

# PyTorch 講座 8

肥田 歩華

# 前回のおさらい

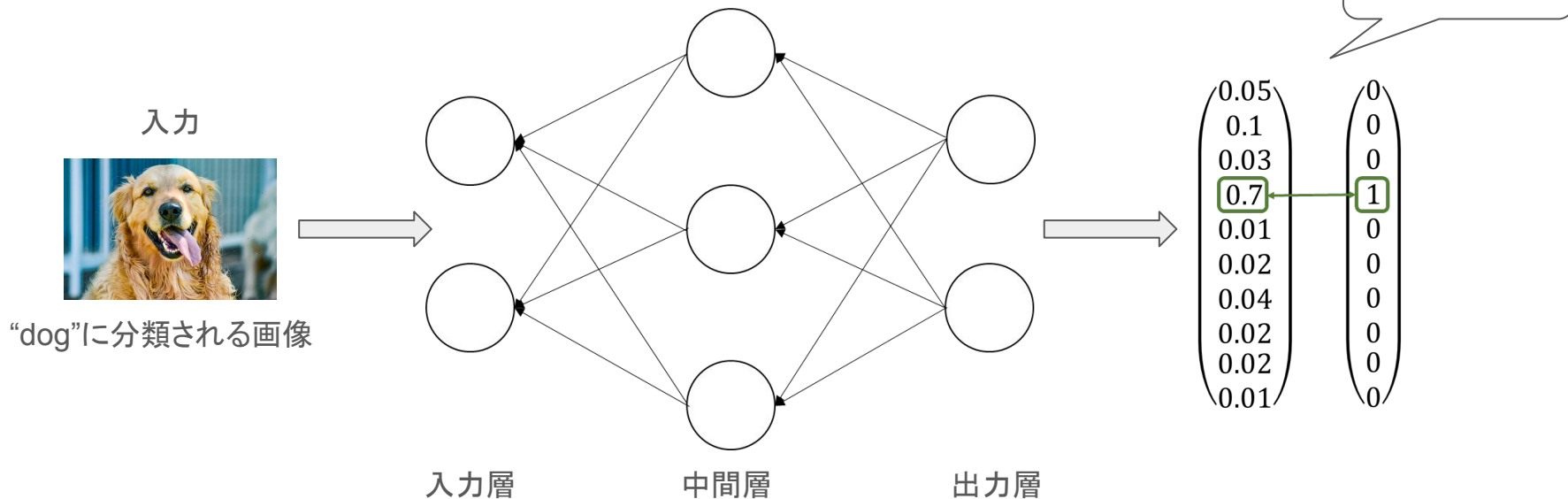
- トレーニング
  - データセットの読み込み
  - モデルの作成
  - データローダーからデータを受け取る
  - 受け取ったデータをモデルに入力
  - モデルからの出力を受け取る
  - 出力を使って誤差を算出
  - 誤差を使って重みを更新

# 今回やること

- 損失関数の説明
- optimizerの説明

# どうやって学習するか(前回のスライド)

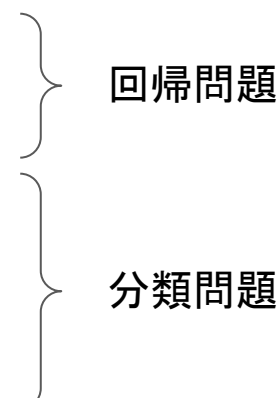
- モデルに訓練データを入力し、予測結果を受け取る
- 予測結果と正解ラベルを比較・誤差を計算
- 正解データとの誤差が最小になるようにパラメータを更新
- 誤差の減少がみられなくなる(収束する)まで繰り返す



# 損失関数

モデルの予測結果と正解ラベルの差を表す関数

代表例

- 平均二乗誤差 (Mean Squared Error Loss)
  - 平均絶対誤差 (Mean Absolute Error Loss)
  - 交差エントロピー誤差 (Cross-Entropy Loss)
  - 二値交差エントロピー (Binary Cross-Entropy Loss)
  - 多クラス交差エントロピー (Categorical Cross-Entropy Loss)
- 
- 回帰問題
- 分類問題

