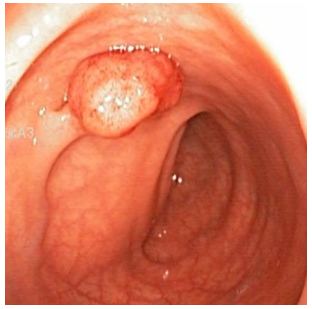
深層学習入門

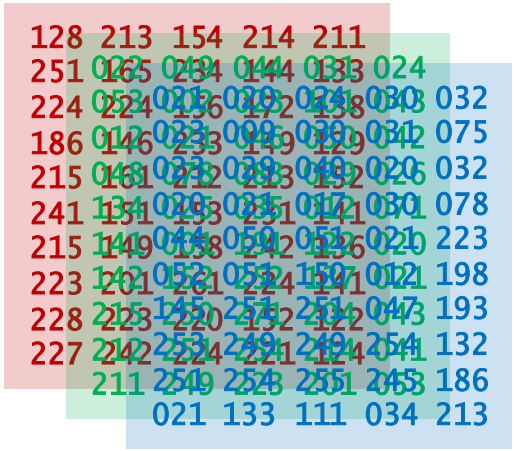
3. 物体分類

物体分類

PyTorch

物体分類



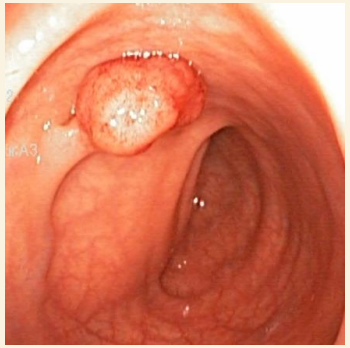


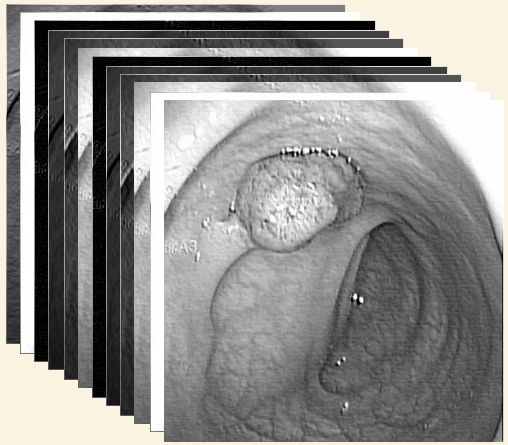
特徴抽出

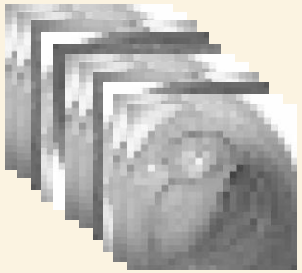
102
073
122
221
104
094
123
223

分類

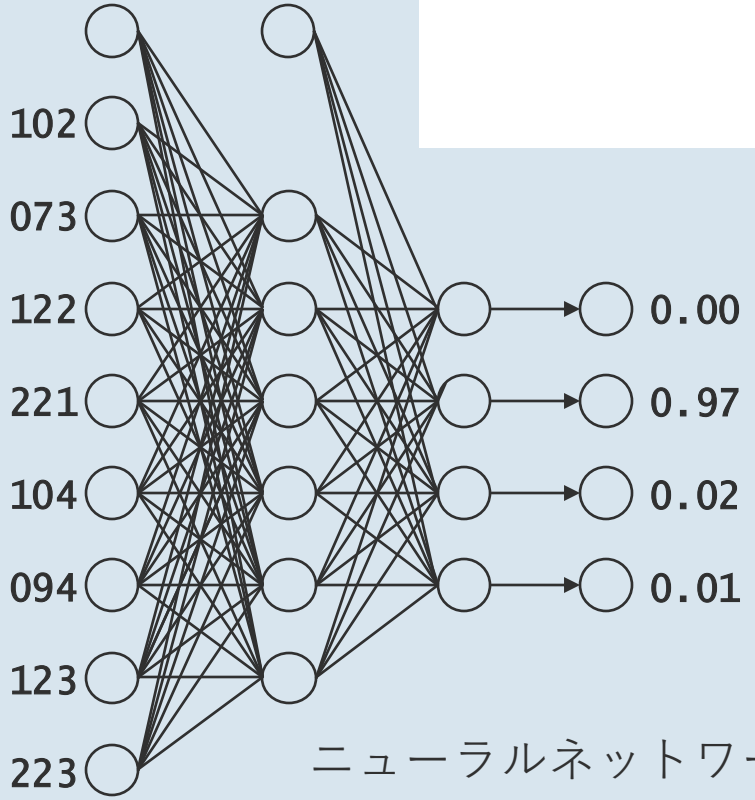
normal 0.00
polyps 0.97
esophagitis 0.02
colitis 0.01







畳み込み



ニューラルネットワーク

畳み込み演算

0	0	0	0	0	0	0	0
0	5	4	3	3	4	5	0
0	3	4	3	4	3	3	0
0	3	5	3	1	1	2	0
0	2	2	2	0	0	1	0
0	1	3	2	1	1	1	0
0	1	2	1	0	1	0	0
0	0	0	0	0	0	0	0

フィルタ

1	0	0
0	0	0
0	0	1

torch.nn.Conv2d()

4	3	4	3	3	0
5	8	5	4	5	4
2	5	4	3	5	3
3	5	6	4	2	1
2	3	2	3	0	0
0	1	3	2	1	1

最大プーリング演算

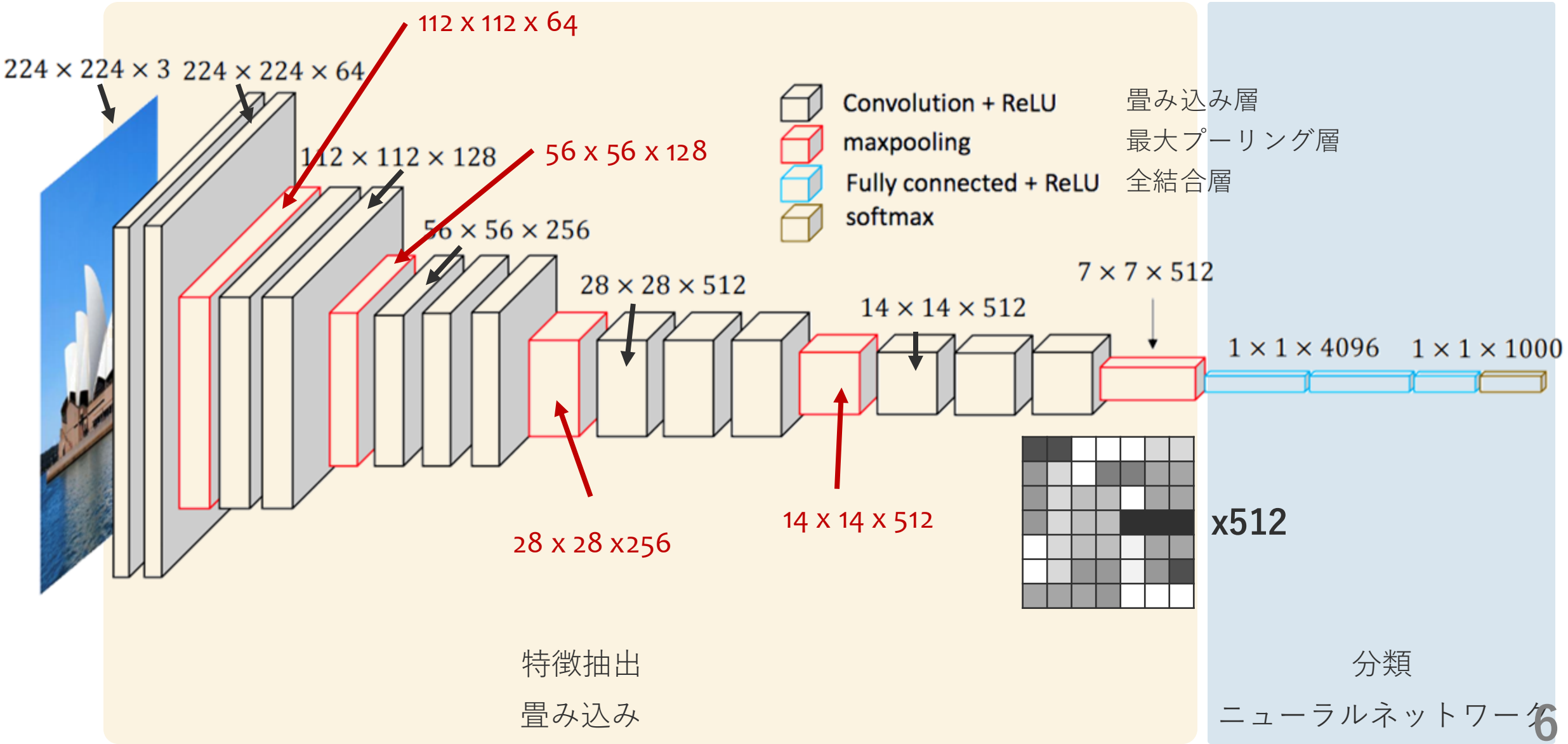
5	4	3	3	4	5
3	4	3	4	3	3
3	5	3	1	1	2
2	2	2	0	0	1
1	3	2	1	1	1
1	2	1	0	1	0

