機械学習と自動微分

2022年1月14日(金)：最適化法 第14回
4講時 14:55 - 16:25

たきがわ　いちがく
瀧川 一学

https://itakigawa.github.io/

理化学研究所 革新知能統合研究センター

自己紹介：瀧川 一学 (たきがわ いちがく)

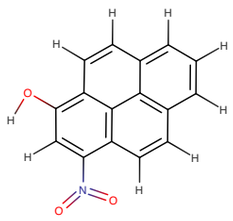
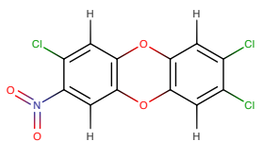
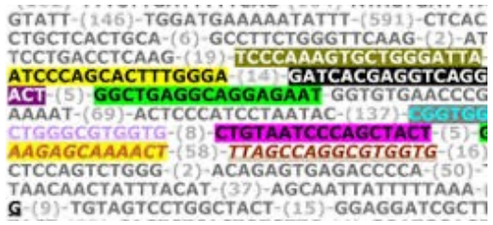
研究テーマ：
機械学習・データマイニングの技術研究 + その**自然科学**での利活用研究

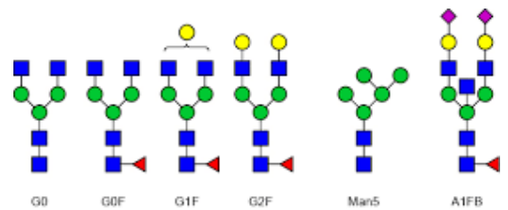
10年 北大	**統計的信号処理とパターン認識 (工学研究科)**
(1995〜2004)	"劣決定信号源分離のL1ノルム最小解の理論分析"
7年 京大	**バイオインフォマティクス (化学研究所)**
(2005〜2011)	**ケモインフォマティクス (薬学研究科)**
7年 北大	**離散構造を伴う機械学習・データ駆動科学**
(2012〜2018)	**(情報科学研究科)**
	+ JSTさきがけ: 材料インフォマティクス
？年 理研(京都)	**AIPセンター iPS細胞連携医学的リスク回避チーム**
(2019〜)	**(北大 化学反応創成研究拠点とクロスアポイント)**

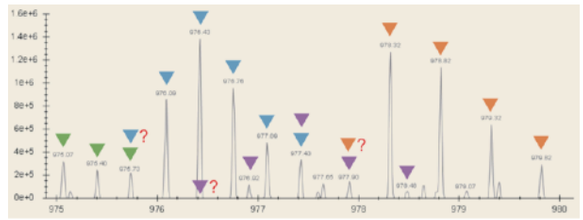
技術的には「離散構造」を伴う機械学習に強い関心あり

- **対象自体が「離散構造」を持つ**

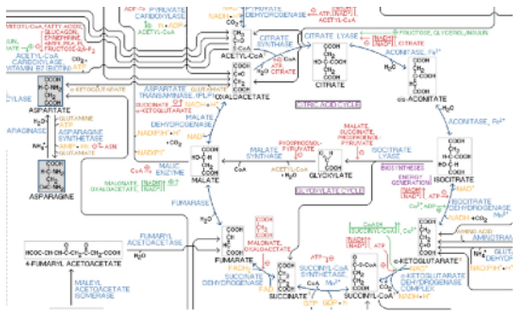
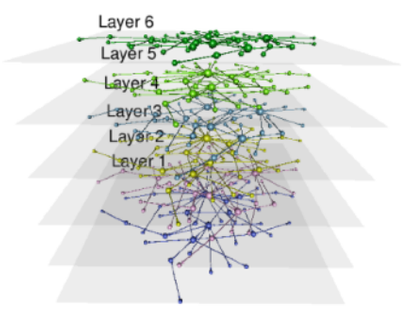
  集合と論理、関係、組合せ、系列・文字列、木、グラフ、代数系、言語、etc

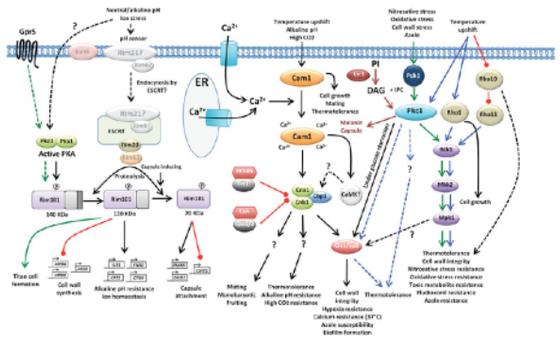
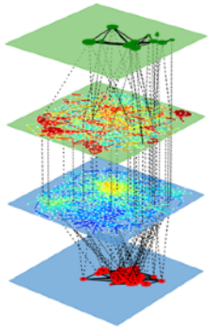








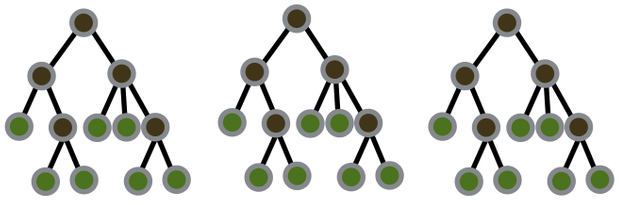
- **対象と対象の間に「離散構造」を持つ**

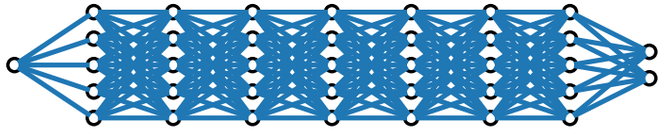



- **モデルが「離散構造」を持つ**

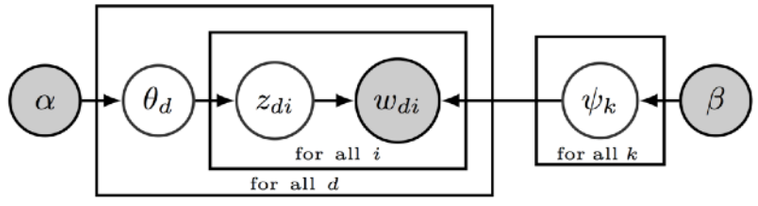
決定木・決定DAG



ニューラルネット



確率的プログラミング



本日の内容

1. 機械学習 = 新しいプログラミング
2. 勾配と勾配降下法
3. 合成関数の自動微分
4. Pythonで作って理解しよう
5. Q & A

このスライドは下記にあとで置いておきます

https://itakigawa.github.io/news.html

1

機械学習 = 新しいプログラミング

- 機械学習とは？
- どういうときに使う？
- 機械学習が「新しいプログラミング」とは？

プログラミング = コンピュータプログラムを作ること

入力を受け取って、書き換え、出力する手順

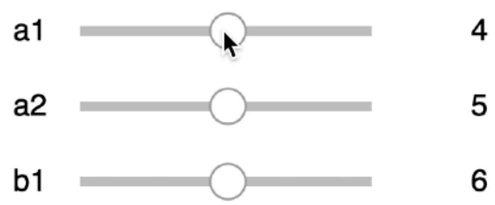
プログラム = 関数

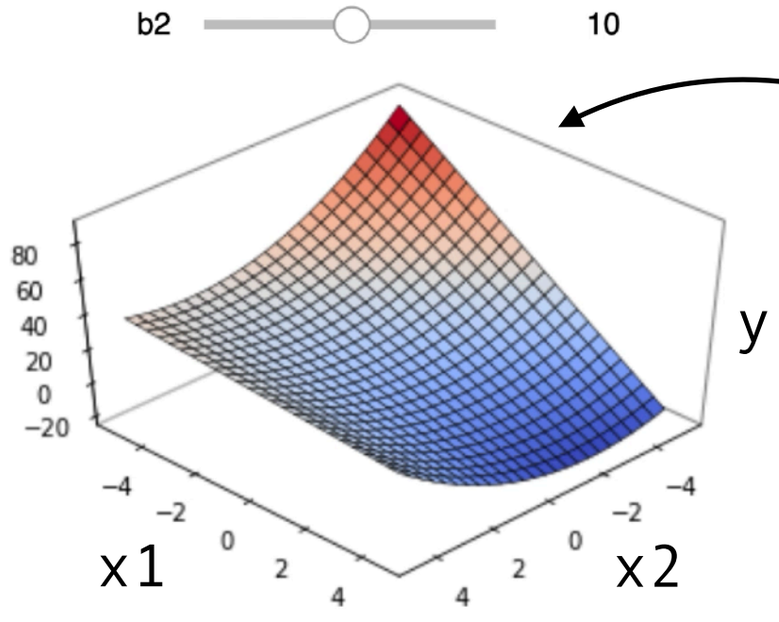
関数 $y = (x_1^2 - a_1 x_1 x_2 + a_2 x_2^2) + (-b_1 x_2 + b_2)$

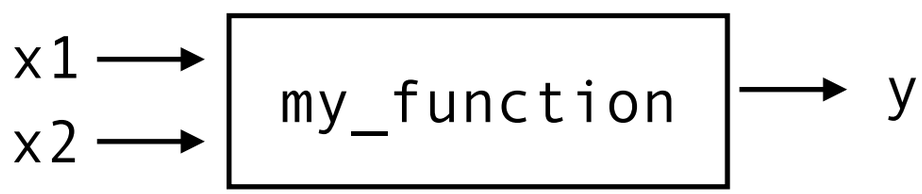
パラメタ変数
(オプション引数)

入力変数　x1
　　x2

出力変数　y







python

```
def my_function(x1, x2, \
                a1=4, a2=5, b1=6, b2=10):
  u = x1*x1 - a1*x1*x2 + a2*x1*x2
  v = -b1*x2 + b2
  y = u + v
  return y
```

```
my_function(1,1)
```

```
6
```

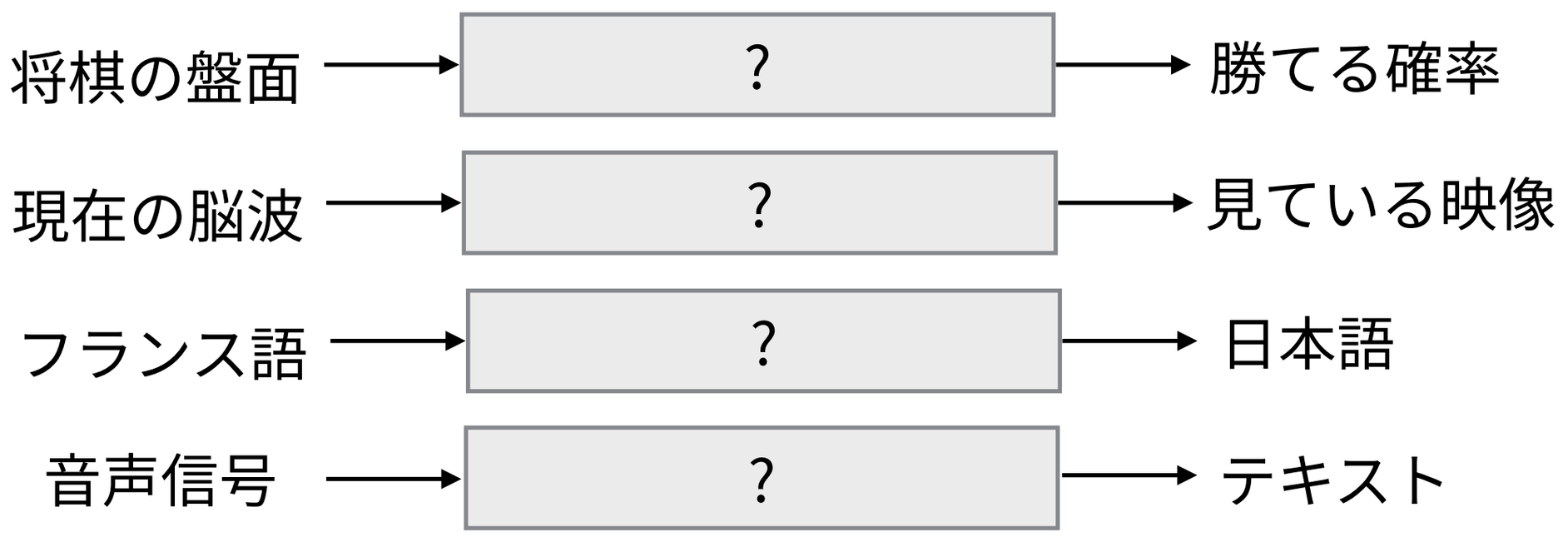
```
my_function(1,1, b1=3)
```

```
9
```

でも、実際のプログラミング(関数の設計)はとても難しい

コンピュータプログラムを作るには入力から出力を得る手順がカンペキにわかってなければいけない。

でも、実際には肝心の手順がよく分からないことが多い



機械学習の出番

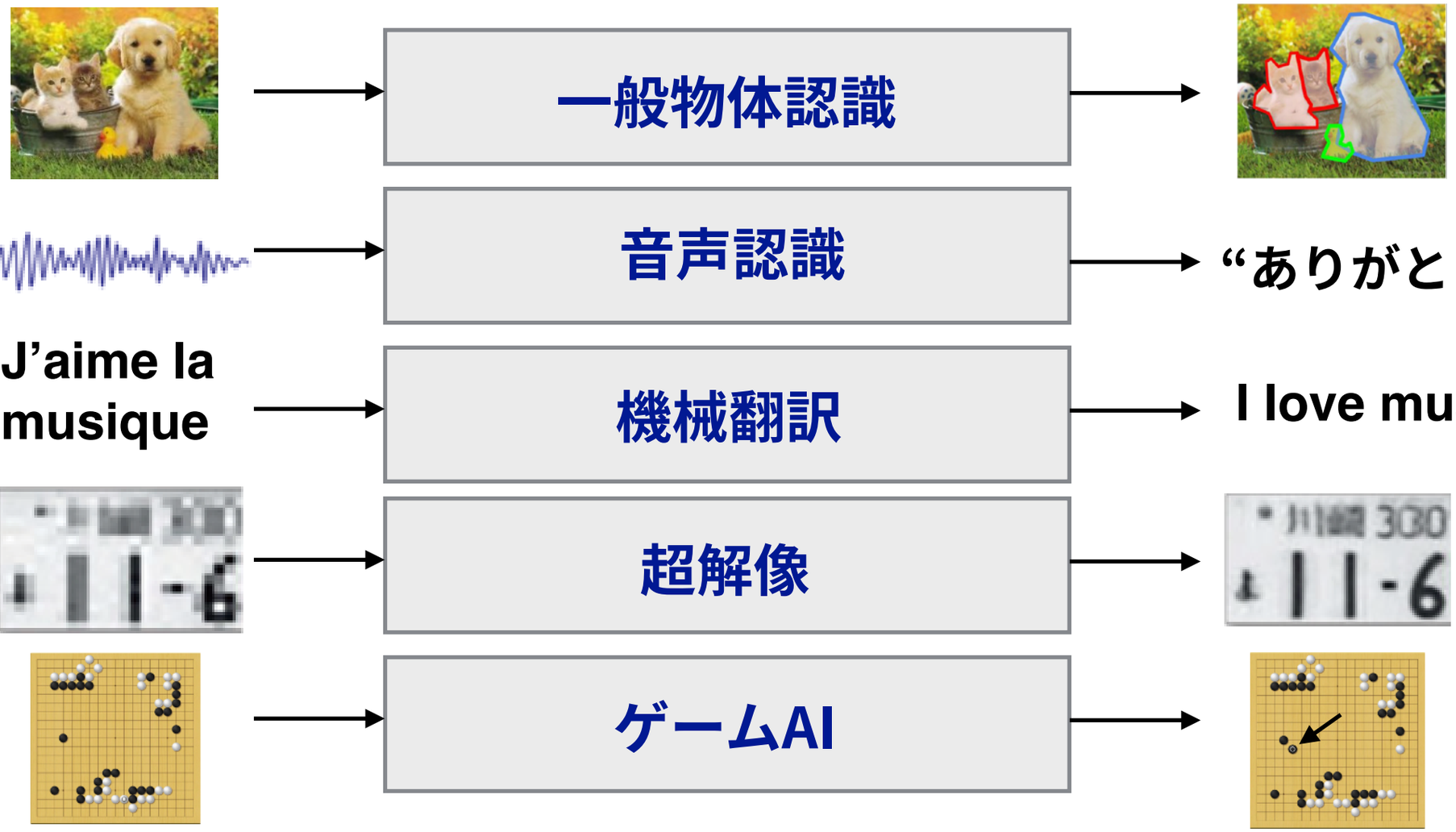
事例：写真をAさんかBさんかに分類するコンピュータプログラムを作りたい。(大量の写真を人手分類するの嫌)

？　？　？　？　？　？　？

- 人間は**簡単にできる**
- が、どうやってやっているのか原理は不明確
- 髪型、角度、照明、背景、表情、化粧、年齢、などを考えると明示的なプログラミングはとても難しい

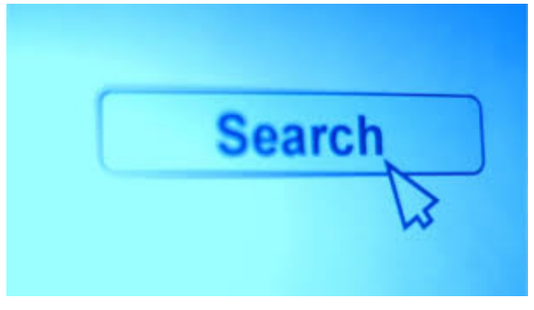
機械学習：新しいプログラミングのパラダイム

入出力の関係がよく分からない変換過程(関数)を大量の入出力の見本例から明示的にプログラミングすることなく構成する技法



機械学習は既にいろんな出口の要に

検索エンジン



機械翻訳

自動運転

医療診断

広告

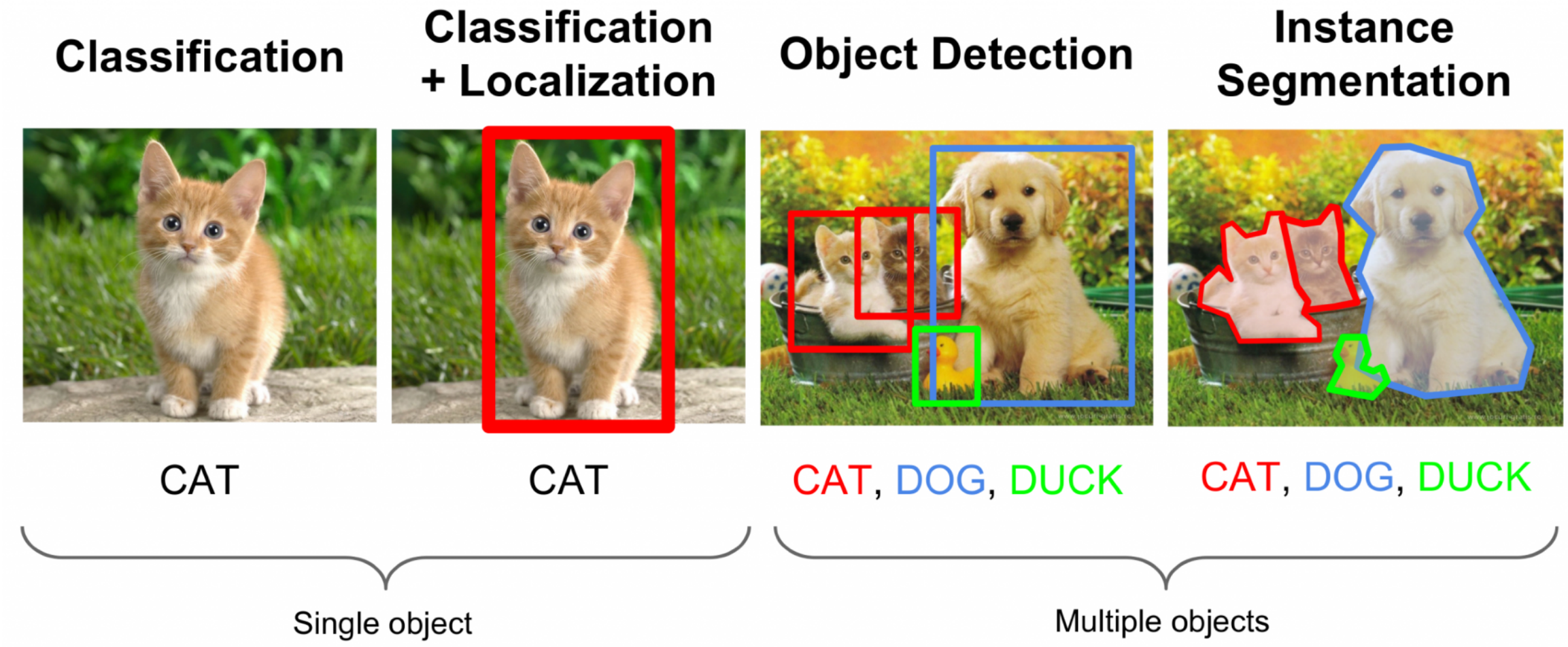
物流・需要予測

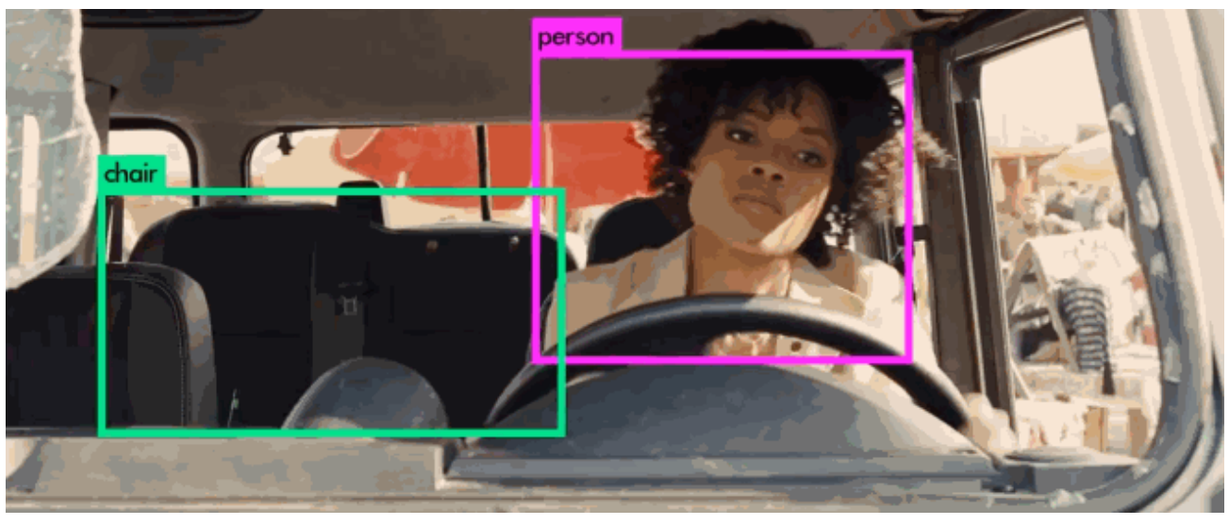
気象予測

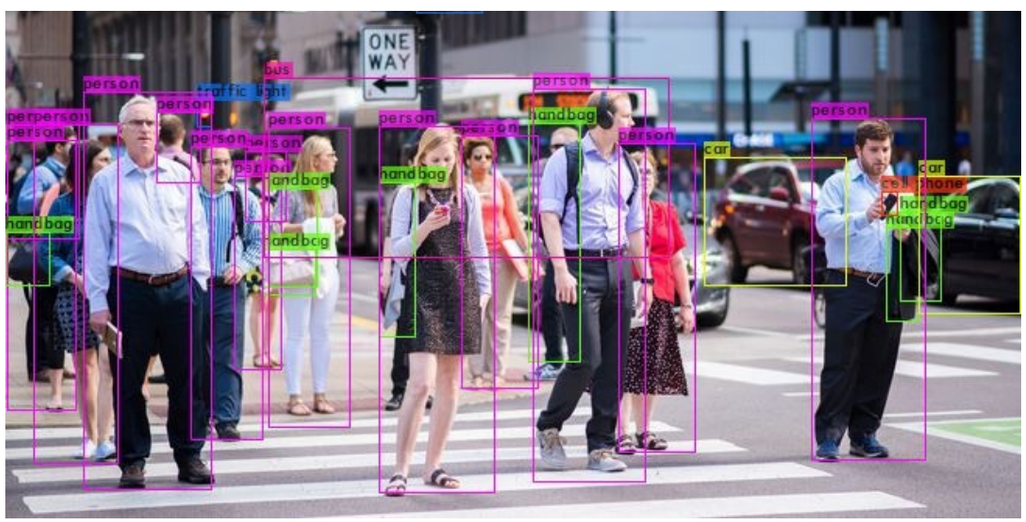
セキュリティ

スマホ、ネット通販、製造業、科学、交通、農業、教育、金融、就職・結婚、…

例：深層ニューラルネットワークによる画像認識







バリエーションでかなり広いタスクが実用レベルで解ける

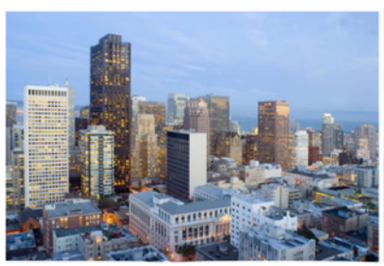
pix2pix

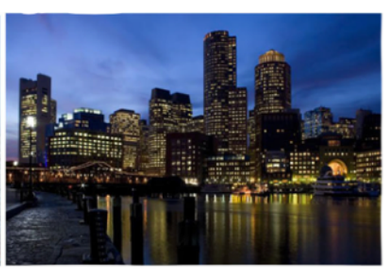
画像生成

OpenPose

姿勢推定

Style Transfer





スタイル転移

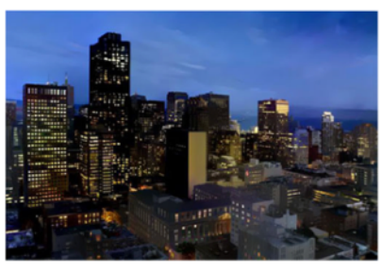
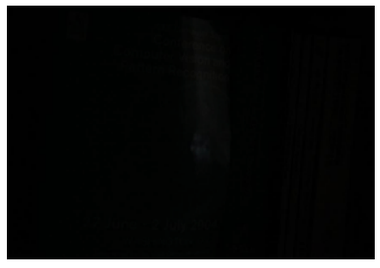
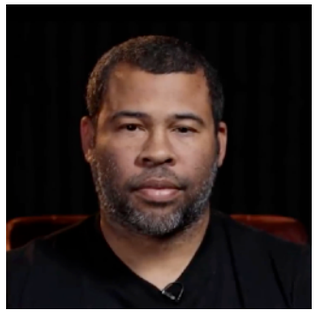


image transformation

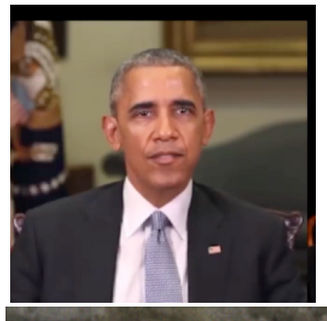


暗所像明瞭化

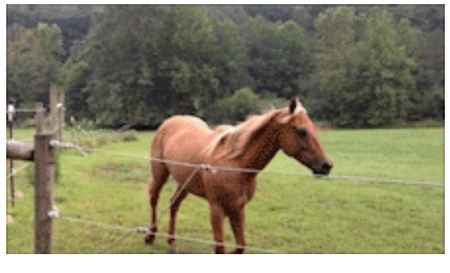
face swapping (“DeepFake”)



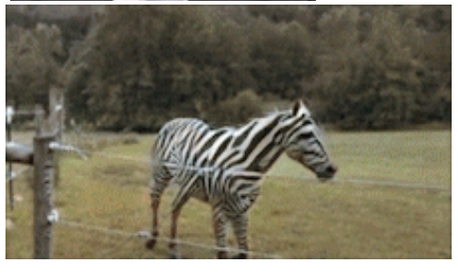
顔部置換



CycleGAN



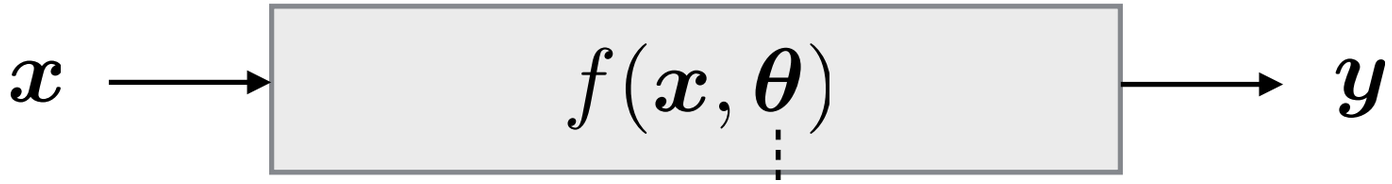
動画生成



機械学習のモデル

機械学習 = **多量の入出力の見本例**から**プログラム**を
自動的に作成する(いいかげんな)技術

機械学習モデル

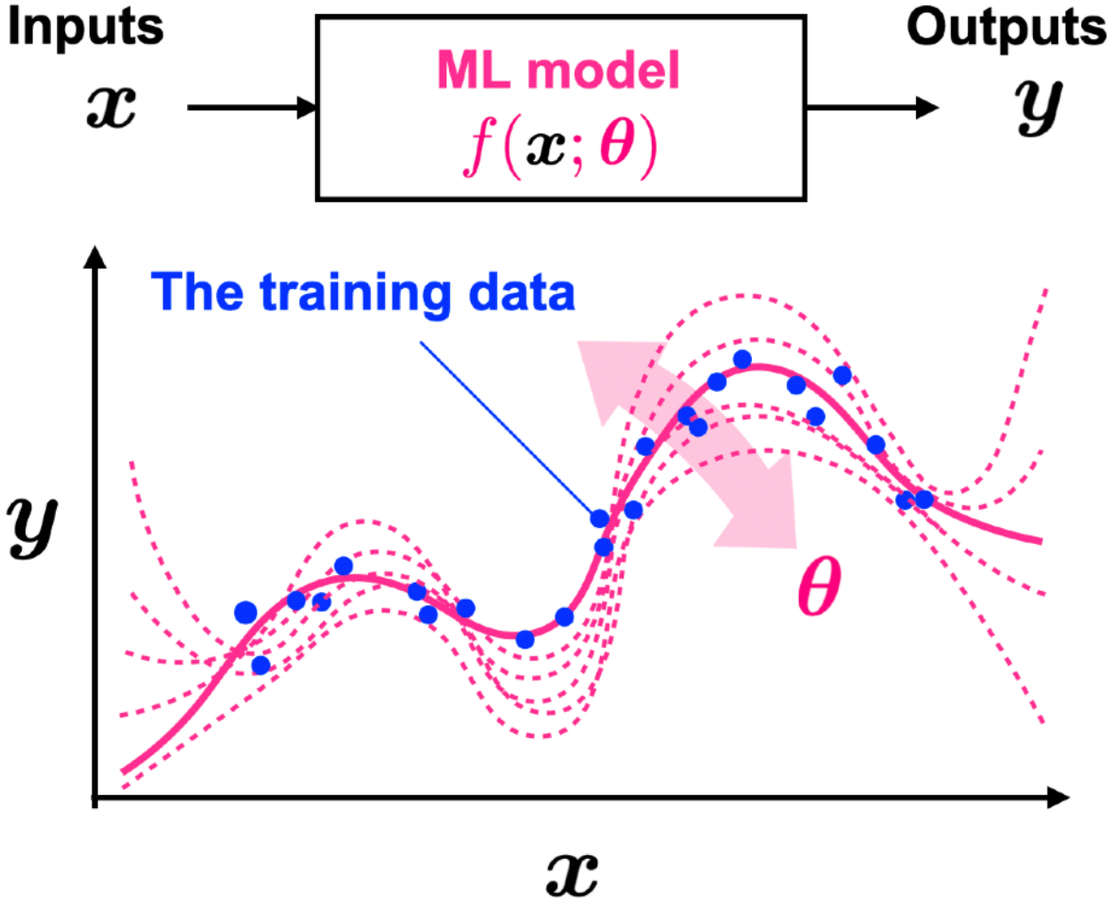


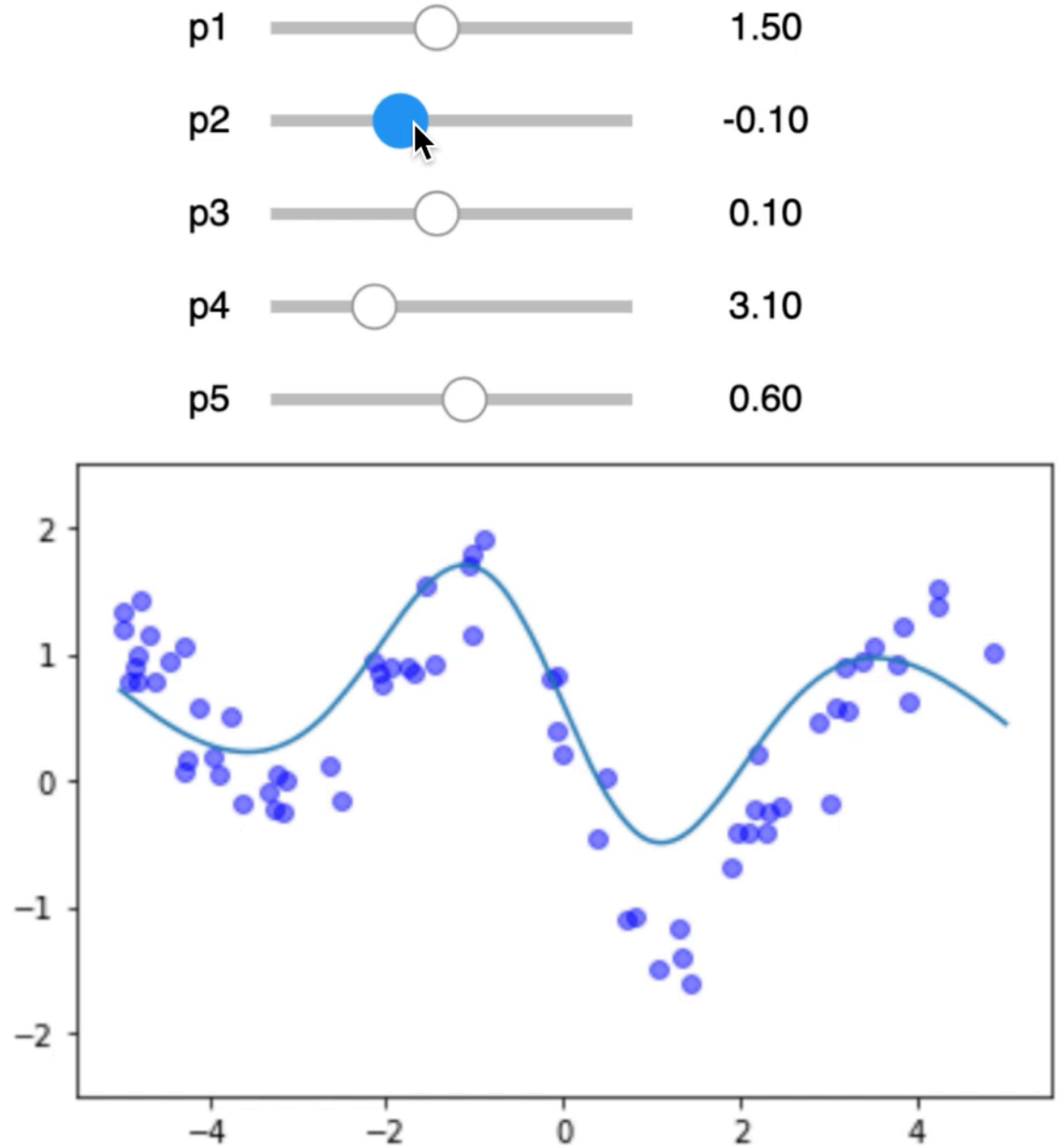
$$\boldsymbol{\theta} = (\theta_1, \theta_2, \ldots, \theta_p)$$

