Telemetry データによる

深層異常検知

JANOG45

伊藤忠テクノソリューションズ株式会社

池上 佳一

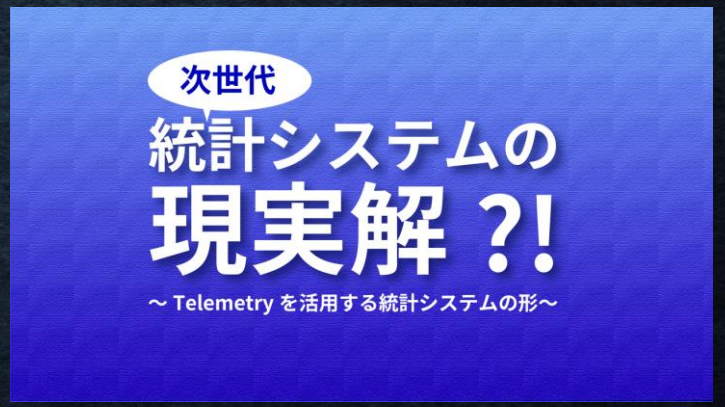
自己紹介

池上佳一

伊藤忠テクノソリューションズ株式会社
情報通信第三本部　技術統轄部

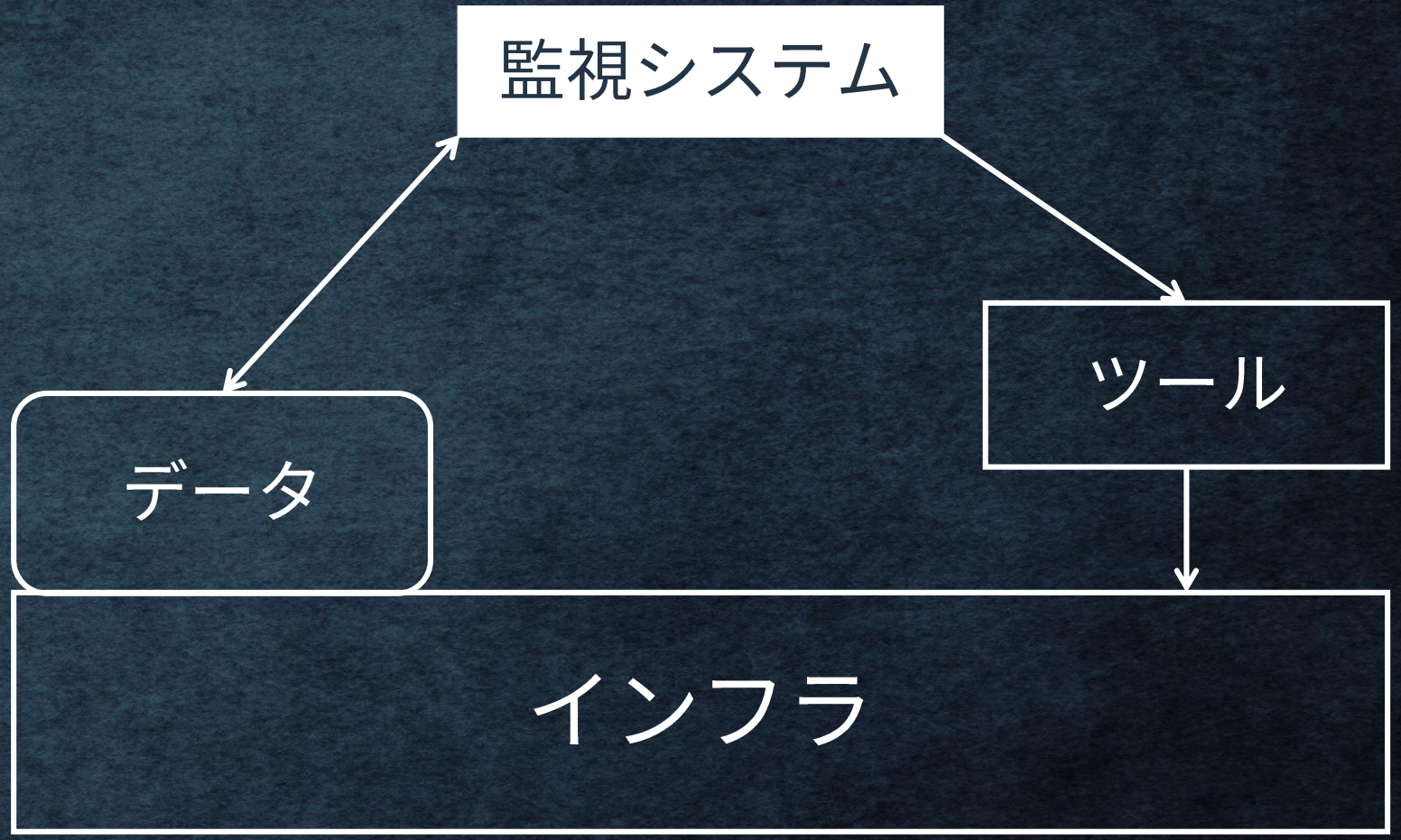
子ども **3人** (長男・次男・長女)
趣味：子どもの写真を撮ること

JANOG43　次世代統計システムの現実解～Telemetryを活用する統計システムの形～



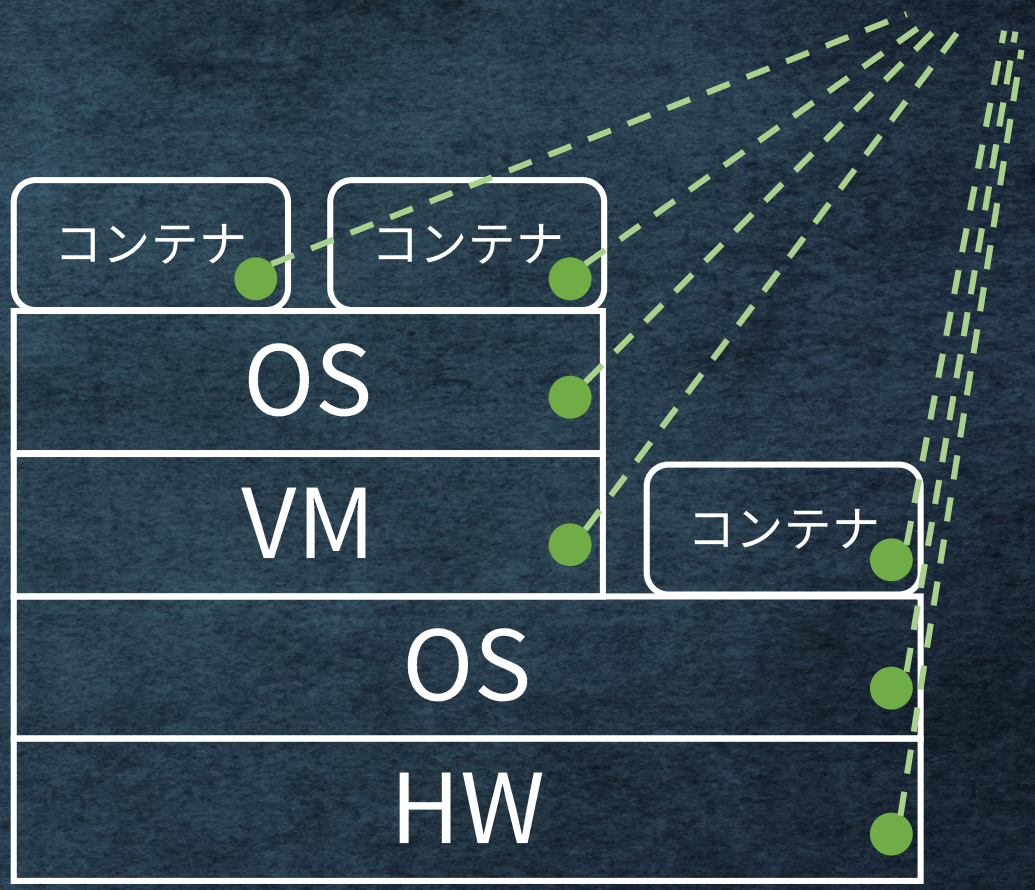
背景

自動化が進む中で、トリガーとなる監視システムは重要な役割を果たす

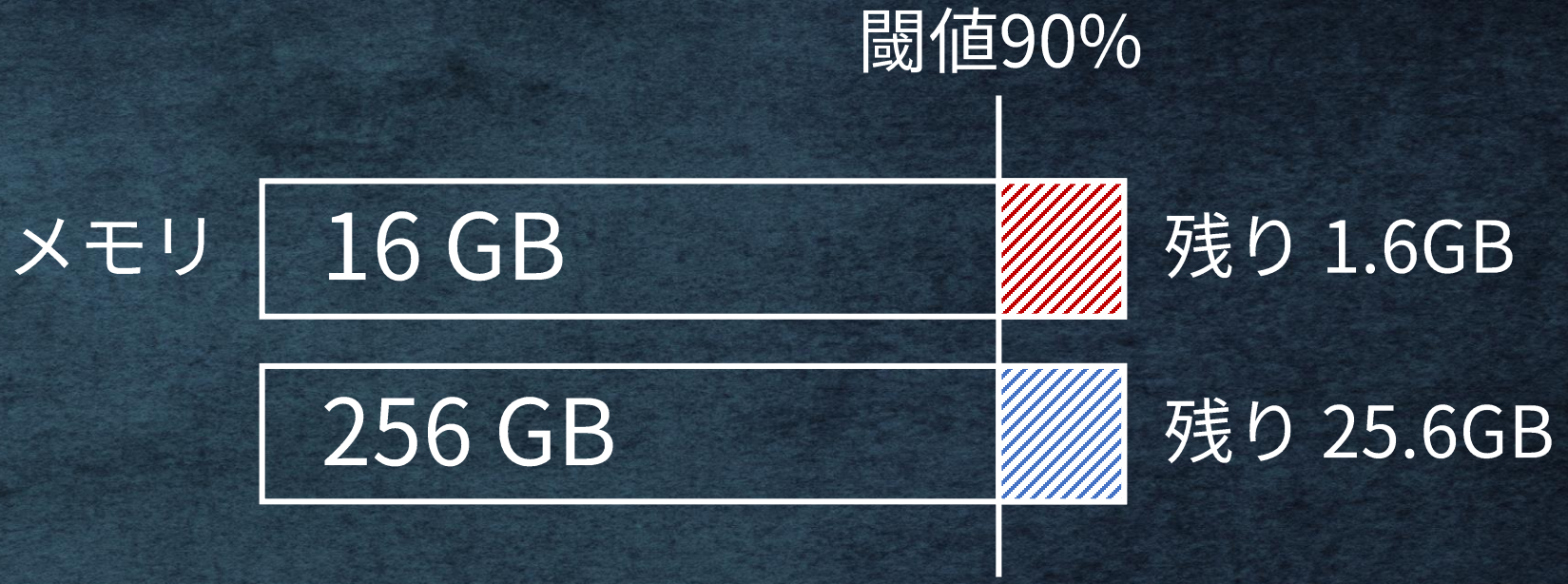


