Open in Colab

```
###from google.colab import drive
###drive.mount("/content/drive")
```

```
Mounted at /content/drive
```

```
###%cd /content/drive/MyDrive/hw2
```

```
/content/drive/MyDrive/hw2
```

Homework 2-1 Phoneme Classification

- Slides: https://speech.ee.ntu.edu.tw/~hylee/ml/ml2021-course-data/hw/HW02/HW02.pdf
- Video (Chinese): https://youtu.be/PdjXnQbu2zo
- Video (English): https://youtu.be/ESRr-VCykBs

The DARPA TIMIT Acoustic-Phonetic Continuous Speech Corpus (TIMIT)

The TIMIT corpus of reading speech has been designed to provide speech data for the acquisition of acoustic-phonetic knowledge and for the development and evaluation of automatic speech recognition systems.

This homework is a multiclass classification task, we are going to train a deep neural network classifier to predict the phonemes for each frame from the speech corpus TIMIT.

link: https://academictorrents.com/details/34e2b78745138186976cbc27939b1b34d18bd5b3

Download Data

Download data from google drive, then unzip it.

You should have `timit_11/train_11.npy`, `timit_11/train_label_11.npy`, and `timit_11/test_11.npy` after running this block.

`timit_11/`

- `train_11.npy`: training data
- `train_label_11.npy`: training label
- `test_11.npy`: testing data

notes: if the google drive link is dead, you can download the data directly from Kaggle and upload it to the workspace

```
!gdown --id '1HPkcmQmFGu-30knddKIa5dNDsR05lIQR' --output data.zip
!unzip data.zip
!ls
```

```
/usr/local/lib/python3.10/dist-packages/gdown/cli.py:138: FutureWarning: Option `--id` was deprecated in version 4.3.1 and will be removed in 5
  warnings.warn(
Downloading...
From (original): https://drive.google.com/uc?id=1HPkcmQmFGu-30knddKIa5dNDsR05lIQR
From (redirected): https://drive.google.com/uc?id=1HPkcmQmFGu-30knddKIa5dNDsR05lIQR&confirm=t&uuid=9d6869e8-1be8-48bf-87be-4f96773fe5d1
To: /content/data.zip
100% 372M/372M [00:06<00:00, 55.7MB/s]
Archive:  data.zip
   creating: timit_11/
  inflating: timit_11/train_11.npy
  inflating: timit_11/test_11.npy
  inflating: timit_11/train_label_11.npy
data.zip  drive  sample_data  timit_11
```

Preparing Data

Load the training and testing data from the `.npy` file (NumPy array).

```python
import numpy as np

print('Loading data ...')

data_root='./timit_11/'
train = np.load(data_root + 'train_11.npy')
train_label = np.load(data_root + 'train_label_11.npy')
test = np.load(data_root + 'test_11.npy')

print('Size of training data: {}'.format(train.shape))
print('Size of testing data: {}'.format(test.shape))
print('Size of training data label: {}'.format(train_label.shape))
```

```
Loading data ...
Size of training data: (1229932, 429)
Size of testing data: (451552, 429)
Size of training data label: (1229932,)
```

Create Dataset

```
import torch
from torch.utils.data import Dataset

class TIMITDataset(Dataset):
    def __init__(self, X, y=None):
        self.data = torch.from_numpy(X).float()
        if y is not None:
            y = y.astype(np.int64)
            self.label = torch.LongTensor(y)
        else:
            self.label = None

    def __getitem__(self, idx):
        if self.label is not None:
            return self.data[idx], self.label[idx]
        else:
            return self.data[idx]

    def __len__(self):
        return len(self.data)
```

Split the labeled data into a training set and a validation set, you can modify the variable `VAL_RATIO` to change the ratio of validation data.

```
VAL_RATIO = 0.05

percent = int(train.shape[0] * (1 - VAL_RATIO))
train_x, train_y, val_x, val_y = train[:percent], train_label[:percent], train[percent:], train_label[percent:]
print('Size of training set: {}'.format(train_x.shape))
print('Size of validation set: {}'.format(val_x.shape))
```

```
Size of training set: (1168435, 429)
Size of validation set: (61497, 429)
```

Create a data loader from the dataset, feel free to tweak the variable `BATCH_SIZE` here.

```
BATCH_SIZE = 2048

from torch.utils.data import DataLoader

train_set = TIMITDataset(train_x, train_y)
val_set = TIMITDataset(val_x, val_y)
train_loader = DataLoader(train_set, batch_size=BATCH_SIZE, shuffle=True) #only shuffle the training data
val_loader = DataLoader(val_set, batch_size=BATCH_SIZE, shuffle=False)
```

Cleanup the unneeded variables to save memory.

