70.単語ベクトルの和による特徴量

解答：

```
import pandas as pd
from sklearn.model_selection import train_test_split

# データの読込
df = pd.read_csv('./newsCorpora_re.csv', header=None, sep='\t', n
ames=['ID', 'TITLE', 'URL', 'PUBLISHER', 'CATEGORY', 'STORY', 'HO
STNAME', 'TIMESTAMP'])

# データの抽出
df = df.loc[df['PUBLISHER'].isin(['Reuters', 'Huffington Post', '
Businessweek', 'Contactmusic.com', 'Daily Mail']), ['TITLE', 'CAT
EGORY']]

# データの分割
train, valid_test = train_test_split(df, test_size=0.2, shuffle=T
rue, random_state=123, stratify=df['CATEGORY'])
valid, test = train_test_split(valid_test, test_size=0.5, shuffle
=True, random_state=123, stratify=valid_test['CATEGORY'])

# 事例数の確認
print('【学習データ】')
print(train['CATEGORY'].value_counts())
print('【検証データ】')
print(valid['CATEGORY'].value_counts())
print('【評価データ】')
print(test['CATEGORY'].value_counts())

from gensim.models import KeyedVectors
# ダウンロードファイルのロード
model = KeyedVectors.load_word2vec_format('GoogleNews-vectors-
negative300.bin.gz', binary=True)

import string
import torch

def transform_w2v(text):
  table = str.maketrans(string.punctuation, ' '*len(string.punctu
ation))
  words = text.translate(table).split()  # 記号をスペースに置換後、ス
ペースで分割してリスト化
```

```
  vec = [model[word] for word in words if word in model]  # 1語ずつベクトル化

  return torch.tensor(sum(vec) / len(vec))  # 平均ベクトルを Tensor 型に変換して出力


# 特徴ベクトルの作成
X_train = torch.stack([transform_w2v(text) for text in train['TITLE']])
X_valid = torch.stack([transform_w2v(text) for text in valid['TITLE']])
X_test = torch.stack([transform_w2v(text) for text in test['TITLE']])

print(X_train.size())
print(X_train)
category_dict = {'b': 0, 't': 1, 'e':2, 'm':3}
y_train = torch.tensor(train['CATEGORY'].map(lambda x: category_dict[x]).values)
y_valid = torch.tensor(valid['CATEGORY'].map(lambda x: category_dict[x]).values)
y_test = torch.tensor(test['CATEGORY'].map(lambda x: category_dict[x]).values)

print(y_train.size())
print(y_train)
# 保存
torch.save(X_train, 'X_train.pt')
torch.save(X_valid, 'X_valid.pt')
torch.save(X_test, 'X_test.pt')
torch.save(y_train, 'y_train.pt')
torch.save(y_valid, 'y_valid.pt')
torch.save(y_test, 'y_test.pt')
```

実行結果：

```
torch.Size([10684, 300])
tensor([[ 0.0837,  0.0056,  0.0068,  ...,  0.0751,  0.0433, -0.0868],
        [ 0.0272,  0.0266, -0.0947,  ..., -0.1046, -0.0489, -0.0092],
        [ 0.0577, -0.0159, -0.0780,  ..., -0.0421,  0.1229,  0.0876],
        ...,
        [ 0.0392, -0.0052,  0.0686,  ..., -0.0175,  0.0061, -0.0224],
        [ 0.0798,  0.1017,  0.1066,  ..., -0.0752,  0.0623,  0.1138],
        [ 0.1664,  0.0451,  0.0508,  ..., -0.0531, -0.0183, -0.0039]])
```

```
torch.Size([10684])
tensor([0,  1,  3,  ...,  0,  3,  2])
```

まとめ：

str.maketrans()メソッドは文字マッピングの変換テーブルを作成するために使用され、2つのパラメータの最も簡単な呼び出しを受け入れる方法について、最初のパラメータは文字列で、変換が必要な文字を表し、2番目のパラメータも文字列は変換のターゲットを表します。

torch.tensor()関数のプロトタイプは次のとおりです,
torch.tensor(data, dtype=None, device=None, requires_grad=False)，その中で data は：list、tuple、array、scalar などのタイプであることができます。
torch.tensor()は、直接参照ではなく data 内のデータ部分からコピーし、元のデータ型から対応する torch.LongTensor、torch.FloatTensor、torch.DoubleTensor を生成することができます。

torch.stack(sequence, dim=0)入力テンソルシーケンスを新しい次元に沿って接続し、シーケンス内のすべてのテンソルは同じ形状でなければならない。stack 関数が返した結果、新しい次元が追加されます。stack()関数が指定した dim パラメータは、新しい次元の（下付き）位置です。