背景

従来と比較して、１台の機器上で情報収集すべき箇所が増えている



背景



異種混在環境で、一律の閾値は適さないこともある

背景

収集項目増加 → 監視項目の選択が大変

異種混在環境 → 閾値の設定が大変

ここをなんとかしたい

今回の概要

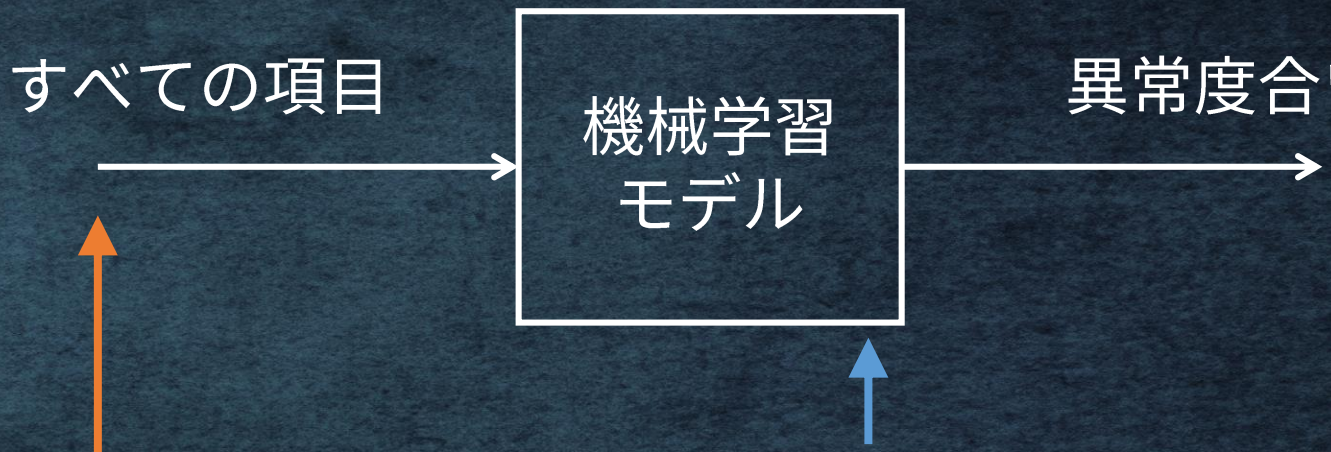
新しい監視のアプローチとして、

短い時間間隔かつスケールするTelemetryのデータを使って、

＋

多様な項目とその重要性をディープラーニングで自動的に学習

イメージ



自動的に学習するので、チューニング不要

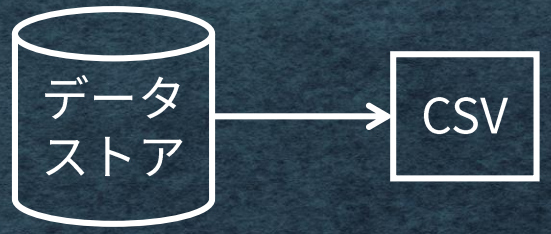
すべての項目を使うので
これまでサイレントだった障害も検知できる

