AI自動作曲プログラミング

第5回

ディープラーニングのプログラミング基礎

Magentaの実践的解説

Tensorflowの基礎

ディープラーニングのプログラミング基礎

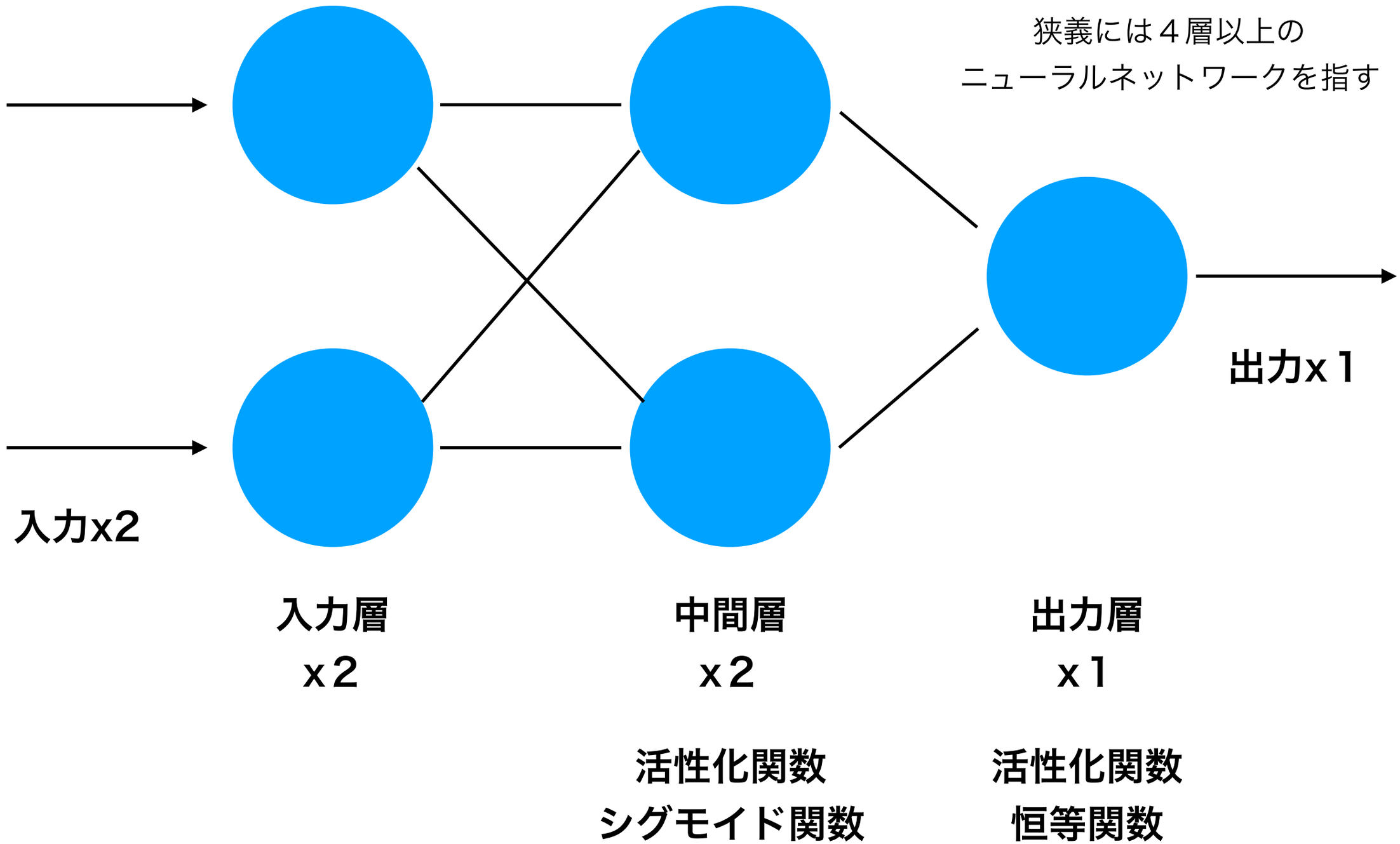
Magentaの実践的解説

Tensorflowの基礎

ディープラーニングのプログラミング基礎

ニューラルネットワークのプログラミング

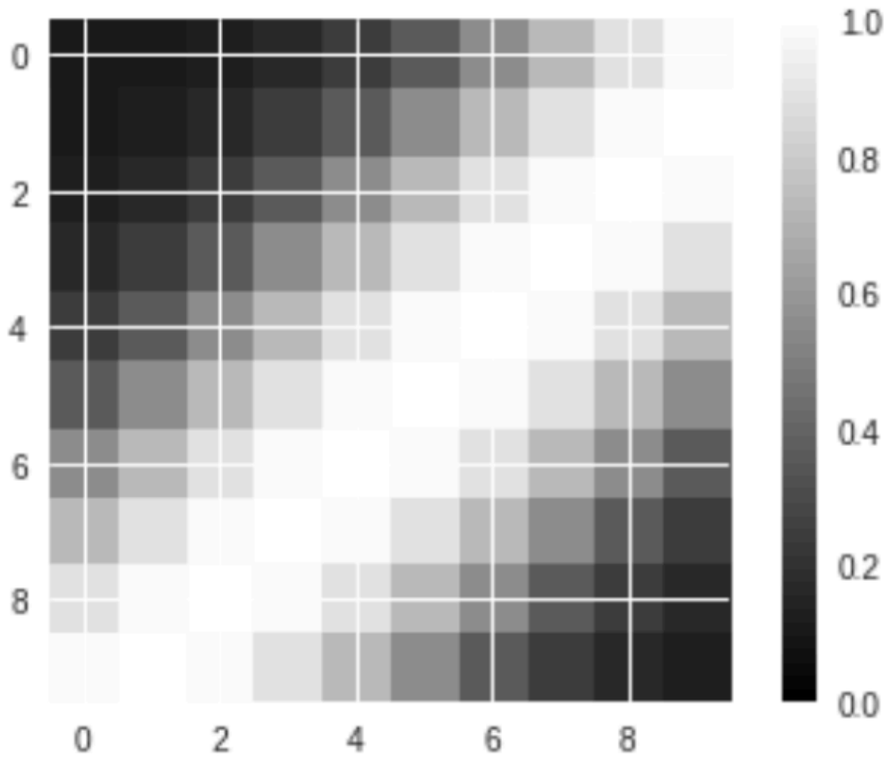
３層のニューラルネットワーク



この３層のニューラルネットワーク（回帰）
をコードで書くと

```
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt

# x、y座標
X = np.arange(-1.0, 1.0, 0.2)  # 要素数は10個
Y = np.arange(-1.0, 1.0, 0.2)

# 出力を格納する10x10のグリッド
Z = np.zeros((10,10))

# 重み
w_im = np.array([[4.0,4.0],
                 [4.0,4.0]])  # 中間層 2x2の行列
w_mo = np.array([[1.0],
                 [-1.0]])     # 出力層 2x1の行列

# バイアス
b_im = np.array([3.0,-3.0]) # 中間層
b_mo = np.array([0.1])      # 出力層

# 中間層
def middle_layer(x, w, b):
    u = np.dot(x, w) + b
    return 1/(1+np.exp(-u)) # シグモイド関数

# 出力層
def output_layer(x, w, b):
    u = np.dot(x, w) + b
    return u  # 恒等関数

# グリッドの各マスでニューラルネットワークの演算
for i in range(10):
    for j in range(10):

        # 順伝播
        inp = np.array([X[i], Y[j]])        # 入力層
        mid = middle_layer(inp, w_im, b_im) # 中間層
        out = output_layer(mid, w_mo, b_mo) # 出力層

        # グリッドにNNの出力を格納
        Z[j][i] = out[0]

# グリッドの表示
plt.imshow(Z, "gray", vmin = 0.0, vmax = 1.0)
plt.colorbar()
plt.show()
```

出力結果

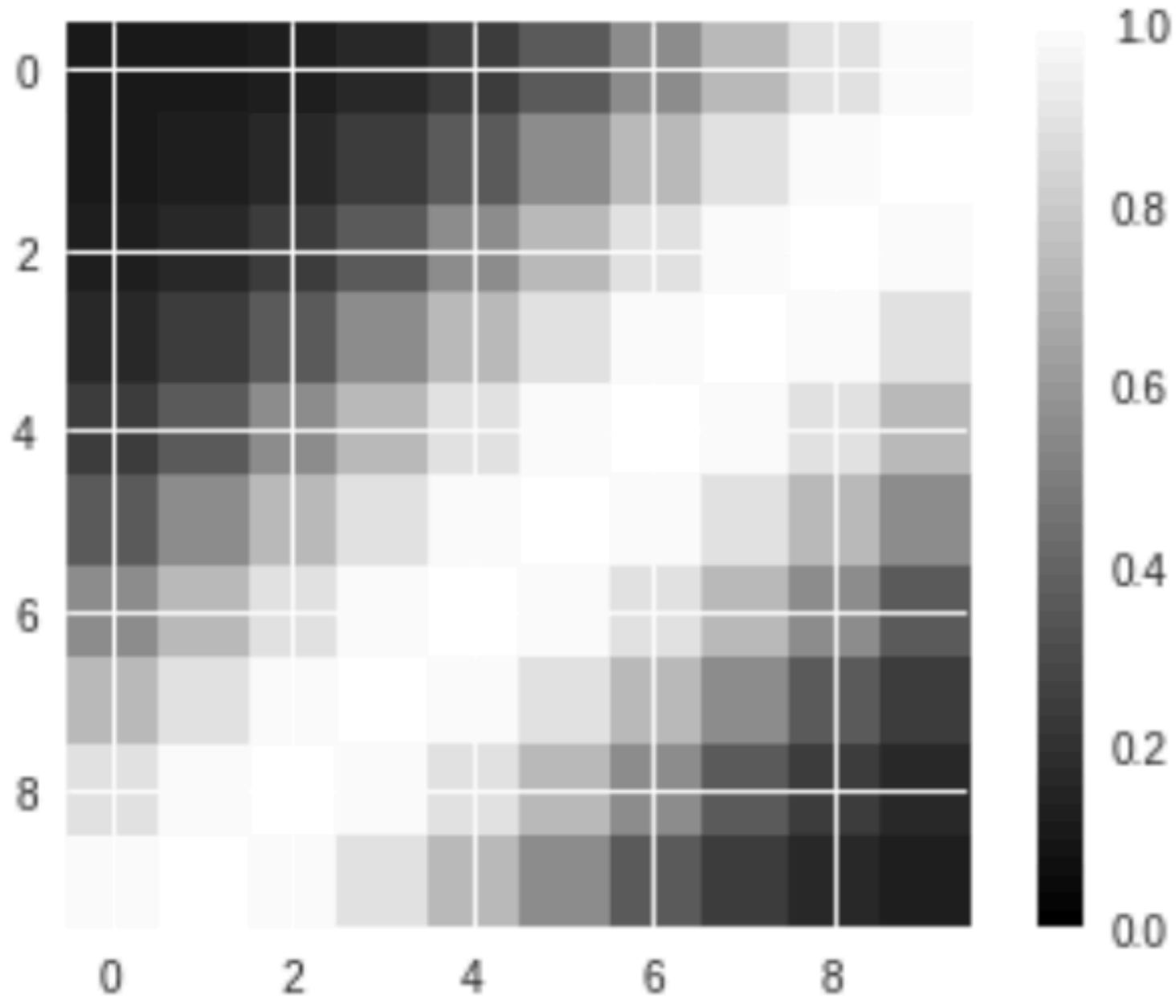


```
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt

# x、y座標
X = np.arange(-1.0, 1.0, 0.2)  # 要素数は10個
Y = np.arange(-1.0, 1.0, 0.2)

# 出力を格納する10x10のグリッド
Z = np.zeros((10,10))

# 重み
w_im = np.array([[4.0,4.0],
                 [4.0,4.0]])  # 中間層 2x2の行列
w_mo = np.array([[1.0],
                 [-1.0]])     # 出力層 2x1の行列
```

```
# バイアス
b_im = np.array([3.0,-3.0]) # 中間層
b_mo = np.array([0.1])      # 出力層

# 中間層
def middle_layer(x, w, b):
    u = np.dot(x, w) + b
    return 1/(1+np.exp(-u)) # シグモイド関数

# 出力層
def output_layer(x, w, b):
    u = np.dot(x, w) + b
    return u  # 恒等関数
```

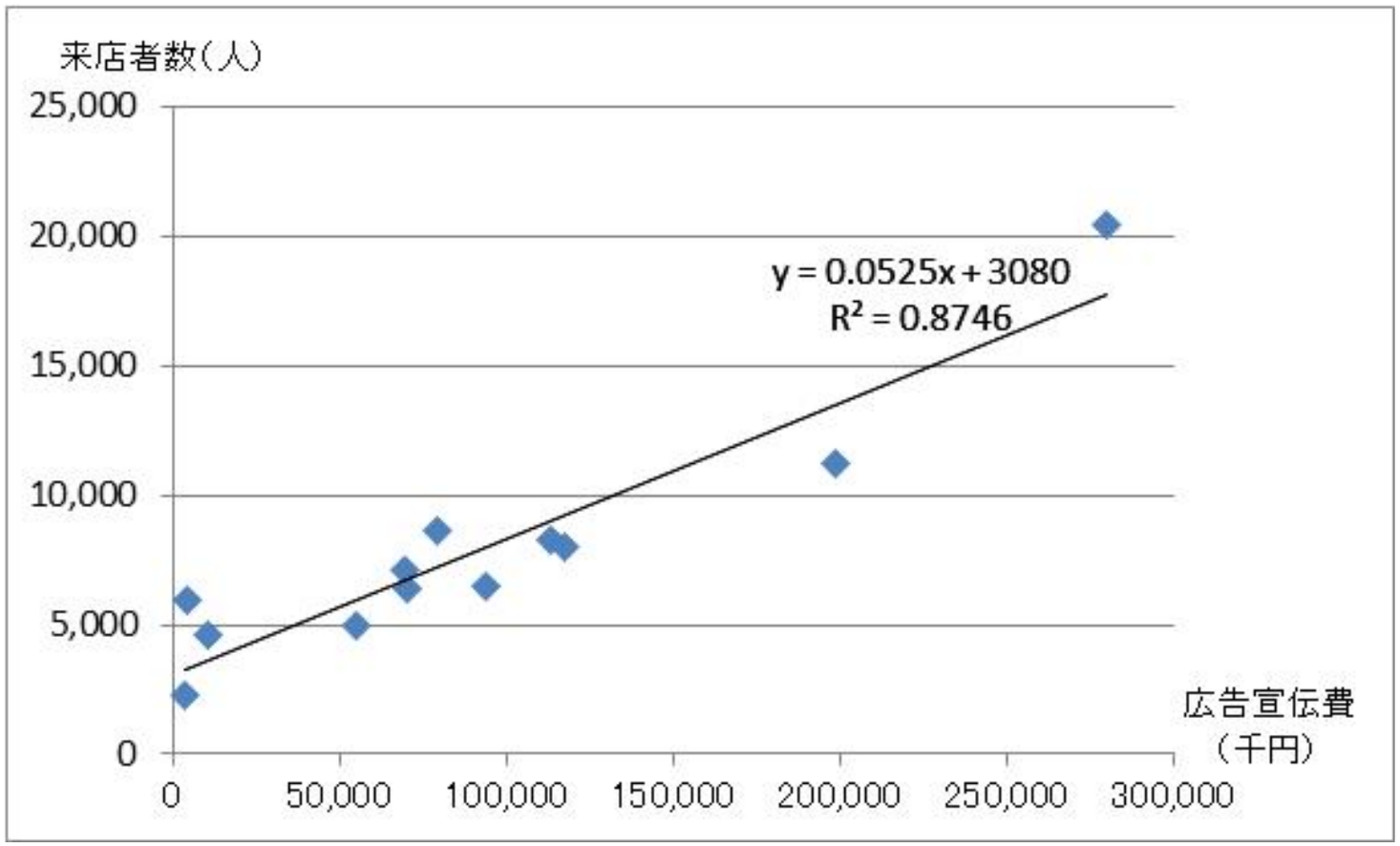
```
# グリッドの各マスでニューラルネットワークの演算
for i in range(10):
    for j in range(10):

        # 順伝播
        inp = np.array([X[i], Y[j]])          # 入力層
        mid = middle_layer(inp, w_im, b_im) # 中間層
        out = output_layer(mid, w_mo, b_mo) # 出力層

        # グリッドにNNの出力を格納
        Z[j][i] = out[0]

# グリッドの表示
plt.imshow(Z, "gray", vmin = 0.0, vmax = 1.0)
plt.colorbar()
plt.show()
```

より複雑なグラフの出力



ではニューラルネットワーク（回帰）
のコードを解説します

回帰分析とは



回帰分析とはより正確に言うと線形単回帰分析です。（線形とは、グラフにすると直線になる式であることを意味する）
例えばグラフにある様に、原因＝広告宣伝費（説明変数）を横軸に、結果＝来店者数（被説明変数）を縦軸にとった散布図を作り、２つの変数が比例関係にある場合に、最も適切な直線を引く作業です。

グラフの回帰分析（線形単回帰分析）の結果は、

$Y=0.525X+3080$
であり

$Y= b^1X+b^0$
という一次関数の数式になる

結果、広告費を10000円増やすごとに5人来客数が増えると予測される。

必要なモジュールのインポート

```
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt

# x、y座標
X = np.arange(-1.0, 1.0, 0.2)  # 要素数は10個
Y = np.arange(-1.0, 1.0, 0.2)

# 出力を格納する10x10のグリッド
Z = np.zeros((10,10))
```

必要なモジュールのインポートと
要素（データ）の数、出力描画用グラフのグリッド設定

必要なモジュールのインポート

matplotlibをJupyter Notebook
で使用する定型文

```
%matplotlib inline

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

計算に必要なモジュールnumpyと
グラフ表示に必要なモジュール
pyplotのインポート

入力層

入力に使用するx、y座標作成

```
# 入力に使用するx、y座標
X = np.arange(-1.0, 1.0, 0.2)  # 要素数は10個
Y = np.arange(-1.0, 1.0, 0.2)
```

numpyのarange関数でデータの配列（numpyの多次元配列ndarray)を作成します

arange関数はstart, end, stepを指定できます

startの数からendの数までstep間隔で要素を作成するという意味です

（注意：endはその数（今回は1.0）を含みません）

```
[-1.0, -0.8, -0.6, -0.4, -0.2, 0, 0.2, 0.4, 0.6, 0.8]
```

が作成されています

numpyのndarrayをprint表記させるには下記の様なコードが必要です

```
np.set_printoptions(formatter={'float': '{: 0.1f}'.format})
print(X)print(X)
```

グラフ表示のため、出力を格納する10x10の空のグリッド作成

```
# 出力を格納する10x10のグリッド
Z = np.zeros((10,10))
```

np.zerosは、すべての要素を0とする配列（詳細にはnp.ndarrayと呼ばれる多次元を扱う配列）を生成するものです
今回は横10、縦10の要素の値0の（空の）配列を作っています。

基本的な使い方

```
numpy.zeros(shape, dtype = float, order = 'C')
```

shape 配列 (1)なら１の要素を持つ１次元配列 (3,4)なら3x4の２次元配列を生成

dtype データ型（省略可能）初期値はfloat64

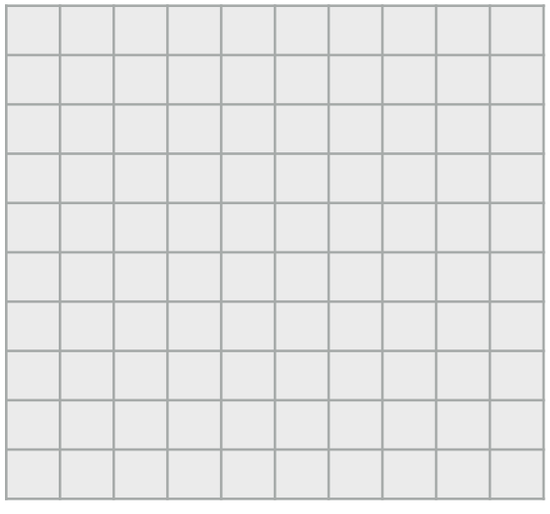
order 'C'または'F' （省略可能）初期値はC 多次元配列を行優先 (C-style) か列優先 (Fortran-style) でメモリに格納するかを指定する

```python
# x、y座標
X = np.arange(-1.0, 1.0, 0.2)  # 要素数は10個
Y = np.arange(-1.0, 1.0, 0.2)
```

X、Yともにnumpyのarange関数で要素を作成

X = [-1.0, -0.8, -0.6, -0.4, -0.2, 0, 0.2, 0.4, 0.6, 0.8]
Y = [-1.0, -0.8, -0.6, -0.4, -0.2, 0, 0.2, 0.4, 0.6, 0.8]

