MPRG勉強会

MNIST, CIFAR10での画像認識， PyTorch

機械知覚&ロボティクスグループ (MPRG)
新田 常顧

MACHINE PERCEPTION AND ROBOTICS GROUP

今回の内容

- データセット
  - MNIST
  - CIFAR-10
- 画像分類モデル
  - Convolutional Neural Network
- PyTorchでの画像分類モデルの実装
  - PyTorch
  - 演習

今回の内容

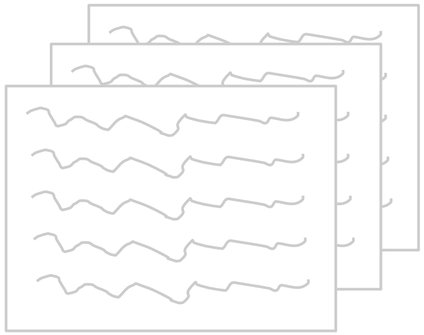
- データセット
  - MNIST
  - CIFAR-10
- 画像分類モデル
  - Convolutional Neural Network
- PyTorchでの画像分類モデルの実装
  - PyTorch
  - 演習

データセット

- ある目的で集められ、一定の形式に整えられたデータの集合体

- 機械学習分野での定義
  - プログラムで処理されるデータの集合体

例



テキスト
- 機械翻訳
- 対話システム

…

画像・動画
- 物体認識
- 自動運転

…

MNIST，CIFAR-10についての説明

音声
- 音声認識

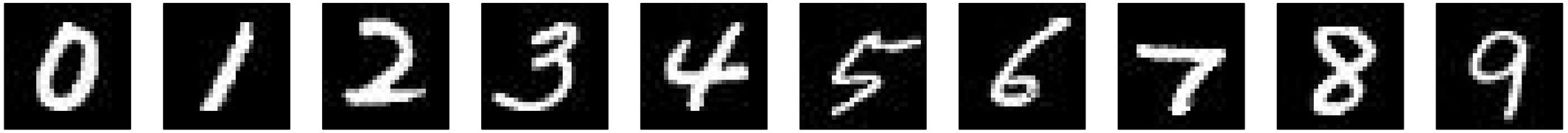
…

MNIST

- 0から9の手書き数字の白黒画像で構成
- データが整形されており，クラス数も少ないため画像分類において高精度が出し易い
  - 初心者向けのチュートリアルとしてよく利用

- 画像サイズ：28×28
- クラス数　：10

- 合計7万枚のデータセット（画像とラベルのペア）
  - 学習用データ　：6万枚
  - テスト用データ：1万枚

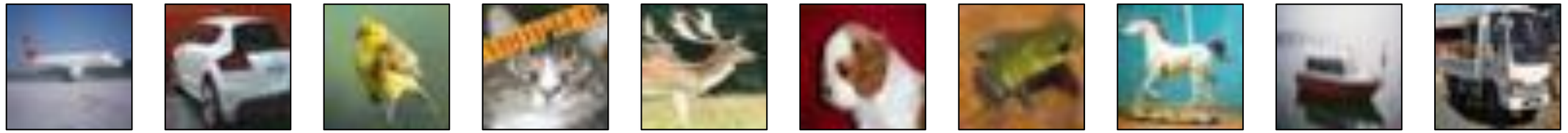


CIFAR-10

- 10種類の物体カラー画像で構成
- データが整形されており，クラス数も少ないため画像分類において高精度が出し易い
  - 初心者向けのチュートリアルとしてよく利用