すべての項目を機械学習させると、相関的な異常も検知できる

どのようにすれば実現できるか？

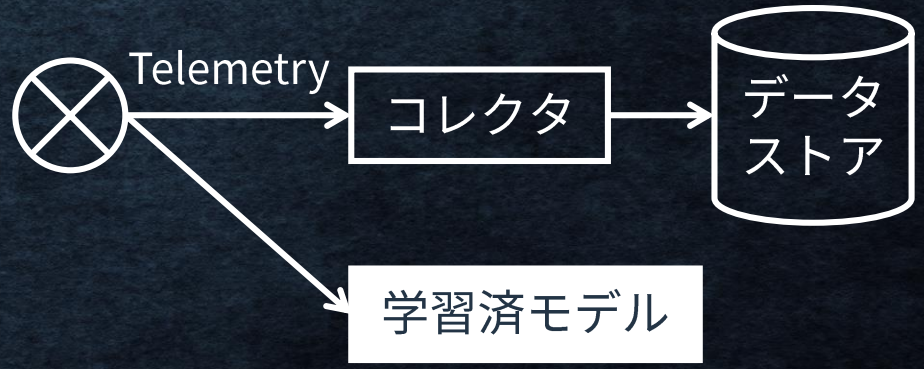
①データを集める

②データを抽出する



③学習する

④検知する



どのようにすれば実現できるか？

学習用のデータ

正常時のデータのみ

異常時のデータはないか、ほとんどないため

機械学習モデル

ニューラルネットワーク

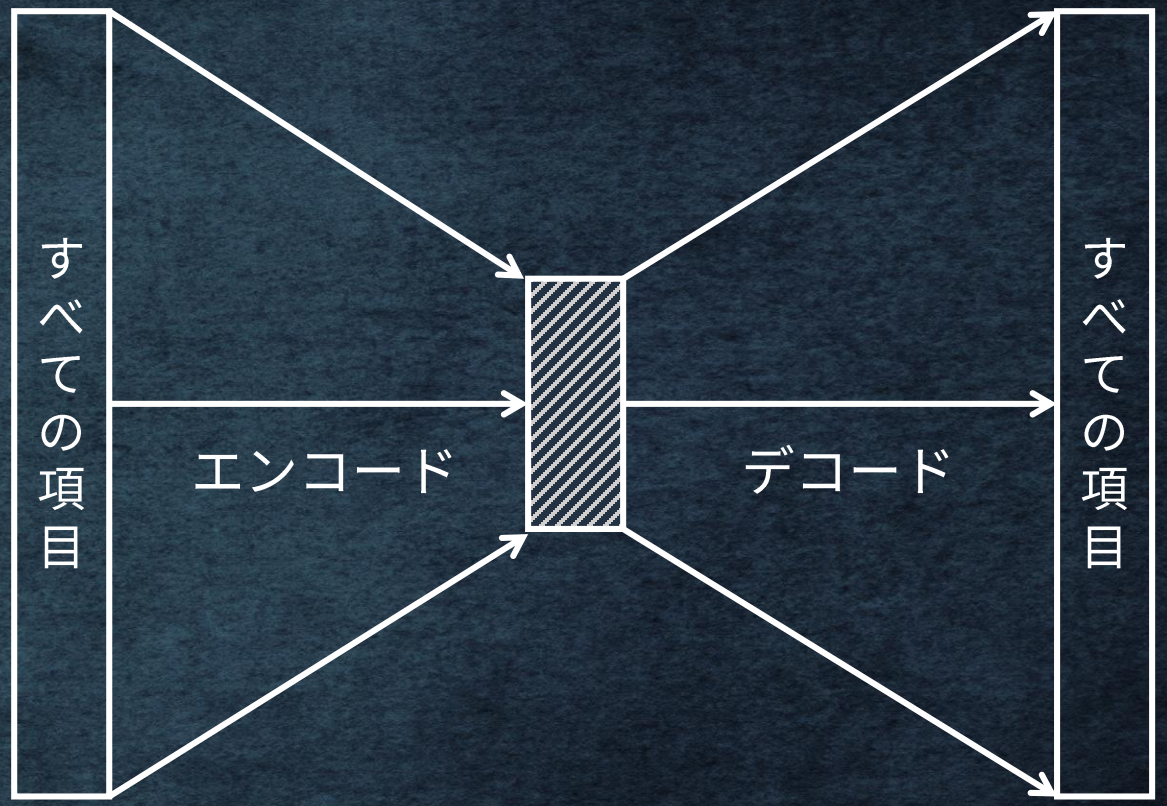
多入力多出力回帰

今回の異常検知モデル

一般的なモデリング

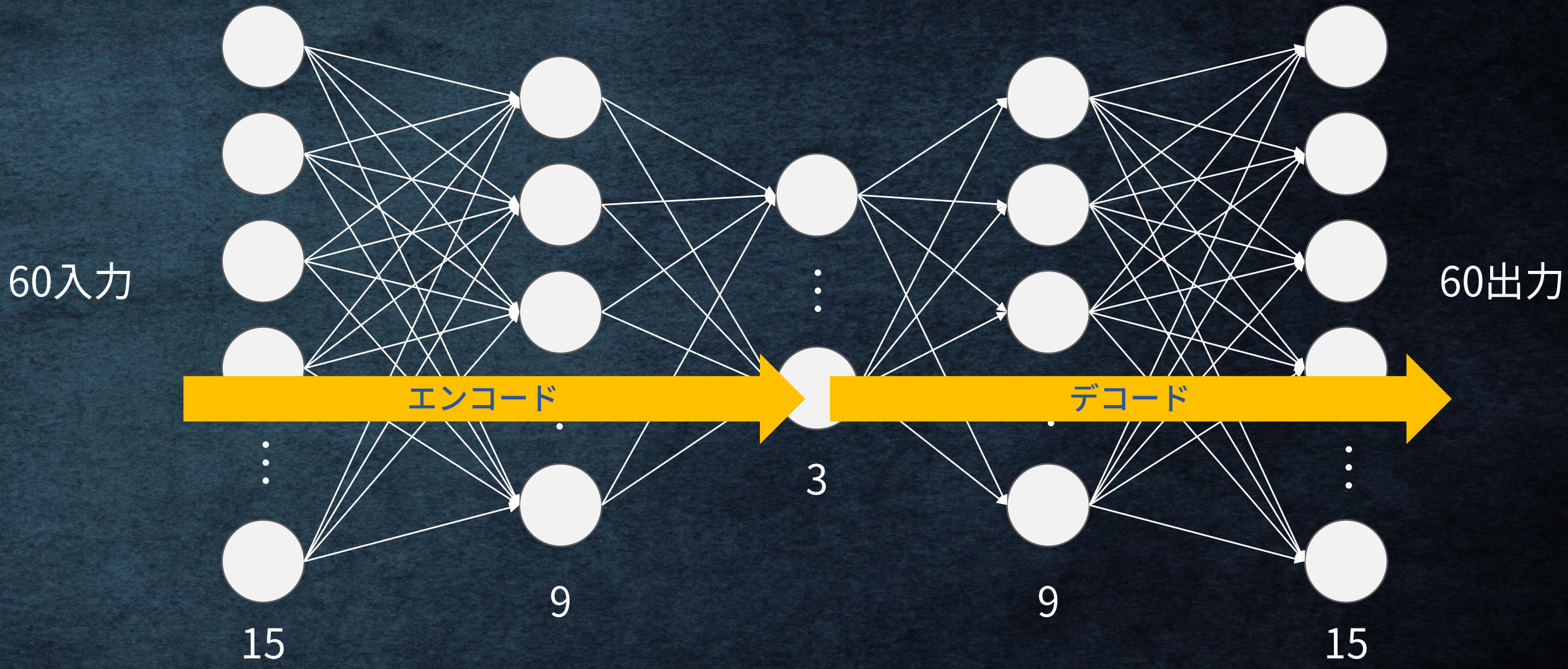
今回のモデリング

ディープオートエンコーダー



異常時のデータは復元誤差が大きくなる

今回のディープオートエンコーダー



