Python と深層学習（PyTorch 編）

１，はじめに

1.1 深層学習による画像認識について

手書き文字とか動物の画像から、文字、動物名を、認識する分類器を作成するとします。分類器の行う仕事は以下となります。

そのままの画像では精度が出ないので、ピクセル単位で変換して、特徴を明確にします。次に、画像を数列に変換して、数式に入力して出力値から、名称を分類します。

この数式を求めるのがコンピュータの仕事のひとつで、学習データ（画像と名称がセットになっている）を使用して、画像と名称が一致する数式の定数(パラメータ)を求めます。

その後、検証用データを用いて、分類器の精度を検証し、実データで使用できるものか判断します。

専門用語を混ぜて言い換えると、以下のステップになります。

1,ニューラルネットワークの定義

2,学習データのネットワークへ入力

3,ロス（アウトプットと正解の差）の計算

4,勾配をネットワークへ反映

5,ネットワークのパラメータ（重み）の更新　　です。

（今回は、ステップ毎に説明できたらと考えています。）

1.2　今回の取組みについて

今回は、フレームワークとして PyTorch を使用します。

現在、有力なフレームワークとして、TensorFlor2.0　と PyTorch　があります。

処理速度は、同等で、シェアとして優勢なのは、研究者は PyTorch、ビジネスユースまで拡大すると、TensorFlor2.0 になるようです。（2022 年度で）

どちらを選択するべきか？の答えは、

『周囲で使用している、フレームワークを使ってみる』とのことです。

前回は TensorFlor2.0 だったので、今回は、PyTorch のチュートリアルにトライします。

（初心者にとっては、使い勝手の大きな差異は無かったです）

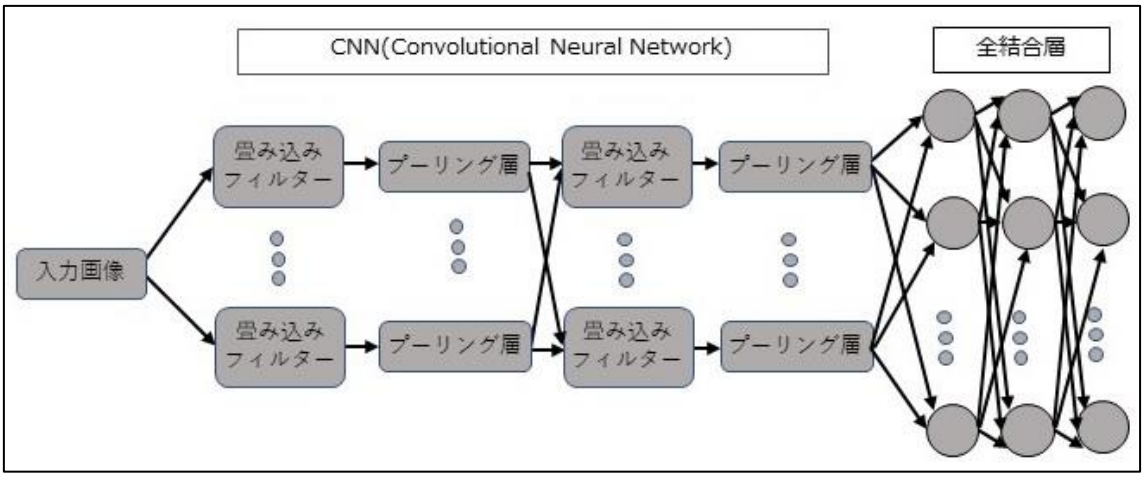
画像のデータは、このチュートリアルでは、CIFAR10 を使います。
CIFAR10 データセットには、飛行機、自動車、鳥、猫、鹿、犬、蛙、馬、船、トラックといった 10 のクラスがあります。画像はカラー(3-channel) で 32×32 pixels です。
学習用データが 5 万枚、検証用データが 1 万枚の計 6 万枚のデータセットです。

フリーの画像認識を目的とした初学者用の深層学習向けの画像データとして、CIFAR10（上記のもの CIFAR100 もあるとのこと)、MNIST（手書き数字の画像データ）があり、TensorFlor2.0 も PyTorch も、ライブラリとして、用意されてます。

実行環境は、Google が提供する　Colaboratory　で行いました。

1.3 ニューラルネットワークについて、

今回は以下のような構成となります。



畳み込み層とは、画像上に小領域（フィルタ）を設け、1 つの特徴量として圧縮（=畳み込み）する工程を画像上をスライドしながら繰り返すことによりできた層のことを指します。このフィルタを ReLU 等の活性化関数で繋ぐことで、CNN が、構築されます。
（フィルタの可視化については、3.1 項　17 ページ　にあります。）

プーリング層とは、画像上に小領域（フィルタ）を設け、その枠内のデータに演算を行うことによりできた層のことを指します。枠内に存在する数値のうち、最も高い数値を特徴量として演算する MAX プーリングや、平均値を採用する平均値プーリングがあります。