- 画像サイズ：$32 \times 32 \times 3$（RGB値）
- クラス数　：10

- 合計6万枚のデータセット（画像とラベルのペア）
  - 学習用データ　：5万枚
  - テスト用データ：1万枚

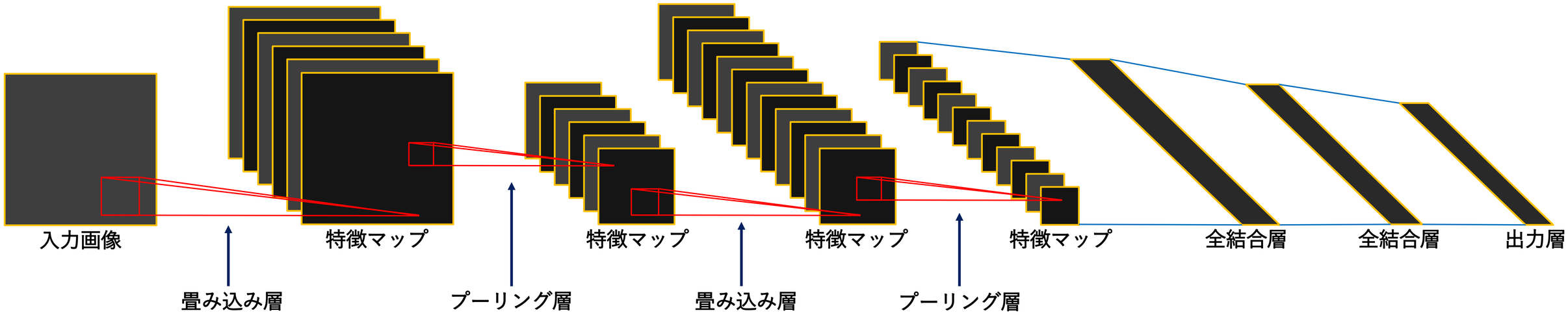


今回の内容

- データセット
  - MNIST
  - CIFAR-10
- 画像分類モデル
  - Convolutional Neural Network
- PyTorchでの画像分類モデルの実装
  - PyTorch
  - 演習

画像分類モデル

- 画像がどのカテゴリに属するか分類するモデル
- Convolutional Neural Network (CNN) [Y.LeCun+, 1998]
  - 畳み込み層とプーリング層を持つニューラルネットワーク
  - 派生モデル：ResNet, DenseNet ,…



- Vision Transformer (ViT) [A. Dosovitskiy+, 2020]
  - 自然言語処理に用いられていたTransformerを画像分類に応用

今回の内容

- データセット
  - MNIST
  - CIFAR-10
- 画像分類モデル
  - Convolutional Neural Network
- PyTorchでの画像分類モデルの実装
  - PyTorch
  - 演習

PyTorch

MPRG

- Pythonのオープンソース機械学習ライブラリ
  - 2016年に初版が公開
  - 現在でもPythonの機械学習ライブラリとして非常に人気

- メリット
  - 直感的なコーディングが可能
    - Numpyと非常に似た扱い方
  - 参照リソースが豊富
    - コミュニティが活発

CNNによるMNISTの学習・分類

- 目的
  - 畳み込みニューラルネットワークを用いてMNISTデータセットに対する文字認識を行う
- URL
  - MNIST_CNN

モジュールのインポート

- 必要なモジュールをインポート

```
1 from time import time  ➡実行時間を計測する機能
2
3 import numpy as np  ➡Pythonで配列を扱う機能
4 import torch ➡Pythonのオープンソース機械学習ライブラリ（PyTorch）
5 import torch.nn as nn ➡PyTorchのニューラルネットワーク(NN)機能
6
7 import torchvision        ➡一般的なデータセット，モデル構造，一般的な画像変換機能
8 import torchvision.transforms as transforms ➡データに前処理を行う機能
9                                              （リサイズ，切り抜き等）
10 import torchsummary  ➡PyTorchで作成したネットワークモデルの詳細を表示できる機能
```

データセットの読み込み

- torchvisionの機能を用いてデータセットを読み込む

```
train_data = torchvision.datasets.MNIST(root="./", train=True, transform=transforms.ToTensor(), download=True)
test_data =  torchvision.datasets.MNIST(root="./", train=False, transform=transforms.ToTensor(), download=True)
```

torchvisionの提供する
データセットを利用

データセットの読み込み

- torchvisionの機能を用いてデータセットを読み込む

```
train_data = torchvision.datasets.MNIST(root="./", train=True, transform=transforms.ToTensor(), download=True)
test_data =  torchvision.datasets.MNIST(root="./", train=False, transform=transforms.ToTensor(), download=True)
```

使用するデータセット
を指定

データセットの読み込み

- torchvisionの機能を用いてデータセットを読み込む

```
train_data = torchvision.datasets.MNIST(root="./", train=True, transform=transforms.ToTensor(), download=True)
test_data =  torchvision.datasets.MNIST(root="./", train=False, transform=transforms.ToTensor(), download=True)
```

データセットの保存先を指定

データセットの読み込み

- torchvisionの機能を用いてデータセットを読み込む

```
train_data = torchvision.datasets.MNIST(root="./", train=True, transform=transforms.ToTensor(), download=True)
test_data =  torchvision.datasets.MNIST(root="./", train=False, transform=transforms.ToTensor(), download=True)
```

データセットの種類を指定
True：学習用データ
False：テスト用データ

データセットの読み込み

- torchvisionの機能を用いてデータセットを読み込む

```
train_data = torchvision.datasets.MNIST(root="./", train=True, transform=transforms.ToTensor(), download=True)
test_data =  torchvision.datasets.MNIST(root="./", train=False, transform=transforms.ToTensor(), download=True)
```

画像の変換
ToTensor()：テンソル型に変換

データセットの読み込みとサイズの確認

- torchvisionの機能を用いてデータセットを読み込む

```
train_data = torchvision.datasets.MNIST(root="./", train=True, transform=transforms.ToTensor(), download=True)
test_data =  torchvision.datasets.MNIST(root="./", train=False, transform=transforms.ToTensor(), download=True)
```