torch.nn.MaxPool2d()

5	4	5
5	3	2
3	2	1

VGG16

https://doi.org/10.1145/3038912.3052638



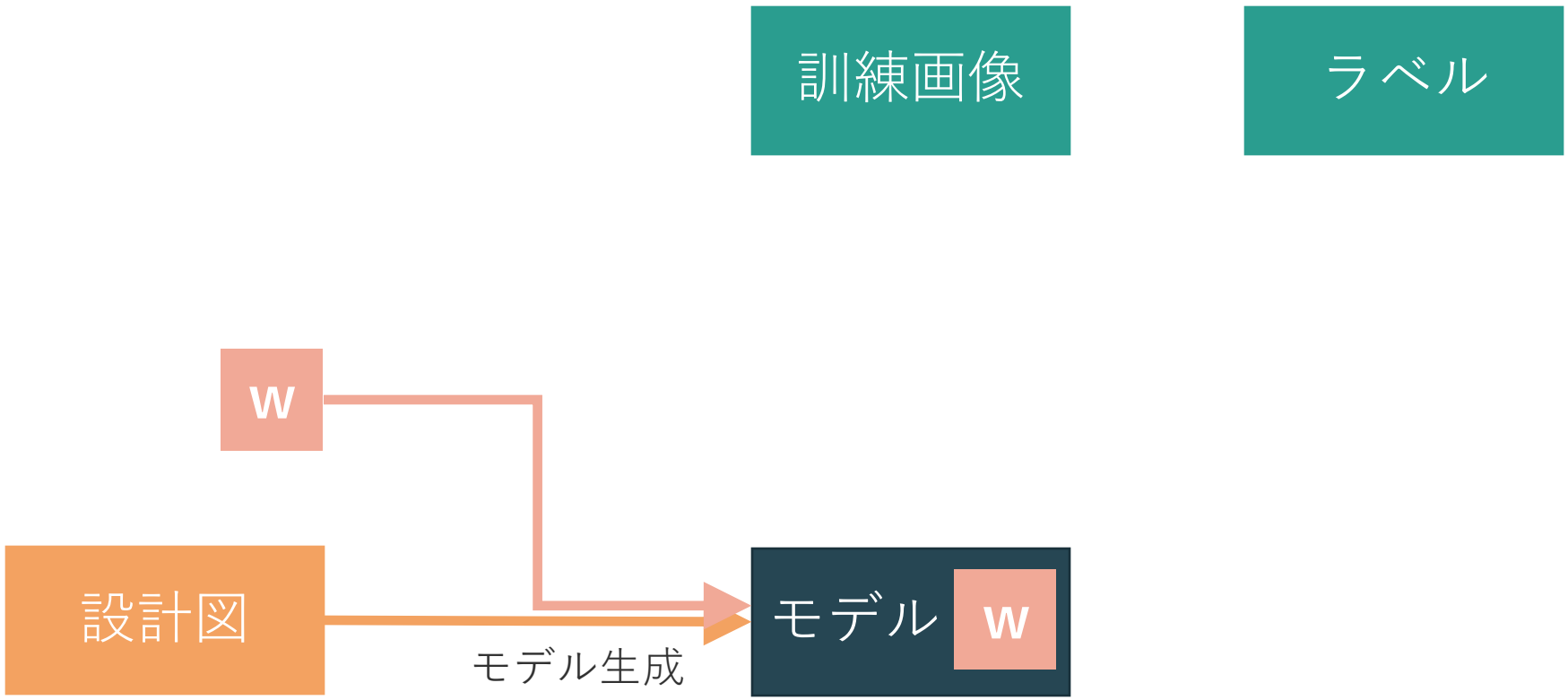
訓練プロセス

訓練画像

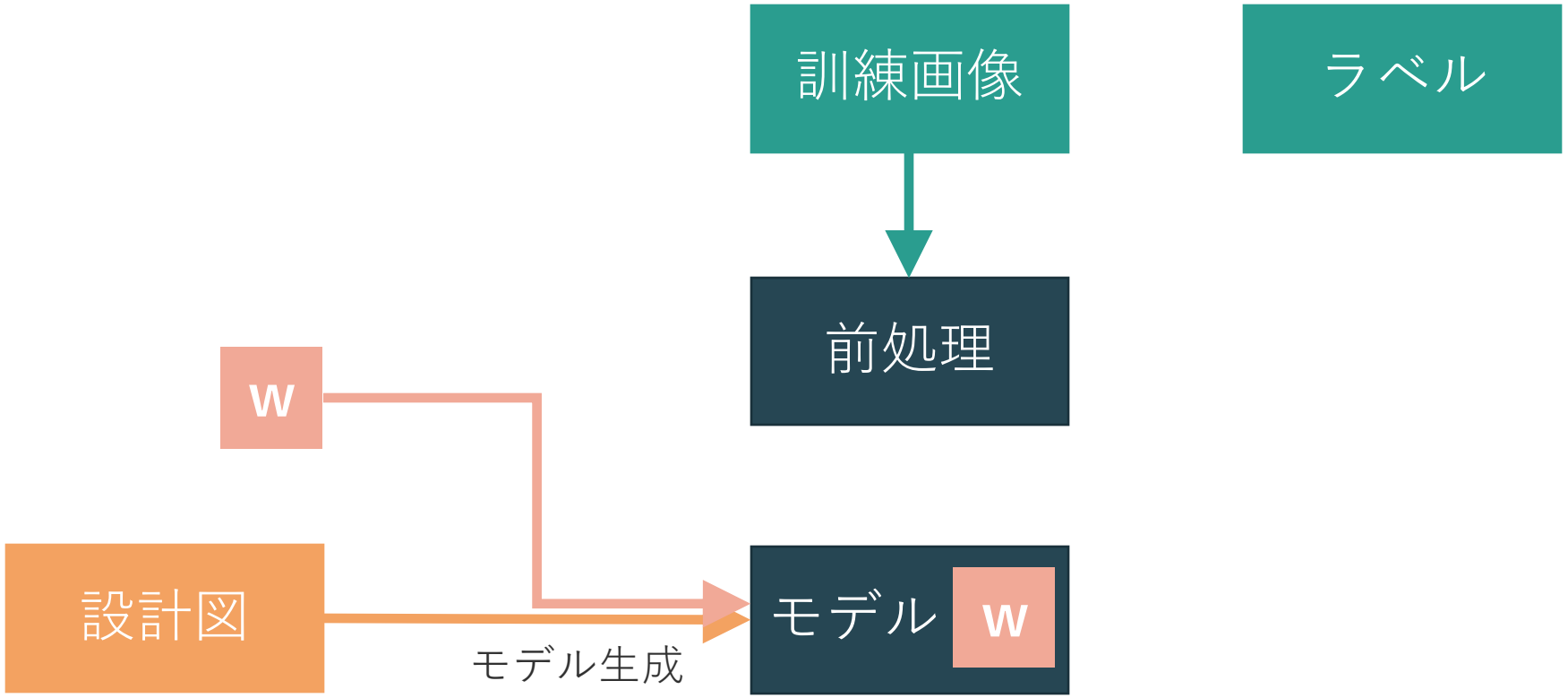
ラベル

daisy
daisy
daisy
dandeliion
dandeliion
roses
roses
sunflowers
daisy
tulips
tulips
tulips
sunflowers
dandeliion
roses
dandeliion
sunflowers
roses

