します。

$$\frac{\partial y}{\partial u} = y(1-y)$$

また、**式07-06**も代入しますが、本書では正解をtで表してきたのでこれに合わせてxをtに置き換えます。

これらにより、**式07-08**は次の形になります。なお、添字は省略されています。

$$\begin{aligned}\delta &= \frac{\partial E}{\partial y}\frac{\partial y}{\partial u} \\ &= \frac{\partial}{\partial y}(E_{rec}+E_{reg})y(1-y) \\ &= (-\frac{t}{y}+\frac{1-t}{1-y})y(1-y) \\ &= -t(1-y)+(1-t)y \\ &= y-t\end{aligned}$$

δを求めることができたので、後はこれまでと同様にして各勾配を求めることができます。

上記の式のように、DecoderのδはE_{reg}の影響を受けないので、DecoderではE_{rec}のみを考慮すればいいことになります。

以上に基づき、出力層を以下の通りに実装します。

OutputLayerクラスの定義

```
# -- 出力層 --
class OutputLayer(BaseLayer):
    def __init__(self, n_upper, n):
```

```
        # Xavierの初期値
        self.w = np.random.randn(n_upper, n) / np.sqrt(n_upper)
        self.b = np.zeros(n)

    def forward(self, x):
        self.x = x
        u = np.dot(x, self.w) + self.b
        self.y = 1/(1+np.exp(-u))  # シグモイド関数

    def backward(self, t):
        delta = self.y - t

        self.grad_w = np.dot(self.x.T, delta)
        self.grad_b = np.sum(delta, axis=0)
        self.grad_x = np.dot(delta, self.w.T)
```

VAEに必要な各層をクラスで実装することができました。次は、これらを使ってVAEを構築します。

7.5 VAEの実装

VAEの実装について解説します。Encoderで潜在変数に手書き数字画像を圧縮した後に、Decoderで元の画像を再構築します。 そして、潜在変数が分布する潜在空間を可視化したうえで、潜在変数が生成画像に与える影響を確かめます。

コード全体を紹介する前に、重要な箇所を解説します。

7.5.1 順伝播と逆伝播

各層を初期化し、順伝播と逆伝播の関数を定義します。n_zは潜在変数の数です。VAEなので、入出力のニューロン数は同じになります。

順伝播と逆伝播の実装コード

```
# -- 各層の初期化 --
# Encoder
middle_layer_enc = MiddleLayer(n_in_out, n_mid)
mu_layer = ParamsLayer(n_mid, n_z)
log_var_layer = ParamsLayer(n_mid, n_z)
z_layer = LatentLayer()
# Decoder
middle_layer_dec = MiddleLayer(n_z, n_mid)
output_layer = OutputLayer(n_mid, n_in_out)

# -- 順伝播 --
def forward_propagation(x_mb):
    # Encoder
    middle_layer_enc.forward(x_mb)
    mu_layer.forward(middle_layer_enc.y)
    log_var_layer.forward(middle_layer_enc.y)
    z_layer.forward(mu_layer.y, log_var_layer.y)
    # Decoder
    middle_layer_dec.forward(z_layer.z)
    output_layer.forward(middle_layer_dec.y)

# -- 逆伝播 --
def backpropagation(t_mb):
    # Decoder
    output_layer.backward(t_mb)
    middle_layer_dec.backward(output_layer.grad_x)
    # Encoder
    z_layer.backward(middle_layer_dec.grad_x)
    log_var_layer.backward(z_layer.grad_log_var)
    mu_layer.backward(z_layer.grad_mu)
    middle_layer_enc.backward(mu_layer.grad_x + \
                              log_var_layer.grad_x)
```

