HW3_RNN

October 3, 2025

1 HW3 Recurent Neural Network

1.1 Overview

In this homework, you will build a bi-directional RNN on diagnosis codes. The recurrent nature of RNN allows us to model the temporal relation of different visits of a patient. More specifically, we will still perform **Heart Failure Prediction**, but with different input formats.

---

[1]:
```
import os
import sys
import pickle
import random
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F

# set seed
seed = 24
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)
os.environ["PYTHONHASHSEED"] = str(seed)


# Define data path
DATA_PATH = "../HW3_RNN-lib/data"
```

1.2 About Raw Data

To get started, we will implement a naive RNN model for heart failure prediction using the diagnosis codes.

We will use the same dataset synthesized from MIMIC-III, but with different input formats.

The data has been preprocessed for you. Let us load them and take a look.

[2]:
```python
pids = pickle.load(open(os.path.join(DATA_PATH,'train/pids.pkl'), 'rb'))
vids = pickle.load(open(os.path.join(DATA_PATH,'train/vids.pkl'), 'rb'))
hfs = pickle.load(open(os.path.join(DATA_PATH,'train/hfs.pkl'), 'rb'))
seqs = pickle.load(open(os.path.join(DATA_PATH,'train/seqs.pkl'), 'rb'))
types = pickle.load(open(os.path.join(DATA_PATH,'train/types.pkl'), 'rb'))
rtypes = pickle.load(open(os.path.join(DATA_PATH,'train/rtypes.pkl'), 'rb'))

assert len(pids) == len(vids) == len(hfs) == len(seqs) == 1000
assert len(types) == 619
```

where

- `pids`: contains the patient ids
- `vids`: contains a list of visit ids for each patient
- `hfs`: contains the heart failure label (0: normal, 1: heart failure) for each patient
- `seqs`: contains a list of visit (in ICD9 codes) for each patient
- `types`: contains the map from ICD9 codes to ICD-9 labels
- `rtypes`: contains the map from ICD9 labels to ICD9 codes

Let us take a patient as an example.

[3]:
```python
# take the 3rd patient as an example

print("Patient ID:", pids[3])
print("Heart Failure:", hfs[3])
print("# of visits:", len(vids[3]))
for visit in range(len(vids[3])):
    print(f"\t{visit}-th visit id:", vids[3][visit])
    print(f"\t{visit}-th visit diagnosis labels:", seqs[3][visit])
    print(f"\t{visit}-th visit diagnosis codes:", [rtypes[label] for label in␣
 ↪seqs[3][visit]])
```

```
Patient ID: 47537
Heart Failure: 0
# of visits: 2
        0-th visit id: 0
        0-th visit diagnosis labels: [12, 103, 262, 285, 290, 292, 359, 416, 39,
225, 275, 294, 326, 267, 93]
        0-th visit diagnosis codes: ['DIAG_041', 'DIAG_276', 'DIAG_518',
'DIAG_560', 'DIAG_567', 'DIAG_569', 'DIAG_707', 'DIAG_785', 'DIAG_155',
'DIAG_456', 'DIAG_537', 'DIAG_571', 'DIAG_608', 'DIAG_529', 'DIAG_263']
        1-th visit id: 1
        1-th visit diagnosis labels: [12, 103, 240, 262, 290, 292, 319, 359,
510, 513, 577, 307, 8, 280, 18, 131]
        1-th visit diagnosis codes: ['DIAG_041', 'DIAG_276', 'DIAG_482',
'DIAG_518', 'DIAG_567', 'DIAG_569', 'DIAG_599', 'DIAG_707', 'DIAG_995',
'DIAG_998', 'DIAG_V09', 'DIAG_584', 'DIAG_031', 'DIAG_553', 'DIAG_070',
'DIAG_305']
```

Note that `seqs` is a list of list of list. That is, `seqs[i][j][k]` gives you the k-th diagnosis codes for the j-th visit for the i-th patient.

And you can look up the meaning of the ICD9 code online. For example, `DIAG_276` represetns *disorders of fluid electrolyte and acid-base balance*.

Further, let see number of heart failure patients.