```python
# 出力を格納する10x10のグリッド
Z = np.zeros((10,10))
```



print(Z)
で空の配列

（10x10の行列）

を確認できる

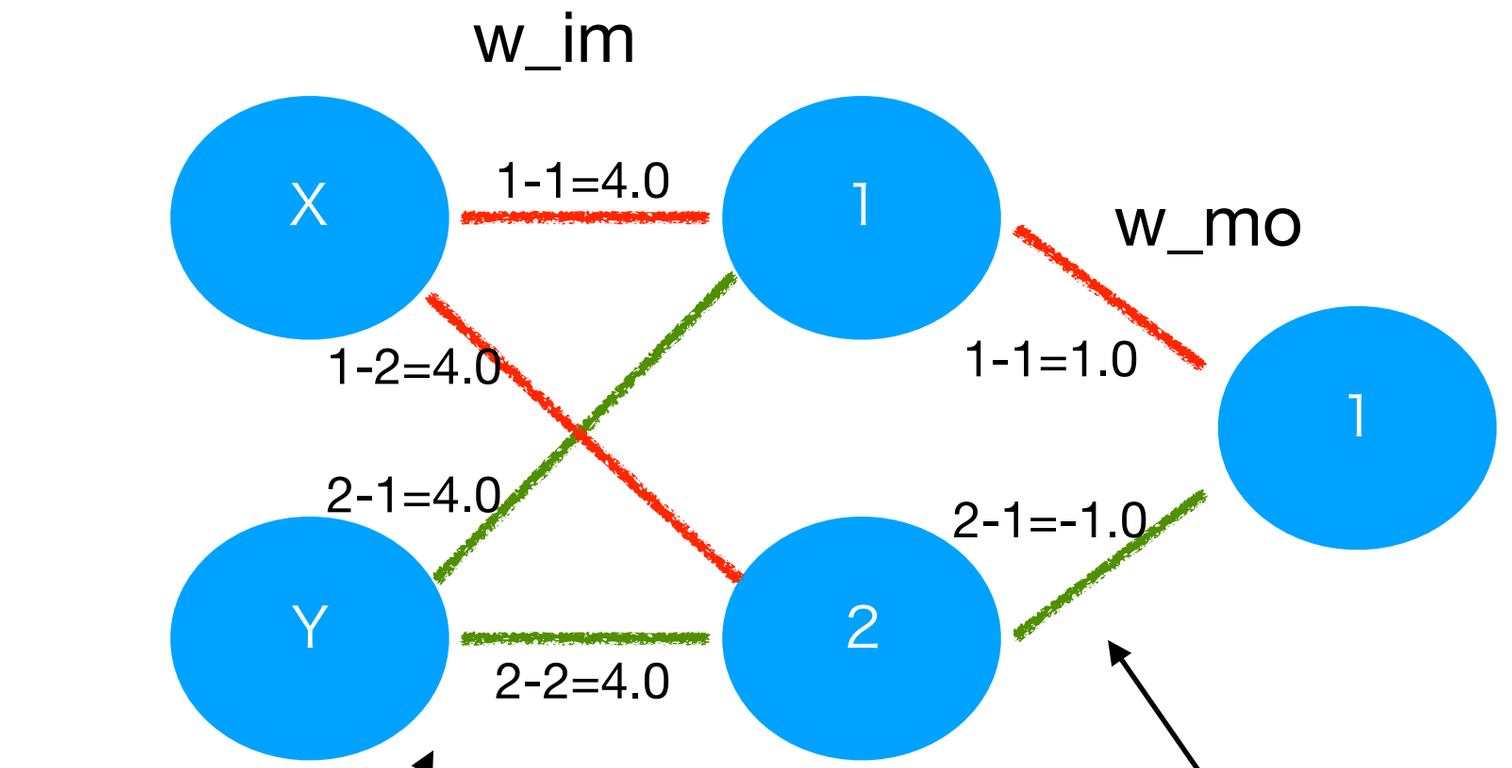
```
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
```

```
# 重み
w_im = np.array([[4.0,4.0],
                 [4.0,4.0]])  # 中間層 2x2の行列
w_mo = np.array([[1.0],
                 [-1.0]])     # 出力層 2x1の行列
```

重み

中間層が2x2

出力層が2x1です。

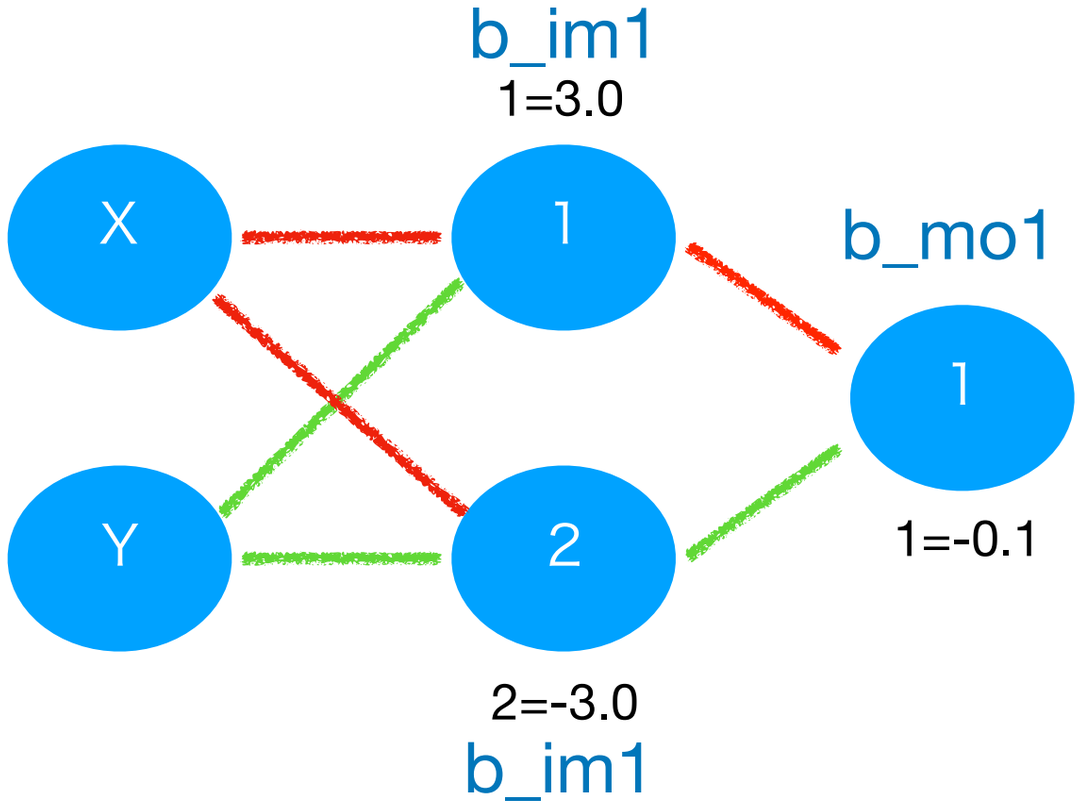


```
# バイアス
b_im = np.array([3.0,-3.0]) # 中間層
b_mo = np.array([0.1])      # 出力層
```

バイアス

中間層が2 出力層が1です。

バイアスの数はニューロンの数に等しくなります



中間層

```python
# 中間層
def middle_layer(x, w, b):
    u = np.dot(x, w) + b
    return 1/(1+np.exp(-u)) # シグモイド関数
```

中間層は入力データ(x)、重み(w)、バイアス(b)を受け取る

numpyのdot関数を用い、行列wとベクトルxの行列積（入力と重みの積の総和）を求め、バイアスbを足す

そこで出た結果uをシグモイド関数に入れる

スカラー、ベクトル、行列、テンソル

スカラー

通常の数値です。

1
2
100

pythonのプログラムで

a=1
b=1.0e5
などもスカラー

ベクトル

スカラーにある向きをもって並べたもの
横に並べる＝横ベクトル
縦に並べる＝縦ベクトル

$$\vec{a} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

$$\vec{b} = (-2.3, 0.25, -1.2, 1.8, 0.41)$$

$$\vec{p} = \begin{pmatrix} p_1 \\ p_2 \\ \vdots \\ p_m \end{pmatrix}$$

$$\vec{q} = (q_1, q_2, \cdots, q_n)$$

numpyのarray関数で
この様に作成できます

```
a = np.array([1, 2, 3])
b = np.array([-2.3, 0.25, -1.2, 1.8, 0.41])
```

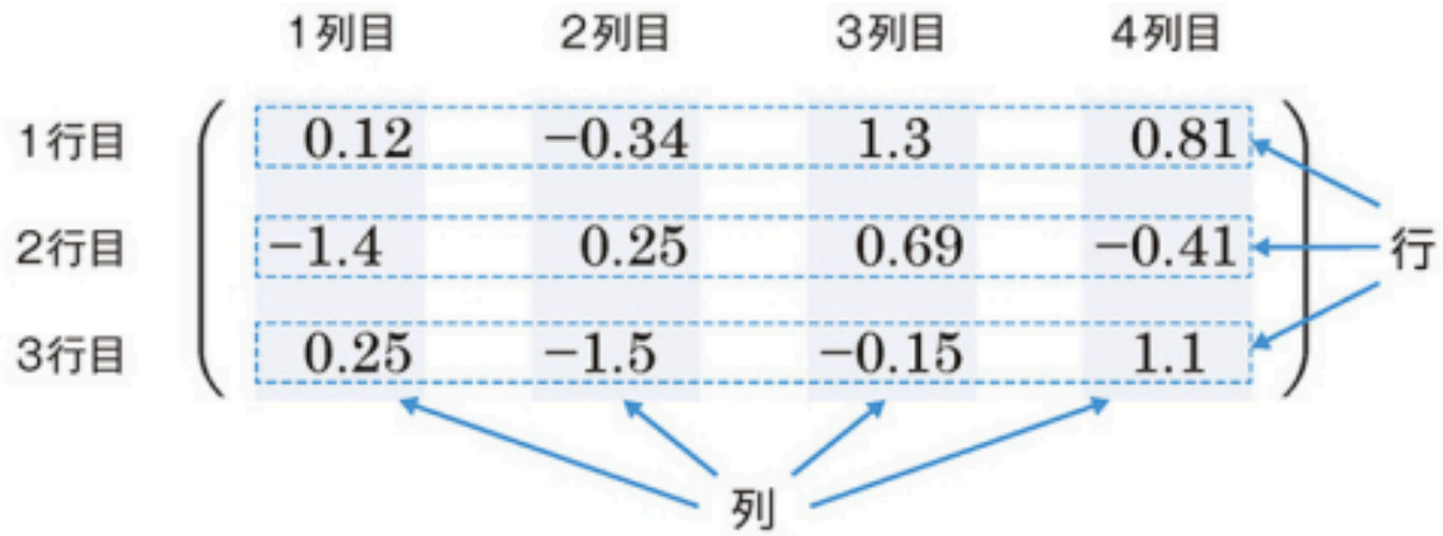
行列

スカラーを格子状に並べたもの

水平方向を**行**

垂直方向を**列**

と呼びます



上記は3 x 4の行列と呼ぶ

$$A = \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{pmatrix}$$

$$P = \begin{pmatrix} p_{11} & p_{12} & \cdots & p_{1n} \\ p_{21} & p_{22} & \cdots & p_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m1} & p_{m2} & \cdots & p_{mn} \end{pmatrix}$$

[[行・・・・]
 [行・・・・]
 [行・・・・]]

スカラー（行）を[ ]で囲み
縦の並び（列）の積み重ねを[ ]で囲む

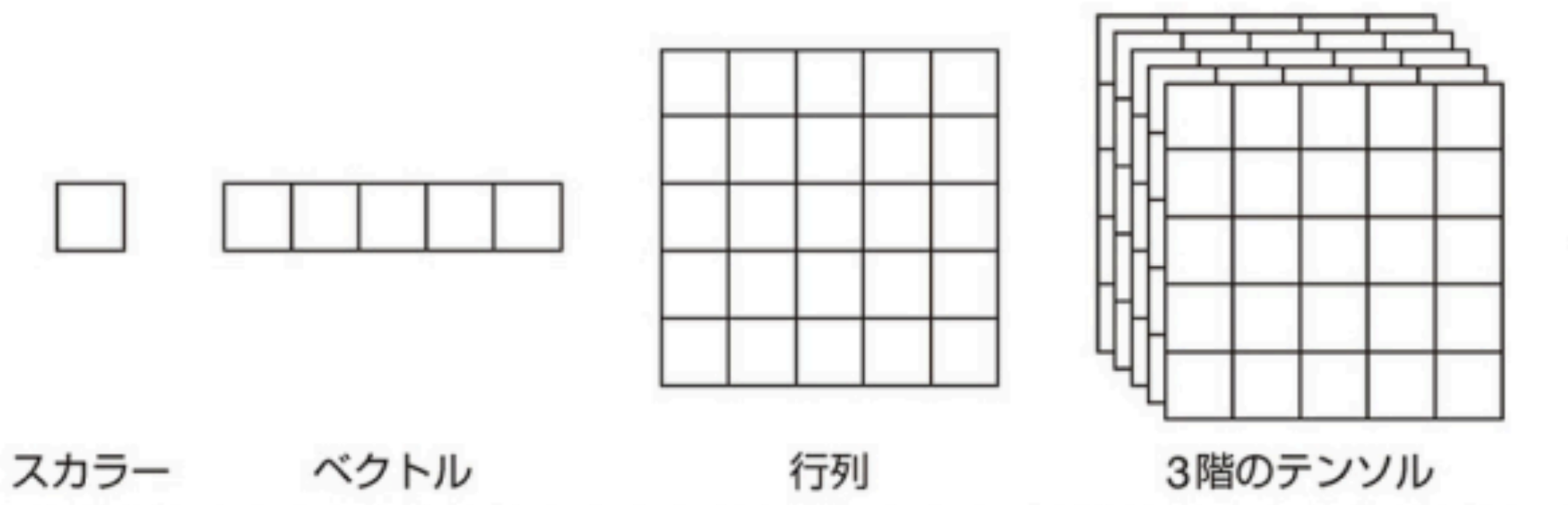
```
a = np.array([[1, 2, 3],
              [4, 5, 6]])
b = np.array([[0.21, 0.14],
              [-1.3, 0.81],
              [0.12, -2.1]])
```

2×3の行列

3×2の行列

テンソル

スカラーを３次元（以上）に並べたものでスカラー、ベクトル、行列を含む



ベクトルを１階のテンソル、行列を２階のテンソル、以降、３階～４階～のテンソルと呼ぶ

１階のテンソルから２階のテンソルの作成

```
b = np.array([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
              16,17,18,19,20,21,22,23,24])

b = b.reshape(4, 6)
print(b)
```

1階のテンソル（ベクトル）

2階のテンソル（行列）

```
[[ 1  2  3  4  5  6]
 [ 7  8  9 10 11 12]
 [13 14 15 16 17 18]
 [19 20 21 22 23 24]]
```

reshapeメソッドで２４個の要素を持つ１階のテンソルを、
（４、６）の要素を持つ２階のテンソル（4 x 6の行列）に変換

２階のテンソルから３階のテンソルの作成

```
b = b.reshape(2, 3, 4)
print(b)
```

```
[[[ 1  2  3  4]
  [ 5  6  7  8]
  [ 9 10 11 12]]

 [[13 14 15 16]
  [17 18 19 20]
  [21 22 23 24]]]
```

reshapeメソッドで（４、６）の要素を持つ２階のテンソルを（２、３、４）の要素を持つ３階のテンソル（２つの3 x 4の行列）に変換
要素数は２４で変わらない

３階のテンソルから４階のテンソルの作成

```
b = b.reshape(2, 2, 3, 2)
print(b)
```

```
[[[[ 1  2]
   [ 3  4]
   [ 5  6]]

  [[ 7  8]
   [ 9 10]
   [11 12]]]


 [[[13 14]
   [15 16]
   [17 18]]

  [[19 20]
   [21 22]
   [23 24]]]]
```

４階のテンソルは２つの（２つの3 x 4の行列）テンソル

要素数は２４で変わらない

numpyで行列積を求める

行列積

$$
\begin{aligned}
AB &= \begin{pmatrix} 0 & 1 & 2 \\ 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} 2 & 1 \\ 2 & 1 \\ 2 & 1 \end{pmatrix} \\
&= \begin{pmatrix} 0\times 2+1\times 2+2\times 2 & 0\times 1+1\times 1+2\times 1 \\ 1\times 2+2\times 2+3\times 2 & 1\times 1+2\times 1+3\times 1 \end{pmatrix} \\
&= \begin{pmatrix} 6 & 3 \\ 12 & 6 \end{pmatrix}
\end{aligned}
$$

２行３列の行列と３行２列の行列の計算

行列積

$$AB = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} 2 & 1 \\ 2 & 1 \\ 2 & 1 \end{pmatrix}$$

0x2 + 1x2 + 2x2

$$AB = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} 2 & 1 \\ 2 & 1 \\ 2 & 1 \end{pmatrix}$$

0x1 + 1x1 + 2x1

$$= \begin{pmatrix} 6 & 3 \\ 12 & 6 \end{pmatrix}$$

$$AB = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} 2 & 1 \\ 2 & 1 \\ 2 & 1 \end{pmatrix}$$

1x2 + 2x2 + 3x2

$$AB = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} 2 & 1 \\ 2 & 1 \\ 2 & 1 \end{pmatrix}$$

1x1 + 2x1 + 3x1

行列積をnumpy.dotメソッドで簡単に計算できます

```
import numpy as np

a = np.array([[0, 1, 2],
              [1, 2, 3]])
b = np.array([[2, 1],
              [2, 1],
              [2, 1]])
  print(np.dot(a, b))
```

```
[[ 6  3]
 [12  6]]
```

２行３列の行列と３行２列の行列の計算

aの列数とbの行数が一致しないとnumpyはエラーを返します

```python
# 中間層
def middle_layer(x, w, b):
    u = np.dot(x, w) + b
    return 1/(1+np.exp(-u)) # シグモイド関数
```

中間層は入力データ(x)、重み(w)、バイアス(b)を受け取る

numpyのdot関数を用い、ベクトルxと行列wの行列積を求め、バイアスbを足す

（x, w, bが受け取る値は後のループ文で出てきます）

（最初に出てきた変数Xとこのxは異なるものです）

（ちなみにPythonでは変数の大文字と小文字は区別されます）

numpyのdot関数を用い、入力データ(ベクトルx)と重み(行列w)
の行列積を求めます

```
X = np.arange(-1.0, 1.0, 0.2)
Y = np.arange(-1.0, 1.0, 0.2)
w_im = np.array([[4.0,4.0],
                 [4.0,4.0]])
```

[X, Y] [[4.0, 4.0]
          [4.0, 4.0]]

（xが受け取る値は後のループ文で出てきますが
XとYのベクトルになります）
X = [-1.0, -0.8, -0.6, -0.4, -0.2, 0, 0.2, 0.4, 0.6, 0.8]
Y = [-1.0, -0.8, -0.6, -0.4, -0.2, 0, 0.2, 0.4, 0.6, 0.8]

入力データ２列x重み２行の行列積

[-1.0,-1.0] [[4.0, 4.0]
               [4.0, 4.0]]

[-1.0,-1.0] [[4.0, 4.0]
               [4.0, 4.0]]

この順番で計算されます

aの列数とbの行数が一致しないとnumpyはエラーを返します