全結合層は、畳み込み演算やプーリング演算で抽出した特徴量を受け取って、特徴量で分類を行うために利用されます。

今回のチュートリアルでは、自らコーディング入力してニューラルネットワークを構成しますが、ResNet や AlexNet, VGGNet のような有名なニューラルネットワークもライブラリに含まれています。（これらは、2010 年から 2017 年に開催された ILSVRC(画像認識の精度を競う大会)で上位となったネットワークです）

簡単に説明すると

ResNet （2015 年　1 位)

152 層で構成される CNN になります。勾配消失問題や劣化問題によって学習が進まない問題を Residual block という手法を使って解決し、152 層という非常に深い層を実現した表現力の高いネットワークです。

AlexNet （2012 年　1 位)

構造がシンプルなため、他のネットワークのベースとして使用されることも多いモデルです。畳み込み層が 5 層で、そのうちのいくつかには MaxPooling 層があります。

また、出力層にはソフトマックス関数を持つ全結合層 3 層が使用されており、合計で 8 層により構成されています。

VGGNet （2014 年 2 位）　16 層 と　19 層があります。

基本的には畳み込みを 2~3 回繰り返してその後に MaxPooling を行う流れの繰り返しです。またその Pooling 処理の後の畳み込みでチャネル数を倍にしていき、Pooling のストライドは常に 2 を使用します。そして各畳み込みの後には活性化関数として ReLU を用います。

各畳み込み処理は 3×3 のカーネルサイズ、ストライド 1、パディングサイズ 1 のパラメータを持ちます。この畳み込みでは特徴マップのサイズが変わらないという点で利用しやすく、他の CNN でも広く利用されています。

実は、今回の CNN のベースは 1998 年に LeNet という名称で開発されました。LeNet は誤差逆伝播(Back Propagation)をよる勾配降下(Gradient Descent)を用いた CNN の学習を提案しています。目的は USPS (アメリカ合衆国郵便公社)向けの「郵便番号の手書き文字の識別問題」にあったようで、現在の畳み込みニューラルネットワークのほぼ完全な基本形といえます。

1.4 引用・参考　WEB

・今回のチュートリアルは、PyTorch の公式 WEB(https://pytorch.org/)　　の下にある DEEP LEARNING WITH PYTORCH: A 60 MINUTE BLITZ (https://pytorch.org/tutorials/beginner/blitz/)　にあります。

ここには、以下の 5 個のチュートリアルがあり、

本技術報告書では、4 番目の　Training a Classifier　で、検証し、解説しています。

1.tensor_tutorial.py　What is PyTorch?　（テンソルとは）

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

2.autograd_tutorial.py　Autograd: Automatic Differentiation（AUTOGRAD とは）

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

3.neural_networks_tutorial.py　Neural Networks　（ニューラルネットワークとは）

https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#

4.cifar10_tutorial.py　Training a Classifier　（画像分類とは）

https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

5.data_parallel_tutorial.py　Optional: Data Parallelism　（データ並列処理）

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

この 4 番目の Training a Classifier　について、WEB には、色々な日本語解説があるのですが、以下のものが適当でした。

・【PyTorch】チュートリアル(日本語版)④ ～TRAINING A CLASSIFIER（画像分類）～

https://www.nemotos.net/?p=4168

3 ページ目については、以下を参考にしています。

・画像分類の 6 つの代表的なアーキテクチャの特徴まとめ

https://ai-kenkyujo.com/artificial-intelligence/ai-architecture-02/

フィルターの可視化については以下です。（17 ページの内容）

・PyTorch で重みの確認と、畳み込み層のカーネルの可視化

https://betashort-lab.com/home/

2.プログラムとその説明

2.1　概略

チュートリアルでは、CIFAR10 の画像データと、下のステップで画像分類器を作ります。

・データの読み込みと標準化

・畳み込みニューラルネットワーク（CNN）の構築

・loss 関数と最適化

・CNN の学習

・学習済み CNN のテスト

2.2　データの読み込みと標準化

```
import torch
import torchvision
import torchvision.transforms as transforms
```

定番のライブラリのインポートです。

（Colaboratory　の環境で実行してるので、インストール済のライブラリです）

```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

Transform はデータに対して行う前処理を行うオブジェクトです。

画像を読み込む際にリサイズや標準化など一連の処理を行いたい場合に使用します。

transforms.Compose で読み込んだデータの前処理関数を連続して構成します。

transforms.ToTensor()　PIL Image オブジェクトをテンソルに変換し、

値の範囲を [0, 255] から [0, 1] にスケールします。Tensor は配列と理解して良いです。

PyTorch では、Tensor 型にしないと、GPU 計算、勾配計算に対応できないです。

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))　標準化します。

１つめの引数(タプル)が RGB それぞれの平均、２つめの引数(タプル)が標準偏差であり、

これらの平均と標準偏差をつかって標準化します。

標準化で、データに対して平均値が 0、標準偏差が 1 になるように計算します。

尺度を揃えるというのが目的です。例えば、体重と身長とのデータで、絶対値でデータ解析を始めるのは、無理がありますねという事で、機械学習等でよく行います。