データをダウンロードするか否かの指定

データセットの読み込みとサイズの確認

- torchvisionの機能を用いてデータセットを読み込む

```
train_data = torchvision.datasets.MNIST(root="./", train=True, transform=transforms.ToTensor(), download=True)
test_data =  torchvision.datasets.MNIST(root="./", train=False, transform=transforms.ToTensor(), download=True)
```

- サイズの確認

```
print(train_data.data.size(), train_data.targets.size())
print(test_data.data.size(), test_data.targets.size())
```

```
torch.Size([60000, 28, 28]) torch.Size([60000])
torch.Size([10000, 28, 28]) torch.Size([10000])
```

サンプル数　画像のサイズ　ラベルの数

ネットワークモデルの定義

- 畳み込み層2層，全結合層3層で構成されるCNNを定義
  - __init__　：構成に必要な層を定義
  - forward　：上記で定義した層を接続して処理するように記述

```
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.l1 = nn.Linear(7*7*32, 1024)
        self.l2 = nn.Linear(1024, 1024)
        self.l3 = nn.Linear(1024, 10)
        self.act = nn.ReLU()
        self.pool = nn.MaxPool2d(2, 2)

    def forward(self, x):
        h = self.pool(self.act(self.conv1(x)))
        h = self.pool(self.act(self.conv2(h)))
        h = h.view(h.size()[0], -1)
        h = self.act(self.l1(h))
        h = self.act(self.l2(h))
        h = self.l3(h)
        return h
```

畳み込み層の定義
入力チャンネル数，出力チャンネル数，カーネルサイズ，
ストライド，パディング

ネットワークモデルの定義

- 畳み込み層2層，全結合層3層で構成されるCNNを定義
  - __init__　：構成に必要な層を定義
  - forward　：上記で定義した層を接続して処理するように記述

```
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.l1 = nn.Linear(7*7*32, 1024)
        self.l2 = nn.Linear(1024, 1024)
        self.l3 = nn.Linear(1024, 10)
        self.act = nn.ReLU()
        self.pool = nn.MaxPool2d(2, 2)

    def forward(self, x):
        h = self.pool(self.act(self.conv1(x)))
        h = self.pool(self.act(self.conv2(h)))
        h = h.view(h.size()[0], -1)
        h = self.act(self.l1(h))
        h = self.act(self.l2(h))
        h = self.l3(h)
        return h
```

全結合層の定義
入力ユニット数，出力ユニット数

ネットワークモデルの定義

- 畳み込み層2層，全結合層3層で構成されるCNNを定義
  - __init__　：構成に必要な層を定義
  - forward　：上記で定義した層を接続して処理するように記述

```
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.l1 = nn.Linear(7*7*32, 1024)
        self.l2 = nn.Linear(1024, 1024)
        self.l3 = nn.Linear(1024, 10)
        self.act = nn.ReLU()
        self.pool = nn.MaxPool2d(2, 2)

    def forward(self, x):
        h = self.pool(self.act(self.conv1(x)))
        h = self.pool(self.act(self.conv2(h)))
        h = h.view(h.size()[0], -1)
        h = self.act(self.l1(h))
        h = self.act(self.l2(h))
        h = self.l3(h)
        return h
```

活性化関数の定義
Rectified Linear Unit (ReLU)を採用

ネットワークモデルの定義

- 畳み込み層2層，全結合層3層で構成されるCNNを定義
  - __init__　：構成に必要な層を定義
  - forward　：上記で定義した層を接続して処理するように記述

```
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.l1 = nn.Linear(7*7*32, 1024)
        self.l2 = nn.Linear(1024, 1024)
        self.l3 = nn.Linear(1024, 10)
        self.act = nn.ReLU()
        self.pool = nn.MaxPool2d(2, 2)

    def forward(self, x):
        h = self.pool(self.act(self.conv1(x)))
        h = self.pool(self.act(self.conv2(h)))
        h = h.view(h.size()[0], -1)
        h = self.act(self.l1(h))
        h = self.act(self.l2(h))
        h = self.l3(h)
        return h
```