```
# 中間層
def middle_layer(x, w, b):
    u = np.dot(x, w) + b
    return 1/(1+np.exp(-u)) # シグモイド関数
```

中間層は入力データ(x)、重み(w)、バイアス(b)を受け取る

numpyのdot関数を用い、ベクトルxと行列wの行列積を求め、バイアスbを足す

$$u = [X, Y] \begin{bmatrix} [4.0, 4.0] \\ [4.0, 4.0] \end{bmatrix} + b$$

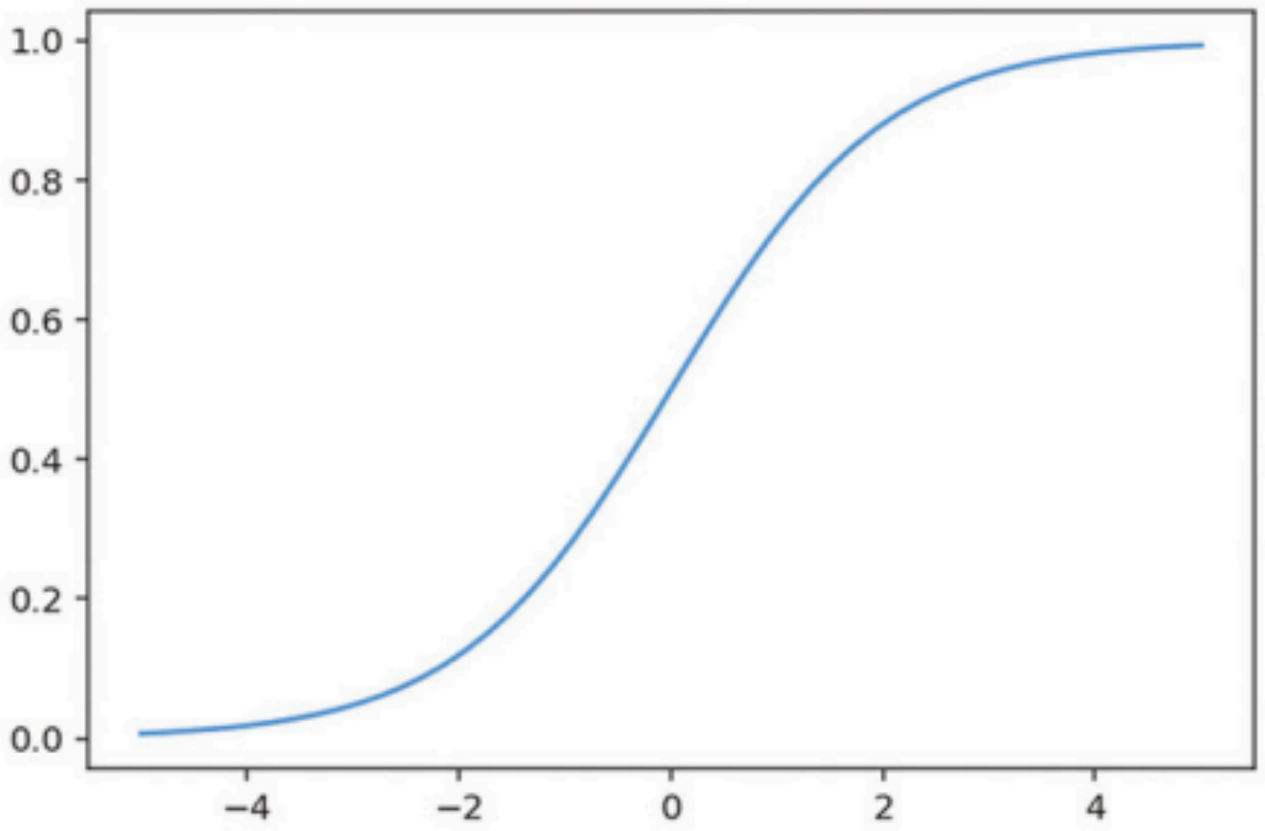
得られた結果uを活性化関数のシグモイド関数に入れて計算

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

活性化関数で使用する

シグモイド関数

シグモイド関数



$$\frac{1}{1+\exp(-u)}$$

expはネイピア数の累乗

$$\frac{1}{1+e^{**}(-u)}$$

expはネイピア数の累乗
なのでこの様な書き方もできます

関数への入力uが小さくなると出力yは0に近づき
関数への入力uが大きくなると出力yは1に近づく

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

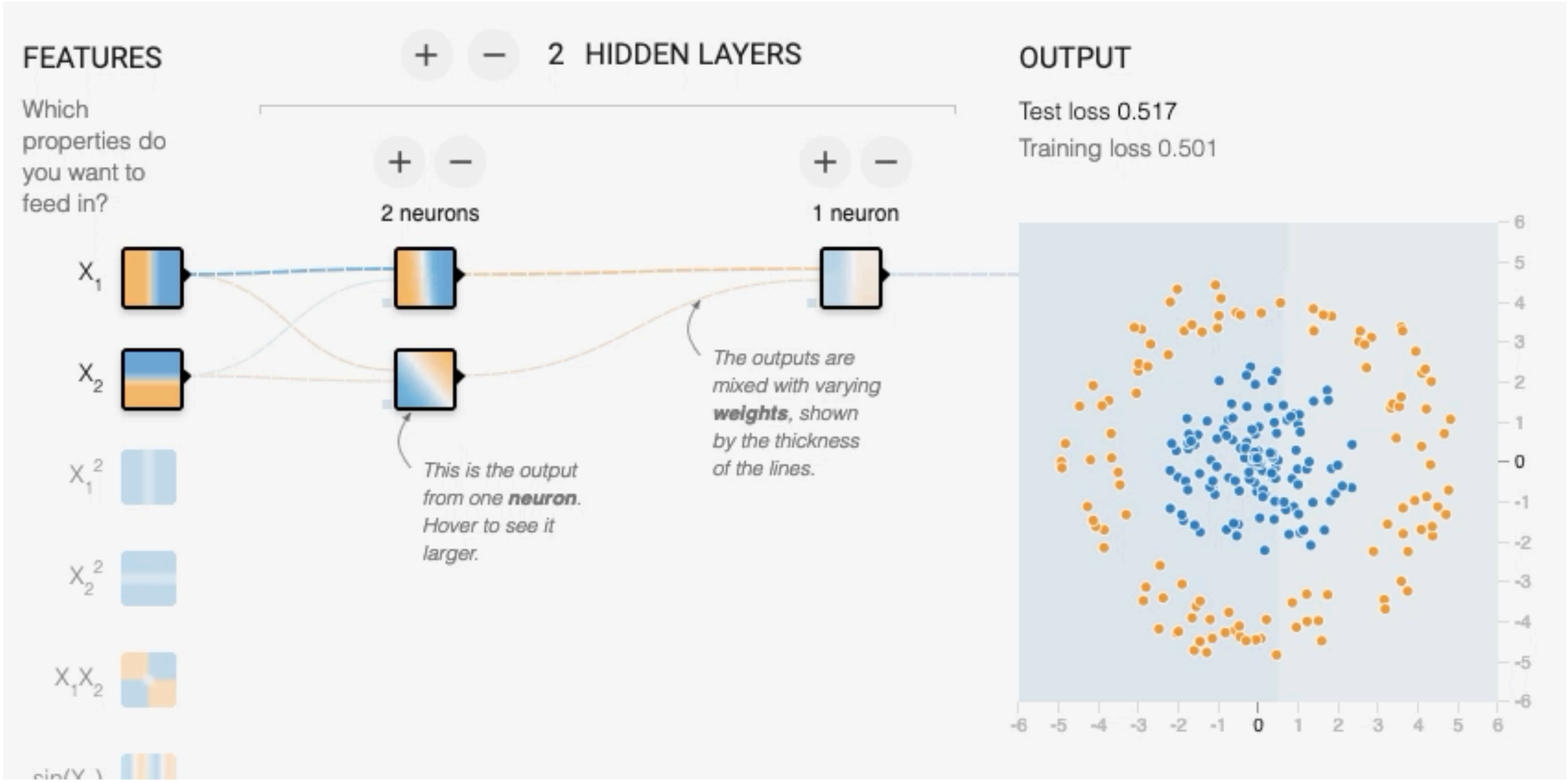
$$\text{中間層の出力} = \frac{1}{1+\exp(-(u=xw+b))}$$

活性化関数について

活性化関数というのは各層での（データx重み+バイアス）のあとに適用する非線形の関数。
これは線形分離できないデータを線形分離できるようにする働きをする。
線形な関数を使うと層を重ねても結局線形のままで、そのまま伸縮するだけなので目的を果たさない。
バックプロパゲーション(誤差逆伝播法)するために微分できる必要がある。

＊バックプロパゲーションは次回以降に解説

動画で確認してみましょう



＊今回の出力結果とは関連がないものであくまでも活性化関数の解説のためです。

出力層

```
def output_layer(x, w, b):
    u = np.dot(x, w) + b        #恒等関数
    return u
```

出力層の活性化関数は恒等関数（値をそのまま出力）のため
numpyのdotメソッドで計算された計算結果をそのまま出力
（x, w, bが受け取る値は後のループ文で出てきます）
（最初に出てきた変数Xとこのxは異なるものです）
（ちなみにPythonでは変数の大文字と小文字は区別されます）

出力層の重みとバイアス

```
w_mo = np.array([[1.0],
                 [-1.0]])     # 出力層 2x1の行列
```

```
b_mo = np.array([0.1])        # 出力層
```

```
def output_layer(x, w, b):
    u = np.dot(x, w) + b        #恒等関数
    return u
```

$$u = [X, Y] \begin{bmatrix} [1.0,] \\ [-1.0]\ ] \end{bmatrix} + 0.1$$

得られた結果uを恒等関数でそのまま出力

（x, w, bが受け取る値は後のループ文で出てきます）

ニューラルネットワークの演算

```
# グリッドの各マスでニューラルネットワークの演算
for i in range(10):
    for j in range(10):

        # 順伝播
        inp = np.array([X[i], Y[j]])          # 入力層
        mid = middle_layer(inp, w_im, b_im) # 中間層
        out = output_layer(mid, w_mo, b_mo) # 出力層

        # グリッドにNNの出力を格納
        Z[j][i] = out[0]
```

ループ文（多重ループのfor文）でデータ要素数分（１０回）x[i]とy[j]について計算をする
inp（入力層の結果）の結果はmid（中間層）に渡されてx重み+バイアスにて計算
出力結果mid（中間層の結果）はout（出力層）に渡されてx重み+バイアスにて計算
グリッドに出力の格納
out[0]は出力層の要素数が１つのため

```
for i in range(10):
    for j in range(10):
```

ループ文（多重ループのfor文）で i と j が（１０になるまで）１０回計算

```
inp = np.array([X[i], Y[j]])        # 入力層
```

XとYそれぞれ i, j 回分の配列を中間層へ渡す（１０回）

X = [-1.0, -0.8, -0.6, -0.4, -0.2, 0, 0.2, 0.4, 0.6, 0.8]
Y = [-1.0, -0.8, -0.6, -0.4, -0.2, 0, 0.2, 0.4, 0.6, 0.8]

```
        mid = middle_layer(inp, w_im, b_im) # 中間層
```

入力層からの結果をx=inp, w=w_im, b=b_imとして計算

```
# 中間層
def middle_layer(x, w, b):
    u = np.dot(x, w) + b
    return 1/(1+np.exp(-u)) # シグモイド関数
```

```
        out = output_layer(mid, w_mo, b_mo) # 出力層
```

中間層からの結果をx=mid, w=w_mo, b=b_moとして計算

```
# 出力層
def output_layer(x, w, b):
    u = np.dot(x, w) + b
    return u  # 恒等関数
```

```
# 中間層
def middle_layer(x, w, b):
    u = np.dot(x, w) + b
    return 1/(1+np.exp(-u)) # シグモイド関数
```

中間層は入力データ(x)、重み(w)、バイアス(b)を受け取る

numpyのdot関数を用い、ベクトルxと行列wの行列積を求め、バイアスbを足す

$$u = [X, Y] \begin{bmatrix} [4.0, 4.0] \\ [4.0, 4.0] \end{bmatrix} + b$$

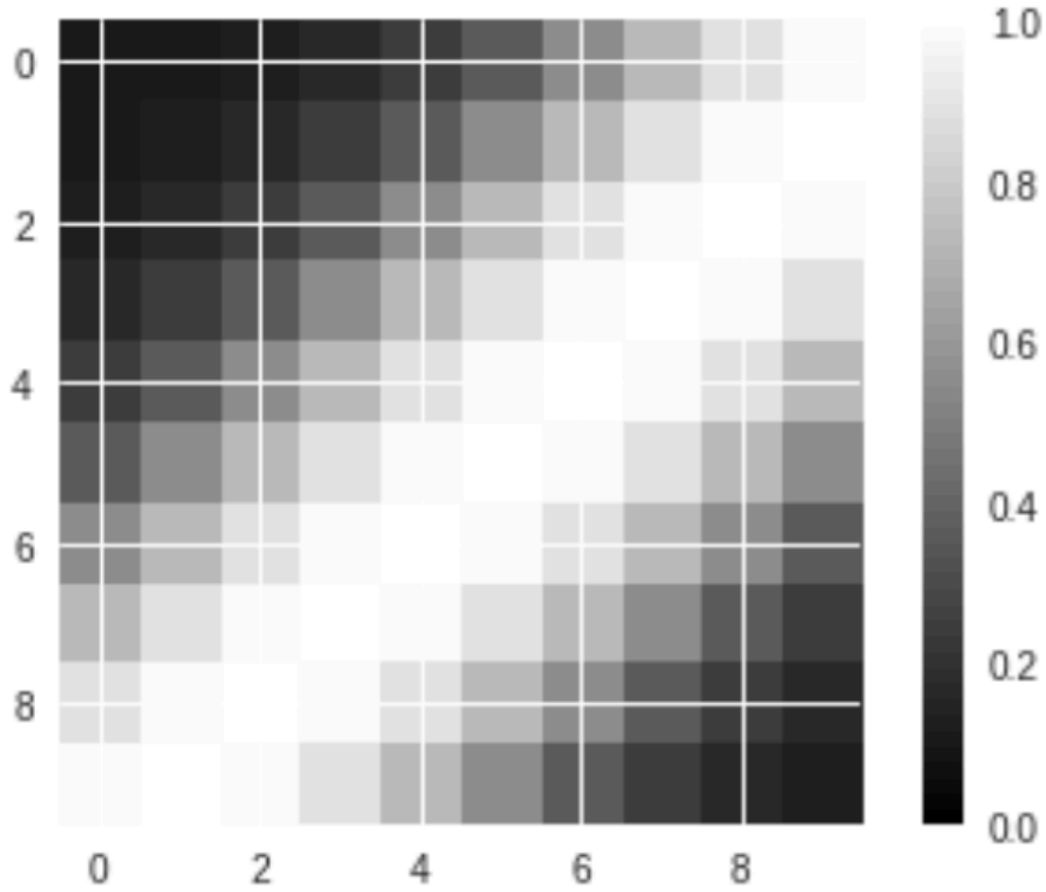
得られた結果uを活性化関数のシグモイド関数に入れて計算

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

グリッドの表示

```
# グリッドの表示
plt.imshow(Z, "gray", vmin = 0.0, vmax = 1.0)
plt.colorbar()
plt.show()
```

matplotlib
のpyplot関数でグリッドに表
示

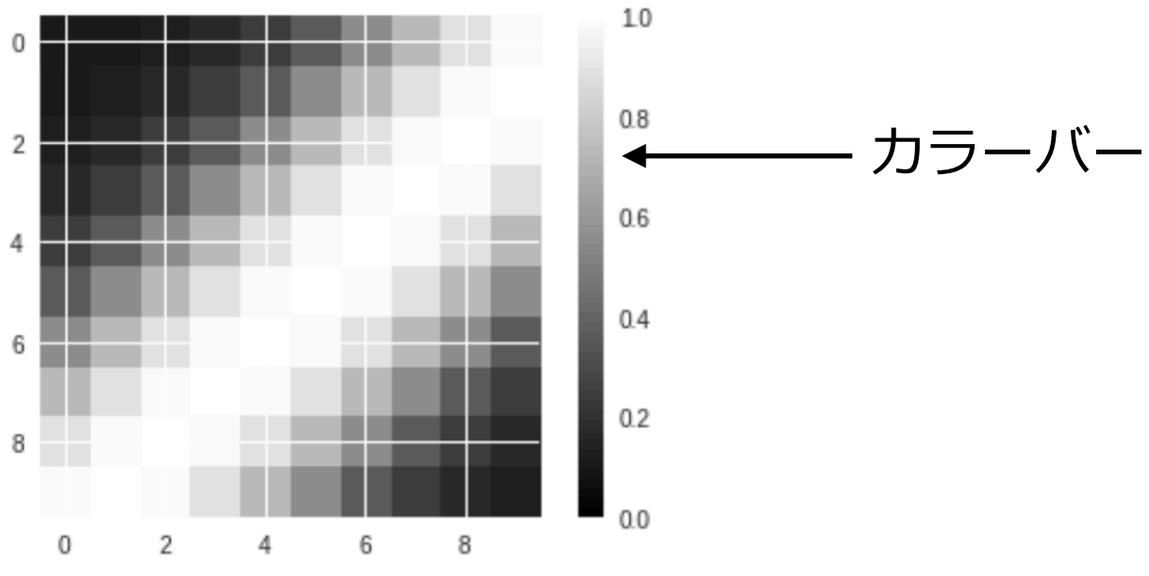


```
plt.imshow(Z, "gray", vmin = 0.0, vmax = 1.0)
```

pyplotのimgshow関数は、配列を画像として表示
Z 表示するグリッドデータ
gray 表示するカラー 今回はグレースケールを選択
vmin 表示するグラフの値の最小値
vmax 表示するグラフの値の最大値

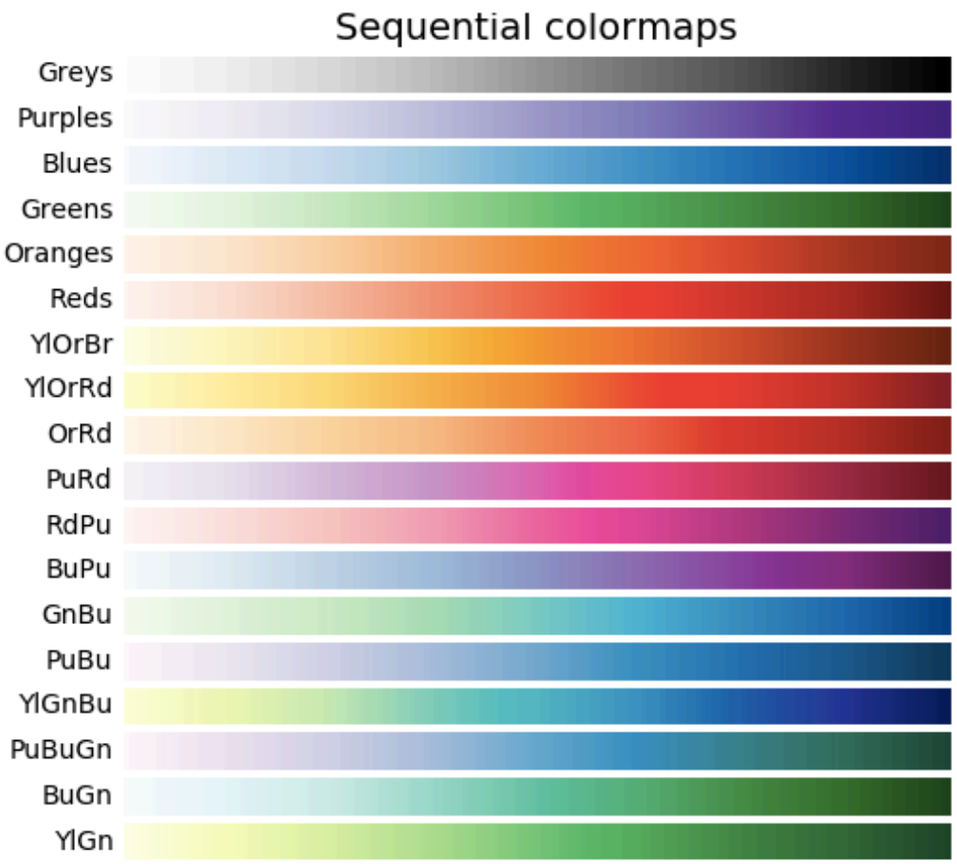
```
plt.colorbar()
```

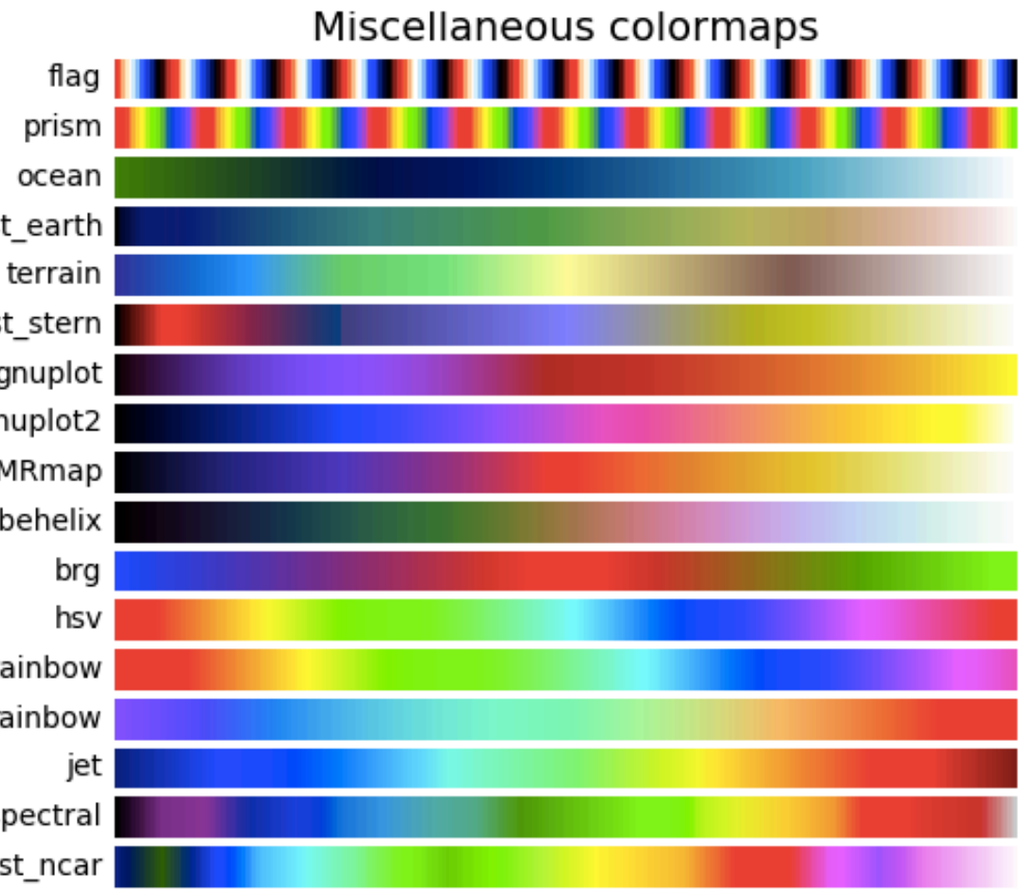
カラーバーの表示