7 1.単層ニューラルネットワークによる予測

解答：

```
from torch import nn

class SLPNet(nn.Module):
  def __init__(self, input_size, output_size):
    super().__init__()
    self.fc = nn.Linear(input_size, output_size, bias=False)
    nn.init.normal_(self.fc.weight, 0.0, 1.0)  # 正規乱数で重みを初期化

  def forward(self, x):
    x = self.fc(x)
    return x
model = SLPNet(300, 4)  # 単層ニューラルネットワークの初期化
y_hat_1 = torch.softmax(model(X_train[:1]), dim=-1)
print(y_hat_1)
Y_hat = torch.softmax(model.forward(X_train[:4]), dim=-1)
print(Y_hat)
```

実行結果：

```
tensor([[0.5374, 0.1117, 0.2608, 0.0902]], grad_fn=<SoftmaxBackward0>)
```

```
tensor([[0.5374, 0.1117, 0.2608, 0.0902],
        [0.6755, 0.1323, 0.1488, 0.0434],
        [0.2768, 0.0644, 0.4152, 0.2436],
        [0.2537, 0.1070, 0.5074, 0.1319]], grad_fn=<SoftmaxBackward0>)
```

まとめ：

nn.Linear( )はニューラルネットワークの線形層を定義し、方法は以下の通り：

torch.nn.Linear(in_features, # 入力されたニューロン数

out_features, # 出力されたニューロン数 bias=True)

$$Y_{n\times o} = X_{n\times i}W_{i\times o} + b$$

7 2.損失と勾配の計算

解答：

```
criterion = nn.CrossEntropyLoss()

l_1 = criterion(model(X_train[:1]), y_train[:1])  # 入力ベクトルは softmax 前の値
model.zero_grad()  # 勾配をゼロで初期化
l_1.backward()  # 勾配を計算
print(f'損失: {l_1:.4f}')
print(f'勾配:\n{model.fc.weight.grad}')

l = criterion(model(X_train[:4]), y_train[:4])
model.zero_grad()
l.backward()
print(f'損失: {l:.4f}')
print(f'勾配:\n{model.fc.weight.grad}')
```

実行結果 ：

```
損失: 0.6210
勾配:
tensor([[-0.0387, -0.0026, -0.0032,  ..., -0.0348, -0.0200,  0.0402],
        [ 0.0093,  0.0006,  0.0008,  ...,  0.0084,  0.0048, -0.0097],
        [ 0.0218,  0.0015,  0.0018,  ...,  0.0196,  0.0113, -0.0226],
        [ 0.0075,  0.0005,  0.0006,  ...,  0.0068,  0.0039, -0.0078]])
```

```
損失: 1.1835
勾配:
tensor([[-0.0046,  0.0059, -0.0182,  ..., -0.0301, -0.0023,  0.0132],
        [-0.0041, -0.0045,  0.0211,  ...,  0.0238,  0.0148,  0.0004],
        [ 0.0193, -0.0064, -0.0188,  ..., -0.0017,  0.0090,  0.0056],
        [-0.0106,  0.0051,  0.0159,  ...,  0.0081, -0.0215, -0.0193]])
```

まとめ：

nn.CrossEntropyLoss()の計算式は次のとおりです。

$$loss(x, class) = -log(\frac{exp(x[class]))}{\sum_i exp(x[i]}) = -x[class]+log(\sum_i exp(x[i]))$$

nn.CrossEntropyLoss（x，lable)、2つのパラメータ
最初のパラメータ:xは入力でありネットワークの最後のレイヤの出力であり、そのshapeは［batchsize，class］である（関数は最初のパラメータ、つまり最後のレイヤの出力が2次元データであり、各ベクトル中の値が異なる種類の確率値であることを要求する）
2つ目のパラメータ：入力されたラベル、つまり計算に参加していないカテゴリのインデックス値です。batch_sizeが1であれば、batch _sizeが2である場合、2つのサンプルに対応する実際のカテゴリをそれぞれ表す（0、1）などの2つの数字があります。

