Multi_reverse

July 18, 2023

[17]:
```
import numpy as np
import torch
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
s = {
    'problem'            : "regression",
    'approach'           : "few-shot learning",
    'method'             : "non-parametric",
    'algorithm'          : "siamese network",
    'goal'               : "learn a distribution using few samples from it",
    'input'              : "samples from a distribution",
    'input type'         : "vectors",
    'input meaning'      : "spectrum",
    'output'             : "samples from a distribution",
    'output type'        : "one number",
    'output meaning'     : "temperature or pressure, depending on distribution",
    'number of ways'     : 2,
    'number of shots'    : 1,
    'number of folds'    : 8,
    'support-query ratio': 0.8,
    'task size'          : 5,
    'learning rate'      : 1e-4,
    'input dimension'    : 10000,
    'output dimension'   : 1,
    'feature dimension'  : 300,
    'epoch'              : 1000,
    'epoch development'  : 100,
    'data'               : 'temperature_230509_discrete',
    'data P'             : 'pressure_230516_discrete',
    'data T'             : 'temperature_230509_discrete',
    'cross validation round': 16,
    'cross validation round development' : 3,
    'batch size'         : 32,
    'best model folder'  : 'single_T_best_model/'
}
```

[18]:
```
import data_accessor as acc
data_names_list = [
```

```
    'temperature_230509_discrete',
    'pressure_230516_discrete'
    ]
data_dictionary = acc.setup(data_names_list)
```

```
loading temperature_230509_discrete__________________________
        input shape (number, dimension): (6000, 10000)
        label shape (number, dimension): (6000, 1)
        there are 16 folds
        4200 for training, 600 for validating, 1200 for testing
loading pressure_230516_discrete__________________________
        input shape (number, dimension): (5000, 10000)
        label shape (number, dimension): (5000, 1)
        there are 16 folds
        3500 for training, 500 for validating, 1000 for testing
```

[19]:
```
import torch.nn as nn
class SingleTaskNetwork(torch.nn.Module):
    def __init__(self, device, input_dimension, feature_dimension,␣
 ↪output_dimension):
        """ Input: input, anchor, anchor label
        Output: prediction for input"""
        super().__init__()
        self.input_dimension = input_dimension
        self.hidden_dimension = 300
        self.feature_hidden_dimension = 36
        self.feature_dimension = feature_dimension
        self.output_dimension = output_dimension
        self.device = device
        self.feature_sequential = torch.nn.Sequential(
            torch.nn.Linear(self.input_dimension, self.hidden_dimension),
            nn.ReLU(),
            torch.nn.Linear(self.hidden_dimension, self.hidden_dimension),
            nn.ReLU(),
            torch.nn.Linear(self.hidden_dimension, self.feature_dimension)
        )
        self.auxiliary_sequential = torch.nn.Sequential(
            torch.nn.Linear(self.feature_dimension, self.
 ↪feature_hidden_dimension),
            nn.ReLU(),
            torch.nn.Linear(self.feature_hidden_dimension, self.
 ↪feature_hidden_dimension),
            nn.ReLU(),
            torch.nn.Linear(self.feature_hidden_dimension, self.
 ↪output_dimension)
        )
        self.to(device)
```

```python
        self.float()
    def forward(self, input):
        feature_input = self.feature_sequential(input)
        prediction = self.auxiliary_sequential(feature_input)
        return prediction
```

[20]:
```python
from tools import SaveBestModel, PatienceEarlyStopping, Scheduler, plot_loss
class Manager:
    """ DOES: train & evaluate a Siamese network
        """
    def __init__(self, epoch, cross_validation_round):
        self._network = SingleTaskNetwork(device, s['input dimension'],␣
↪s['feature dimension'], s['output dimension'])
        self._network.apply(self.initializer)
        self._learning_rate = s['learning rate']
        self._optimizer = torch.optim.Adam(
            params=self._network.parameters(), lr=self._learning_rate,
            weight_decay=3e-3)
        self._energy = nn.MSELoss()
        self._train_loss = []
        self._valid_loss = []
        self._test_loss = []
        self._epoch = epoch
        self._stopper = PatienceEarlyStopping(patience=5, min_delta=1e-7)
        self._cross_validation_round = cross_validation_round
        self._saver = SaveBestModel(s['best model folder'])
        self._scheduler = Scheduler(optimizer=self._optimizer,
            minimum_learning_rate=1e-6, patience=5, factor=0.5)
    def initializer(self, layer):
        if type(layer) == nn.Linear:
            nn.init.kaiming_normal_(layer.weight) # normal version
    def _step(self, job):
        input, input_label = job
        # print(f"input dtype is {input_1.dtype}")
        prediction = self._network(input)
        loss = self._energy(input_label, prediction)
        return loss
    def train(self, train_dataloader, valid_dataloader):
        """ DOES: calculate loss from tasks
            NOTE: we have a BATCH of tasks here """
        for e in range(self._epoch):
            # print(f"train() epoch {e}")
            batch_train_loss = []
            for _, batch in enumerate(train_dataloader):
                self._optimizer.zero_grad()
                loss = self._step(batch)
                loss.backward()
```

```python
                self._optimizer.step()
                batch_train_loss.append(loss.item())
            self._train_loss.append(np.mean(batch_train_loss))
            batch_valid_loss = []
            with torch.no_grad():
                for _, batch in enumerate(valid_dataloader):
                    loss = self._step(batch)
                    batch_valid_loss.append(loss.item())
            self._valid_loss.append(np.mean(batch_valid_loss))
            # saving, early stopping, scheduler for EACH epoch!
            self._saver(current_loss=np.mean(batch_valid_loss),
                  model=self._network,
                  round=self._cross_validation_round
                  )
            self._scheduler(np.mean(batch_valid_loss))
            self._stopper(np.mean(batch_valid_loss))
            if self._stopper.early_stop == True:
                print(f"EARLY STOPPING @ epoch {e}")
                break
        # summary printout, after we're done with epochs
        print(f"min train loss: {np.min(self._train_loss)}")
        print(f"min valid loss: {np.min(self._valid_loss)}")
        plot_loss(self._train_loss, self._valid_loss)
        return np.min(self._valid_loss)
    def test(self, test_dataloader):
        with torch.no_grad():
            batch_test_loss = []
            for _, batch in enumerate(test_dataloader):
                loss = self._step(batch)
                batch_test_loss.append(loss.item())
            self._test_loss.append(np.mean(batch_test_loss))
        return np.min(self._test_loss)
```

