8 0. ID 番号への変換

解答：

```
from collections import defaultdict
import string

# 単語の頻度集計
d = defaultdict(int)
table = str.maketrans(string.punctuation, ' '*len(string.punctuation))  # 記号をスペースに置換するテーブル
for text in train['TITLE']:
  for word in text.translate(table).split():
    d[word] += 1
d = sorted(d.items(), key=lambda x:x[1], reverse=True)

# 単語 ID 辞書の作成
word2id = {word: i + 1 for i, (word, cnt) in enumerate(d) if cnt > 1}  # 出現頻度が 2 回以上の単語を登録

print(f'ID 数: {len(set(word2id.values()))}\n')
print('頻度上位 20 語')
for key in list(word2id)[:20]:
    print(f'{key}: {word2id[key]}')

def tokenizer(text, word2id=word2id, unk=0):
  """ 入力テキストをスペースで分割し ID 列に変換(辞書になければ unk で指定した数字を設定)"""
  table = str.maketrans(string.punctuation, ' '*len(string.punctuation))
  return [word2id.get(word, unk) for word in text.translate(table).split()]

# 確認
text = train.iloc[1, train.columns.get_loc('TITLE')]
print(f'テキスト: {text}')
print(f'ID 列: {tokenizer(text)}')
```

実行結果：

```
テキスト: Amazon Plans to Fight FTC Over Mobile-App Purchases
ID列: [169, 539, 1, 683, 1237, 82, 279, 1898, 4199]
```

まとめ：

Defaultdict(): 辞書で値を検索すると、key が存在しない場合はデフォルトではなく KeyError エラーが返され、

defaultdict 関数を使用できます。

例えば：

```
lst = ['A', 'B', 'C', 'D', 'e']
dic = {}

for i in lst:
    dic[i] += 1
print(dic)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-5-6a40d983f141> in <module>
      3
      4 for i in lst:
----> 5     dic[i] += 1
      6 print(dic)

KeyError: 'A'
```

```
lst = ['A', 'B', 'C', 'D', 'e']
dic = {}

for i in lst:
    dic.setdefault(i, 0)
    dic[i] += 1
print(dic)
```

```
{'A': 1, 'B': 1, 'C': 1, 'D': 1, 'e': 1}
```

8 1. ID 番号への変換

解答：

```
import torch
from torch import nn

class RNN(nn.Module):
  def __init__(self, vocab_size, emb_size, padding_idx, output_size, hidden_size):
    super().__init__()
    self.hidden_size = hidden_size
    self.emb = nn.Embedding(vocab_size, emb_size, padding_idx=padding_idx)
    self.rnn = nn.RNN(emb_size, hidden_size, nonlinearity='tanh', batch_first=True)
    self.fc = nn.Linear(hidden_size, output_size)

  def forward(self, x):
    self.batch_size = x.size()[0]
    hidden = self.init_hidden(x.device)  # h0 のゼロベクトルを作成
    emb = self.emb(x)
    # emb.size() = (batch_size, seq_len, emb_size)
    out, hidden = self.rnn(emb, hidden)
    # out.size() = (batch_size, seq_len, hidden_size)
    out = self.fc(out[:, -1, :])
    # out.size() = (batch_size, output_size)
```

```python
    return out

  def init_hidden(self, device):
    hidden = torch.zeros(1, self.batch_size, self.hidden_size, device=device)
    return hidden

from torch.utils.data import Dataset

class CreateDataset(Dataset):
  def __init__(self, X, y, tokenizer):
    self.X = X
    self.y = y
    self.tokenizer = tokenizer

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

  def __getitem__(self, index):  # Dataset[index]で返す値を指定
    text = self.X[index]
    inputs = self.tokenizer(text)

    return {
      'inputs': torch.tensor(inputs, dtype=torch.int64),
      'labels': torch.tensor(self.y[index], dtype=torch.int64)
    }

# ラベルベクトルの作成
category_dict = {'b': 0, 't': 1, 'e':2, 'm':3}
y_train = train['CATEGORY'].map(lambda x: category_dict[x]).values
y_valid = valid['CATEGORY'].map(lambda x: category_dict[x]).values
y_test = test['CATEGORY'].map(lambda x: category_dict[x]).values

# Dataset の作成
dataset_train = CreateDataset(train['TITLE'], y_train, tokenizer)
dataset_valid = CreateDataset(valid['TITLE'], y_valid, tokenizer)
dataset_test = CreateDataset(test['TITLE'], y_test, tokenizer)

print(f'len(Dataset)の出力: {len(dataset_train)}')
print('Dataset[index]の出力:')
for var in dataset_train[1]:
  print(f'  {var}: {dataset_train[1][var]}')

# パラメータの設定
VOCAB_SIZE = len(set(word2id.values())) + 1  # 辞書の ID 数 + パディング ID
EMB_SIZE = 300
PADDING_IDX = len(set(word2id.values()))
OUTPUT_SIZE = 4
HIDDEN_SIZE = 50
```

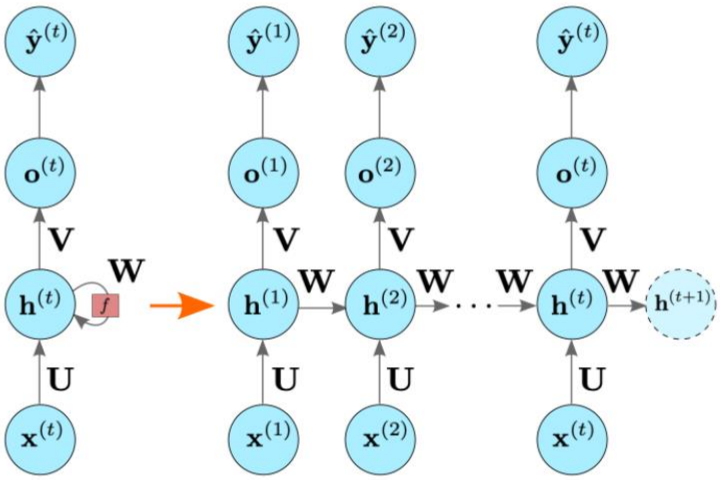
```python
# モデルの定義
model = RNN(VOCAB_SIZE, EMB_SIZE, PADDING_IDX, OUTPUT_SIZE, HIDDEN_SIZE)

# 先頭10件の予測値取得
for i in range(10):
  X = dataset_train[i]['inputs']
  print(torch.softmax(model(X.unsqueeze(0)), dim=-1))
```

実行結果：

```
len(Dataset)の出力: 10684
Dataset[index]の出力:
  inputs: tensor([ 169,   539,     1,   683,  1237,    82,   279,  1898,  4199])
  labels: 1

tensor([[0.3734, 0.1783, 0.2586, 0.1898]], grad_fn=<SoftmaxBackward0>)
tensor([[0.2396, 0.1822, 0.1574, 0.4208]], grad_fn=<SoftmaxBackward0>)
tensor([[0.3754, 0.2619, 0.1763, 0.1863]], grad_fn=<SoftmaxBackward0>)
tensor([[0.3235, 0.3424, 0.1401, 0.1939]], grad_fn=<SoftmaxBackward0>)
tensor([[0.1947, 0.2843, 0.2630, 0.2579]], grad_fn=<SoftmaxBackward0>)
tensor([[0.2378, 0.3448, 0.2350, 0.1824]], grad_fn=<SoftmaxBackward0>)
tensor([[0.3298, 0.2474, 0.3094, 0.1135]], grad_fn=<SoftmaxBackward0>)
tensor([[0.4547, 0.2189, 0.0605, 0.2660]], grad_fn=<SoftmaxBackward0>)
tensor([[0.2966, 0.1322, 0.4413, 0.1299]], grad_fn=<SoftmaxBackward0>)
tensor([[0.4311, 0.2433, 0.1396, 0.1860]], grad_fn=<SoftmaxBackward0>)
```

まとめ：

RNNのモデル紹介（https://cvml-expertguide.net/terms/dl/rnn/）：



(a) 簡略形　　(b) 展開形

ここで，具体的な各変数の役割をイメージしやすくなるように，「自然言語処理(NLP)でのRNNによる文章予測」における3つの変数例を以下に提示してみる：

$X^{(t)}$：入力の特徴表現ベクトル．ニューラル言語モデルとしてRNNを用いる場合は，元の単語を低次元ベクトルに埋め込んだword2vecやGloVeなどの，単語の埋め込みベクトル（分散表現）を用いることが多い

$h^{(t)}$：潜在変数ベクトル．系列の最初からフレームまでの，潜在状態の変化全てを蓄積した記憶に相当する．つまり系列全体のデータの情報を状態変数として保持しているものであるが，RNN の基本モデルの場合は，あまり昔のフレームの記憶までは保持できていない．ただ，LSTM や GRU などでは，少し長期の状態まで記憶している．

$o^{(t)}$：出力のベクトル．出力が連続値のスカラーやベクトルの場合は，これが最終的な各フレームでの予測出力となる．

$y^{(t)}$：出力のクラス確率．自然言語処理系や音声認識・音声生成の RNN では，次のフレームの単語クラス確率を予測する(語彙サイズがの場合).