73.確率的勾配降下法による学習

解答：

```
from torch.utils.data import Dataset

class NewsDataset(Dataset):
  def __init__(self, X, y):  # datasetの構成要素を指定
    self.X = X
    self.y = y

  def __len__(self):  # len(dataset)で返す値を指定
    return len(self.y)

  def __getitem__(self, idx):  # dataset[idx]で返す値を指定
    return [self.X[idx], self.y[idx]]

from torch.utils.data import DataLoader

# Datasetの作成
dataset_train = NewsDataset(X_train, y_train)
dataset_valid = NewsDataset(X_valid, y_valid)
dataset_test = NewsDataset(X_test, y_test)
```

```
# Dataloaderの作成
dataloader_train = DataLoader(dataset_train, batch_size=1, shuffl
e=True)
dataloader_valid = DataLoader(dataset_valid, batch_size=len(datas
et_valid), shuffle=False)
dataloader_test = DataLoader(dataset_test, batch_size=len(dataset
_test), shuffle=False)

# モデルの定義
model = SLPNet(300, 4)

# 損失関数の定義
criterion = nn.CrossEntropyLoss()

# オプティマイザの定義
optimizer = torch.optim.SGD(model.parameters(), lr=1e-1)

# 学習
num_epochs = 10
for epoch in range(num_epochs):
  # 訓練モードに設定
  model.train()
  loss_train = 0.0
  for i, (inputs, labels) in enumerate(dataloader_train):
    # 勾配をゼロで初期化
    optimizer.zero_grad()

    # 順伝播 + 誤差逆伝播 + 重み更新
    outputs = model(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()

    # 損失を記録
    loss_train += loss.item()

  # バッチ単位の平均損失計算
  loss_train = loss_train / i

  # 検証データの損失計算
  model.eval()
  with torch.no_grad():
    inputs, labels = next(iter(dataloader_valid))
    outputs = model(inputs)
```

```
    loss_valid = criterion(outputs, labels)

  # ログを出力
  print(f'epoch: {epoch + 1}, loss_train: {loss_train:.4f}, loss_
valid: {loss_valid:.4f}')
```

実行結果 ：

```
epoch: 1, loss_train: 0.4659, loss_valid: 0.3546
epoch: 2, loss_train: 0.3105, loss_valid: 0.3219
epoch: 3, loss_train: 0.2811, loss_valid: 0.3122
epoch: 4, loss_train: 0.2647, loss_valid: 0.3070
epoch: 5, loss_train: 0.2552, loss_valid: 0.3081
epoch: 6, loss_train: 0.2480, loss_valid: 0.3048
epoch: 7, loss_train: 0.2424, loss_valid: 0.3041
epoch: 8, loss_train: 0.2388, loss_valid: 0.3032
epoch: 9, loss_train: 0.2348, loss_valid: 0.3053
epoch: 10, loss_train: 0.2316, loss_valid: 0.3088
```

まとめ：

ＳＧＤ方法

ランダム勾配降下（SGD）は単純だが非常に有効な方法であり、ベクトルマシン、論理回帰（LR）などの凸損失関数下の線形分類器の学習を支援するために多用される。また、SGDはテキスト分類と自然言語処理でよく遭遇する大規模な疎機械学習問題に成功した。

SGD は分類計算にも回帰計算にも使用できる。

SGD アルゴリズムはサンプルからランダムに 1 組を抽出し、訓練後に勾配によって 1 回更新し、それから 1 組を抽出し、再び更新し、サンプル量とその大きさの場合、すべてのサンプルを訓練しなくても許容範囲内の損失値のモデルを得ることができるかもしれない。

（重点：反復ごとにサンプルのセットを使用する。）

Loop {

for i=1 to m, {

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)} \qquad \text{(for every } j\text{)}.$$

}

}

利点：

（1）すべての訓練データ上の損失関数ではなく、各ラウンドの反復の中で、ランダムにある訓練データ上の損失関数を最適化するため、各ラウンドのパラメータの更新速度が大幅に加速する。

短所：

（1）精度が低下する。ターゲット関数が強い凸関数の場合でも、SGD は線形収束を行うことができないためです。

（2）単一サンプルが全体サンプルの傾向を表すわけではないため、局所最適に収束する可能性がある。
（3）並列実装が容易ではない。

74.正確率の計測

解答：

```
def calculate_accuracy(model, loader):
  model.eval()
  total = 0
  correct = 0
  with torch.no_grad():
    for inputs, labels in loader:
      outputs = model(inputs)
      pred = torch.argmax(outputs, dim=-1)
      total += len(inputs)
      correct += (pred == labels).sum().item()

  return correct / total

acc_train = calculate_accuracy(model, dataloader_train)
acc_test = calculate_accuracy(model, dataloader_test)
print(f'正解率（学習データ）：{acc_train:.3f}')
print(f'正解率（評価データ）：{acc_test:.3f}')
```

実行結果：

```
正解率（学習データ）：0.924
正解率（評価データ）：0.903
```

まとめ：

argmax 関数：torch.argmax（input，dim＝None，keepdim＝False）は指定された次元の最大値のシーケンス番号を返し、dim は：the demention to reduce.つまり dim という次元を、この次元の最大値にする index を定義する。