# 平均二乗誤差 (MSELoss: Mean Squared Error Loss)

```
CLASS torch.nn.MSELoss(size_average=None, reduce=None, reduction='mean') \[SOURCE\]
```

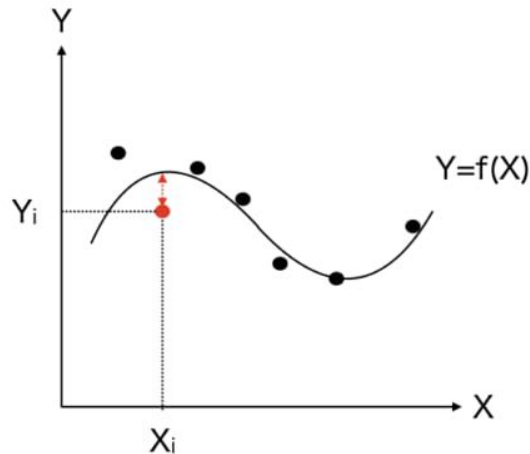
```
>>> loss = nn.MSELoss()
>>> input = torch.randn(3, 5, requires_grad=True)
>>> target = torch.randn(3, 5)
>>> output = loss(input, target)
>>> output.backward()
```

予測値と正解値との差の二乗の平均

- 主に回帰問題に使われる

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (\hat{y}_i - y_i)^2$$

$$= \frac{1}{\text{データ数}} \sum_{i\text{番目}=1\text{から}}^{\text{データ数まで総和}} (\text{予測値}_{i\text{番目}} - \text{正解値}_{i\text{番目}})^2$$



# 交差エントロピー誤差 (Cross-Entropy Loss)

```
CLASS torch.nn.CrossEntropyLoss(weight=None, size_average=None,  
    ignore_index=-100, reduce=None, reduction='mean',  
    label_smoothing=0.0) [SOURCE]
```

```
>>> # Example of target with class indices  
>>> loss = nn.CrossEntropyLoss()  
>>> input = torch.randn(3, 5, requires_grad=True)  
>>> target = torch.empty(3, dtype=torch.long).random_(5)  
>>> output = loss(input, target)  
>>> output.backward()
```

自然対数eを底とするモデル出力値のlog値と正解データ値を乗算したものの総和

- 主に多クラス分類問題で使われる

$$\text{LogLoss} = -\frac{1}{n} \sum_{i=1}^n (y_i \log p_i + (1 - y_i) \log(1 - p_i))$$

各データのLogLoss値

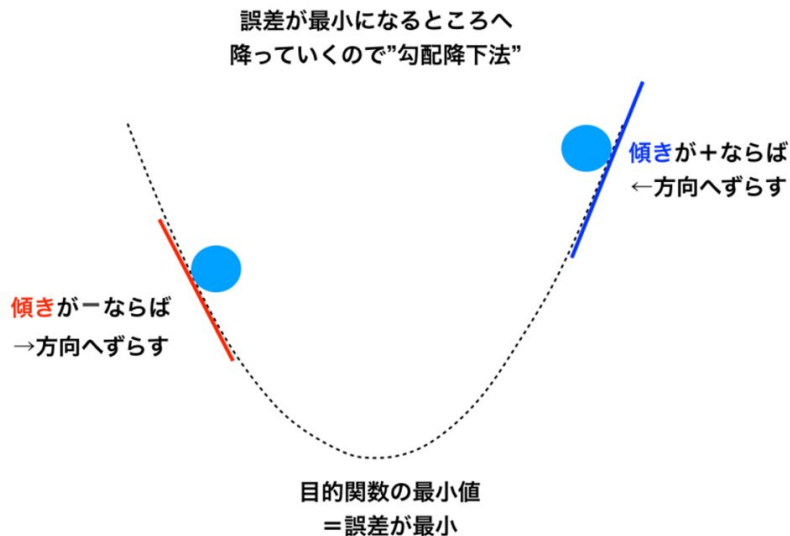
$$\begin{aligned} &= (\text{正解値} * -\text{自然対数}(\text{予測値})) \\ &\quad + ((1 - \text{正解値}) * -\text{自然対数}(1 - \text{予測値})) \\ &= y * -\log(p) + (1 - y) * -\log(1 - p) \\ &= -y \log(p) - ((1 - y) \log(1 - p)) \\ &= -y \log(p) - ((1 - y) \log(1 - p)) \end{aligned}$$