```
plt.imshow(Z, "gray", vmin = 0.0, vmax = 1.0)
```

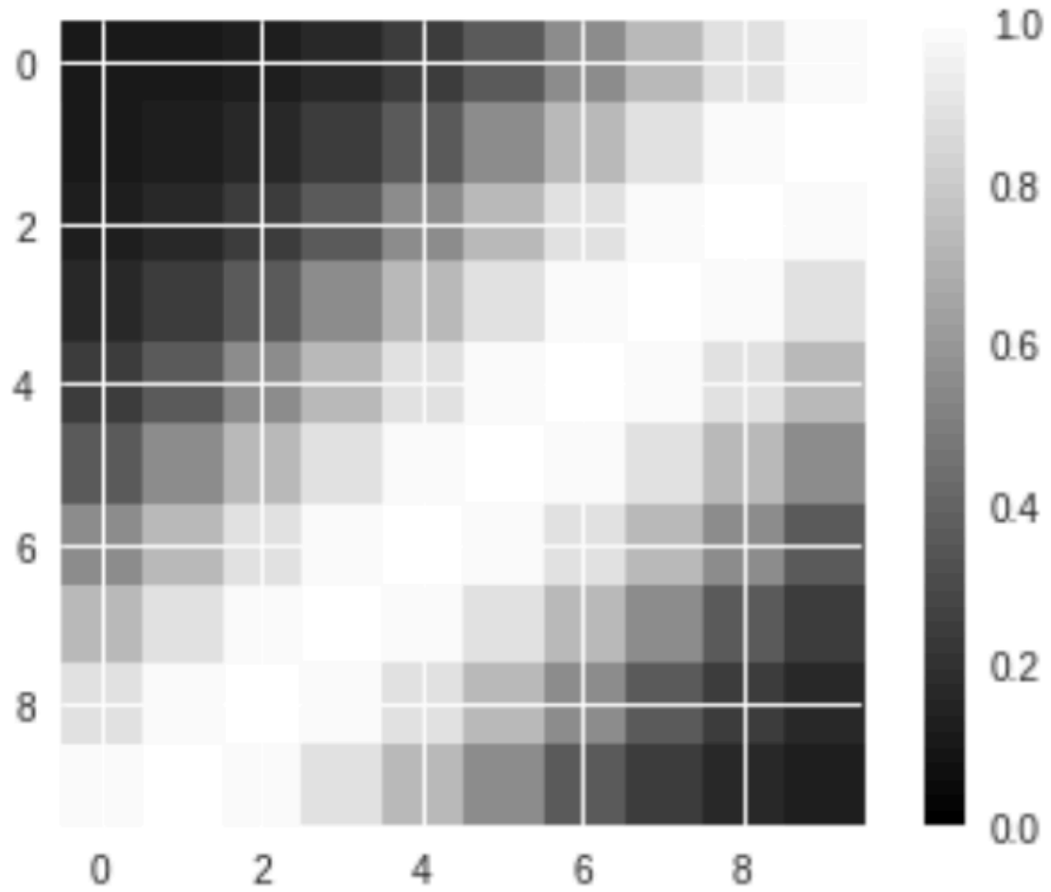
plt.imshow(Z, cmap="gray") cmapを省略しているコード
公式サイトにカラーマップがあります
https://matplotlib.org/examples/color/colormaps_reference.html