```
batch_size = 4

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)
```

出力内容

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100% 170498071/170498071 [00:05<00:00, 33885553.49it/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

Train データと Test データをそれぞれ読み込んでいきます。

torchvision.datasets.CIFAR10　データの読み込みに使います。Dataset は、入力データとそれに対応するラベルを 1 組返すモジュールです。

download=True であれば root に画像を保存します。（この場合は、./data）

train=True、train=False とすることで Train データと Test データをに分けます。

テンソルに変換・標準化されたデータ(transform)を読み込みたいので、

transform=transform テンソルに変換・標準化されたデータ(transform)の読み込みです。

torch.utils.data.DataLoader は、データとラベルをバッチサイズで取得します。

batch_size 分のデータを生成します（ミニバッチと呼びます）。

shuffle　は、True あればデータの頭からの分割はしない、となります。

num_workers（並列実行数）はどれだけ並列処理するかを設定します。

（Colaboratory は 2）データを返すときは、データを tensor 型に変換して返します。

```
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

分類する 10 個のラベル名を、plane、car、bird、cat、deer、dog、frog、horse、ship、truck、をタプルとして classes に格納します。

```
import matplotlib.pyplot as plt
import numpy as np
```

定番のライブラリのインポートです。（画像の可視化に必要なライブラリ）

```
# functions to show an image
def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
```

画像にするため、Tensor 型のデータを、NumPy 配列にする関数を定義します。

img = img / 2 + 0.5　入力を plt.imshow の[0,1]にあわす。

npimg = img.numpy()　NumPy 配列に変換します。

plt.imshow(np.transpose(npimg, (1, 2, 0))) PyTorch の画像は、各次元が(チャネル数(RGB), 高さ, 横幅)と並んでいますが、plt.show は (高さ, 横幅, チャネル数)で並んでいる必要があるので np.transpose で並び替えます。

```
# get some random training images
dataiter = iter(trainloader)
images, labels = next(dataiter)
```

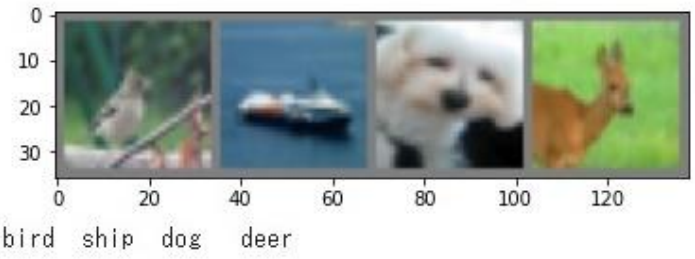
dataiter = iter(trainloader)　iter を使うことで各バッチごとの読出しができます。

images, labels = next(dataiter) : dataiter.next()を呼ぶことで次々とバッチごとに画像とそのラベルを返してくれます。もう一度 next()を呼び出すと 2 番目の要素を、さらにも

う一度呼び出すと 3 番目の要素を取り出すことができるといった具合に配列の要素を順番に取り出すことが可能です。取り出せなくなった時点で StopIteration 例外が発生します。

```
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join(f'{classes[labels[j]]:5s}' for j in range(batch_size)))
```

出力内容



imshow を使って学習データの画像 4 点を可視化します。

引数の torchvision.utils.make_grid は複数の画像を横並びにして 1 枚の画像にします。

print の引数ついて、f’ f 文字列置換を行い join()関数により、一つの文字列に連結します。

2.3 畳み込みニューラルネットワーク(CNN)の構築

```
import torch.nn as nn
import torch.nn.functional as F
```

定番のライブラリのインポートです。(ニューラルネットワーク作成のライブラリ)