ParamsLayerクラスからは、潜在変数の平均値を出力するmu_layerと、標準偏差の2乗の対数を出力するlog_var_layerのインスタンスを生成します。これらの層の出力は、LatentLayerクラスから生成するz_layerの入力となります。

z_layerは潜在変数を出力しますが、これはDecoderの入力となります。

7.5.2 全体のコード

以下は全体のコードです。訓練データの用意、各層のクラス、順伝播と逆伝播の関数、ミニバッチ法が順次実装されています。

誤差は各エポックの終了後に測定しますが、再構成誤差と正則化項を別々に測定して記録します。学習終了後には、これらの誤差および全体の誤差の推移を表示します。

全体コードと実行結果（再構成誤差、正則化項、全体の誤差の推移）

```
import numpy as np
# import cupy as np              # GPUの場合
import matplotlib.pyplot as plt
from sklearn import datasets

# -- 各設定値 --
img_size = 8                     # 画像の高さと幅
n_in_out = img_size * img_size   # 入出力層のニューロン数
n_mid = 16                       # 中間層のニューロン数
n_z = 2

eta = 0.001    # 学習係数
epochs = 201
batch_size = 32
interval = 20  # 経過の表示間隔

# -- 訓練データ --
digits_data = datasets.load_digits()
x_train = np.asarray(digits_data.data)
x_train /= 15  # 0-1の範囲に
t_train = digits_data.target

# -- 全結合層の継承元 --
class BaseLayer:
    def update(self, eta):
        self.w -= eta * self.grad_w
        self.b -= eta * self.grad_b

# -- 中間層 --
class MiddleLayer(BaseLayer):
    def __init__(self, n_upper, n):
        # Heの初期値
        self.w = np.random.randn(n_upper, n) * np.sqrt(2/n_upper)
```

```
        self.b = np.zeros(n)

    def forward(self, x):
        self.x = x
        self.u = np.dot(x, self.w) + self.b
        self.y = np.where(self.u <= 0, 0, self.u) # ReLU

    def backward(self, grad_y):
        delta = grad_y * np.where(self.u <= 0, 0, 1)

        self.grad_w = np.dot(self.x.T, delta)
        self.grad_b = np.sum(delta, axis=0)
        self.grad_x = np.dot(delta, self.w.T)

# -- 正規分布のパラメータを求める層 --
class ParamsLayer(BaseLayer):
    def __init__(self, n_upper, n):
        # Xavierの初期値
        self.w = np.random.randn(n_upper, n) / np.sqrt(n_upper)
        self.b = np.zeros(n)

    def forward(self, x):
        self.x = x
        u = np.dot(x, self.w) + self.b
        self.y = u  # 恒等関数

    def backward(self, grad_y):
        delta = grad_y

        self.grad_w = np.dot(self.x.T, delta)
        self.grad_b = np.sum(delta, axis=0)
        self.grad_x = np.dot(delta, self.w.T)

# -- 出力層 --
class OutputLayer(BaseLayer):
    def __init__(self, n_upper, n):
        # Xavierの初期値
        self.w = np.random.randn(n_upper, n) / np.sqrt(n_upper)
        self.b = np.zeros(n)

    def forward(self, x):
        self.x = x
        u = np.dot(x, self.w) + self.b
```