75.損失と正解率のプロット

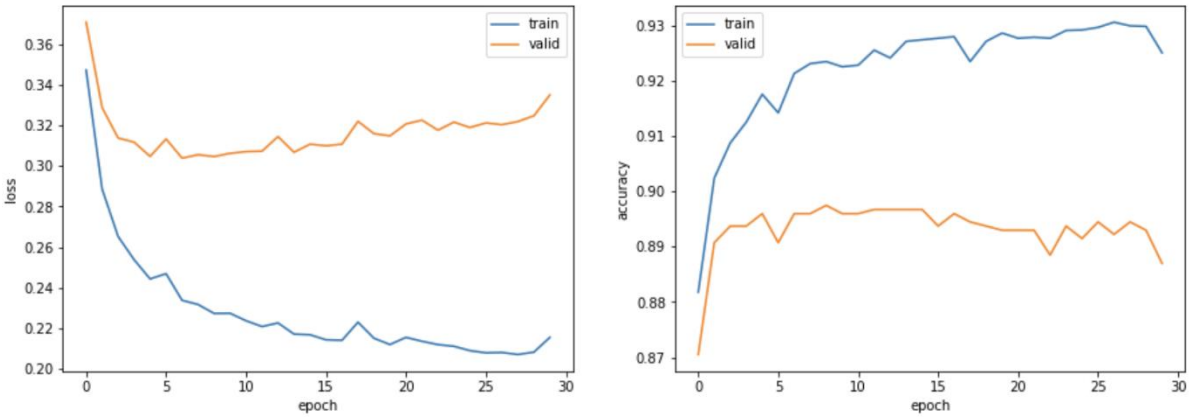
解答：

```
def calculate_loss_and_accuracy(model, criterion, loader):
  model.eval()
  loss = 0.0
```

```
    total = 0
    correct = 0
    with torch.no_grad():
      for inputs, labels in loader:
        outputs = model(inputs)
        loss += criterion(outputs, labels).item()
        pred = torch.argmax(outputs, dim=-1)
        total += len(inputs)
        correct += (pred == labels).sum().item()

    return loss / len(loader), correct / total

# モデルの定義
model = SLPNet(300, 4)

# 損失関数の定義
criterion = nn.CrossEntropyLoss()

# オプティマイザの定義
optimizer = torch.optim.SGD(model.parameters(), lr=1e-1)

# 学習
num_epochs = 30
log_train = []
log_valid = []
for epoch in range(num_epochs):
  # 訓練モードに設定
  model.train()
  for inputs, labels in dataloader_train:
    # 勾配をゼロで初期化
    optimizer.zero_grad()

    # 順伝播 + 誤差逆伝播 + 重み更新
    outputs = model(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()

  # 損失と正解率の算出
  loss_train, acc_train = calculate_loss_and_accuracy(model, crit
erion, dataloader_train)
  loss_valid, acc_valid = calculate_loss_and_accuracy(model, crit
erion, dataloader_valid)
  log_train.append([loss_train, acc_train])
```

```
  log_valid.append([loss_valid, acc_valid])

from matplotlib import pyplot as plt
import numpy as np

# 視覚化
fig, ax = plt.subplots(1, 2, figsize=(15, 5))
ax[0].plot(np.array(log_train).T[0], label='train')
ax[0].plot(np.array(log_valid).T[0], label='valid')
ax[0].set_xlabel('epoch')
ax[0].set_ylabel('loss')
ax[0].legend()
ax[1].plot(np.array(log_train).T[1], label='train')
ax[1].plot(np.array(log_valid).T[1], label='valid')
ax[1].set_xlabel('epoch')
ax[1].set_ylabel('accuracy')
ax[1].legend()
plt.show()
```

実行結果：



まとめ：

torch.optim.SGD はオプティマイザクラスを返します。
sgd=torch.optim.SGD(paramater, lr=0.5)；lr は学習率、paramater はパラメータを表す。
sgd.zero_grad()各点の勾配をクリア。
sgd.step() 最適化の実行。

76. チェックポイント

解答：

```
# モデルの定義
model = SLPNet(300, 4)

# 損失関数の定義
criterion = nn.CrossEntropyLoss()

# オプティマイザの定義
optimizer = torch.optim.SGD(model.parameters(), lr=1e-1)

# 学習
num_epochs = 10
log_train = []
log_valid = []
for epoch in range(num_epochs):
  # 訓練モードに設定
  model.train()
  for inputs, labels in dataloader_train:
    # 勾配をゼロで初期化
    optimizer.zero_grad()

    # 順伝播 + 誤差逆伝播 + 重み更新
    outputs = model(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()

  # 損失と正解率の算出
  loss_train, acc_train = calculate_loss_and_accuracy(model, crit
erion, dataloader_train)
  loss_valid, acc_valid = calculate_loss_and_accuracy(model, crit
erion, dataloader_valid)
  log_train.append([loss_train, acc_train])
  log_valid.append([loss_valid, acc_valid])

  # チェックポイントの保存
  torch.save({'epoch': epoch, 'model_state_dict': model.state_dic
t(), 'optimizer_state_dict': optimizer.state_dict()}, f'checkpoin
t{epoch + 1}.pt')

  # ログを出力
```

```
  print(f'epoch: {epoch + 1}, loss_train: {loss_train:.4f}, accur
acy_train: {acc_train:.4f}, loss_valid: {loss_valid:.4f}, accurac
y_valid: {acc_valid:.4f}')
```