[4]:
```
print("number of heart failure patients:", sum(hfs))
print("ratio of heart failure patients: %.2f" % (sum(hfs) / len(hfs)))
```

```
number of heart failure patients: 548
ratio of heart failure patients: 0.55
```

[6]:
```
# ========================
# Utilities & Reproducibility
# ========================
import os, random, numpy as np, torch
from torch import nn
from torch.utils.data import Dataset, DataLoader
from sklearn.metrics import precision_score, recall_score, f1_score,␣
↪roc_auc_score

def set_seed(seed: int = 24):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    os.environ["PYTHONHASHSEED"] = str(seed)

set_seed(24)
torch.set_float32_matmul_precision("high") if hasattr(torch,␣
↪"set_float32_matmul_precision") else None
```

Now we have the data. Let us build the naive RNN.

1.3 1 Build the dataset [30 points]

1.3.1 1.1 CustomDataset [5 points]

First, let us implement a custom dataset using PyTorch class `Dataset`, which will characterize the key features of the dataset we want to generate.

We will use the sequences of diagnosis codes `seqs` as input and heart failure `hfs` as output.

[7]:
```
class CustomDataset(Dataset):
    """
    Stores sequences (list of patients; each patient = list of visits; each␣
↪visit = list of code indices)
```

```
    and labels (0/1). Do NOT convert to tensors here; leave that to the␣
↪collate_fn.
    """
    def __init__(self, sequences, labels):
        self.sequences = sequences
        self.labels = labels

    def __len__(self):
        return len(self.sequences)

    def __getitem__(self, idx):
        return self.sequences[idx], self.labels[idx]
```

[8]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

dataset = CustomDataset(seqs, hfs)

assert len(dataset) == 1000
```

1.3.2 1.2 Collate Function [20 points]

As you note that, we do not convert the data to tensor in the built `CustomDataset`. Instead, we will do this using a collate function `collate_fn()`.

This collate function `collate_fn()` will be called by `DataLoader` after fetching a list of samples using the indices from `CustomDataset` to collate the list of samples into batches.

For example, assume the `DataLoader` gets a list of two samples.

```
[ [ [0, 1, 2], [8, 0] ],
  [ [12, 13, 6, 7], [12], [23, 11] ] ]
```

where the first sample has two visits `[0, 1, 2]` and `[8, 0]` and the second sample has three visits `[12, 13, 6, 7]`, `[12]`, and `[23, 11]`.

The collate function `collate_fn()` is supposed to pad them into the same shape (3, 4), where 3 is the maximum number of visits and 4 is the maximum number of diagnosis codes.

```
[ [ [0, 1, 2, *0*], [8, 0, *0*, *0*], [*0*, *0*, *0*, *0*]  ],
  [ [12, 13, 6, 7], [12, *0*, *0*, *0*], [23, 11, *0*, *0*] ] ]
```

Further, the padding information will be stored in a mask with the same shape, where 1 indicates that the diagnosis code at this position is from the original input, and 0 indicates that the diagnosis code at this position is the padded value.

```
[ [ [1, 1, 1, 0], [1, 1, 0, 0], [0, 0, 0, 0] ],
  [ [1, 1, 1, 1], [1, 0, 0, 0], [1, 1, 0, 0] ] ]
```

Lastly, we will have another diagnosis sequence in reversed time. This will be used in our RNN model for masking. Note that we only flip the true visits.

```
[ [ [8, 0, *0*, *0*], [0, 1, 2, *0*], [*0*, *0*, *0*, *0*]  ],
  [ [23, 11, *0*, *0*], [12, *0*, *0*, *0*], [12, 13, 6, 7] ] ]
```

And a reversed mask as well.