プーリング層の定義
Max Poolingを採用

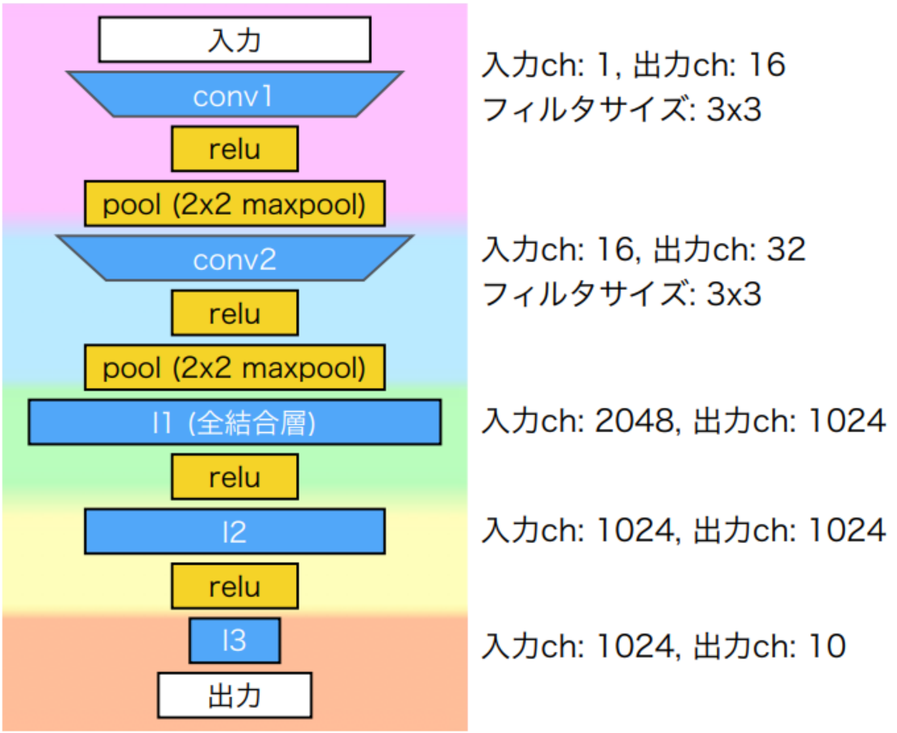
ネットワークモデルの定義

- 畳み込み層2層，全結合層3層で構成されるCNNを定義
  - __init__　：構成に必要な層を定義
  - forward　：上記で定義した層を接続して処理するように記述

```
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.l1 = nn.Linear(7*7*32, 1024)
        self.l2 = nn.Linear(1024, 1024)
        self.l3 = nn.Linear(1024, 10)
        self.act = nn.ReLU()
        self.pool = nn.MaxPool2d(2, 2)

    def forward(self, x):
        h = self.pool(self.act(self.conv1(x)))
        h = self.pool(self.act(self.conv2(h)))
        h = h.view(h.size()[0], -1)
        h = self.act(self.l1(h))
        h = self.act(self.l2(h))
        h = self.l3(h)
        return h
```

このネットワークを図で表すと……



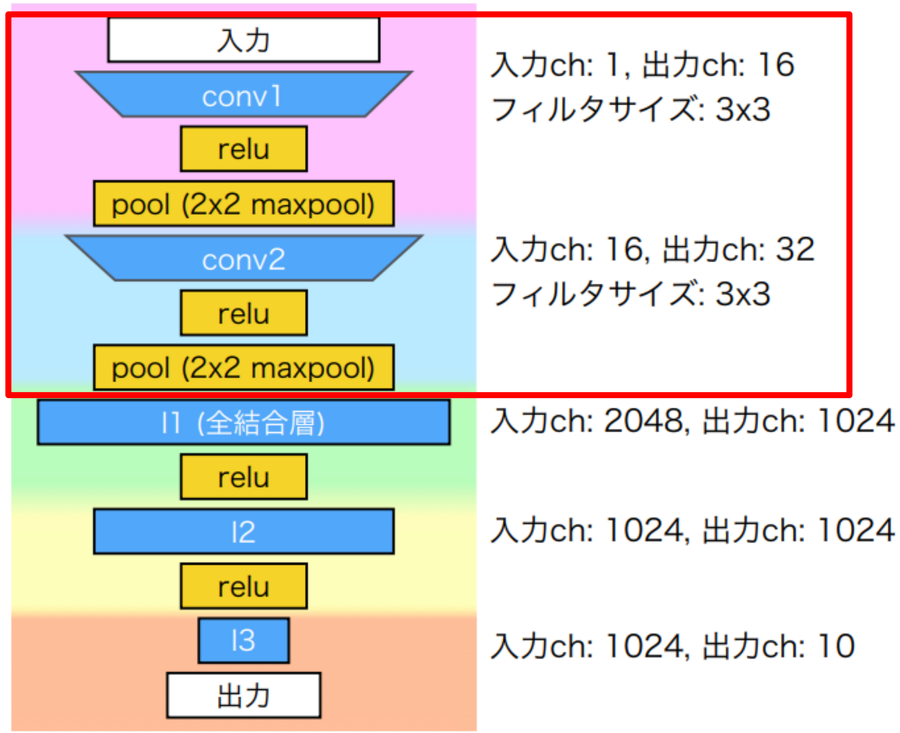
ネットワークモデルの定義

- 畳み込み層2層，全結合層3層で構成されるCNNを定義
  - __init__　：構成に必要な層を定義
  - forward　：上記で定義した層を接続して処理するように記述

```
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.l1 = nn.Linear(7*7*32, 1024)
        self.l2 = nn.Linear(1024, 1024)
        self.l3 = nn.Linear(1024, 10)
        self.act = nn.ReLU()
        self.pool = nn.MaxPool2d(2, 2)

    def forward(self, x):
        h = self.pool(self.act(self.conv1(x)))
        h = self.pool(self.act(self.conv2(h)))
        h = h.view(h.size()[0], -1)
        h = self.act(self.l1(h))
        h = self.act(self.l2(h))
        h = self.l3(h)
        return h
```