82．確率的勾配降下法による学習

解答：

```
from torch.utils.data import DataLoader
import time
from torch import optim

def calculate_loss_and_accuracy(model, dataset, device=None, criterion=None):
  """損失・正解率を計算"""
  dataloader = DataLoader(dataset, batch_size=1, shuffle=False)
  loss = 0.0
  total = 0
  correct = 0
  with torch.no_grad():
    for data in dataloader:
      # デバイスの指定
      inputs = data['inputs'].to(device)
      labels = data['labels'].to(device)

      # 順伝播
      outputs = model(inputs)

      # 損失計算
      if criterion != None:
        loss += criterion(outputs, labels).item()

      # 正解率計算
      pred = torch.argmax(outputs, dim=-1)
      total += len(inputs)
      correct += (pred == labels).sum().item()

  return loss / len(dataset), correct / total


def train_model(dataset_train, dataset_valid, batch_size, model, criterion, optimiz
er, num_epochs, collate_fn=None, device=None):
```

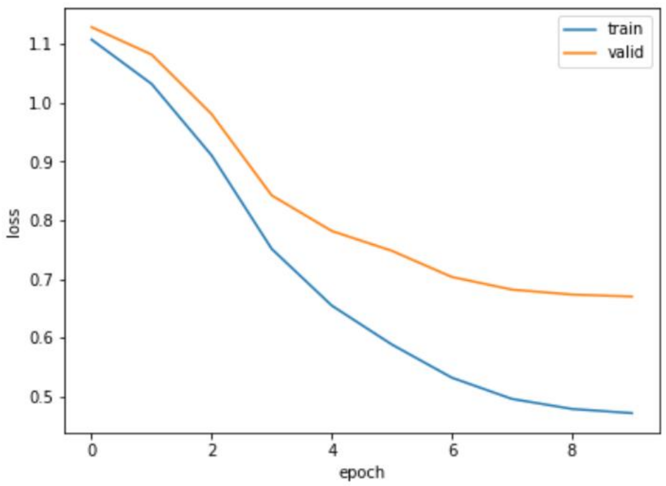
```
  """モデルの学習を実行し、損失・正解率のログを返す"""

  # デバイスの指定
  model.to(device)

  # dataloaderの作成
  dataloader_train = DataLoader(dataset_train, batch_size=batch_size, shuffle=True,
 collate_fn=collate_fn)
  dataloader_valid = DataLoader(dataset_valid, batch_size=1, shuffle=False)

  # スケジューラの設定
  scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, num_epochs, eta_min=1
e-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 data 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 = data['inputs'].to(device)
      labels = data['labels'].to(device)
      outputs = model(inputs)
      loss = criterion(outputs, labels)
      loss.backward()
      optimizer.step()

    # 評価モードに設定
    model.eval()

    # 損失と正解率の算出
    loss_train, acc_train = calculate_loss_and_accuracy(model, dataset_train, devic
e, criterion=criterion)
    loss_valid, acc_valid = calculate_loss_and_accuracy(model, dataset_valid, devic
e, criterion=criterion)
    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_dict(), 'optimizer_
state_dict': optimizer.state_dict()}, f'checkpoint{epoch + 1}.pt')
```

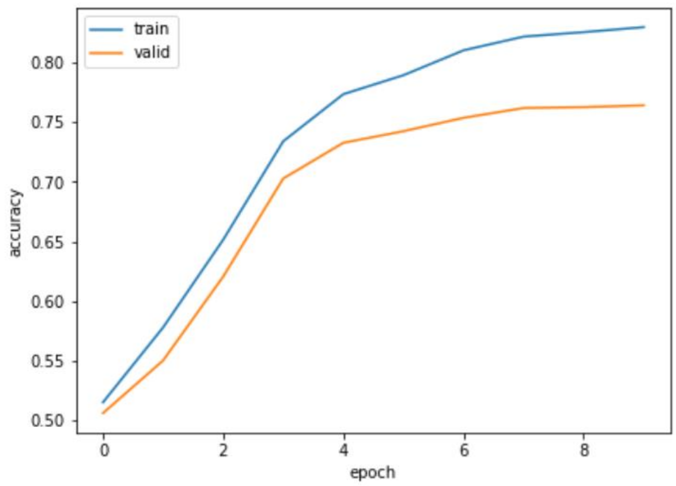
```python
    # 終了時刻の記録
    e_time = time.time()

    # ログを出力
    print(f'epoch: {epoch + 1}, loss_train: {loss_train:.4f}, accuracy_train: {acc_train:.4f}, loss_valid: {loss_valid:.4f}, accuracy_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}

import numpy as np
from matplotlib import pyplot as plt

def visualize_logs(log):
  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()

# パラメータの設定
VOCAB_SIZE = len(set(word2id.values())) + 1
EMB_SIZE = 300
PADDING_IDX = len(set(word2id.values()))
OUTPUT_SIZE = 4
HIDDEN_SIZE = 50
LEARNING_RATE = 1e-3
BATCH_SIZE = 1
NUM_EPOCHS = 10

# モデルの定義
model = RNN(VOCAB_SIZE, EMB_SIZE, PADDING_IDX, OUTPUT_SIZE, HIDDEN_SIZE)
```

```
# 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=LEARNING_RATE)

# モデルの学習
log = train_model(dataset_train, dataset_valid, BATCH_SIZE, model, criterion, optimizer, NUM_EPOCHS)
```

実行結果：

```
epoch: 1, loss_train: 1.1077, accuracy_train: 0.5151, loss_valid: 1.1289, accuracy_valid: 0.5060, 116.8172sec
epoch: 2, loss_train: 1.0321, accuracy_train: 0.5779, loss_valid: 1.0822, accuracy_valid: 0.5501, 114.4225sec
epoch: 3, loss_train: 0.9103, accuracy_train: 0.6514, loss_valid: 0.9806, accuracy_valid: 0.6205, 120.6466sec
epoch: 4, loss_train: 0.7512, accuracy_train: 0.7341, loss_valid: 0.8423, accuracy_valid: 0.7028, 117.1084sec
epoch: 5, loss_train: 0.6546, accuracy_train: 0.7736, loss_valid: 0.7816, accuracy_valid: 0.7328, 115.6100sec
epoch: 6, loss_train: 0.5883, accuracy_train: 0.7895, loss_valid: 0.7479, accuracy_valid: 0.7425, 118.9912sec
epoch: 7, loss_train: 0.5318, accuracy_train: 0.8104, loss_valid: 0.7033, accuracy_valid: 0.7537, 119.3757sec
epoch: 8, loss_train: 0.4956, accuracy_train: 0.8219, loss_valid: 0.6818, accuracy_valid: 0.7620, 119.1641sec
epoch: 9, loss_train: 0.4785, accuracy_train: 0.8256, loss_valid: 0.6734, accuracy_valid: 0.7627, 114.7393sec
epoch: 10, loss_train: 0.4714, accuracy_train: 0.8298, loss_valid: 0.6700, accuracy_valid: 0.7642, 117.6056sec
```






```
正解率（学習データ）：0.830
正解率（評価データ）：0.775
```