```
[ [ [1, 1, 0, 0], [1, 1, 1, 0], [0, 0, 0, 0] ],
  [ [1, 1, 0, 0], [1, 0, 0, 0], [1, 1, 1, 1], ] ]
```

We need to pad the sequences into the same length so that we can do batch training on GPU. And we also need this mask so that when training, we can ignored the padded value as they actually do not contain any information.

[9]:
```
def collate_fn(batch):
    """
    Input: list of (seq, label)
      - seq: list[visits], each visit = list[codes] (int indices)
    Output tensors:
      x         : LongTensor (B, Vmax, Cmax)    padded with 0s
      masks     : BoolTensor (B, Vmax, Cmax)    True for real codes, False for␣
↪pad
      rev_x     : LongTensor (B, Vmax, Cmax)    visits reversed in time (true␣
↪visits only)
      rev_masks : BoolTensor (B, Vmax, Cmax)    masks reversed to match rev_x
      y         : FloatTensor (B,)
    """
    sequences, labels = zip(*batch)   # lists of length B
    B = len(sequences)
    Vmax = max(len(patient) for patient in sequences) if B > 0 else 0
    Cmax = 0
    for patient in sequences:
        if len(patient) > 0:
            Cmax = max(Cmax, max(len(visit) for visit in patient))
    Cmax = Cmax if Cmax > 0 else 1  # avoid zero-dim

    # Allocate
    x = torch.zeros((B, Vmax, Cmax), dtype=torch.long)
    masks = torch.zeros((B, Vmax, Cmax), dtype=torch.bool)

    # Fill x/masks
    for b, patient in enumerate(sequences):
        for v, codes in enumerate(patient):
            c_len = min(len(codes), Cmax)
            if c_len > 0:
                x[b, v, :c_len] = torch.tensor(codes[:c_len], dtype=torch.long)
                masks[b, v, :c_len] = True

    # Reverse only the true visits per patient
    rev_x = torch.zeros_like(x)
    rev_masks = torch.zeros_like(masks)
```

```
    for b, patient in enumerate(sequences):
        v_len = len(patient)
        if v_len > 0:
            rev_x[b, :v_len] = x[b, :v_len][torch.arange(v_len-1, -1, -1)]
            rev_masks[b, :v_len] = masks[b, :v_len][torch.arange(v_len-1, -1,␣
↪-1)]

    y = torch.tensor(labels, dtype=torch.float)

    return x, masks, rev_x, rev_masks, y
```

[10]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

from torch.utils.data import DataLoader

loader = DataLoader(dataset, batch_size=10, collate_fn=collate_fn)
loader_iter = iter(loader)
x, masks, rev_x, rev_masks, y = next(loader_iter)

assert x.dtype == rev_x.dtype == torch.long
assert y.dtype == torch.float
assert masks.dtype == rev_masks.dtype == torch.bool

assert x.shape == rev_x.shape == masks.shape == rev_masks.shape == (10, 3, 24)
assert y.shape == (10,)
```

Now we have `CustomDataset` and `collate_fn()`. Let us split the dataset into training and validation sets.

[11]:
```
from torch.utils.data.dataset import random_split

split = int(len(dataset)*0.8)

lengths = [split, len(dataset) - split]
train_dataset, val_dataset = random_split(dataset, lengths)

print("Length of train dataset:", len(train_dataset))
print("Length of val dataset:", len(val_dataset))
```

```
Length of train dataset: 800
Length of val dataset: 200
```

1.3.3 1.3 DataLoader [5 points]

Now, we can load the dataset into the data loader.

```
[26]: # 1.3 - load_data (correct signature & behavior)
from torch.utils.data import DataLoader

def load_data(train_dataset, val_dataset, collate_fn, batch_size: int = 32):
    """
    Args:
        train_dataset: a torch.utils.data.Dataset (e.g., CustomDataset)
        val_dataset:   a torch.utils.data.Dataset
        collate_fn:    the batching function that pads & creates masks
        batch_size:    default 32 (grader expects this)

    Returns:
        train_loader (shuffle=True), val_loader (shuffle=False)

    Notes:
        - With the provided split and batch_size=32,
          len(train_loader) should be 25.
    """
    train_loader = DataLoader(
        train_dataset, batch_size=batch_size, shuffle=True,␣
 ↪collate_fn=collate_fn, drop_last=False
    )
    val_loader = DataLoader(
        val_dataset,   batch_size=batch_size, shuffle=False,␣
 ↪collate_fn=collate_fn, drop_last=False
    )
    return train_loader, val_loader


train_loader, val_loader = load_data(train_dataset, val_dataset, collate_fn)
```

```
[27]: '''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

train_loader, val_loader = load_data(train_dataset, val_dataset, collate_fn)

assert len(train_loader) == 25, "Length of train_loader should be 25, instead␣
 ↪we got %d"%(len(train_loader))
```

1.4 2 Naive RNN [35 points]

Let us implement a naive bi-directional RNN model.