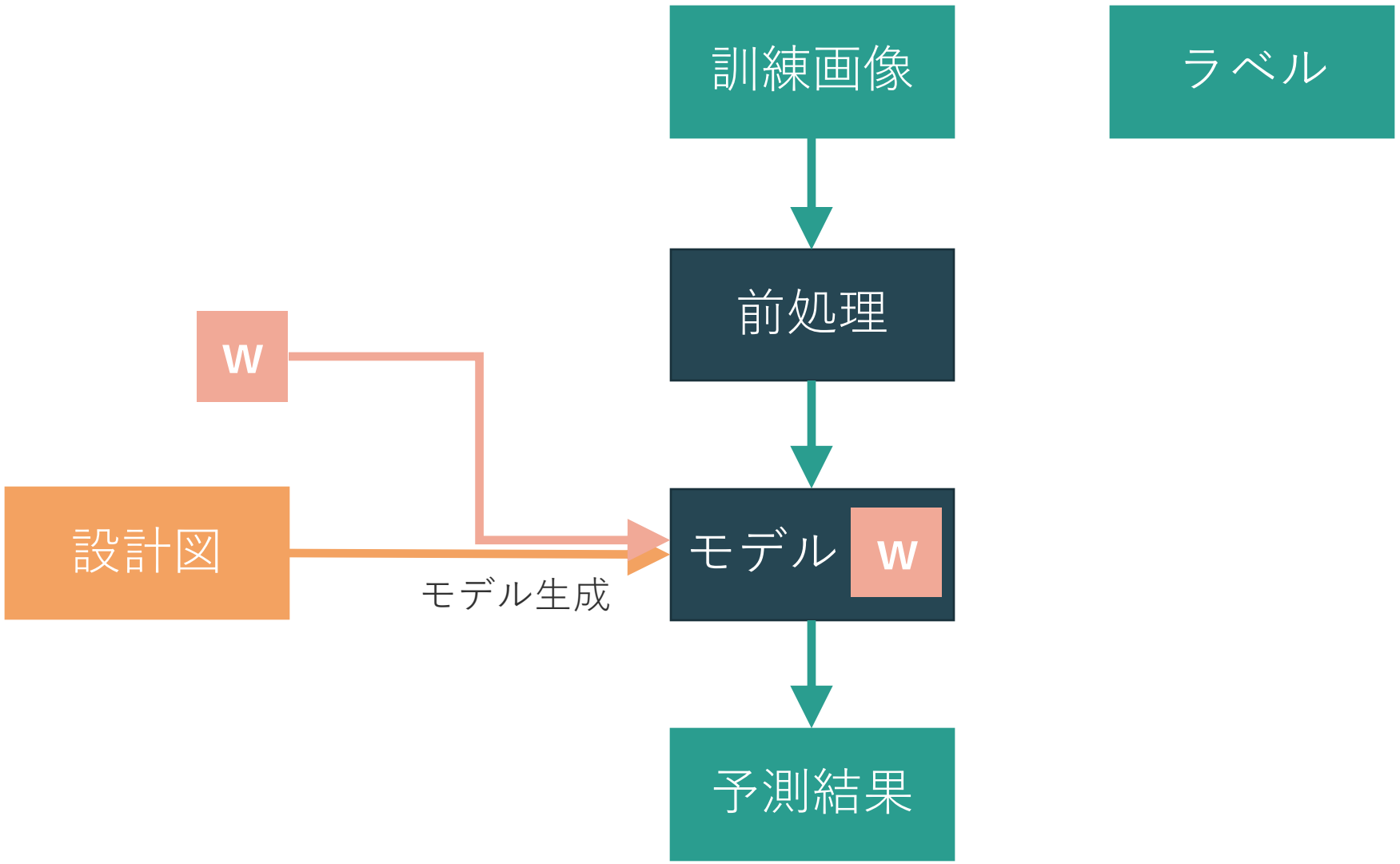
訓練プロセス



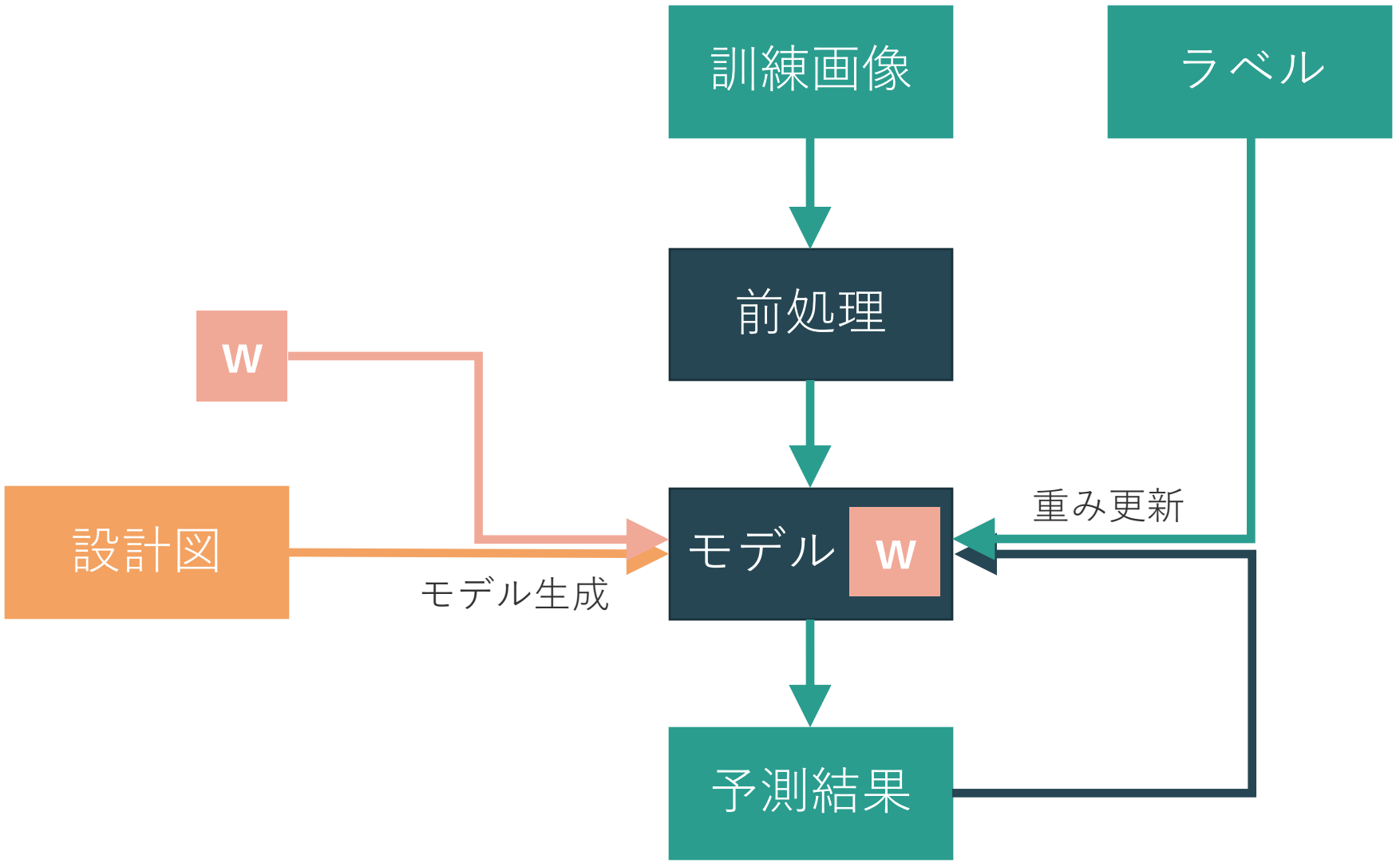
訓練プロセス



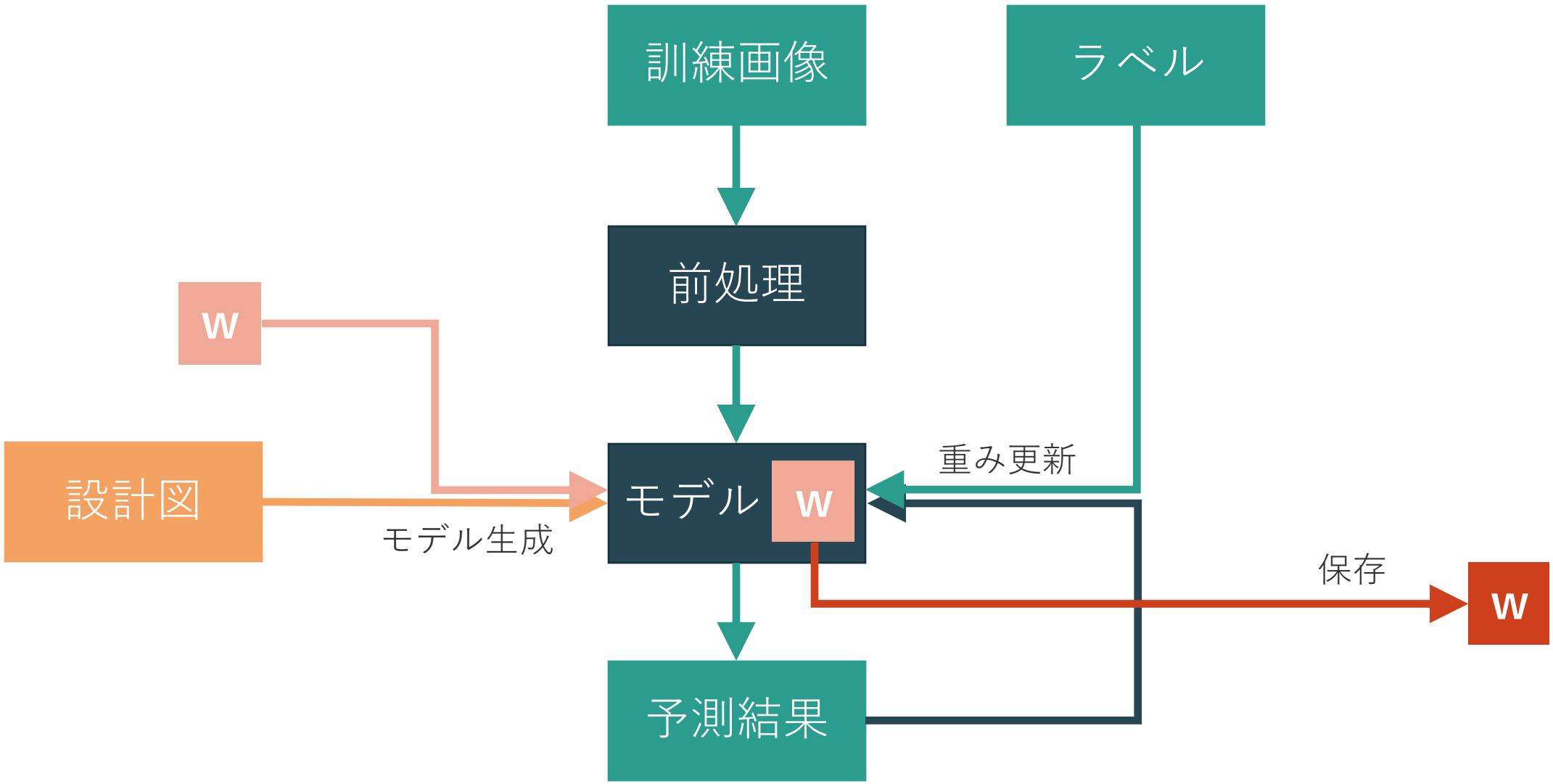
訓練プロセス



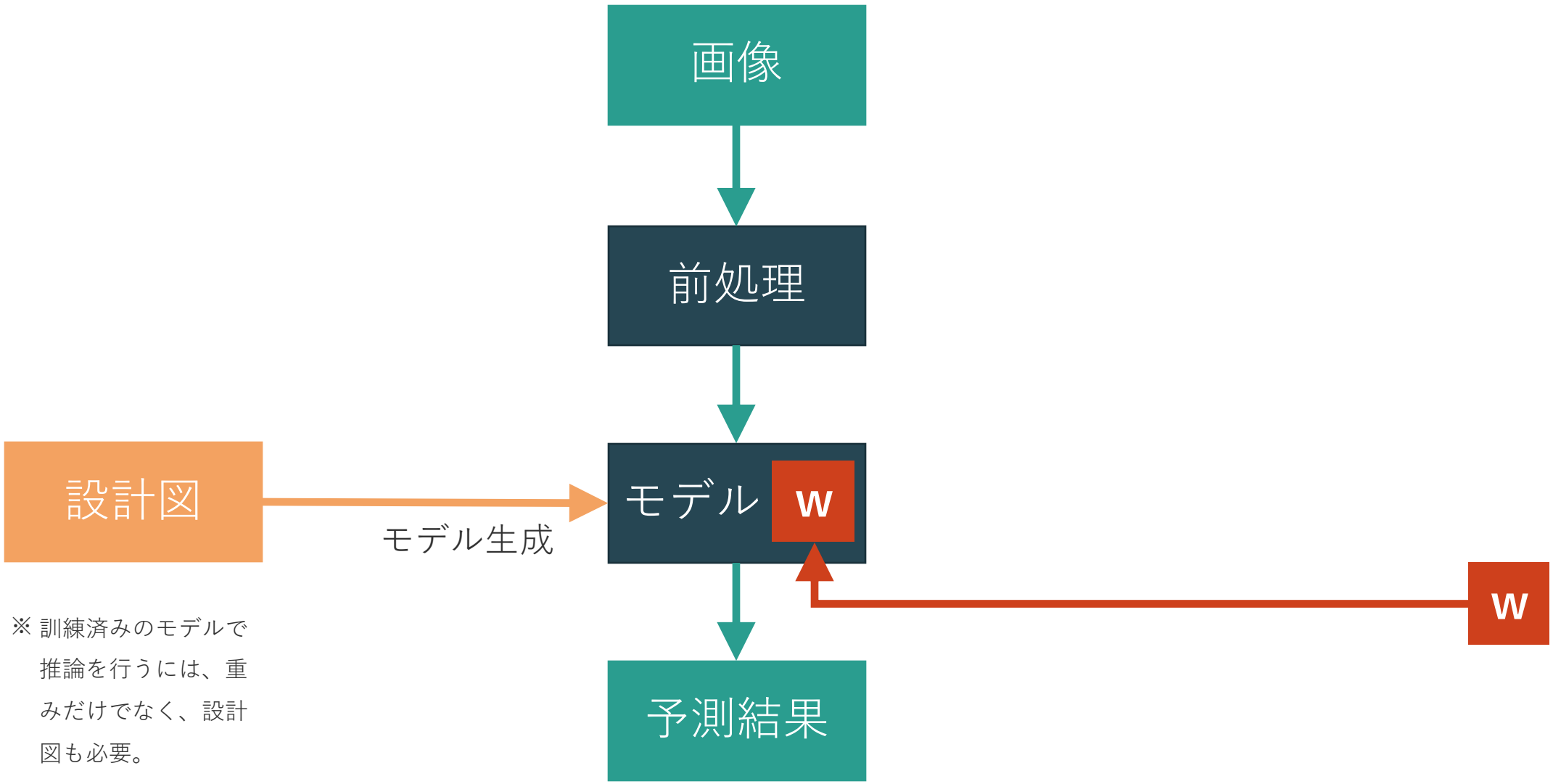
訓練プロセス



訓練プロセス



推論プロセス



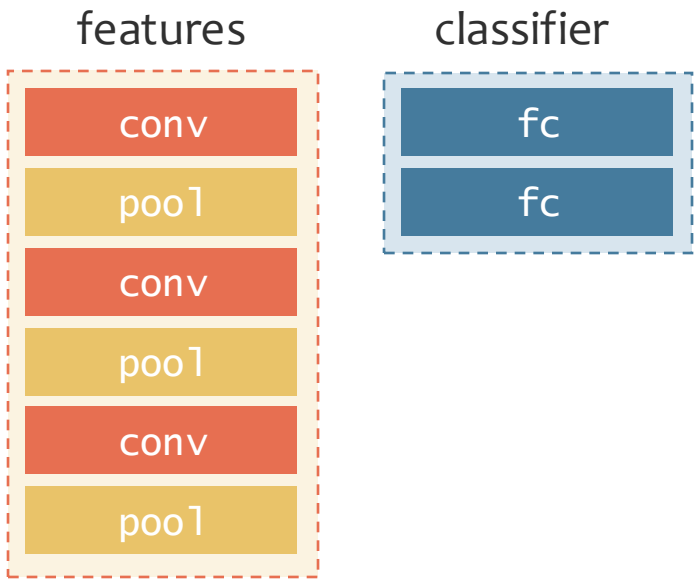
※ 訓練済みのモデルで推論を行うには、重みだけでなく、設計図も必要。

モデル設計

```
class LiteCNN(torch.nn.Module):
def __init__(self, num_classes, input_size=224):
    super().__init__()

    self.features = torch.nn.Sequential(
        torch.nn.Conv2d(3, 32, kernel_size=5),
        torch.nn.ReLU(),
        torch.nn.MaxPool2d(kernel_size=2, stride=2),
        torch.nn.Conv2d(32, 64, kernel_size=5),
        torch.nn.ReLU(),
        torch.nn.MaxPool2d(kernel_size=2, stride=2),