まとめ：

確率的勾配降下法（かくりつてきこうばいこうかほう、英: stochastic gradient descent, SGD）は、連続最適化問題に対する勾配法の乱択アルゴリズム。バッチ学習である最急降下法をオンライン学習に改良したアルゴリズムである。目的関数が微分可能な和の形であることを必要とする。

データセットを $\mathbf{x} = (x_1, x_2, \cdots, x_n)$、最適化対象のパラ メータを W としたとき、目標関数 f（w; x）が以下のように 各サンプルから計算される関数 fi（w；xi）の和の形で表せる 場合を対象とします。

$$f(\mathbf{w};\mathbf{x}) = \sum_{i=1}^{N} f_i(\mathbf{w};x_i)$$

例えば、最小二乗誤差はサンプル x_i に対応する正解を y_i、 パラメータで表される関数の形状（$g(x)=ax+b$）を $g(x)$としたとき、

$f(\mathbf{w};\mathbf{x}) = \frac{1}{N}\sum_{i=1}^{N}(g(\mathbf{w};x_i) - y_i)^2$ と和で表す形になっています。

確率的勾配降下法は、すべてのサンプルを使って損失を計算 する代わりに、ランダムに選んだサンプル 1 つで計算した f_i の勾配でパラメータを更新します。

8 3. ミニバッチ化・GPU 上での学習

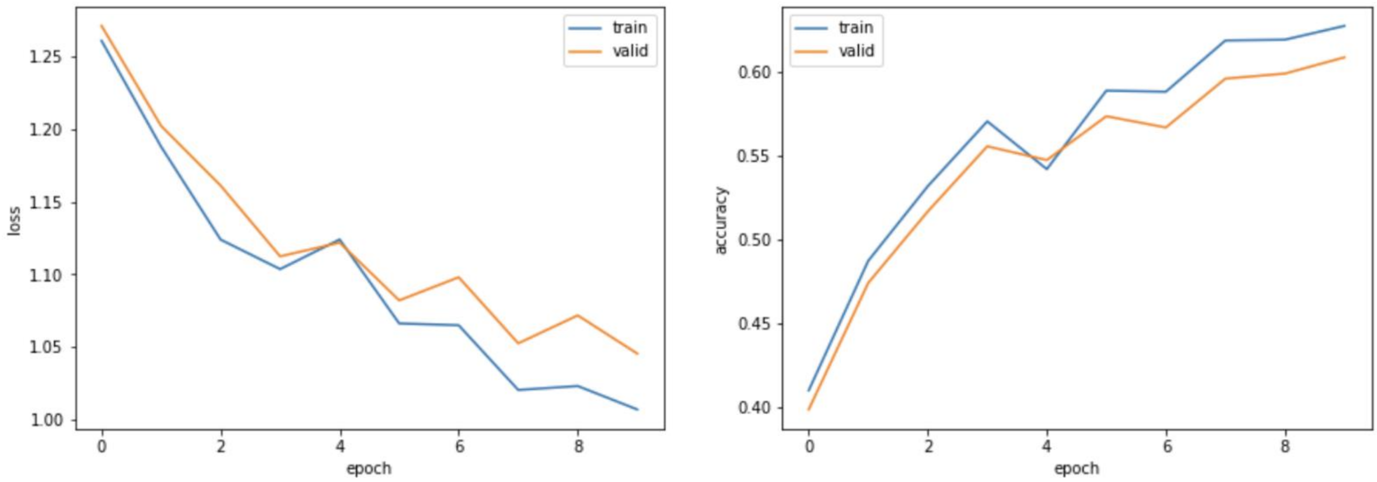
解答：

```
class Padsequence():
  """Dataloaderからミニバッチを取り出すごとに最大系列長でパディング"""
  def __init__(self, padding_idx):
    self.padding_idx = padding_idx

  def __call__(self, batch):
    sorted_batch = sorted(batch, key=lambda x: x['inputs'].shape[0], reverse=True)
    sequences = [x['inputs'] for x in sorted_batch]
    sequences_padded = torch.nn.utils.rnn.pad_sequence(sequences, batch_first=True,
 padding_value=self.padding_idx)
    labels = torch.LongTensor([x['labels'] for x in sorted_batch])

    return {'inputs': sequences_padded, 'labels': labels}

# パラメータの設定
VOCAB_SIZE = len(set(word2id.values())) + 1
EMB_SIZE = 300
PADDING_IDX = len(set(word2id.values()))
OUTPUT_SIZE = 4
HIDDEN_SIZE = 50
LEARNING_RATE = 5e-2
BATCH_SIZE = 32
NUM_EPOCHS = 10

# モデルの定義
model = RNN(VOCAB_SIZE, EMB_SIZE, PADDING_IDX, OUTPUT_SIZE, HIDDEN_SIZE)

# 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=LEARNING_RATE)

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

# モデルの学習
log = train_model(dataset_train, dataset_valid, BATCH_SIZE, model, criterion, optim
izer, NUM_EPOCHS, collate_fn=Padsequence(PADDING_IDX), device=device)
```

```python
# ログの可視化
visualize_logs(log)

# 正解率の算出
_, acc_train = calculate_loss_and_accuracy(model, dataset_train, device)
_, acc_test = calculate_loss_and_accuracy(model, dataset_test, device)
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：

```
epoch: 1, loss_train: 1.2608, accuracy_train: 0.4096, loss_valid: 1.2711, accuracy_valid: 0.3982, 8.5319sec
epoch: 2, loss_train: 1.1879, accuracy_train: 0.4870, loss_valid: 1.2022, accuracy_valid: 0.4738, 7.4613sec
epoch: 3, loss_train: 1.1239, accuracy_train: 0.5315, loss_valid: 1.1611, accuracy_valid: 0.5165, 7.6046sec
epoch: 4, loss_train: 1.1036, accuracy_train: 0.5703, loss_valid: 1.1124, accuracy_valid: 0.5554, 7.5934sec
epoch: 5, loss_train: 1.1240, accuracy_train: 0.5417, loss_valid: 1.1218, accuracy_valid: 0.5472, 7.5613sec
epoch: 6, loss_train: 1.0662, accuracy_train: 0.5886, loss_valid: 1.0821, accuracy_valid: 0.5734, 7.6435sec
epoch: 7, loss_train: 1.0649, accuracy_train: 0.5880, loss_valid: 1.0980, accuracy_valid: 0.5666, 7.6188sec
epoch: 8, loss_train: 1.0204, accuracy_train: 0.6186, loss_valid: 1.0525, accuracy_valid: 0.5958, 7.5759sec
epoch: 9, loss_train: 1.0231, accuracy_train: 0.6191, loss_valid: 1.0717, accuracy_valid: 0.5988, 7.5654sec
epoch: 10, loss_train: 1.0069, accuracy_train: 0.6273, loss_valid: 1.0454, accuracy_valid: 0.6085, 7.6461sec
```




```
正解率（学習データ）：0.627
正解率（評価データ）：0.605
```

まとめ：

84. 単語ベクトルの導入

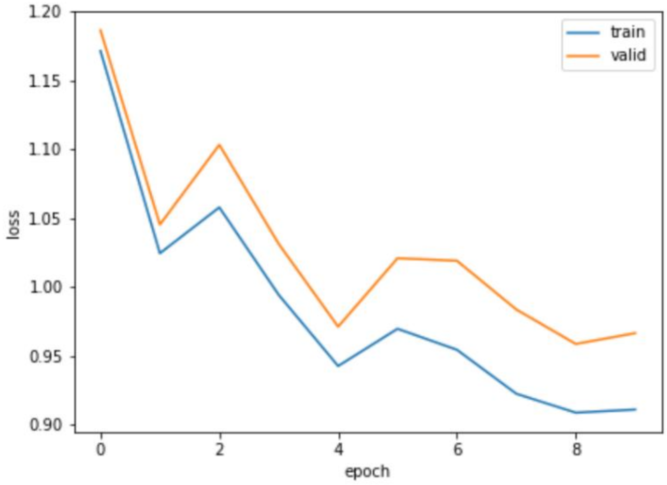
解答：

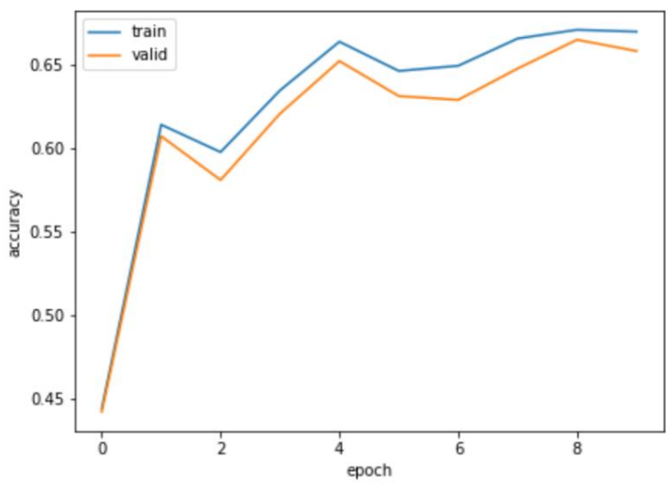
```python
from gensim.models import KeyedVectors
import numpy as np
# 学習済みモデルのロード
model = KeyedVectors.load_word2vec_format('./GoogleNews-vectors-negative300.bin.gz', binary=True)