notes: if you need to use these variables later, then you may remove this block or clean up unneeded variables later the data size is quite huge, so be aware of memory usage in colab

```
import gc

del train, train_label, train_x, train_y, val_x, val_y
gc.collect()
```

```
293
```

Create Model

Define model architecture, you are encouraged to change and experiment with the model architecture.

```
import torch
import torch.nn as nn
"""
class Classifier(nn.Module):
    def __init__(self):
        super(Classifier, self).__init__()
        self.layer1 = nn.Linear(429, 1024)
        self.layer2 = nn.Linear(1024, 512)
        self.layer3 = nn.Linear(512, 128)
        self.out = nn.Linear(128, 39)

        self.bn1=nn.BatchNorm1d(1024)
        self.bn2=nn.BatchNorm1d(512)
        self.bn3=nn.BatchNorm1d(128)


        self.act_fn = nn.ReLU()
        self.dropout= nn.Dropout(0.5)

    def forward(self, x):
        x=self.layer1(x)
        x=self.bn1(x)
        x=self.act_fn(x)
        x=self.dropout(x)

        x=self.layer2(x)
        x=self.bn2(x)
        x=self.act_fn(x)
        x=self.dropout(x)

        x=self.layer3(x)
        x=self.bn3(x)
        x=self.act_fn(x)
        x=self.dropout(x)

        x = self.out(x)

        return x
"""
"""
class Classifier(nn.Module):
    def __init__(self):
```

```
        super(Classifier, self).__init__()
        self.layer1 = nn.Linear(429, 1024)
        self.layer2 = nn.Linear(1024, 2048)
        self.layer3 = nn.Linear(2048, 4096)
        self.layer4 = nn.Linear(4096, 2048)
        self.layer5 = nn.Linear(2048, 1024)
        self.layer6 = nn.Linear(1024, 512)
        self.out = nn.Linear(512, 39)

        self.bn1=nn.BatchNorm1d(1024)
        self.bn2=nn.BatchNorm1d(2048)
        self.bn3=nn.BatchNorm1d(4096)
        self.bn4=nn.BatchNorm1d(2048)
        self.bn5=nn.BatchNorm1d(1024)
        self.bn6=nn.BatchNorm1d(512)

        self.act_fn = nn.ReLU()
        self.dropout= nn.Dropout(0.4)

    def forward(self, x):
        x=self.layer1(x)
        x=self.bn1(x)
        x=self.act_fn(x)
        x=self.dropout(x)

        x=self.layer2(x)
        x=self.bn2(x)
        x=self.act_fn(x)
        x=self.dropout(x)

        x=self.layer3(x)
        x=self.bn3(x)
        x=self.act_fn(x)
        x=self.dropout(x)

        x=self.layer4(x)
        x=self.bn4(x)
        x=self.act_fn(x)
        x=self.dropout(x)

        x=self.layer5(x)
        x=self.bn5(x)
```

```
        x=self.act_fn(x)
        x=self.dropout(x)

        x=self.layer6(x)
        x=self.bn6(x)
        x=self.act_fn(x)
        x=self.dropout(x)

        x = self.out(x)

        return x """

class Classifier(nn.Module):
    def __init__(self):
        super(Classifier, self).__init__()
        self.layer1 = nn.Linear(429, 2048)
        self.layer2 = nn.Linear(2048, 2048)
        self.layer3 = nn.Linear(2048, 4096)
        self.layer4 = nn.Linear(4096, 2048)
        self.layer5 = nn.Linear(2048, 1024)
        self.layer6 = nn.Linear(1024, 512)
        self.out = nn.Linear(512, 39)

        self.bn1=nn.BatchNorm1d(2048)
        self.bn2=nn.BatchNorm1d(2048)
        self.bn3=nn.BatchNorm1d(4096)
        self.bn4=nn.BatchNorm1d(2048)
        self.bn5=nn.BatchNorm1d(1024)
        self.bn6=nn.BatchNorm1d(512)

        self.act_fn = nn.ReLU()
        self.dropout= nn.Dropout(0.4)

    def forward(self, x):
        x=self.layer1(x)
        x=self.bn1(x)
        x=self.act_fn(x)
        x=self.dropout(x)

        x=self.layer2(x)
        x=self.bn2(x)
        x=self.act_fn(x)
```

```
        x=self.dropout(x)

        x=self.layer3(x)
        x=self.bn3(x)
        x=self.act_fn(x)
        x=self.dropout(x)

        x=self.layer4(x)
        x=self.bn4(x)
        x=self.act_fn(x)
        x=self.dropout(x)

        x=self.layer5(x)
        x=self.bn5(x)
        x=self.act_fn(x)
        x=self.dropout(x)

        x=self.layer6(x)
        x=self.bn6(x)
        x=self.act_fn(x)
        x=self.dropout(x)

        x = self.out(x)

        return x
```