# 最適化アルゴリズム (Optimizer)

損失関数の値を最小化するようなパラメータを見つけ  
最適化(Optimization)を行う機構

## 代表例

- SGD : 最も基本的なアルゴリズム
- Momentum SGD : SGD + 「慣性」の概念
- AdaGrad : 学習が進むほど学習率を小さくしていく
- RMSProp : AdaGrad + 最近の勾配ほど強く影響
- Adam : momentumSGD + RMSprop



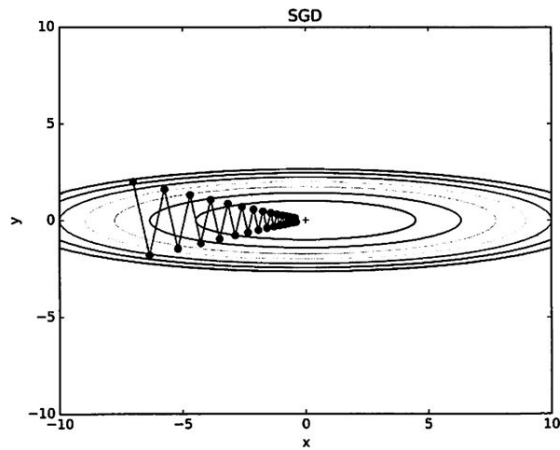


# 確率的勾配降下法 (SGD : Stochastic Gradient Descent)

```
CLASS torch.optim.SGD(params, lr=<required parameter>, momentum=0,  
    dampening=0, weight_decay=0, nesterov=False, *, maximize=False,  
    foreach=None, differentiable=False) [SOURCE]
```

```
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1,  
momentum=0.9)  
>>> optimizer.zero_grad()  
>>> loss_fn(model(input), target).backward()  
>>> optimizer.step()
```

パラメータの1回の更新に、  
データセット全体ではなく  
ランダムに取り出したミニバッチを使って勾配を計算



# Momentum SGD

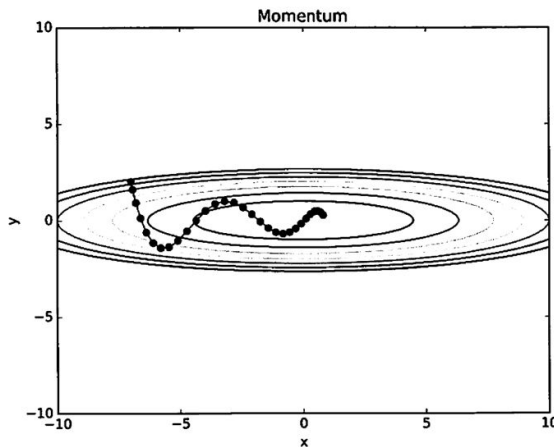
```
CLASS torch.optim.SGD(params, lr=<required parameter>, momentum=0,  
    dampening=0, weight_decay=0, nesterov=False, *, maximize=False,  
    foreach=None, differentiable=False) \[SOURCE\]
```

SGDに慣性の概念を取り入れたもの

Momentum:「運動量」

同一方向への移動の積み重ねで加速する(収束が速い)

ちょっとした山や平地も乗り越えられる(局所解の回避)