```
plt.show()
```

グリッドの表示



ディープラーニングのプログラミング基礎

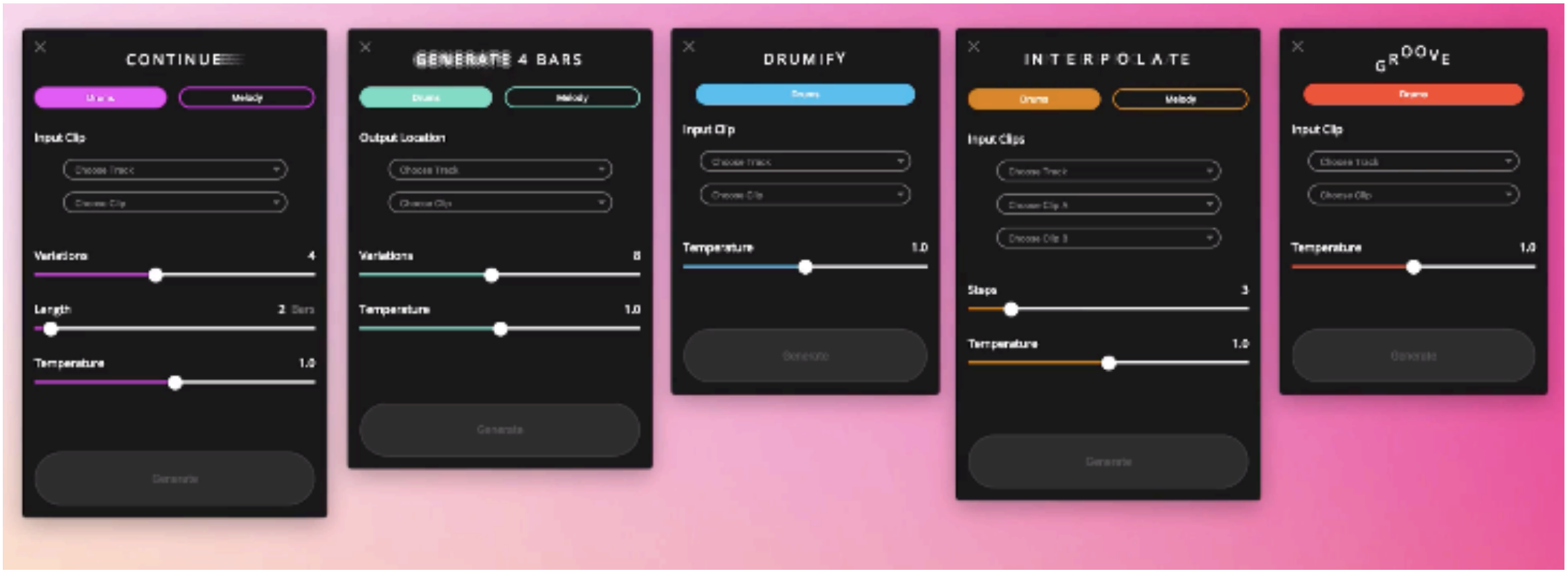
Magentaの実践的解説

Tensorflowの基礎

Magenta Studio
がv0.1で正式リリース

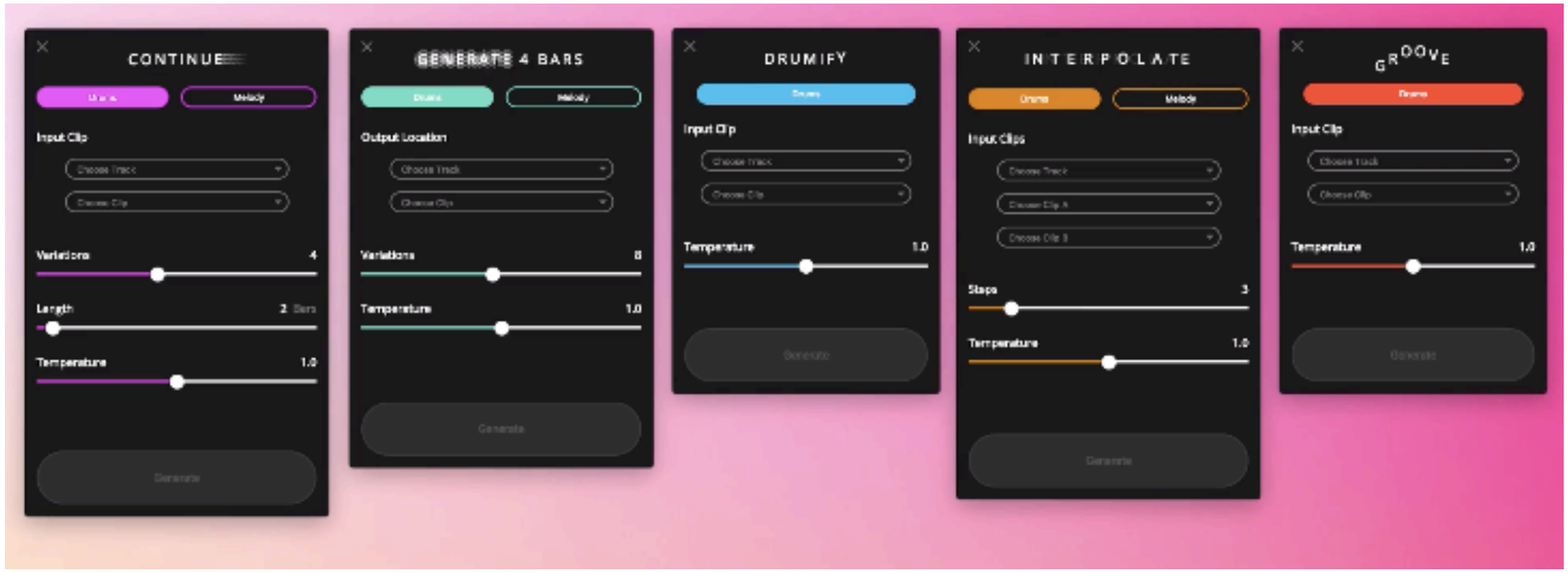
Magenta Studio

機械学習を使ったオープンソースの音楽プラグインのコレクション
スタンドアローン版とAbleton Live版がある
スタンドアローンのWINDOWS版は近日登場予定



スタンドアローン版とAbleton Live版は基本的に同じ機能
違いはMIDIの読み込みと書き出し方法

Magenta Studio



CONTINUE, INTERPOLATE, GROOVAE, GENERATE 4 BARS, Drunifyの5種類のプラグイン

Magenta Studioとは

機械学習を使ったオープンソースの音楽プラグインのコレクションMagenta.js（MagentaのJavascript版）をコードを書く必要やブラウザ、node.jsの利用を必要とせず直接使用できる。コマンドラインやウェブブラウザを介さずとも、多くの音楽家に実際のDAW音楽制作環境でMagenta使用を広めるため今後取り組みが強化される模様

ソースコードが公開されているために、独自の学習データを使用する事なども可能

アプリの制作はElectronを使用しているので自身でのMagentaデスクトップアプリの開発も可能

Ableton Liveでの使用のためにMAX for Liveを使用し、Node for MAXでMagentaとをつなぐ

Ableton LIVEとは

Ableton社製のDAW（音楽制作）ソフト
クリップスロットにMIDIファイルやオーディオファイルを読み込ませ、組み合わせることによって楽曲を制作する。
標準搭載される多数のシンセやプラグインのほか、外部ソフトとの同期や、MAXやMAX for Liveを使用したMagentaとの連携など最新の音楽制作環境を提供する

Ableton LIVEとは

Ableton社製のDAW（音楽制作）ソフト（最新版は１０）
クリップスロットにMIDIファイルやオーディオファイルを読み込ませ、組み合わせることによって楽曲を制作する。
標準搭載される多数のシンセやプラグインのほか、外部ソフトとの同期や、MAXやMAX for Liveを使用したMagentaとの連携など最新の音楽制作環境を提供する

MAXとは

Cyclong74社製のオーディオビジュアルプログラミングソフト（最新版は８）
パッチングにより、オーディオ、エフェクト、アート、3D、プロジェクション・マッピングをビジュアルプログラミング可能。
node.js使用でMagentaとの連携、開発も行える

Electronとは

JavaScript, HTML, CSS といったWeb技術を利用してネイティブアプリケーションを作成するためのフレームワーク
Webサイトを作成する感覚でデスクトップアプリを開発できる

https://electronjs.org/

Magenta Studio v0.1
の各解説

Magenta Studio

5種に共通の使用法

・ファイル読み込み

MIDIファイルを読み込みGENERATEボタンで生成

・ファイルの出力先

標準ではインプットフォルダーと同じフォルダー

任意で指定し変更できる

・TEMPERATURE

ニューラルネットワークの生成バリエーションの度合いをコントロール

高いほど変化させた複雑なバリエーション（時に混沌とするほど）低いほど元の学習データに近い生成を行う

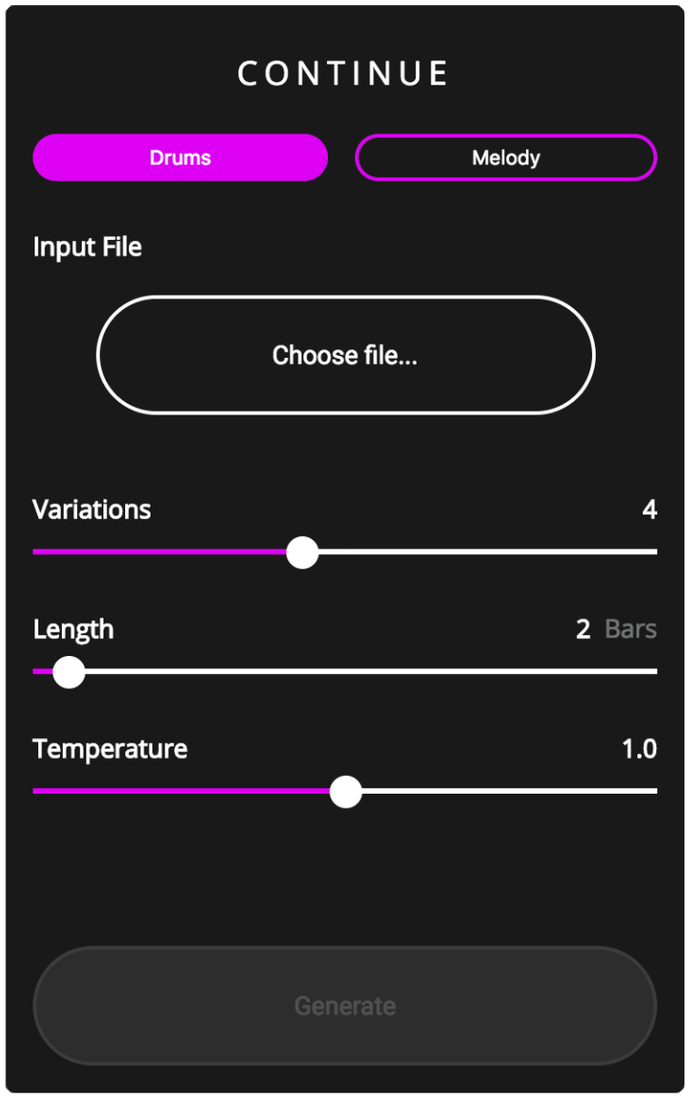
・メロディーは単音のみ対応

・ドラムはGM MIDI配列の９音のみ

対応ドラム音表

Instrument	Pitch
Bass drum/Snare drum	36/38
Closed/Open hi-hat	42/46
Low/Mid/High tom	45/48/50
Crash/Ride cymbal	49/51

Magenta Studio



CONTINUE

RNN(recurrent neural network)を使用し、入力したMIDIファイル（ドラムもしくはメロディー）に追従する新たなバリエーションを生成する

CONTINUE

Drums | Melody

Input Clip

Choose Track

Choose Clip

Variations 4

Length 2 Bars

Temperature 1.0

Generate

Continue [Continue]

Link | Ext | Tap | 120.00 | 4 / 4 | 1 Bar | 107 . 3 . 3 | Key | MIDI | 1 %

1 Grand Piano P

Drop Files and Devices Here

Master

Clip | Notes | Fold

D3-D4 | :2 | *2 | Rev | Inv | Legato | Dupl.Region

Start 1 1 1 | End 2 1 1 | Loop | Position 1 1 1 | Length 1 0 0

Signature 4 / 4 | Groove None | Commit

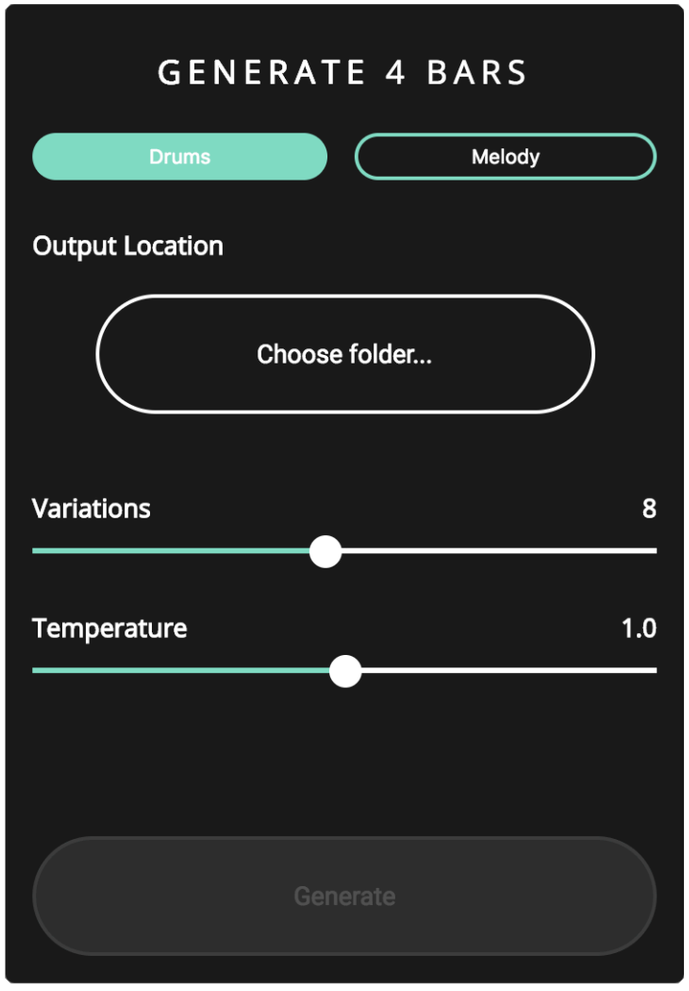
Pgm Change | Bank --- | Sub --- | Pgm ---

C4 D4 C4 C4 B3 A3 A3 G3 F3 F3 E3 E3 D3

1/16

1-Grand Piano Piano Harp Time

Magenta Studio



GENERATE