多数のパラメタの値を動かして
出力が望む値になるようにする

機械学習 = (高次元での)曲面のフィッティング

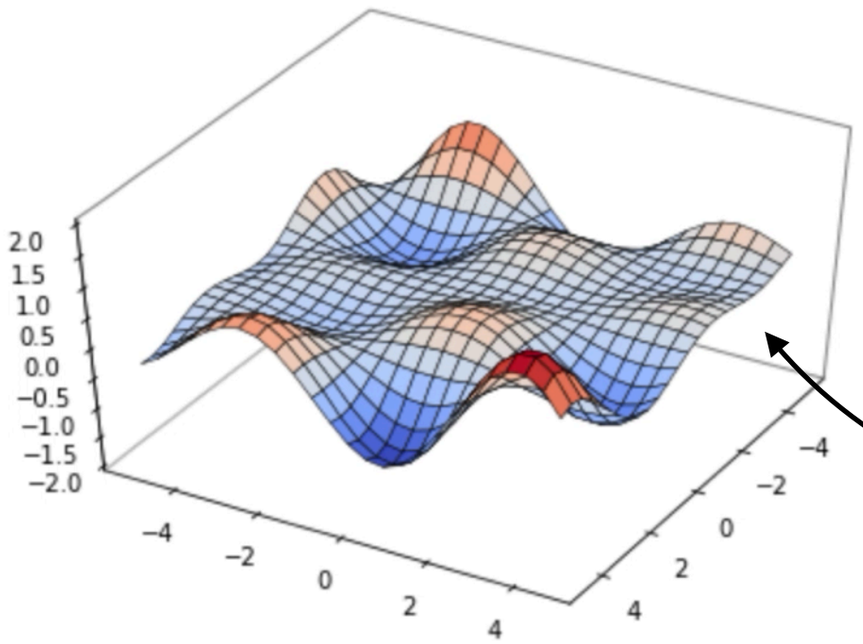
入力: 1次元、出力: 1次元であれば曲線のあてはめ

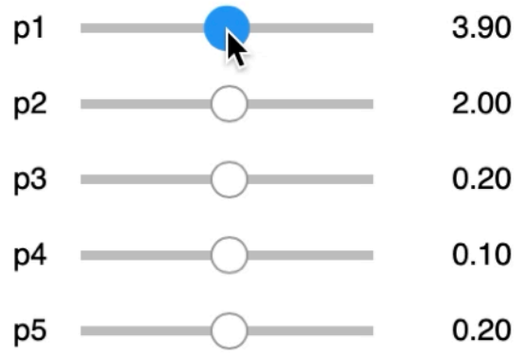




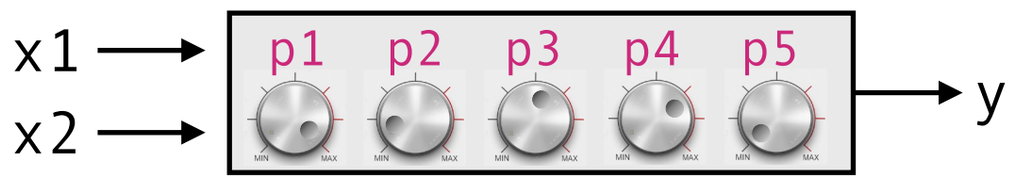
機械学習モデル = 多数のパラメタで形を変えられる曲面

機械学習モデルはパラメタの値設定で形(挙動)を変えられる関数



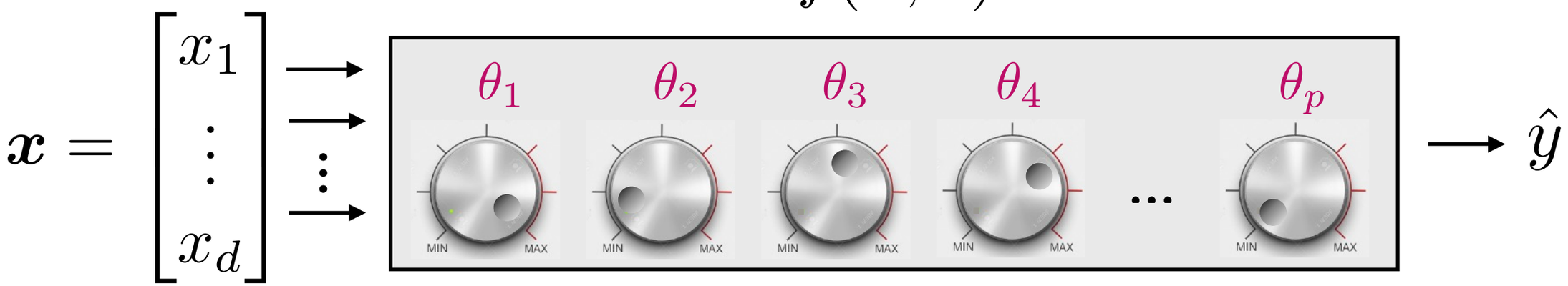


パラメタ



一般の機械学習モデルはパラメタがたくさんある

$$f(\boldsymbol{x};\boldsymbol{\theta})$$



機械学習 = パラメタ値の最適化 (予測が最も当たるように)

最小化問題 (モデルの"学習" or フィッティング)

$$\min_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) + \Omega(\boldsymbol{\theta})$$

見本例の値　モデル出力値

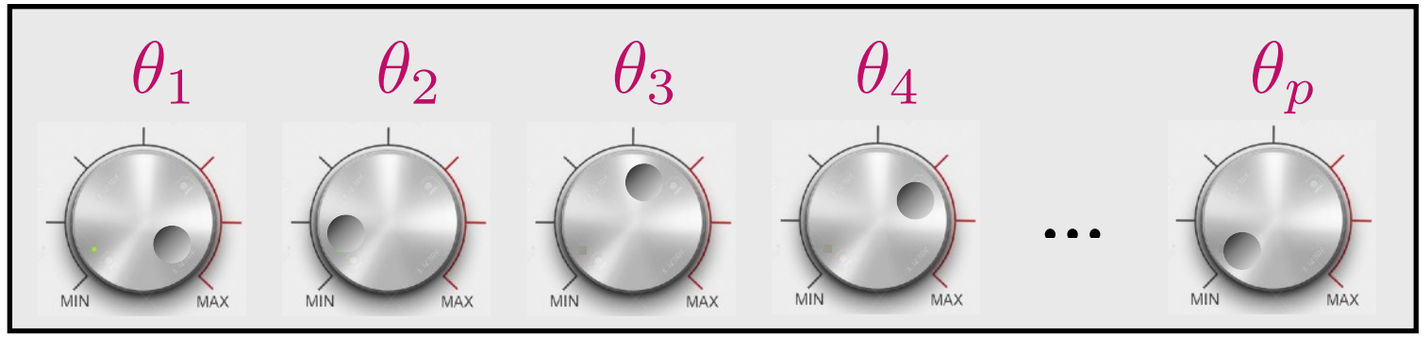
$$\text{where } L(\boldsymbol{\theta}) = \sum_{i=1}^{n} \text{error}(y_i, f(\boldsymbol{x}_i; \boldsymbol{\theta}))$$

$\Omega(\boldsymbol{\theta})$
見本例に依存しない補正項

パラメタ値をいじって出力ができるだけ見本例に合うように！！

$$f(\boldsymbol{x}; \boldsymbol{\theta})$$

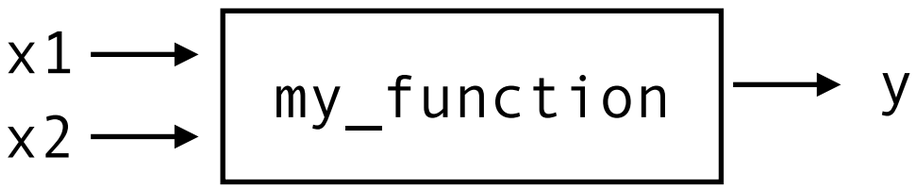
$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix}$$



$\hat{y}$

例

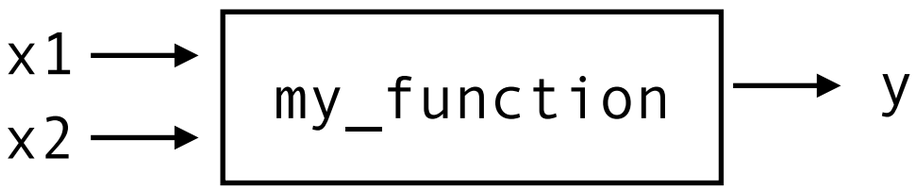
```python
def my_function(x1, x2, \
       a1=4, a2=5, b1=6, b2=10):
  u = x1*x1 - a1*x1*x2 + a2*x1*x2
  v = -b1*x2 + b2
  y = u + v
  return y
```



a1,a2,b1,b2の値を「いい感じに」
動かして出力yが望む値になるように
したい！

今日伝えたいポイント

```python
def my_function(x1, x2, \
       a1=4, a2=5, b1=6, b2=10):
  u = x1*x1 - a1*x1*x2 + a2*x1*x2
  v = -b1*x2 + b2
  y = u + v
  return y
```



現代的な「深層学習」は
「**自動微分**プログラミング」

∂y/∂a1

= a1の値をちょっとだけ変えると
出力yの値はどう変わるか?

この(偏)微分の値の計算を自動で
やってくれる

勾配法という方法と組合わせると**プログラムとして**
書けるものなら何でもパラメタ値の「**最適化**」ができる！

最適化 = 関数の最小値(または最大値)を求める

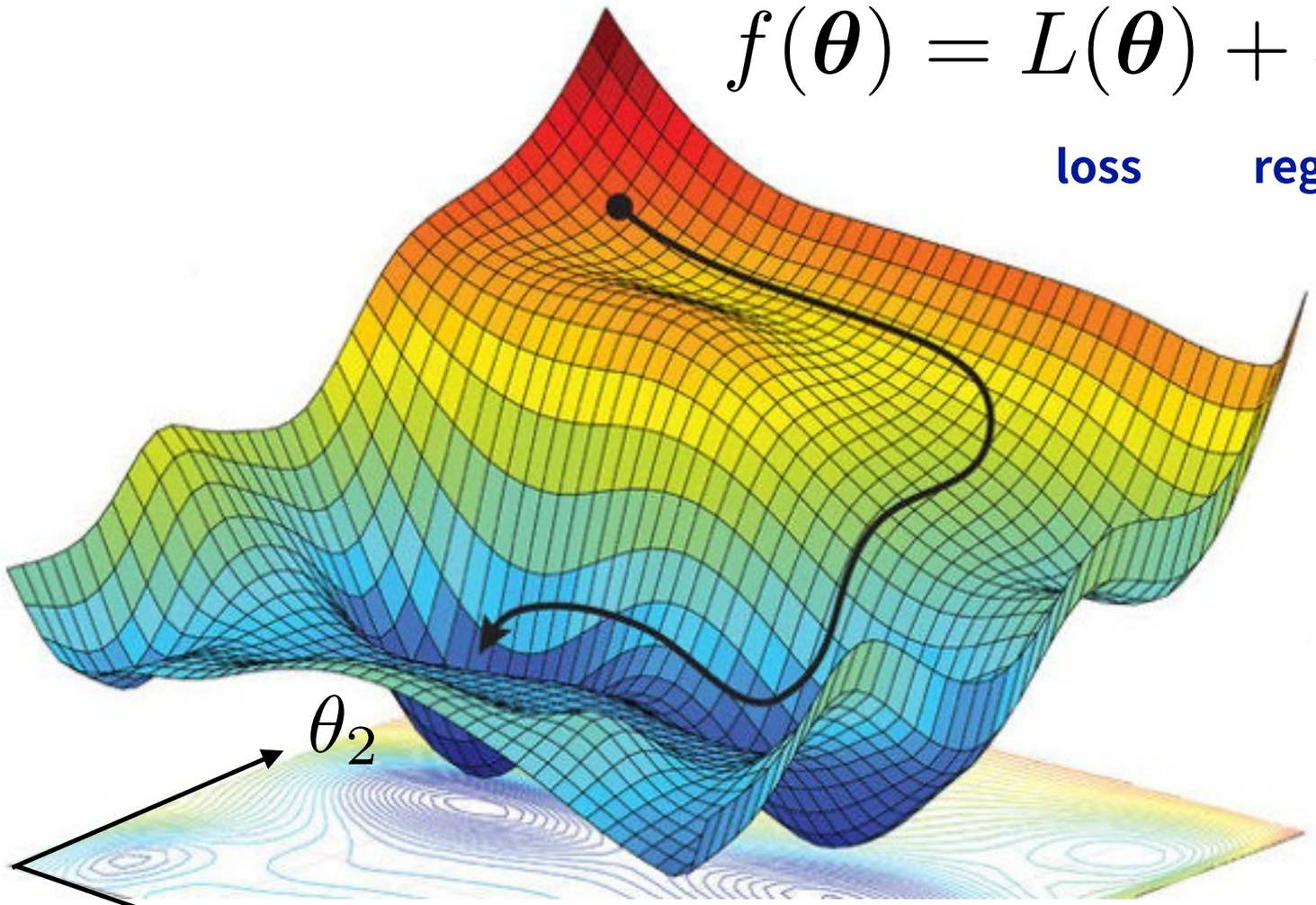
(制約なし)最小化問題 (モデルの"学習" or フィッティング)

$$\min_{\boldsymbol{\theta}} f(\boldsymbol{\theta})$$

$$f(\boldsymbol{\theta}) = L(\boldsymbol{\theta}) + \Omega(\boldsymbol{\theta})$$

loss　　regularization

本日のテーマ



$$\boldsymbol{\theta} = (\theta_1, \theta_2)$$

ただし、めちゃくちゃ高次元で！！

https://arxiv.org/abs/1905.11946

M: Million = 100万

Model	Top-1 Acc.	Top-5 Acc.	#Params
EfficientNet-B0	**77.1%**	**93.3%**	**5.3M**
ResNet-50 (He et al., 2016)	76.0%	93.0%	26M
DenseNet-169 (Huang et al., 2017)	76.2%	93.2%	14M
EfficientNet-B1	**79.1%**	**94.4%**	**7.8M**
ResNet-152 (He et al., 2016)	77.8%	93.8%	60M
DenseNet-264 (Huang et al., 2017)	77.9%	93.9%	34M
Inception-v3 (Szegedy et al., 2016)	78.8%	94.4%	24M
Xception (Chollet, 2017)	79.0%	94.5%	23M
EfficientNet-B2	**80.1%**	**94.9%**	**9.2M**
Inception-v4 (Szegedy et al., 2017)	80.0%	95.0%	48M
Inception-resnet-v2 (Szegedy et al., 2017)	80.1%	95.1%	56M
EfficientNet-B3	**81.6%**	**95.7%**	**12M**
ResNeXt-101 (Xie et al., 2017)	80.9%	95.6%	84M
PolyNet (Zhang et al., 2017)	81.3%	95.8%	92M
EfficientNet-B4	**82.9%**	**96.4%**	**19M**
SENet (Hu et al., 2018)	82.7%	96.2%	146M
NASNet-A (Zoph et al., 2018)	82.7%	96.2%	89M
AmoebaNet-A (Real et al., 2019)	82.8%	96.1%	87M
PNASNet (Liu et al., 2018)	82.9%	96.2%	86M
EfficientNet-B5	**83.6%**	**96.7%**	**30M**
AmoebaNet-C (Cubuk et al., 2019)	83.5%	96.5%	155M
EfficientNet-B6	**84.0%**	**96.8%**	**43M**
EfficientNet-B7	**84.3%**	**97.0%**	**66M**
GPipe (Huang et al., 2018)	84.3%	97.0%	557M

→ 2600万パラメタ (ResNet-50)

→ 8400万パラメタ (ResNeXt-101)

ただし、めちゃくちゃ高次元で！！

```python
def count_param(model):
  return sum(p.numel() for p in model.parameters() if p.requires_grad)
```

```python
for name in ["densenet121", "resnet50", "densenet161", "resnet101", \
             "alexnet", "resnext101_32x8d", "vgg19_bn", ]:
  model = models.__dict__[name](pretrained=False)
  print(f'# params {count_param(model):12,} ({name})')
```

```
# params     7,978,856 (densenet121)
# params    25,557,032 (resnet50)
# params    28,681,000 (densenet161)
# params    44,549,160 (resnet101)
# params    61,100,840 (alexnet)
# params    88,791,336 (resnext101_32x8d)
# params   143,678,248 (vgg19_bn)
```

```python
for i in range(8):
  name = f"efficientnet-b{i}"
  model = EfficientNet.from_name(name)
  print(f'# params {count_param(model):12,} ({name})')
```

```
# params     5,288,548 (efficientnet-b0)
# params     7,794,184 (efficientnet-b1)
# params     9,109,994 (efficientnet-b2)
# params    12,233,232 (efficientnet-b3)
# params    19,341,616 (efficientnet-b4)
# params    30,389,784 (efficientnet-b5)
# params    43,040,704 (efficientnet-b6)
# params    66,347,960 (efficientnet-b7)
```

https://arxiv.org/abs/2005.14165

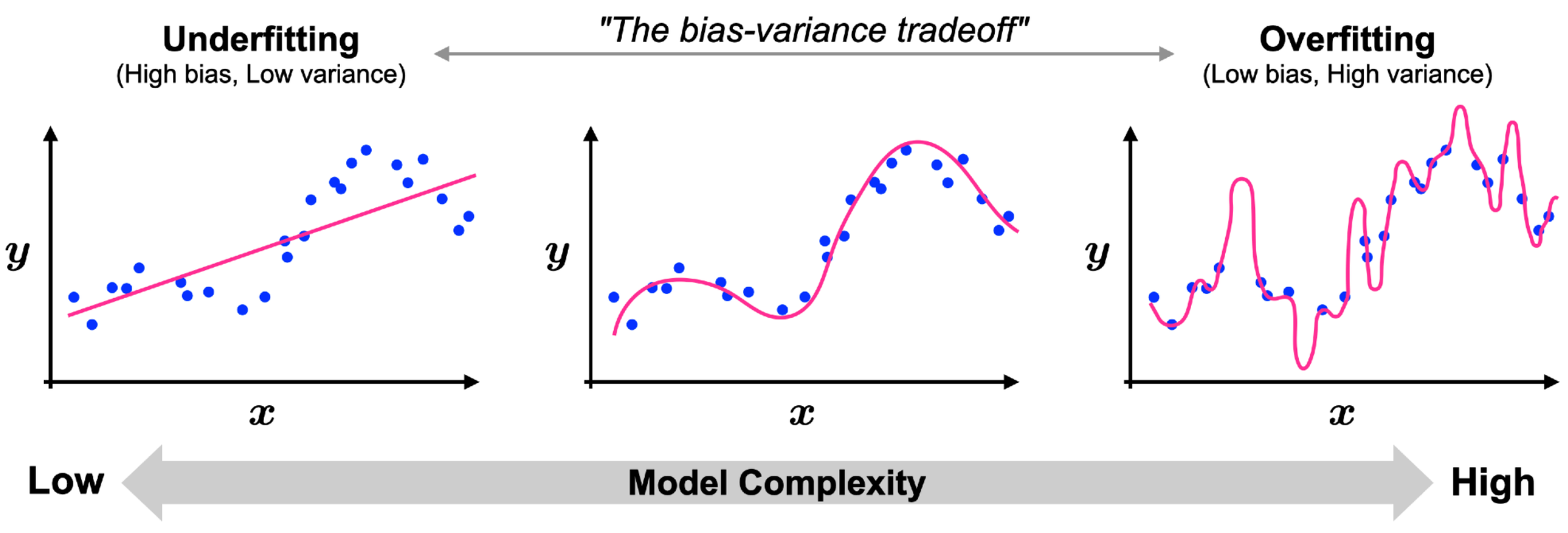
Model Name	n_{params}	n_{layers}
GPT-3 Small	125M	12
GPT-3 Medium	350M	24
GPT-3 Large	760M	24
GPT-3 XL	1.3B	24
GPT-3 2.7B	2.7B	32
GPT-3 6.7B	6.7B	32
GPT-3 13B	13.0B	40
GPT-3 175B or “GPT-3”	175.0B	96

1750億パラメタ （GPT-3）

ちなみに：なぜ「正則化(Regularization)」が必要か？

正則化 = 「モデルの曲面」が複雑になりすぎるのを防ぐ制約機構

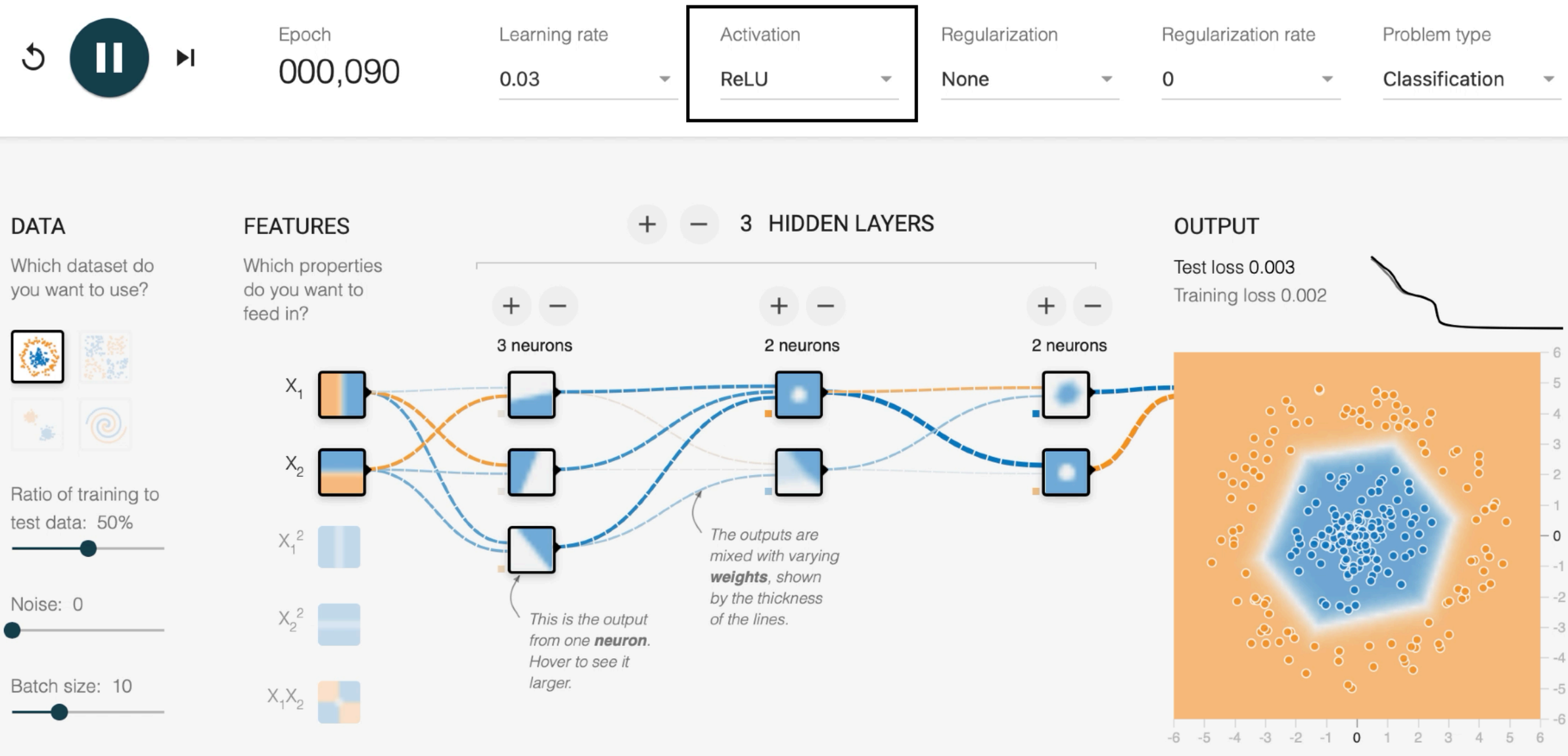
訓練データは常に「有限個」なので、複雑すぎるモデルを
フィッティングすると過学習が起きてしまう (高次元では特に)



機械学習モデルの例：MLP (多層パーセプトロン)

PyTorch

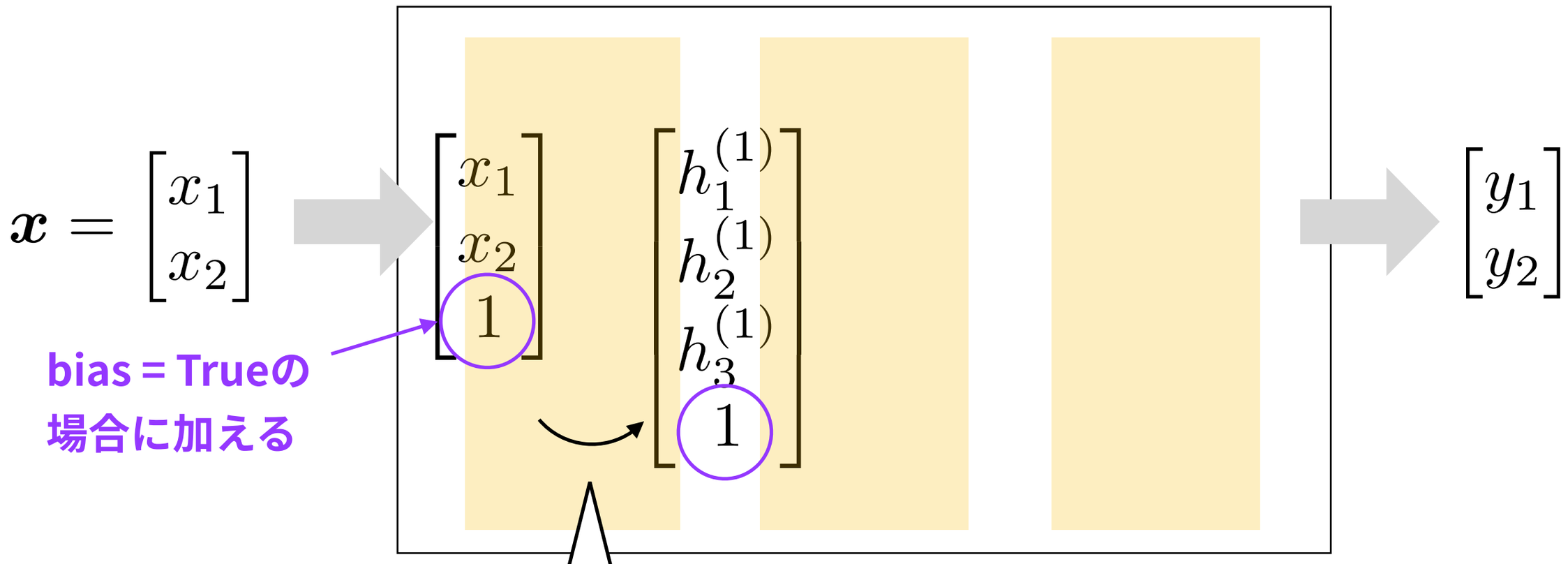
```
import torch.nn as nn
nn.Sequential(nn.Linear(2, 3), nn.ReLU(), nn.Linear(3, 2), nn.ReLU(), nn.Linear(2, 2))
```



https://playground.tensorflow.org

機械学習モデルの例：MLP (多層パーセプトロン)

```
import torch.nn as nn
nn.Sequential(nn.Linear(2, 3), nn.ReLU(), nn.Linear(3, 2), nn.ReLU(), nn.Linear(2, 2))
```



nn.Linear(2, 3)

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix}$$

行列パラメタを乗算

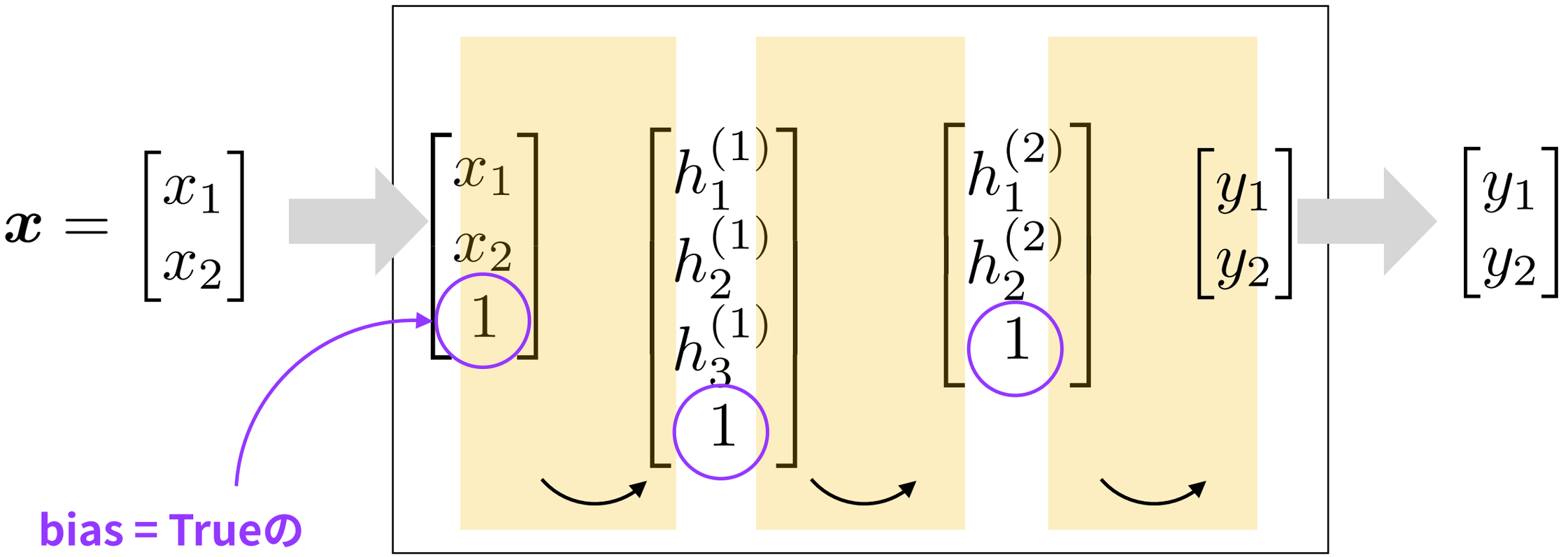
nn.ReLU()

$$\begin{bmatrix} \max(z_1, 0) \\ \max(z_2, 0) \\ \max(z_3, 0) \end{bmatrix} = \begin{bmatrix} h_1^{(1)} \\ h_2^{(1)} \\ h_3^{(1)} \end{bmatrix}$$

負値をゼロ置換(ReLU)