# 学習済み単語ベクトルの取得
VOCAB_SIZE = len(set(word2id.values())) + 1
```

```python
EMB_SIZE = 300
weights = np.zeros((VOCAB_SIZE, EMB_SIZE))
words_in_pretrained = 0
for i, word in enumerate(word2id.keys()):
  try:
    weights[i] = model[word]
    words_in_pretrained += 1
  except KeyError:
    weights[i] = np.random.normal(scale=0.4, size=(EMB_SIZE,))
weights = torch.from_numpy(weights.astype((np.float32)))

print(f'学習済みベクトル利用単語数: {words_in_pretrained} / {VOCAB_SIZE}')
print(weights.size())

# パラメータの設定
VOCAB_SIZE = len(set(word2id.values())) + 1
EMB_SIZE = 300
PADDING_IDX = len(set(word2id.values()))
OUTPUT_SIZE = 4
HIDDEN_SIZE = 50
NUM_LAYERS = 2
LEARNING_RATE = 5e-2
BATCH_SIZE = 32
NUM_EPOCHS = 10

# モデルの定義
model = RNN(VOCAB_SIZE, EMB_SIZE, PADDING_IDX, OUTPUT_SIZE, HIDDEN_SIZE, NUM_LAYERS
, emb_weights=weights, bidirectional=True)

# 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=LEARNING_RATE)

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

# モデルの学習
log = train_model(dataset_train, dataset_valid, BATCH_SIZE, model, criterion, optim
izer, NUM_EPOCHS, collate_fn=Padsequence(PADDING_IDX), device=device)
```

実行結果：

```
epoch: 1, loss_train: 1.1711, accuracy_train: 0.4129, loss_valid: 1.1851, accuracy_valid: 0.4057, 7.9074sec
epoch: 2, loss_train: 1.0893, accuracy_train: 0.5586, loss_valid: 1.1086, accuracy_valid: 0.5329, 7.5439sec
epoch: 3, loss_train: 1.0519, accuracy_train: 0.6215, loss_valid: 1.0895, accuracy_valid: 0.5966, 7.4945sec
epoch: 4, loss_train: 1.1317, accuracy_train: 0.5617, loss_valid: 1.1902, accuracy_valid: 0.5419, 7.5341sec
epoch: 5, loss_train: 1.0519, accuracy_train: 0.5904, loss_valid: 1.1049, accuracy_valid: 0.5734, 7.4435sec
epoch: 6, loss_train: 1.0585, accuracy_train: 0.5918, loss_valid: 1.1290, accuracy_valid: 0.5659, 7.6834sec
epoch: 7, loss_train: 0.9157, accuracy_train: 0.6677, loss_valid: 0.9676, accuracy_valid: 0.6460, 7.8654sec
epoch: 8, loss_train: 0.9307, accuracy_train: 0.6578, loss_valid: 0.9933, accuracy_valid: 0.6243, 7.5633sec
epoch: 9, loss_train: 0.9189, accuracy_train: 0.6623, loss_valid: 0.9748, accuracy_valid: 0.6370, 7.5143sec
epoch: 10, loss_train: 0.9347, accuracy_train: 0.6531, loss_valid: 0.9944, accuracy_valid: 0.6265, 7.5765sec
```






```
正解率（学習データ）：0.670
正解率（評価データ）：0.663
```

まとめ：

事前学習済み単語ベクトルをモデルに利用する場合、その単語をすべて利用する方法（辞書を置き換える方法）と、手元のデータの辞書はそのまま利用し、それらの単語ベクトルの初期値としてのみ利用する方法があります。 今回は後者の方法を採用し、すでに作成している辞書に対応する単語ベクトルを抽出します。

8 5. 双方向 RNN・多層化

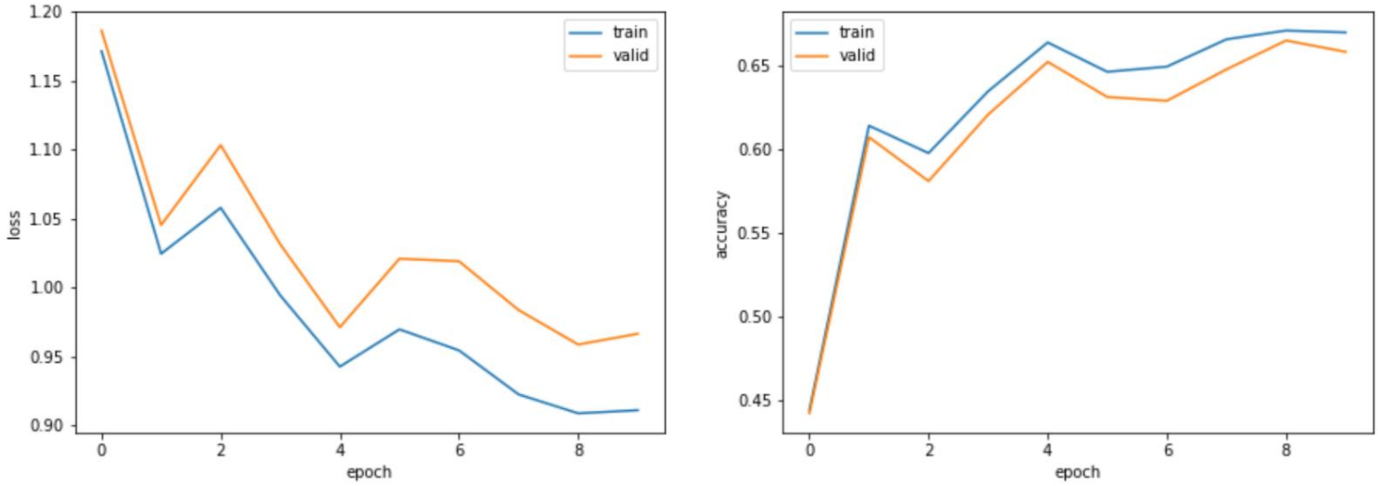
解答：

```
# パラメータの設定
VOCAB_SIZE = len(set(word2id.values())) + 1
EMB_SIZE = 300
PADDING_IDX = len(set(word2id.values()))
OUTPUT_SIZE = 4
HIDDEN_SIZE = 50
NUM_LAYERS = 2
LEARNING_RATE = 5e-2
BATCH_SIZE = 32
NUM_EPOCHS = 10

# モデルの定義
model = RNN(VOCAB_SIZE, EMB_SIZE, PADDING_IDX, OUTPUT_SIZE, HIDDEN_SIZE, NUM_LAYERS
, emb_weights=weights, bidirectional=True)

# 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=LEARNING_RATE)

# デバイスの指定
device = torch.cuda.set_device(0)

# モデルの学習
log = train_model(dataset_train, dataset_valid, BATCH_SIZE, model, criterion, optim
izer, NUM_EPOCHS, collate_fn=Padsequence(PADDING_IDX), device=device)
# ログの可視化
visualize_logs(log)