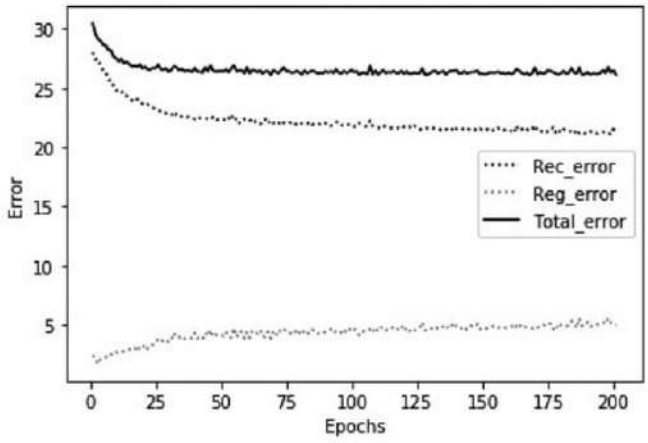
```
        self.y = 1/(1+np.exp(-u))  # シグモイド関数

    def backward(self, t):
        delta = self.y - t

        self.grad_w = np.dot(self.x.T, delta)
        self.grad_b = np.sum(delta, axis=0)
        self.grad_x = np.dot(delta, self.w.T)

# -- 潜在変数をサンプリングする層 --
class LatentLayer:
    def forward(self, mu, log_var):
        self.mu = mu             # 平均値
        self.log_var = log_var   # 分散の対数

        self.epsilon = np.random.randn(*log_var.shape)
        self.z = mu + self.epsilon*np.exp(log_var/2)

    def backward(self, grad_z):
        self.grad_mu = grad_z + self.mu
        self.grad_log_var = grad_z*self.epsilon/2*np.exp( \
                self.log_var/2) - 0.5*(1-np.exp(self.log_var))

# -- 各層の初期化 --
# Encoder
middle_layer_enc = MiddleLayer(n_in_out, n_mid)
mu_layer = ParamsLayer(n_mid, n_z)
log_var_layer = ParamsLayer(n_mid, n_z)
z_layer = LatentLayer()
# Decoder
middle_layer_dec = MiddleLayer(n_z, n_mid)
output_layer = OutputLayer(n_mid, n_in_out)

# -- 順伝播 --
def forward_propagation(x_mb):
    # Encoder
    middle_layer_enc.forward(x_mb)
    mu_layer.forward(middle_layer_enc.y)
    log_var_layer.forward(middle_layer_enc.y)
    z_layer.forward(mu_layer.y, log_var_layer.y)
    # Decoder
    middle_layer_dec.forward(z_layer.z)
    output_layer.forward(middle_layer_dec.y)
```

```
# -- 逆伝播 --
def backpropagation(t_mb):
    # Decoder
    output_layer.backward(t_mb)
    middle_layer_dec.backward(output_layer.grad_x)
    # Encoder
    z_layer.backward(middle_layer_dec.grad_x)
    log_var_layer.backward(z_layer.grad_log_var)
    mu_layer.backward(z_layer.grad_mu)
    middle_layer_enc.backward(mu_layer.grad_x + \
                              log_var_layer.grad_x)

# -- パラメータの更新 --
def update_params():
    middle_layer_enc.update(eta)
    mu_layer.update(eta)
    log_var_layer.update(eta)
    middle_layer_dec.update(eta)
    output_layer.update(eta)

# -- 誤差を計算 --
def get_rec_error(y, t):
    eps = 1e-7
    return -np.sum(t*np.log(y+eps) + \
            (1-t)*np.log(1-y+eps)) / len(y)

def get_reg_error(mu, log_var):
    return -np.sum(1 + log_var - mu**2 - \
            np.exp(log_var)) / len(mu)

rec_error_record = []
reg_error_record = []
total_error_record = []
n_batch = len(x_train) // batch_size  # 1エポックあたりのバッチ数
for i in range(epochs):

    # -- 学習 --
    index_random = np.arange(len(x_train))
    np.random.shuffle(index_random)  # インデックスをシャッフル
    for j in range(n_batch):

        # ミニバッチを取り出す
```

```
            mb_index = index_random[j*batch_size : (j+1)*batch_size]
            x_mb = x_train[mb_index, :]

            # 順伝播と逆伝播
            forward_propagation(x_mb)
            backpropagation(x_mb)

            # 重みとバイアスの更新
            update_params()

        # -- 誤差を求める --
        forward_propagation(x_train)

        rec_error = get_rec_error(output_layer.y, x_train)
        reg_error = get_reg_error(mu_layer.y, log_var_layer.y)
        total_error = rec_error + reg_error

        rec_error_record.append(rec_error)
        reg_error_record.append(reg_error)
        total_error_record.append(total_error)

        # -- 経過の表示 --
        if i%interval == 0:
            print("Epoch:", i, "Rec_error:", rec_error,
                  "Reg_error", reg_error,
                  "Total_error", total_error)

    plt.plot(range(1, len(rec_error_record)+1),
                   rec_error_record, label="Rec_error")
    plt.plot(range(1, len(reg_error_record)+1),
                   reg_error_record, label="Reg_error")
    plt.plot(range(1, len(total_error_record)+1),
                   total_error_record, label="Total_error")
    plt.legend()
    plt.xlabel("Epochs")
    plt.ylabel("Error")
    plt.show()
```

```
Epoch: 0 Rec_error: 27.62062025048551 Reg_error 3.0034780083555432
Total_error 30.62409825884105
Epoch: 20 Rec_error: 23.031885000169883 Reg_error 3.766025276816711
Total_error 26.797910276986592
```

```
Epoch: 40 Rec_error: 22.160684972437732 Reg_error 4.439101177687534
Total_error 26.599786150125265
Epoch: 60 Rec_error: 21.91640602328129 Reg_error 4.4683495192054385
Total_error 26.38475554248673
Epoch: 80 Rec_error: 21.85564285356529 Reg_error 4.5080898557951095
Total_error 26.3637327093604
Epoch: 100 Rec_error: 21.569457618615868 Reg_error 4.705692556892333
Total_error 26.2751501755082

    ⋮

Epoch: 180 Rec_error: 21.247450364170252 Reg_error 4.972593261884016
Total_error 26.220043626054267
Epoch: 200 Rec_error: 21.079783291460135 Reg_error 5.1622551060133866
Total_error 26.24203839747352
```