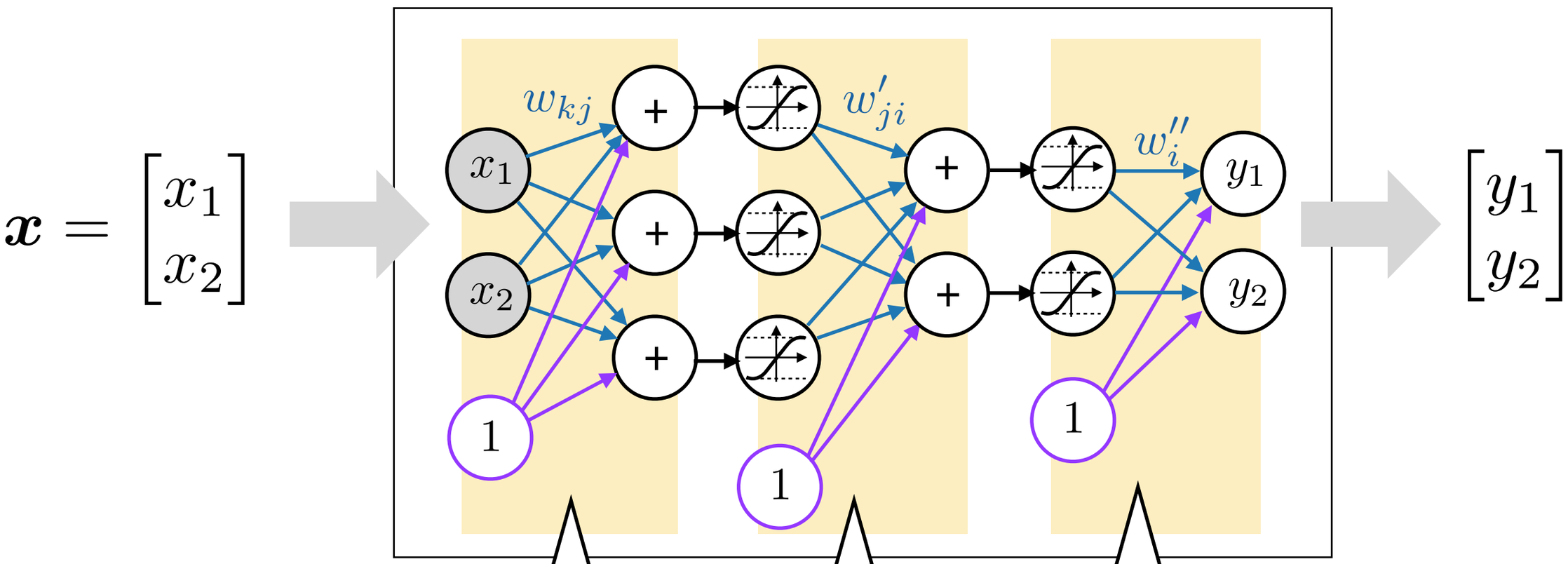
機械学習モデルの例：MLP (多層パーセプトロン)

```
import torch.nn as nn
nn.Sequential(nn.Linear(2, 3), nn.ReLU(), nn.Linear(3, 2), nn.ReLU(), nn.Linear(2, 2))
```



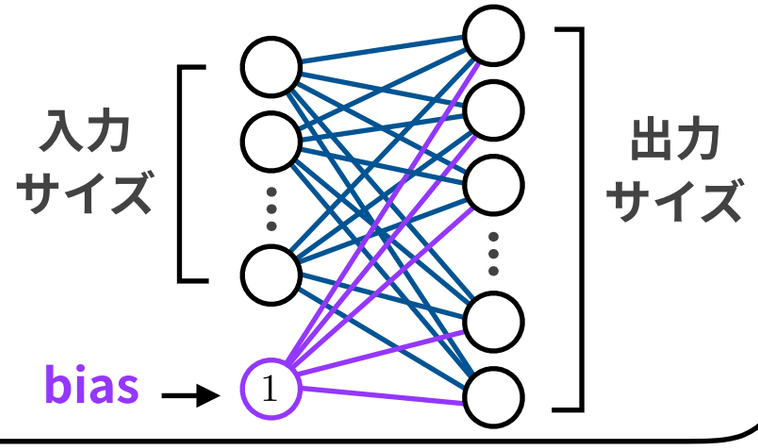
機械学習モデルの例：MLP (多層パーセプトロン)

```
import torch.nn as nn
nn.Sequential(nn.Linear(2, 3), nn.ReLU(), nn.Linear(3, 2), nn.ReLU(), nn.Linear(2, 2))
```



torch.nn.Linear

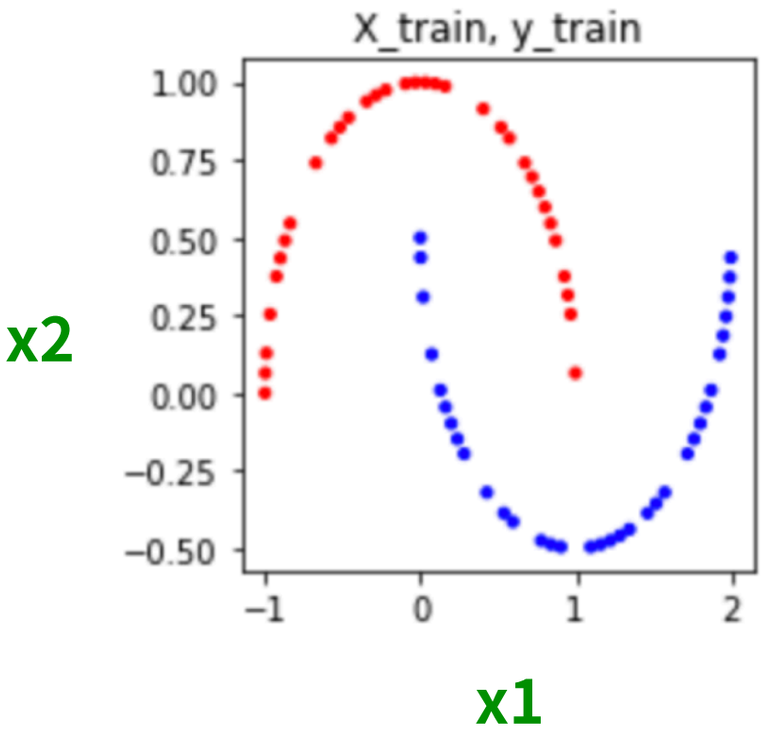
- **in_features** 入力サイズ
- **out_features** 出力サイズ
- **bias (True/False)** biasの有無

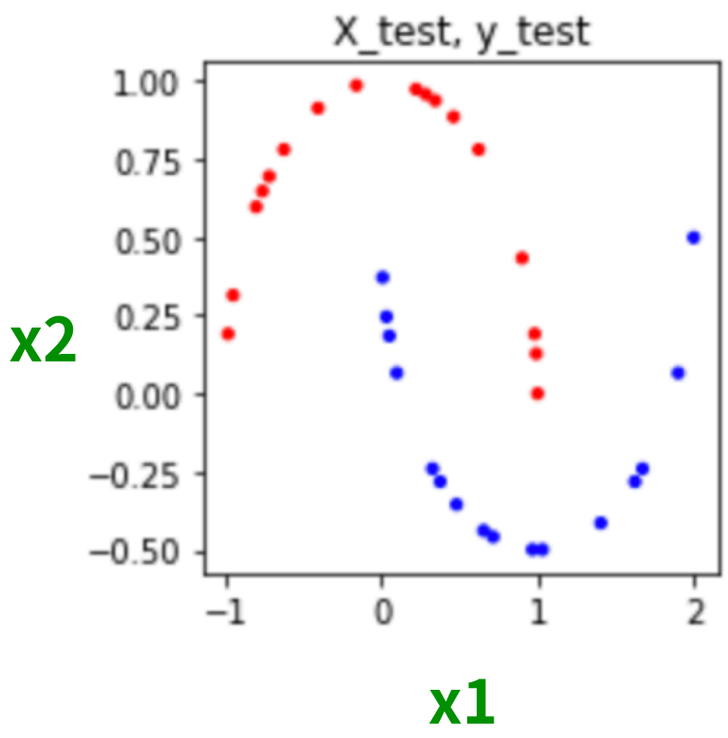


例

X: 2次元の点 (x1, x2)
y: 色 (赤 or 青)

赤 y = 0
青 y = 1





見本例 (訓練データ)

検証例 (検証データ)

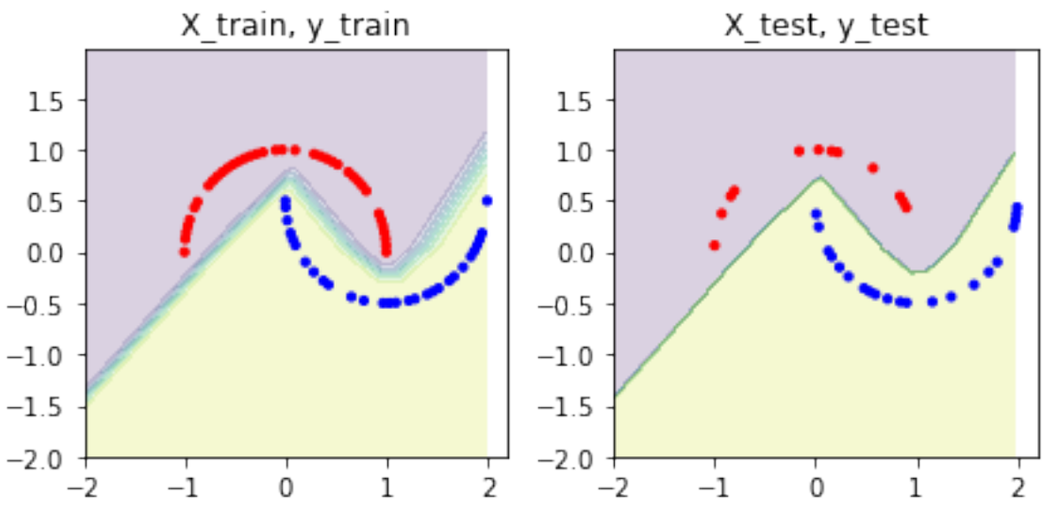
見本例と違うデータを予測させて
みて答えあわせして正答率をみる！

結果

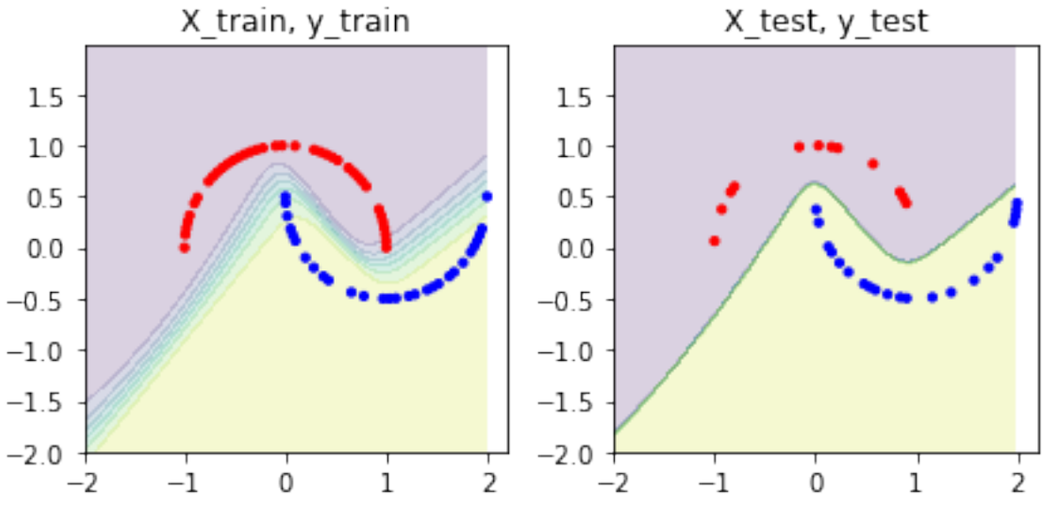
X: 2次元の点 (x1, x2)
y: 色 (赤 or 青)

赤 y = 0
青 y = 1

model = nn.Sequential(nn.Linear(2, 10), nn.ReLU(), nn.Linear(10, 2))



model = nn.Sequential(nn.Linear(2, 10), nn.Sigmoid(), nn.Linear(10, 2))



コード

https://colab.research.google.com/drive/1tqRxaayH-O9VRxwr8XoMn8JZnBqElmZM?usp=sharing

```
import torch.nn as nn
import torch.optim
from torch.utils.data import DataLoader
```

```
trainloader = DataLoader(traindata, batch_size=16)
testloader = DataLoader(testdata, batch_size=16)
```

```
model = nn.Sequential(nn.Linear(2, 10), nn.Sigmoid(), nn.Linear(10, 2))
#model = nn.Sequential(nn.Linear(2, 10), nn.ReLU(), nn.Linear(10, 2))
```

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.1)
```

```
for epoch in range(50):
    for (inputs, labels) in trainloader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

学習のメインループ (pytorch)

```
model = ...

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

for epoch in range(100):
    for (inputs, labels) in trainloader:
        optimizer.zero_grad()
        loss = criterion(model(inputs), labels)
        loss.backward()
        optimizer.step()
```