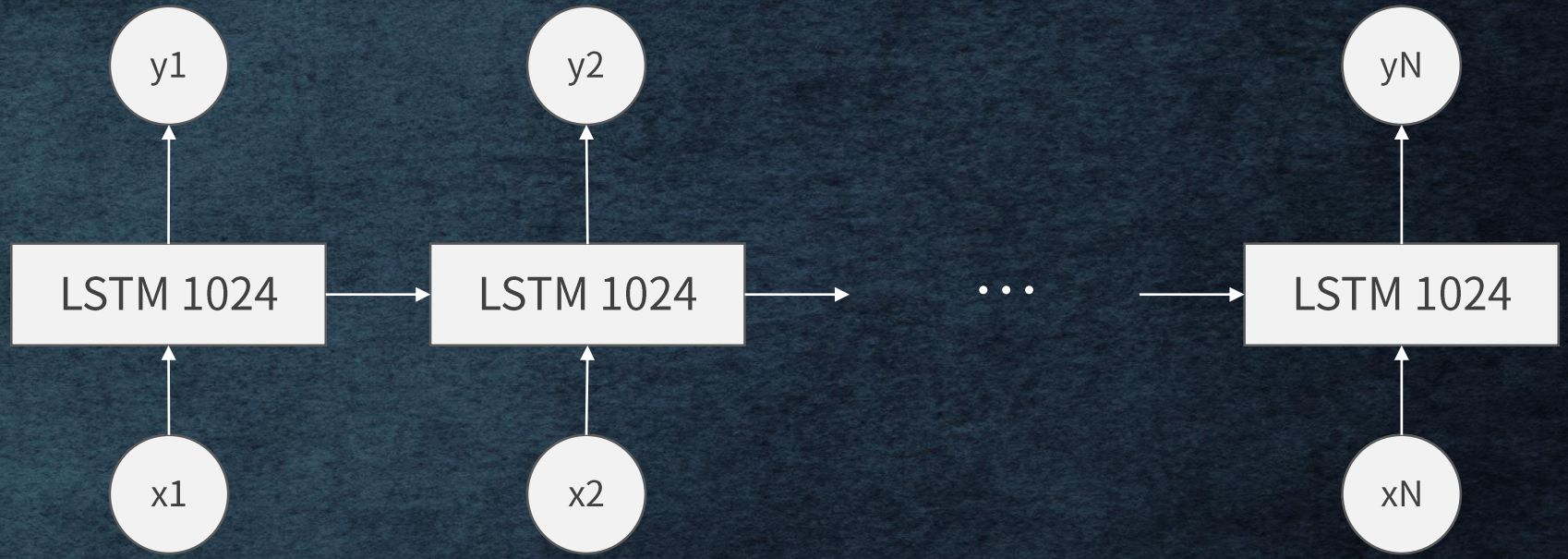
リカレントニューラルネットワーク

すべての項目　⇒　そのときのシステムの状態

正常状態の時系列変化を学習する

予測誤差が大きい場合に異常と判断する

今回のリカレントニューラルネットワーク



畳み込みニューラルネットワーク

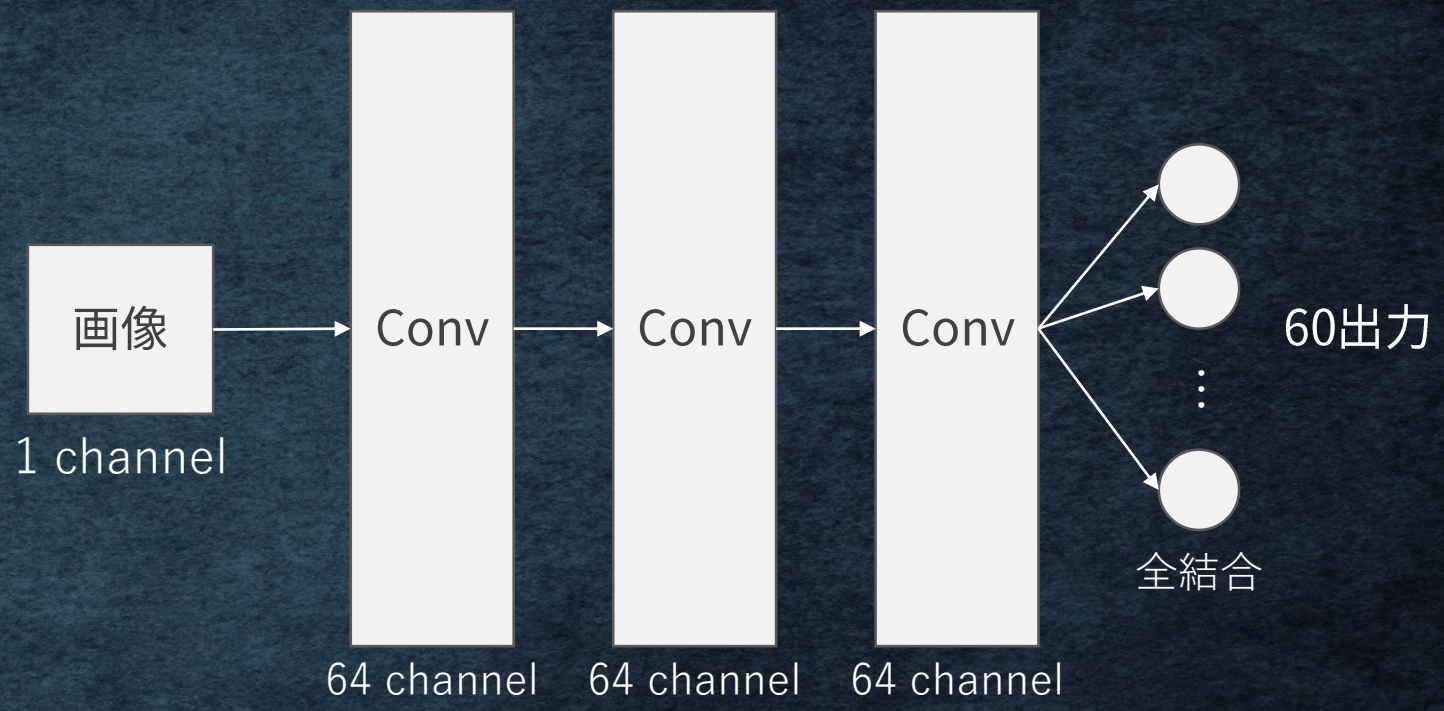
ある瞬間の状態はただの数値の列

65001	0	1	12	7	1

65001	0	1	12	7	2
65001	1	1	10	7	1
65001	0	1	10	7	2
65001	1	1	12	7	1
65001	0	1	10	1	1
65001	0	1	10	1	2
65001	2	1	12	1	1
65001	0	1	12	1	2