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
device = torch.device("cuda:0")
net = Net()
net = Net().to(device)
```

・ニューラルネットワークの構成を定義します。

class Net(nn.Module): Net というクラスを、親クラスの nn.Module を継承する形で定義します。(クラスの継承は、PyTorch ではなく Python の手法です)

def __init__(self): 変数の初期化を行います(コントラクタ)。Python の手法です。

super().__init__() 親クラスを初期化します。Python の手法です。

self.conv1 = nn.Conv2d(3, 6, 5) 畳込みの定義で、 入力チャンネル数 3、

出力チャンネル数(=フィルターの数)6、カーネルサイズ 5 を示します。

self.pool = nn.MaxPool2d(2, 2)　pooling の定義で、カーネルサイズは 2x2 で、半分になります。

self.fc1 = nn.Linear(16 * 5 * 5, 120)　全結合層の定義で、入力は、32x32 のピクセルの画像を 5x5 カーネルで畳込みをして、プーリングで半分にするプロセスを 2 回行うので、画像は 5x5 のピクセルになります。16 は最後のフィルター数です。（フィルター数だけ画像が出来ます）

故に　入力サイズはベクトル数 400 です。出力サイズは、ベクトル数 120　としてます。400 個のニューロンから 120 個のニューロン出力するという事です。

self.fc2 = nn.Linear(120, 84) 全結合層の定義で、入力サイズ 120 出力サイズ 84 です。

self.fc3 = nn.Linear(84, 10)　最後は 10 個の出力で、確率の大の label に分類されます。

def forward(self, x):　層の定義をおこなった後、実行する関数です。

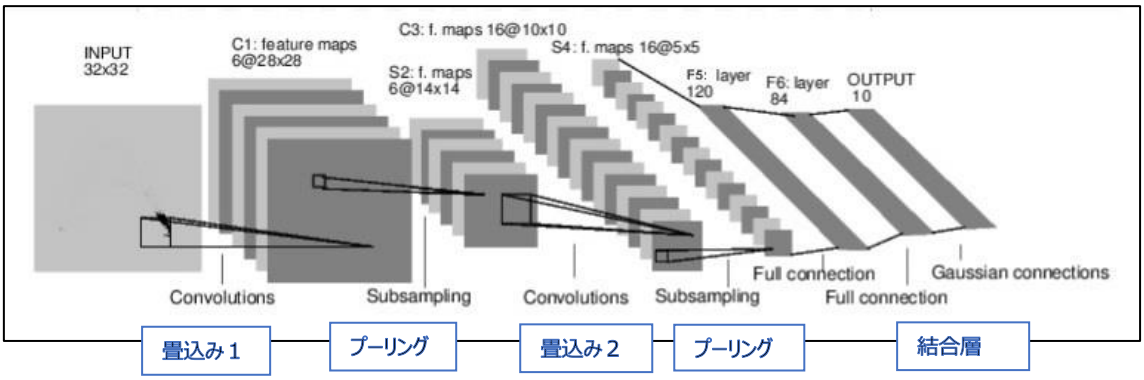
x = self.pool(F.relu(self.conv1(x)))　F.relu　で　relu 関数を呼んでます。

x = torch.flatten(x, 1) # flatten all dimensions except batch

特徴画像をベクトルにする。全結合層は 1 次元ベクトルしか入力することができません。しかし畳み込み層の出力はチャネル数×縦×横の三次元となります。そのままでは線形層に入力することができないため、畳み込み層の出力を一度バラバラに分解し一次元化し、結合層に入力します。このバラバラにする作業を torch.flatten()が担っています。

net = Net()　子クラスのインスタンス化（オブジェクト化）しています。

下図が今回のニューラルネットワーク構成図です。



2.4 loss 関数と最適化関数

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

ここでは、損失（分類が外れた確率）の算出方法と、損失を最小化していく手法の定義をしています。

criterion = nn.CrossEntropyLoss()　損失関数として交差エントロピー(Cross-Entropy)を使います。交差エントロピーは、正解の確率分布と推定の確率分の対数との積で表します。正解と推定の差の 2 乗で算出する 2 乗和誤差が一般ですが、SGD（確率的勾配降下法）には、交差エントロピーが計算時の相性が良いようです。

optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

最適化関数として、確率的勾配降下法（SGD）を使います。

net.parameters()は先程定義した CNN のパラメータ(重み)で、これが optim.SGD で更新され、最小の損失になるパラメータ(重み)を探します。方法として、損失対パラメータの勾配が一番小さい点を見つける事になります。

lr=0.001 は学習率が 0.001 であることを示しています。学習率は、学習速度であり、現在のパラメータ値から次のパラメータ値に移動する量を規定します。学習率が 1 であれば、現在の勾配と同じ量で次に移動させます。

momentum（慣性項）では、SGD の momentum を設定することができます。デフォルトは momentum=0 です。

2.5 CNN の学習　(構築した CNN の学習をします)

```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)
        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.3f}')
            running_loss = 0.0

print('Finished Training')
```

出力内容

```
[1,  2000] loss: 2.208
[1,  4000] loss: 1.872
[1,  6000] loss: 1.686
[1,  8000] loss: 1.604
[1, 10000] loss: 1.539
[1, 12000] loss: 1.472
[2,  2000] loss: 1.421
[2,  4000] loss: 1.373
[2,  6000] loss: 1.334
[2,  8000] loss: 1.311
[2, 10000] loss: 1.297
[2, 12000] loss: 1.298
Finished Training
```

CNN の学習は、ネットワーク内のパラメータを更新し、正しい答えが導き出せるようにパラメータを最適化します。‘epoch‘ は、全データ分 1 回分の単位です。

for epoch in range(2): 簡単のため、学習回数 epoch を 2 回にします。

for i, data in enumerate(trainloader, 0): enumerate で インデックス番号, 要素の順に取得します。

inputs, labels = data 学習データと教師ラベルデータを取得します。

optimizer.zero_grad() optimizer は計算した勾配(grad)を記録し蓄積し続けるので、一度 optimizer.zero_grad()で勾配初期化します。

outputs = net(inputs) Train データを CNN に入力します。(2.3 項で作成した class Net() の def forward を実行します。)

loss = criterion(outputs, labels) CNN からの出力 outputs と実際の答え labels との間の交差エントロピーを計算します。返り値は tensor 型になります。

loss.backward() 勾配情報の計算を行います。(loss から input までの勾配 パラメータの微分係数を計算)

optimizer.step() loss.backward()で計算した勾配をもとに、パラメータを更新します。

running_loss += loss.item() ロスを加算します。

if i % 2000 == 1999: %は剰余演算子です。(割り算の余りを返します)

2000 ミニバッチずつ進捗を表示を指定してます。