        torch.nn.Conv2d(64, 128, kernel_size=5),
        torch.nn.ReLU(),
        torch.nn.MaxPool2d(kernel_size=2, stride=2)
    )

    with torch.no_grad():
        n_features = self.features(torch.zeros(1, 3, input_size, input_size)).numel()

    self.classifier = torch.nn.Sequential(
        torch.nn.Linear(n_features, 2048),
        torch.nn.ReLU(),
        torch.nn.Dropout(0.5),
        torch.nn.Linear(2048, num_classes)
    )


    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x
```

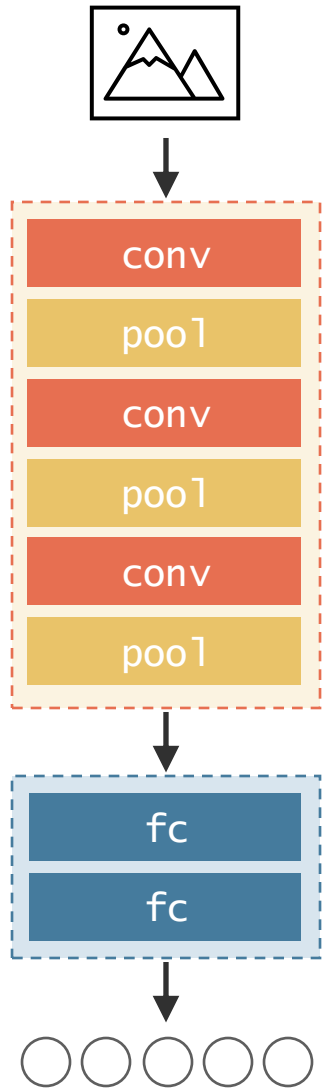
モデル設計

```
class LiteCNN(torch.nn.Module):
    def __init__(self, num_classes, input_size=224):
        super().__init__()

        self.features = torch.nn.Sequential(
            torch.nn.Conv2d(3, 32, kernel_size=5),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2),
            torch.nn.Conv2d(32, 64, kernel_size=5),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2),
            torch.nn.Conv2d(64, 128, kernel_size=5),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2)
        )

        with torch.no_grad():
            n_features = self.features(torch.zeros(1, 3, input_size, input_size)).numel()

        self.classifier = torch.nn.Sequential(
            torch.nn.Linear(n_features, 2048),
            torch.nn.ReLU(),
            torch.nn.Dropout(0.5),
            torch.nn.Linear(2048, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x
```