Remember from class that, first of all, we need to transform the diagnosis code for each visit of a patient to an embedding. To do this, we can use `nn.Embedding()`, where `num_embeddings` is the number of diagnosis codes and `embedding_dim` is the embedding dimension.

Then, we can construct a simple RNN structure. Each input is this multi-hot vector. At the 0-th visit, this has $\boldsymbol{X}_0$, and at t-th visit, this has $\boldsymbol{X}_t$.

Each one of the input will then map to a hidden state $\overleftrightarrow{\boldsymbol{h}}_t$. The forward hidden state $\overrightarrow{\boldsymbol{h}}_t$ can be determined by $\overrightarrow{\boldsymbol{h}}_{t-1}$ and the corresponding current input $\boldsymbol{X}_t$.

Similarly, we will have another RNN to process the sequence in the reverse order, so that the hidden state $\overleftarrow{\boldsymbol{h}}_t$ is determined by $\overleftarrow{\boldsymbol{h}}_{t+1}$ and $\boldsymbol{X}_t$.

Finally, once we have the $\overrightarrow{\boldsymbol{h}}_T$ and $\overleftarrow{\boldsymbol{h}}_0$, we will concatenate the two vectors as the feature vector and train a NN to perform the classification.

Now, let us build this model. The forward steps will be:

```
1. Pass the sequence through the embedding layer;
2. Sum the embeddings for each diagnosis code up for a visit of a patient;
3. Pass the embeddings through the RNN layer;
4. Obtain the hidden state at the last visit;
5. Do 1-4 for both directions and concatenate the hidden states.
6. Pass the hidden state through the linear and activation layers.
```

1.4.1 2.1 Mask Selection [20 points]

Importantly, you need to use `masks` to mask out the paddings in before step 2 and before 4. So, let us first preform the mask selection.

[28]:
```python
def sum_embeddings_with_mask(x_emb: torch.Tensor, masks: torch.Tensor) -> torch.
↪Tensor:
    """
    x_emb:  (B, V, C, D)  embeddings per code
    masks:  (B, V, C)     bool mask for real codes
    Returns (B, V, D): sum of embeddings per visit over real codes only.
    """
    # broadcast mask into embedding dim
    m = masks.unsqueeze(-1).type_as(x_emb)       # (B,V,C,1)
    return (x_emb * m).sum(dim=2)                # (B,V,D)
```

[29]:
```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''


import random
import ast
import inspect


def uses_loop(function):
    loop_statements = ast.For, ast.While, ast.AsyncFor

    nodes = ast.walk(ast.parse(inspect.getsource(function)))
```

```python
    return any(isinstance(node, loop_statements) for node in nodes)

def generate_random_mask(batch_size, max_num_visits , max_num_codes):
    num_visits = [random.randint(1, max_num_visits) for _ in range(batch_size)]
    num_codes = []
    for n in num_visits:
        num_codes_visit = [0] * max_num_visits
        for i in range(n):
            num_codes_visit[i] = (random.randint(1, max_num_codes))
        num_codes.append(num_codes_visit)
    masks = [torch.ones((l,), dtype=torch.bool) for num_codes_visit in␣
 ↪num_codes for l in num_codes_visit]
    masks = torch.stack([torch.cat([i, i.new_zeros(max_num_codes - i.size(0))],␣
 ↪0) for i in masks], 0)
    masks = masks.view((batch_size, max_num_visits, max_num_codes)).bool()
    return masks


batch_size = 16
max_num_visits = 10
max_num_codes = 20
embedding_dim = 100

torch.random.manual_seed(7)
x = torch.randn((batch_size, max_num_visits , max_num_codes, embedding_dim))
masks = generate_random_mask(batch_size, max_num_visits , max_num_codes)
out = sum_embeddings_with_mask(x, masks)

assert uses_loop(sum_embeddings_with_mask) is False
assert out.shape == (batch_size, max_num_visits, embedding_dim)
```