本日の技術的なメインテーマ：勾配法と自動微分

深層モデルの学習 (pytorchの最小記述でたった7行!)のコア部分で何をやっているのかを理解する

```
for epoch in range(num_epochs):
    for (inputs, labels) in dataloader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

optimizer.zero_grad()

loss = criterion(model(inputs), labels) → 自動微分 (forward)

loss.backward() → 自動微分 (backward)

optimizer.step() 勾配法

2

勾配と勾配降下法

- 偏微分と勾配
- 勾配法とは？
- 具体例

モデルの学習 = パラメタの値を最適にする (最適化)

見本例を予測

予測値　教師の値

$\boldsymbol{x}_1 \longrightarrow f(\boldsymbol{x}, \boldsymbol{\theta}) \longrightarrow \hat{y}_1 \quad y_1$

$\boldsymbol{x}_2 \longrightarrow f(\boldsymbol{x}, \boldsymbol{\theta}) \longrightarrow \hat{y}_2 \quad y_2$

$\boldsymbol{x}_n \longrightarrow f(\boldsymbol{x}, \boldsymbol{\theta}) \longrightarrow \hat{y}_n \quad y_n$

$\boldsymbol{\theta} = (\theta_1, \theta_2, \ldots, \theta_p)$

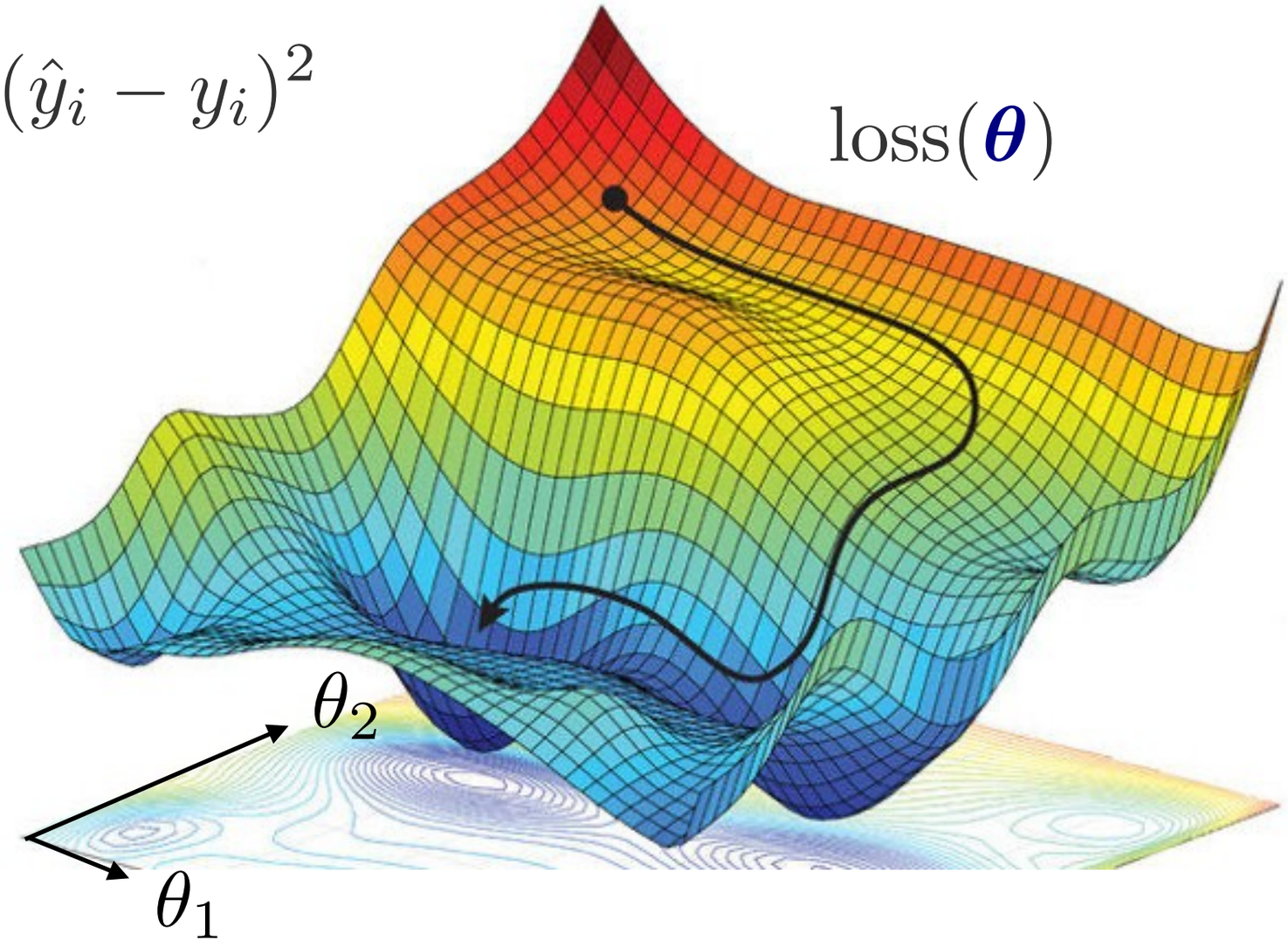
予測値と教師の値が近づくように
パラメタ $\boldsymbol{\theta}$ を調整

「Loss関数」の最小値を見つければよい

n個の見本の全体について予測値と教師の値を近づけるよう θ を動かす

関数 $\mathrm{loss}(\theta)$ の値が小さくなるようにパラメタ θ を動かす

$$\mathrm{loss}(\boldsymbol{\theta}) = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$$



$$\boldsymbol{\theta} = (\theta_1, \theta_2)$$

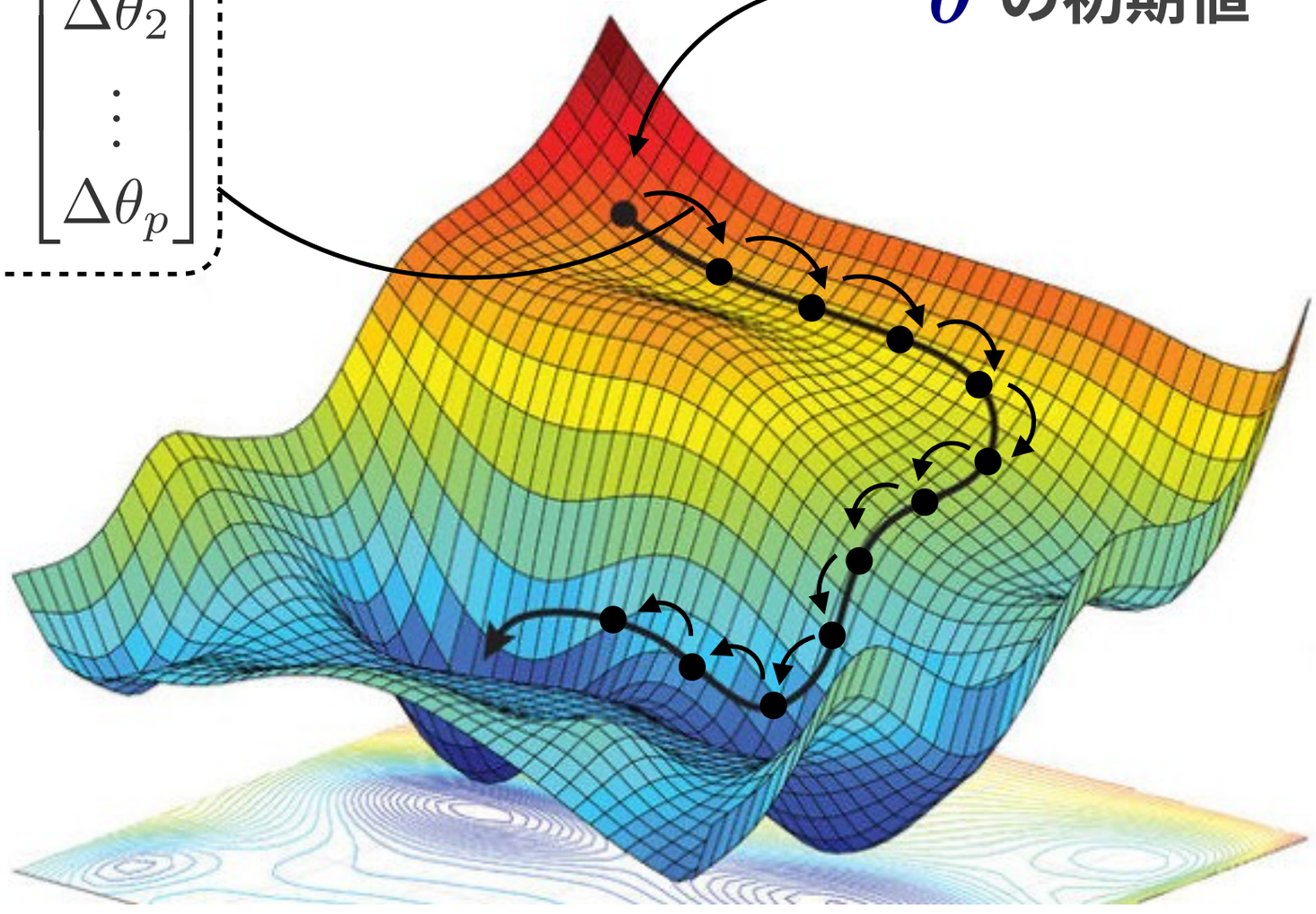
「ちょっと動かす」を反復：パラメタの最適化の基本方針

- $\boldsymbol{\theta}$ をランダム値で初期化
- $\mathrm{loss}(\boldsymbol{\theta})$ の値が小さくなるように $\boldsymbol{\theta}$ を**ちょっとだけ**変える

$$\begin{bmatrix}\theta_1\\ \theta_2\\ \vdots\\ \theta_p\end{bmatrix} \leftarrow \begin{bmatrix}\theta_1\\ \theta_2\\ \vdots\\ \theta_p\end{bmatrix} + 0.01 \times \begin{bmatrix}\Delta\theta_1\\ \Delta\theta_2\\ \vdots\\ \Delta\theta_p\end{bmatrix}$$

くりかえし

$\boldsymbol{\theta}$ の初期値



0.01 1ステップの“歩幅”

学習率 (step size)

Q. どう動かす？ → A.「勾配」に比例する方向

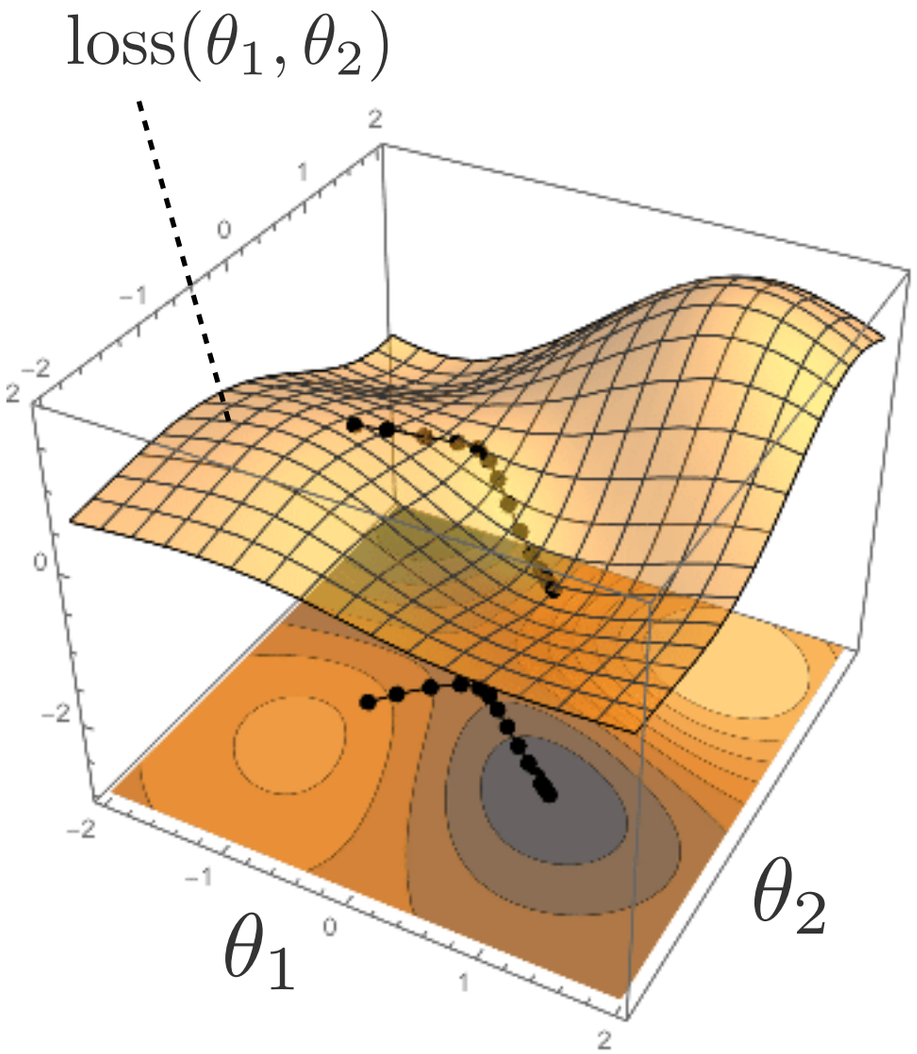
$$\begin{bmatrix}\theta_1\\ \theta_2\\ \vdots\\ \theta_p\end{bmatrix} \leftarrow \begin{bmatrix}\theta_1\\ \theta_2\\ \vdots\\ \theta_p\end{bmatrix} + 0.01 \times \begin{bmatrix}\Delta\theta_1\\ \Delta\theta_2\\ \vdots\\ \Delta\theta_p\end{bmatrix}$$

この方向を
どう選ぶ？

0.01

1ステップの
“歩幅”

学習率
(step size)



勾配ベクトルに比例する量を**ちょっと引く**と
最寄りの極小値に向かうことがわかっている

$$\begin{bmatrix}\Delta\theta_1\\ \Delta\theta_2\\ \vdots\\ \Delta\theta_p\end{bmatrix} \propto \begin{bmatrix}\partial \mathrm{loss}(\boldsymbol{\theta})/\partial\theta_1\\ \partial \mathrm{loss}(\boldsymbol{\theta})/\partial\theta_2\\ \vdots\\ \partial \mathrm{loss}(\boldsymbol{\theta})/\partial\theta_p\end{bmatrix}$$

勾配ベクトル

勾配 = 各変数での偏微分の値を並べたベクトル

点 a での関数 $f(x)$ の**勾配(gradient)**

$$\boldsymbol{a} = (a_1, \ldots, a_d)$$
$$\boldsymbol{x} = (x_1, \ldots, x_d)$$

$$\nabla f(\boldsymbol{a}) = \left[\begin{array}{c} \partial f(\boldsymbol{x})/\partial x_1 \\ \partial f(\boldsymbol{x})/\partial x_2 \\ \vdots \\ \partial f(\boldsymbol{x})/\partial x_d \end{array}\right]\Bigg|_{\boldsymbol{x}=\boldsymbol{a}}$$

i 成分は

x_i での偏微分

点 $(a_1, \ldots, a_d)$ で x_i だけをほんのちょっと(Δだけ)増やしたら

関数 f の値はどれくらい増えるのか？

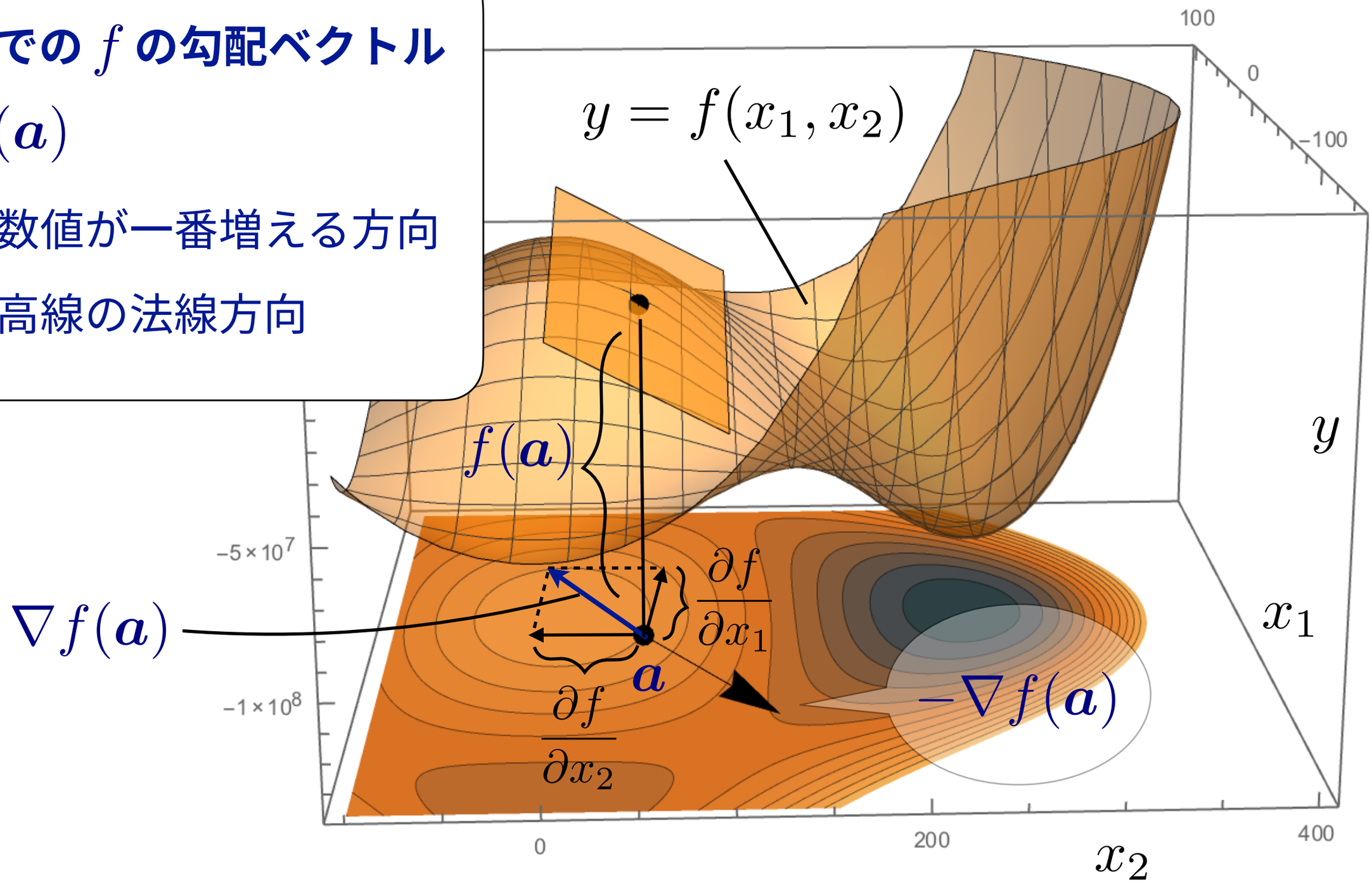
$$\left.\frac{\partial f(\boldsymbol{x})}{\partial x_i}\right|_{\boldsymbol{x}=\boldsymbol{a}} = \lim_{\Delta \to 0} \frac{f(a_1, \ldots, a_i + \Delta, \ldots, a_d) - f(\boldsymbol{a})}{\Delta}$$

勾配 = 各変数での偏微分の値を並べたベクトル

点 a での f の勾配ベクトル

$\nabla f(\boldsymbol{a})$

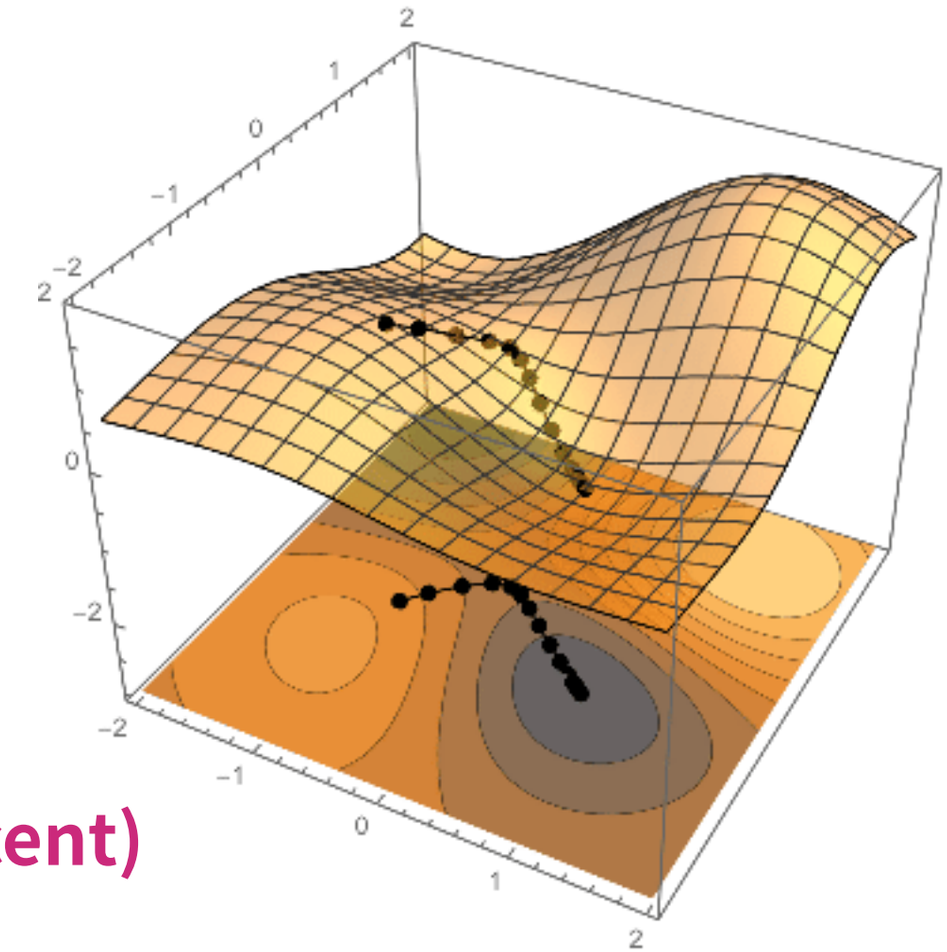
- 関数値が一番増える方向
- 等高線の法線方向



勾配法

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leftarrow \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix}$$

$$y = f(x_1, x_2)$$



最小化の場合(関数値を下げたい)

勾配の反対方向に動かせばよい

$$\begin{bmatrix} \Delta x_1 \\ \Delta x_2 \end{bmatrix} = -\eta \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix}$$

勾配降下
(Gradient descent)

ステップサイズ: その方向へどれだけ大きく進むか

(とりあえず通り過ぎないよう小さい値に設定しておく)

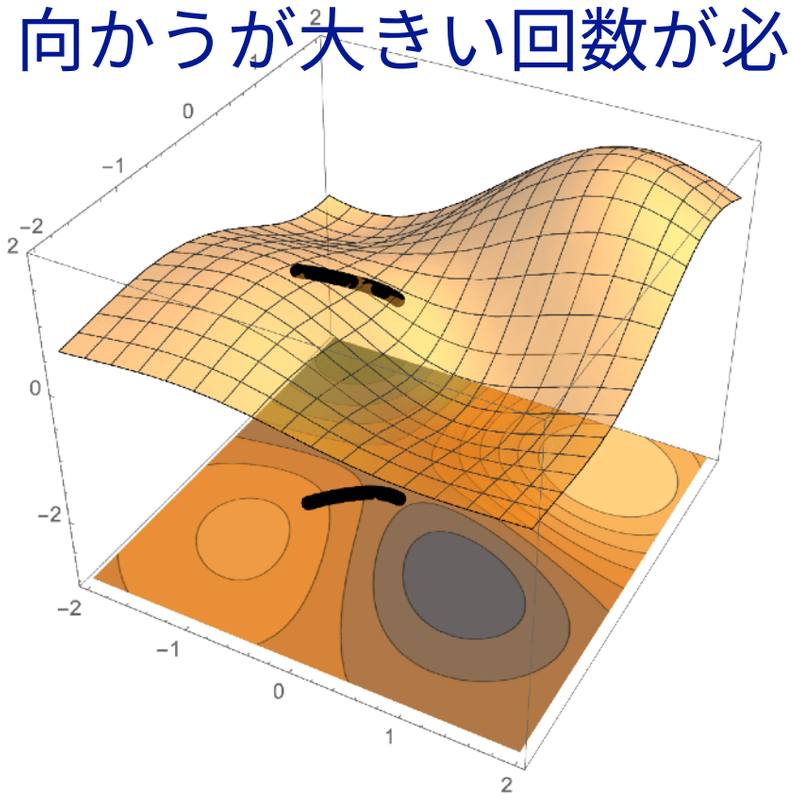
学習率/ステップサイズ = 歩幅を決める

$$\begin{bmatrix}\theta_1\\ \theta_2\\ \vdots\\ \theta_p\end{bmatrix} \leftarrow \begin{bmatrix}\theta_1\\ \theta_2\\ \vdots\\ \theta_p\end{bmatrix} + \underbrace{0.01}_{\text{lr}} \times \begin{bmatrix}\Delta\theta_1\\ \Delta\theta_2\\ \vdots\\ \Delta\theta_p\end{bmatrix}$$

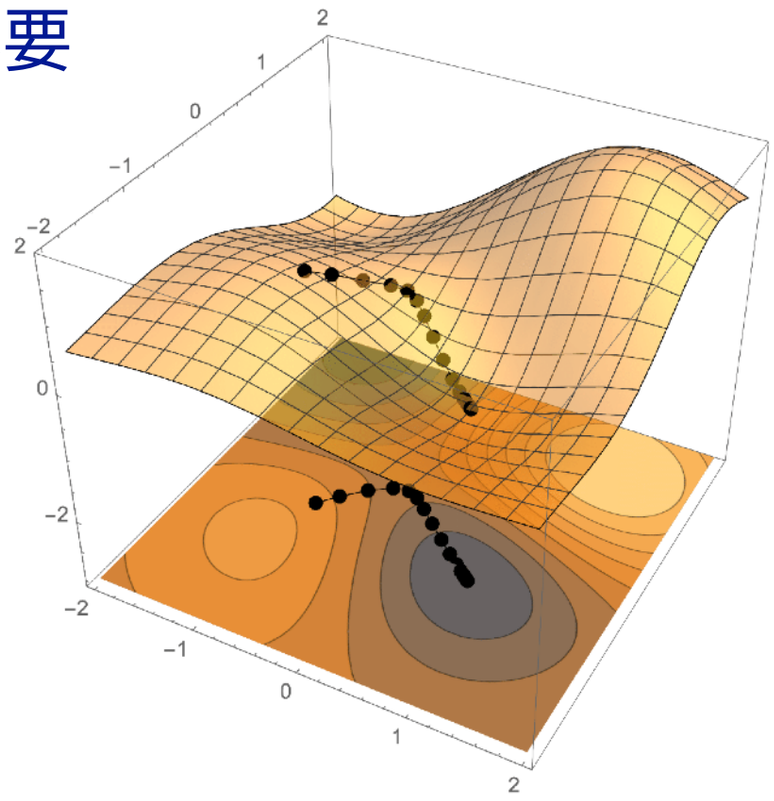
この学習率(learning rate; lr)の値はどれくらいにする？

lr 小

確実に最寄りの極小値に向かうが大きい回数が必要

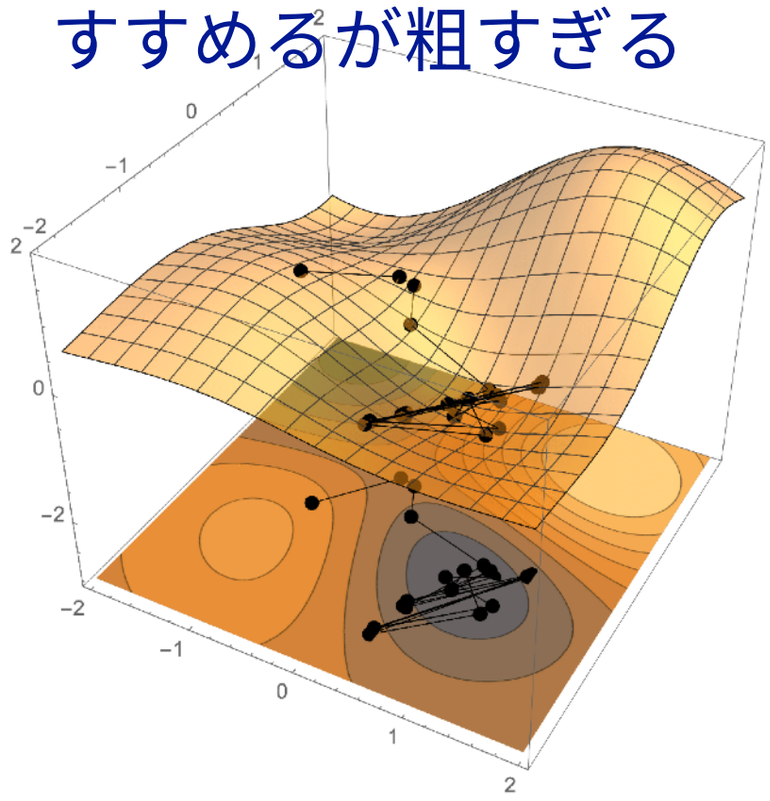


lr 中



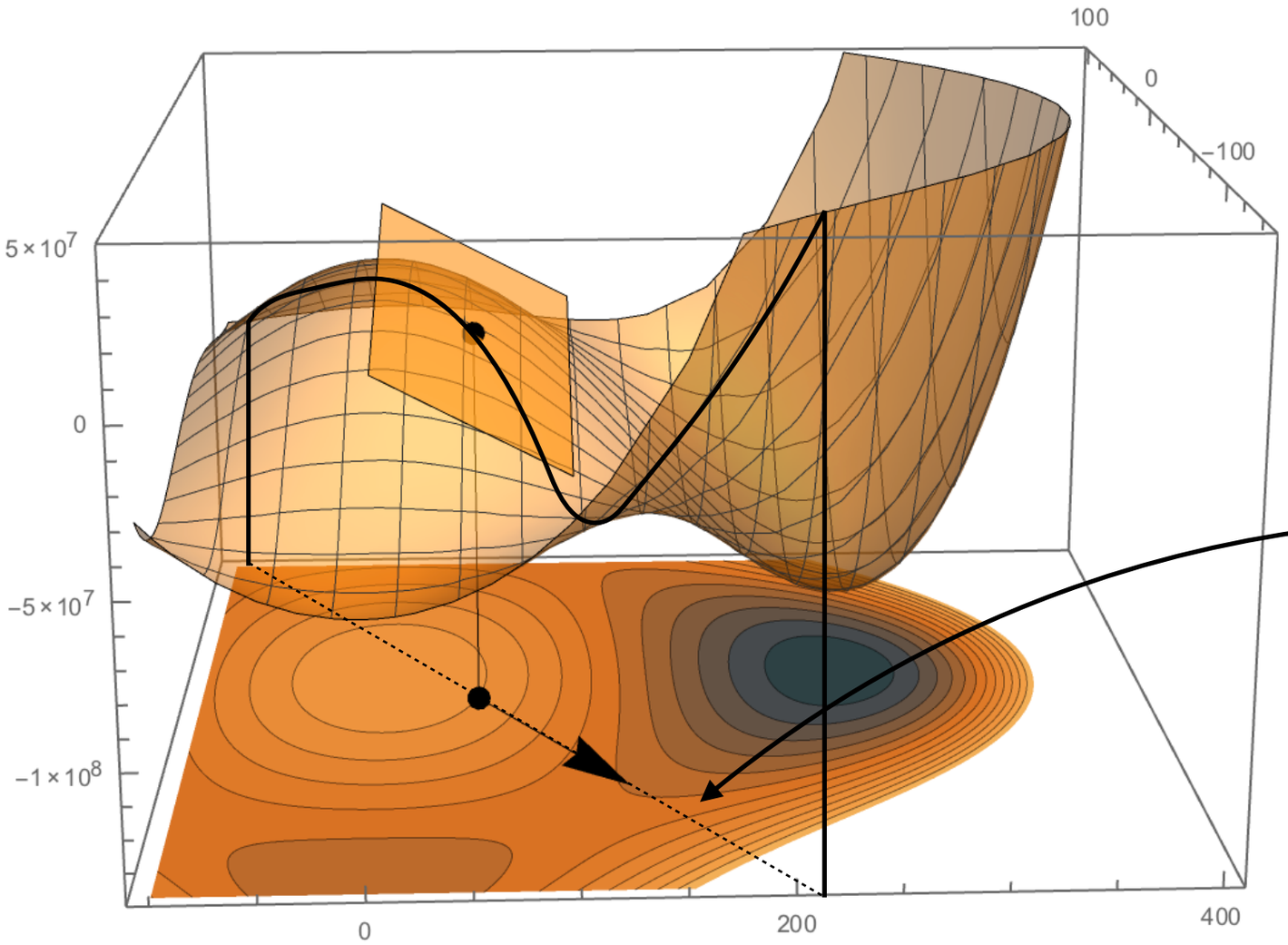
lr 大

少ない回数で大きくすすめるが粗すぎる



まじめにやるなら…：毎回、直線探索 (line search)

極小解への収束を保証するためにはステップサイズを真面目に
決めないとダメ。(でも実際はサボって小さい定数にすること多い)

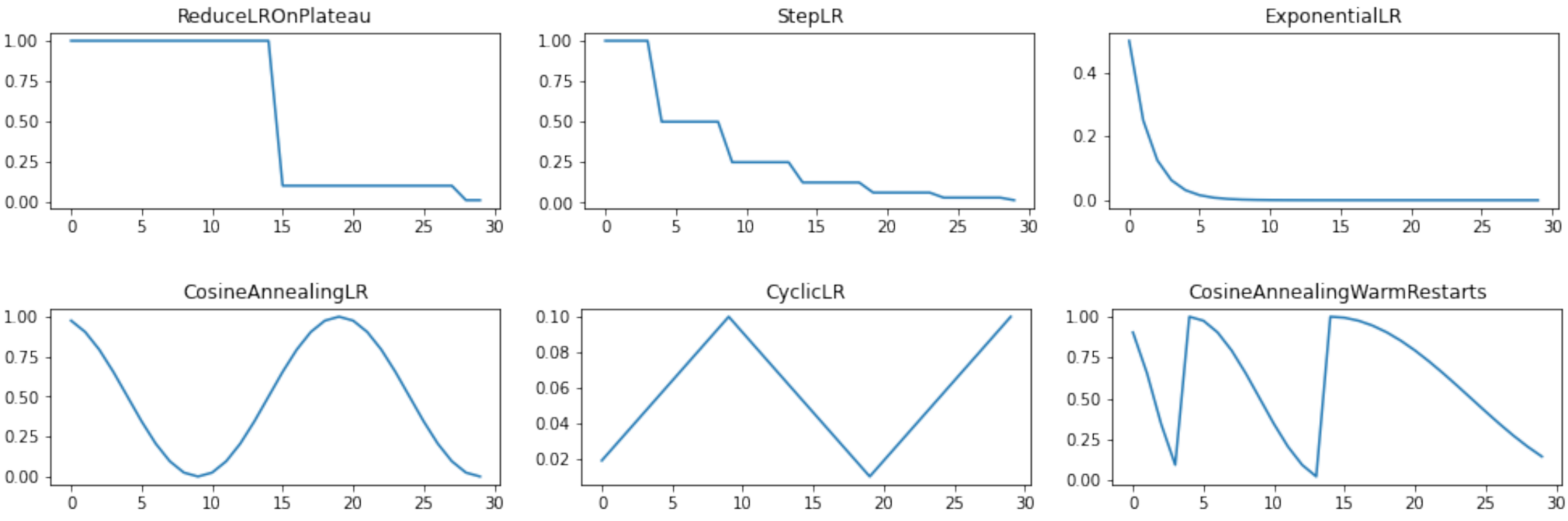


勾配の反対方向の直線上
を**どれくらい進むか**
(ステップサイズ)を
この直線上での関数値で
一番小さくなるところを
ちゃんと決める。

学習率スケジューリング

機械学習では大きい値から始めて更新回数が進むにつれて
値を小さくしていくなど**lr値のスケジューリング**をする
(直線探索は評価が高コストなケースではしない)

torch.optim.lr_schedulerの例



ここまでのまとめ：機械学習

① モデルを定義　$\boldsymbol{x} \longrightarrow \boxed{f(\boldsymbol{x}, \boldsymbol{\theta})} \longrightarrow \boldsymbol{y}$

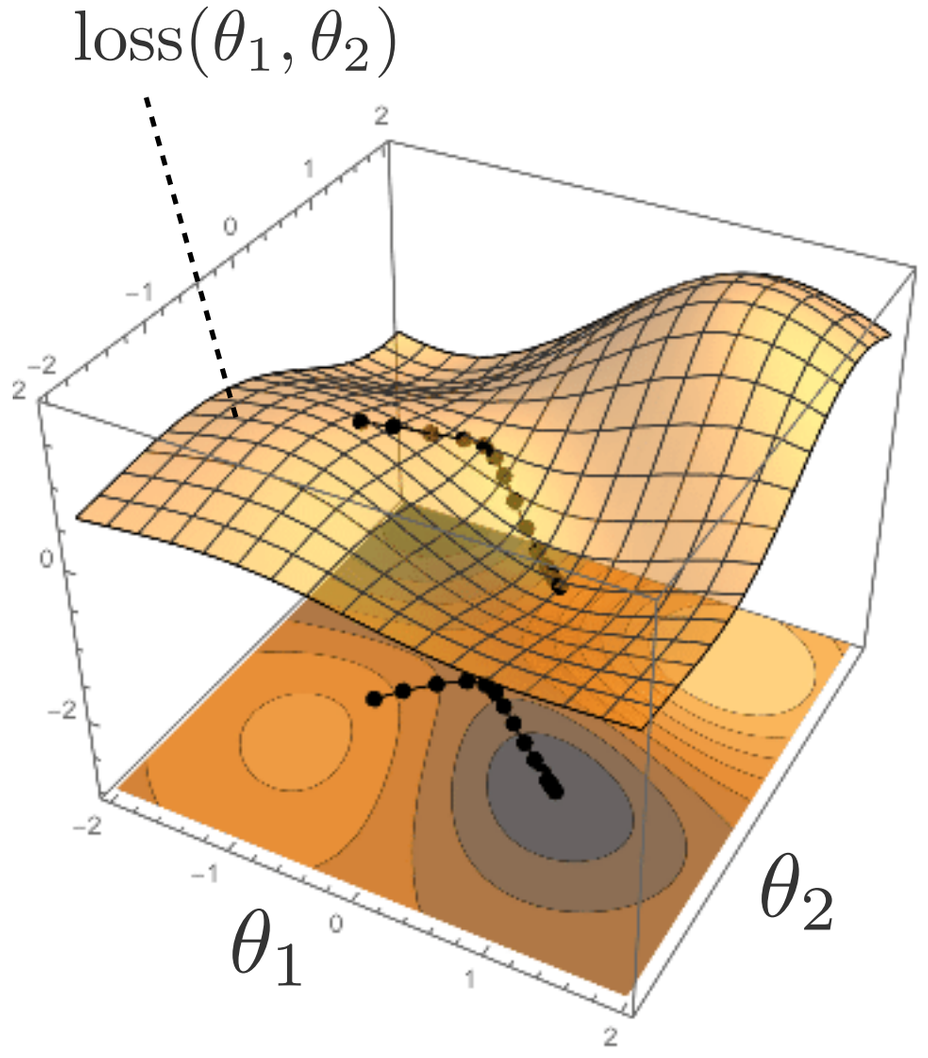
② Lossを定義　$\mathrm{loss}(\boldsymbol{\theta}) = \sum_{i=1}^{n} \mathrm{error}(f(\boldsymbol{x}_i, \boldsymbol{\theta}), y_i) + \Omega(\boldsymbol{\theta})$

③ 勾配法：Lossの勾配を使ってモデルパラメタ $\boldsymbol{\theta}$ を最適化

$$\begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_p \end{bmatrix} \leftarrow \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_p \end{bmatrix} - \eta \times \begin{bmatrix} \partial \mathrm{loss}(\boldsymbol{\theta})/\partial\theta_1 \\ \partial \mathrm{loss}(\boldsymbol{\theta})/\partial\theta_2 \\ \vdots \\ \partial \mathrm{loss}(\boldsymbol{\theta})/\partial\theta_p \end{bmatrix}$$

学習率 (step size)

勾配ベクトル



3

合成関数の自動微分

- 合成関数の偏微分と連鎖律
- 自動微分とは？
- 具体例
- 微分可能プログラミング

計算グラフと自動微分：複雑なモデルの勾配の計算

機械学習モデル

$$\boldsymbol{x}_i \longrightarrow \boxed{f(\boldsymbol{x}, \boldsymbol{\theta})} \longrightarrow \hat{y}_i$$

小さくしたい尺度

$$\begin{aligned}\mathrm{loss}(\boldsymbol{\theta}) &= \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \\ &= \sum_{i=1}^{n} (f(\boldsymbol{x}_i, \boldsymbol{\theta}) - y_i)^2\end{aligned}$$

勾配ベクトル

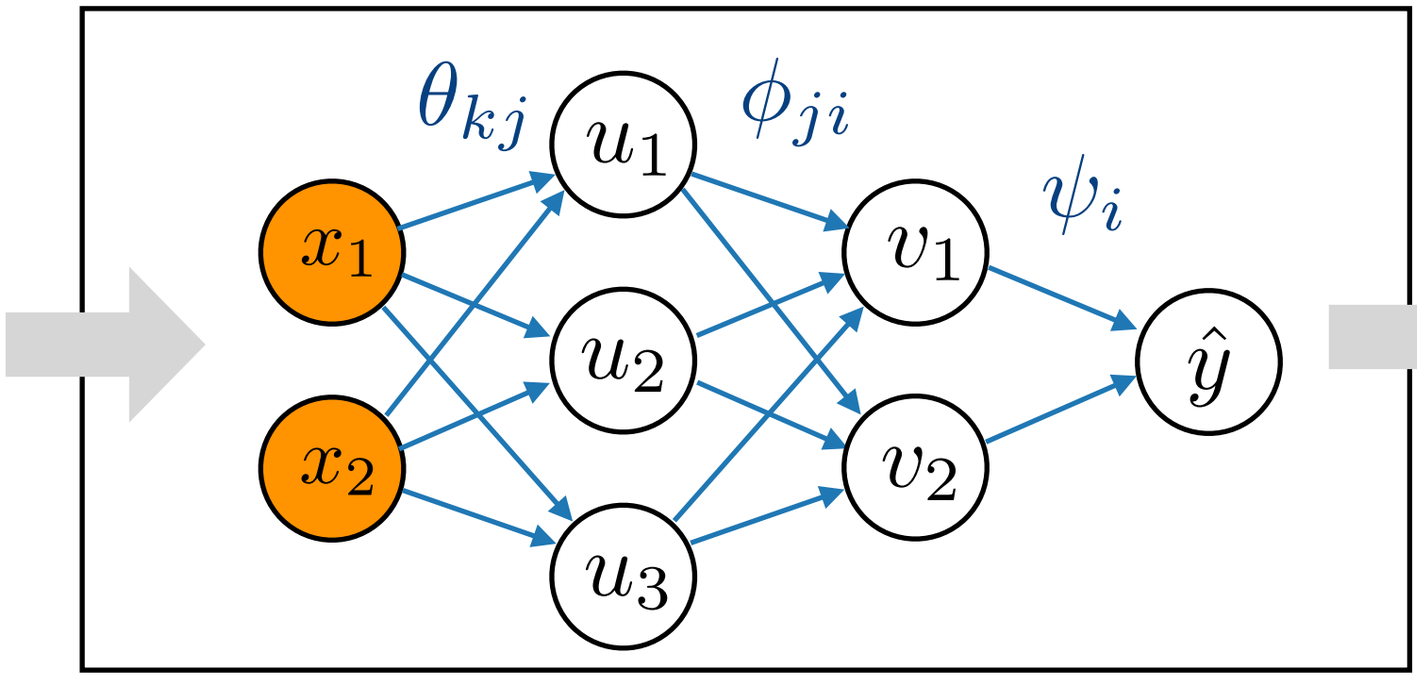
$$\begin{bmatrix} \partial \mathrm{loss}(\boldsymbol{\theta})/\partial\theta_1 \\ \partial \mathrm{loss}(\boldsymbol{\theta})/\partial\theta_2 \\ \vdots \\ \partial \mathrm{loss}(\boldsymbol{\theta})/\partial\theta_p \end{bmatrix}$$

← **これをどう計算するかという話**

機械学習モデルやそのLoss = 素演算の合成関数

計算の出力が別の計算の入力になってネストされる「合成関数」

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$



$$\hat{y}$$

$$u_1 = \sigma(\theta_{11}x_1 + \theta_{21}x_2)$$
$$u_2 = \sigma(\theta_{12}x_1 + \theta_{22}x_2)$$
$$u_3 = \sigma(\theta_{13}x_1 + \theta_{23}x_2)$$
$$v_1 = \sigma(\phi_{11}u_1 + \phi_{21}u_2 + \phi_{31}u_3)$$
$$v_2 = \sigma(\phi_{12}u_1 + \phi_{22}u_2 + \phi_{32}u_3)$$

$$\hat{y} = \psi_1 v_1 + \psi_2 v_2$$
$$L = (y - \hat{y})^2$$

合成関数 $(x_1, x_2) \mapsto \hat{y}$

$(x_1, x_2, y) \mapsto \hat{L}$

合成関数の微分：連鎖律 (Chain Rule)

https://ja.wikipedia.org/wiki/連鎖律

f、gを微分可能な関数とするとき、合成関数$f \circ g$の導関数に対して成り立つ関係式

$$(f \circ g)'(x) = (f(g(x)))' = f'(g(x))g'(x)$$

を、連鎖律という。ライプニッツの記法では

$$\frac{df}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx}$$

となる。積分法においては、置換積分に対応する。

例) $f(g(x))$ の微分を計算

$$(f(g(x)))' = f'(g(x)) \cdot g'(x) \qquad (\sin(x^2))' = \cos(x^2) \cdot (2x)$$

中そのまま、外ビブン × 中ビブン

微分を予め手計算してハードコーディングは面倒すぎ！！

合成関数

$$
\begin{aligned}
u_1 &= \sigma(\theta_{11}x_1 + \theta_{21}x_2) \\
u_2 &= \sigma(\theta_{12}x_1 + \theta_{22}x_2) \\
u_3 &= \sigma(\theta_{13}x_1 + \theta_{23}x_2) \\
v_1 &= \sigma(\phi_{11}u_1 + \phi_{21}u_2 + \phi_{31}u_3) \\
v_2 &= \sigma(\phi_{12}u_1 + \phi_{22}u_2 + \phi_{32}u_3) \\
\hat{y} &= \psi_1 v_1 + \psi_2 v_2 \\
L &= (y - \hat{y})^2
\end{aligned}
$$

データ入力

$$
\begin{aligned}
(x_1, x_2) &= (-1.0, 2.0) \\
y &= 4.5
\end{aligned}
$$

Sigmoid関数

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

のとき $\boldsymbol{\theta} = \begin{bmatrix} 0.1 & -0.1 & 1.2 \\ 0.9 & 0.0 & -0.7 \end{bmatrix}, \boldsymbol{\phi} = \begin{bmatrix} 1.1 & -5.4 \\ 2.2 & 6.9 \\ 8.8 & -3.8 \end{bmatrix}, \boldsymbol{\psi} = \begin{bmatrix} 6.9 \\ -11.0 \end{bmatrix}$ での

$\dfrac{\partial L}{\partial \theta_{23}}$ の値を計算してみよう！

モデルが変わるたびにこの計算とコーディングをやるのは嫌だ！！

「自動微分」降臨！

```
import torch

x = torch.tensor([-1.0, 2.0])
y = torch.tensor(4.5)

theta = torch.tensor([[0.1, -0.1, 1.2],[0.9, 0.0, -0.7]],
                     requires_grad=True)
phi = torch.tensor([[1.1, -5.4], [2.2, 6.9], [8.8, -3.8]],
                   requires_grad=True)
psi = torch.tensor([6.9, -11.0],
                   requires_grad=True)
```

```
u = torch.sigmoid(theta.T @ x)
v = torch.sigmoid(phi.T @ u)
y_hat = psi @ v
L = (y-y_hat)**2
```