```
PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)
```

学習したモデル(CNN)を cifar_net.pth という名前で保存します。pth は PyTorch のモデルの拡張子です。モデルのパラメータを取り出すには、net.state_dict()を使います。

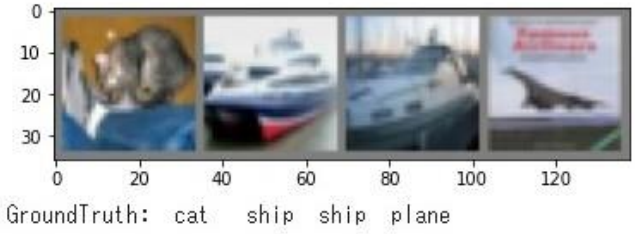
モデルの保存には、torch.save でできます。

2.6 学習済み CNN でテストデータをテストする。

```
dataiter = iter(testloader)
images, labels = next(dataiter)

# print images
imshow(torchvision.utils.make_grid(images))
print('GroundTruth: ', ' '.join(f'{classes[labels[j]]:5s}' for j in range(4)))
```

出力内容




```
GroundTruth:  cat   ship  ship  plane
```

CNN の性能を Test データを使って評価したいと思います。

まず、Test データを表示し画像とラベル（正解名）を確認します。

```
net = Net()
net.load_state_dict(torch.load(PATH))
```

出力内容

```
<All keys matched successfully>
```

次に、練習のために保存した学習済みのモデル(cifar_net.pth)を読み込みます。

モデルの読み込みには、net.load_state_dict を使います。

```
outputs = net(images)
```

読み込んだモデルに画像データを入力します。

```
_, predicted = torch.max(outputs, 1)
print('Predicted: ', ' '.join(f'{classes[predicted[j]]:5s}'
                               for j in range(4)))
```

出力内容

```
Predicted:  cat   car   car   ship
```

_, predicted = torch.max(outputs, 1)　引数の 1 は行方向の最大値を求めるの意味です。

予測は　cat　しか　正解できませんでした。正解率 25%です。

次に　10000 個の Test データについて、正解率を算出します。

```
correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for our outputs
with torch.no_grad():
    for data in testloader:
        images, labels = data
        # calculate outputs by running images through the network
        outputs = net(images)
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the network on the 10000 test images: {100 * correct // total} %')
```

出力内容

```
Accuracy of the network on the 10000 test images: 56 %
```

with torch.no_grad():　まずは、勾配を初期化していま

for data in testloader:　testloader に格納されているバッチデータを 1 づつ取り出して

data に格納します。

images, labels = data　検証データと教師ラベルデータを取得

outputs = net(images)　検証データをモデルに渡し予測

_, predicted = torch.max(outputs.data, 1)　予測結果を取得

total += labels.size(0)　トータル数をカウントします。

correct += (predicted == labels).sum().item()　正解数をカウントします。

正解率は　56%　で、半分しか正解できませんでした。

選択肢が 10 個あり、1 択で当てる訳ですので、偶然での確率は 10%です。

ネットワークの学習はしてるようです。

次に、クラス内の種類の正解率を算出します。

```
# prepare to count predictions for each class
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}

# again no gradients needed
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predictions = torch.max(outputs, 1)
        # collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1


# print accuracy for each class
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')
```

出力内容

```
Accuracy for class: plane is 70.5 %
Accuracy for class: car   is 67.0 %
Accuracy for class: bird  is 45.6 %
Accuracy for class: cat   is 35.6 %
Accuracy for class: deer  is 56.0 %
Accuracy for class: dog   is 31.2 %
Accuracy for class: frog  is 68.0 %
Accuracy for class: horse is 66.1 %
Accuracy for class: ship  is 68.4 %
Accuracy for class: truck is 58.8 %
```

正解率は、cat 、dog が 30%台で低く、　高いのは、plane 70% でした。

3 おまけ

3.1 学習後のフィルター（カーネル）を可視化してみます。

```
net.state_dict()
```

出力内容

```
OrderedDict([('conv1.weight',
              tensor([[[[-0.1065,  0.0697,  0.1512,  0.0919,  0.2468],
                        [-0.0552,  0.1669, -0.0037, -0.1331, -0.0485],
                        [ 0.1569,  0.2482,  0.0876, -0.3201, -0.1180],
                        [ 0.1493,  0.2219,  0.0747, -0.3736,  0.0080],
                        [ 0.0414,  0.0290,  0.0643, -0.2271, -0.0740]],