ある区間の状態は画像

今回の畳み込みニューラルネットワーク



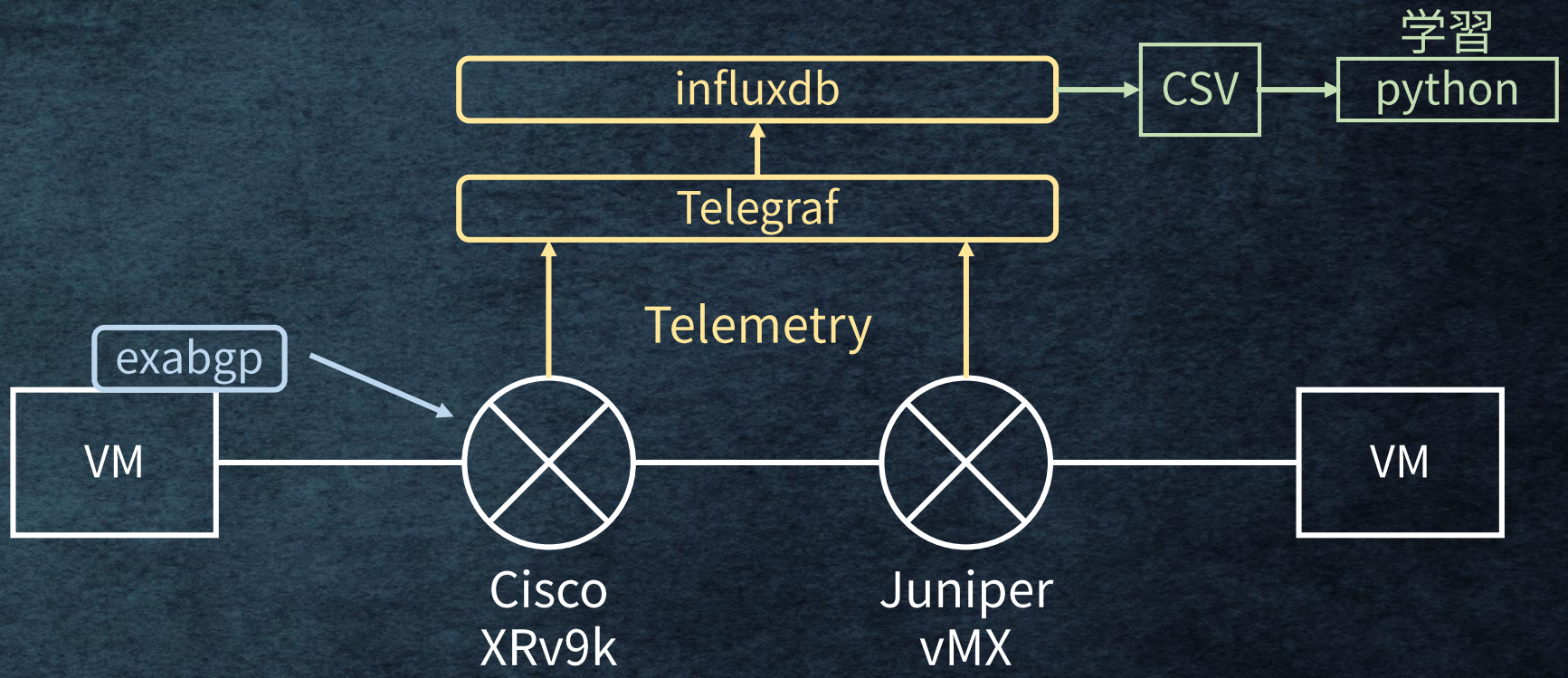
畳み込みニューラルネットワーク

ある区間の状態から次の状態への変化を学習する

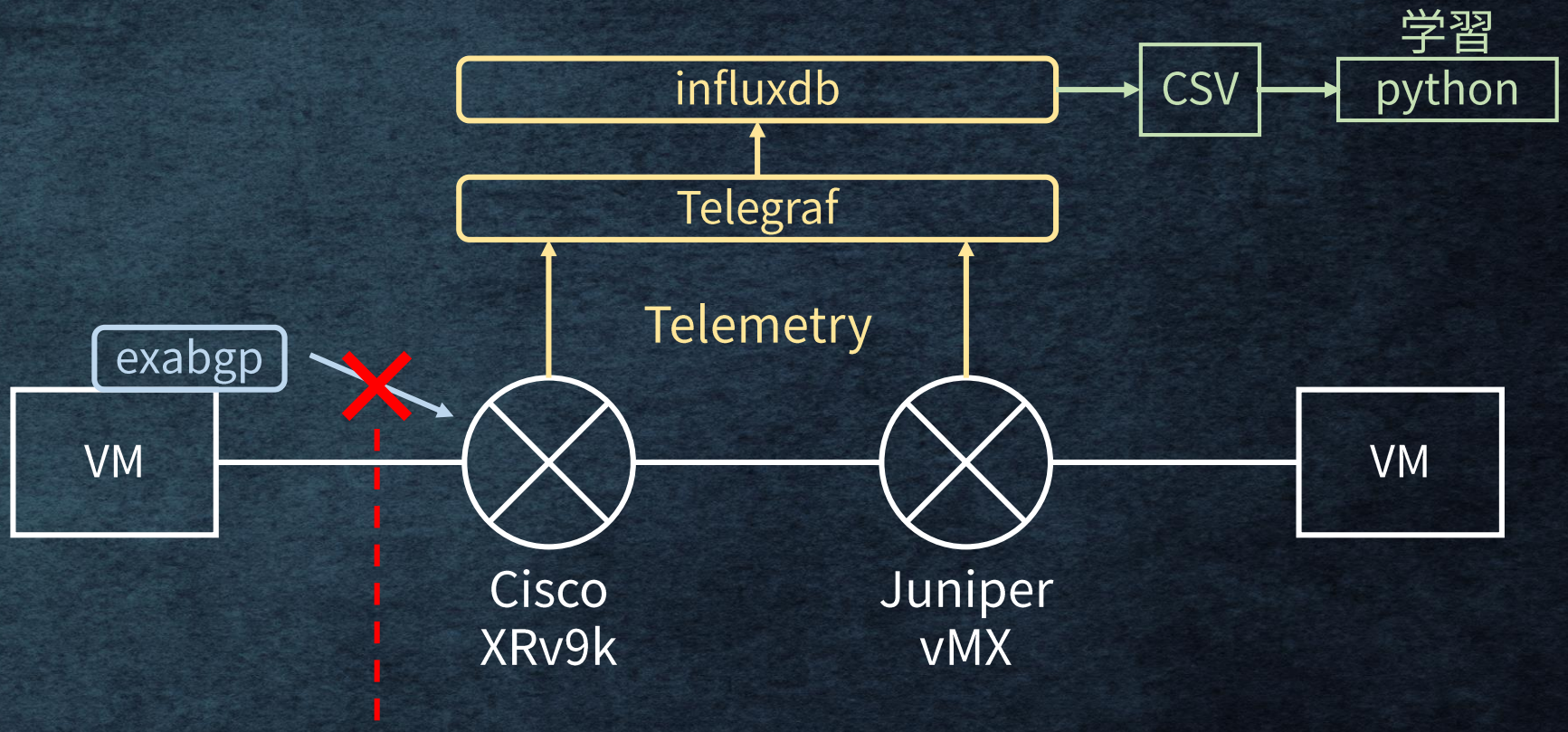
過去の一定時間の傾向からの予測誤差で異常検知する

デモ環境

デモ環境