# 正解率の算出
_, acc_train = calculate_loss_and_accuracy(model, dataset_train, device)
_, acc_test = calculate_loss_and_accuracy(model, dataset_test, device)
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：

```
epoch: 1, loss_train: 1.1712, accuracy_train: 0.4441, loss_valid: 1.1861, accuracy_valid: 0.4424, 14.8131sec
epoch: 2, loss_train: 1.0243, accuracy_train: 0.6139, loss_valid: 1.0451, accuracy_valid: 0.6070, 12.4651sec
epoch: 3, loss_train: 1.0576, accuracy_train: 0.5975, loss_valid: 1.1031, accuracy_valid: 0.5808, 12.6171sec
epoch: 4, loss_train: 0.9941, accuracy_train: 0.6344, loss_valid: 1.0311, accuracy_valid: 0.6205, 12.4815sec
epoch: 5, loss_train: 0.9425, accuracy_train: 0.6635, loss_valid: 0.9711, accuracy_valid: 0.6519, 12.6238sec
epoch: 6, loss_train: 0.9695, accuracy_train: 0.6460, loss_valid: 1.0207, accuracy_valid: 0.6310, 12.4684sec
epoch: 7, loss_train: 0.9543, accuracy_train: 0.6491, loss_valid: 1.0189, accuracy_valid: 0.6287, 12.9547sec
epoch: 8, loss_train: 0.9224, accuracy_train: 0.6654, loss_valid: 0.9835, accuracy_valid: 0.6475, 12.8360sec
epoch: 9, loss_train: 0.9087, accuracy_train: 0.6706, loss_valid: 0.9585, accuracy_valid: 0.6647, 12.4603sec
epoch: 10, loss_train: 0.9109, accuracy_train: 0.6695, loss_valid: 0.9663, accuracy_valid: 0.6579, 12.6056sec
```




```
正解率（学習データ）：0.670
正解率（評価データ）：0.663
```

まとめ：

双方向を指定する引数である bidirectional を True とし、また NUM_LAYERS を 2 に設定して学習を実行します。

8 6. 畳み込みニューラルネットワーク（CNN）

解答：

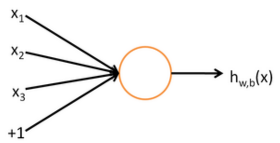
```
from torch.nn import functional as F

class CNN(nn.Module):
  def __init__(self, vocab_size, emb_size, padding_idx, output_size, out_channels, 
kernel_heights, stride, padding, emb_weights=None):
    super().__init__()
    if emb_weights != None:  # 指定があれば埋め込み層の重みを emb_weights で初期化
      self.emb = nn.Embedding.from_pretrained(emb_weights, padding_idx=padding_idx)
    else:
      self.emb = nn.Embedding(vocab_size, emb_size, padding_idx=padding_idx)
    self.conv = nn.Conv2d(1, out_channels, (kernel_heights, emb_size), stride, (pad
ding, 0))
    self.drop = nn.Dropout(0.3)
    self.fc = nn.Linear(out_channels, output_size)

  def forward(self, x):
    # x.size() = (batch_size, seq_len)
    emb = self.emb(x).unsqueeze(1)
    # emb.size() = (batch_size, 1, seq_len, emb_size)
    conv = self.conv(emb)
    # conv.size() = (batch_size, out_channels, seq_len, 1)
    act = F.relu(conv.squeeze(3))
    # act.size() = (batch_size, out_channels, seq_len)
    max_pool = F.max_pool1d(act, act.size()[2])
    # max_pool.size() = (batch_size, out_channels, 1) -> seq_len 方向に最大値を取得
    out = self.fc(self.drop(max_pool.squeeze(2)))
    # out.size() = (batch_size, output_size)
    return out
# パラメータの設定
VOCAB_SIZE = len(set(word2id.values())) + 1
EMB_SIZE = 300
PADDING_IDX = len(set(word2id.values()))
OUTPUT_SIZE = 4
OUT_CHANNELS = 100
KERNEL_HEIGHTS = 3
STRIDE = 1
PADDING = 1
# モデルの定義
model = CNN(VOCAB_SIZE, EMB_SIZE, PADDING_IDX, OUTPUT_SIZE, OUT_CHANNELS, KERNEL_HE
IGHTS, STRIDE, PADDING, emb_weights=weights)

# 先頭 10 件の予測値取得
for i in range(10):
  X = dataset_train[i]['inputs']
  print(torch.softmax(model(X.unsqueeze(0)), dim=-1))
```

実行結果：

```
tensor([[0.2294, 0.2677, 0.2247, 0.2783]], grad_fn=<SoftmaxBackward0>)
tensor([[0.2491, 0.2842, 0.2134, 0.2533]], grad_fn=<SoftmaxBackward0>)
tensor([[0.2583, 0.2929, 0.2054, 0.2433]], grad_fn=<SoftmaxBackward0>)
tensor([[0.2540, 0.2935, 0.2142, 0.2384]], grad_fn=<SoftmaxBackward0>)
tensor([[0.2654, 0.2533, 0.2070, 0.2742]], grad_fn=<SoftmaxBackward0>)
tensor([[0.2757, 0.2865, 0.2093, 0.2285]], grad_fn=<SoftmaxBackward0>)
tensor([[0.2342, 0.2627, 0.2349, 0.2682]], grad_fn=<SoftmaxBackward0>)
tensor([[0.2570, 0.2903, 0.1863, 0.2664]], grad_fn=<SoftmaxBackward0>)
tensor([[0.2698, 0.2554, 0.2487, 0.2261]], grad_fn=<SoftmaxBackward0>)
tensor([[0.2802, 0.2755, 0.2017, 0.2426]], grad_fn=<SoftmaxBackward0>)
```

まとめ：

ニューラルネットワークの各ユニットを図に示します:



$$h_{W,b}(x) = f(W^Tx) = f(\textstyle\sum_{i=1}^3 W_i x_i + b)$$

なお、このユニットは Logistic 回帰モデルと呼ばれることもある。複数のセルを組み合わせて階層構造を持つと、ニューラルネットワークモデルが形成される。

$$\begin{aligned}
a_1^{(2)} &= f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}) \\
a_2^{(2)} &= f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}) \\
a_3^{(2)} &= f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}) \\
h_{W,b}(x) = a_1^{(3)} &= f(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)})
\end{aligned}$$

2、3、4、5、…個の隠れ層があるまで広げることができます。
ニューラルネットワークの訓練方法も Logistic と類似しているが、その多層性のため、チェーン式の導波法則を利用して隠れた層のノードを導波する必要がある。すなわち、勾配降下+チェーン式の導波法則であり、専門名は逆方向伝播である