```
L.backward()
```

```
theta.grad.data[1, 2]
```

```
tensor(-1.6196)
```

PyTorch

自分で微分しなくても
なんかいけた！

$$\frac{\partial L}{\partial \theta_{23}} = -1.6196$$

計算グラフ：素演算の合成関数のグラフ表現

例 $\mathrm{loss} = (\hat{y} - y)^2$ 予測モデル (特に意味はないToy Example)

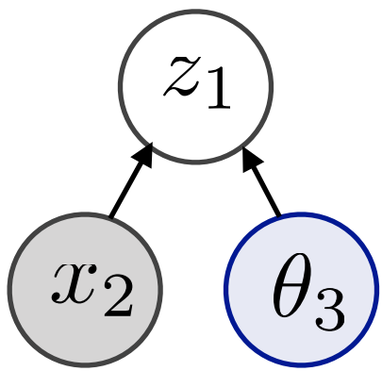
$$\hat{y} = f(x_1, x_2, \theta_1, \theta_2, \theta_3)$$
$$= \theta_1 x_1 - \theta_2 \cos(\theta_3 + x_2)$$

計算グラフ：素演算の合成関数のグラフ表現

例 $\text{loss} = (\hat{y} - y)^2$ 予測モデル (特に意味はないToy Example)

$$\hat{y} = f(x_1, x_2, \theta_1, \theta_2, \theta_3)$$
$$= \theta_1 x_1 - \theta_2 \cos(\underbrace{\theta_3 + x_2}_{z_1})$$

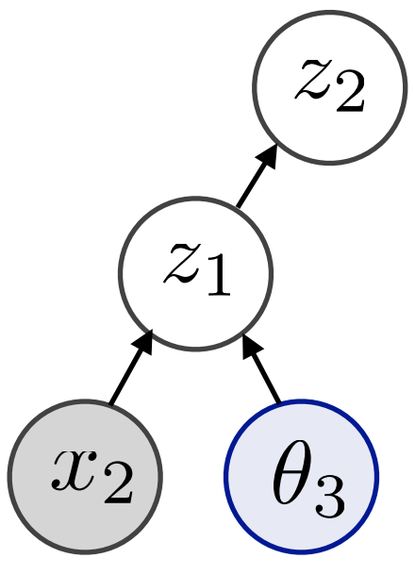
合成関数 $z_1 = \theta_3 + x_2$



計算グラフ：素演算の合成関数のグラフ表現

例 $\mathrm{loss} = (\hat{y} - y)^2$ 予測モデル (特に意味はないToy Example)

$$\hat{y} = f(x_1, x_2, \theta_1, \theta_2, \theta_3)$$
$$= \theta_1 x_1 - \theta_2 \cos(\underbrace{\theta_3 + x_2}_{z_1})$$
$$\underbrace{\qquad\qquad}_{z_2}$$



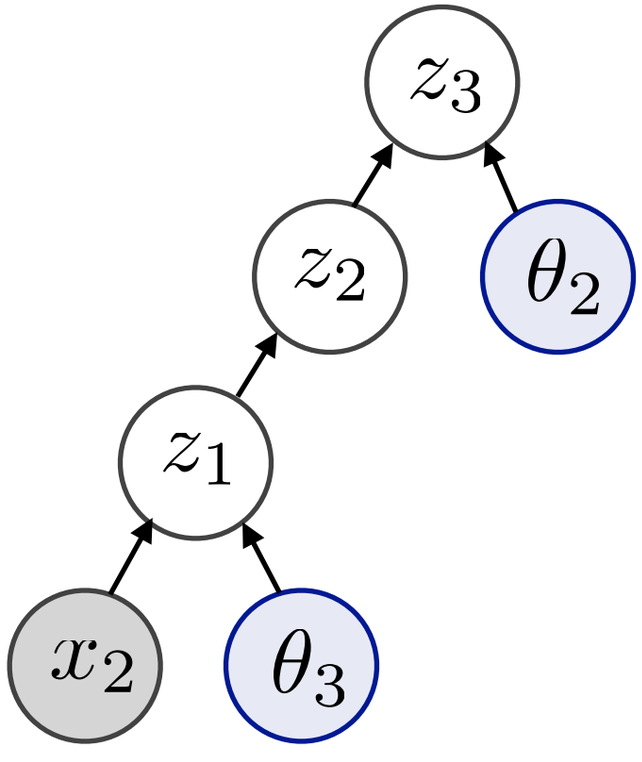
合成関数 $z_1 = \theta_3 + x_2$

$z_2 = \cos(z_1)$

計算グラフ：素演算の合成関数のグラフ表現

例 $\mathrm{loss} = (\hat{y} - y)^2$ 予測モデル (特に意味はないToy Example)

$$\hat{y} = f(x_1, x_2, \theta_1, \theta_2, \theta_3)$$
$$= \theta_1 x_1 - \theta_2 \cos(\underbrace{\theta_3 + x_2}_{z_1})$$
$$\underbrace{\qquad\qquad}_{z_2}$$
$$\underbrace{\qquad\qquad\qquad\qquad}_{z_3}$$

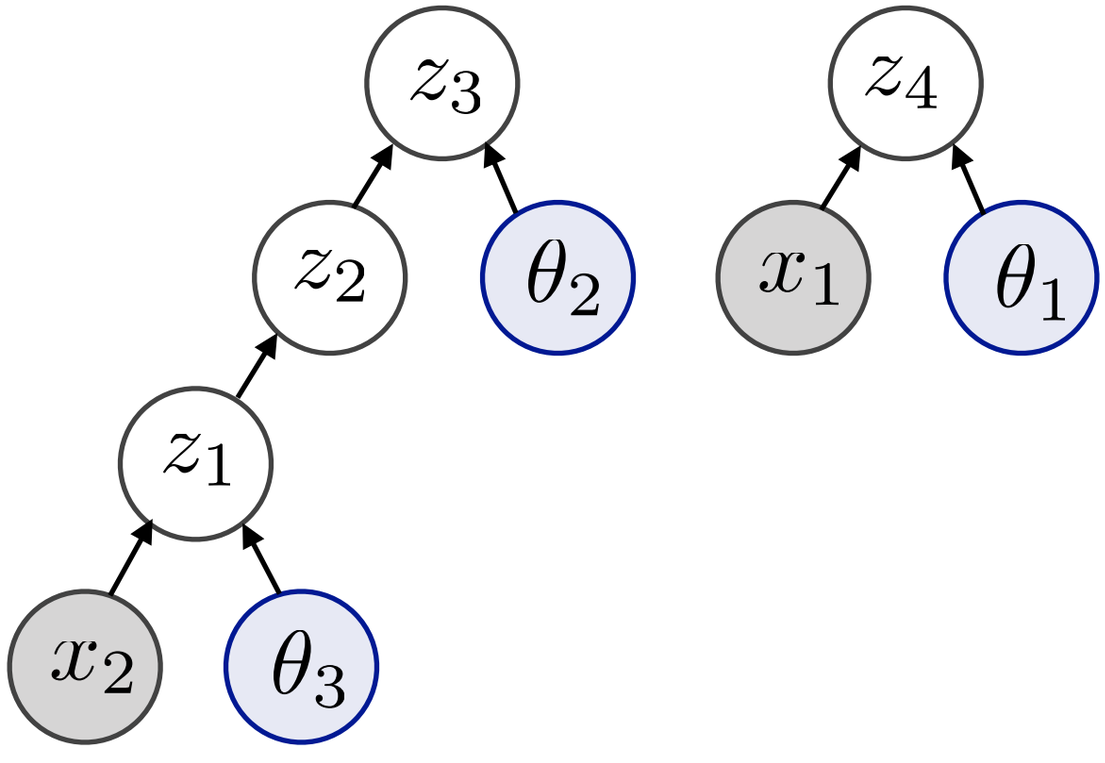


合成関数
$$z_1 = \theta_3 + x_2$$
$$z_2 = \cos(z_1)$$
$$z_3 = \theta_2 z_2$$

計算グラフ：素演算の合成関数のグラフ表現

例 $\mathrm{loss} = (\hat{y} - y)^2$ 予測モデル (特に意味はないToy Example)

$$\hat{y} = f(x_1, x_2, \theta_1, \theta_2, \theta_3)$$
$$= \underbrace{\theta_1 x_1}_{z_4} - \theta_2 \underbrace{\cos(\underbrace{\theta_3 + x_2}_{z_1})}_{z_2}$$
$$\underbrace{\qquad\qquad}_{z_3}$$



合成関数

$$z_1 = \theta_3 + x_2$$
$$z_2 = \cos(z_1)$$
$$z_3 = \theta_2 z_2$$
$$z_4 = \theta_1 x_1$$

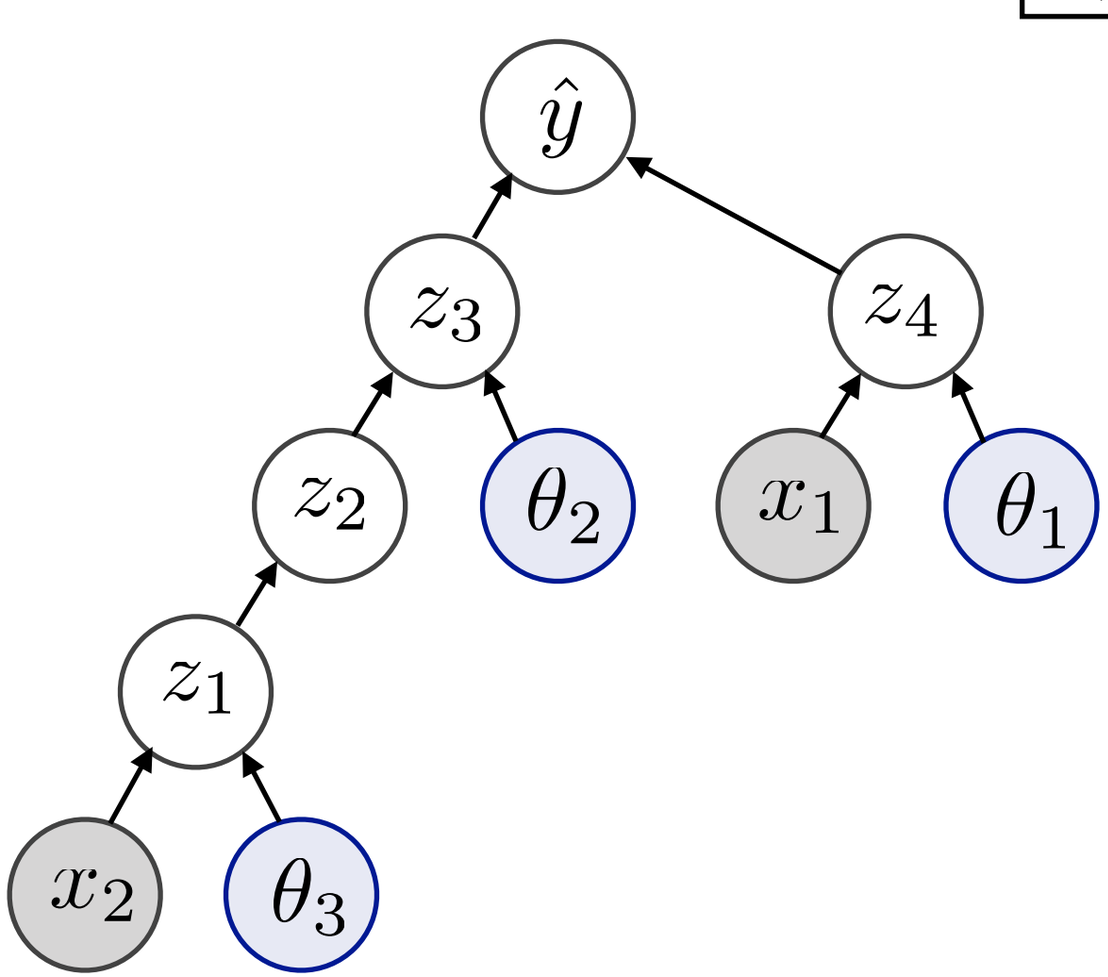
計算グラフ：素演算の合成関数のグラフ表現

例 $\text{loss} = (\hat{y} - y)^2$ 予測モデル (特に意味はないToy Example)

$$\hat{y} = f(x_1, x_2, \theta_1, \theta_2, \theta_3)$$
$$= \underbrace{\theta_1 x_1}_{z_4} - \theta_2 \cos(\underbrace{\theta_3 + x_2}_{z_1})$$

z_2

z_3



合成関数

$$z_1 = \theta_3 + x_2$$
$$z_2 = \cos(z_1)$$
$$z_3 = \theta_2 z_2$$
$$z_4 = \theta_1 x_1$$
$$\hat{y} = z_4 - z_3$$

計算グラフ：素演算の合成関数のグラフ表現

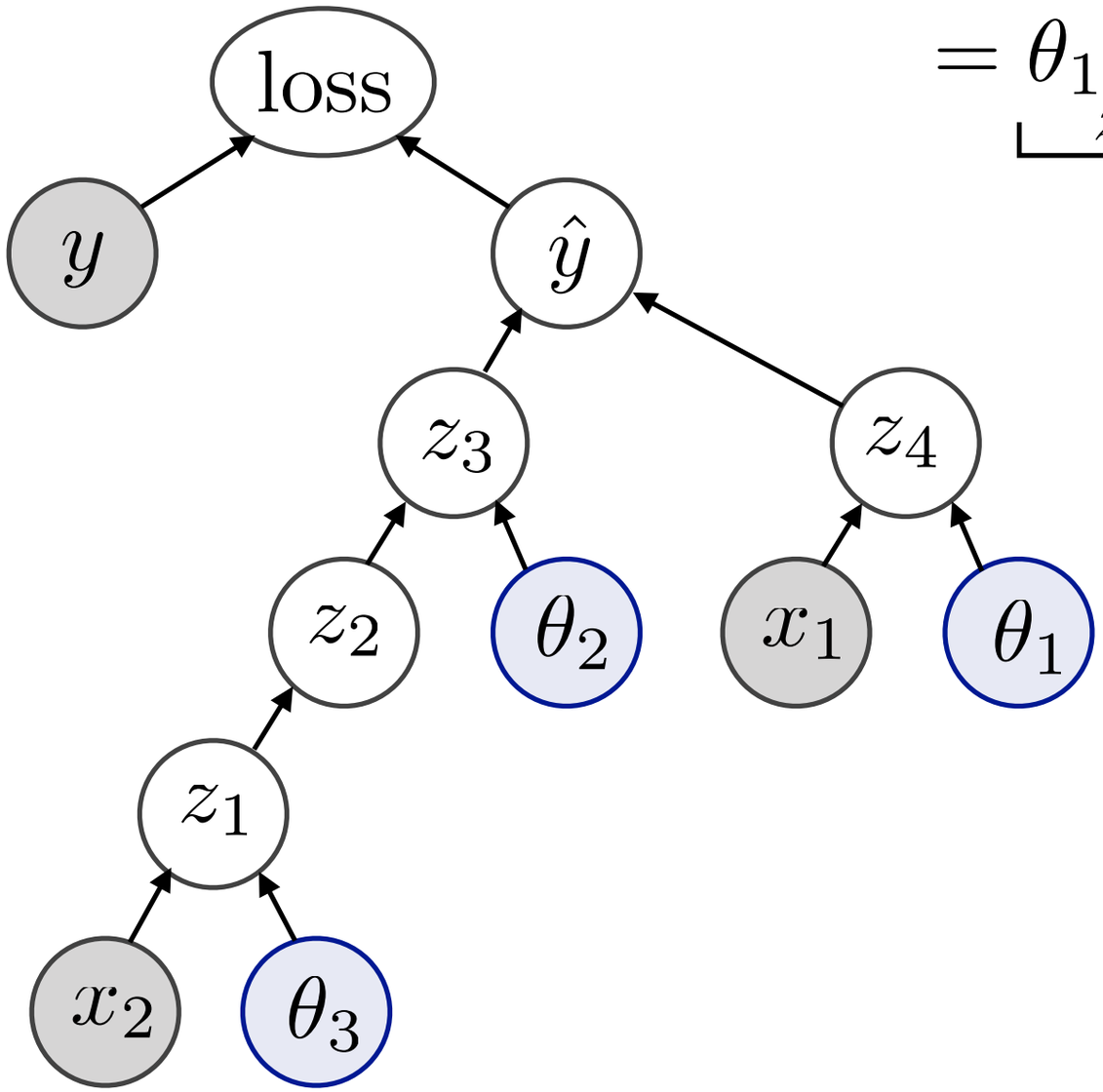
例 $\text{loss} = (\hat{y} - y)^2$ 予測モデル (特に意味はないToy Example)

$$\hat{y} = f(x_1, x_2, \theta_1, \theta_2, \theta_3)$$

$$= \underbrace{\theta_1 x_1}_{z_4} - \theta_2 \cos(\underbrace{\theta_3 + x_2}_{z_1})$$

z_2

z_3

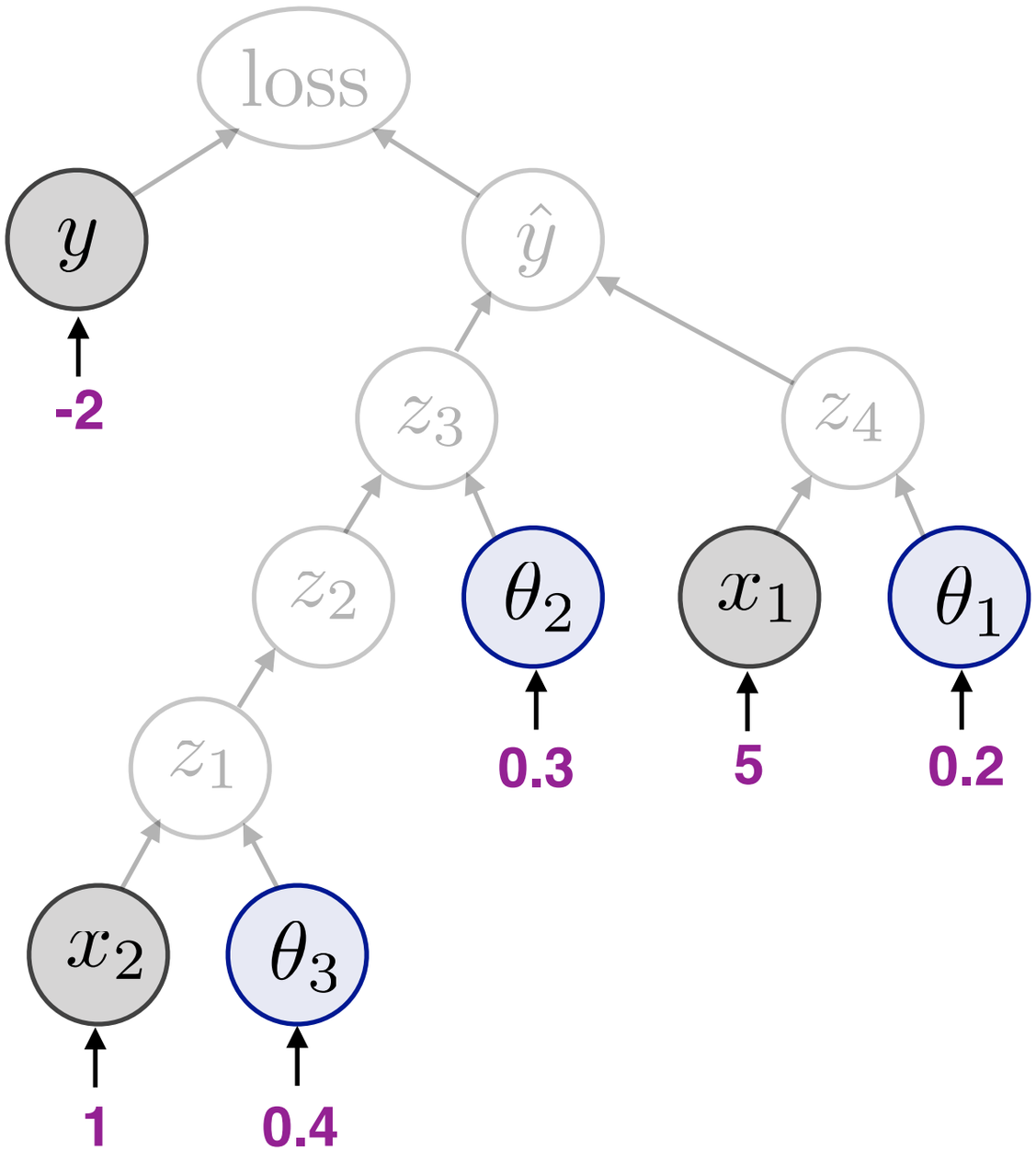


合成関数

$$z_1 = \theta_3 + x_2$$
$$z_2 = \cos(z_1)$$
$$z_3 = \theta_2 z_2$$
$$z_4 = \theta_1 x_1$$
$$\hat{y} = z_4 - z_3$$
$$\text{loss} = (\hat{y} - y)^2$$

forward: 計算グラフの変数に値をセットしてみる

このとき loss や $\hat{y}$ の値は？



$$\text{loss} = (\hat{y} - y)^2$$

$$\hat{y} = z_4 - z_3$$

$$z_4 = \theta_1 x_1$$

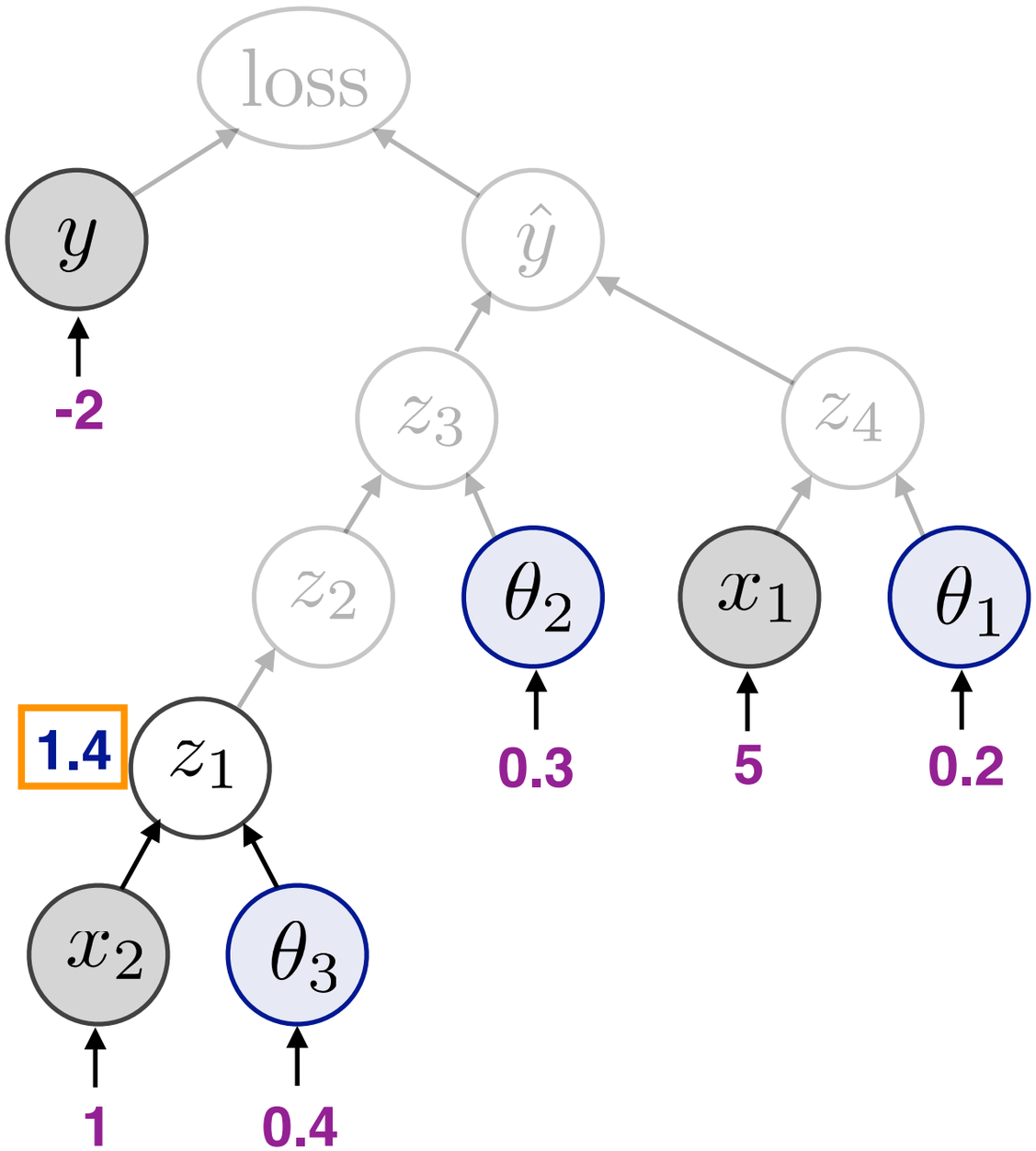
$$z_3 = \theta_2 z_2$$

$$z_2 = \cos(z_1)$$

$$z_1 = \theta_3 + x_2$$

forward: 計算グラフの変数に値をセットしてみる

このとき loss や $\hat{y}$ の値は？



$$\text{loss} = (\hat{y} - y)^2$$

$$\hat{y} = z_4 - z_3$$

$$z_4 = \theta_1 x_1$$

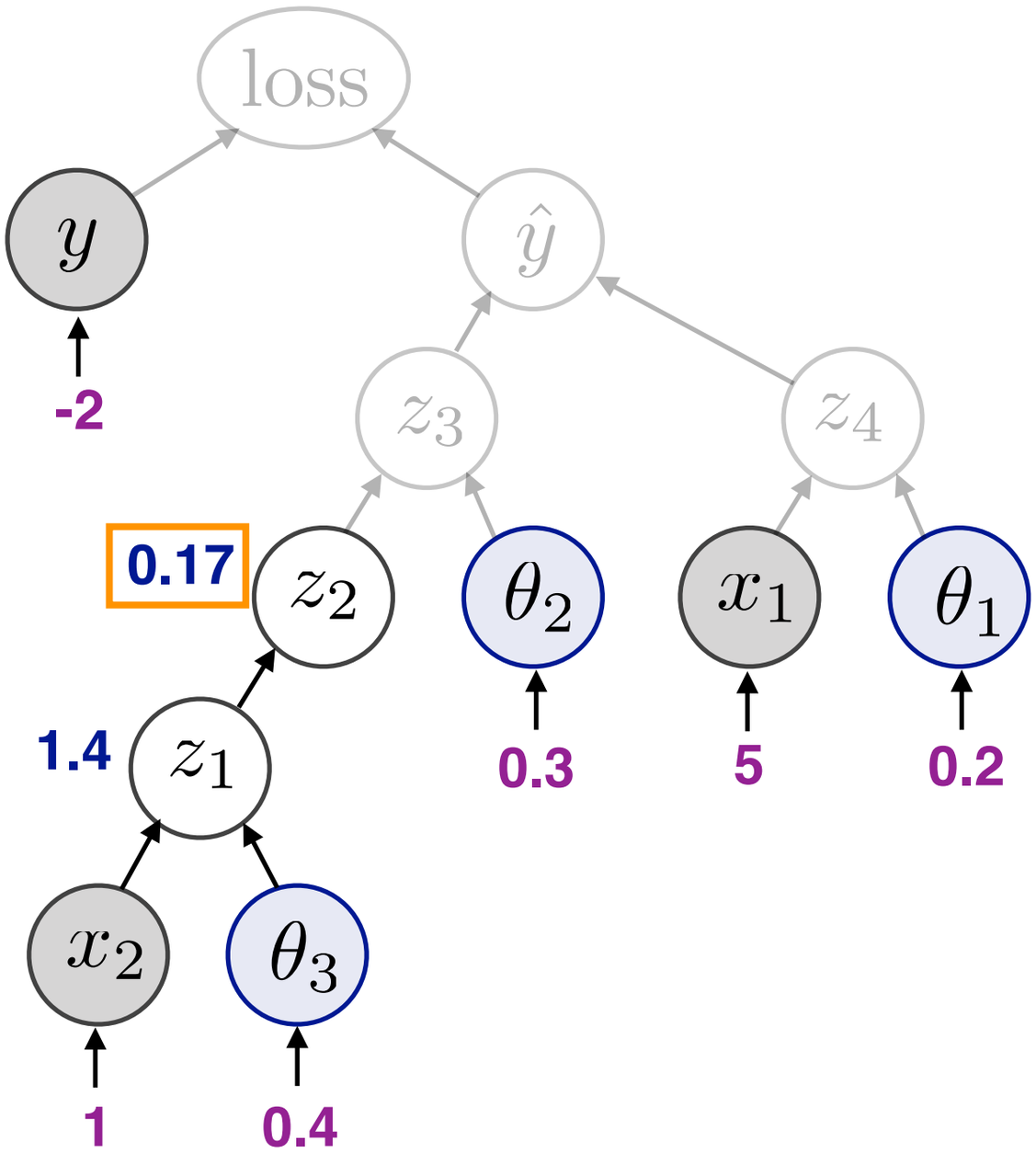
$$z_3 = \theta_2 z_2$$

$$z_2 = \cos(z_1)$$

$$z_1 = \theta_3 + x_2$$

forward: 計算グラフの変数に値をセットしてみる

このとき loss や $\hat{y}$ の値は？



$$\text{loss} = (\hat{y} - y)^2$$

$$\hat{y} = z_4 - z_3$$

$$z_4 = \theta_1 x_1$$

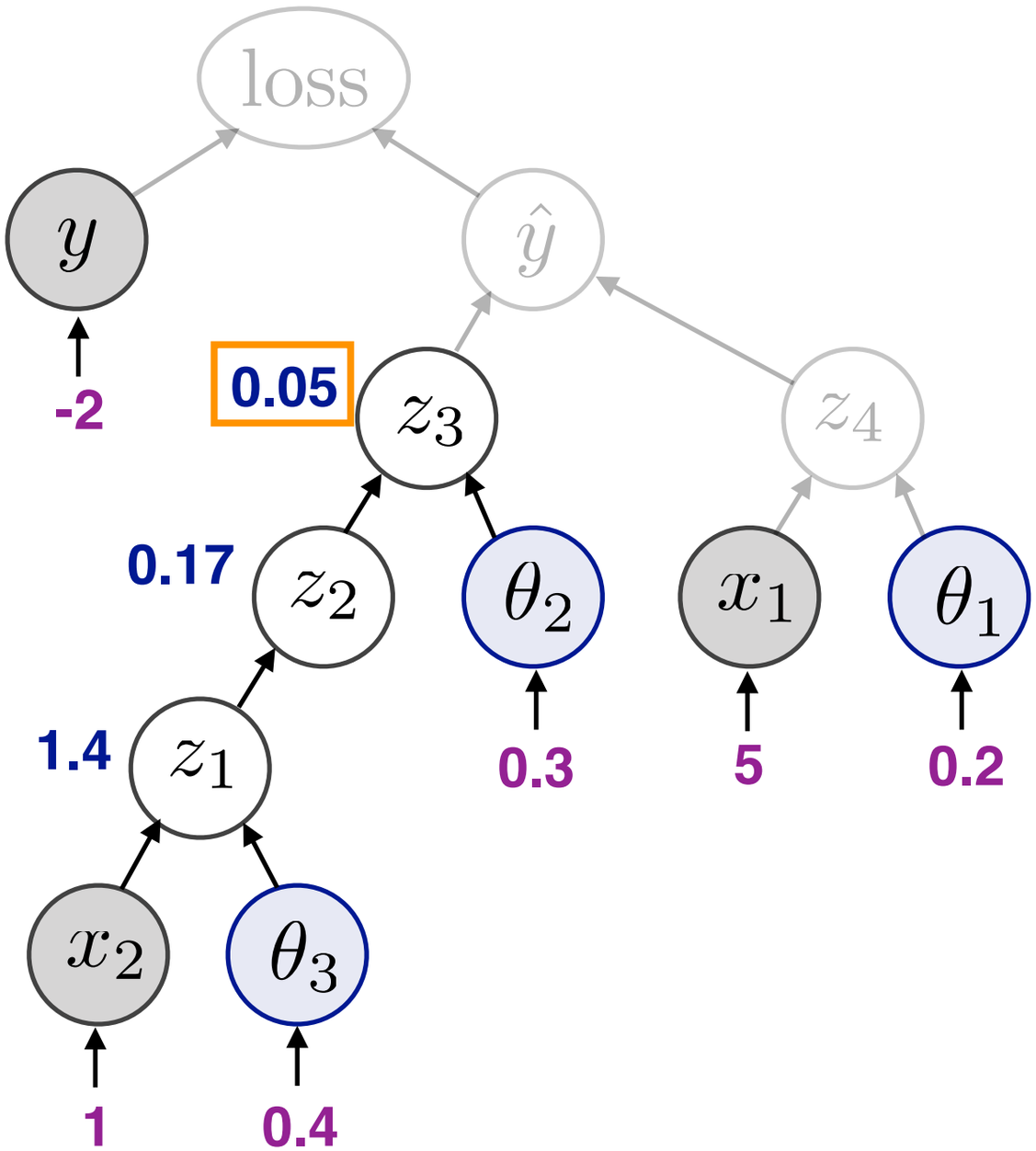
$$z_3 = \theta_2 z_2$$

$$z_2 = \cos(z_1)$$

$$z_1 = \theta_3 + x_2$$

forward: 計算グラフの変数に値をセットしてみる

このとき loss や $\hat{y}$ の値は？



$$\text{loss} = (\hat{y} - y)^2$$

$$\hat{y} = z_4 - z_3$$

$$z_4 = \theta_1 x_1$$

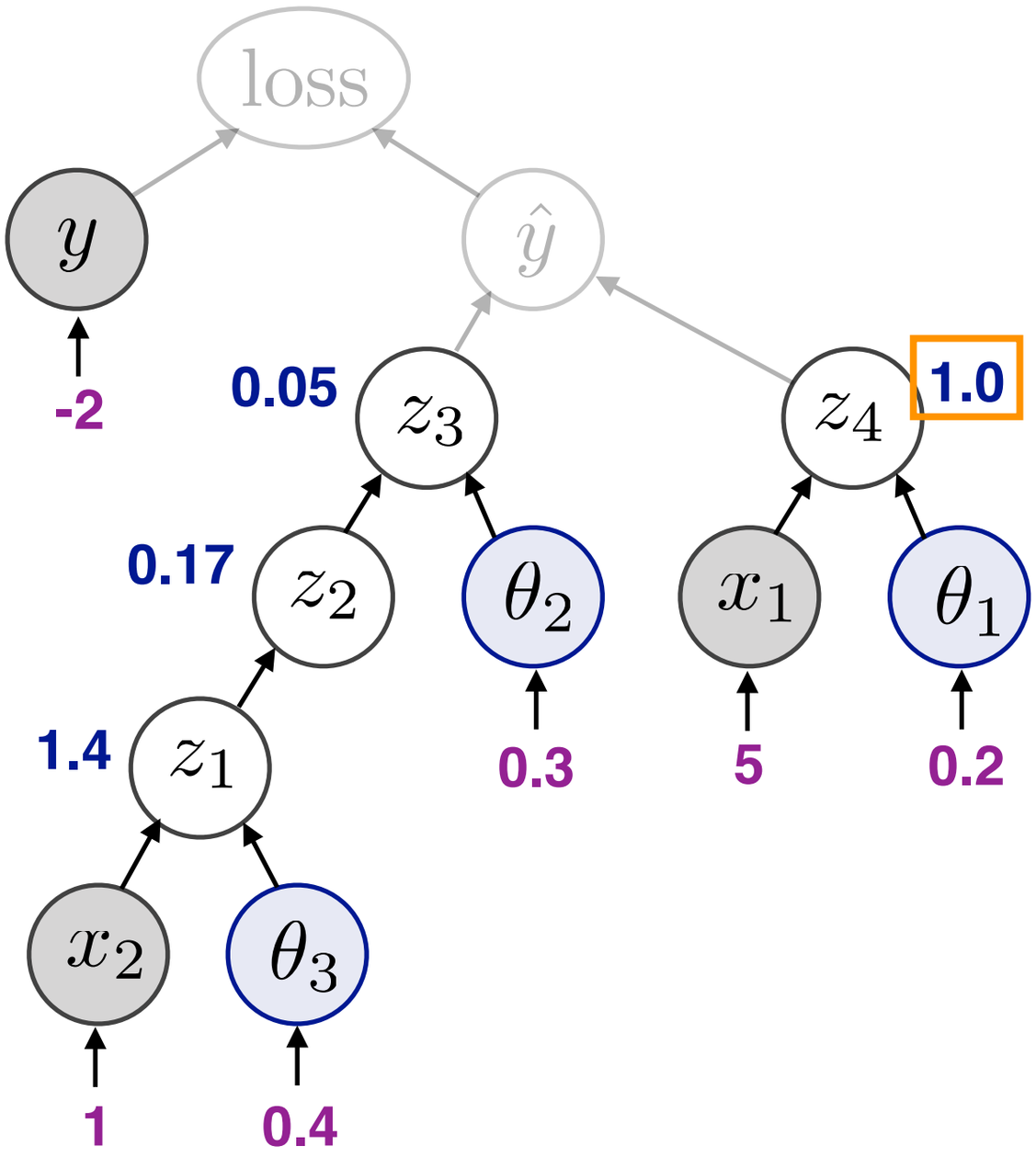
$$z_3 = \theta_2 z_2$$

$$z_2 = \cos(z_1)$$

$$z_1 = \theta_3 + x_2$$

forward: 計算グラフの変数に値をセットしてみる

このとき loss や $\hat{y}$ の値は？



$$\text{loss} = (\hat{y} - y)^2$$

$$\hat{y} = z_4 - z_3$$

$$z_4 = \theta_1 x_1$$

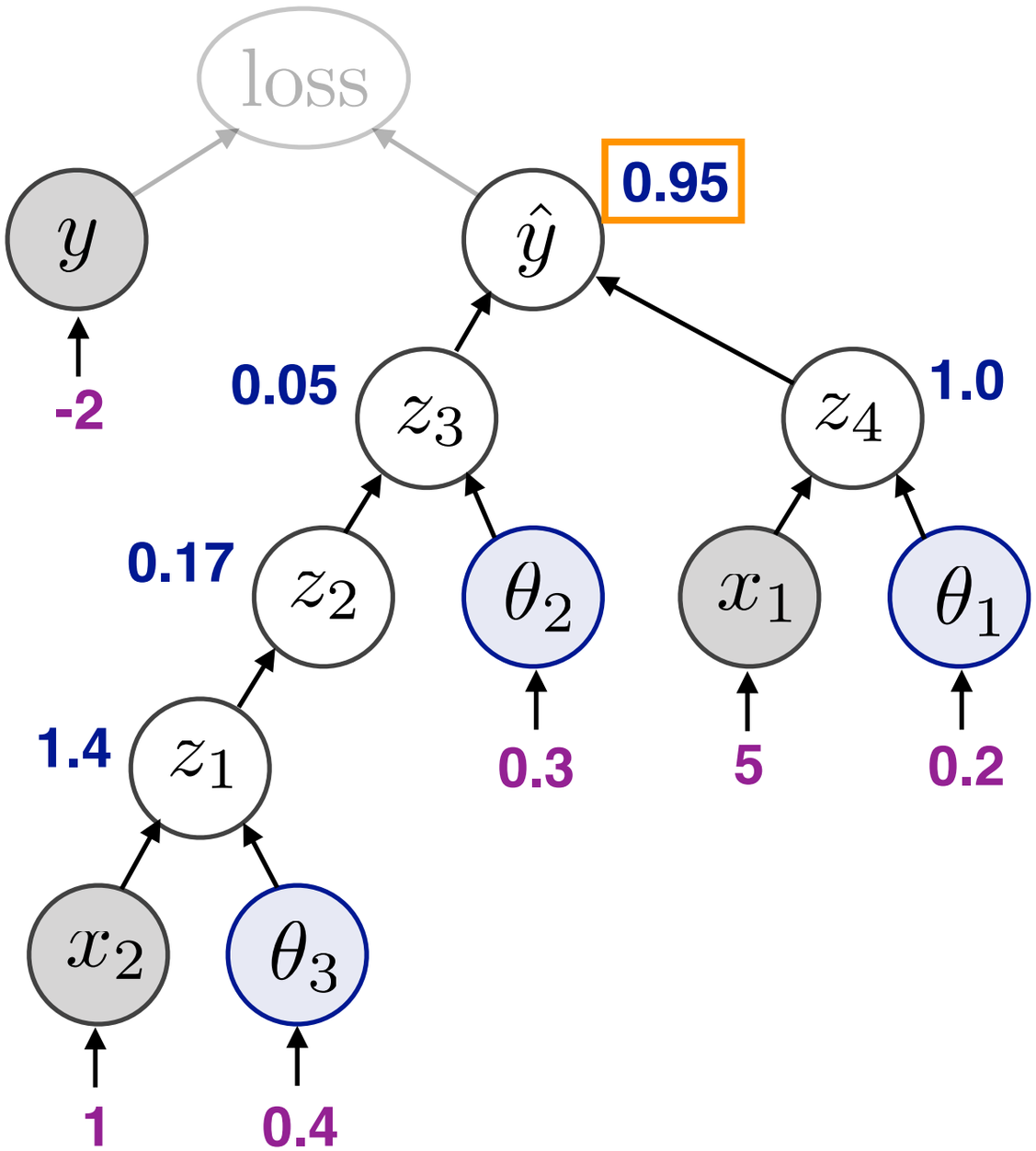
$$z_3 = \theta_2 z_2$$

$$z_2 = \cos(z_1)$$

$$z_1 = \theta_3 + x_2$$

forward: 計算グラフの変数に値をセットしてみる

このとき loss や $\hat{y}$ の値は？



$$\text{loss} = (\hat{y} - y)^2$$

$$\hat{y} = z_4 - z_3$$

$$z_4 = \theta_1 x_1$$

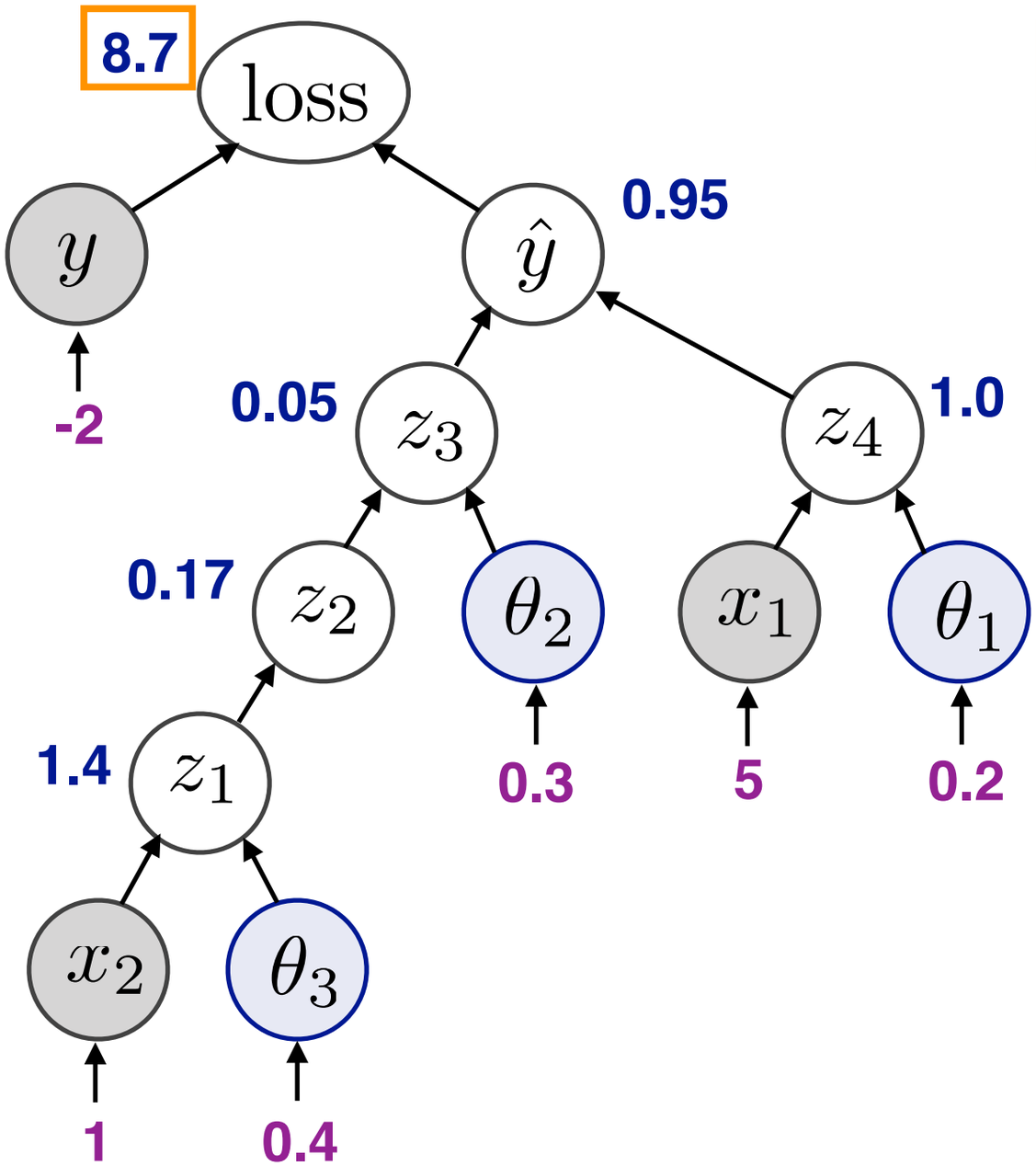
$$z_3 = \theta_2 z_2$$

$$z_2 = \cos(z_1)$$

$$z_1 = \theta_3 + x_2$$

forward: 計算グラフの変数に値をセットしてみる

このとき loss や $\hat{y}$ の値は？



$$\text{loss} = (\hat{y} - y)^2$$

$$\hat{y} = z_4 - z_3$$

$$z_4 = \theta_1 x_1$$

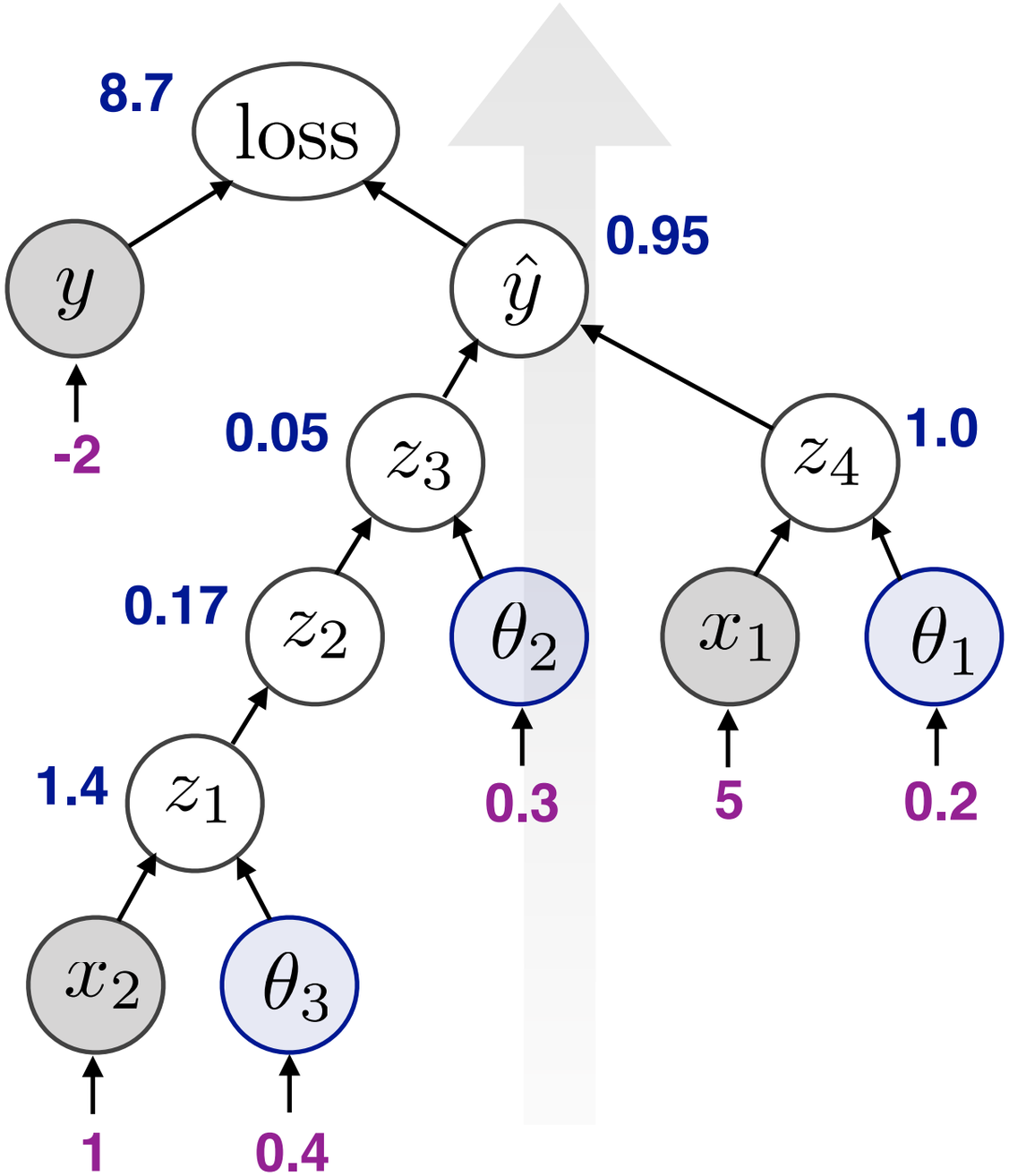
$$z_3 = \theta_2 z_2$$