実行結果：

```
epoch: 1, loss_train: 0.3371, accuracy_train: 0.8855, loss_valid: 0.3618, accuracy_valid: 0.8810
epoch: 2, loss_train: 0.2864, accuracy_train: 0.9027, loss_valid: 0.3246, accuracy_valid: 0.8915
epoch: 3, loss_train: 0.2744, accuracy_train: 0.9042, loss_valid: 0.3234, accuracy_valid: 0.8840
epoch: 4, loss_train: 0.2530, accuracy_train: 0.9139, loss_valid: 0.3092, accuracy_valid: 0.8975
epoch: 5, loss_train: 0.2441, accuracy_train: 0.9166, loss_valid: 0.3065, accuracy_valid: 0.8945
epoch: 6, loss_train: 0.2434, accuracy_train: 0.9182, loss_valid: 0.3066, accuracy_valid: 0.8975
epoch: 7, loss_train: 0.2332, accuracy_train: 0.9222, loss_valid: 0.3045, accuracy_valid: 0.8937
epoch: 8, loss_train: 0.2299, accuracy_train: 0.9208, loss_valid: 0.3047, accuracy_valid: 0.8907
epoch: 9, loss_train: 0.2278, accuracy_train: 0.9219, loss_valid: 0.3070, accuracy_valid: 0.8885
epoch: 10, loss_train: 0.2259, accuracy_train: 0.9244, loss_valid: 0.3057, accuracy_valid: 0.8937
```

まとめ：

Torch. save()
まず辞書を作成し、3 つのパラメータを保存します。
state = {'net':model.state_dict(), 'optimizer':optimizer.state_dict(),'epoch':epoch}

torch. save() :torch. save(state, dir) ここで dir は保存ファイルの絶対パス+保存ファイル名を表す

7 7. ミニバッチ化

解答：

```
import time

def train_model(dataset_train, dataset_valid, batch_size, model,
criterion, optimizer, num_epochs):
  # dataloader の作成
  dataloader_train = DataLoader(dataset_train, batch_size=batch_s
ize, shuffle=False)
  dataloader_valid = DataLoader(dataset_valid, batch_size=len(dat
aset_valid), shuffle=False)

  # 学習
  log_train = []
  log_valid = []
  for epoch in range(num_epochs):
    # 開始時刻の記録
    s_time = time.time()
```

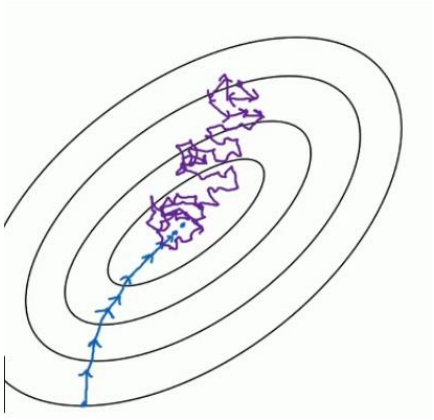
```
    # 訓練モードに設定
    model.train()
    for inputs,labels in dataloader_train:
     # 勾配をゼロで初期化
      optimizer.zero_grad()

      # 順伝播 + 誤差逆伝播 + 重み更新
      outputs = model(inputs)
      loss = criterion(outputs, labels)
      loss.backward()
      optimizer.step()

    # 損失と正解率の算出
    loss_train, acc_train = calculate_loss_and_accuracy(model, cr
iterion, dataloader_train)
    loss_valid, acc_valid = calculate_loss_and_accuracy(model, cr
iterion, dataloader_valid)
    log_train.append([loss_train, acc_train])
    log_valid.append([loss_valid, acc_valid])

    # チェックポイントの保存
    torch.save({'epoch': epoch, 'model_state_dict': model.state_d
ict(), 'optimizer_state_dict': optimizer.state_dict()}, f'checkpo
int{epoch + 1}.pt')

    # 終了時刻の記録
    e_time = time.time()

    # ログを出力
    print(f'epoch: {epoch + 1}, loss_train: {loss_train:.4f}, acc
uracy_train: {acc_train:.4f}, loss_valid: {loss_valid:.4f}, accur
acy_valid: {acc_valid:.4f}, {(e_time - s_time):.4f}sec')

  return {'train': log_train, 'valid': log_valid}

from torch.utils.data import DataLoader

# dataset の作成
dataset_train = NewsDataset(X_train, y_train)
dataset_valid = NewsDataset(X_valid, y_valid)

# モデルの定義
model = SLPNet(300, 4)
```

```
# 損失関数の定義
criterion = nn.CrossEntropyLoss()

# オプティマイザの定義
optimizer = torch.optim.SGD(model.parameters(), lr=1e-1)

# モデルの学習
for batch_size in [2 ** i for i in range(11)]:
  print(f'バッチサイズ: {batch_size}')
  log = train_model(dataset_train, dataset_valid, batch_size, mod
el, criterion, optimizer, 1)
```