8 7 .確率的勾配降下法による CNN の学習

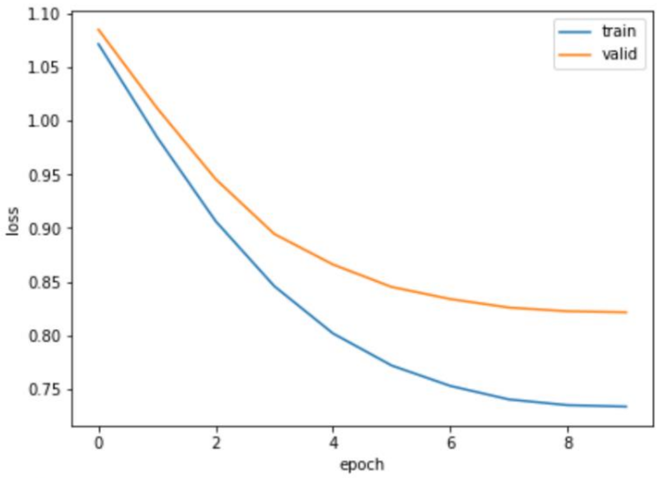
解答：

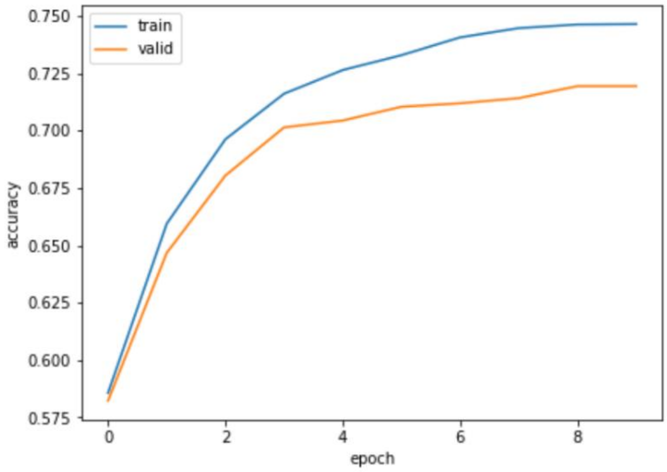
```
# パラメータの設定
VOCAB_SIZE = len(set(word2id.values())) + 1
EMB_SIZE = 300
PADDING_IDX = len(set(word2id.values()))
OUTPUT_SIZE = 4
OUT_CHANNELS = 100
KERNEL_HEIGHTS = 3
STRIDE = 1
PADDING = 1
LEARNING_RATE = 5e-2
BATCH_SIZE = 64
NUM_EPOCHS = 10
```

```
# モデルの定義
model = CNN(VOCAB_SIZE, EMB_SIZE, PADDING_IDX, OUTPUT_SIZE, OUT_CHANNELS, KERNEL_HE
IGHTS, STRIDE, PADDING, emb_weights=weights)

# 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=LEARNING_RATE)

# デバイスの指定
device = torch.cuda.set_device(0)

# モデルの学習
log = train_model(dataset_train, dataset_valid, BATCH_SIZE, model, criterion, optim
izer, NUM_EPOCHS, collate_fn=Padsequence(PADDING_IDX), device=device)

# ログの可視化
visualize_logs(log)

# 正解率の算出
_, acc_train = calculate_loss_and_accuracy(model, dataset_train, device)
_, acc_test = calculate_loss_and_accuracy(model, dataset_test, device)
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：

```
epoch: 1, loss_train: 1.0717, accuracy_train: 0.5857, loss_valid: 1.0853, accuracy_valid: 0.5823, 8.5524sec
epoch: 2, loss_train: 0.9849, accuracy_train: 0.6594, loss_valid: 1.0120, accuracy_valid: 0.6467, 8.5199sec
epoch: 3, loss_train: 0.9064, accuracy_train: 0.6962, loss_valid: 0.9457, accuracy_valid: 0.6804, 8.3986sec
epoch: 4, loss_train: 0.8458, accuracy_train: 0.7160, loss_valid: 0.8945, accuracy_valid: 0.7013, 8.4950sec
epoch: 5, loss_train: 0.8018, accuracy_train: 0.7263, loss_valid: 0.8661, accuracy_valid: 0.7043, 8.6055sec
epoch: 6, loss_train: 0.7719, accuracy_train: 0.7328, loss_valid: 0.8451, accuracy_valid: 0.7103, 8.6976sec
epoch: 7, loss_train: 0.7528, accuracy_train: 0.7405, loss_valid: 0.8339, accuracy_valid: 0.7118, 8.6223sec
epoch: 8, loss_train: 0.7402, accuracy_train: 0.7446, loss_valid: 0.8259, accuracy_valid: 0.7141, 8.5499sec
epoch: 9, loss_train: 0.7349, accuracy_train: 0.7462, loss_valid: 0.8225, accuracy_valid: 0.7193, 8.6445sec
epoch: 10, loss_train: 0.7335, accuracy_train: 0.7463, loss_valid: 0.8215, accuracy_valid: 0.7193, 8.5127sec
```






```
正解率（学習データ）：0.746
正解率（評価データ）：0.721
```

88.パラメータチューニング

解答：

```
from torch.nn import functional as F

class textCNN(nn.Module):
  def __init__(self, vocab_size, emb_size, padding_idx, output_size, out_channels,
conv_params, drop_rate, emb_weights=None):
    super().__init__()
    if emb_weights != None:  # 指定があれば埋め込み層の重みを emb_weights で初期化
      self.emb = nn.Embedding.from_pretrained(emb_weights, padding_idx=padding_idx)
    else:
      self.emb = nn.Embedding(vocab_size, emb_size, padding_idx=padding_idx)
    self.convs = nn.ModuleList([nn.Conv2d(1, out_channels, (kernel_height, emb_size
), padding=(padding, 0)) for kernel_height, padding in conv_params])
    self.drop = nn.Dropout(drop_rate)
    self.fc = nn.Linear(len(conv_params) * out_channels, output_size)

  def forward(self, x):
    # x.size() = (batch_size, seq_len)
    emb = self.emb(x).unsqueeze(1)
    # emb.size() = (batch_size, 1, seq_len, emb_size)
    conv = [F.relu(conv(emb)).squeeze(3) for i, conv in enumerate(self.convs)]
    # conv[i].size() = (batch_size, out_channels, seq_len + padding * 2 - kernel_he
ight + 1)
    max_pool = [F.max_pool1d(i, i.size(2)) for i in conv]
    # max_pool[i].size() = (batch_size, out_channels, 1) -> seq_len 方向に最大値を取得
    max_pool_cat = torch.cat(max_pool, 1)
    # max_pool_cat.size() = (batch_size, len(conv_params) * out_channels, 1)  -> フ
ィルター別の結果を結合
    out = self.fc(self.drop(max_pool_cat.squeeze(2)))
    # out.size() = (batch_size, output_size)
    return out

import optuna
def objective(trial):
    # チューニング対象パラメータのセット
  emb_size = int(trial.suggest_discrete_uniform('emb_size', 100, 300, 100))
  out_channels = int(trial.suggest_discrete_uniform('out_channels', 50, 200, 50))
  drop_rate = trial.suggest_discrete_uniform('drop_rate', 0.0, 0.5, 0.1)
  learning_rate = trial.suggest_loguniform('learning_rate', 5e-4, 5e-2)
  momentum = trial.suggest_discrete_uniform('momentum', 0.5, 0.9, 0.1)
  batch_size = int(trial.suggest_discrete_uniform('batch_size', 16, 128, 16))

  # 固定パラメータの設定
  VOCAB_SIZE = len(set(word2id.values())) + 1
  PADDING_IDX = len(set(word2id.values()))
  OUTPUT_SIZE = 4
```

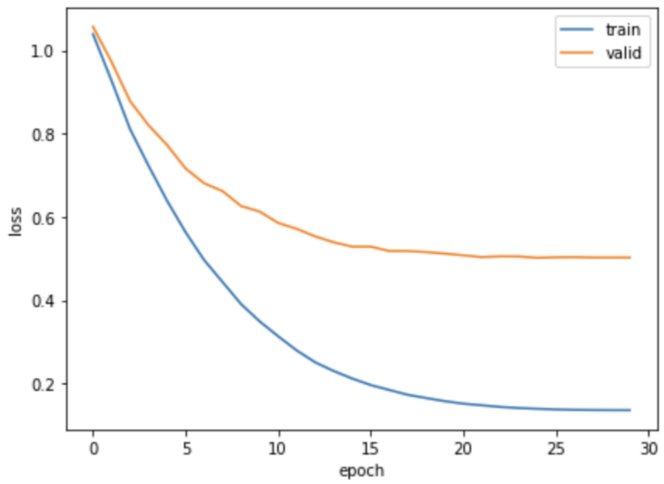
```
  CONV_PARAMS = [[2, 0], [3, 1], [4, 2]]
  NUM_EPOCHS = 30

  # モデルの定義
  model = textCNN(VOCAB_SIZE, emb_size, PADDING_IDX, OUTPUT_SIZE, out_channels, CON
V_PARAMS, drop_rate, emb_weights=weights)

  # 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=learning_rate, momentum=moment
um)

  # デバイスの指定
  device = 'cuda' if torch.cuda.is_available() else 'cpu'

  # モデルの学習
  log = train_model(dataset_train, dataset_valid, batch_size, model, criterion, opt
imizer, NUM_EPOCHS, collate_fn=Padsequence(PADDING_IDX), device=device)