$$z_2 = \cos(z_1)$$

$$z_1 = \theta_3 + x_2$$

forward

このとき loss や $\hat{y}$ の値は？



$$\text{loss} = (\hat{y} - y)^2$$

$$\hat{y} = z_4 - z_3$$

$$z_4 = \theta_1 x_1$$

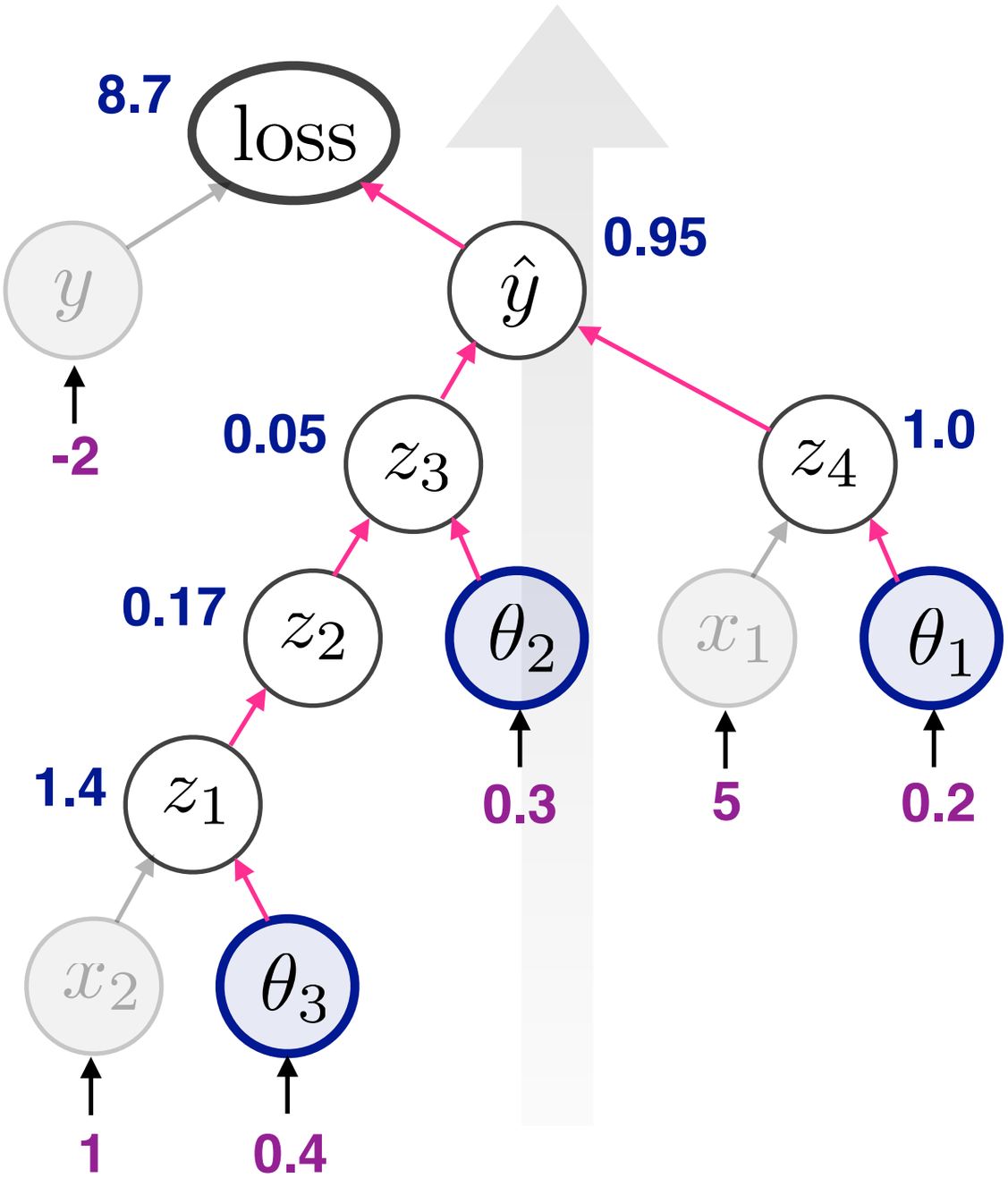
$$z_3 = \theta_2 z_2$$

$$z_2 = \cos(z_1)$$

$$z_1 = \theta_3 + x_2$$

forwardでついでにやること：偏微分値も計算

勾配を計算する変数はついでに**各素演算の偏微分値**も計算しておく



$$\text{loss} = (\hat{y} - y)^2$$

$$\hat{y} = z_4 - z_3$$

$$z_4 = \theta_1 x_1$$

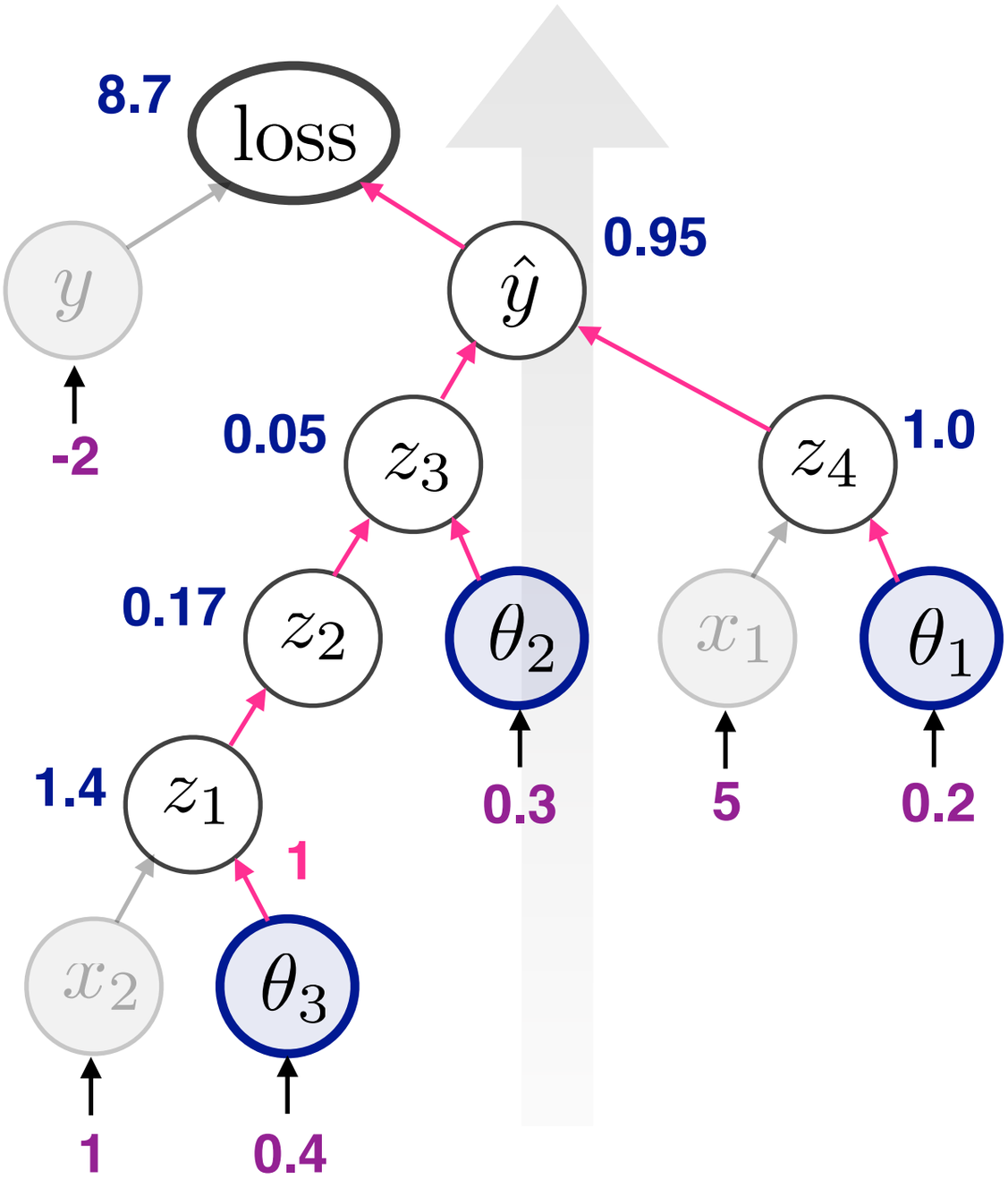
$$z_3 = \theta_2 z_2$$

$$z_2 = \cos(z_1)$$

$$z_1 = \theta_3 + x_2$$

forwardでついでにやること：偏微分値も計算

勾配を計算する変数はついでに**各素演算の偏微分値**も計算しておく



$$\mathrm{loss} = (\hat{y} - y)^2$$

$$\hat{y} = z_4 - z_3$$

$$z_4 = \theta_1 x_1$$

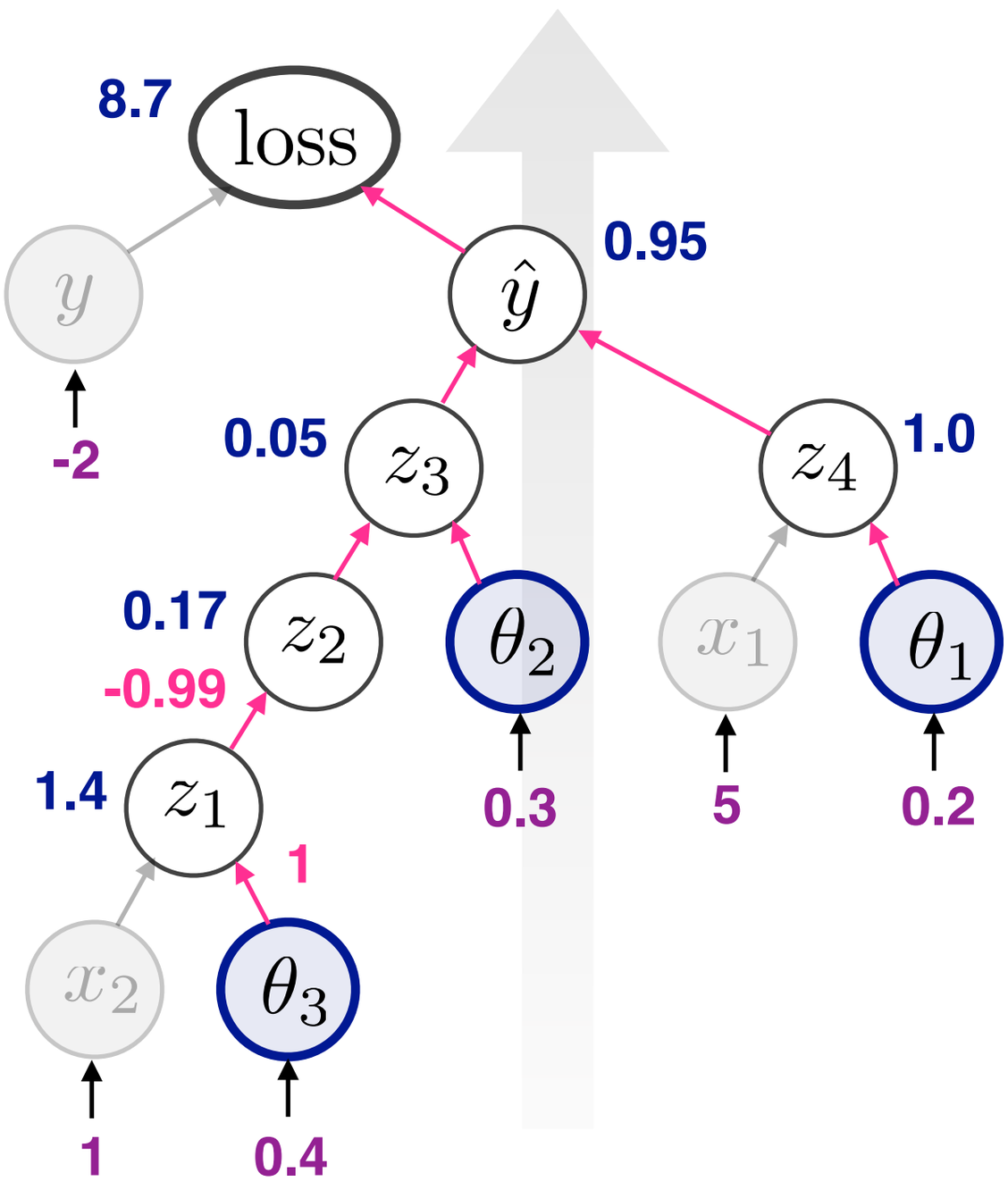
$$z_3 = \theta_2 z_2$$

$$z_2 = \cos(z_1)$$

$$z_1 = \theta_3 + x_2 \qquad \partial z_1 / \partial \theta_3 = 1$$

forwardでついでにやること：偏微分値も計算

勾配を計算する変数はついでに**各素演算の偏微分値**も計算しておく



$$\mathrm{loss} = (\hat{y} - y)^2$$

$$\hat{y} = z_4 - z_3$$

$$z_4 = \theta_1 x_1$$

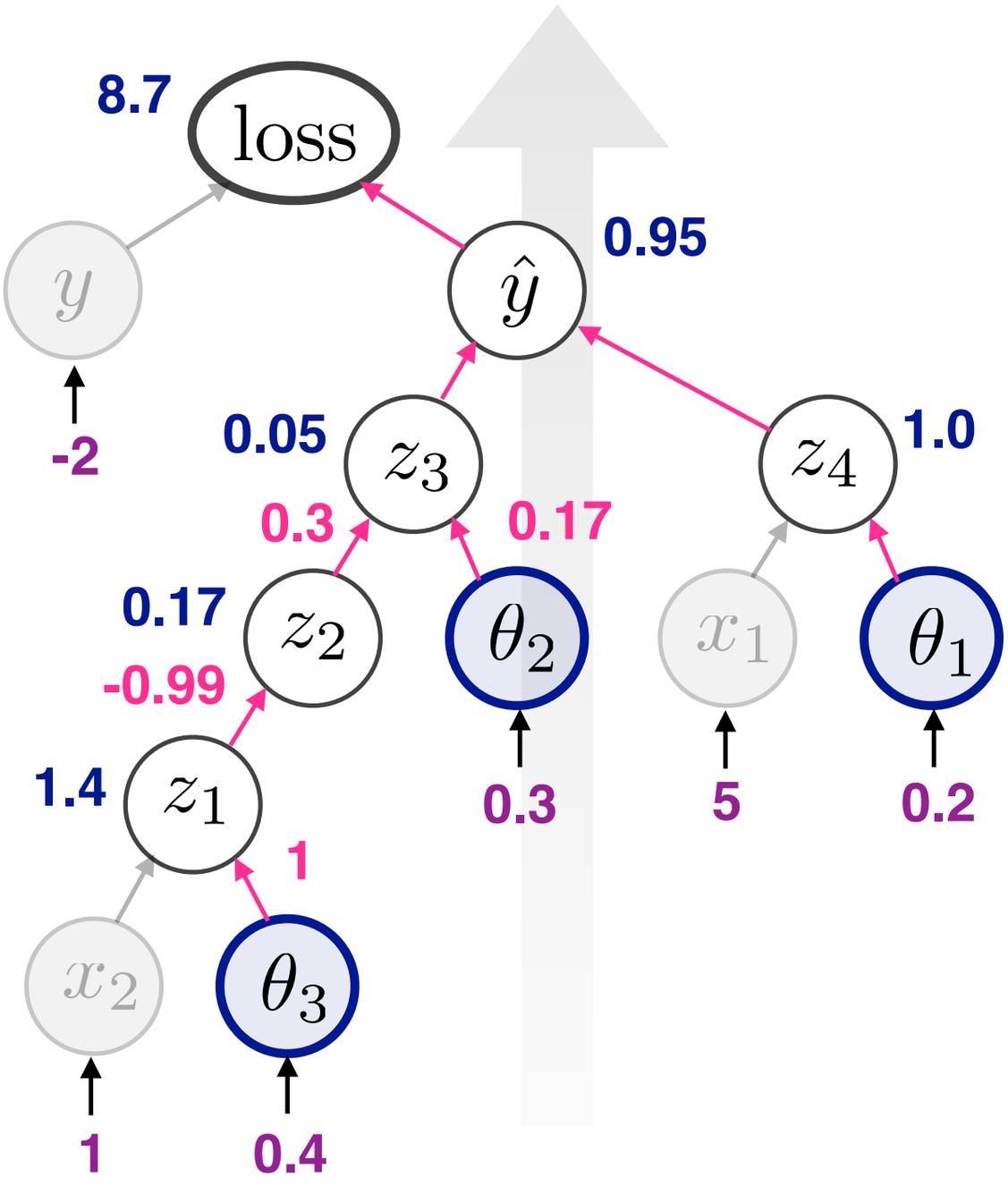
$$z_3 = \theta_2 z_2$$

$$z_2 = \cos(z_1) \qquad \partial z_2 / \partial z_1 = -\sin(z_1)$$

$$z_1 = \theta_3 + x_2 \qquad \partial z_1 / \partial \theta_3 = 1$$

forwardでついでにやること：偏微分値も計算

勾配を計算する変数はついでに**各素演算の偏微分値**も計算しておく



$$\mathrm{loss} = (\hat{y} - y)^2$$

$$\hat{y} = z_4 - z_3$$

$$z_4 = \theta_1 x_1$$

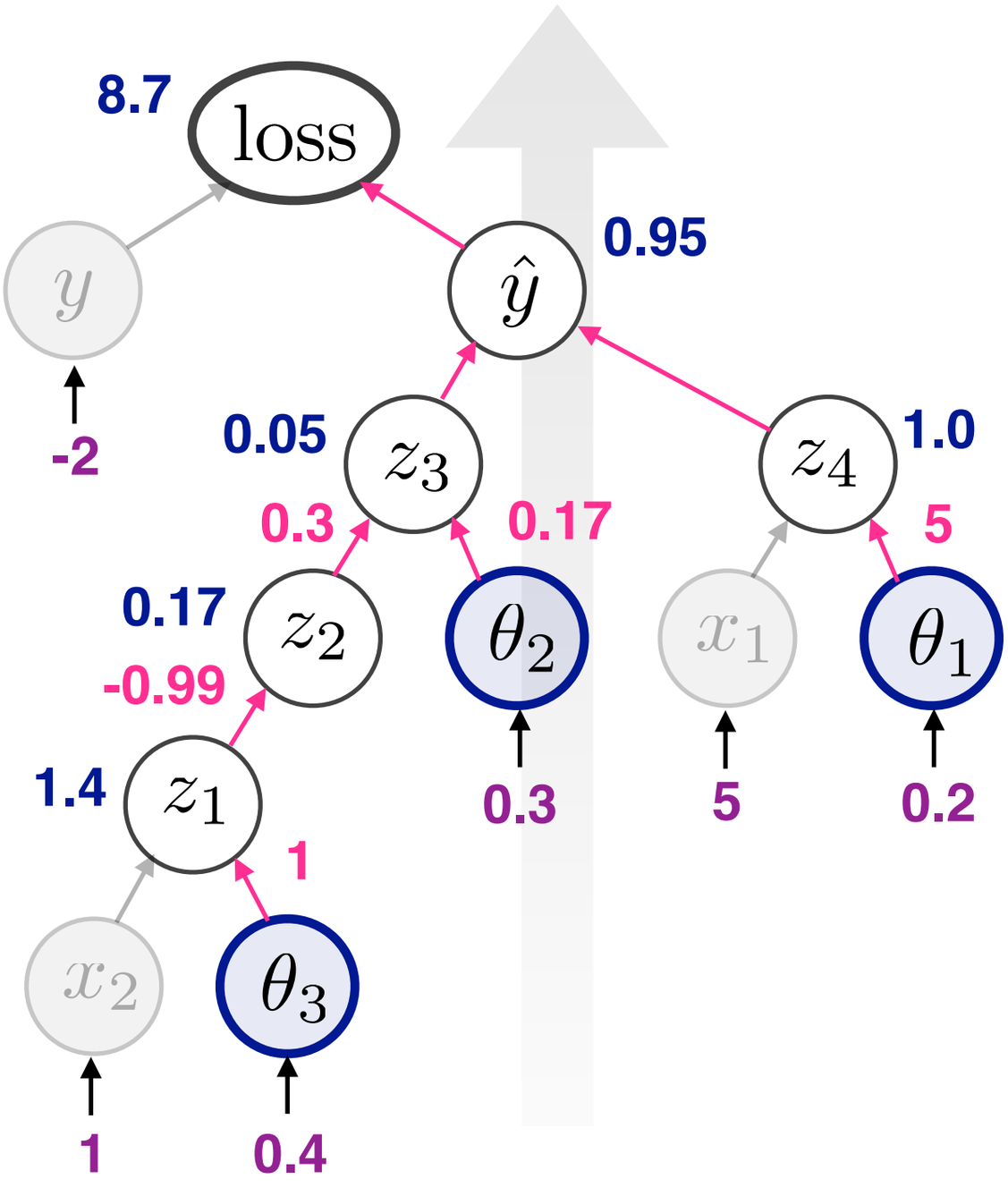
$$z_3 = \theta_2 z_2 \qquad \begin{cases} \partial z_3/\partial z_2 = \theta_2 \\ \partial z_3/\partial \theta_2 = z_2 \end{cases}$$

$$z_2 = \cos(z_1) \qquad \partial z_2/\partial z_1 = -\sin(z_1)$$

$$z_1 = \theta_3 + x_2 \qquad \partial z_1/\partial \theta_3 = 1$$

forwardでついでにやること：偏微分値も計算

勾配を計算する変数はついでに**各素演算の偏微分値**も計算しておく



$$\mathrm{loss} = (\hat{y} - y)^2$$

$$\hat{y} = z_4 - z_3$$

$$z_4 = \theta_1 x_1 \qquad \partial z_4/\partial \theta_1 = x_1$$

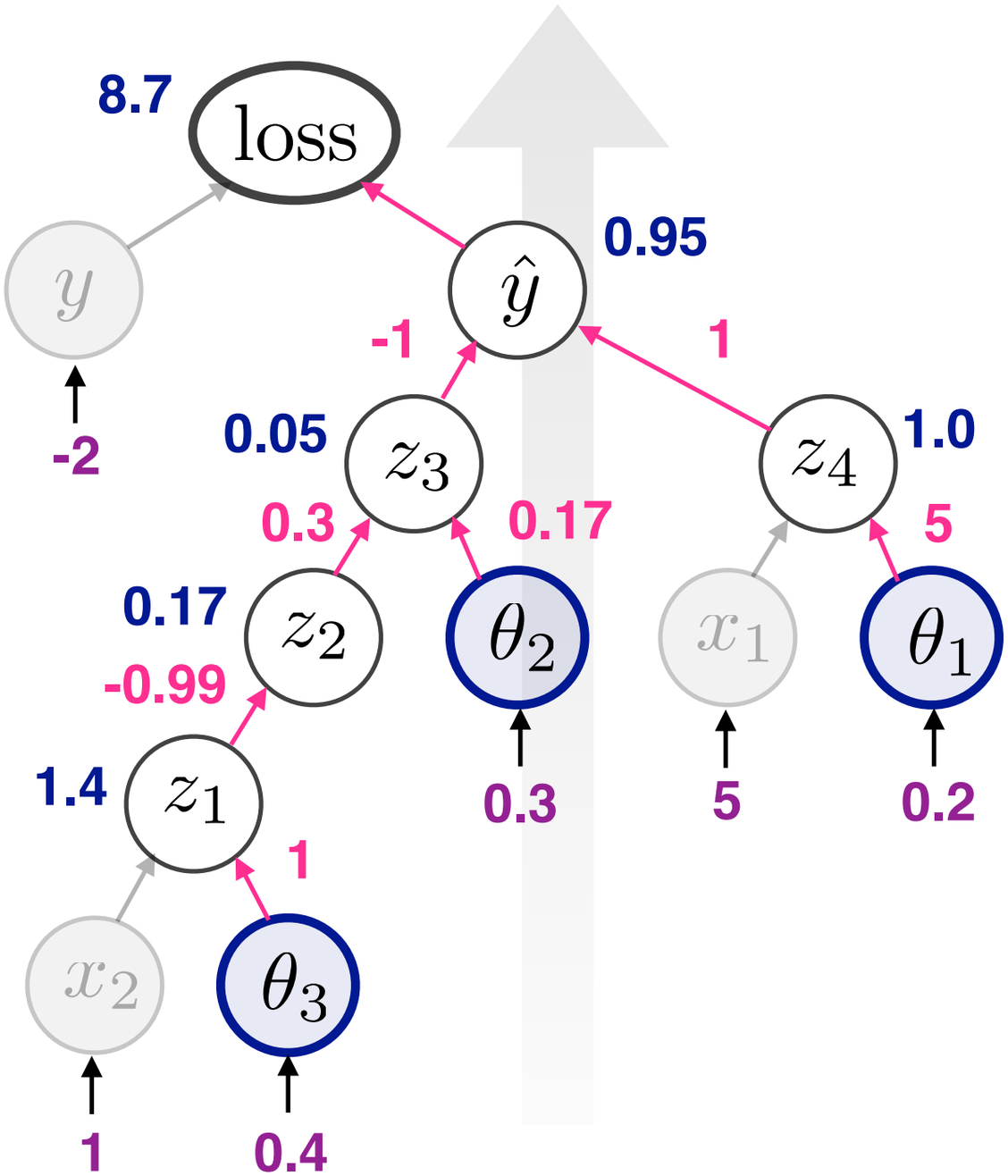
$$z_3 = \theta_2 z_2 \qquad \begin{cases} \partial z_3/\partial z_2 = \theta_2 \\ \partial z_3/\partial \theta_2 = z_2 \end{cases}$$

$$z_2 = \cos(z_1) \qquad \partial z_2/\partial z_1 = -\sin(z_1)$$

$$z_1 = \theta_3 + x_2 \qquad \partial z_1/\partial \theta_3 = 1$$

forwardでついでにやること：偏微分値も計算

勾配を計算する変数はついでに**各素演算の偏微分値**も計算しておく



$$\text{loss} = (\hat{y} - y)^2$$

$$\hat{y} = z_4 - z_3 \quad \begin{cases} \partial\hat{y}/\partial z_3 = -1 \\ \partial\hat{y}/\partial z_4 = 1 \end{cases}$$

$$z_4 = \theta_1 x_1 \quad \partial z_4/\partial\theta_1 = x_1$$

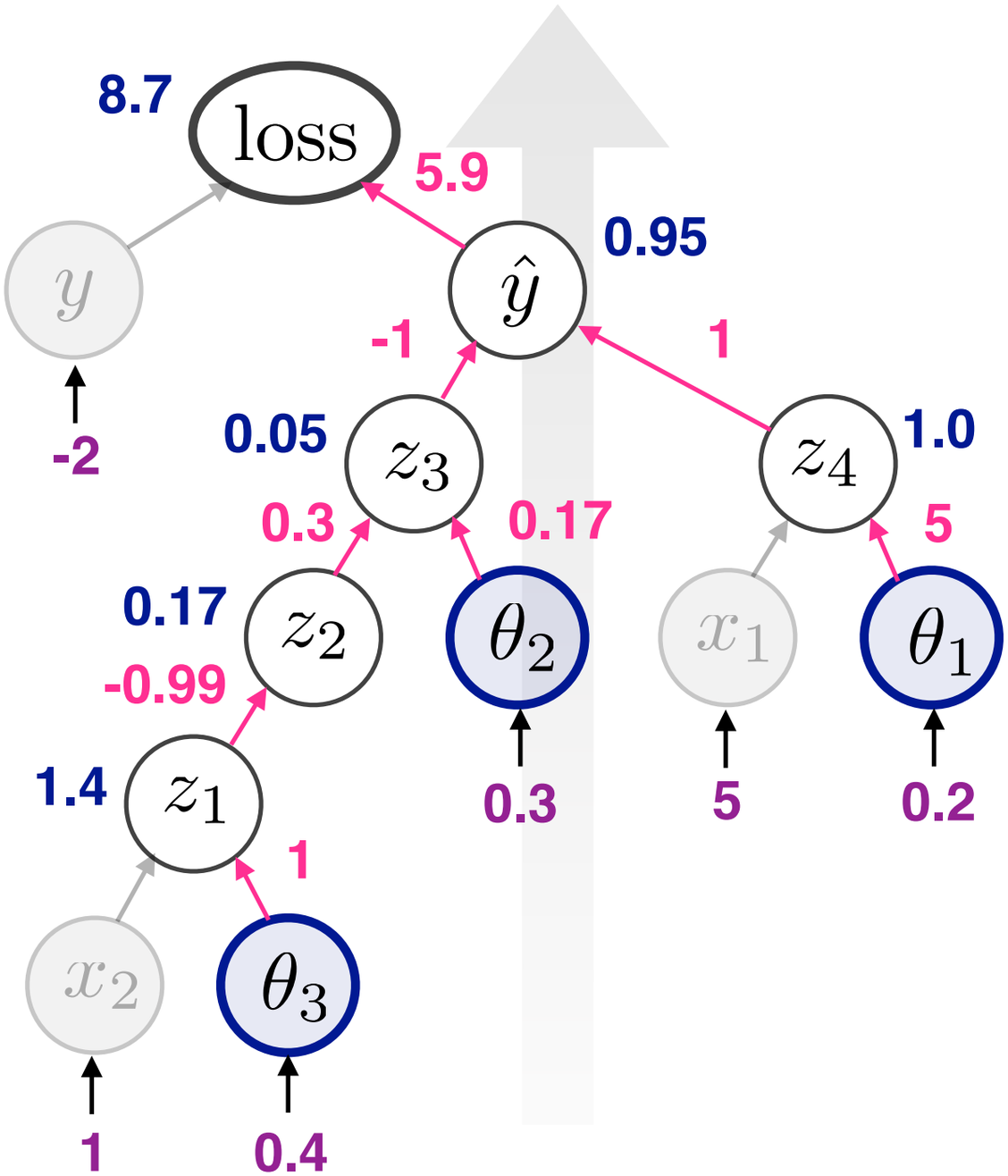
$$z_3 = \theta_2 z_2 \quad \begin{cases} \partial z_3/\partial z_2 = \theta_2 \\ \partial z_3/\partial\theta_2 = z_2 \end{cases}$$

$$z_2 = \cos(z_1) \quad \partial z_2/\partial z_1 = -\sin(z_1)$$

$$z_1 = \theta_3 + x_2 \quad \partial z_1/\partial\theta_3 = 1$$

forwardでついでにやること：偏微分値も計算

勾配を計算する変数はついでに**各素演算の偏微分値**も計算しておく



$$\text{loss} = (\hat{y} - y)^2 \quad \partial\text{loss}/\partial\hat{y} = 2(\hat{y} - y)$$

$$\hat{y} = z_4 - z_3 \quad \begin{cases} \partial\hat{y}/\partial z_3 = -1 \\ \partial\hat{y}/\partial z_4 = 1 \end{cases}$$

$$z_4 = \theta_1 x_1 \quad \partial z_4/\partial\theta_1 = x_1$$

$$z_3 = \theta_2 z_2 \quad \begin{cases} \partial z_3/\partial z_2 = \theta_2 \\ \partial z_3/\partial\theta_2 = z_2 \end{cases}$$

$$z_2 = \cos(z_1) \quad \partial z_2/\partial z_1 = -\sin(z_1)$$

$$z_1 = \theta_3 + x_2 \quad \partial z_1/\partial\theta_3 = 1$$

forwardでついでにやること：偏微分値も計算

合成に使える**素演算**は予め決めてそれぞれの微分公式も持っておく

変数の合成に使える素演算の例

- **加減乗除**
  torch.add, torch.sub, torch.mul, torch.div, ..
- **三角関数**
  torch.sin, torch.cos, torch.tan, torch.asin,…
- **指数・対数・べき**
  torch.pow, torch.log, torch.square, torch.sqrt, ..
- **行列演算**
  torch.matmul, torch.dot, torch.inverse, torch.det, ..
- **操作**
  torch.sum, torch.mean, torch.max, torch.argmin, ..

https://pytorch.org/docs/stable/torch.html#math-operations

$$\text{loss} = (\hat{y} - y)^2 \quad \partial\text{loss}/\partial\hat{y} = 2(\hat{y} - y)$$

$$\hat{y} = z_4 - z_3 \quad \begin{cases} \partial\hat{y}/\partial z_3 = -1 \\ \partial\hat{y}/\partial z_4 = 1 \end{cases}$$

$$z_4 = \theta_1 x_1 \quad \partial z_4/\partial\theta_1 = x_1$$

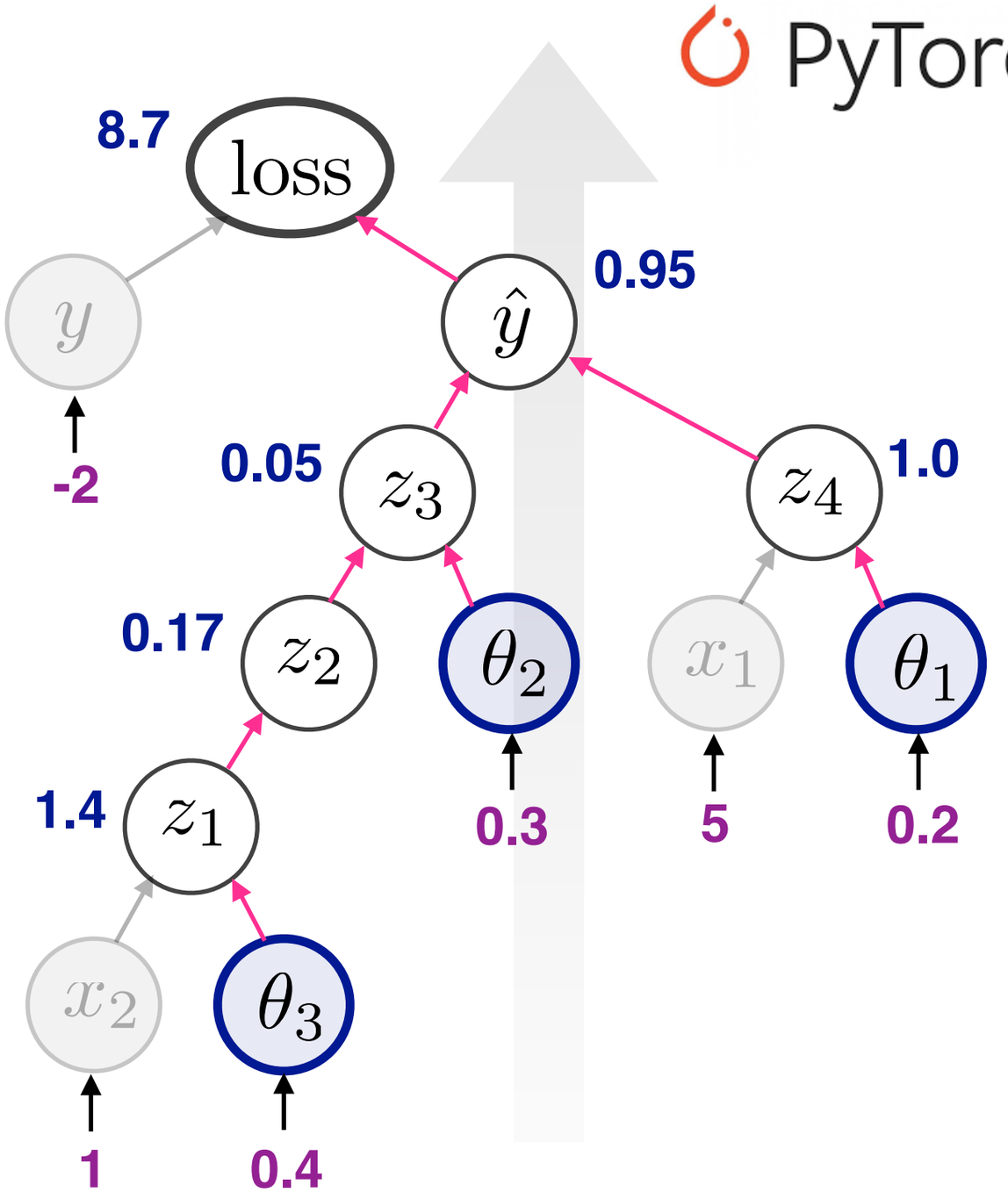
$$z_3 = \theta_2 z_2 \quad \begin{cases} \partial z_3/\partial z_2 = \theta_2 \\ \partial z_3/\partial\theta_2 = z_2 \end{cases}$$

$$z_2 = \cos(z_1) \quad \partial z_2/\partial z_1 = -\sin(z_1)$$

$$z_1 = \theta_3 + x_2 \quad \partial z_1/\partial\theta_3 = 1$$

PyTorch

forward



PyTorch