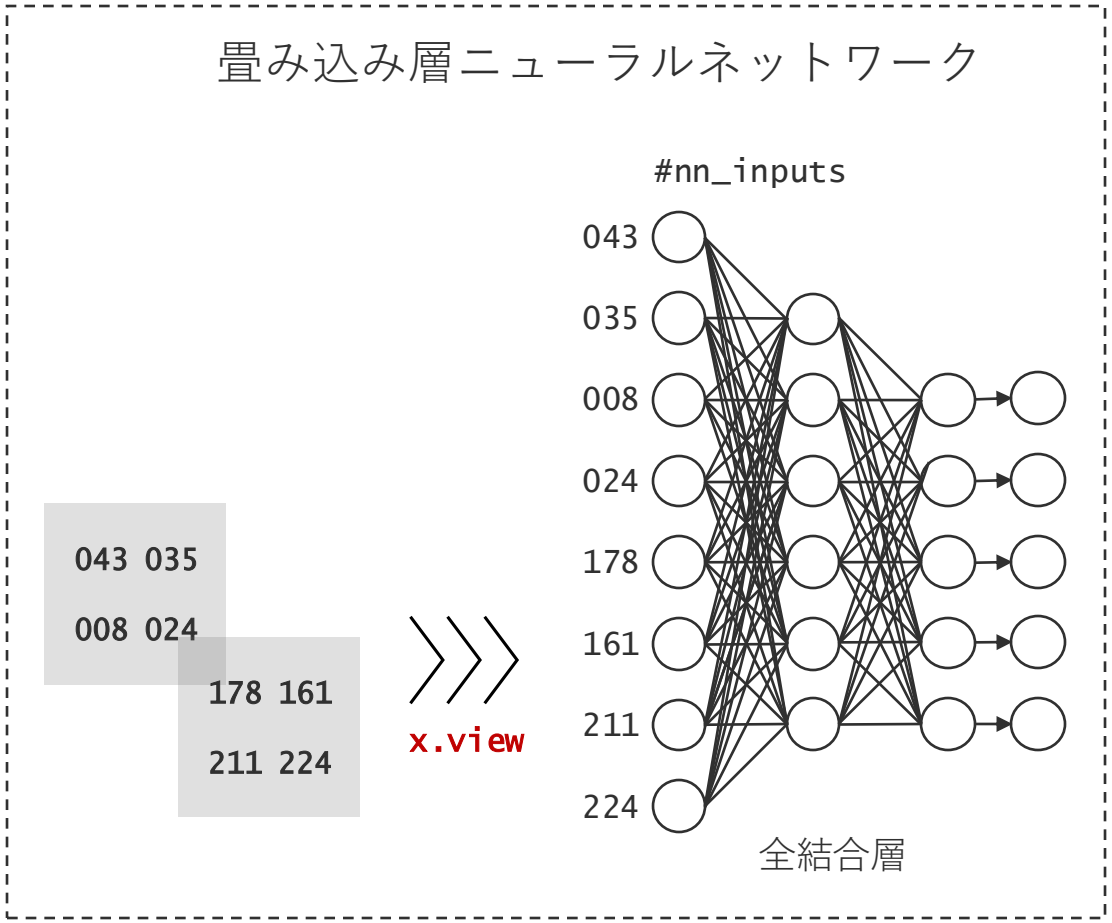
モデル設計

```
class LiteCNN(torch.nn.Module):
    def __init__(self, num_classes, input_size=224):
        super().__init__()

        self.features = torch.nn.Sequential(
            torch.nn.Conv2d(3, 32, kernel_size=5),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2),
            torch.nn.Conv2d(32, 64, kernel_size=5),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2),
            torch.nn.Conv2d(64, 128, kernel_size=5),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2)
        )

        with torch.no_grad():
            n_features = self.features(torch.zeros(1, 3, input_size, input_size)).numel()

        self.classifier = torch.nn.Sequential(
            torch.nn.Linear(n_features, 2048),
            torch.nn.ReLU(),
            torch.nn.Dropout(0.5),
            torch.nn.Linear(2048, num_classes)
        )


    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x
```



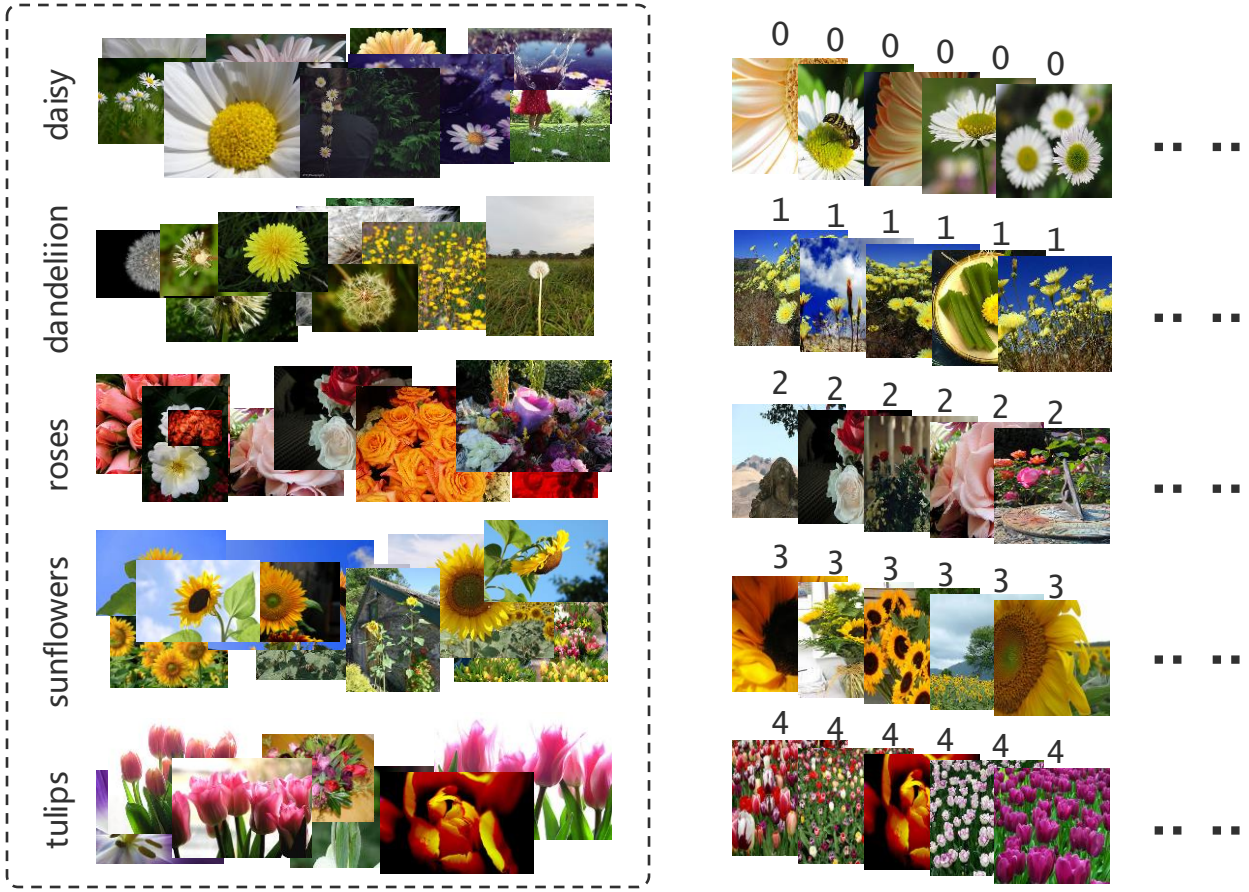
モデル設計

```
class LiteCNN(torch.nn.Module):
    def __init__(self, num_classes, input_size=224):
        super().__init__()

        self.features = torch.nn.Sequential(
            torch.nn.Conv2d(3, 32, kernel_size=5),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2),
            torch.nn.Conv2d(32, 64, kernel_size=5),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2),
            torch.nn.Conv2d(64, 128, kernel_size=5),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=2, stride=2)
        )

        with torch.no_grad():
            n_features = self.features(torch.zeros(1, 3, input_size, input_size)).numel()

        self.classifier = torch.nn.Sequential(
            torch.nn.Linear(n_features, 2048),
            torch.nn.ReLU(),
            torch.nn.Dropout(0.5),
            torch.nn.Linear(2048, num_classes)
        )


    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x
```



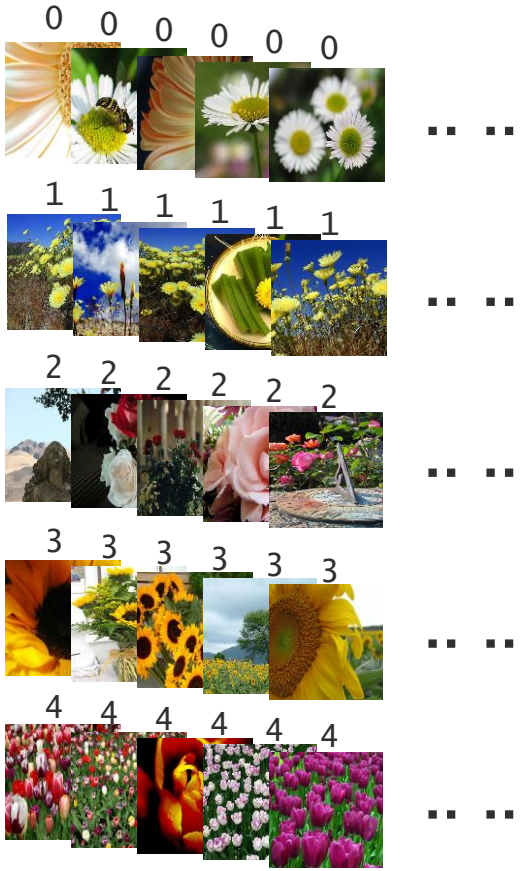
前処理


画像
前処理
バッチ化
モデル


daisy
dandeliion
roses
sunflowers
tulips
0 0 0 0 0 0
1 1 1 1 1 1
2 2 2 2 2 2
3 3 3 3 3 3
4 4 4 4 4 4
3 4 3 1
#1
0 1 2 0
#2
4 2 4 0
#3
2 2 4 0
#4