実行結果：

```
バッチサイズ: 1
epoch: 1, loss_train: 0.3243, accuracy_train: 0.8899, loss_valid: 0.3601, accuracy_valid: 0.8780, 4.7599sec
バッチサイズ: 2
epoch: 1, loss_train: 0.2985, accuracy_train: 0.8990, loss_valid: 0.3371, accuracy_valid: 0.8847, 2.3002sec
バッチサイズ: 4
epoch: 1, loss_train: 0.2901, accuracy_train: 0.9028, loss_valid: 0.3300, accuracy_valid: 0.8877, 1.1790sec
バッチサイズ: 8
epoch: 1, loss_train: 0.2865, accuracy_train: 0.9042, loss_valid: 0.3274, accuracy_valid: 0.8877, 0.6511sec
バッチサイズ: 16
epoch: 1, loss_train: 0.2848, accuracy_train: 0.9049, loss_valid: 0.3263, accuracy_valid: 0.8870, 0.3906sec
バッチサイズ: 32
epoch: 1, loss_train: 0.2840, accuracy_train: 0.9052, loss_valid: 0.3257, accuracy_valid: 0.8877, 0.2572sec
バッチサイズ: 64
epoch: 1, loss_train: 0.2836, accuracy_train: 0.9055, loss_valid: 0.3255, accuracy_valid: 0.8892, 0.1951sec
バッチサイズ: 128
epoch: 1, loss_train: 0.2831, accuracy_train: 0.9054, loss_valid: 0.3253, accuracy_valid: 0.8892, 0.1550sec
バッチサイズ: 256
epoch: 1, loss_train: 0.2837, accuracy_train: 0.9054, loss_valid: 0.3253, accuracy_valid: 0.8892, 0.1249sec
バッチサイズ: 512
epoch: 1, loss_train: 0.2834, accuracy_train: 0.9054, loss_valid: 0.3252, accuracy_valid: 0.8892, 0.1137sec
バッチサイズ: 1024
epoch: 1, loss_train: 0.2845, accuracy_train: 0.9054, loss_valid: 0.3252, accuracy_valid: 0.8892, 0.2178sec
```

まとめ：



上の図を勾配降下空間と見なす。下の青い部分は full batch で、上は mini batch です。前述の mini batch のように反復損失関数が毎回減少するわけではないので、かなり回り道をしているように見えます。しかし、全体は最適解に向かって反復している。しかも mini batch は epoch 1 つで 5000 歩歩いたので、full batch は epoch 1 つで 1 歩しかありません。だからミニバーチは回り道をしたが、かなり速くなる。

78. GPU上での学習

解答：

```
def calculate_loss_and_accuracy(model, criterion, loader, device)
:
  model.eval()
  loss = 0.0
  total = 0
  correct = 0
  with torch.no_grad():
    for inputs, labels in loader:
      inputs = inputs.to(device)
      labels = labels.to(device)
      outputs = model(inputs)
      loss += criterion(outputs, labels).item()
      pred = torch.argmax(outputs, dim=-1)
      total += len(inputs)
      correct += (pred == labels).sum().item()

  return loss / len(loader), correct / total


def train_model(dataset_train, dataset_valid, batch_size, model,
criterion, optimizer, num_epochs, device=None):
  # GPUに送る
  model.to(device)

  # dataloaderの作成
  dataloader_train = DataLoader(dataset_train, batch_size=batch_s
ize, shuffle=True)
  dataloader_valid = DataLoader(dataset_valid, batch_size=len(dat
aset_valid), shuffle=False)

  # 学習
  log_train = []
  log_valid = []
  for epoch in range(num_epochs):
    # 開始時刻の記録
    s_time = time.time()

    # 訓練モードに設定
    model.train()
    for inputs, labels in dataloader_train:
```

```
            # 勾配をゼロで初期化
            optimizer.zero_grad()

            # 順伝播 + 誤差逆伝播 + 重み更新
            inputs = inputs.to(device)
            labels = labels.to(device)
            outputs = model.forward(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

        # 損失と正解率の算出
        loss_train, acc_train = calculate_loss_and_accuracy(model, cr
iterion, dataloader_train, device)
        loss_valid, acc_valid = calculate_loss_and_accuracy(model, cr
iterion, dataloader_valid, device)
        log_train.append([loss_train, acc_train])
        log_valid.append([loss_valid, acc_valid])

        # チェックポイントの保存
        torch.save({'epoch': epoch, 'model_state_dict': model.state_d
ict(), 'optimizer_state_dict': optimizer.state_dict()}, f'checkpo
int{epoch + 1}.pt')

        # 終了時刻の記録
        e_time = time.time()

        # ログを出力
        print(f'epoch: {epoch + 1}, loss_train: {loss_train:.4f}, acc
uracy_train: {acc_train:.4f}, loss_valid: {loss_valid:.4f}, accur
acy_valid: {acc_valid:.4f}, {(e_time - s_time):.4f}sec')

    return {'train': log_train, 'valid': log_valid}
```