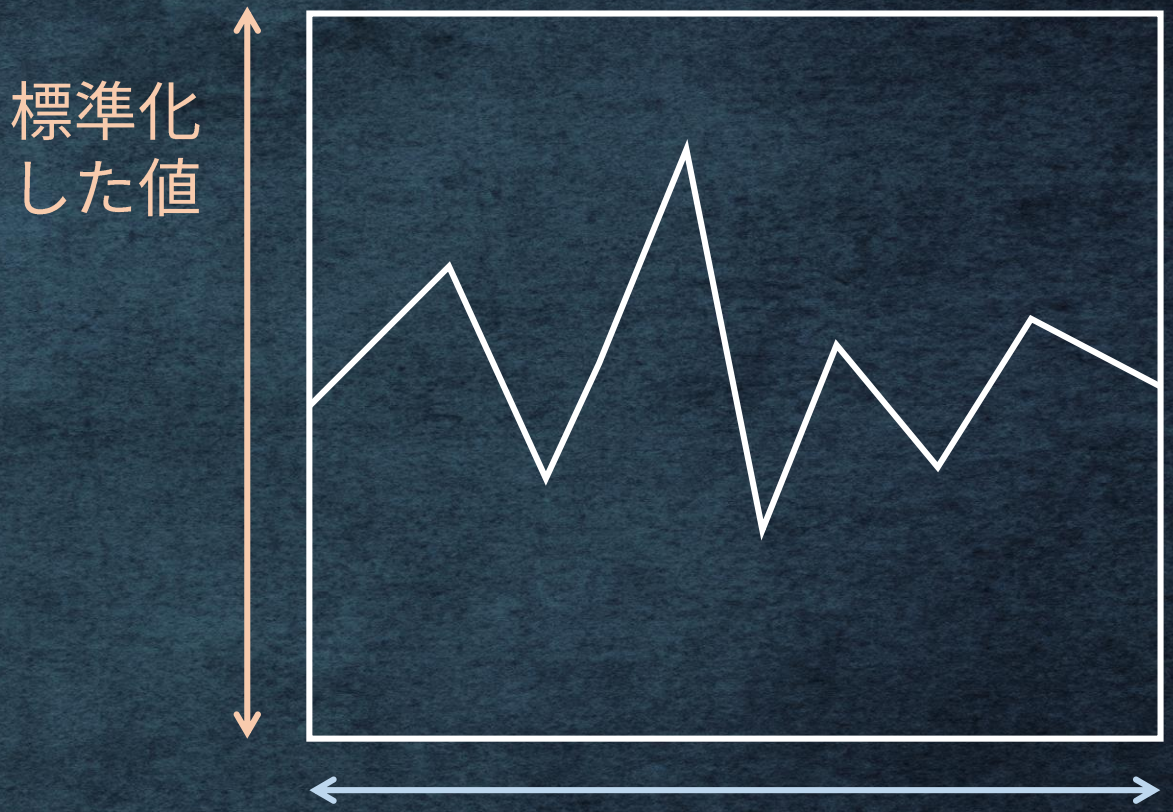

学習
influxdb
CSV
python
Telegraf
Telemetry
exabgp
VM
VM
Cisco
XRv9k
Juniper
vMX

デモ内容



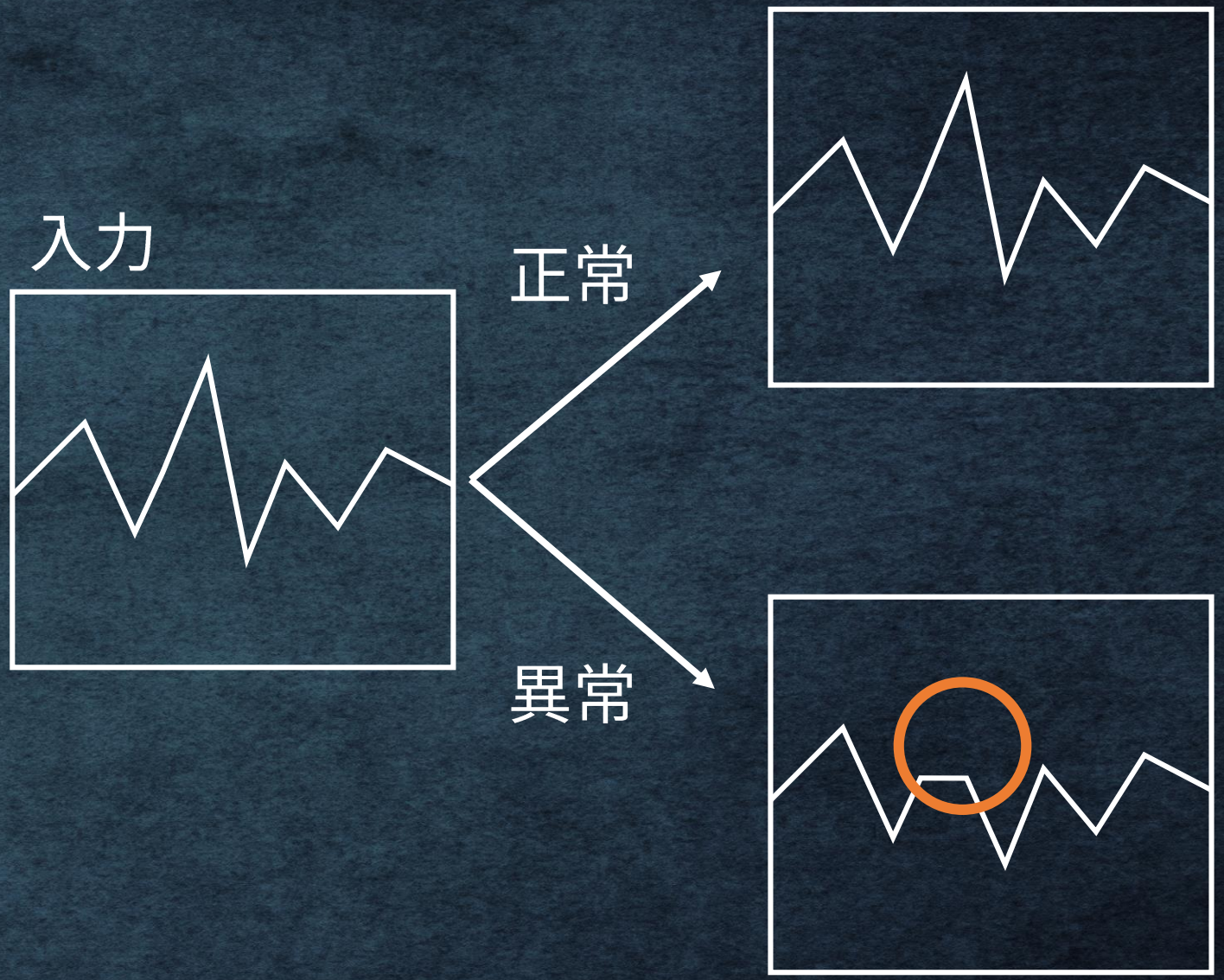
経路数が変化する時の異常度合い

デモ内容