※わかりやすいように一部線種を変更しています。

誤差の推移のグラフからは、再構成誤差(Rec_error)と正則化項(Reg_error)が均衡し、全体の誤差(Total_error)が動かなくなったことが確認できます。潜在変数の範囲を広げることで入出力を一致させようとする働きを、正則化項が抑制していることになります。

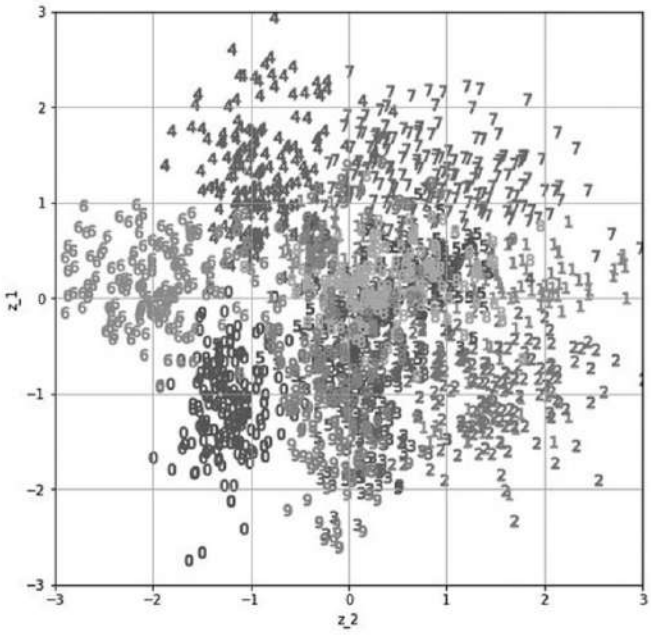
7.5.3 潜在空間の可視化

今回は可視化を容易にするために、潜在変数は2つしか使っていません。この2つの潜在変数を平面にプロットし、潜在空間を可視化します。

入力画像はそれが何の数であるかを示すラベルとペアになっていますが、ラベルの文字をマーカーとして使います。

潜在変数の分布

```
# 潜在変数を計算
forward_propagation(x_train)

# 潜在変数を平面にプロット
plt.figure(figsize=(8, 8))
for i in range(10):
    zt = z_layer.z[t_train==i]
    z_1 = zt[:, 0]  # y軸
    z_2 = zt[:, 1]  # x軸
    marker = "$"+str(i)+"$"  # 数値をマーカーに
    plt.scatter(z_2.tolist(), z_1.tolist(), marker=marker, s=75)

plt.xlabel("z_2")
plt.ylabel("z_1")
plt.xlim(-3, 3)
plt.ylim(-3, 3)
plt.grid()
plt.show()
```



マーカーの数字は入力画像が表す数字

この散布図では、各マーカーが数字ごとにグループを作っている様子が確認できます。これは、ラベルごとに異なる潜在空間の領域が占められていることを意味します。単一のラベルに占められている領域もありますが、ラベルが複数重なっている領域も

あります。

このように、VAEは入力を潜在空間に割り当てるように学習します。明瞭に領域が形作られるので、潜在変数が生成するデータにどのように影響を与えるのか、人間にとって理解が比較的容易です。