                       [[ 0.0426, -0.0014, -0.1731, -0.1591, -0.0739],
                        [ 0.1108,  0.2754, -0.0970, -0.3663, -0.3075],
                        [ 0.1039,  0.3307,  0.1131, -0.3378, -0.3257],
                        [ 0.0635,  0.1674, -0.0157, -0.3296, -0.2199],
                        [ 0.1804,  0.1442, -0.0095, -0.2207, -0.0204]],

                       [[ 0.0104,  0.0470, -0.0602, -0.0995, -0.2043],
                        [ 0.1799,  0.2892,  0.1068, -0.4211, -0.2511],
                        [ 0.2970,  0.3259,  0.0917, -0.4644, -0.2222],
                        [ 0.2646,  0.3899,  0.1291, -0.2492, -0.2547],
                        [ 0.2541,  0.3418,  0.0166, -0.0317,  0.0010]]],


                      [[[-0.1766, -0.0893, -0.1765, -0.1863, -0.0898],
                        [-0.2531, -0.3045, -0.1831, -0.2824, -0.1555],
                        [-0.2971, -0.3340, -0.2118, -0.1706, -0.0304],
                        [-0.0963, -0.0473, -0.1059, -0.0882, -0.1200],
                        [-0.1232, -0.0604, -0.2059,  0.0424, -0.0007]],

                       [[-0.0759, -0.0799,  0.0414,  0.0765,  0.0059],
                        [-0.0170,  0.1388,  0.0607,  0.0566, -0.0386],
                        [ 0.0495,  0.0217,  0.0470,  0.1233,  0.0422],
                        [-0.1007,  0.0375,  0.0713,  0.0528,  0.0559],
                        [-0.1848, -0.1587,  0.0112, -0.1608, -0.0457]],

                       [[ 0.2860,  0.2894,  0.2834,  0.2094,  0.1471],
                        [ 0.2276,  0.3870,  0.2272,  0.3179,  0.1902],
                        [ 0.2884,  0.4428,  0.2602,  0.1306,  0.0193],
                        [ 0.1184,  0.2489,  0.3064,  0.1494,  0.1842],
                        [ 0.0976,  0.2160,  0.2392,  0.1493, -0.0872]]],
```

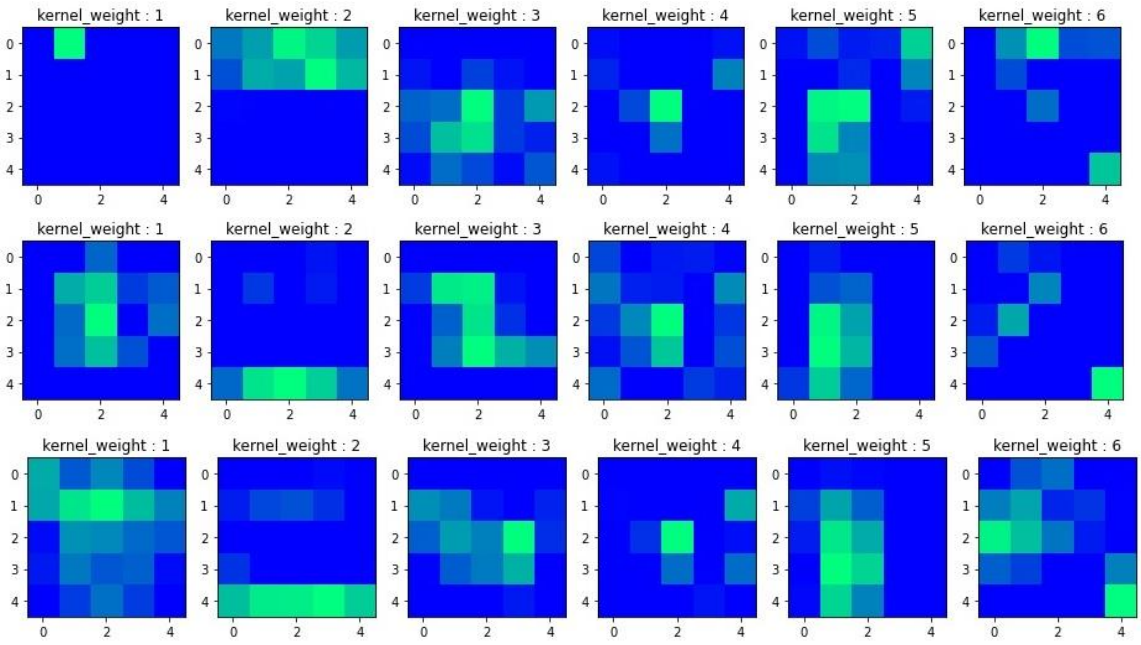
net.state_dict()　で、層の厚み、パラメータを確認します。

出力内容は、全部ではなく一部のみです。赤枠が、畳込み層１で６個出力した中の１個に、

当たるフィルターで、３チャンネルですので、５x ５のフィルタが３個あります。