このネットワークを図で表すと……



ネットワークモデルの定義

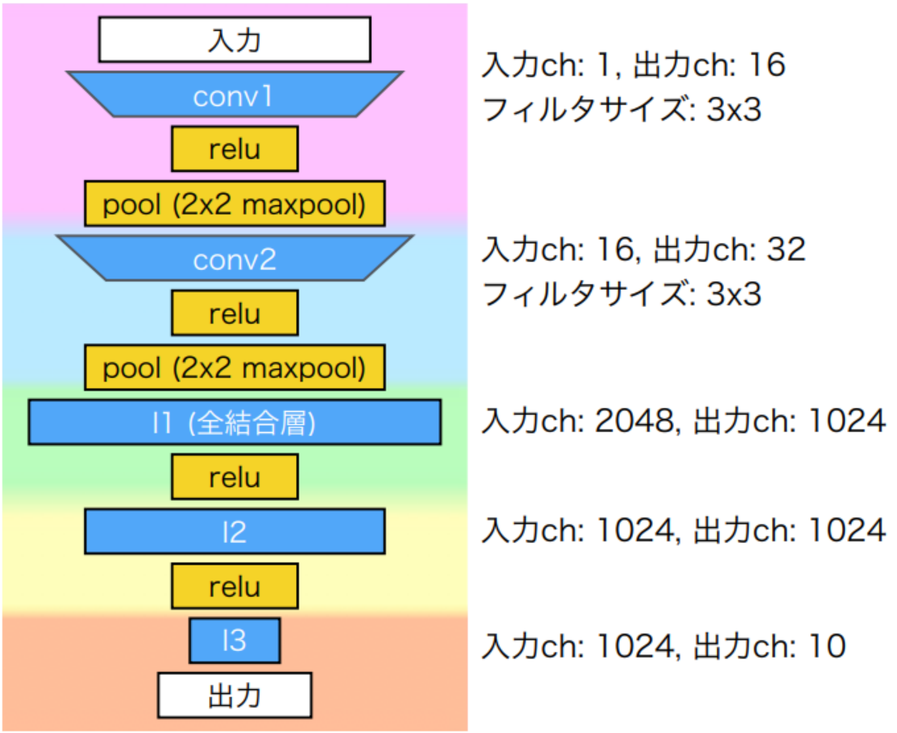
- 畳み込み層2層，全結合層3層で構成されるCNNを定義
  - __init__　：構成に必要な層を定義
  - forward　：上記で定義した層を接続して処理するように記述

```
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.l1 = nn.Linear(7*7*32, 1024)
        self.l2 = nn.Linear(1024, 1024)
        self.l3 = nn.Linear(1024, 10)
        self.act = nn.ReLU()
        self.pool = nn.MaxPool2d(2, 2)

    def forward(self, x):
        h = self.pool(self.act(self.conv1(x)))
        h = self.pool(self.act(self.conv2(h)))
        h = h.view(h.size()[0], -1)
        h = self.act(self.l1(h))
        h = self.act(self.l2(h))
        h = self.l3(h)
        return h
```