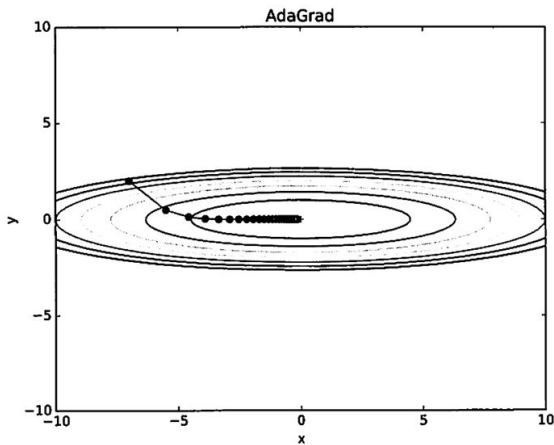
# AdaGrad

```
CLASS torch.optim.Adagrad(params, lr=0.01, lr_decay=0, weight_decay=0,  
    initial_accumulator_value=0, eps=1e-10, foreach=None, *,  
    maximize=False, differentiable=False) [SOURCE]
```

学習が進むにつれて学習率を小さくしていく手法  
最初は大きく学習し、次第に小さく学習する

過去の勾配の二乗和を保持し、  
平方の逆数を学習率に乗算

$$\begin{aligned}h_0 &= \epsilon \\h_t &= h_{t-1} + \nabla E(\mathbf{w}^t)^2 \\ \eta_t &= \frac{\eta_0}{\sqrt{h_t}} \\ \mathbf{w}^{t+1} &= \mathbf{w}^t - \eta_t \nabla E(\mathbf{w}^t)\end{aligned}$$



# RMSProp

```
CLASS torch.optim.RMSprop(params, lr=0.01, alpha=0.99, eps=1e-08,  
    weight_decay=0, momentum=0, centered=False, foreach=None,  
    maximize=False, differentiable=False) [SOURCE]
```

AdaGradは一度学習率が小さくなると学習されなくなる



過去の勾配を徐々に忘れて、新しい勾配の情報を大きく反映

移動指数平均を用いて、  
過去の勾配を指数関数的にスケールダウンさせる

$$h_t = \alpha h_{t-1} + (1 - \alpha) \nabla E(\mathbf{w}^t)^2$$

$$\eta_t = \frac{\eta_0}{\sqrt{h_t} + \epsilon}$$

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta_t \nabla E(\mathbf{w}^t)$$

# Adam

```
CLASS torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08,  
    weight_decay=0, amsgrad=False, *, foreach=None, maximize=False,  
    capturable=False, differentiable=False, fused=None) [SOURCE]
```

momentumSGDとRMSpropを組み合わせたアルゴリズム

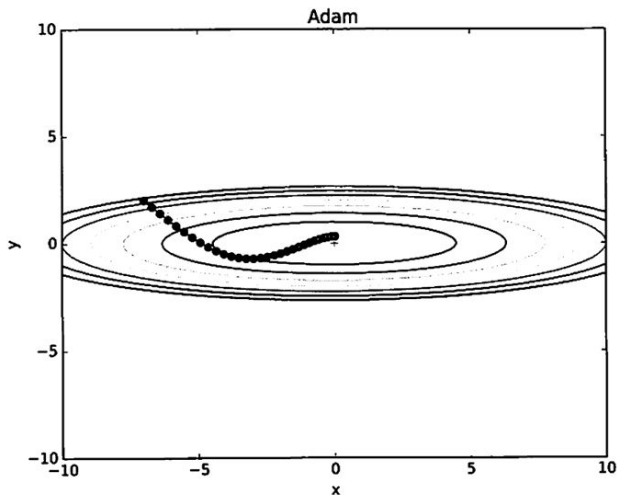
$$m_{t+1} = \beta_1 m_t + (1 - \beta_1) \nabla E(\mathbf{w}^t)$$

$$v_{t+1} = \beta_2 v_t + (1 - \beta_2) \nabla E(\mathbf{w}^t)^2$$

$$\hat{m} = \frac{m_{t+1}}{1 - \beta_1^t}$$

$$\hat{v} = \frac{v_{t+1}}{1 - \beta_2^t}$$

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \alpha \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}$$