```
plt.figure(figsize=(16,16))
for i in range(6):
    kernel_weight=np.array(net.state_dict()['conv1.weight'])[i,0].reshape(5,5 )
    kernel_weight = np.clip(kernel_weight, 0, 1)
    plt.subplot(3, 6, i+1)
    plt.title(f"kernel_weight : {i+1}")
    plt.imshow(kernel_weight, cmap='winter')
```

出力内容



畳込み 1 の層で、フィルター（カーネル 5x5）が　6 個あり、

カラーですので、3 チャンネルになり、計　18 個のフィルターが表示されます。

これらのフィルタが適切かは、よく分かりませんが、学習データの違い、epoch の回数により、変化します。上は、epoch=2 の例ですので、次項で、epoch=10 と比較します。

3.2　正解率をあげる取組　（epoch＝2　と　epoch=10 との比較）

・データを 10 回、更新すると　loss は　1.298　から　0.837　になります。

epoch＝2

```
[1,  2000] loss: 2.208
[1,  4000] loss: 1.872
[1,  6000] loss: 1.686
[1,  8000] loss: 1.604
[1, 10000] loss: 1.539
[1, 12000] loss: 1.472
[2,  2000] loss: 1.421
[2,  4000] loss: 1.373
[2,  6000] loss: 1.334
[2,  8000] loss: 1.311
[2, 10000] loss: 1.297
[2, 12000] loss: 1.298
Finished Training
```

epoch＝10

```
[9,  2000] loss: 0.815
[9,  4000] loss: 0.826
[9,  6000] loss: 0.838
[9,  8000] loss: 0.854
[9, 10000] loss: 0.860
[9, 12000] loss: 0.890
[10,  2000] loss: 0.770
[10,  4000] loss: 0.786
[10,  6000] loss: 0.819
[10,  8000] loss: 0.822
[10, 10000] loss: 0.840
[10, 12000] loss: 0.837
Finished Training
```

・全テストデータの正解率は　56%　から　62%　になります

epoch＝2

```
Accuracy of the network on the 10000 test images: 56 %
```

epoch＝10

```
Accuracy of the network on the 10000 test images: 62 %
```

・クラス別の正解率も、概ね、向上します。

epoch＝2

```
Accuracy for class: plane is 70.5 %
Accuracy for class: car   is 67.0 %
Accuracy for class: bird  is 45.6 %
Accuracy for class: cat   is 35.6 %
Accuracy for class: deer  is 56.0 %
Accuracy for class: dog   is 31.2 %
Accuracy for class: frog  is 68.0 %
Accuracy for class: horse is 66.1 %
Accuracy for class: ship  is 68.4 %
Accuracy for class: truck is 58.8 %
```

epoch＝10