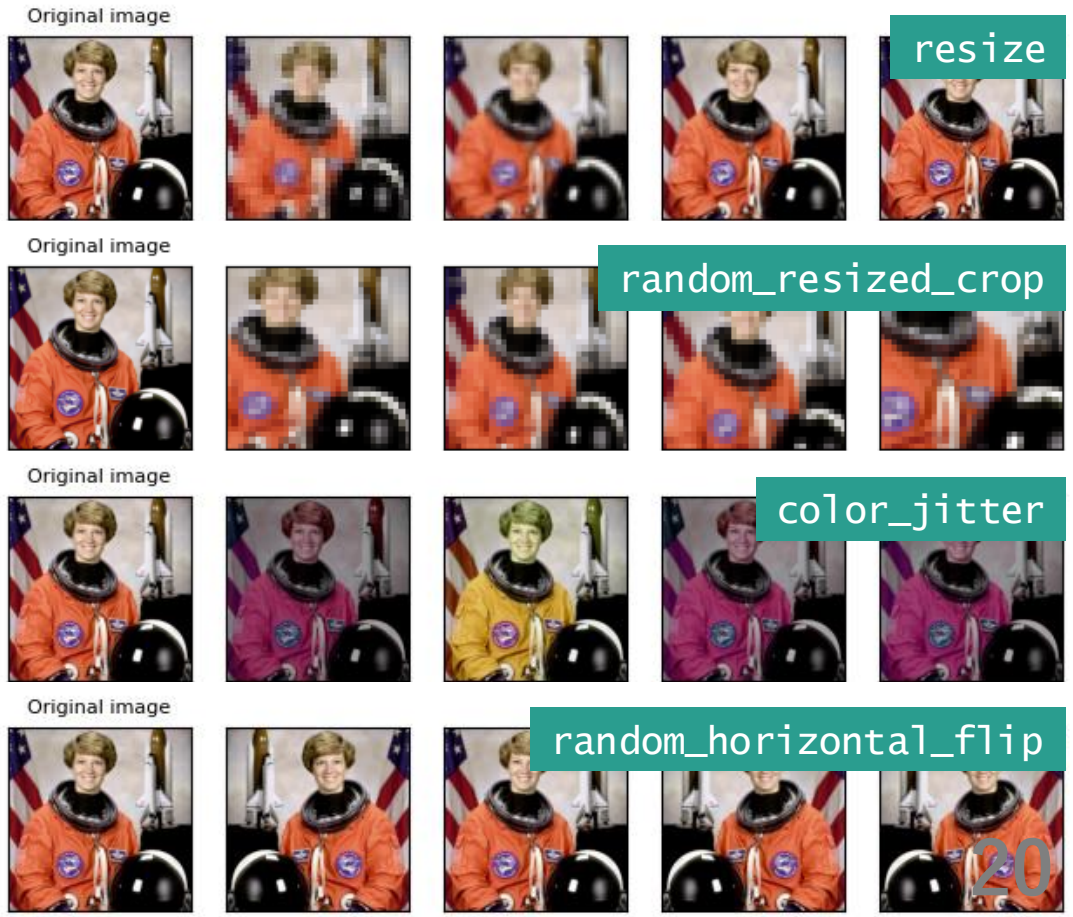
前処理

- 訓練画像を整理し、画像と教師ラベルの対応づけを行う、全訓練画像のリストを作成する。
- 画像に対する前処理を定義する。
  - 画像を行列データとして読み取る
  - サイズ変更、色調調整、アフィン変換
  - テンソルに変換
- リストの i 番目の画像がリクエストされたら、その画像を前処理して返す。



前処理

- 訓練画像を整理し、画像と教師ラベルの対応づけを行う、全訓練画像のリストを作成する。
- 画像に対する前処理を定義する。
  - 画像を行列データとして読み取る
  - サイズ変更、色調調整、アフィン変換
  - テンソルに変換
- リストの i 番目の画像がリクエストされたら、その画像を前処理して返す。



前処理

- 訓練画像を整理し、画像と教師ラベルの対応づけを行う、全訓練画像のリストを作成する。
- 画像に対する前処理を定義する。
  - 画像を行列データとして読み取る
  - サイズ変更、色調調整、アフィン変換
  - テンソルに変換
- リストの i 番目の画像がリクエストされたら、その画像を前処理して返す。

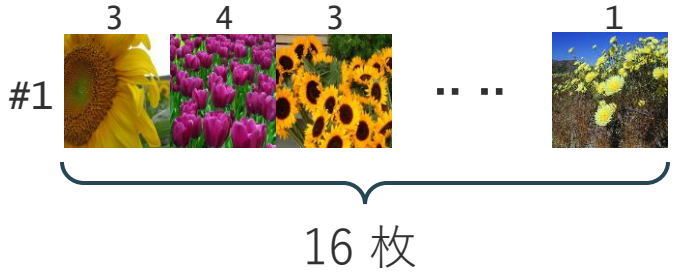
```
class Dataset:
  def __init__(images, labels):
    self.x = images
    self.y = labels
    self.transform = transforms

  def __len__():
    retrun len(self.x)

  def __getitem__(i):
    x = self.transform(self.x[i])
    y = self.y[i]
    return x, y
```

```
transforms = [
  Resize(),
  Affine(),
  Tensor(),
]
```

バッチ化

- 画像リストを取得し、必要に応じて順番を調整し、バッチ化の処理を行う。

```
train_dataset = Dataset(images, labels)

train_dataloader = DataLoader(
  train_dataset,
  batch_size=16, shuffle=True, num_workers=2)
```




```
class Dataset:
  def __init__(images, labels):
    self.x = images
    self.y = labels
    self.transform = transforms

  def __len__():
    retrun len(self.x)

  def __getitem__(i):
    x = self.transform(self.x[i])
    y = self.y[i]
    return x, y
```

```
transforms = [
  Resize(),
  Affine(),
  Tensor(),
]
```

モデル訓練

- 小分けした画像データを少しずつモデルに代入し訓練を行う。

```
train_dataset = Dataset(images, labels)

train_dataloader = DataLoader(
  train_dataset,
  batch_size=16, shuffle=True, num_workers=2)
```

```
for inputs, labels in train_dataloder:
  optimizer.zero_grad()
  outputs = model(inputs)
  loss = criterion(outputs, inputs)
  loss.backward()
  optimizer.step()
```

```
class Dataset:
  def __init__(images, labels):
    self.x = images
    self.y = labels
    self.transform = transforms

  def __len__():
    retrun len(self.x)

  def __getitem__(i):
    x = self.transform(self.x[i])
    y = self.y[i]
    return x, y
```

```
transforms = [
  Resize(),
  Affine(),
  Tensor(),
]
```