MIDIファイルの必要なく、新たな４小節のパターンを生成
ドラムもしくはメロディー４小説
数百万曲の学習データを元にしているとされる

GENERATE 4 BARS

Drums | Melody

Output Location

1-Grand Piano

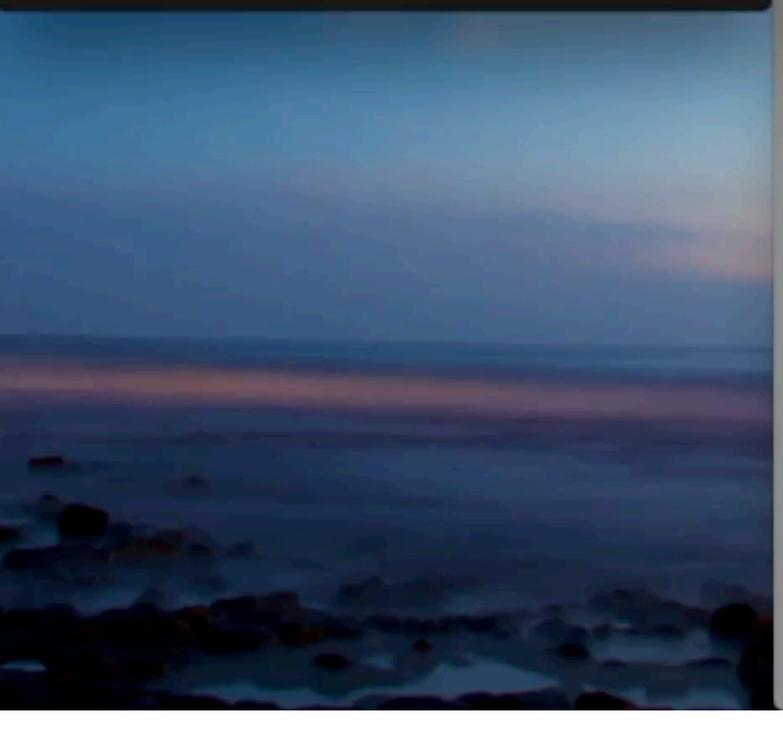
1/4 GENERATE

Variations 4

Temperature 1.0

Output **4** clips to **Clip Slots 1-4**

Generate



Generate [Generate]

Link | Ext | Tap | 120.00 | 4 / 4 | 1 Bar | 10 . 1 . 2 | Key | MIDI

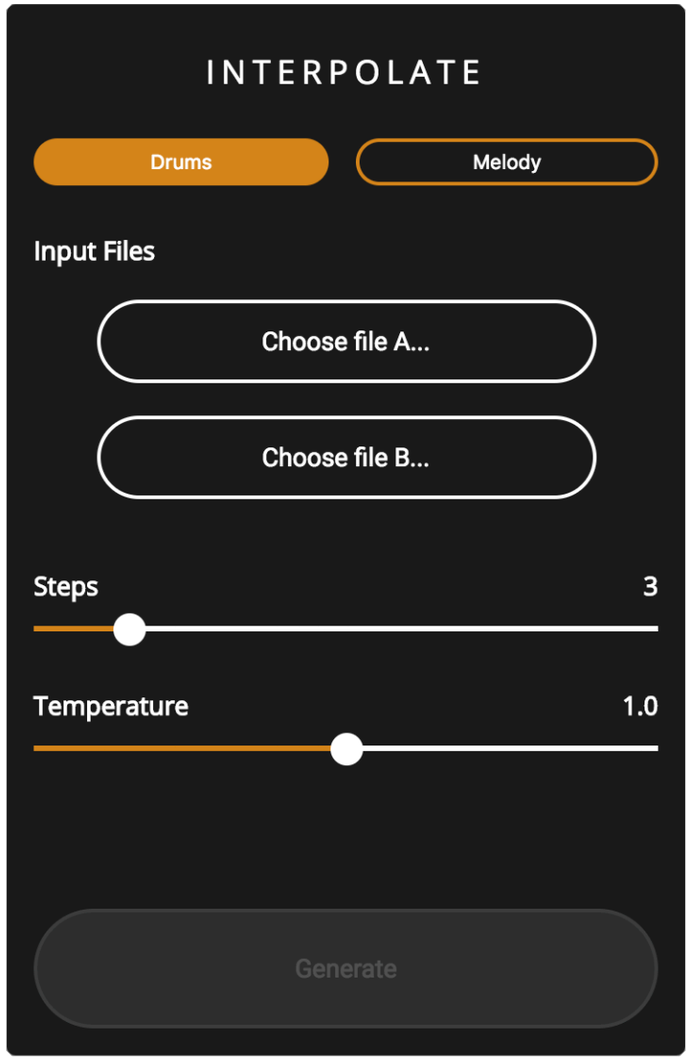
1 Grand Piano | 2 Datai Kit | Master

Drop Files and Devices Here

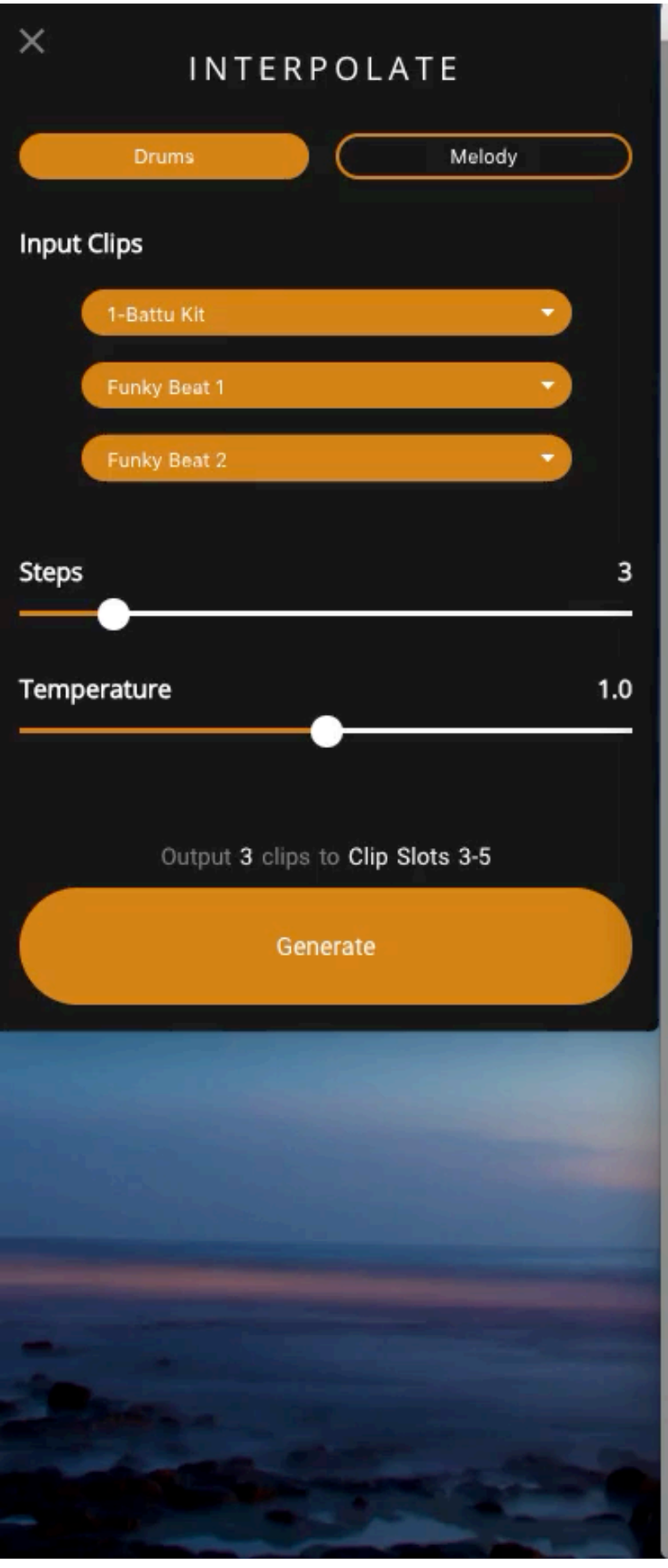
No clip selected.

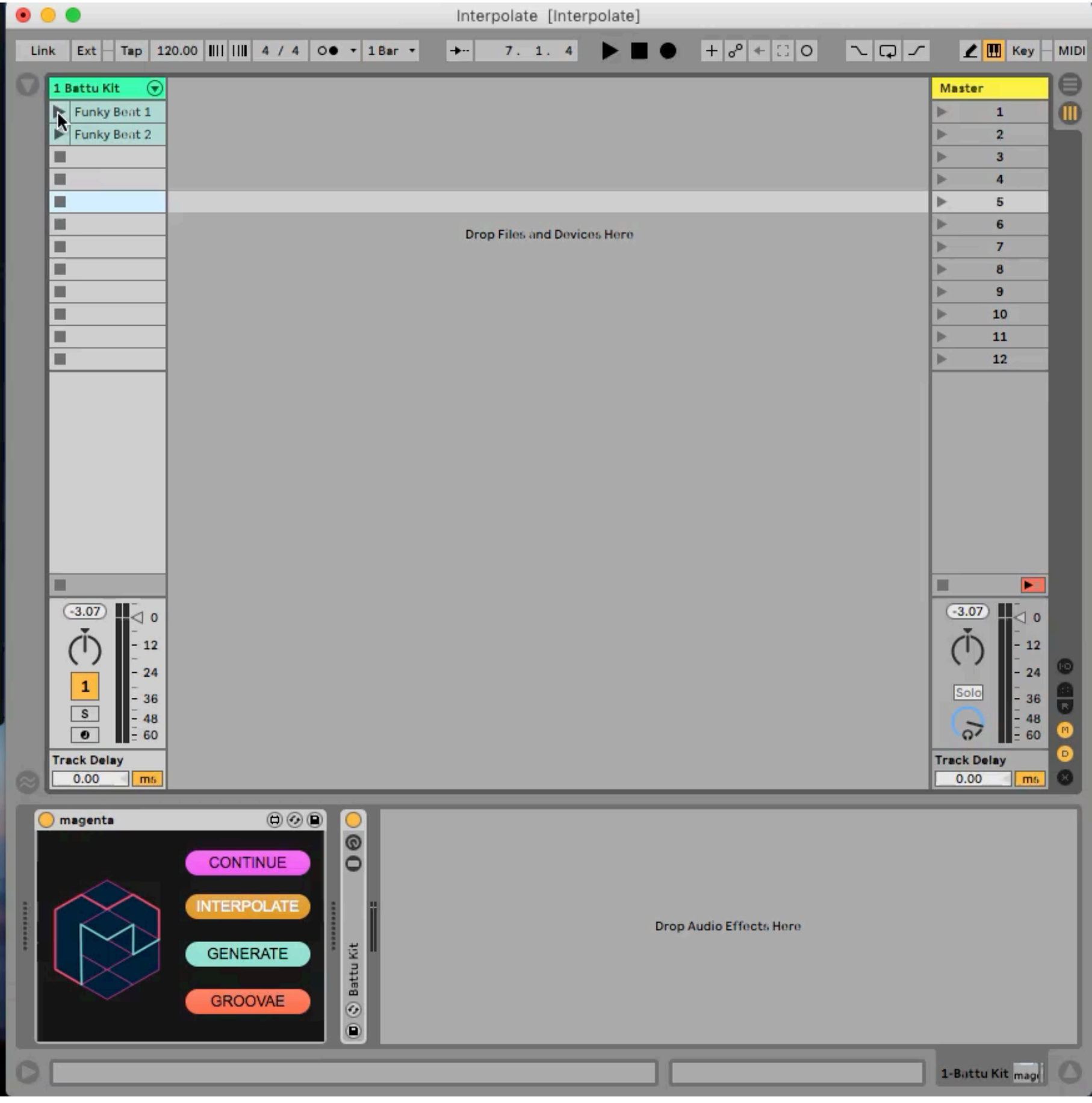
1-Grand Piano

Magenta Studio

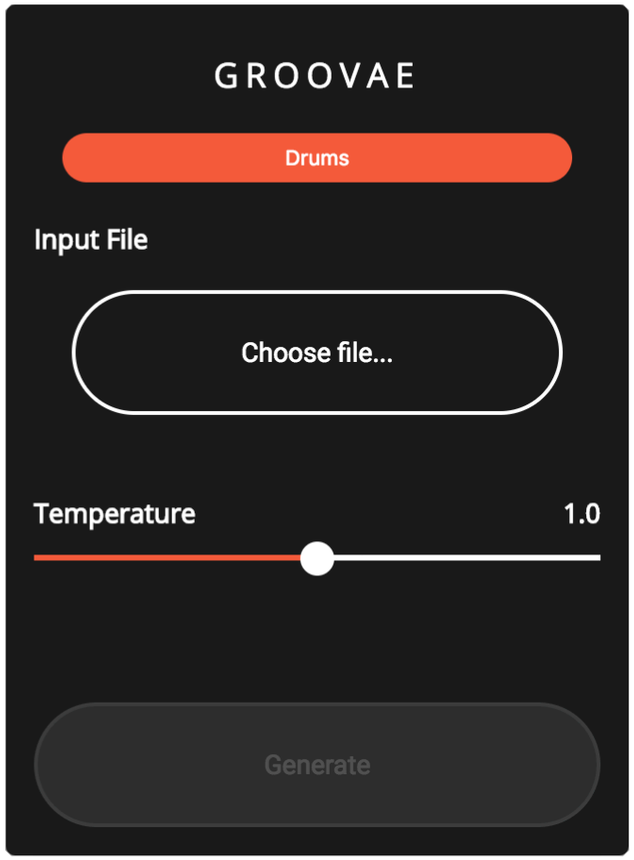


INTERPOLATE
VAE(Variational Auto Encoder)を使用し、２つのMIDIファイル（ドラムもしくはメロディー）を融合したバリエーションを生成
ファイルは４小説以下のみ


INTERPOLATE
Drums
Melody
Input Clips
1-Battu Kit
Funky Beat 1
Funky Beat 2
Steps
3
Temperature
1.0
Output 3 clips to Clip Slots 3-5
Generate

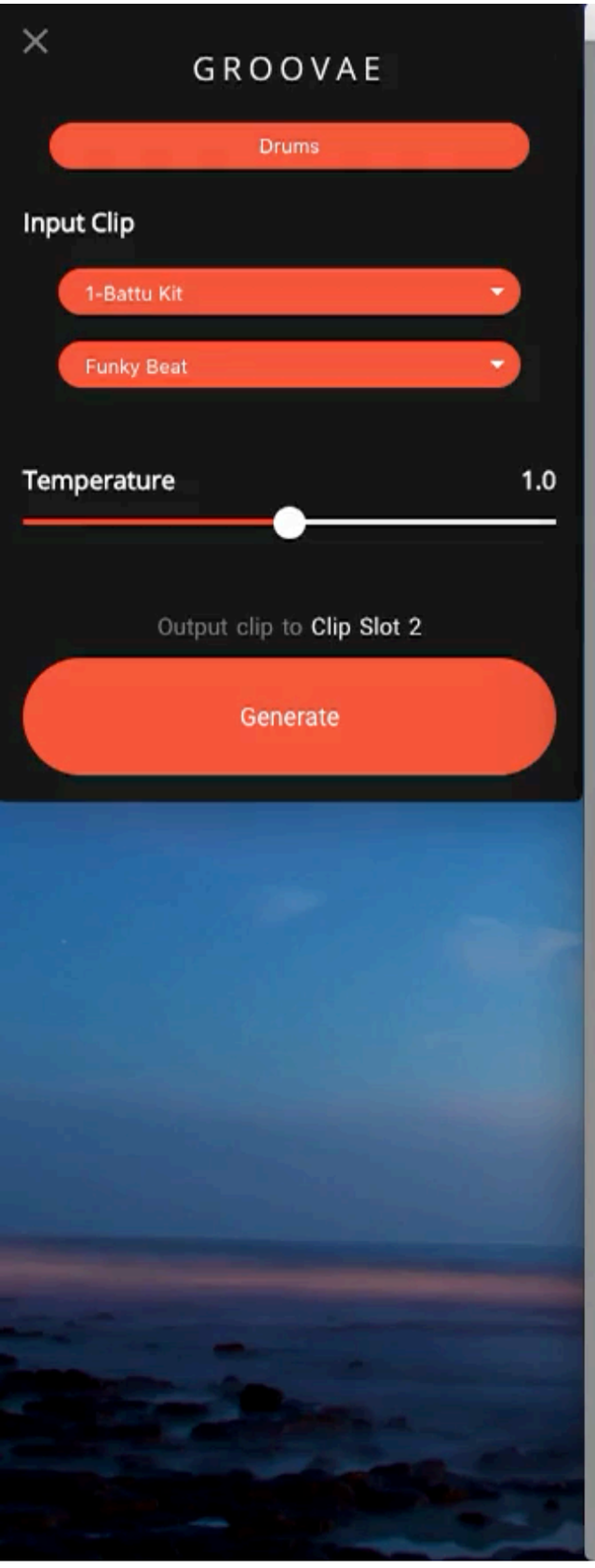

Interpolate [Interpolate]
Link
Ext
Tap
120.00
4 / 4
1 Bar
7 . 1 . 4
Key
MIDI
1 Battu Kit
Funky Beat 1
Funky Beat 2
Drop Files and Devices Here
Master
1
2
3
4
5
6
7
8
9
10
11
12
-3.07
0
12
24
36
48
60
1
S
Track Delay
0.00
ms
Solo
magenta
CONTINUE
INTERPOLATE
GENERATE
GROOVAE
Battu Kit
Drop Audio Effects Here
1-Battu Kit

Magenta Studio

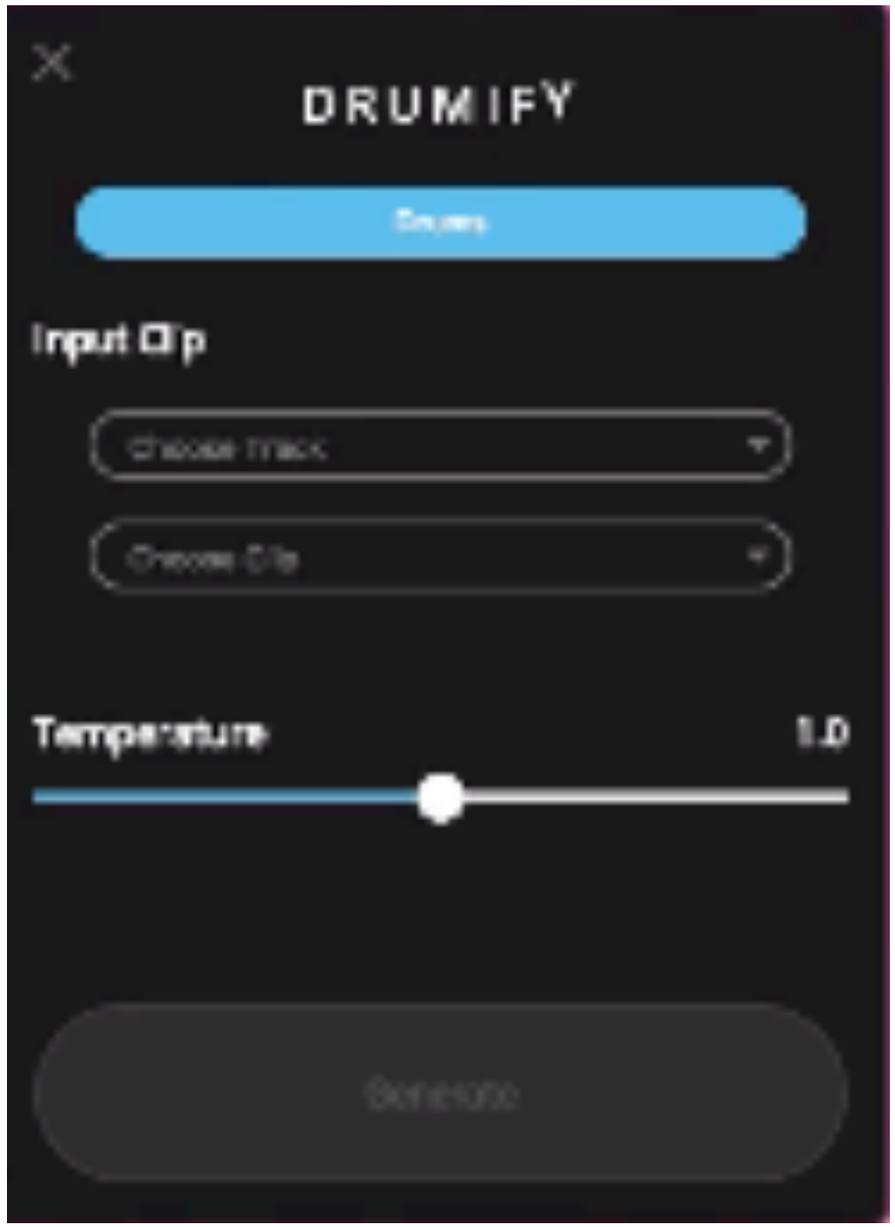


GrooVAE
入力されたドラムパターンにベロシティーやタイミングの変化でノリを加える

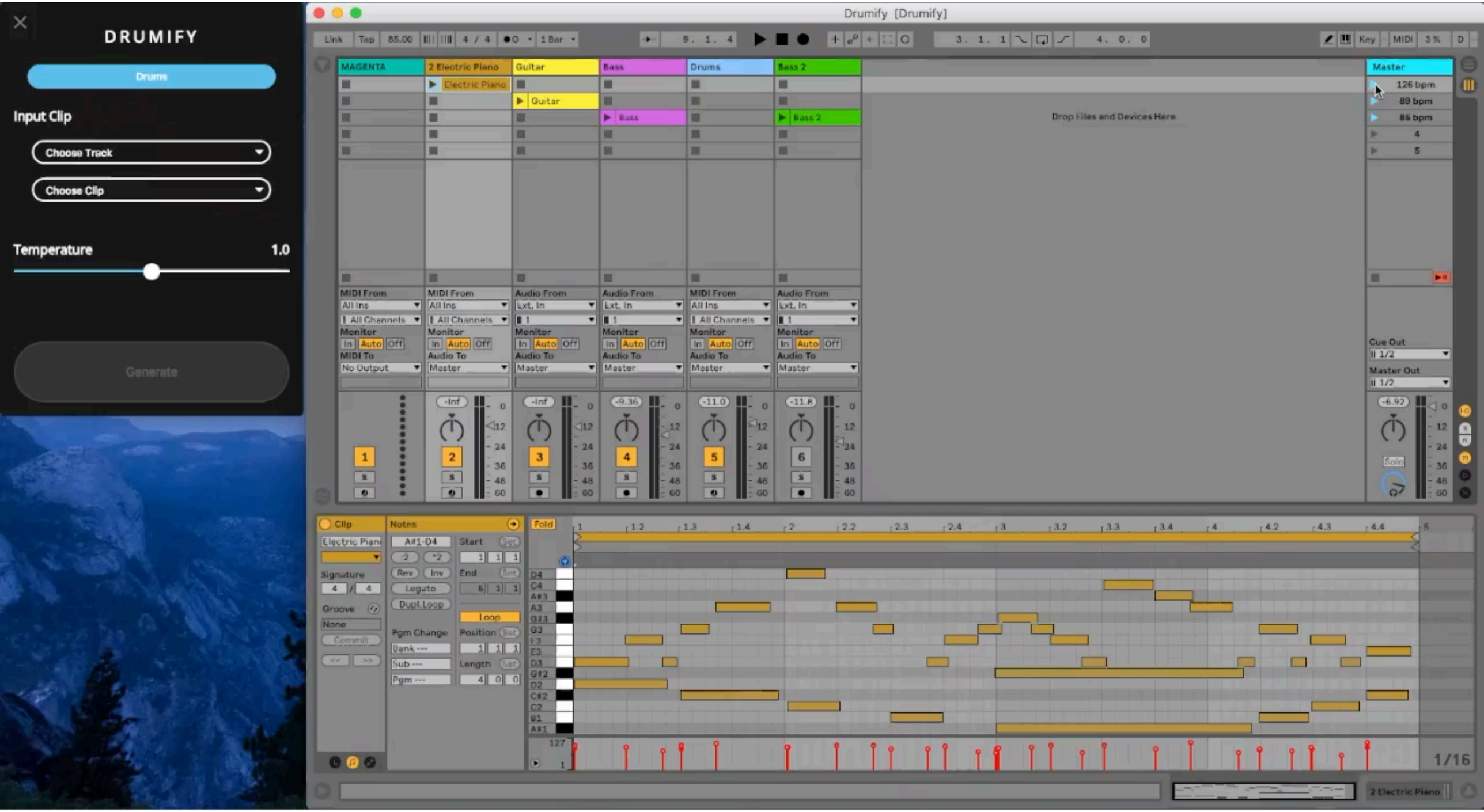
レコーディングされた人間のドラム演奏15時間分を学習
方法はGoogle翻訳に用いられたものに近いとされる


GROOVAE
Drums
Input Clip
1-Battu Kit
Funky Beat
Temperature
1.0
Output clip to Clip Slot 2
Generate
GrooVAE [Generate]
Link
Ext
Tap
120.00
4 / 4
1 Bar
175 . 3 . 2
Key
MIDI
1 %
1 Battu Kit
2 Contemporary Bass
Funky Beat
Bs1 Funk Cmaj 124bpm
Drop Files and Devices Here
Master
-2.20
-2.65
2.36
Solo
No clip selected.
1-Battu Kit

Magenta Studio



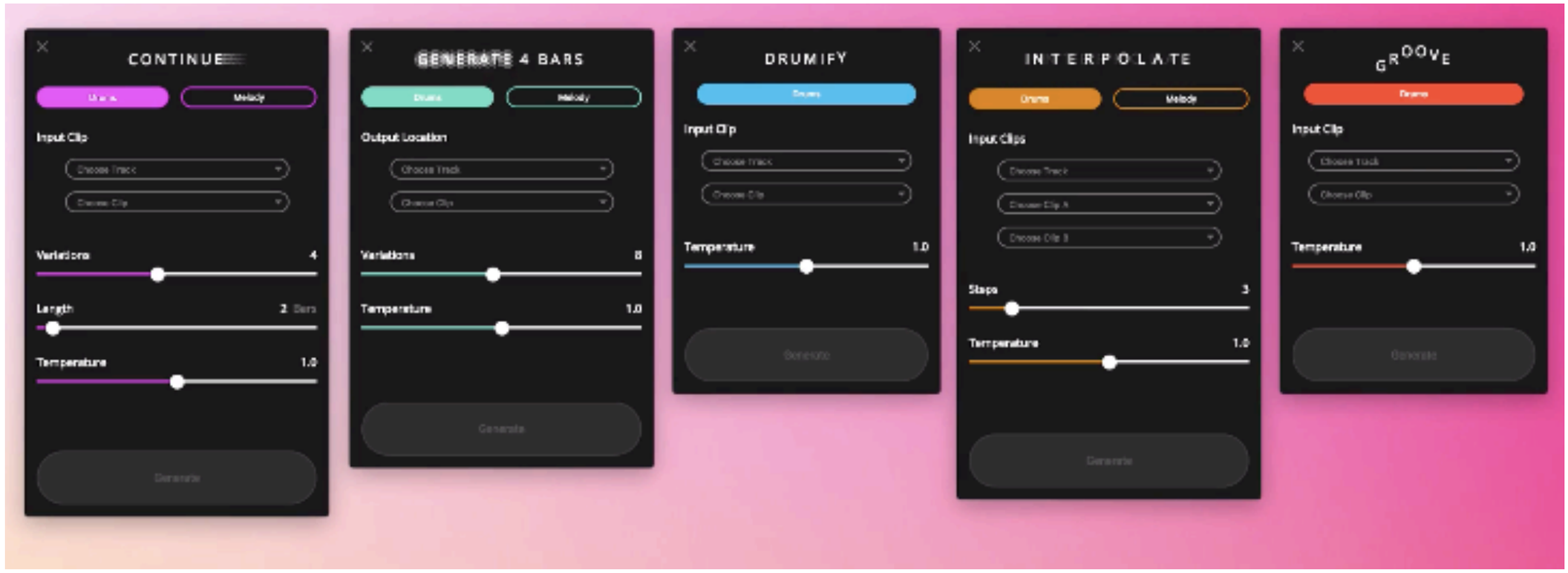
DRUMIFY
入力された楽器演奏のMIDIファイル（メロディーやベース、コード進行まで！）のパターンやアクセントに従い、適したドラムパターンを生成する


DRUMIFY
Drums
Input Clip
Choose Track
Choose Clip
Temperature
1.0
Generate
Drumify [Drumify]
Drop Files and Devices Here
Clip
Notes
Fold
Electric Piano
2 Electric Piano

LiveもMAXもないから使えない、、、

スタンドアローン版(MACのみ)
もあります

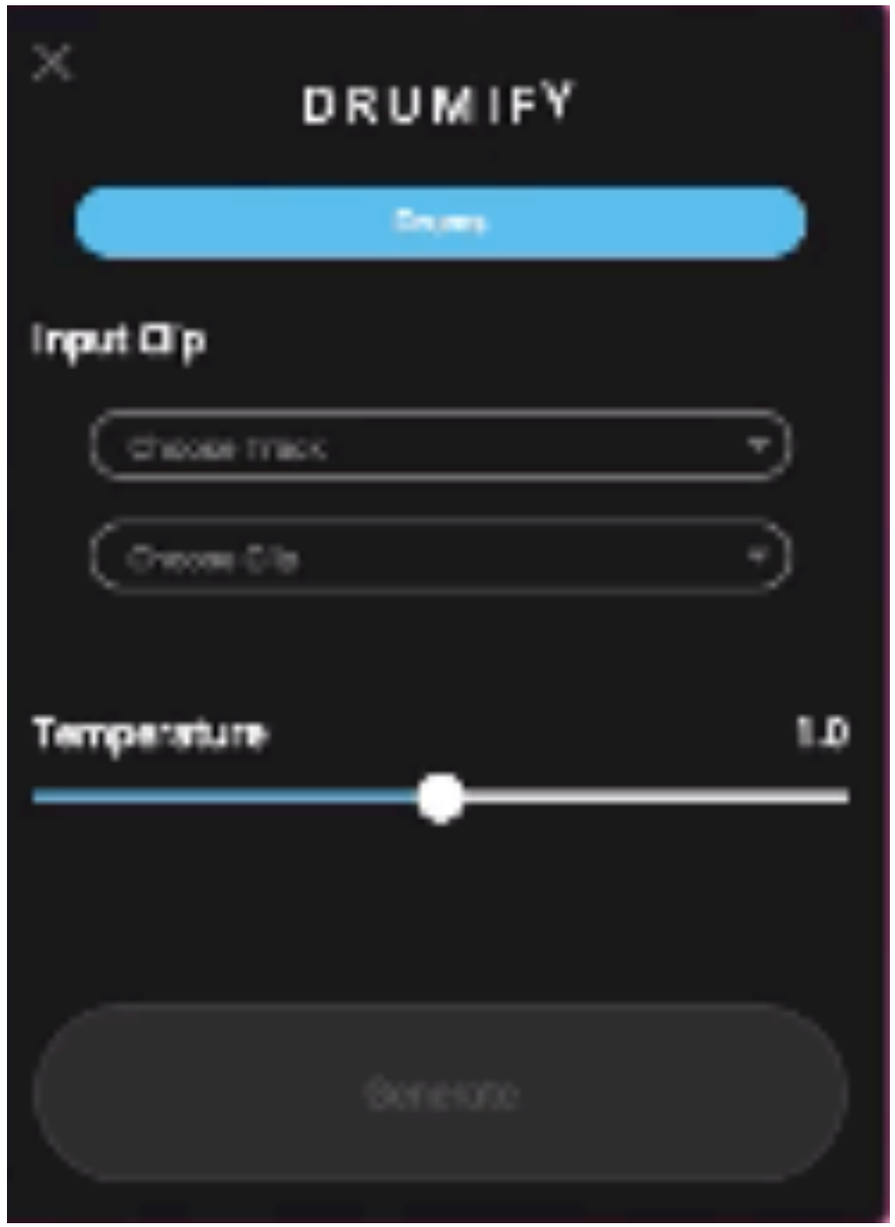
Magenta Studio スタンドアローン版



スタンドアローン版とAbleton Live版は基本的に同じ機能
違いはMIDIのインアウト
CONTINUE, INTERPOLATE, GROOVAE, GENERATE 4 BARS,
Drumifyの5種類のプラグイン

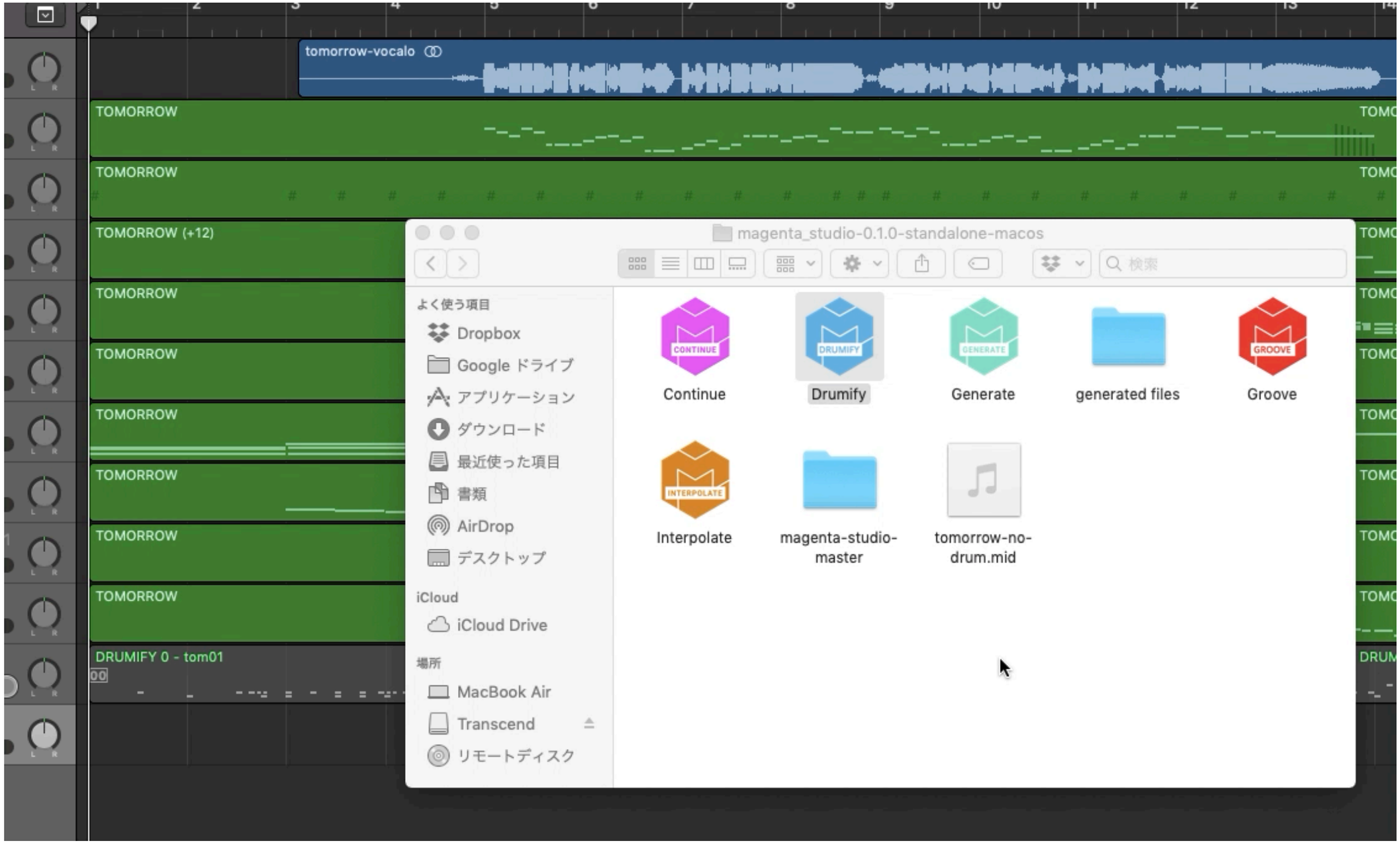
DRUMIFYでドラムトラックを
作ってみましょう

Magenta Studio



DRUMIFY
入力された楽器演奏のMIDIファイル（メロディーやベース、コード進行まで！）のパターンやアクセントに従い、適したドラムパターンを生成する

Drumify スタンドアローン（動画）



Magentaでメロディーを生成して

Drumifyでドラムトラックを作る

Magentaでメロディーを生成

Melody RNN
Attention
を使用します

Melody RNN
でメロディー（単音）生成させます