7.5.4 画像の生成

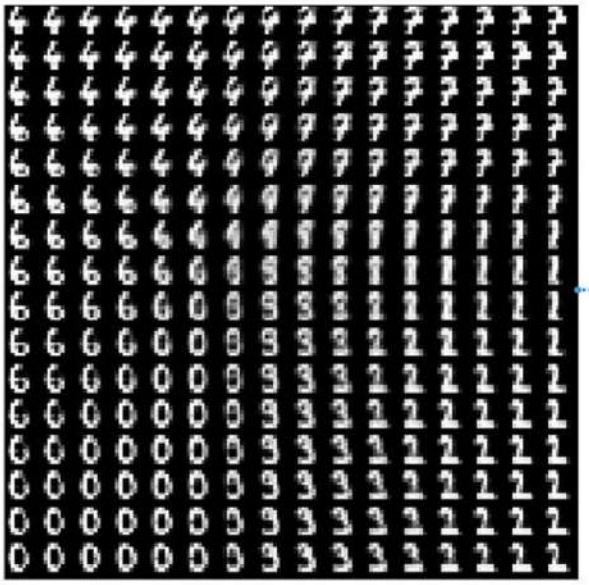
訓練済みVAEのDecoderを使って、画像を生成します。潜在変数を連続的に変化させて、生成される画像がどのように変化するのかを確認します。

画像を16×16枚生成して並べますが、x軸、y軸でDecoderの入力となる潜在変数を変化させます。

画像を生成するコード

```
# 画像の設定
n_img = 16  # 画像を16x16並べる
img_size_spaced = img_size + 2
# 全体の画像
matrix_image = np.zeros((img_size_spaced*n_img,
                         img_size_spaced*n_img))

# 潜在変数
z_1 = np.linspace(3, -3, n_img)  # 行
z_2 = np.linspace(-3, 3, n_img)  # 列

#  潜在変数を変化させて画像を生成
for i, z1 in enumerate(z_1):
    for j, z2 in enumerate(z_2):
        x = np.array([float(z1), float(z2)])
        middle_layer_dec.forward(x)  # Decoder
        output_layer.forward(middle_layer_dec.y)  # Decoder
        image = output_layer.y.reshape(img_size, img_size)
        top = i*img_size_spaced
        left = j*img_size_spaced
        matrix_image[top : top+img_size,
                     left : left+img_size] = image

plt.figure(figsize=(8, 8))
plt.imshow(matrix_image.tolist(), cmap="Greys_r")
```

```
# 軸目盛りのラベルと線を消す
plt.tick_params(labelbottom=False, labelleft=False,
                bottom=False, left=False)
plt.show()
```



VAEによって生成された16×16枚の画像。x軸、y軸の潜在変数はそれぞれ-3から3まで変化

Decoderにより、16×16枚の画像が生成されました。横軸、縦軸方向で潜在変数が変化していますが、それに伴う画像の変化が確認できます。ある数字とある数字の間には、それらの中間の画像を見ることができます。ラベルが重ならず、単一のラベルに対応する領域の数字は明瞭ですが、複数のラベルが重なる領域の数字は不明瞭です。

今回の例では、たった2つの潜在変数に8×8の画像を圧縮することができたことになります。このように、データの特徴を少数の潜在変数に圧縮することができて、なおかつ潜在変数が生成データに与える影響が明瞭であるのがVAEの興味深い点です。

なお、潜在変数の数はVAEの表現力に大きく影響しますので、興味のある方は潜在変数の数を増やす実験をしてみてはいかがでしょうか。

7.6 VAEの派生技術

本章の最後に、VAEの派生技術をいくつか簡単に紹介します。

7.6.1 Conditional VAE

Conditional VAE（→参考文献[16]）では、潜在変数だけではなくラベルもDecoderに入力することで、ラベルを指定した生成を行います。以下の図の例は2、3、4のようにラベルを指定した手書き数字画像の生成です。

Conditional VAEによるラベルを指定した手書き数字画像の生成

参考文献 [16]より引用

それぞれ縦横2つの潜在変数を変化させていますが、同じ文字でも書き方が変化していることがわかります。VAEは教師なし学習ですが、これに教師あり学習の要素を加えて半教師あり学習にすることで、復元するデータの指定を行うことが可能になります。