# 課題説明

# 最終課題

## (必須)

画像分類モデルを自前で組んで、CIFAR-10テストデータで分類精度55%以上を達成する。

実装するファイル

- dataset.py : データセットの読み込み
- model.py : モデルの作成
- train.py : モデルの学習・保存
- evaluation.py : 保存したモデルを読み込み、テストデータで精度を評価

## (任意)

ResNet18で学習を行い、自分で組んだモデルと分類精度を比較する。

TorchvisionからResNetを利用することができる

- <https://pytorch.org/vision/main/models/generated/torchvision.models.resnet18.html>

# 最終課題

## train.py : モデルの保存

```
train.py ×
codes > 08 > example > train.py > ...
1  # モジュールのインポート
2  import torch
3  import torch.nn as nn
4  from torch.utils.data import DataLoader
5  import torch.optim as optim
6
7  # dataset.py内のdatasets関数をインポート
8  from dataset import cifar_dataset
9  # model.py内のCNNクラスをインポート
10 from model import CNN
11
12 # 保存先のパス
13 model_path = 'cifar_cnn.pth'
14
15 # データローダーからデータを受け取る
16 train_data, _ = cifar_dataset()
17 train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
18
19 # モデル、損失関数、最適化関数の定義
20 model = CNN()
21 criterion = nn.CrossEntropyLoss()
22 optimizer = optim.SGD(model.parameters(), lr=0.01)
23
```

```
train.py ×
codes > 08 > example > train.py > ...
24 if __name__ == "__main__":
25     epochs = 20
26
27     for epoch in range(epochs):
28         train_loss = 0
29         train_acc = 0
30
31         #train
32         model.train()
33         for i, (images, labels) in enumerate(train_loader):
34             optimizer.zero_grad()
35             outputs = model(images)
36             loss = criterion(outputs, labels)
37             train_loss += loss.item()
38             train_acc += (outputs.max(1)[1] == labels).sum().item()
39             loss.backward()
40             optimizer.step()
41
42         avg_train_loss = train_loss / len(train_loader)
43         avg_train_acc = train_acc / len(train_loader.dataset)
44
45         # モデルの保存
46         torch.save({
47             'epoch': epoch,
48             'model_state_dict': model.state_dict(),
49             'optimizer_state_dict': optimizer.state_dict(),
50             'loss': avg_train_loss
51         }, model_path)
52
53     print('Epoch: {}, Loss: {loss:.4f}'.format(epoch+1, i+1, loss=avg_train_loss))
```



# 最終課題

## evaluation.py : 保存したモデルの読み込み

evaluation.py

codes > 08 > example > evaluation.py > ...

```
1  # モジュールのインポート
2  import torch
3  from torch.utils.data import DataLoader
4
5  # dataset.py内のdatasets関数をインポート
6  from dataset import cifar_dataset
7  # model.py内のCNNクラスをインポート
8  from model import CNN
9
10 # 保存されたモデルのパス
11 model_path = 'cifar_cnn.pth'
12
13 # データローダーからデータを受け取る
14 _, test_data = cifar_dataset()
15 test_loader = DataLoader(test_data, batch_size=64, shuffle=False)
16
17 # モデルの定義
18 model = CNN()
19
20 # 保存されたモデルの読み込み
21 checkpoint = torch.load(model_path)
22 model.load_state_dict(checkpoint['model_state_dict'])
23 model.eval()
24
```

evaluation.py

codes > 08 > example > evaluation.py > ...

```
25 if __name__=="__main__":
26     val_acc = 0
27
28     with torch.no_grad():
29         for images, labels in test_loader:
30             outputs = model(images)
31             val_acc += (outputs.max(1)[1] == labels).sum().item()
32     avg_val_acc = val_acc / len(test_loader.dataset)
33
34     print('Accuracy: {:.4f}'.format(avg_val_acc))
```

# 最終課題

## 提出方法

- zip化したファイルをGoogle Driveにアップロード(提出先URLは後日メールで告知)
- zipファイル名は“学籍番号\_pytorch”

## 提出期限

1月下旬(提出期限はメールで告知)