```
python magenta/models/melody_rnn/melody_rnn_generate.py \
--config=attention_rnn \
--bundle_file=attention_rnn.mag \
--output_dir=outputs/attention  \
—num_outputs=5  --num_steps=128  —primer_melody="[60]"
```

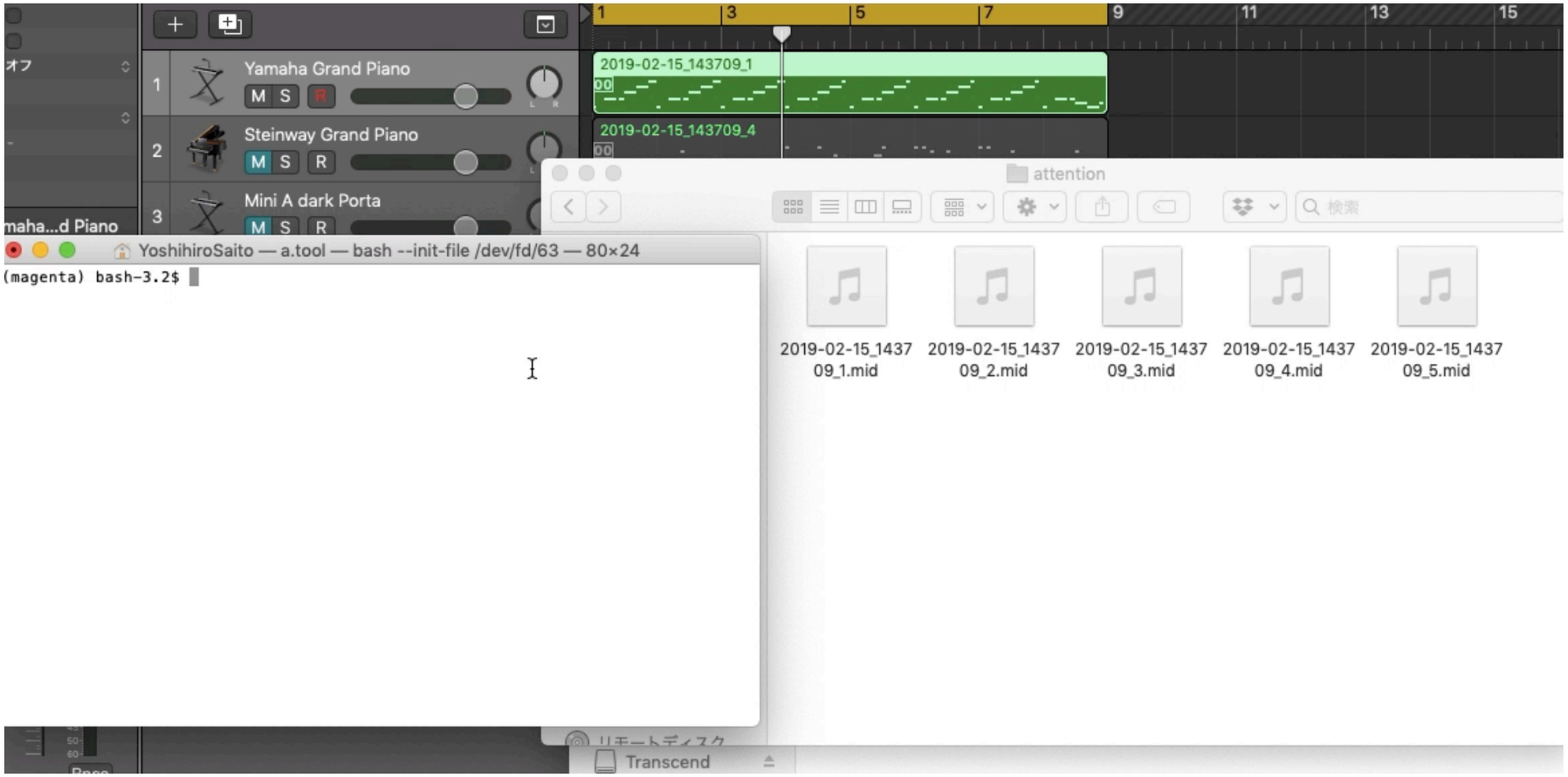
バックスラッシュ（＼）を入れる理由は
改行できないコードを改行するためです

生成する
MIDIファイルの数

生成する
MIDIファイルの長さ
１小説＝16
８小説=128

最初のガイドとなる音
は60=C3

Melody RNN メロディー（単音）生成（動画）



Melody RNNで生成したメロディーを元に
Drumifyでドラムトラックを作る

Melody RNNで生成したメロディーを元に
Drumifyでドラムトラックを作る

Melody RNNで
メロディーを作る

Melody RNN

Melody RNNの
メロディーを元に
Drumifyで
ドラムトラックを
作る

Drumify

合わせて
１曲にしてみる

Melody RNN

Drumify

Ableton Live
で使用できる他のMagentaプラグイン

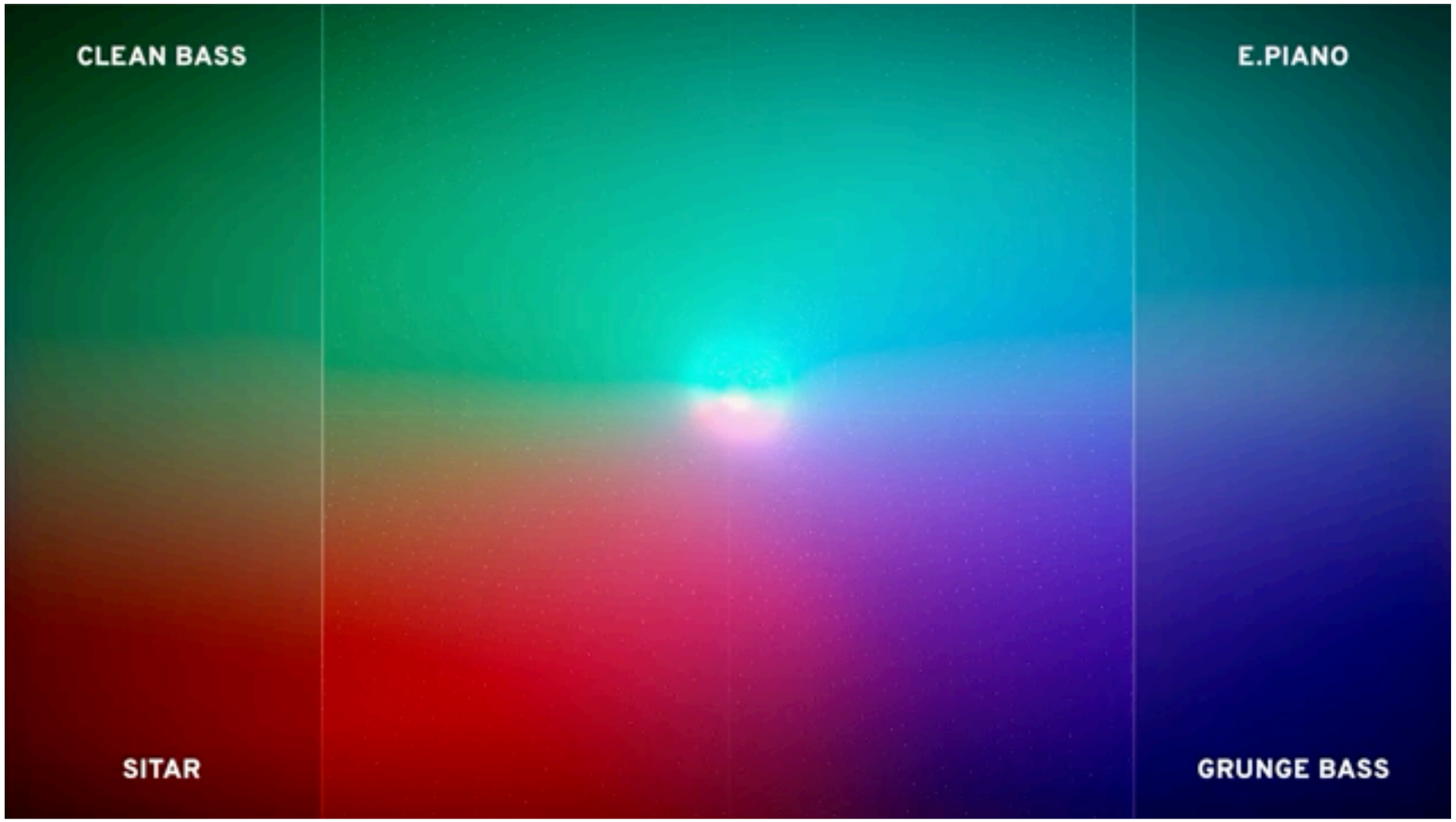
Nsynth

Nsynthとは

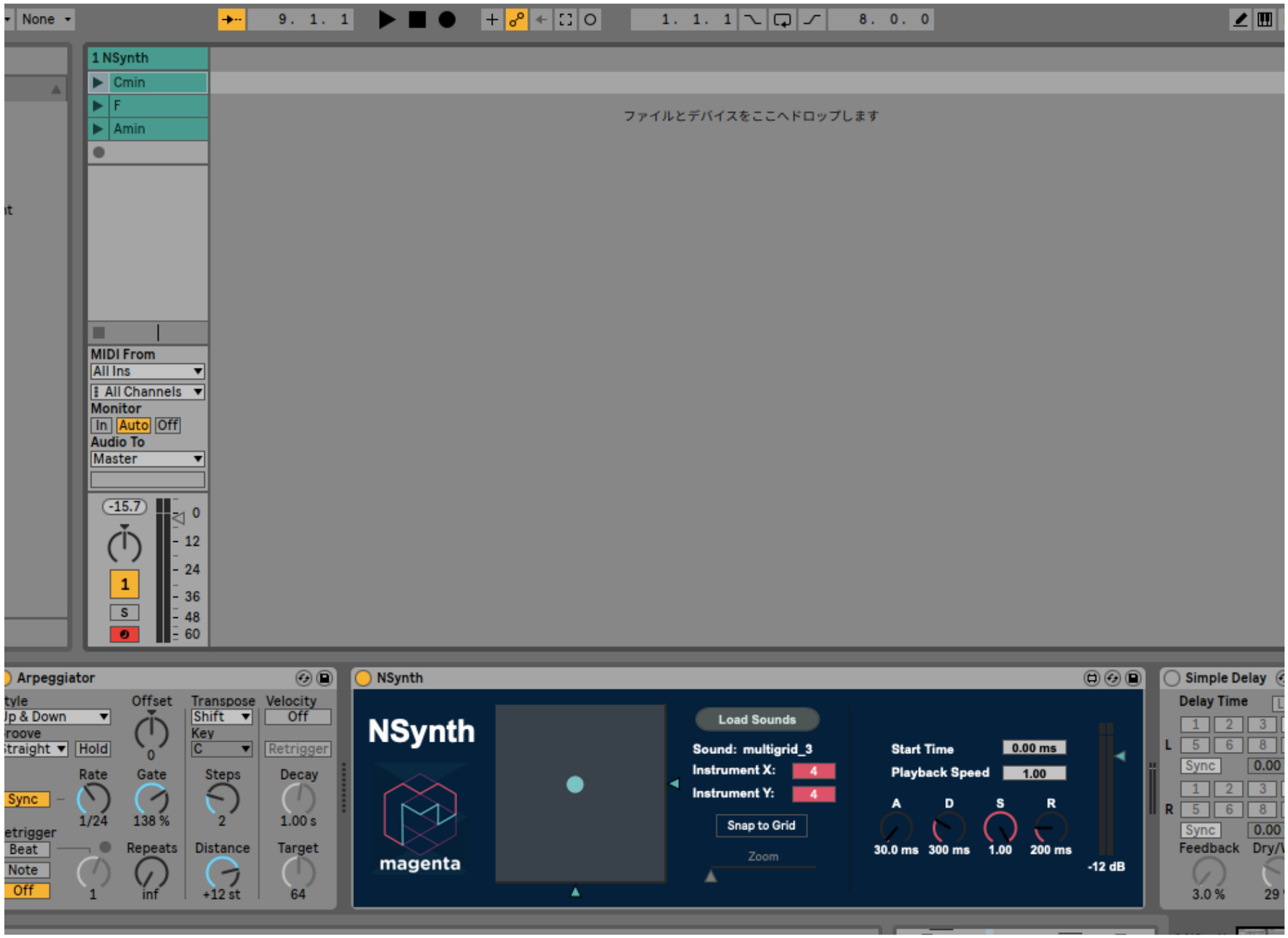
機械学習（ディープラーニング）を使用し、音の波形から特徴を１６要素抽出。

２つの音色をミックスする事で新しい音色を演奏する事ができるシンセサイザー

Nsynthとは（動画）

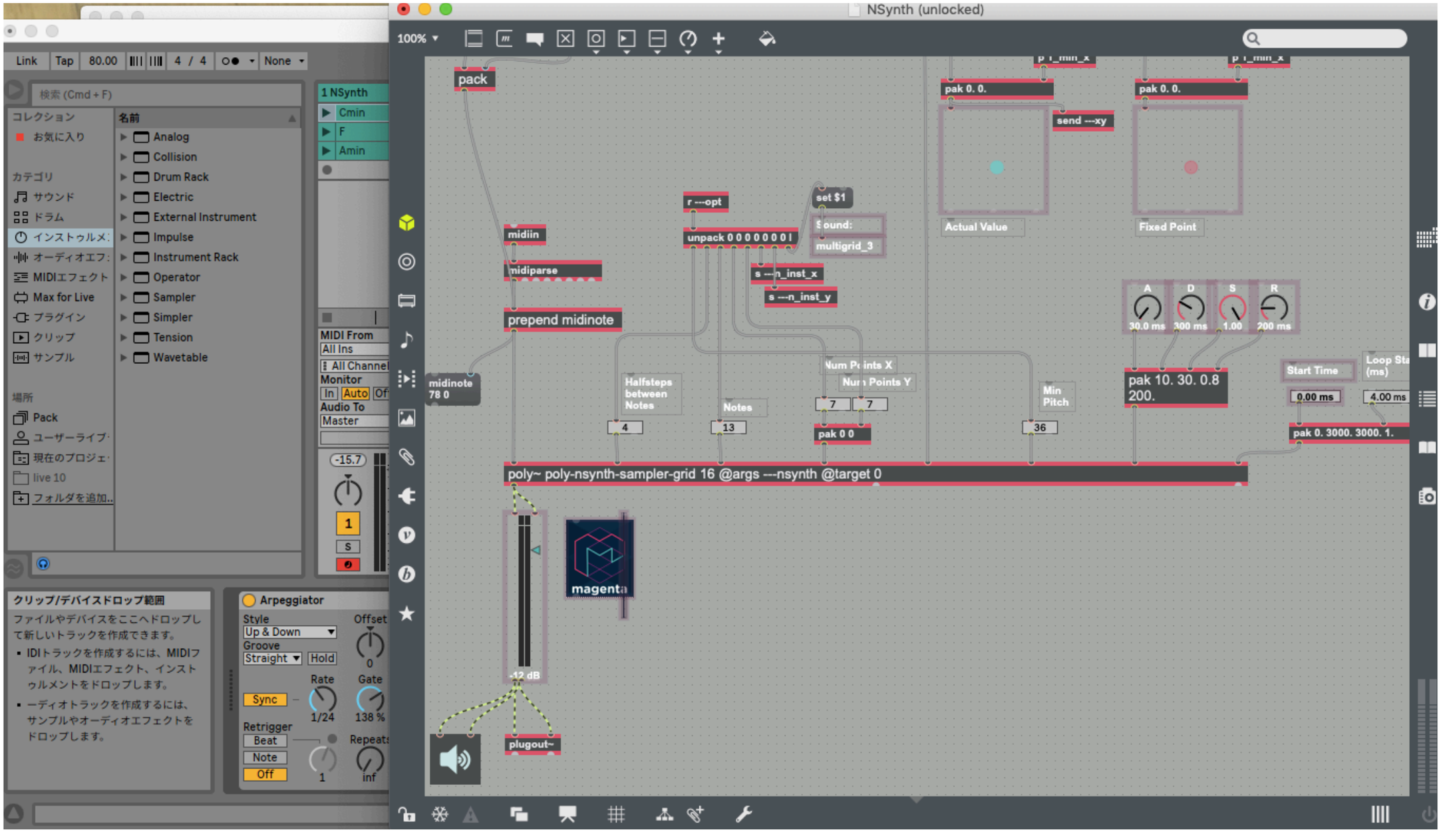


Nsynth
Ableton Liveのプラグインとして使用できます



Nsynth

MAX for Liveでエディットや開発もできます



Ableton LiveとMAX
を使用してMagentaプラグインを
ご自身で作る事が可能です！

AIを使用したマスタリングソフト

iZotopeのOzone8

AIを使用したマスタリングソフト
iZotopeのOzone8

AIを使用したマスタリングソフト
iZotopeのOzone8

他の楽曲の波形データを参照して、
それに似たマスタリングを自動でしてくれる

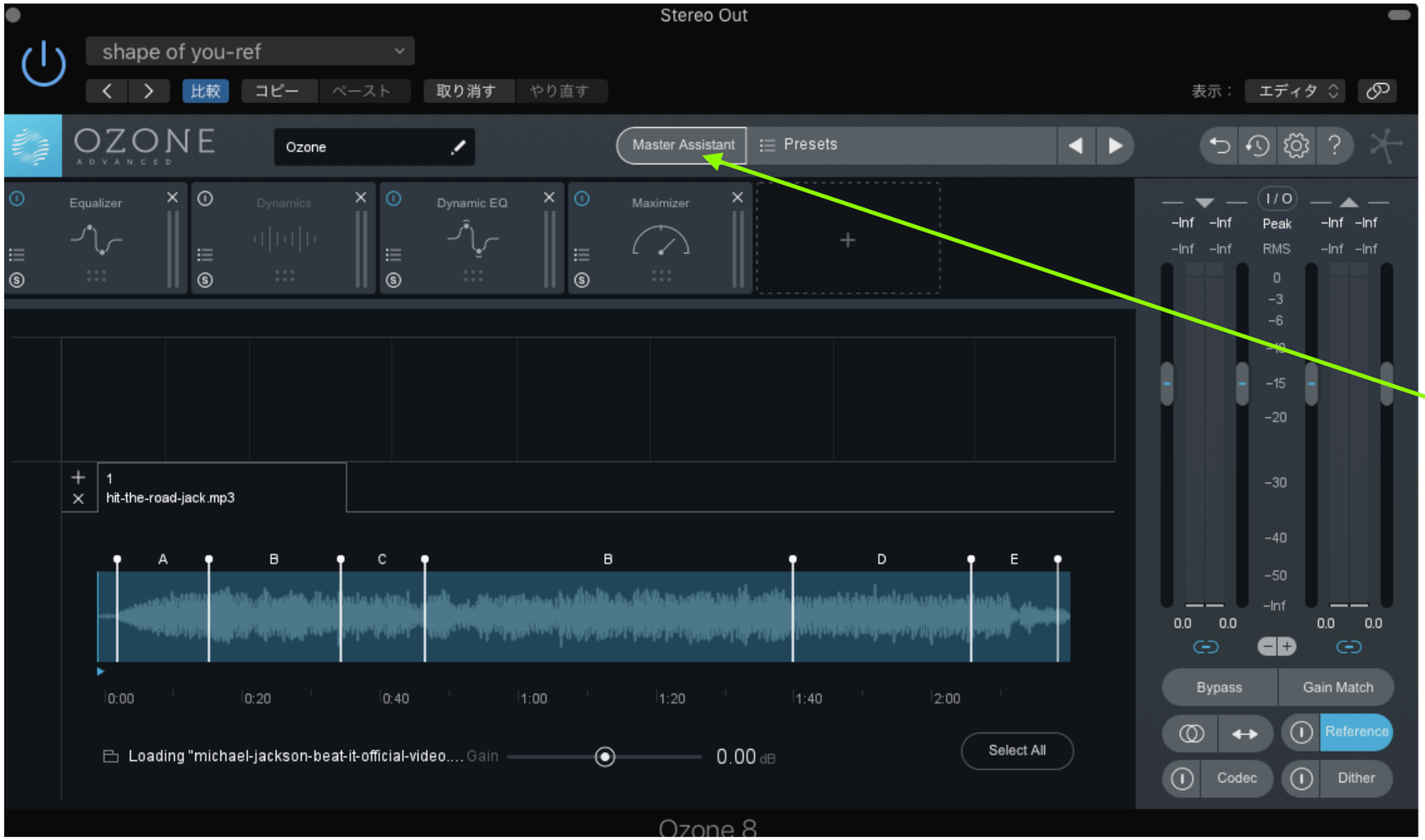
iZotopeのOzone8

reference
をクリック

iZotopeのOzone8

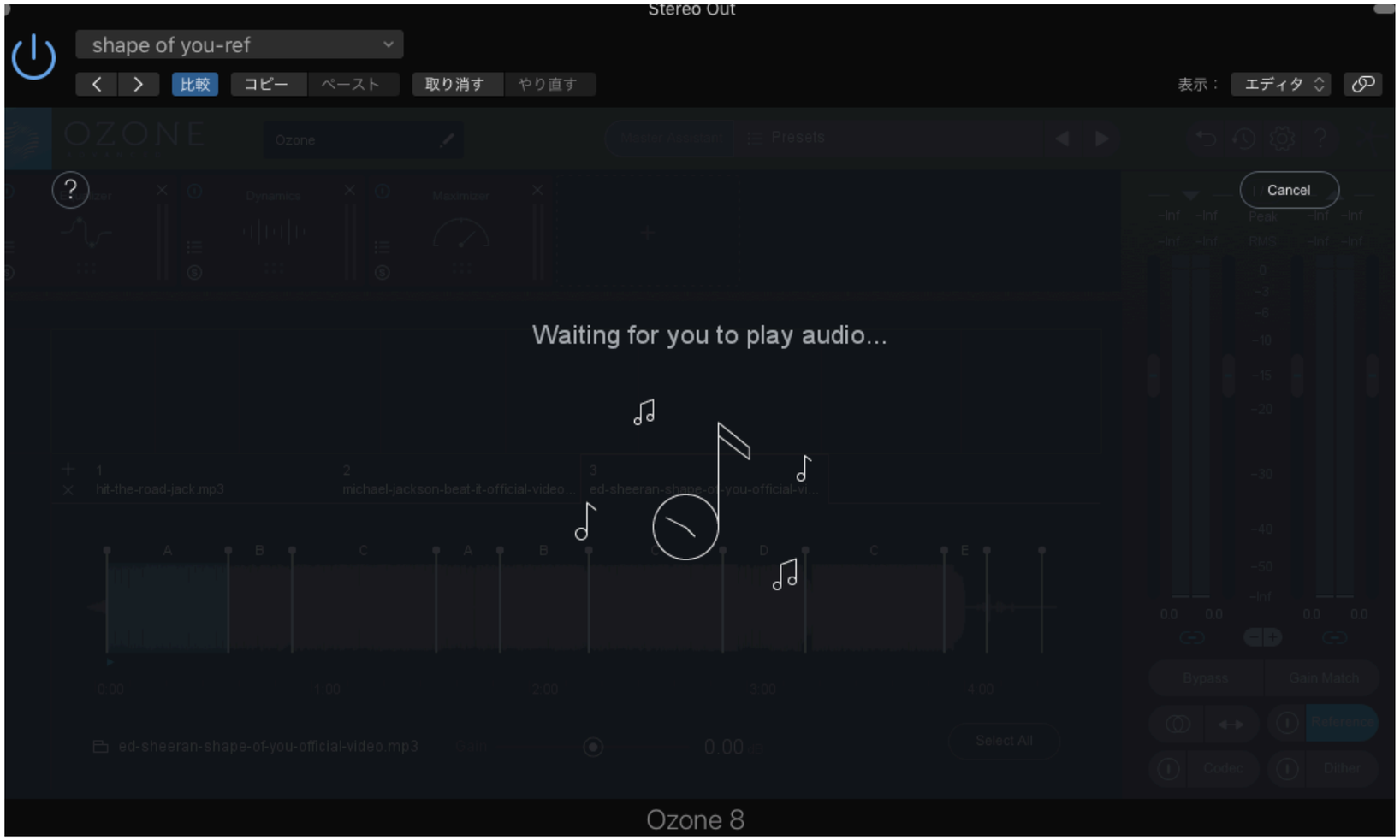
+をクリック
してロード

iZotopeのOzone8



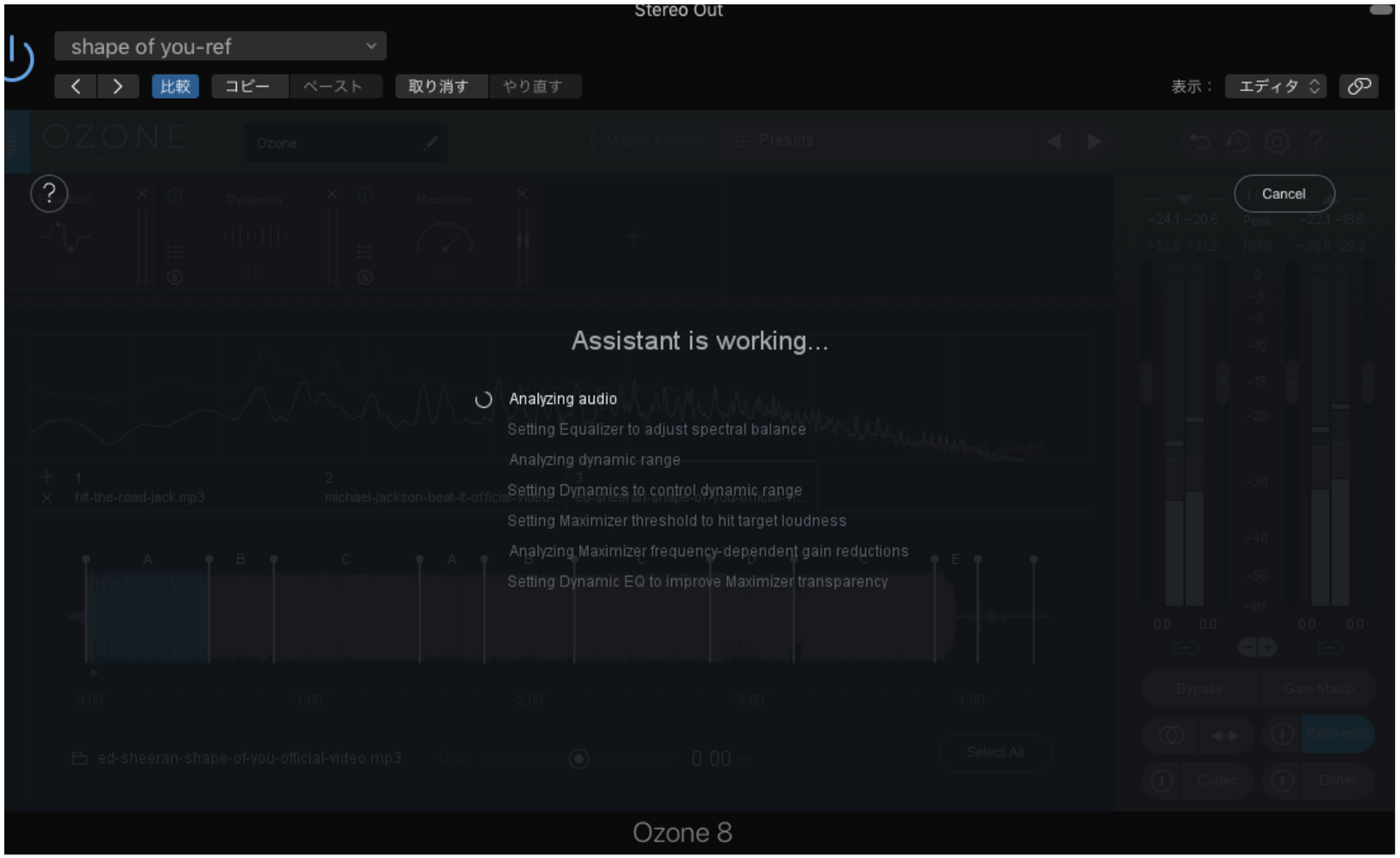
Mastering Assistant
をクリック

iZotopeのOzone8



準備画面になるので
曲の１番ボリューム
の高いところを
再生

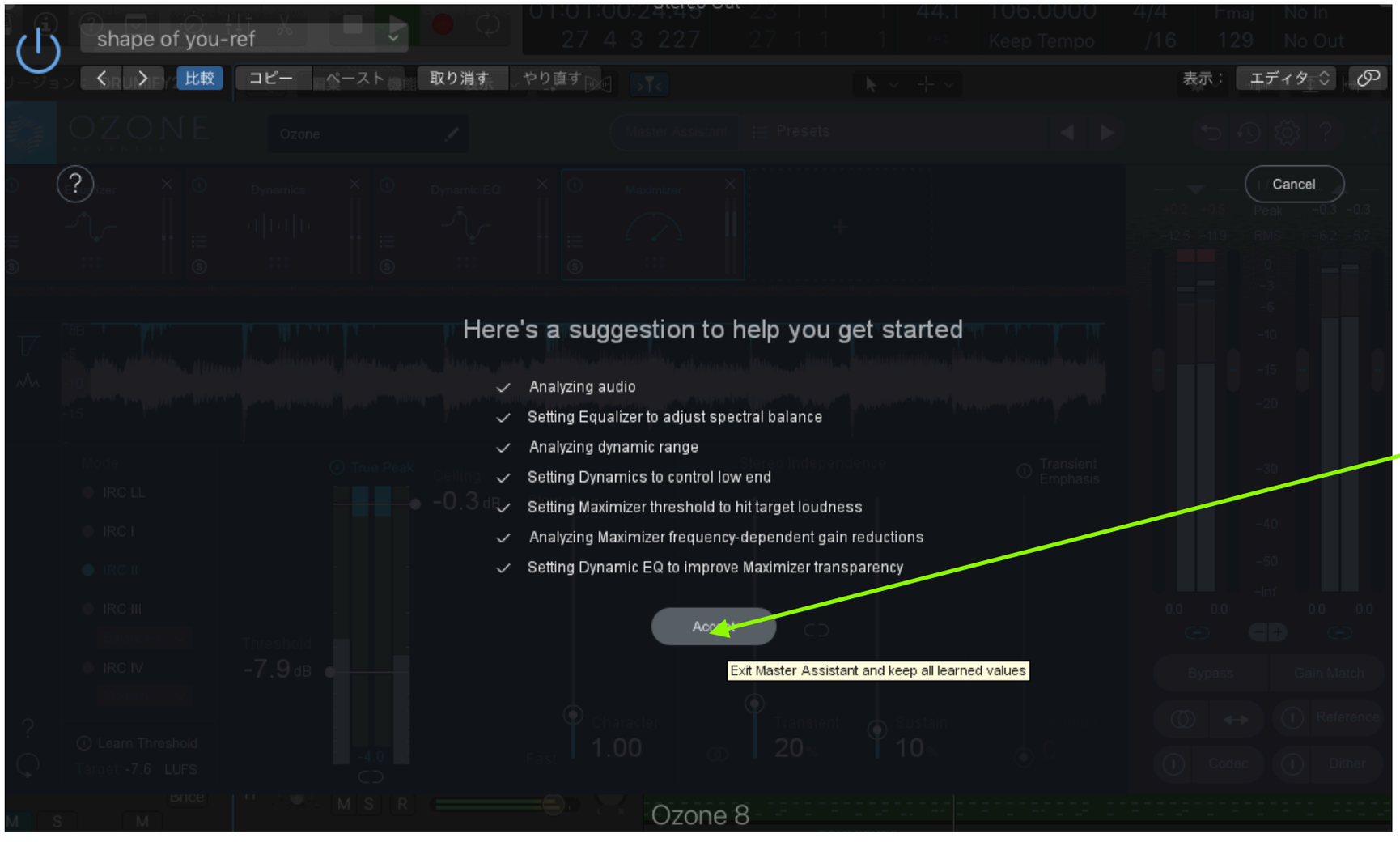
iZotopeのOzone8



Ozone 8

Masteringが
開始されます

iZotopeのOzone8



Mastering
を採用の場合
acceptボタン
を押す

ディープラーニングのプログラミング基礎

Magentaの実践的解説

Tensorflowの基礎

Magentaはtensorflowをベースにしています

tensorflowのtf.app.flags
モジュール

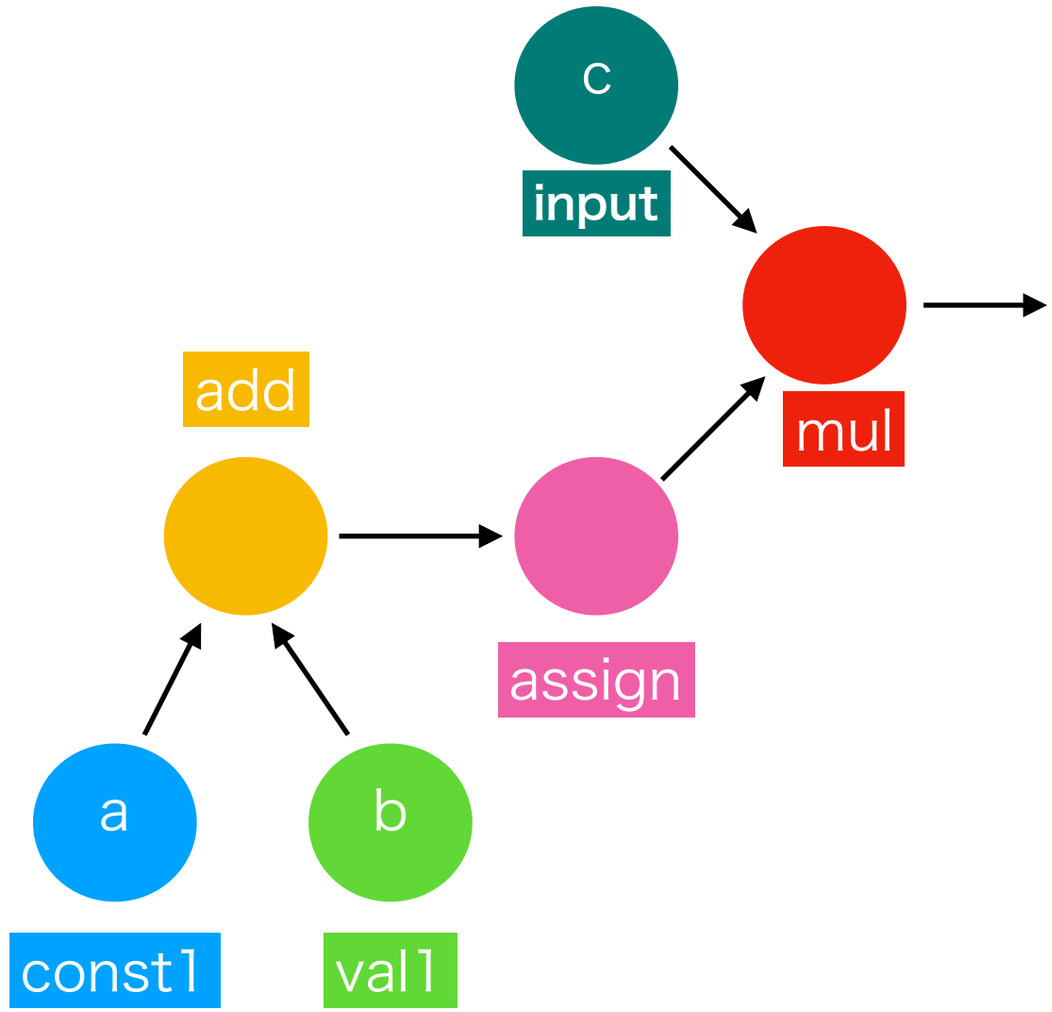
```
import tensorflow as tf