  # 損失の算出
  loss_valid, _ = calculate_loss_and_accuracy(model, dataset_valid, device, criteri
on=criterion)

  return loss_valid

study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=1)

print(study.best_params)
print(study.best_value)

print('Best trial:')
trial = study.best_trial
print('  Value: {:.3f}'.format(trial.value))
print('  Params: ')
for key, value in trial.params.items():
  print('    {}: {}'.format(key, value))

# パラメータの設定
VOCAB_SIZE = len(set(word2id.values())) + 1
EMB_SIZE = int(trial.params['emb_size'])
PADDING_IDX = len(set(word2id.values()))
OUTPUT_SIZE = 4
OUT_CHANNELS = int(trial.params['out_channels'])
CONV_PARAMS = [[2, 0], [3, 1], [4, 2]]
DROP_RATE = trial.params['drop_rate']
LEARNING_RATE = trial.params['learning_rate']
```

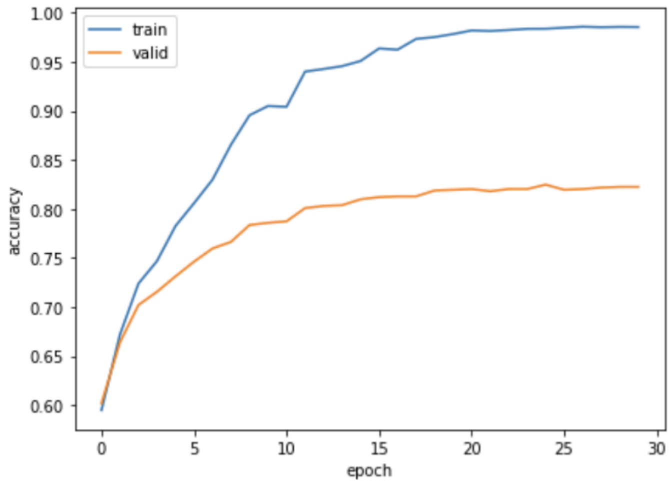
```python
BATCH_SIZE = int(trial.params['batch_size'])
NUM_EPOCHS = 30

# モデルの定義
model = textCNN(VOCAB_SIZE, EMB_SIZE, PADDING_IDX, OUTPUT_SIZE, OUT_CHANNELS, CONV_PARAMS, DROP_RATE, emb_weights=weights)
print(model)

# 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=LEARNING_RATE, momentum=0.9)

# デバイスの指定
device = 'cuda' if torch.cuda.is_available() else 'cpu'

# モデルの学習
log = train_model(dataset_train, dataset_valid, BATCH_SIZE, model, criterion, optimizer, NUM_EPOCHS, collate_fn=Padsequence(PADDING_IDX), device=device)

# ログの可視化
visualize_logs(log)

# 正解率の算出
_, acc_train = calculate_loss_and_accuracy(model, dataset_train, device)
_, acc_test = calculate_loss_and_accuracy(model, dataset_test, device)
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 ：

```
epoch: 1, loss_train: 1.1113, accuracy_train: 0.5204, loss_valid: 1.1139, accuracy_valid: 0.5314, 32.0435sec
epoch: 2, loss_train: 1.0609, accuracy_train: 0.5618, loss_valid: 1.0742, accuracy_valid: 0.5614, 30.3755sec
epoch: 3, loss_train: 1.0216, accuracy_train: 0.6107, loss_valid: 1.0420, accuracy_valid: 0.6003, 30.1211sec
epoch: 4, loss_train: 0.9835, accuracy_train: 0.6263, loss_valid: 1.0116, accuracy_valid: 0.6310, 30.7844sec
epoch: 5, loss_train: 0.9439, accuracy_train: 0.6725, loss_valid: 0.9805, accuracy_valid: 0.6490, 31.3859sec
epoch: 6, loss_train: 0.9051, accuracy_train: 0.6934, loss_valid: 0.9483, accuracy_valid: 0.6684, 30.0443sec
epoch: 7, loss_train: 0.8689, accuracy_train: 0.7050, loss_valid: 0.9208, accuracy_valid: 0.6781, 30.4202sec
epoch: 8, loss_train: 0.8358, accuracy_train: 0.7133, loss_valid: 0.8921, accuracy_valid: 0.6886, 31.5759sec
epoch: 9, loss_train: 0.8059, accuracy_train: 0.7243, loss_valid: 0.8697, accuracy_valid: 0.7013, 30.3970sec
epoch: 10, loss_train: 0.7781, accuracy_train: 0.7335, loss_valid: 0.8511, accuracy_valid: 0.7103, 30.5039sec
epoch: 11, loss_train: 0.7549, accuracy_train: 0.7394, loss_valid: 0.8360, accuracy_valid: 0.7133, 29.9979sec
epoch: 12, loss_train: 0.7323, accuracy_train: 0.7490, loss_valid: 0.8216, accuracy_valid: 0.7178, 30.3283sec
epoch: 13, loss_train: 0.7119, accuracy_train: 0.7545, loss_valid: 0.8089, accuracy_valid: 0.7186, 30.0871sec
epoch: 14, loss_train: 0.6940, accuracy_train: 0.7580, loss_valid: 0.7975, accuracy_valid: 0.7156, 31.5772sec
epoch: 15, loss_train: 0.6770, accuracy_train: 0.7649, loss_valid: 0.7858, accuracy_valid: 0.7283, 30.7176sec
epoch: 16, loss_train: 0.6630, accuracy_train: 0.7686, loss_valid: 0.7754, accuracy_valid: 0.7320, 30.6278sec
epoch: 17, loss_train: 0.6498, accuracy_train: 0.7726, loss_valid: 0.7685, accuracy_valid: 0.7283, 30.8282sec
epoch: 18, loss_train: 0.6388, accuracy_train: 0.7761, loss_valid: 0.7619, accuracy_valid: 0.7313, 30.4881sec
epoch: 19, loss_train: 0.6290, accuracy_train: 0.7802, loss_valid: 0.7563, accuracy_valid: 0.7335, 30.0214sec
epoch: 20, loss_train: 0.6207, accuracy_train: 0.7821, loss_valid: 0.7515, accuracy_valid: 0.7328, 30.3044sec
epoch: 21, loss_train: 0.6138, accuracy_train: 0.7855, loss_valid: 0.7485, accuracy_valid: 0.7343, 31.9310sec
epoch: 22, loss_train: 0.6083, accuracy_train: 0.7860, loss_valid: 0.7447, accuracy_valid: 0.7365, 30.4349sec
epoch: 23, loss_train: 0.6037, accuracy_train: 0.7881, loss_valid: 0.7415, accuracy_valid: 0.7358, 30.7053sec
epoch: 24, loss_train: 0.6002, accuracy_train: 0.7908, loss_valid: 0.7400, accuracy_valid: 0.7365, 30.4327sec
epoch: 25, loss_train: 0.5976, accuracy_train: 0.7913, loss_valid: 0.7385, accuracy_valid: 0.7380, 30.7227sec
epoch: 26, loss_train: 0.5957, accuracy_train: 0.7915, loss_valid: 0.7376, accuracy_valid: 0.7380, 30.4236sec
epoch: 27, loss_train: 0.5945, accuracy_train: 0.7920, loss_valid: 0.7369, accuracy_valid: 0.7388, 31.1956sec
epoch: 28, loss_train: 0.5938, accuracy_train: 0.7924, loss_valid: 0.7364, accuracy_valid: 0.7365, 30.3217sec
epoch: 29, loss_train: 0.5935, accuracy_train: 0.7924, loss_valid: 0.7361, accuracy_valid: 0.7373, 30.1466sec
epoch: 30, loss_train: 0.5934, accuracy_train: 0.7924, loss_valid: 0.7361, accuracy_valid: 0.7365, 30.1529sec
```

```
Best trial:
  Value: 0.736
  Params:
    emb_size: 300.0
    out_channels: 150.0
    drop_rate: 0.2
    learning_rate: 0.0067812240297429
    momentum: 0.7
    batch_size: 64.0
```






```
正解率（学習データ）：0.986
正解率（評価データ）：0.844
```