全結合層に入力するために平坦化を行う

このネットワークを図で表すと……



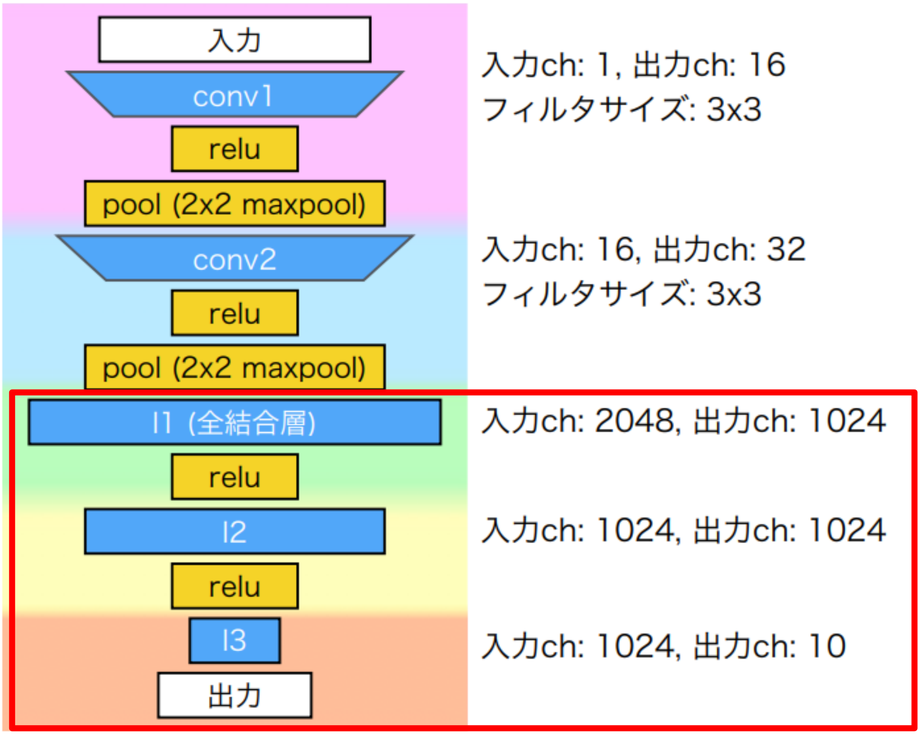
ネットワークモデルの定義

- 畳み込み層2層，全結合層3層で構成されるCNNを定義
  - __init__　：構成に必要な層を定義
  - forward　：上記で定義した層を接続して処理するように記述

```
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.l1 = nn.Linear(7*7*32, 1024)
        self.l2 = nn.Linear(1024, 1024)
        self.l3 = nn.Linear(1024, 10)
        self.act = nn.ReLU()
        self.pool = nn.MaxPool2d(2, 2)

    def forward(self, x):
        h = self.pool(self.act(self.conv1(x)))
        h = self.pool(self.act(self.conv2(h)))
        h = h.view(h.size()[0], -1)
        h = self.act(self.l1(h))
        h = self.act(self.l2(h))
        h = self.l3(h)
        return h
```

このネットワークを図で表すと……



ネットワークの作成

- 定義したCNNクラスを呼び出してネットワークモデルを作成
- 学習するための最適化手法を設定
- 定義したモデルの詳細情報をtorchsummary.summary()関数で表示

```
model = CNN()
if use_cuda:
    model.cuda()

optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

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

```
if use_cuda:
    torchsummary.summary(model, (1, 28, 28), device='cuda')
else:
    torchsummary.summary(model, (1, 28, 28), device='cpu')
```

学習(1/2)

- 読み込んだデータセットと作成したネットワークを用いて学習を行う

```
batch_size = 100
epoch_num = 10

train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True)

criterion = nn.CrossEntropyLoss()
if use_cuda:
    criterion.cuda()

model.train()
```

1回の誤差を算出するデータ数

学習(1/2)

- 読み込んだデータセットと作成したネットワークを用いて学習を行う

```
batch_size = 100
epoch_num = 10

train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True)

criterion = nn.CrossEntropyLoss()
if use_cuda:
    criterion.cuda()

model.train()
```

学習回数

学習(1/2)

- 読み込んだデータセットと作成したネットワークを用いて学習を行う

```
batch_size = 100
epoch_num = 10
```

データをバッチサイズに分けて取得できるDataLoader

```
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True)

criterion = nn.CrossEntropyLoss()
if use_cuda:
    criterion.cuda()

model.train()
```

学習(1/2)

- 読み込んだデータセットと作成したネットワークを用いて学習を行う

```
batch_size = 100
epoch_num = 10

train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True)

criterion = nn.CrossEntropyLoss()
if use_cuda:
    criterion.cuda()

model.train()
```

誤差関数

学習 (1/2)

- 読み込んだデータセットと作成したネットワークを用いて学習を行う

```
batch_size = 100
epoch_num = 10

train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=True)

criterion = nn.CrossEntropyLoss()
if use_cuda:
    criterion.cuda()

model.train()
```

モデルを学習モードに移行

学習 (2/2)

- 読み込んだデータセットと作成したネットワークを用いて学習を行う

```
train_start = time()
for epoch in range(1, epoch_num+1):
    sum_loss = 0.0
    count = 0

    for image, label in train_loader:

        if use_cuda:
            image = image.cuda()
            label = label.cuda()

        y = model(image)

        loss = criterion(y, label)
        model.zero_grad()
        loss.backward()
        optimizer.step()

        sum_loss += loss.item()
        pred = torch.argmax(y, dim=1)
        count += torch.sum(pred == label)
```

時間計測の開始

学習 (2/2)

- 読み込んだデータセットと作成したネットワークを用いて学習を行う

```
train_start = time()
for epoch in range(1, epoch_num+1):
    sum_loss = 0.0
    count = 0

    for image, label in train_loader:

        if use_cuda:
            image = image.cuda()
            label = label.cuda()

        y = model(image)

        loss = criterion(y, label)
        model.zero_grad()
        loss.backward()
        optimizer.step()

        sum_loss += loss.item()
        pred = torch.argmax(y, dim=1)
        count += torch.sum(pred == label)
```

epochで指定した回数分学習を回す

学習 (2/2)

- 読み込んだデータセットと作成したネットワークを用いて学習を行う