```
import torch
```

```
y = torch.tensor(-2)
x1 = torch.tensor(5)
x2 = torch.tensor(1)
theta1 = torch.tensor(0.2, requires_grad=True)
theta2 = torch.tensor(0.3, requires_grad=True)
theta3 = torch.tensor(0.4, requires_grad=True)
```

```
z1 = theta3 + x2
z2 = torch.cos(z1)
z3 = theta2 * z2
z4 = theta1 * x1
y_hat = z4 - z3

loss = (y_hat-y)**2
```

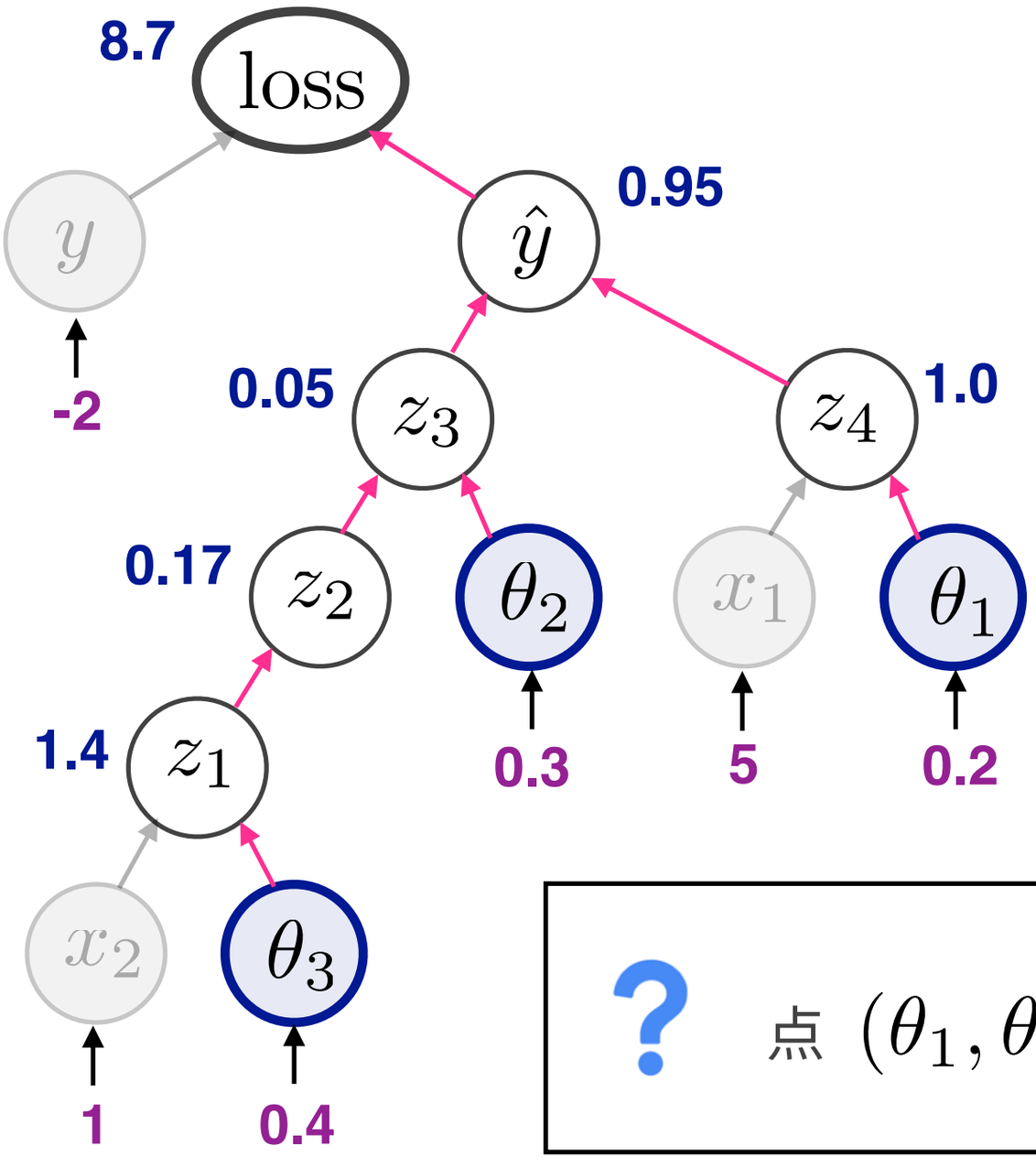
```
z1, z2, z3, z4, y_hat, loss
```

```
(tensor(1.4000, grad_fn=<AddBackward0>),
 tensor(0.1700, grad_fn=<CosBackward>),
 tensor(0.0510, grad_fn=<MulBackward0>),
 tensor(1., grad_fn=<MulBackward0>),
 tensor(0.9490, grad_fn=<SubBackward0>),
 tensor(8.6967, grad_fn=<PowBackward0>))
```

https://pytorch.org/tutorials/beginner/nlp/pytorch_tutorial.html

自動微分(リバースモード) "Back Propagation"

本当に計算したいのは**勾配を計算する変数**に関する**偏微分**



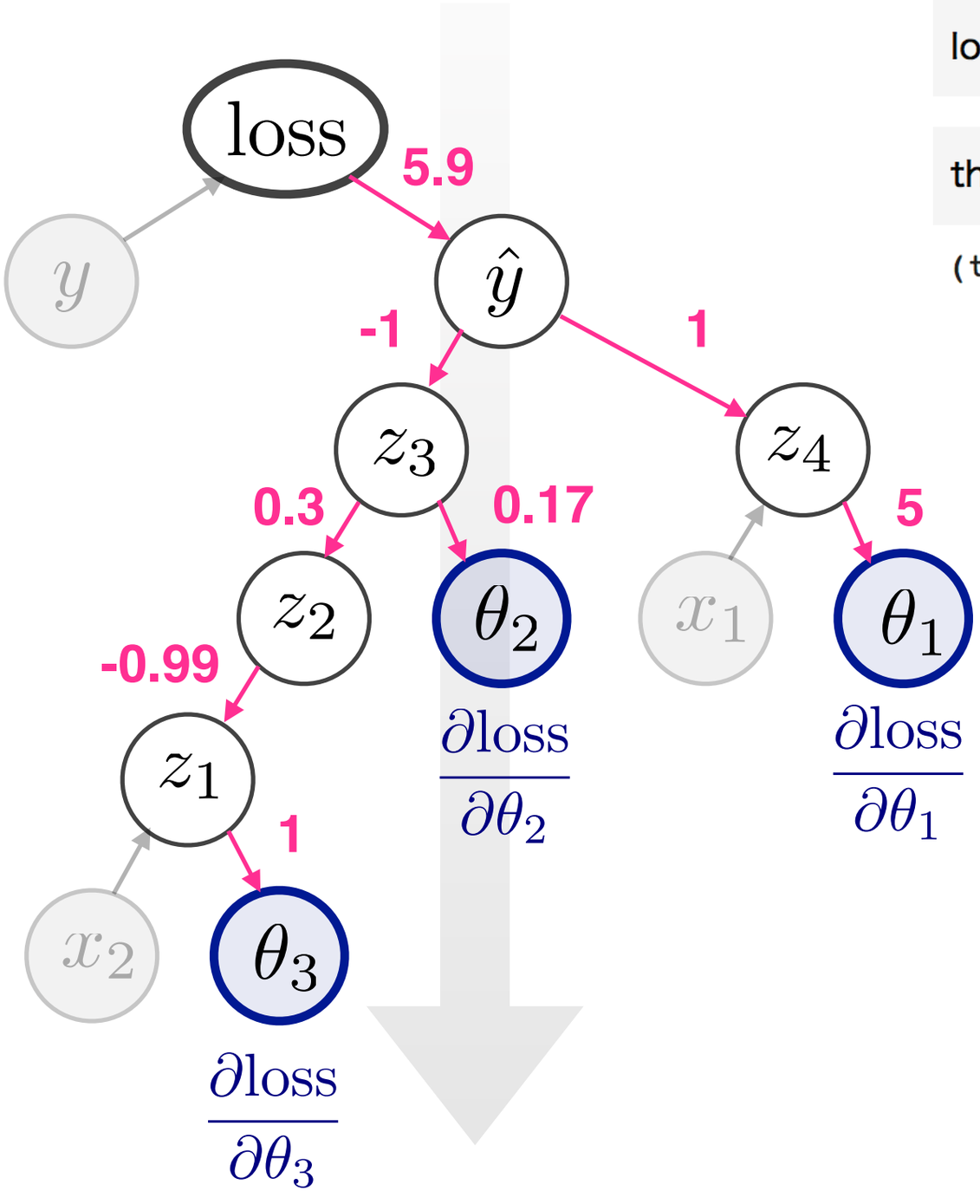
勾配ベクトル $= \begin{bmatrix} \frac{\partial \text{loss}}{\partial \theta_1} \\ \frac{\partial \text{loss}}{\partial \theta_2} \\ \frac{\partial \text{loss}}{\partial \theta_3} \end{bmatrix}$

$\theta_1, \theta_2, \theta_3$ を少し動かしたときの loss の変化量 **(勾配法が使いたい)**

? 点 $(\theta_1, \theta_2, \theta_3) = (0.2, 0.3, 0.4)$ での値は？

backward

合成関数の微分の連鎖律より、計算グラフを使って簡単に計算できる！




```
loss.backward()
```

```
theta1.grad, theta2.grad, theta3.grad
```

```
(tensor(29.4901), tensor(-1.0025), tensor(1.7437))
```

$$\frac{\partial \text{loss}}{\partial \theta_1} = \frac{\partial \text{loss}}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z_4} \times \frac{\partial z_4}{\partial \theta_1}$$

5.9 × 1 × 5 = **29.5**

$$\frac{\partial \text{loss}}{\partial \theta_2} = \frac{\partial \text{loss}}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z_3} \times \frac{\partial z_3}{\partial \theta_2}$$

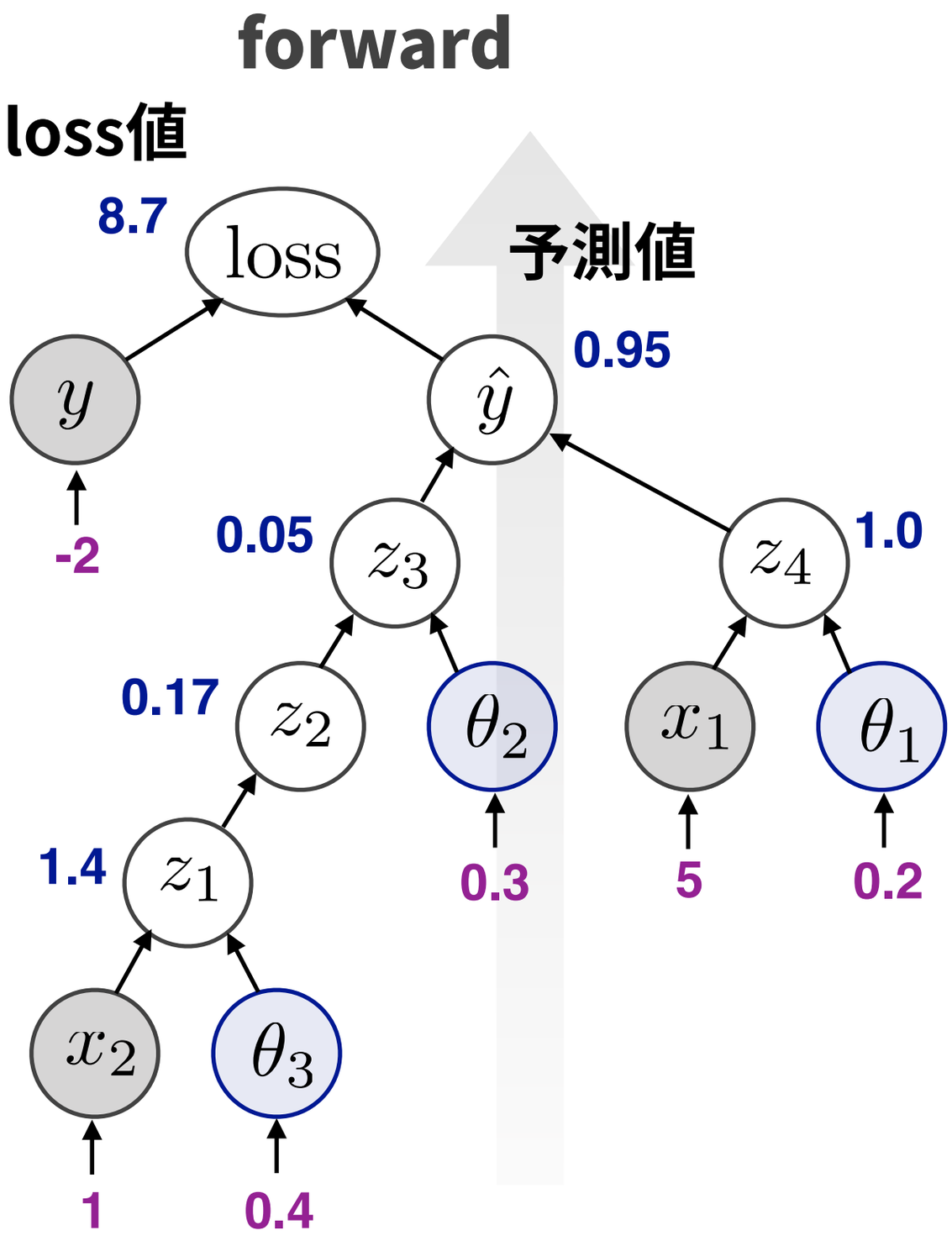
5.9 × -1 × 0.17 = **-1.00**

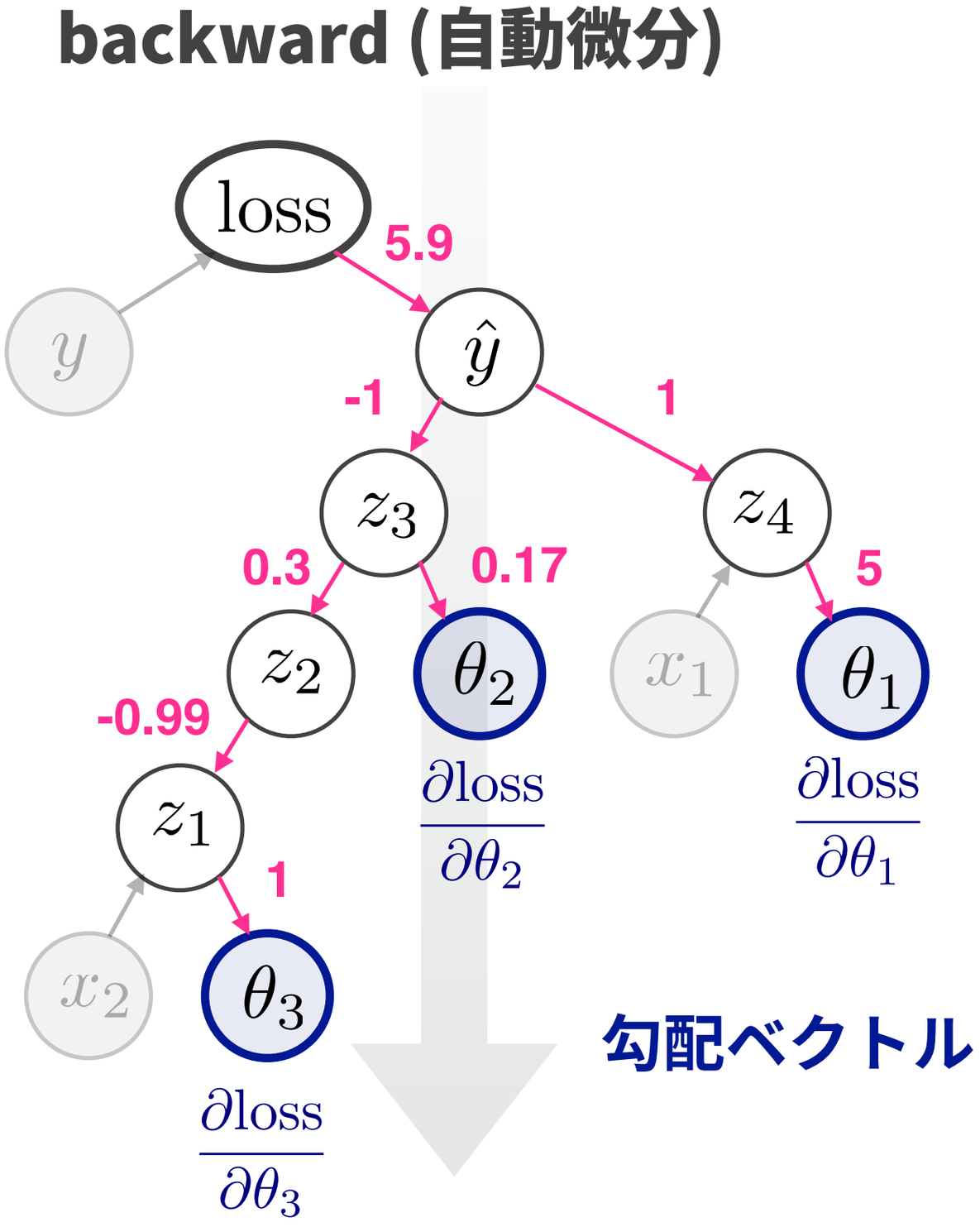
$$\frac{\partial \text{loss}}{\partial \theta_3} = \frac{\partial \text{loss}}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z_3} \times \frac{\partial z_3}{\partial z_2} \times \frac{\partial z_2}{\partial z_1} \times \frac{\partial z_1}{\partial \theta_3}$$

5.9 × -1 × 0.3 × -0.99 × 1

= **1.75**

プログラムとは究極的には素演算の何らかの合成関数！





深層学習のココロ："微分可能な"プログラミング

プログラムの不確定な箇所はパラメタにしておき後で機械学習

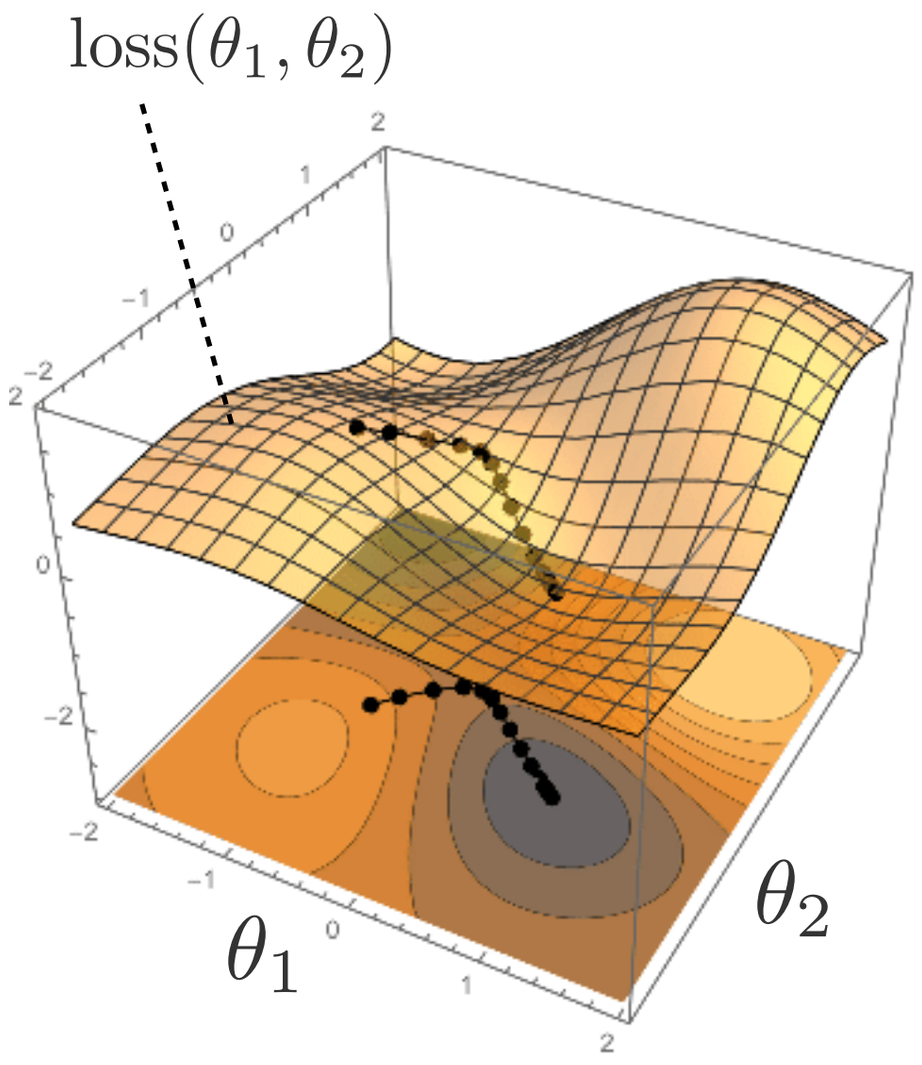
どんなプログラムも素演算の合成関数(計算グラフ)になってさえいれば自動微分＋勾配降下法でパラメタを最適にできる！

$$\begin{bmatrix}\theta_1\\ \theta_2\\ \vdots\\ \theta_p\end{bmatrix} \leftarrow \begin{bmatrix}\theta_1\\ \theta_2\\ \vdots\\ \theta_p\end{bmatrix} - \eta \times \begin{bmatrix}\partial \mathrm{loss}(\boldsymbol{\theta})/\partial\theta_1\\ \partial \mathrm{loss}(\boldsymbol{\theta})/\partial\theta_2\\ \vdots\\ \partial \mathrm{loss}(\boldsymbol{\theta})/\partial\theta_p\end{bmatrix}$$

学習率
(step size)

勾配ベクトル

自動微分で計算



4

Pythonで作って理解しよう

- PyTorchとtorch.autograd
- micrograd
- tinygrad
- JAX

PyTorchとtorch.autograd

PyTorch

https://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html

Run in Google Colab | Download Notebook | View on GitHub

A GENTLE INTRODUCTION TO `TORCH.AUTOGRAD`

`torch.autograd` is PyTorch's automatic differentiation engine that powers neural network training. In this section, you will get a conceptual understanding of how autograd helps a neural network train.

Background

Neural networks (NNs) are a collection of nested functions that are executed on some input data. These functions are defined by *parameters* (consisting of weights and biases), which in PyTorch are stored in tensors.

Training a NN happens in two steps:

Forward Propagation: In forward prop, the NN makes its best guess about the correct output. It runs the input data through each of its functions to make this guess.

Backward Propagation: In backprop, the NN adjusts its parameters proportionate to the error in its guess. It does this by traversing backwards from the output, collecting the derivatives of the error with respect to the parameters of the functions (*gradients*), and optimizing the parameters using gradient descent. For a more detailed walkthrough of backprop, check out this video from 3Blue1Brown.

PyTorchとtorch.autograd

PyTorch