まとめ：

（1）学習率

学習率（learning rate または lr）とは、最適化アルゴリズムにおけるネットワーク重みの更新の幅の大きさを指す。学習率は一定であってもよく、徐々に低下してもよく、運動量に基づくものであってもよく、適応的であってもよい。異なる最適化アルゴリズムが異なる学習率を決定する。学習率が大きすぎるとモデルが収束せず、損失 loss が上下に揺れ続ける可能性がある、学習率が小さすぎるとモデルの収束速度が遅くなり、より長い時間の訓練が必要になります。通常、lr は［0.01、0.001、0.0001］の値をとる

（2）batch_size

batch_size は訓練ニューラルネットワークごとにモデルに送られるサンプル数であり、畳み込みニューラルネットワークでは、多くのバッチがネットワークをより速く収束させることができるが、メモリリソースの制限により、バッチが大きすぎるとメモリが不足したり、プログラムカーネルが崩壊したりする可能性がある。bath_size は通常、[16、32、64128]の値をとる

（3）optimizer

現在 Adam は急速に収束し、よく使用される最適化器である。ランダム勾配降下（SGD）は収束が遅いが、運動量 Momentum を加えることで収束が加速し、同時に運動量のランダム勾配降下アルゴリズムはモデルが収束するとより高い精度が得られるという最適解を持っている。通常は速度を求めるなら Adam の方が多い。

（4）反復回数

反復回数とは、トレーニングセット全体がニューラルネットワークに入力されてトレーニングを行う回数であり、テストエラー率とトレーニングエラー率の差が小さい場合、現在の反復回数が適切であると考えることができる、テストエラーが最初に小さくなって大きくなった場合は、反復回数が大きすぎて、反復回数を小さくする必要があります。そうしないと、オーバーフィットが発生しやすくなります。

8 9. 事前学習済み言語モデルからの転移学習

解答：

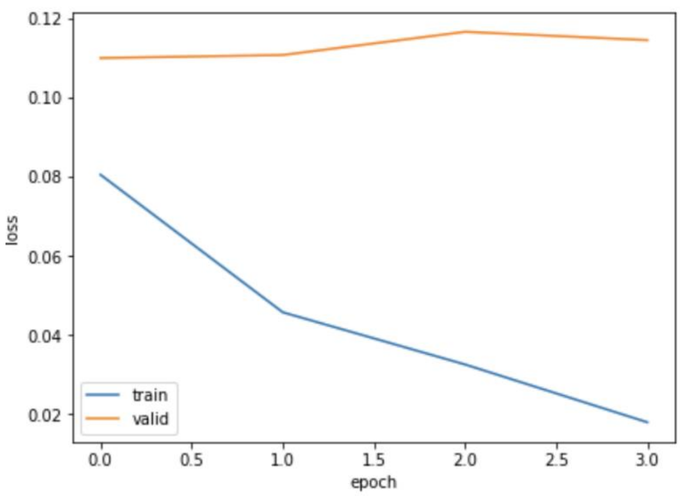
```
# BERT 分類モデルの定義
class BERTClass(torch.nn.Module):
  def __init__(self, drop_rate, otuput_size):
    super().__init__()
    self.bert = BertModel.from_pretrained('bert-base-uncased')
    self.drop = torch.nn.Dropout(drop_rate)
    self.fc = torch.nn.Linear(768, otuput_size)

  def forward(self, ids, mask):
    _, out = self.bert(ids, attention_mask=mask, return_dict=False)
    out = self.fc(self.drop(out))
    return out

def calculate_loss_and_accuracy(model, criterion, loader, device):
  """ 損失・正解率を計算"""
  model.eval()
  loss = 0.0
  total = 0
  correct = 0
  with torch.no_grad():
    for data in loader:
      # デバイスの指定
      ids = data['ids'].to(device)
      mask = data['mask'].to(device)
      labels = data['labels'].to(device)

      # 順伝播
      outputs = model(ids, mask)

      # 損失計算
      loss += criterion(outputs, labels).item()

      # 正解率計算
      pred = torch.argmax(outputs, dim=-1).cpu().numpy() # バッチサイズの長さの予測ラベル配列
      labels = torch.argmax(labels, dim=-1).cpu().numpy()  # バッチサイズの長さの正解ラベル配列
      total += len(labels)
      correct += (pred == labels).sum().item()

  return loss / len(loader), correct / total
```

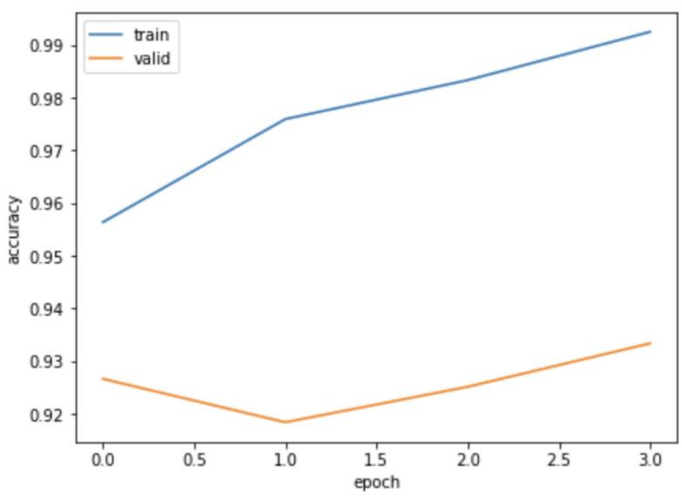
```
def train_model(dataset_train, dataset_valid, batch_size, model, criterion, optimiz
er, num_epochs, device=None):
  """モデルの学習を実行し、損失・正解率のログを返す"""
  # デバイスの指定
  model.to(device)

  # dataloaderの作成
  dataloader_train = DataLoader(dataset_train, batch_size=batch_size, shuffle=True)
  dataloader_valid = DataLoader(dataset_valid, batch_size=len(dataset_valid), shuff
le=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 data in dataloader_train:
      # デバイスの指定
      ids = data['ids'].to(device)
      mask = data['mask'].to(device)
      labels = data['labels'].to(device)

      # 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.forward(ids, mask)
      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, criterion, dataloade
r_train, device)
    loss_valid, acc_valid = calculate_loss_and_accuracy(model, criterion, dataloade
r_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_dict(), 'optimizer_
state_dict': optimizer.state_dict()}, f'checkpoint{epoch + 1}.pt')

    # 終了時刻の記録
```

```python
    e_time = time.time()

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

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

# パラメータの設定
DROP_RATE = 0.4
OUTPUT_SIZE = 4
BATCH_SIZE = 32
NUM_EPOCHS = 4
LEARNING_RATE = 2e-5

# モデルの定義
model = BERTClass(DROP_RATE, OUTPUT_SIZE)

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

# オプティマイザの定義
optimizer = torch.optim.AdamW(params=model.parameters(), lr=LEARNING_RATE)

# デバイスの指定
device = 'cuda' if cuda.is_available() else 'cpu'

# モデルの学習
log = train_model(dataset_train, dataset_valid, BATCH_SIZE, model, criterion, optimizer, NUM_EPOCHS, device=device)

# ログの可視化
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()

# 正解率の算出
def calculate_accuracy(model, dataset, device):
  # Dataloaderの作成
```

```
  loader = DataLoader(dataset, batch_size=len(dataset), shuffle=False)

  model.eval()
  total = 0
  correct = 0
  with torch.no_grad():
    for data in loader:
      # デバイスの指定
      ids = data['ids'].to(device)
      mask = data['mask'].to(device)
      labels = data['labels'].to(device)

      # 順伝播 + 予測値の取得 + 正解数のカウント
      outputs = model.forward(ids, mask)
      pred = torch.argmax(outputs, dim=-1).cpu().numpy()
      labels = torch.argmax(labels, dim=-1).cpu().numpy()
      total += len(labels)
      correct += (pred == labels).sum().item()

  return correct / total

print(f'正解率（学習データ）：{calculate_accuracy(model, dataset_train, device):.3f}')
print(f'正解率（検証データ）：{calculate_accuracy(model, dataset_valid, device):.3f}')
print(f'正解率（評価データ）：{calculate_accuracy(model, dataset_test, device):.3f}')
```

実行結果 ：





正解率（学習データ）: 0.993
正解率（検証データ）: 0.933
正解率（評価データ）: 0.946