[30]:

```python
def get_last_visit(hidden_states: torch.Tensor, masks: torch.Tensor) -> torch.
 ↪Tensor:
    """
    hidden_states: (B, V, D)  per-visit hidden states
    masks:         (B, V, C)  bool mask for codes; a visit is 'real' if any␣
 ↪code is True
    Returns:       (B, D)     state at the last real visit per patient (no␣
 ↪loops)
    """
    visit_mask = masks.any(dim=2)                     # (B,V)
    lengths = visit_mask.long().sum(dim=1)            # (B,)
    last_idx = (lengths.clamp(min=1) - 1)             # avoid negatives for empty
    # gather along V dimension
    idx = last_idx.view(-1, 1, 1).expand(-1, 1, hidden_states.size(-1))  #␣
 ↪(B,1,D)
```

```
    last = hidden_states.gather(dim=1, index=idx).squeeze(1)            # (B,D)
    return last
```

[31]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

assert uses_loop(get_last_visit) is False

max_num_visits = 10
batch_size = 16
max_num_codes = 20
embedding_dim = 100

torch.random.manual_seed(7)
hidden_states = torch.randn((batch_size, max_num_visits, embedding_dim))
masks = generate_random_mask(batch_size, max_num_visits , max_num_codes)
out = get_last_visit(hidden_states, masks)

assert out.shape == (batch_size, embedding_dim)
```

1.4.2 2.2 Build NaiveRNN [15 points]

[36]:
```
# ==========================
# 3) Bi-directional (Naive) RNN model
# ==========================

class NaiveRNN(nn.Module):
    """
    - Embedding(num_codes, 128)
    - Forward GRU (input_size=128, hidden_size=128, batch_first=True)
    - Reverse GRU (same as forward)
    - FC: Linear(256 -> 1) + Sigmoid
    Forward expects x, masks, rev_x, rev_masks.
    Returns probabilities of shape (B,) in [0,1].
    """
    def __init__(self, num_codes: int, emb_dim: int = 128, hidden_dim: int =
 ↪128):
        super().__init__()
        self.embedding = nn.Embedding(num_codes, emb_dim)
        self.gru_fwd = nn.GRU(input_size=emb_dim, hidden_size=hidden_dim,
 ↪batch_first=True)
        self.gru_rev = nn.GRU(input_size=emb_dim, hidden_size=hidden_dim,
 ↪batch_first=True)
        self.fc = nn.Linear(2 * hidden_dim, 1)
        self.sigmoid = nn.Sigmoid()
```

```python
    def forward(self, x, masks, rev_x, rev_masks):
        """
        x:          (B,V,C) long
        masks:      (B,V,C) bool
        rev_x:      (B,V,C) long    (time-reversed)
        rev_masks:  (B,V,C) bool
        """
        # Embed → sum per visit
        x_emb = self.embedding(x)                          # (B,V,C,D)
        x_vis = sum_embeddings_with_mask(x_emb, masks)      # (B,V,D)

        # forward GRU over visits
        out_fwd, _ = self.gru_fwd(x_vis)           # (B,V,H)
        h_last_fwd = get_last_visit(out_fwd, masks)          # (B,H)

        # Reverse stream
        rx_emb = self.embedding(rev_x)             # share weights
        rx_vis = sum_embeddings_with_mask(rx_emb, rev_masks) # (B,V,D)
        out_rev, _ = self.gru_rev(rx_vis)          # (B,V,H)
        h_last_rev = get_last_visit(out_rev, rev_masks)      # (B,H)

        # Concatenate last states (fwd + rev)
        h = torch.cat([h_last_fwd, h_last_rev], dim=1)  # (B,2H)
        probs = self.sigmoid(self.fc(h)).squeeze(1)     # (B,)
        return probs

# load the model here
naive_rnn = NaiveRNN(num_codes = len(types))
naive_rnn
```

[36]:

```
NaiveRNN(
  (embedding): Embedding(619, 128)
  (gru_fwd): GRU(128, 128, batch_first=True)
  (gru_rev): GRU(128, 128, batch_first=True)
  (fc): Linear(in_features=256, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