```
Accuracy for class: plane is 63.5 %
Accuracy for class: car   is 79.9 %
Accuracy for class: bird  is 47.5 %
Accuracy for class: cat   is 36.2 %
Accuracy for class: deer  is 60.8 %
Accuracy for class: dog   is 50.7 %
Accuracy for class: frog  is 74.7 %
Accuracy for class: horse is 68.3 %
Accuracy for class: ship  is 71.4 %
Accuracy for class: truck is 73.3 %
```

epoch を増やしても、どこかで、正解率は飽和するようです。

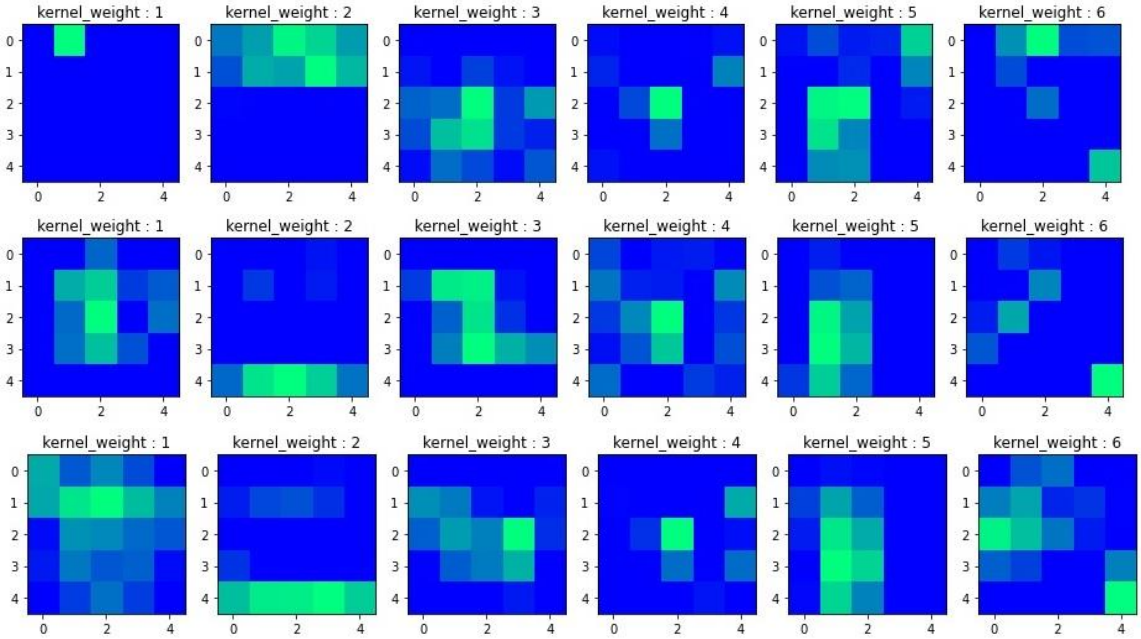
次の正解率アップの手段は、モデル、最適化関数、等の見直しになります。

ここからが、深層学習の入り口となるようです。（本報告書は、ここまでですが）

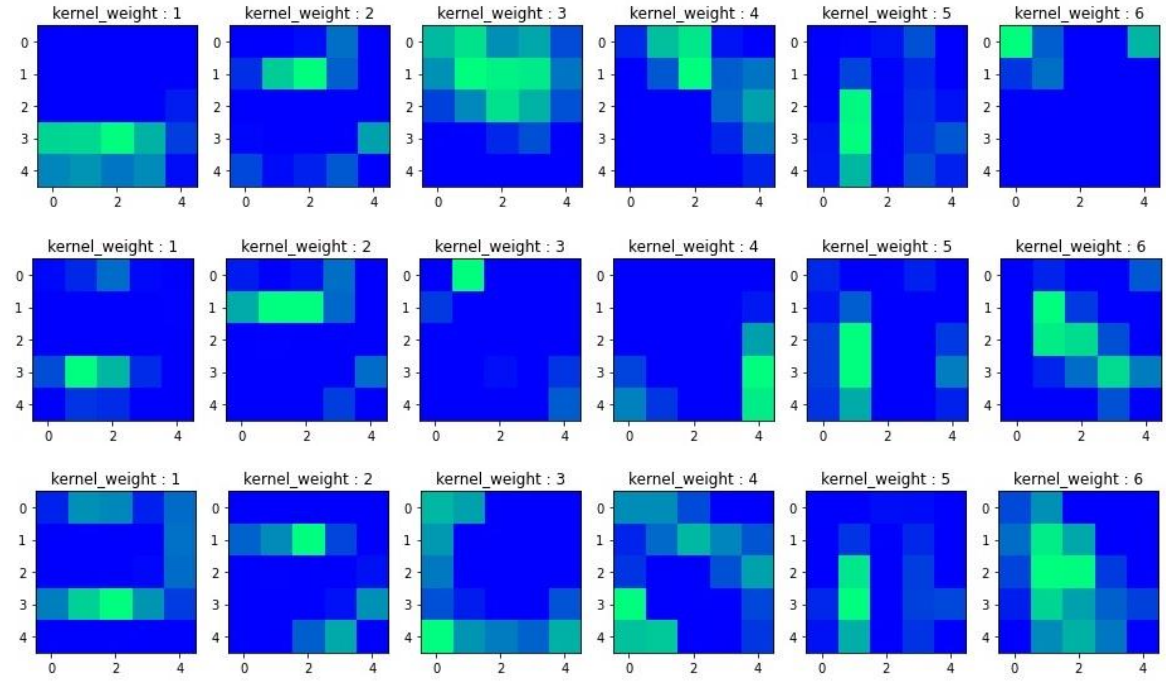
epoch=2 と　epoch=10 とのフィルター（カーネル）の変化です。

学習した成果があるのでしょうか？

epoch=2



epoch = 10



3.3　忘備録も兼ねて、分かりやすいと思った WEB を紹介します。

・Google　Colaboratory 関連は沢山 WEB 情報がありますが、丁寧な説明のものです。

Google　Colaboratory の使い方

https://www.nemotos.net/?p=4099

Google　Colaboratory は、所定の時間内にアクセスしないとシャットダウンされます。コードが消えるわけではありませんが、再開始する時は、最初から実行する必要があります。（学習用の環境と考えた方が良いようです）

・PyTorch　に分かりやすく、丁寧にまとめてあります。

PyTorch で CIFAR-10 を CNN に学習させる【PyTorch 基礎】前編（後編にもアクセス）

https://rightcode.co.jp/blog/information-technology/pytorch-cifar-10-cnn-learning

・PyTorch で　手書き数字（MNIST）分類を行う解説書です。

PyTorch で CNN を徹底解説

https://qiita.com/mathlive/items/8e1f9a8467fff8dfd03c

・以下は、フィルタを可視化するコードで、私用の忘備録です。

```
for i in range(6):
    kernel_weight = np.array(net.state_dict()['conv1.weight'])[i].reshape(5,5,3)
    kernel_weight = np.clip(kernel_weight, 0, 1)
    plt.subplot(3, 2, i+1)
    plt.title(f"kernel_weight : {i+1}")
    plt.imshow(kernel_weight)
```