モデル訓練

```
# dataset
train_dataset = Dataset(images, labels)
train_dataloader = DataLoader(train_dataset, batch_size=4)

# model
model = LiteCNN()
model.train()

# parameters
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.001)
n_epochs = 50

# training
for epoch in range(n_epochs):
  running_loss = 0.0

  # mini batch
  for inputs, labels in train_dataloader:
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
    running_loss += loss.item()
```

- モデルを訓練モードに切り替え、計算時に生じた勾配情報を保存する。

モデル訓練

```
# dataset
train_dataset = Dataset(images, labels)
train_dataloader = DataLoader(train_dataset, batch_size=4)

# model
model = LiteCNN()
model.train()

# parameters
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.001)
n_epochs = 50

# training
for epoch in range(n_epochs):
  running_loss = 0.0

  # mini batch
  for inputs, labels in train_dataloader:
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
    running_loss += loss.item()
```

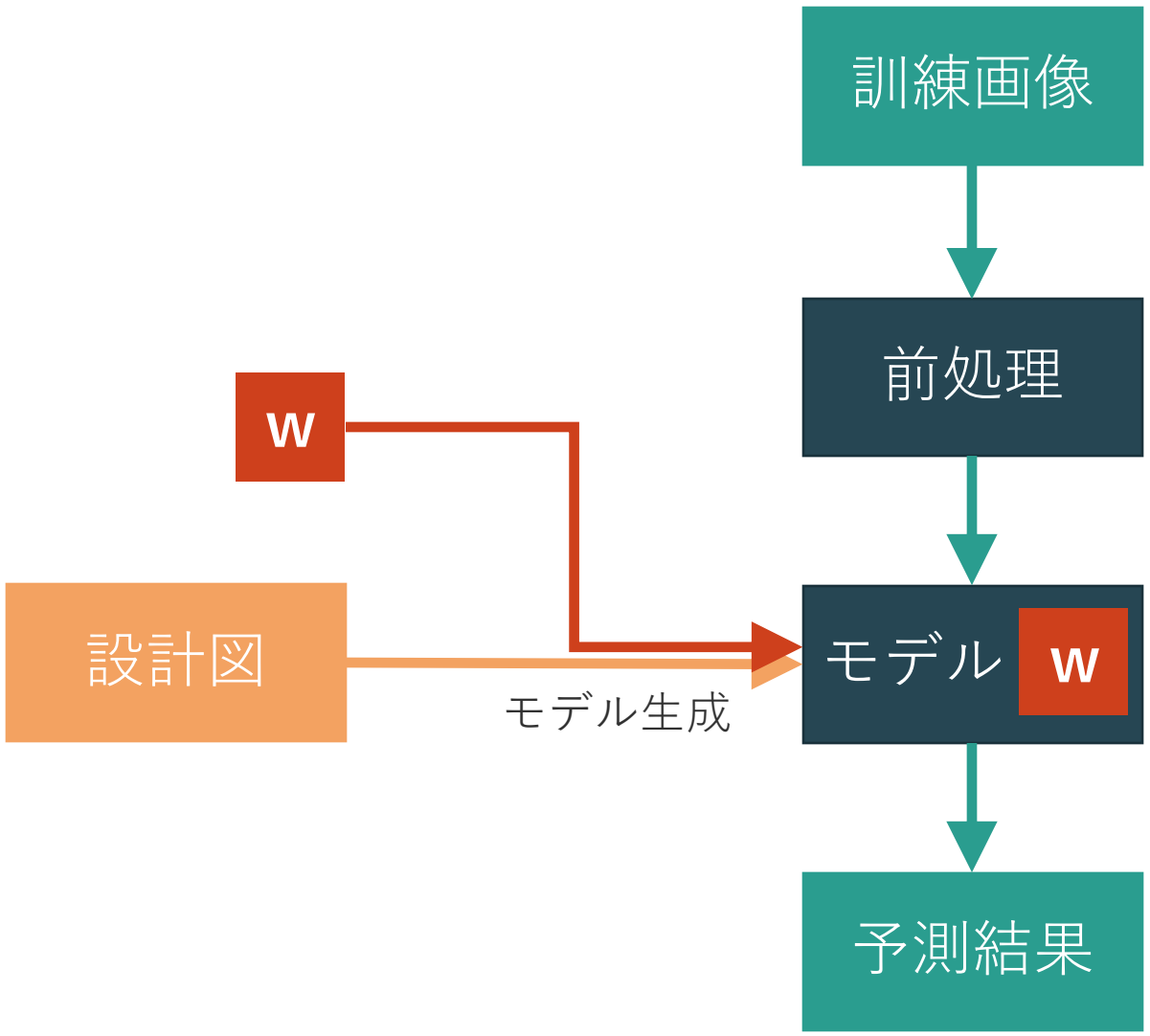
- 損失関数
  - 多クラス分類問題では一般的にクロスエントロピー関数を用いる。
  - 回帰分析では MSELoss などを使用する。
  - 独自で定義した関数を使用することもできる。その際、outputs と labels を受け取り、1 つの値を返す関数を定義すればよい。
- 最適化アルゴリズム
  - SGD や Adam などがよく使われる。
- エポック
  - 過学習を起こさない程度のエポックを設定。
  - 検証データを使用して early stopping を組み込む。

モデル訓練

```
# dataset
train_dataset = Dataset(images, labels)
train_dataloader = DataLoader(train_dataset, batch_size=4)

# model
model = LiteCNN()
model.train()

# parameters
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.001)
n_epochs = 50

# training
for epoch in range(n_epochs):
  running_loss = 0.0

  # mini batch
  for inputs, labels in train_dataloader:
    optimizer.zero_grad()           # 1
    outputs = model(inputs)         # 2
    loss = criterion(outputs, labels)  # 3
    loss.backward()                 # 4
    optimizer.step()                # 5
    running_loss += loss.item()
```

1. 古い損失（勾配情報）を消去する。
2. 画像データをモデルに代入し予測する。
3. 予測値と教師ラベルから損失（誤差）を計算する。
4. 損失をネットワーク全体に逆伝播する。
5. 逆伝播された損失を利用してパラメータを更新する。

推論




```
# model
model = LiteCNN()
model.load_state_dict(torch.load('weight.pth'))
model.eval()


# preprocess an image
img_fpath = '/path/to/image.jpg'
im = PIL.Image.open(img_fpath)
im = ImageOps.exif_transpose(im)
im = transform(im)
im = im.unsqueeze(0)


# inference
output = model(input)
# [[0.0935, -5.4933, -1.1156, 6.0684, 0.0409]]

output_softmax = torch.softmax(output, dim=1)
# [[0.0025, 0.0000, 0.0001, 0.9950, 0.0023]]

output_sigmoid = torch.softmax(output)
# [[0.5233, 0.0041, 0.2468, 0.9977, 0.5102]]
```

ResNet

```
from torch.nn as nn
from torchvision import models

model = models.resnet18(weights='DEFAULT')


print(model) モデル構造を出力し、出力層を確認



model.fc = nn.Linear(512, 4)
```

出力数を修正

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

デフォルトの出力数は 1000

ResNet

```
from torch.nn as nn
from torchvision import models

model = models.densenet121(weights='DEFAULT')


print(model) モデル構造を出力し、出力層を確認



model.classifier = nn.Linear(1024, 4)
```

出力数を修正

```
DenseNet(
  (features): Sequential(
    (conv0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
    (norm0): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu0): ReLU(inplace=True)
    (pool0): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
    (denseblock1): _DenseBlock(
      (denselayer1): _DenseLayer(
        (norm1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(64, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer2): _DenseLayer(
        (norm1): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(96, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer3): _DenseLayer(
        (norm1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(128, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer16): _DenseLayer(
        (norm1): BatchNorm2d(992, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(992, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
    )
    (norm5): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (classifier): Linear(in_features=1024, out_features=1000, bias=True)
)
```

デフォルトの出力数は 1000