FLAGS = tf.app.flags.FLAGS

tf.app.flags.DEFINE_string(
    'run_dir', None,
    'Path to the directory where the latest checkpoint will be loaded from.')

tf.app.flags.DEFINE_string(
    'checkpoint_file', None,
    'Path to the checkpoint file. run_dir will take priority over this flag.')

tf.app.flags.DEFINE_string(
    'bundle_file', None,
    'Path to the bundle file. If specified, this will take priority over '
    'run_dir and checkpoint_file, unless save_generator_bundle is True, in '
    'which case both this flag and either run_dir or checkpoint_file are '
    'required')
```

簡単な計算グラフのコードで
Tensorflowの基礎を学びます

tf計算グラフのコード

```
import tensorflow as tf

a = tf.constant(3, name='const1') #定数a
b = tf.Variable(0, name='val1') #変数b

# aとbを足す
add = tf.add(a,b)

#変数bに足した結果をアサイン
assign = tf.assign(b, add)
c = tf.placeholder(tf.int32, name='input') #入力c

# アサインした結果とcを掛け算
mul = tf.multiply(assign, c)

#変数の初期化
init = tf.global_variables_initializer()

with tf.Session() as sess:
    #初期化を実行
    sess.run(init)
    for i in range(3):
        # 掛け算を計算するまでのループを実行
        print(sess.run(mul, feed_dict={c:3}))
```

TensorFlowでは
計算グラフは入力→処理→出力をノードとエッジで表す



TensorFlowの計算グラフにおいて各ノード（丸）はオペレーション（何らかの計算処理）を意味する。
オペレーションは必ず何かしらのTensorを出力します。
そのため、計算グラフのエッジ（矢印）はどのオペレーションの出力Tensorがどのオペレーションに入力されたかという流れを表します。

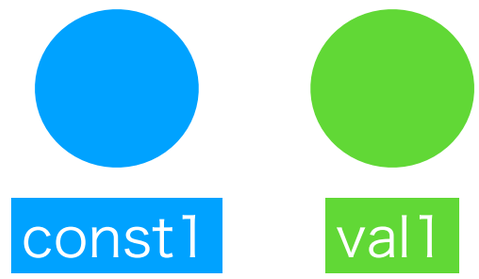
tf計算グラフのコード

```
import tensorflow as tf
```

tensorflowのインポートはこの１行で行えます（要インストール）

```
a = tf.constant(3, name='const1') #定数a
b = tf.Variable(0, name='val1') #変数b
```

tf.constantは定数を、tf. Variableは変数を出力するオペレーション
tf.constantは一度値を決めると変更不可
tf. Variableは後でも変更可能
tf.constantの引数はint（整数）もしくはfloat（少数）
tf. Variableの初期値にはそれ以外にtensorオブジェクトを与えることもできる
nameで名前を定義



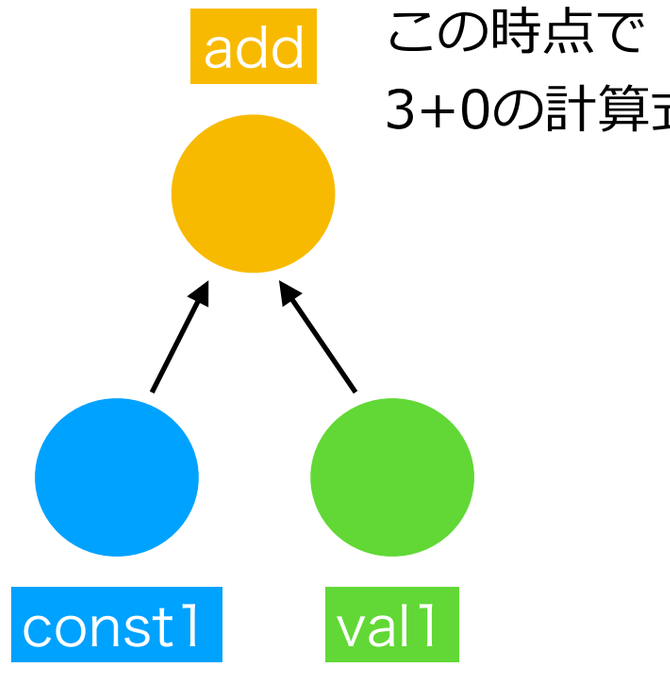
定数:3
変更不可

変数:初期値0
変更可能

tf計算グラフのコード

```
a = tf.constant(3, name='const1') #定数a
b = tf.Variable(0, name='val1') #変数b
add = tf.add(a,b)
```

tf.add()で定数a（3）と変数b（0）を
加算する処理を計算グラフに追加
3+0なので3
（tf.addは計算式であって実際の処理はまた別のオペレーションで行います）



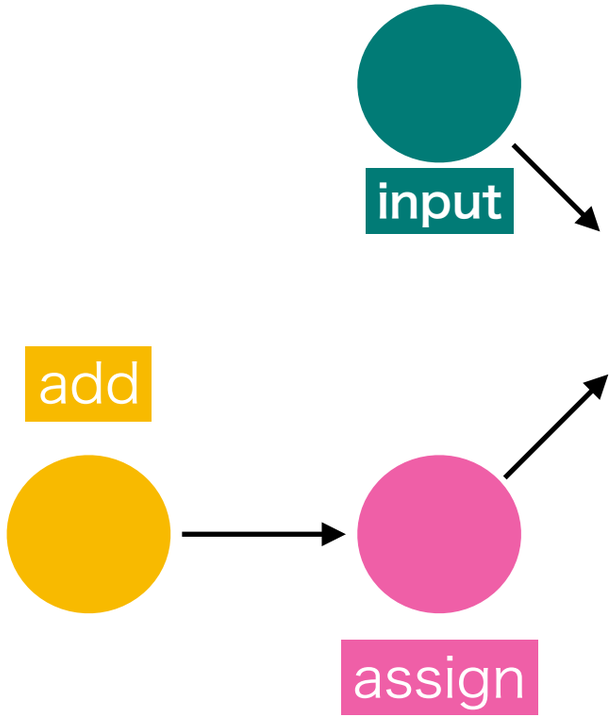
この時点で
3+0の計算式

定数:3
変更不可

変数:初期値0
変更可能

tf計算グラフのコード

```
#変数bに足した結果をアサイン
assign = tf.assign(b, add)
c = tf.placeholder(tf.int32, name='input') #入力c
```



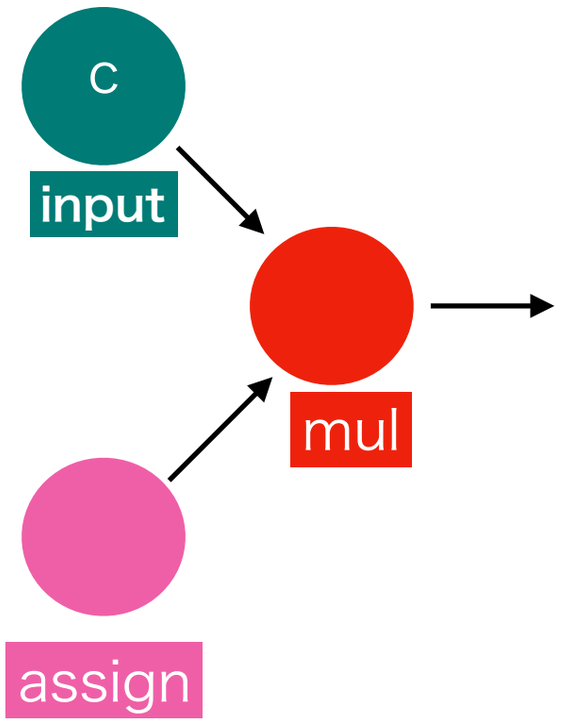
tf.assign()は値を変数に与えるオペレーションで
tf.Variable()を構成する要素の一つ
ここでは先ほどの変数b(初期値０)にaddの加算結果(3)を
アサイン

tf.placeholder()は、実行時に引数の値を指定するオペレーション
tf.constantやtf.Variableの様に初期値を指定しない
型だけを決めておき、実行時に辞書型で値を決める
ここではc（名称input)をint（整数型）で指定

tf計算グラフのコード

```
# アサインした結果とcを掛け算
mul = tf.multiply(assign, c)
```

tf.multiplyは乗算（掛け算）
ここではassign(3)と実行時にcとを乗算



tf計算グラフのコード

```
#変数の初期化
init = tf.global_variables_initializer()

with tf.Session() as sess:
    #初期化を実行
    sess.run(init)
    for i in range(3):
        # 掛け算を計算するまでのループを実行
        print(sess.run(mul, feed_dict={c:3}))
```

tf.global_variables_initializer()で変数を初期化

tensorflowで計算、グラフ表示する場合はそれまで定義した
全ての変数をinitializerに従って初期化する必要があります

tf.Sessionで計算結果を作成したグラフを実行しprint関数で
表示する事ができます

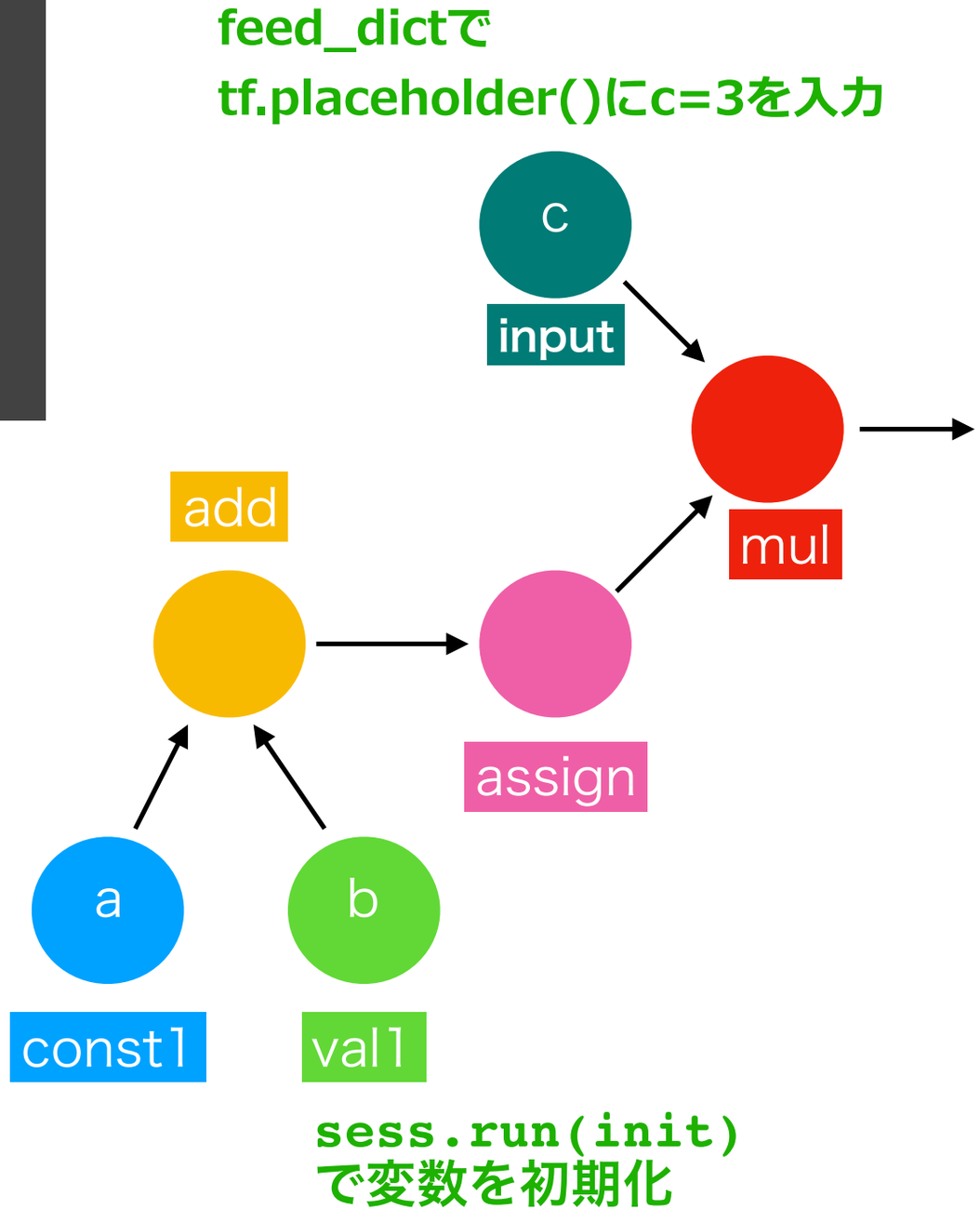
tf計算グラフのコード

```
#変数の初期化
init = tf.global_variables_initializer()

with tf.Session() as sess:
    #初期化を実行
    sess.run(init)
    for i in range(3):
        # 掛け算を計算するループ回数
        print(sess.run(mul, feed_dict={c:3}))
```

sess.run(init)で変数を初期化
ループ文（for文）で計算回数を指定
feed_dictでtf.placeholder()にC:3を入力
この実行ではC=3となる

assignに３が入力、input（変数c)に3が入力
これを乗算するので出力は9
２回目はassignに6が入力、inputは変わらず3が入力
これを乗算するので出力は18
３回目はassign9で出力は27

ループ文でmulだけの実行の様に見えますがmulに関わる全ての計算式が実行されます



tf計算グラフのコード

```
import tensorflow as tf

a = tf.constant(3, name='const1') #定数a = 3
b = tf.Variable(0, name='val1') #変数b (1回目 =0)   (2回目 =3)   (3回目 =6)

# aとbを足す
add = tf.add(a,b) # (1回目 add=3+0)   (2回目 add=3+3)   (3回目 add=3+6)

#変数bに足した結果をアサイン
assign = tf.assign(b, add) # (1回目 assign=3)   (2回目 assign=6)   (3回目 assign=9)
c = tf.placeholder(tf.int32, name='input') #入力cは(sess.run(mul, feed_dict={c:3}))で定義

# アサインした結果とcを掛け算
mul = tf.multiply(assign, c) # (1回目 (assign=3)*c)   (2回目 (assign=6)*c)   (3回目
(assign=9)*c)

#変数の初期化
init = tf.global_variables_initializer()

with tf.Session() as sess:
    #初期化を実行
    sess.run(init)
    for i in range(3):  # sess.runのmulに関わる計算式のみ繰り返し実行されます
        # 掛け算を計算するループ回数
        print(sess.run(mul, feed_dict={c:3}))
#(1回目 (assign=3)*(c=3))=9 (2回目  (assign=6)*(c=3))=18   (3回目  (assign=9)*(c=3))=27
```

tf計算グラフのコード　loop 1 回目

```
import tensorflow as tf

a = tf.constant(3, name='const1') #定数a = 3
b = tf.Variable(0, name='val1') #変数b (1回目 =0)

# aとbを足す
add = tf.add(a,b) # (1回目 add=3+0)

#変数bに足した結果をアサイン
assign = tf.assign(b, add) # (1回目 assign=3)
c = tf.placeholder(tf.int32, name='input') #入力cは(sess.run(mul, feed_dict={c:3}))で定義

# アサインした結果とcを掛け算
mul = tf.multiply(assign, c) # (1回目 (assign=3)*c)

#変数の初期化
init = tf.global_variables_initializer()

with tf.Session() as sess:
    #初期化を実行
    sess.run(init)
    for i in range(3):  # sess.runのmulに関わる計算式のみ繰り返し実行されます
        # 掛け算を計算するループ回数
        print(sess.run(mul, feed_dict={c:3}))  #(1回目 (assign=3)*(c=3))=9
```

tf計算グラフのコード　loop２回目

```
import tensorflow as tf

a = tf.constant(3, name='const1') #定数a = 3
b = tf.Variable(0, name='val1') #変数b (2回目 =3)

# aとbを足す
add = tf.add(a,b) # (2回目 add=3+3)

#変数bに足した結果をアサイン
assign = tf.assign(b, add) # (2回目 assign=6)
c = tf.placeholder(tf.int32, name='input') #入力cは(sess.run(mul, feed_dict={c:3}))で定義

# アサインした結果とcを掛け算
mul = tf.multiply(assign, c) # (2回目 (assign=6)*c)

#変数の初期化
init = tf.global_variables_initializer()

with tf.Session() as sess:
    #初期化を実行
    sess.run(init)
    for i in range(3):  # sess.runのmulに関わる計算式のみ繰り返し実行されます
        # 掛け算を計算するループ回数
        print(sess.run(mul, feed_dict={c:3})) # (２回目  (assign=6)*(c=3))=18
```

tf計算グラフのコード　loop3回目

```
import tensorflow as tf

a = tf.constant(3, name='const1') #定数a = 3
b = tf.Variable(0, name='val1') #変数b (3回目 =6)

# aとbを足す
add = tf.add(a,b) # (3回目 add=3+6)

#変数bに足した結果をアサイン
assign = tf.assign(b, add) # (3回目 assign=9)
c = tf.placeholder(tf.int32, name='input') #入力cは(sess.run(mul, feed_dict={c:3}))で定義

# アサインした結果とcを掛け算
mul = tf.multiply(assign, c) #  (3回目 (assign=9)*c)


#変数の初期化
init = tf.global_variables_initializer()

with tf.Session() as sess:
    #初期化を実行
    sess.run(init)
    for i in range(3):  # sess.runのmulに関わる計算式のみ繰り返し実行されます
        # 掛け算を計算するループ回数
        print(sess.run(mul, feed_dict={c:3})) # (3回目  (assign=9)*(c=3))=27
```