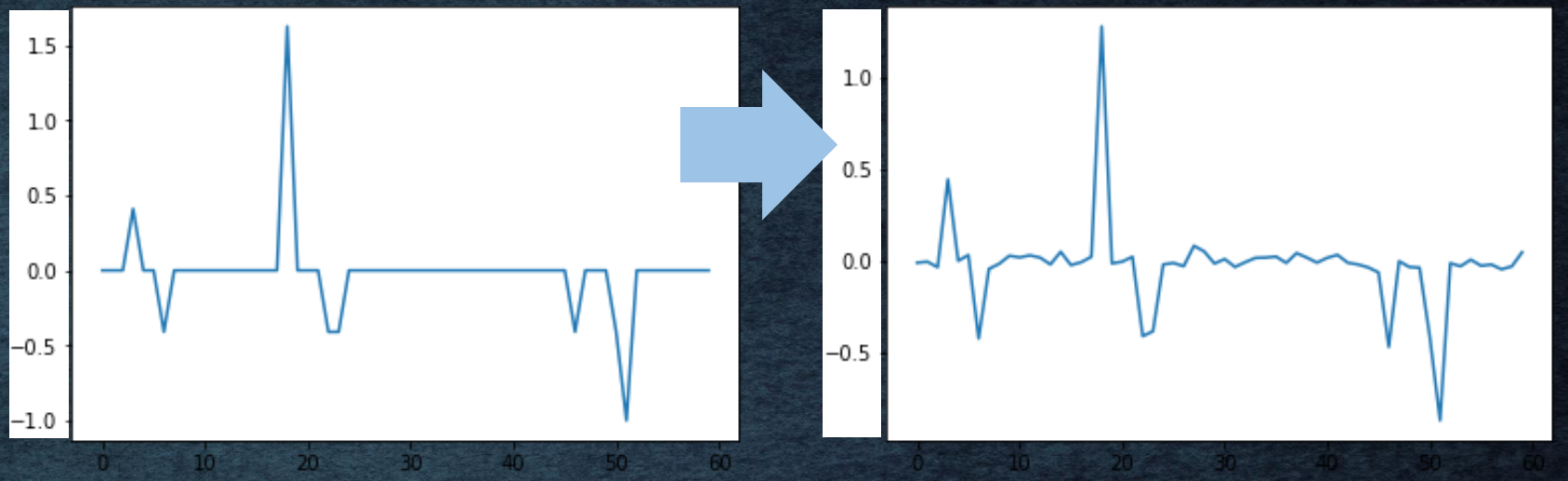


ある瞬間のシステムの
状態をなんとなく可視化

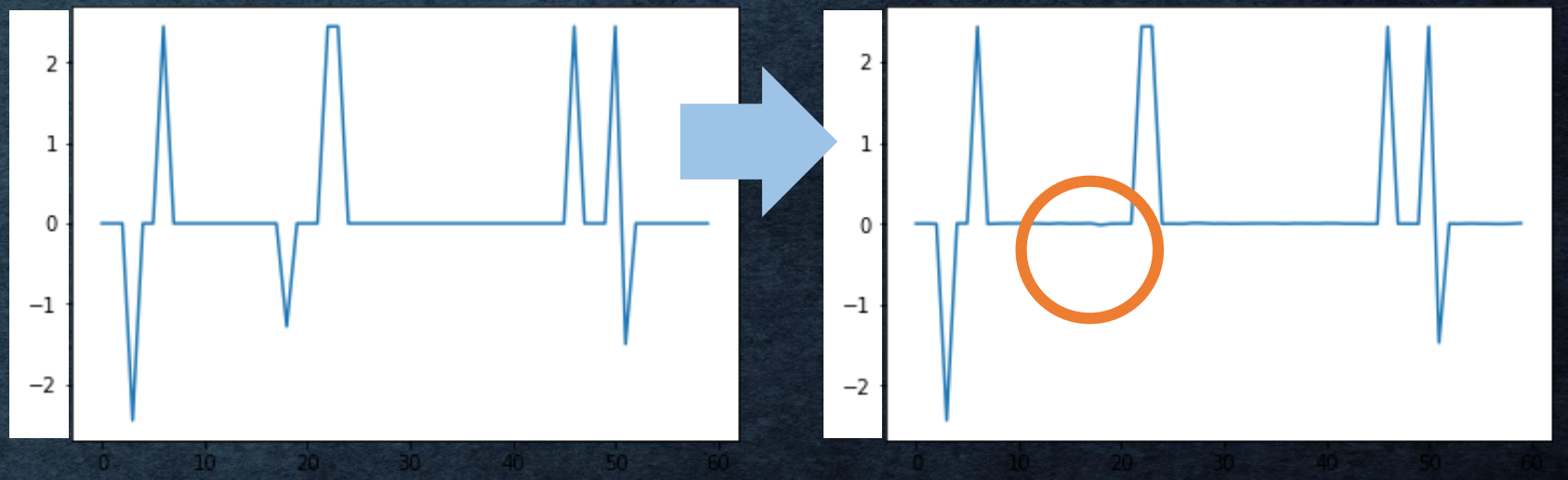
デモ内容



デモ補足


1.5
1.0
0.5
0.0
−0.5
−1.0
0
10
20
30
40
50
60
1.0
0.5
0.0
−0.5
0
10
20
30
40
50
60