[ ]:

```python
from torch.utils.data import DataLoader
from tools import DefaultDataset, SaveBestCrossValidationModel

CV_saver = SaveBestCrossValidationModel(s['best model folder'])
test_indices = data_dictionary[s['data T']]['test indices']
epoch = s['epoch']
print(f"data: {s['data P']} then {s['data T']}")
cross_validation_loss = []
for cross_validation_round in range(s['cross validation round']):
    if cross_validation_round < s['cross validation round']:
        print(f"CV round {cross_validation_round}_________________________________")
        network_object = Manager(epoch, cross_validation_round)
        print(f"using {s['data P']}")
```

```
        _ = network_object.train(
            DataLoader(DefaultDataset(
            data_dictionary[s['data P']]['data'],
            data_dictionary[s['data P']]['label'],
            data_dictionary[s['data P']]['train␣
↪indices'][cross_validation_round],
            device=device,), shuffle=False, batch_size=s['batch size']),
            DataLoader(DefaultDataset(
            data_dictionary[s['data P']]['data'],
            data_dictionary[s['data P']]['label'],
            data_dictionary[s['data P']]['valid␣
↪indices'][cross_validation_round],
            device=device,), shuffle=False, batch_size=s['batch size']))
        print(f"using {s['data T']}")
        network_object._saver.reset()
        network_object._stopper.reset()
        network_object._train_loss = []
        network_object._valid_loss = []
        print(f"reset: train & valid loss, early stopper, saver")
        valid_loss = network_object.train(
            DataLoader(DefaultDataset(
            data_dictionary[s['data T']]['data'],
            data_dictionary[s['data T']]['label'],
            data_dictionary[s['data T']]['train␣
↪indices'][cross_validation_round],
            device=device,), shuffle=False, batch_size=s['batch size']),
            DataLoader(DefaultDataset(
            data_dictionary[s['data T']]['data'],
            data_dictionary[s['data T']]['label'],
            data_dictionary[s['data T']]['valid␣
↪indices'][cross_validation_round],
            device=device,), shuffle=False, batch_size=s['batch size']))
        CV_saver(current_loss=valid_loss, round=cross_validation_round)
        cross_validation_loss.append(valid_loss)
print()
print(f"\nbest model is: {CV_saver.best_model_name} with {CV_saver.
↪current_best_loss}")
print(f"The aggregate performance is: mean {np.mean(cross_validation_loss)},␣
↪std {np.std(cross_validation_loss)}")
```

[ ]:

```
network_object._network.load_state_dict(torch.load(s['best model folder'] +␣
↪CV_saver.best_model_name))
test_loss = network_object.test(
            DataLoader(DefaultDataset(
            data_dictionary[s['data P']]['data'],
            data_dictionary[s['data P']]['label'],
```

```
            data_dictionary[s['data P']]['test indices'],
            device=device,), shuffle=False, batch_size=s['batch size']))
print(f"testing loss: for {s['data P']}: {test_loss}")
test_loss = network_object.test(
            DataLoader(DefaultDataset(
            data_dictionary[s['data T']]['data'],
            data_dictionary[s['data T']]['label'],
            data_dictionary[s['data T']]['test indices'],
            device=device,), shuffle=False, batch_size=s['batch size']))
print(f"testing loss: for {s['data T']}: {test_loss}")
```

```
testing loss: for temperature_230509_discrete: 280.7644894248561
testing loss: for pressure_230516_discrete: 0.0003345543432260456
```