```
train_start = time()
for epoch in range(1, epoch_num+1):
    sum_loss = 0.0
    count = 0

    for image, label in train_loader:

        if use_cuda:
            image = image.cuda()
            label = label.cuda()

        y = model(image)

        loss = criterion(y, label)
        model.zero_grad()
        loss.backward()
        optimizer.step()

        sum_loss += loss.item()
        pred = torch.argmax(y, dim=1)
        count += torch.sum(pred == label)
```

epoch毎に学習中の誤差(sum_loss)・正解回数(count)をリセット

学習 (2/2)

- 読み込んだデータセットと作成したネットワークを用いて学習を行う

```
train_start = time()
for epoch in range(1, epoch_num+1):
    sum_loss = 0.0
    count = 0

    for image, label in train_loader:

        if use_cuda:
            image = image.cuda()
            label = label.cuda()

        y = model(image)

        loss = criterion(y, label)
        model.zero_grad()
        loss.backward()
        optimizer.step()

        sum_loss += loss.item()
        pred = torch.argmax(y, dim=1)
        count += torch.sum(pred == label)
```

DataLoaderにより，バッチごとに画像とラベルを取り出す

学習 (2/2)

- 読み込んだデータセットと作成したネットワークを用いて学習を行う

```
train_start = time()
for epoch in range(1, epoch_num+1):
    sum_loss = 0.0
    count = 0

    for image, label in train_loader:

        if use_cuda:
            image = image.cuda()
            label = label.cuda()

        y = model(image)

        loss = criterion(y, label)
        model.zero_grad()
        loss.backward()
        optimizer.step()

        sum_loss += loss.item()
        pred = torch.argmax(y, dim=1)
        count += torch.sum(pred == label)
```

モデルにデータ(image)を入力し，出力(y)を得る

学習 (2/2)

- 読み込んだデータセットと作成したネットワークを用いて学習を行う

```
train_start = time()
for epoch in range(1, epoch_num+1):
    sum_loss = 0.0
    count = 0

    for image, label in train_loader:

        if use_cuda:
            image = image.cuda()
            label = label.cuda()

        y = model(image)

        loss = criterion(y, label)
        model.zero_grad()
        loss.backward()
        optimizer.step()

        sum_loss += loss.item()
        pred = torch.argmax(y, dim=1)
        count += torch.sum(pred == label)
```

誤差計算(loss)
逆伝播
パラメータの更新

学習 (2/2)

- 読み込んだデータセットと作成したネットワークを用いて学習を行う

```
train_start = time()
for epoch in range(1, epoch_num+1):
    sum_loss = 0.0
    count = 0

    for image, label in train_loader:

        if use_cuda:
            image = image.cuda()
            label = label.cuda()

        y = model(image)

        loss = criterion(y, label)
        model.zero_grad()
        loss.backward()
        optimizer.step()

        sum_loss += loss.item()
        pred = torch.argmax(y, dim=1)
        count += torch.sum(pred == label)
```

Lossの合計値(sum_loss)の算出
モデルの予測クラス(perd)の算出
正解枚数の算出 (count)
これらをprintすることで，epoch毎に学習の経過を観察することが可能

テスト

- 学習したネットワークモデルの評価を行う
  - 基本的な処理は学習時と同じ
  - 誤差計算や勾配計算，パラメータの更新は行わない

```
test_loader = torch.utils.data.DataLoader(test_data, batch_size=100, shuffle=False)
```

テスト用データのためにDataLoaderを定義
ここで学習用データを用いると正しい評価が不可能

```
model.eval()

count = 0
with torch.no_grad():
    for image, label in test_loader:

        if use_cuda:
            image = image.cuda()
            label = label.cuda()

        y = model(image)

        pred = torch.argmax(y, dim=1)
        count += torch.sum(pred == label)

print("test accuracy: {}".format(count.item() / 10000.))
```

テスト

- 学習したネットワークモデルの評価を行う
  - 基本的な処理は学習時と同じ
  - 誤差計算や勾配計算，パラメータの更新は行わない

```
test_loader = torch.utils.data.DataLoader(test_data, batch_size=100, shuffle=False)

model.eval()

count = 0
with torch.no_grad():
    for image, label in test_loader:

        if use_cuda:
            image = image.cuda()
            label = label.cuda()

        y = model(image)

        pred = torch.argmax(y, dim=1)
        count += torch.sum(pred == label)

print("test accuracy: {}".format(count.item() / 10000.))
```

モデルを評価モードに移行

テスト

- 学習したネットワークモデルの評価を行う
  - 基本的な処理は学習時と同じ
  - 誤差計算や勾配計算，パラメータの更新は行わない