実行結果：

```
バッチサイズ: 1
epoch: 1, loss_train: 0.3338, accuracy_train: 0.8838, loss_valid: 0.3592, accuracy_valid: 0.8780, 12.4628sec
バッチサイズ: 2
epoch: 1, loss_train: 0.3058, accuracy_train: 0.8951, loss_valid: 0.3385, accuracy_valid: 0.8832, 4.8231sec
バッチサイズ: 4
epoch: 1, loss_train: 0.2963, accuracy_train: 0.8995, loss_valid: 0.3316, accuracy_valid: 0.8907, 2.5039sec
バッチサイズ: 8
epoch: 1, loss_train: 0.2908, accuracy_train: 0.9014, loss_valid: 0.3272, accuracy_valid: 0.8885, 1.3289sec
バッチサイズ: 16
epoch: 1, loss_train: 0.2891, accuracy_train: 0.9013, loss_valid: 0.3269, accuracy_valid: 0.8937, 0.7169sec
バッチサイズ: 32
epoch: 1, loss_train: 0.2879, accuracy_train: 0.9023, loss_valid: 0.3260, accuracy_valid: 0.8937, 0.4529sec
```

```
バッチサイズ: 64
epoch: 1, loss_train: 0.2875, accuracy_train: 0.9025, loss_valid: 0.3256, accuracy_valid: 0.8930, 0.3794sec
バッチサイズ: 128
epoch: 1, loss_train: 0.2869, accuracy_train: 0.9025, loss_valid: 0.3255, accuracy_valid: 0.8937, 0.2607sec
バッチサイズ: 256
epoch: 1, loss_train: 0.2869, accuracy_train: 0.9025, loss_valid: 0.3254, accuracy_valid: 0.8937, 0.2211sec
バッチサイズ: 512
epoch: 1, loss_train: 0.2871, accuracy_train: 0.9025, loss_valid: 0.3254, accuracy_valid: 0.8937, 0.1904sec
バッチサイズ: 1024
epoch: 1, loss_train: 0.2888, accuracy_train: 0.9025, loss_valid: 0.3253, accuracy_valid: 0.8937, 0.3066sec
```

まとめ：

ここで、device=torch.device(「cpu」)は cpu を使用し、device=torch.device(「cuda」)は GPU を使用します。
デバイスを指定すると、モデルを対応するデバイスにロードする必要があります。その場合は、model=model.to（device）を使用して、モデルを対応するデバイスにロードする必要があります。