この技術により、たとえば同じ筆跡を持つ別の文字を生成することが可能になるかもしれません。この場合、AIが筆跡を覚えることになります。

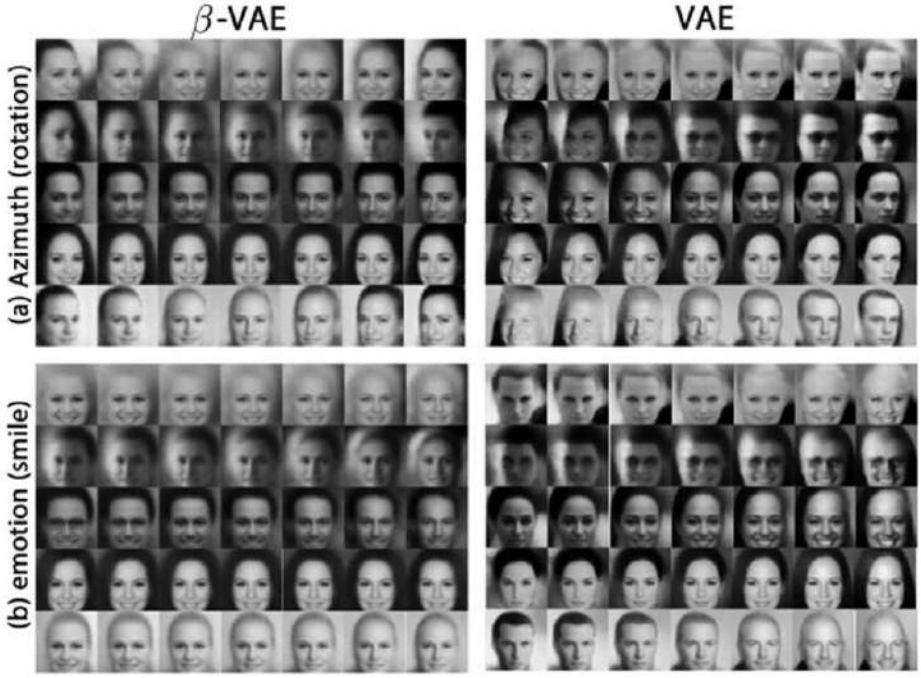
7.6.2 β-VAE

β-VAE（→参考文献[17]）は、画像特徴の「disentanglement」、すなわち「もつれ」を解くことが特徴で、画像の特徴を潜在空間上で分離することができます。

たとえば顔の画像の場合、1つ目の潜在変数が目の形、2つ目の潜在変数が顔の向きのように、潜在変数の各要素が独立した異なる特徴を担当することになります。これにより、たとえば1つ目の潜在変数を調整することで目の形を調整し、2つ目の潜在変数を調整することで顔の向きを調整するようなことが可能になります。

以下の図は、1つの潜在変数のみを変化させて生成した顔画像を並べたものです。

β-VAEによる顔画像の生成



参考文献[17]より引用

この図では左側がβ-VAEの結果で、右側がVAEの結果です。上の段が顔の向きの変化で、下の段が表情の変化です。潜在変数を変化させた際にVAEでは顔の向きや表情以外も変化していますが、β-VAEでは顔の向きと表情以外は変化していません。

以上のように、β-VAEは潜在変数を使って画像などの特徴を要素に分解できる興味深い技術です。

7.6.3 Vector Quantised-VAE

オリジナルのVAEには、潜在変数がデータの特徴をうまく捉えることができなくなる "posterior collapse" と呼ばれる現象により、生成画像がぼやけたものになってしまう問題がありました。

この問題に対処するために、Vector Quantised-VAE（→参考文献[18]）では、潜在変数を離散値、すなわち0、1、2、...などのとびとびの値に変換します。これは、画像をEnconderに入力し、出力である潜在変数のベクトルを「コードブック」にマッピングすることで実装されます。

このようにして画像の特徴を離散的な潜在空間に圧縮にすることにより、高品質な画像の生成が可能になります。以下の図はVector Quantised-VAEによって生成された画像ですが、128x128ピクセルの鮮明な画像が生成されています。