0.11


2
1
0
−1
−2
0
10
20
30
40
50
60
2
1
0
−1
−2
0
10
20
30
40
50
60

0.17

まとめ

監視項目の選択が大変 → すべての項目を活用

閾値の設定が大変 → 自動で学習

今後の課題

- 精度
  ※100%にはならない
- 異常分類
  ※異常のナレッジ積み上げが必要

質問・議論

ハイパーパラメータはどうやって決める？

学習に**GPU**は必要？

SNMPでも使える？

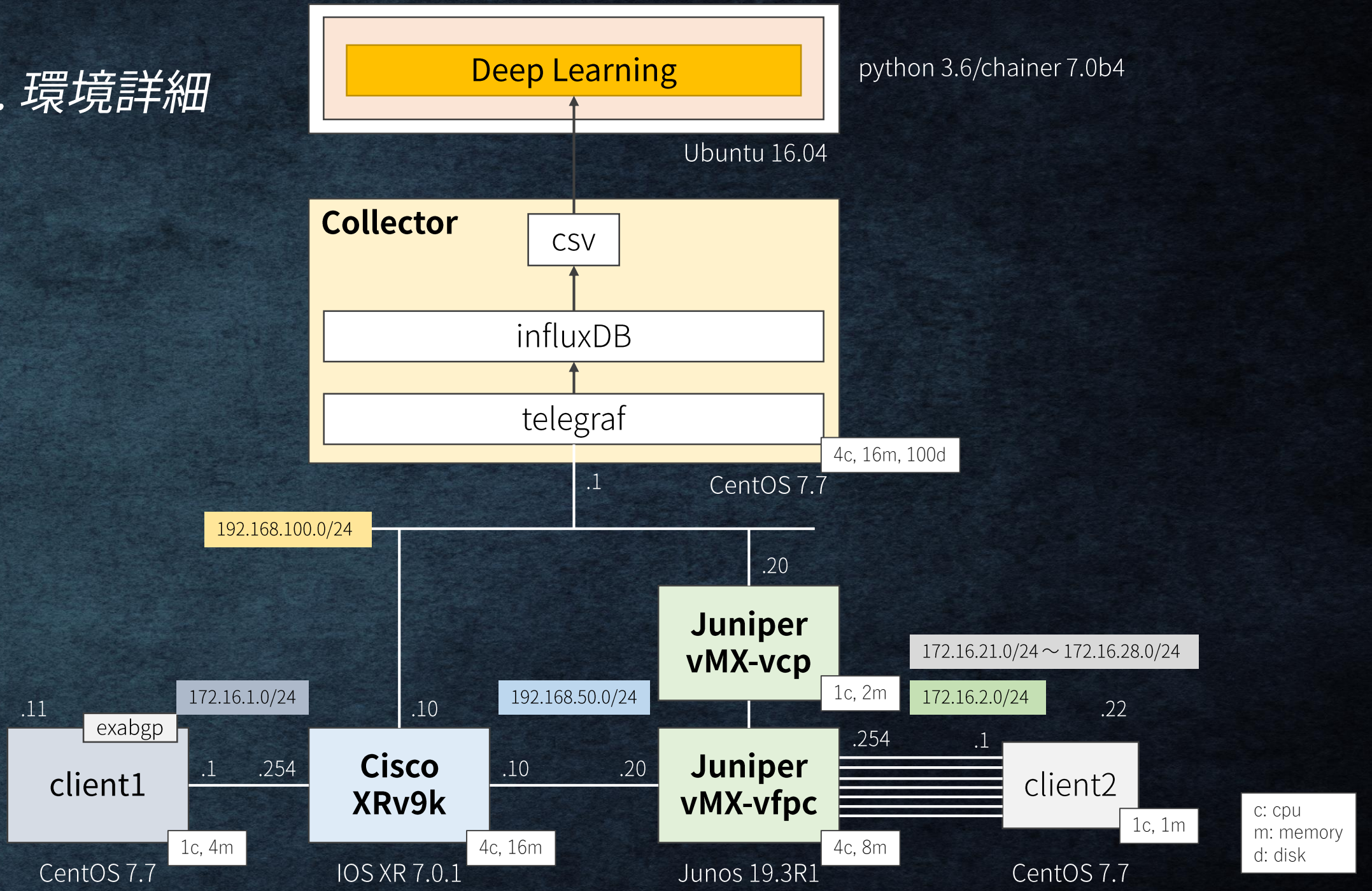
やってみたいがどうやってはじめればいい？

どのくらいの**性能**があれば使える？

最終的な**閾値**はどう決める？

学習データはどれくらい用意すればいい？

Appendix. 環境詳細



Appendix. ディープオートエンコーダー

```
import chainer
import chainer.links as L
import chainer.functions as F

class DAE(chainer.Chain):

    def __init__(self, n_input=60, n_mid1=15, n_mid2=9, n_mid3=3):
        initializer = chainer.initializers.HeNormal()
        super().__init__()
        with self.init_scope():
            self.fc1 = L.Linear(None, n_mid1, initialW=initializer)
            self.fc2 = L.Linear(None, n_mid2, initialW=initializer)
            self.fc3 = L.Linear(None, n_mid3, initialW=initializer)
            self.fc4 = L.Linear(None, n_mid2, initialW=initializer)
            self.fc5 = L.Linear(None, n_mid1, initialW=initializer)
            self.fcZ = L.Linear(None, n_input)

    def __call__(self, x):
        h = F.relu(self.fc1(x))
        h = F.relu(self.fc2(h))
        h = F.relu(self.fc3(h))
        h = F.relu(self.fc4(h))
        h = F.relu(self.fc5(h))
        h = self.fcZ(h)
        return h
```

Appendix. リカレントニューラルネットワーク

```
import chainer
import chainer.links as L
import chainer.functions as F

class LSTM(chainer.Chain):

    def __init__(self, n_input=60):
        initializer = chainer.initializers.GlorotNormal()
        super().__init__()
        with self.init_scope():
            self.l1  = L.LSTM(None, 1024)
            self.fcZ = L.Linear(None, n_input, initialW=initializer)

    def __call__(self, x):
        self.reset_state()
        h = self.l1(x)
        h = self.fcZ(h)
        return h

    def reset_state(self):
        self.l1.reset_state()
```

Appendix. 畳み込みニューラルネットワーク

```
import chainer
import chainer.links as L
import chainer.functions as F

class Block(chainer.Chain):

    def __init__(self, n_out, ksize, pad=1):
        super().__init__()
        with self.init_scope():
            self.conv = L.Convolution2D(None, n_out, ksize, pad=pad, nobias=True)
            self.BN   = L.BatchNormalization(n_out)

    def __call__(self, x):
        h = self.conv(x)
        h = self.BN(h)
        return F.relu(h)
```

```
class CNN(chainer.Chain):

    def __init__(self, n_out=60):
        super().__init__()
        with self.init_scope():
            self.block1_1 = Block(64, 6)
            self.block1_2 = Block(64, 4)
            self.block1_3 = Block(64, 2)
            self.fcZ   = L.Linear(None, n_out, nobias=True)

            self.BRN   = L.BatchRenormalization(1)

    def __call__(self, x):
        h = self.BRN(x)
        h = self.block1_1(h)
        h = F.average_pooling_2d(h, ksize=4, stride=4)
        h = self.block1_2(h)
        h = F.average_pooling_2d(h, ksize=2, stride=2)
        h = self.block1_3(h)
        return self.fcZ(h)
```