79. 多層ニューラルネットワーク

解答：

```
from torch.nn import functional as F

class MLPNet(nn.Module):
  def __init__(self, input_size, mid_size, output_size, mid_layer
s):
    super().__init__()
    self.mid_layers = mid_layers
    self.fc = nn.Linear(input_size, mid_size)
    self.fc_mid = nn.Linear(mid_size, mid_size)
    self.fc_out = nn.Linear(mid_size, output_size)
    self.bn = nn.BatchNorm1d(mid_size)

  def forward(self, x):
    x = F.relu(self.fc(x))
    for _ in range(self.mid_layers):
      x = F.relu(self.bn(self.fc_mid(x)))
    x = F.relu(self.fc_out(x))

    return x
from torch import optim

def calculate_loss_and_accuracy(model, criterion, loader, device)
:
```

```
  model.eval()
  loss = 0.0
  total = 0
  correct = 0
  with torch.no_grad():
    for inputs, labels in loader:
      inputs = inputs.to(device)
      labels = labels.to(device)
      outputs = model(inputs)
      loss += criterion(outputs, labels).item()
      pred = torch.argmax(outputs, dim=-1)
      total += len(inputs)
      correct += (pred == labels).sum().item()

  return loss / len(loader), correct / total


def train_model(dataset_train, dataset_valid, batch_size, model,
criterion, optimizer, num_epochs, device=None):
  # GPU に送る
  model.to(device)

  # dataloader の作成
  dataloader_train = DataLoader(dataset_train, batch_size=batch_s
ize, shuffle=True)
  dataloader_valid = DataLoader(dataset_valid, batch_size=len(dat
aset_valid), shuffle=False)

  # スケジューラの設定
  scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, num
_epochs, eta_min=1e-5, last_epoch=-1)

  # 学習
  log_train = []
  log_valid = []
  for epoch in range(num_epochs):
    # 開始時刻の記録
    s_time = time.time()

    # 訓練モードに設定
    model.train()
    for inputs, labels in dataloader_train:
      # 勾配をゼロで初期化
      optimizer.zero_grad()
```

```
            # 順伝播 + 誤差逆伝播 + 重み更新
            inputs = inputs.to(device)
            labels = labels.to(device)
            outputs = model.forward(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

        # 損失と正解率の算出
        loss_train, acc_train = calculate_loss_and_accuracy(model, cr
iterion, dataloader_train, device)
        loss_valid, acc_valid = calculate_loss_and_accuracy(model, cr
iterion, dataloader_valid, device)
        log_train.append([loss_train, acc_train])
        log_valid.append([loss_valid, acc_valid])

        # チェックポイントの保存
        torch.save({'epoch': epoch, 'model_state_dict': model.state_d
ict(), 'optimizer_state_dict': optimizer.state_dict()}, f'checkpo
int{epoch + 1}.pt')

        # 終了時刻の記録
        e_time = time.time()

        # ログを出力
        print(f'epoch: {epoch + 1}, loss_train: {loss_train:.4f}, acc
uracy_train: {acc_train:.4f}, loss_valid: {loss_valid:.4f}, accur
acy_valid: {acc_valid:.4f}, {(e_time - s_time):.4f}sec')

        # 検証データの損失が 3 エポック連続で低下しなかった場合は学習終了
        if epoch > 2 and log_valid[epoch - 3][0] <= log_valid[epoch -
 2][0] <= log_valid[epoch - 1][0] <= log_valid[epoch][0]:
            break

        # スケジューラを 1 ステップ進める
        scheduler.step()

    return {'train': log_train, 'valid': log_valid}
# dataset の作成
dataset_train = NewsDataset(X_train, y_train)
dataset_valid = NewsDataset(X_valid, y_valid)

# モデルの定義
```

```
model = MLPNet(300, 200, 4, 1)

# 損失関数の定義
criterion = nn.CrossEntropyLoss()

# オプティマイザの定義
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)

# デバイスの指定
device = torch.device('cuda')

# モデルの学習
log = train_model(dataset_train, dataset_valid, 64, model, criter
ion, optimizer, 1000, device)
```

実行結果：

```
epoch: 1, loss_train: 1.1663, accuracy_train: 0.4372, loss_valid: 1.1620, accuracy_valid: 0.4379, 0.4955sec
epoch: 2, loss_train: 1.0511, accuracy_train: 0.5255, loss_valid: 1.0498, accuracy_valid: 0.5225, 0.4383sec
epoch: 3, loss_train: 0.8347, accuracy_train: 0.7569, loss_valid: 0.8384, accuracy_valid: 0.7627, 0.4611sec
epoch: 4, loss_train: 0.6991, accuracy_train: 0.7761, loss_valid: 0.7025, accuracy_valid: 0.7754, 0.4387sec
epoch: 5, loss_train: 0.6428, accuracy_train: 0.7792, loss_valid: 0.6479, accuracy_valid: 0.7792, 0.4594sec
epoch: 6, loss_train: 0.5990, accuracy_train: 0.7852, loss_valid: 0.6068, accuracy_valid: 0.7792, 0.4472sec
epoch: 7, loss_train: 0.5711, accuracy_train: 0.7921, loss_valid: 0.5796, accuracy_valid: 0.7874, 0.4610sec
epoch: 8, loss_train: 0.5406, accuracy_train: 0.8034, loss_valid: 0.5505, accuracy_valid: 0.8001, 0.4412sec
epoch: 9, loss_train: 0.5163, accuracy_train: 0.8190, loss_valid: 0.5279, accuracy_valid: 0.8099, 0.4446sec
```

結果の一部

まとめ：

MLP：多層感知器（Multi-Layer Perceptron、MLP）は人工ニューラルネットワーク（Artificial Neural Network、ANN）とも呼ばれ、入出力層を除いて、その中間に複数の隠線層があることができる。最も簡単な MLP には、単純なニューラルネットワークと呼ぶには、入力層、隠線層、出力層という隠線層が必要です。習慣の原因は私は後でニューラルネットワークと呼ばれます。一般的に言えば、ニューラルネットワークは生物ニューラルネットワークを模倣した技術であり、複数の特徴値を接続し、線形と非線形の組み合わせを経て、最終的に 1 つの目標を達成することによって。
文の長さが一定ではないので、Bag-of-Word-Vectors を使って簡単に語ベクトルを加算し、MLP を使うのが一般的です。この方法は比較的簡単で、訓練速度が速く、結果もそれほど悪くありません。コンテキスト情報を利用していないだけです。