```
import torch

a = torch.tensor(2., requires_grad=True)
b = torch.tensor(6., requires_grad=True)
Q = 3*a**3 - b**2
```

$$Q = 3a^3 - b^2$$

これがforward
(関数定義しながら計算グラフが作られる)

```
Q
```

```
tensor(-12., grad_fn=<SubBackward0>)
```

```
Q.backward()
```

backwardでは計算グラフの葉ノードの
変数まで偏微分値(.grad)を伝播計算
aka “back propagation”

```
a.grad, b.grad
```

```
(tensor(36.), tensor(-12.))
```

```
print(9*a**2 == a.grad)
print(-2*b == b.grad)
```

$$\frac{\partial Q}{\partial a} = 9a^2 \quad \frac{\partial Q}{\partial b} = -2b$$

```
tensor(True)
tensor(True)
```

torch.no_grad

PyTorch

```
%%time
a = torch.tensor(2.)
b = torch.tensor(6.)
print(3*a**3 - b**2)
```

```
tensor(-12.)
CPU times: user 1.78 ms, sys: 0 ns, total: 1.78 ms
Wall time: 1.86 ms
```

requires=Trueな変数がなければ
torch.tensorはnumpy.arrayと
ほぼ変わらない

```
%%time
a = torch.tensor(2., requires_grad=True)
b = torch.tensor(6., requires_grad=True)
print(3*a**3 - b**2)
```

```
tensor(-12., grad_fn=<SubBackward0>)
CPU times: user 1.29 ms, sys: 0 ns, total: 1.29 ms
Wall time: 3.24 ms
```

requires=Trueな変数があると
あとでbackward()される可能性が
あるため計算グラフを作ってしまう

```
%%time
a = torch.tensor(2., requires_grad=True)
b = torch.tensor(6., requires_grad=True)
with torch.no_grad():
  print(3*a**3 - b**2)
```

```
tensor(-12.)
CPU times: user 543 µs, sys: 770 µs, total: 1.31 ms
Wall time: 1.33 ms
```

例えば「学習済みのモデルで予測値を
計算したいだけ」など、
あとでbackward()せず、ただ値を
計算したいだけの場合は
torch.no_grad()ブロックで行う

backward vs torch.autograd

PyTorch

```
import torch

a = torch.tensor(2., requires_grad=True)
b = torch.tensor(6., requires_grad=True)
Q = 3*a**3 - b**2

Q.backward()
print(a.grad, b.grad)
```

```
tensor(36.) tensor(-12.)
```

```
a = torch.tensor(2., requires_grad=True)
b = torch.tensor(6., requires_grad=True)
Q = 3*a**3 - b**2

a_grad = torch.autograd.grad(Q, a)
b_grad = torch.autograd.grad(Q, b)

print(a_grad, b_grad)
```

```
(tensor(36.),) (tensor(-12.),)
```

backward()

計算グラフの葉ノードの変数まで
偏微分値を計算し、その変数vの
v.gradフィールドに加える

torch.autograd.grad

与えたoutからinまでだけ伝播させ
与えられた偏微分値を求める

(機械学習では陽にtorch.autogradを
呼び出すことは少ないが、同様に
hessianやjacobianの計算が可能)

backward()は各変数の.gradに値を積算する

PyTorch

リセットありで2回backward()

```
a = torch.tensor(2., requires_grad=True)
b = torch.tensor(6., requires_grad=True)

Q1 = 3*a**3 - b**2
Q1.backward()
print(a.grad, b.grad, (9*a**2).data, (-2*b).data)

a.grad = torch.tensor(0.0)
b.grad = torch.tensor(0.0)

Q2 = -4*a**2 - b**4
Q2.backward()
print(a.grad, b.grad, (-8*a).data, (-4*b**3).data)
```

リセット

```
tensor(36.) tensor(-12.) tensor(36.) tensor(-12.)
tensor(-16.) tensor(-864.) tensor(-16.) tensor(-864.)
```

a.grad: 36, -16

b.grad: -12, -864

リセットなしで2回backward()

```
a = torch.tensor(2., requires_grad=True)
b = torch.tensor(6., requires_grad=True)

Q1 = 3*a**3 - b**2
Q1.backward()
print(a.grad, b.grad, (9*a**2).data, (-2*b).data)

Q2 = -4*a**2 - b**4
Q2.backward()
print(a.grad, b.grad, (-8*a).data, (-4*b**3).data)
```

```
tensor(36.) tensor(-12.) tensor(36.) tensor(-12.)
tensor(20.) tensor(-876.) tensor(-16.) tensor(-864.)
```

どんどん足し込まれていく

a.grad = 20 = 36 + (-16)

b.grad = -876 = -12 + (-864)

機械学習ではこの挙動は便利 (数値計算上は紛らわしいので注意)

多次元配列 torch.tensor

PyTorch

numpyなどの多次元配列の演算を扱う ＋「自動微分(autograd)」機構

https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html

```
x = torch.tensor([[1.0, 2.0], [-1.0, 0.0]])

print(f"Shape of tensor: {x.shape}")
print(f"Datatype of tensor: {x.dtype}")
print(f"Device tensor is stored on: {x.device}")
```

```
Shape of tensor: torch.Size([2, 2])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

```
x.requires_grad
```

```
False
```

```
x.grad == None
```

```
True
```

```
x @ x
```

```
tensor([[-1.,  2.],
        [-1., -2.]])
```

```
m = x.svd()
m.U, m.S, m.V
```

```
(tensor([[-0.9732,  0.2298],
         [ 0.2298,  0.9732]]),
 tensor([2.2882, 0.8740]),
 tensor([[-0.5257, -0.8507],
         [-0.8507,  0.5257]]))
```

```
x**2, x.exp().sum(), x.trace()
```

```
(tensor([[1., 4.],
         [1., 0.]]), tensor(11.4752), tensor(1.))
```

多次元配列で自動微分をやる枠組み

PyTorch

```
import torch

a = torch.tensor([2., 3., 1.], requires_grad=True)
b = torch.tensor([6., 4., 1.], requires_grad=True)
Q = 3*a**3 - b**2
```

```
Q
```

```
tensor([-12.,  65.,   2.], grad_fn=<SubBackward0>)
```

```
Q.backward(gradient=torch.tensor([1., 1., 1.]))
```

```
a.grad, b.grad
```

```
(tensor([36., 81.,  9.]), tensor([-12.,  -8.,  -2.]))
```

```
print(9*a**2 == a.grad)
print(-2*b == b.grad)
```

```
tensor([True, True, True])
tensor([True, True, True])
```

ベクトル量でもできる！
(中でJacobianも計算)

backwardを始める値

$\frac{\partial Q}{\partial Q} = 1$ **の指定が必要**

PyTorchの学習のコア部分

```
model = ...

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

for epoch in range(100):
    for (inputs, labels) in trainloader:
        optimizer.zero_grad()
        loss = criterion(model(inputs), labels)
        loss.backward()
        optimizer.step()
```

optimizer.zero_grad() model.parameters()のgradを0で初期化

loss = criterion(model(inputs), labels) 自動微分 (forward)

loss.backward() 自動微分 (backward)

optimizer.step() model.parameters()をgradで更新

先ほどのモデル：もっとPyTorchぽくして再実装…

予測モデル　$\hat{y} = f(x_1, x_2, \theta_1, \theta_2, \theta_3)$

$= \theta_1 x_1 - \theta_2 \cos(\theta_3 + x_2)$

```
from torch.nn.parameter import Parameter

class f:
  def __init__(self):
    self.theta1 = Parameter(torch.tensor(0.2))
    self.theta2 = Parameter(torch.tensor(0.3))
    self.theta3 = Parameter(torch.tensor(0.4))

  def parameters(self):
    return [self.theta1, self.theta2, self.theta3]

  def forward(self, x1, x2):
    z1 = self.theta3 + x2
    z2 = torch.cos(z1)
    z3 = self.theta2 * z2
    z4 = self.theta1 * x1
    return z4 - z3
```

モデルパラメタを定義・初期化

$\theta_1 = 0.2$

$\theta_2 = 0.3$

$\theta_3 = 0.4$

forward計算を定義
(計算グラフをつくる)

訓練データを適当に用意…

予測モデル　$\hat{y} = f(x_1, x_2, \theta_1, \theta_2, \theta_3)$

$= \theta_1 x_1 - \theta_2 \cos(\theta_3 + x_2)$

訓練データ(3点)

$$\begin{cases} \boldsymbol{x}_1 = (5.0, 1.0), & y_1 = -8.2 \\ \boldsymbol{x}_2 = (-7.0, 7.0), & y_2 = -9.5 \\ \boldsymbol{x}_3 = (1.2, 4.0), & y_3 = -0.9 \end{cases}$$

```
model = f()
```

```
data = [((5.0, 1.0), -8.2),
        ((-7.0, 7.0), 9.5),
        ((1.2, 4.0), -0.9)]
```

初期状態での出力値を計算してみる(forwardで)

予測モデル　$\hat{y} = f(x_1, x_2, \theta_1, \theta_2, \theta_3)$

$= \theta_1 x_1 - \theta_2 \cos(\theta_3 + x_2)$

訓練データ(3点)

$$\begin{cases} \boldsymbol{x}_1 = (5.0, 1.0), & y_1 = -8.2 \\ \boldsymbol{x}_2 = (-7.0, 7.0), & y_2 = -9.5 \\ \boldsymbol{x}_3 = (1.2, 4.0), & y_3 = -0.9 \end{cases}$$

初期値 $\theta_1 = 0.2, \theta_2 = 0.3, \theta_3 = 0.4$ のままの**モデル出力**

```
for (x1, x2), y in data:
  with torch.no_grad():
    y_pred = model.forward(x1, x2)
  print(f'input: {x1:+}, {x2:+}, out_true: {y:+.2f}, out_pred: {y_pred:+.2f}')
```

```
input: +5.0, +1.0, out_true: -8.20, out_pred: +0.95
input: -7.0, +7.0, out_true: +9.50, out_pred: -1.53
input: +1.2, +4.0, out_true: -0.90, out_pred: +0.33
```

ロスを定義し、勾配法でパラメタを動かして最小化

ロス関数　$$\mathrm{loss}(\theta_1, \theta_2, \theta_3) = \sum_{i=1}^{3} (y_i - f(\boldsymbol{x}_i))^2$$

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

for epoch in range(100):
  optimizer.zero_grad()
  loss = 0.0
  for (x1, x2), y in data:
    loss += (y - model.forward(x1, x2))**2
  loss.backward()
  optimizer.step()
```

ロスを定義し、勾配法でパラメタを動かして最小化

```
model.parameters()
```

```
[Parameter containing:
 tensor(-1.4882, requires_grad=True), Parameter containing:
 tensor(1.0298, requires_grad=True), Parameter containing:
 tensor(-0.2610, requires_grad=True)]
```

```
for (x1, x2), y in data:
  with torch.no_grad():
    y_pred = model.forward(x1, x2)
  print(f'input: {x1:+}, {x2:+}, out_true: {y:+.2f}, out_pred: {y_pred:+.2f}')
```

```
input: +5.0, +1.0, out_true: -8.20, out_pred: -8.21
input: -7.0, +7.0, out_true: +9.50, out_pred: +9.49
input: +1.2, +4.0, out_true: -0.90, out_pred: -0.93
```

悪くないだろう

optimizerも自前で作ると全体構造がわかる

```python
class grad_descent:
  def __init__(self, params, lr):
    self.params = params
    self.lr = lr

  def zero_grad(self):
    for p in self.params:
      if p.grad is not None:
        p.grad = torch.tensor(0.0)

  @torch.no_grad()
  def step(self):
    for p in self.params:
      p += -self.lr * p.grad
```

① **zero_grad()**

各パラメタのgradをゼロに

`p.grad.zero_()`と書く方がよい

② **step()**

勾配降下

`p.add_(p.grad, alpha=-self.lr)`

と書く方がよい

```python
model = f()
optimizer = grad_descent(model.parameters(), lr=0.01)

for epoch in range(100):
  optimizer.zero_grad()  ①
  loss = 0.0
  for (x1, x2), y in data:
    loss += (y - model.forward(x1, x2))**2
  loss.backward()
  optimizer.step()  ②
```

本日の技術的なメインテーマ：勾配法と自動微分

深層モデルの学習 (pytorchの最小記述でたった7行!)のコア部分で何をやっているのかを理解する

```
for epoch in range(num_epochs):
    for (inputs, labels) in dataloader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

optimizer.zero_grad()

loss = criterion(model(inputs), labels) → 自動微分 (forward)

loss.backward() → 自動微分 (backward)

optimizer.step() 勾配法

確率的勾配降下：実際はミニバッチごとに勾配降下する

```
for epoch in range(num_epochs):
    for (inputs, labels) in dataloader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

`torch.utils.data.DataLoader`

データをミニバッチごとに
切り分けてiterateしてくれる

バッチ $\text{loss} = \sum_{i=1}^{1000} \text{error}(f(\boldsymbol{x}_i), y_i) \longrightarrow \nabla\text{loss}$ で勾配降下

ミニバッチ (batch_size=100の場合, 100枚ごとにloss計算→勾配降下)

1 epoch

$$\begin{cases} \text{loss} = \sum_{i=1}^{100} \text{error}(f(\boldsymbol{x}_i), y_i) \longrightarrow \nabla\text{loss}\text{で勾配降下} \\ \text{loss} = \sum_{i=101}^{200} \text{error}(f(\boldsymbol{x}_i), y_i) \longrightarrow \nabla\text{loss}\text{で勾配降下} \\ \vdots \\ \text{loss} = \sum_{i=901}^{1000} \text{error}(f(\boldsymbol{x}_i), y_i) \longrightarrow \nabla\text{loss}\text{で勾配降下} \end{cases}$$

micrograd: Python100行での自動微分系 (スカラ値のみ)

https://github.com/karpathy/micrograd

```
from micrograd.engine import Value

a = Value(-4.0)
b = Value(2.0)
c = a + b
d = a * b + b**3
c += c + 1
c += 1 + c + (-a)
d += d * 2 + (b + a).relu()
d += 3 * d + (b - a).relu()
e = c - d
f = e**2
g = f / 2.0
g += 10.0 / f
print(f'{g.data:.4f}') # prints 24.7041, the outcome of this forward pass
g.backward()
print(f'{a.grad:.4f}') # prints 138.8338, i.e. the numerical value of dg/da
print(f'{b.grad:.4f}') # prints 645.5773, i.e. the numerical value of dg/db
```

```
micrograd
├── engine.py  (97 lines)
└── nn.py      (60 lines)
```

モダンな自動微分(計算グラフの構築からbackward計算まで)のエッセンスはすべてこの短いコードの中に！

(作者はあのAndrej Karpathy)

ただしスカラ値のみ

tinygrad：Python1000行で自動微分系 (多次元配列)

https://github.com/geohot/tinygrad

Unit Tests passing

For something in between a pytorch and a karpathy/micrograd

This may not be the best deep learning framework, but it is a deep learning framework.

The promise of small

tinygrad will always be below 1000 lines. If it isn't, we will revert commits until tinygrad becomes smaller.

```
from tinygrad.tensor import Tensor

x = Tensor.eye(3)
y = Tensor([[2.0,0,-2.0]])
z = y.matmul(x).sum()
z.backward()

print(x.grad)  # dz/dx
print(y.grad)  # dz/dy
```

```
import torch

x = torch.eye(3, requires_grad=True)
y = torch.tensor([[2.0,0,-2.0]], requires_grad=True)
z = y.matmul(x).sum()
z.backward()

print(x.grad)  # dz/dx
print(y.grad)  # dz/dy
```

tinygrad：Python1000行で完全な自動微分系を目指す

Due to its extreme simplicity, it aims to be the easiest framework to add new accelerators to, with support for both inference and training. Support the simple basic ops, and you get SOTA vision `extra/efficientnet.py` and language `extra/transformer.py` models. We are working on support for the Apple Neural Engine.

Eventually, we will build custom hardware for tinygrad, and it will be blindingly fast. Now, it is slow.

- geohotが勉強を兼ねて趣味で作っている自動微分フレームワーク
- スカラ値のみのmicrogradと汎用フレームワークpytorchの間を目指す
- **ただし1000行を超えない！！**
- 訓練・推論いずれもアクセラレータ(GPU)をサポート
- ANE (Apple Neural Engine)のサポートの作業中
- EfficientNetやTransformerの実装例まである (YOLOやBERTも視野に?)
- 開発過程のストリーミングがYouTubeにある https://www.youtube.com/c/georgehotzarchive/search?query=tinygrad
- githubのソースをinitial commitから追ってみると追体験もできる

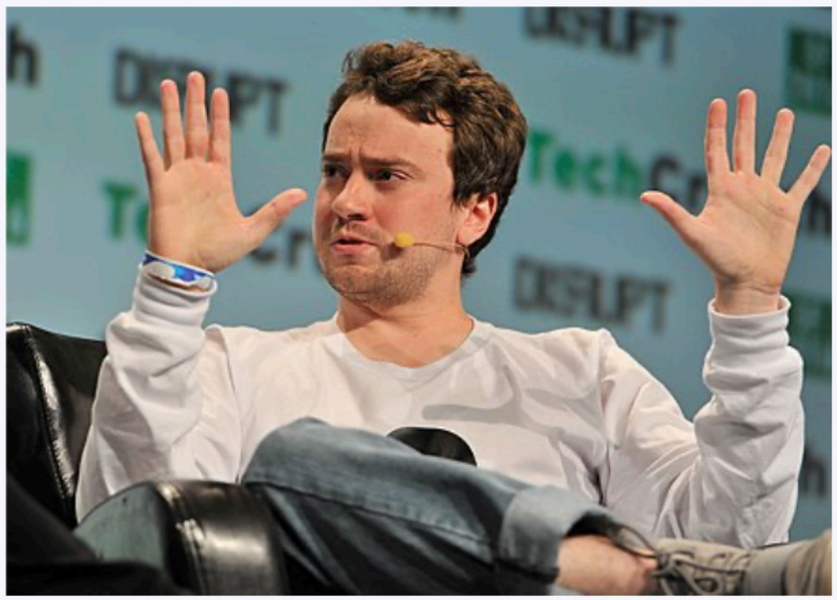
tinygrad

Neural network example (from test/test_mnist.py)

```
from tinygrad.tensor import Tensor
import tinygrad.optim as optim

class TinyBobNet:
  def __init__(self):
    self.l1 = Tensor.uniform(784, 128)
    self.l2 = Tensor.uniform(128, 10)

  def forward(self, x):
    return x.dot(self.l1).relu().dot(self.l2).logsoftmax()

model = TinyBobNet()
optim = optim.SGD([model.l1, model.l2], lr=0.001)

# ... and complete like pytorch, with (x,y) data

out = model.forward(x)
loss = out.mul(y).mean()
optim.zero_grad()
loss.backward()
optim.step()
```

蛇足：geohotとcomma.ai

George Hotz



Hotz in 2016

Born	George Francis Hotz Jr. October 2, 1989 (age 31) Glen Rock, New Jersey, US
Other names	geohot
Notable work	Jailbreak, comma.ai
Website	www.geohot.com

comma.ai

- 高校卒業後の2007年8月(17才)に世界ではじめてiPhoneのSIM lockを解除するのに成功したiOS jailbreaksで有名な伝説のハッカー
- 2009年末には誰も成功していなかったPlaystation 3のジェイルブレイクに成功。海賊版がプレイできるようになったことによりSCE米国支社が提訴
- 2014年6月、Androidのroot化ツールTowelrootをリリース。root化が難航し約180万円の賞金が掛けられていたGalaxy S5にも対応
- 2016年、通常の自動車を自動運転カーにアップグレードするキットを販売する会社Comma.aiを立ち上げ

JAX: より高速な自動微分系を目指して

JAX is NumPy on the CPU, GPU, and TPU, with great automatic differentiation for high-performance machine learning research.

└→ Google Colabで使えるのでGPU/TPUインスタンスで試してみてね

https://github.com/google/jax

JAX: Autograd and XLA

Continuous integration passing | pypi v0.2.8

Quickstart | **Transformations** | **Install guide** | **Neural net libraries** | **Change logs** | **Reference docs** | **Code search**

News: JAX tops largest-scale MLPerf Training 0.7 benchmarks!

NumPy

nature

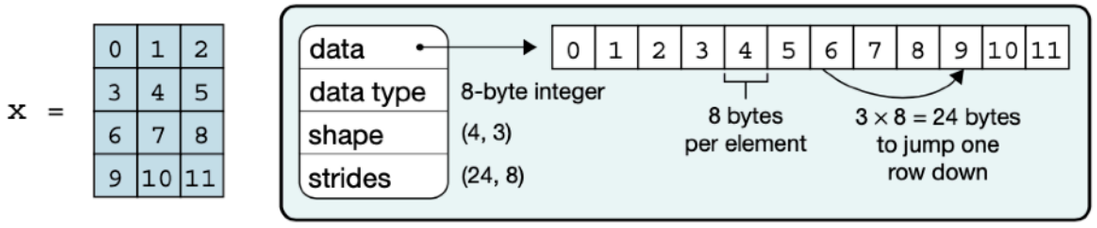
Review Article | Open Access | Published: 16 September 2020

Array programming with NumPy

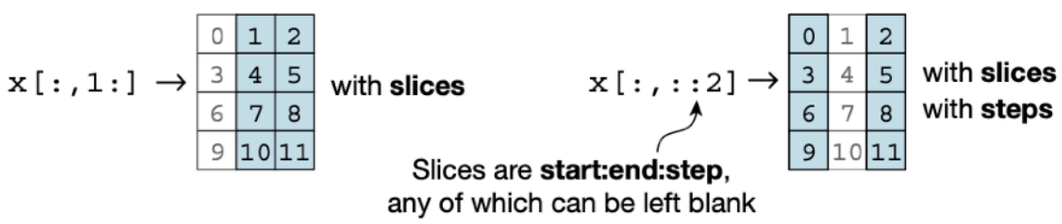
Charles R. Harris, K. Jarrod Millman ✉, [...] Travis E. Oliphant

Nature **585**, 357–362(2020) | Cite this article

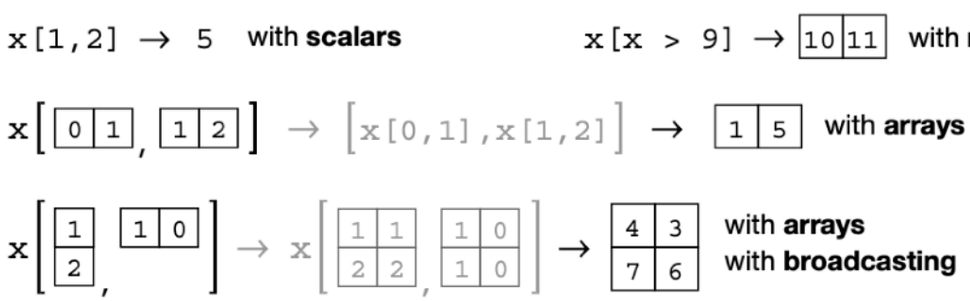
a Data structure



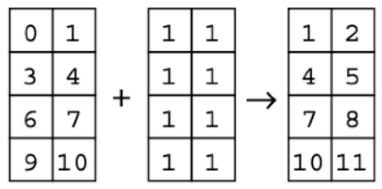
b Indexing (view)



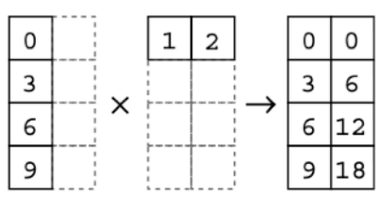
c Indexing (copy)



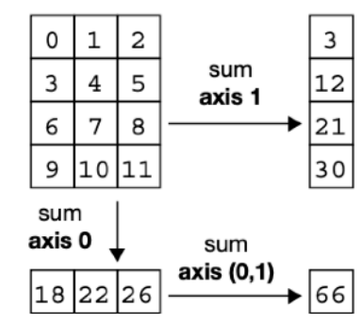
d Vectorization



e Broadcasting



f Reduction



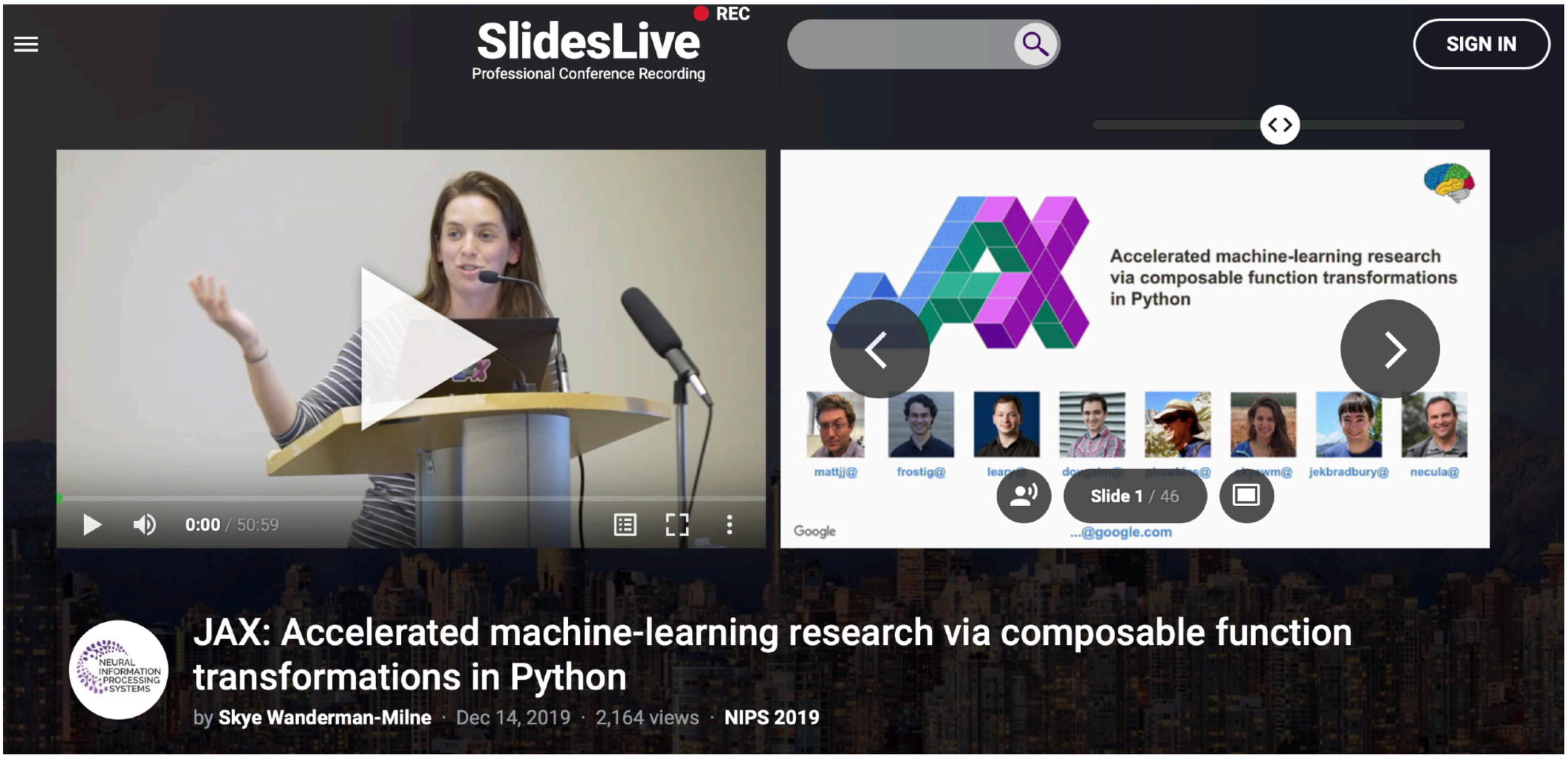
g Example

```
In [1]: import numpy as np

In [2]: x = np.arange(12)

In [3]: x = x.reshape(4, 3)

In [4]: x
Out[4]:
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])

In [5]: np.mean(x, axis=0)
Out[5]: array([4.5, 5.5, 6.5])

In [6]: x = x - np.mean(x, axis=0)

In [7]: x
Out[7]:
array([[-4.5, -4.5, -4.5],
       [-1.5, -1.5, -1.5],
       [ 1.5,  1.5,  1.5],
       [ 4.5,  4.5,  4.5]])
```

JAX

Talk and demo by Skye Wanderman-Milne @ NeurIPS2019

https://slideslive.com/38923687/jax-accelerated-machinelearning-research-via-composable-function-transformations-in-python



JAX

https://github.com/google/jax

Transformations

At its core, JAX is an extensible system for transforming numerical functions. Here are four of primary interest: `grad`, `jit`, `vmap`, and `pmap`.

Automatic differentiation with `grad`

JAX has roughly the same API as Autograd. The most popular function is `grad` for reverse-mode gradients:

```
from jax import grad
import jax.numpy as jnp

def tanh(x):  # Define a function
  y = jnp.exp(-2.0 * x)
  return (1.0 - y) / (1.0 + y)

grad_tanh = grad(tanh)  # Obtain its gradient function
print(grad_tanh(1.0))   # Evaluate it at x = 1.0
# prints 0.4199743
```

You can differentiate to any order with `grad`.

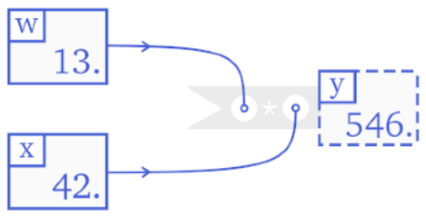
```
print(grad(grad(grad(tanh)))(1.0))
# prints 0.62162673
```

JAX

PyTorch

Executing code produces graph/tape

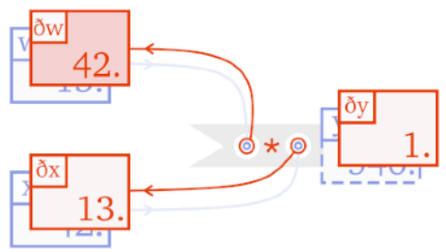
```
w = torch.tensor(13.)
x = torch.tensor(42.)
y = w * x
```



⟹

Backprop/reverse-mode autodiff
by following the graph/tape

```
y.backward()
```

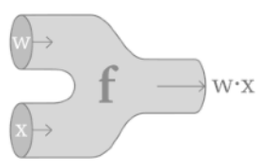



```
grad_w = w.grad
```

JAX

Define pure function

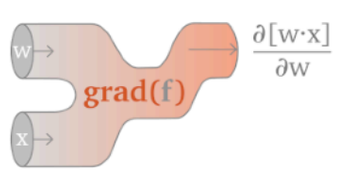
```
def f(w, x):
  return w * x
```



⟹

JAX creates gradient function

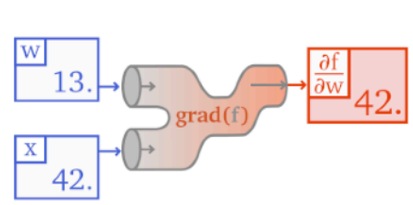
```
df_dw = jax.grad(f)
# ⇒ df_dw(w, x) == x
```



⟹

Evaluate that to get gradients

```
w = jnp.array(13.)
x = jnp.array(42.)
grad_w = df_dw(w, x)
```



From PyTorch to JAX: towards neural net frameworks that purify stateful code

`https://sjmielke.com/jax-purify.htm`

自動微分: JAX vs PyTorch

```
import jax
import jax.numpy as jnp
import jax.nn as jnn
```

```
def loss_fn(theta, phi, psi, x, y):
  u = jnn.sigmoid(theta.T @ x)
  v = jnn.sigmoid(phi.T @ u)
  y_hat = psi @ v
  return (y-y_hat)**2
```

```
grad_fn = jax.grad(loss_fn, argnums=(0))
```

```
x = jnp.array([-1.0, 2.0])
y = jnp.array(4.5)

theta = jnp.array([[0.1, -0.1, 1.2],[0.9, 0.0, -0.7]])
phi = jnp.array([[1.1, -5.4], [2.2, 6.9], [8.8, -3.8]])
psi = jnp.array([6.9, -11.0])

theta_grad = grad_fn(theta, phi, psi, x, y)
```

```
theta_grad[1, 2]
```

```
DeviceArray(-1.6196026, dtype=float32)
```

```
import torch

x = torch.tensor([-1.0, 2.0])
y = torch.tensor(4.5)

theta = torch.tensor([[0.1, -0.1, 1.2],[0.9, 0.0, -0.7]],
                     requires_grad=True)
phi = torch.tensor([[1.1, -5.4], [2.2, 6.9], [8.8, -3.8]],
                   requires_grad=True)
psi = torch.tensor([6.9, -11.0],
                   requires_grad=True)
```

```
u = torch.sigmoid(theta.T @ x)
v = torch.sigmoid(phi.T @ u)
y_hat = psi @ v
L = (y-y_hat)**2
```

```
L.backward()
```

```
theta.grad.data[1, 2]
```

```
tensor(-1.6196)
```

本日の内容

1. 機械学習 = 新しいプログラミング
2. 勾配と勾配降下法
3. 合成関数の自動微分
4. Pythonで作って理解しよう
5. Q & A

このスライドは下記にあとで置いておきます

https://itakigawa.github.io/news.html