Training

```
#check device
def get_device():
    return 'cuda' if torch.cuda.is_available() else 'cpu'
```

Fix random seeds for reproducibility.

```
# fix random seed
def same_seeds(seed):
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)
    np.random.seed(seed)
    torch.backends.cudnn.benchmark = False
    torch.backends.cudnn.deterministic = True
```

Feel free to change the training parameters here.

```
# fix random seed for reproducibility
same_seeds(0)

# get device
device = get_device()
print(f'DEVICE: {device}')

# training parameters
num_epoch = 100                              # number of training epoch
learning_rate = 0.0001              # learning rate
l2_weight_decay = 1e-4            # l2 regularization
weight_decay_l1 = 0.0001                   # l1 regularization
weight_decay_l2 = 0.001            # l2 regularization

# the path where checkpoint saved
model_path = './model.ckpt'

# create model, define a loss function, and optimizer
model = Classifier().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate,weight_decay=l2_weight_decay)

def cal_regularization(model, weight_decay_l1, weight_decay_l2):
        l1 = 0
        l2 = 0
        for i in model.parameters():
                l1 += torch.sum(abs(i))
                l2 += torch.sum(torch.pow(i, 2))
        return weight_decay_l1 * l1 + weight_decay_l2 * l2
```

```
DEVICE: cuda
```

```
# start training

best_acc = 0.0
for epoch in range(num_epoch):
        train_acc = 0.0
        train_loss = 0.0
        val_acc = 0.0
        val_loss = 0.0
```

```
# training
model.train() # set the model to training mode
for i, data in enumerate(train_loader):
    inputs, labels = data
    inputs, labels = inputs.to(device), labels.to(device)
    optimizer.zero_grad()
    outputs = model(inputs)
    batch_loss = criterion(outputs, labels)
    _, train_pred = torch.max(outputs, 1) # get the index of the class with the highest probability
    #batch_loss.backward()
    (batch_loss + cal_regularization(model, weight_decay_l1, weight_decay_l2)).backward()
    optimizer.step()

    train_acc += (train_pred.cpu() == labels.cpu()).sum().item()
    train_loss += batch_loss.item()

# validation
if len(val_set) > 0:
    model.eval() # set the model to evaluation mode
    with torch.no_grad():
        for i, data in enumerate(val_loader):
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = model(inputs)
            batch_loss = criterion(outputs, labels)
            _, val_pred = torch.max(outputs, 1)

            val_acc += (val_pred.cpu() == labels.cpu()).sum().item() # get the index of the class with the highest
            val_loss += batch_loss.item()

        print('[{:03d}/{:03d}] Train Acc: {:3.6f} Loss: {:3.6f} | Val Acc: {:3.6f} loss: {:3.6f}'.format(
            epoch + 1, num_epoch, train_acc/len(train_set), train_loss/len(train_loader), val_acc/len(val_set), val_loss/l
        ))

        # if the model improves, save a checkpoint at this epoch
        if val_acc > best_acc:
            best_acc = val_acc
            torch.save(model.state_dict(), model_path)
            print('saving model with acc {:.3f}'.format(best_acc/len(val_set)))
else:
    print('[{:03d}/{:03d}] Train Acc: {:3.6f} Loss: {:3.6f}'.format(
        epoch + 1, num_epoch, train_acc/len(train_set), train_loss/len(train_loader)
    ))
```

```
                ''

# if not validating, save the last epoch
if len(val_set) == 0:
        torch.save(model.state_dict(), model_path)
        print('saving model at last epoch')
```

```
[001/100] Train Acc: 0.502188 Loss: 1.733399 | Val Acc: 0.625120 loss: 1.223793
saving model with acc 0.625
[002/100] Train Acc: 0.598171 Loss: 1.325369 | Val Acc: 0.646487 loss: 1.158968
saving model with acc 0.646
[003/100] Train Acc: 0.613821 Loss: 1.270208 | Val Acc: 0.657056 loss: 1.111435
saving model with acc 0.657
[004/100] Train Acc: 0.621990 Loss: 1.243764 | Val Acc: 0.649300 loss: 1.118785
[005/100] Train Acc: 0.626294 Loss: 1.229161 | Val Acc: 0.660162 loss: 1.090466
saving model with acc 0.660
[006/100] Train Acc: 0.629771 Loss: 1.218003 | Val Acc: 0.659252 loss: 1.098082
[007/100] Train Acc: 0.632809 Loss: 1.209947 | Val Acc: 0.672732 loss: 1.071693
```