Vector Quantised-VAEによって生成された画像

参考文献 [18]より引用

7.6.4 Vector Quantised-VAE-2

VQ-VAE-2は、VQ-VAEを階層構造にすることでさらに高解像度の画像を生成できるようにした技術です(→参考文献[19])。VQ-VAE-2では、潜在表現を異なるスケールごとに、階層的に学習します。この潜在表現は元の画像よりも大幅に小さくなりますが、これをDecoderに入力することでより非常に鮮明でリアルな画像を再構築することができます。

以下の図はVQ-VAE-2によって生成された1024x1024の顔画像です。

VQ-VAE-2によって生成された画像

参考文献[19]より引用

非常に鮮明で、顔の特徴はまったく崩れていません。このようなサイズが比較的大きい画像でも、VQ-VAE-2を使えばうまく特徴を捉えて潜在空間に圧縮することが可能になります。

以上のようにVAEの技術は日々発展を続けており、AIの新たな可能性を示し続けています。

まとめ

本章では、生成モデルの一種であるVAEについて解説しました。

VAEの誤差を定義したうえで順伝播、逆伝播を数式で表し、VAEに必要な層をクラスとして実装しました。

VAEの実装前にオートエンコーダを実装しましたが、画像の特徴をニューロン数の少ない中間層に圧縮することができました。しかしながら、中間層が生成画像どのような影響を与えるのか、把握するのは困難でした。

その後VAEを構築して訓練し、潜在変数が分布する潜在空間を可視化したうえで、潜在変数が生成画像に与える影響を確かめました。たった2つの潜在変数に8×8の画像を圧縮することができて、なおかつ潜在変数が生成画像に与える影響は明瞭でした。

このように、VAEは表現力と柔軟性が高く、連続性を表現できるので注目を集めています。この技術の発展により、AIの可能性がさらに広がるのではないでしょうか。

Column AIと倫理

2017年1月、カリフォルニア州アシロマに全世界からAIの研究者とさまざまな分野の専門家が集まり、BENEFICIAL AI 2017カンファレンスが開催されました。そこでは、人類にとって有益なAIについて5日間にわたる議論がありましたが、その成果として2017年2月3日に発表されたのが、「アシロマAI原則」(Asilomar AI Principles)です。

アシロマAI原則
https://futureoflife.org/ai-principles/

この原則には23の項目があり、AIの研究、倫理、未来などに対してさまざまな方針が提案されています。各項目は上記のリンク先で確認することができますが、要点を以下のようにまとめます。

- AIシステムは説明、検証可能とすべきである
- AIと人類の倫理観、価値観を一致させるべきである
- AIによってもたらされる利益は、人類全体で共有すべきである
- AIは人類文明を尊重すべきである
- 高度なAIは世界に多大な影響を及ぼしうるため、慎重な管理が必要である

AIの行末に対しては、AIの専門家だけではなく多くの人々が希望と警戒心を同時に抱いています。 今後、AIシステムの開発者には開発力だけではなく高い倫理観が求められるようになることでしょう。

また、上記のアシロマAI原則には含まれていませんが、もし脳と人工知能の境界が曖昧になった場合、人権をどのように扱うかという問題が生じます。現在はホモ・サピエンスとそれ以外という線引きがされていますが、ヒトのような感情を持つ人工知能が生まれた場合、その線引きはどうなるのでしょうか。

さらに踏み込むと、AIが宇宙に進出することになった場合、人類のいない宇宙でも人間中心の倫理、価値観を貫く必要はあるのでしょうか。地球外の環境は地球表面の環境に最適化された人間にとって過酷すぎるので、地球外はAI主体の世界になるかもしれません。

ある意味、AIは我々人類が生み出し、育てつつある「子供」のようなものなのではないでしょうか。育て方が良ければ世界に調和と繁栄をもたらしますが、悪いとAIが搾取のために利用されたり、癌細胞のように制御不能になってしまうかもしれません。そのような意味で、AIの「親」として今生きている人類の責任は重大です。

願わくは、優しさと賢さを兼ね備えた子に育ってほしいものです。