```
test_loader = torch.utils.data.DataLoader(test_data, batch_size=100, shuffle=False)

model.eval()

count = 0
with torch.no_grad():
    for image, label in test_loader:

        if use_cuda:
            image = image.cuda()
            label = label.cuda()

        y = model(image)

        pred = torch.argmax(y, dim=1)
        count += torch.sum(pred == label)

print("test accuracy: {}".format(count.item() / 10000.))
```

正解数(count)の定義と初期化

テスト

- 学習したネットワークモデルの評価を行う
  - 基本的な処理は学習時と同じ
  - 誤差計算や勾配計算，パラメータの更新は行わない

```
test_loader = torch.utils.data.DataLoader(test_data, batch_size=100, shuffle=False)

model.eval()

count = 0
with torch.no_grad():
    for image, label in test_loader:

        if use_cuda:
            image = image.cuda()
            label = label.cuda()

        y = model(image)

        pred = torch.argmax(y, dim=1)
        count += torch.sum(pred == label)

print("test accuracy: {}".format(count.item() / 10000.))
```

勾配更新は行わない設定にする

テスト

- 学習したネットワークモデルの評価を行う
  - 基本的な処理は学習時と同じ
  - 誤差計算や勾配計算，パラメータの更新は行わない

```
test_loader = torch.utils.data.DataLoader(test_data, batch_size=100, shuffle=False)

model.eval()

count = 0
with torch.no_grad():
    for image, label in test_loader:

        if use_cuda:
            image = image.cuda()
            label = label.cuda()

        y = model(image)

        pred = torch.argmax(y, dim=1)
        count += torch.sum(pred == label)

print("test accuracy: {}".format(count.item() / 10000.))
```

バッチごとにテスト用データとラベルを取り出す

テスト

- 学習したネットワークモデルの評価を行う
  - 基本的な処理は学習時と同じ
  - 誤差計算や勾配計算，パラメータの更新は行わない

```
test_loader = torch.utils.data.DataLoader(test_data, batch_size=100, shuffle=False)

model.eval()

count = 0
with torch.no_grad():
    for image, label in test_loader:

        if use_cuda:
            image = image.cuda()
            label = label.cuda()

        y = model(image)

        pred = torch.argmax(y, dim=1)
        count += torch.sum(pred == label)

print("test accuracy: {}".format(count.item() / 10000.))
```

モデルに画像(image)を入れ，出力(y)を得る

テスト

- 学習したネットワークモデルの評価を行う
  - 基本的な処理は学習時と同じ
  - 誤差計算や勾配計算，パラメータの更新は行わない

```
test_loader = torch.utils.data.DataLoader(test_data, batch_size=100, shuffle=False)

model.eval()

count = 0
with torch.no_grad():
    for image, label in test_loader:

        if use_cuda:
            image = image.cuda()
            label = label.cuda()

        y = model(image)

        pred = torch.argmax(y, dim=1)
        count += torch.sum(pred == label)

print("test accuracy: {}".format(count.item() / 10000.))
```

モデルの予測クラス(pred)の算出
正解数(count)のカウント

テスト

- 学習したネットワークモデルの評価を行う
  - 基本的な処理は学習時と同じ
  - 誤差計算や勾配計算，パラメータの更新は行わない

```
test_loader = torch.utils.data.DataLoader(test_data, batch_size=100, shuffle=False)

model.eval()

count = 0
with torch.no_grad():
    for image, label in test_loader:

        if use_cuda:
            image = image.cuda()
            label = label.cuda()

        y = model(image)

        pred = torch.argmax(y, dim=1)
        count += torch.sum(pred == label)

print("test accuracy: {}".format(count.item() / 10000.))
```

正解数からテスト用データの総数を割ることで
正解率の算出

課題

- CIFAR-10データセットを用いた画像分類を行う
  - URL：CIFAR-10 CNN

1. ネットワークの構造を変更し，認識精度の変化を確認する
   - 中間層のユニット数や，層数，活性化関数などを変更
2. 学習の設定を変更し，認識精度の変化を確認
   - バッチサイズ，学習回数，学習率，最適化手法などを変更
3. 認識精度が向上するように1,2を変更
   - 色々やってみてより高い認識精度を目指す

課題の提出

- 提出物
  1. 実験過程や結果，考察などをまとめたレポート
     - WordやTexで作成し，pdf化したもの
  2. 最も精度が良かった時のファイル
     - **工夫した点をコメントで書くこと**
     - Colabを.ipynbファイルでダウンロードし，Githubにアップロード
     - 「ファイル」➡「ダウンロード」➡「.ipynbをダウンロード」
  - ファイル名の指定は特に無し，例が作成してあるので参考にしてください

- 提出場所
  - Github　：spring_seminar_2023/07_sminar_課題提出/学籍番号_漢字名前/ ~(.pdf/.ipynb)
  - URL　　：spring_seminar_2023/07_seminar_課題提出/

- 提出期限
  - 4/3 (月)