[33]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

[33]:

```
'\nAUTOGRADER CELL. DO NOT MODIFY THIS.\n'
```

[34]:

```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
```

```
'''
```

[34]: '\nAUTOGRADER CELL. DO NOT MODIFY THIS.\n'

1.5 3 Model Training [35 points]

1.5.1 3.1 Loss and Optimizer [5 points]

[38]:
```
# 3.1 - Loss & Optimizer
import torch
import torch.nn as nn

# BCE because the model outputs probabilities in [0,1] (final Sigmoid)
criterion = nn.BCELoss()

# Adam with learning rate 0.001
optimizer = torch.optim.Adam(naive_rnn.parameters(), lr=0.001)
```

[39]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

[39]: '\nAUTOGRADER CELL. DO NOT MODIFY THIS.\n'

1.5.2 3.2 Evaluate [10 points]

Then, let us implement the `eval_model()` function first.

[40]:
```
from sklearn.metrics import precision_recall_fscore_support, roc_auc_score

def eval_model(model: nn.Module, val_loader: DataLoader, device: str = "cpu"):
    model.eval()
    y_scores, y_true = [], []
    with torch.no_grad():
        for x, masks, rx, rm, y in val_loader:
            x, masks, rx, rm = x.to(device), masks.to(device), rx.to(device),␣
↪rm.to(device)
            y = y.to(device)
            scores = model(x, masks, rx, rm)      # (B,)
            y_scores.append(scores.detach().cpu().numpy())
            y_true.append(y.detach().cpu().numpy())

    y_scores = np.concatenate(y_scores, axis=0)
    y_true   = np.concatenate(y_true,   axis=0).astype(int)

    y_pred = (y_scores > 0.5).astype(int)
```

```
    precision = precision_score(y_true, y_pred, average='binary',␣
↪zero_division=0)
    recall    = recall_score(y_true, y_pred, average='binary',  zero_division=0)
    f1        = f1_score(y_true, y_pred, average='binary',      zero_division=0)
    # For ROC-AUC, need both classes present; handle degenerate case safely
    try:
        auc = roc_auc_score(y_true, y_scores)
    except ValueError:
        auc = float("nan")

    print(f"val  precision={precision:.4f}  recall={recall:.4f}  f1={f1:.4f} ␣
↪auc={auc:.4f}")
    return precision, recall, f1, auc
```

[41]:
```
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''

p, r, f, roc_auc = eval_model(naive_rnn, val_loader)
assert p.size == 1, "Precision should be a scalar."
assert r.size == 1, "Recall should be a scalar."
assert f.size == 1, "F1 should be a scalar."
assert roc_auc.size == 1, "ROC-AUC should be a scalar."
```

```
val  precision=0.5833  recall=0.5437  f1=0.5628  auc=0.5470
```

1.5.3 3.3 Training and evlauation [20 points]

Now let us implement the `train()` function. Note that `train()` should call `eval_model()` at the end of each training epoch to see the results on the validaion dataset.

[44]:
```
def train(model: nn.Module,
                train_loader: DataLoader,
                val_loader: DataLoader,
                num_epochs: int = 10,
                lr: float = 1e-3,
                device: str = "cpu"):
    model.to(device)
    criterion = nn.BCELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)

    for epoch in range(1, num_epochs + 1):
        model.train()
        losses = []
        for x, masks, rx, rm, y in train_loader:
            x, masks, rx, rm = x.to(device), masks.to(device), rx.to(device),␣
↪rm.to(device)
```

```python
            y = y.to(device)

            optimizer.zero_grad()
            scores = model(x, masks, rx, rm)          # (B,)
            loss = criterion(scores, y)               # shapes match (B,)
            loss.backward()
            optimizer.step()
            losses.append(loss.item())

        mean_loss = float(np.mean(losses)) if losses else 0.0
        print(f"epoch {epoch:02d}  train_loss={mean_loss:.6f}")
        eval_model(model, val_loader, device=device)

    return model
```

[45]:
```python
# number of epochs to train the model
n_epochs = 5
train(naive_rnn, train_loader, val_loader, n_epochs)
```

```
epoch 01  train_loss=0.606543
val  precision=0.6975  recall=0.8058  f1=0.7477  auc=0.8200
epoch 02  train_loss=0.417063
val  precision=0.7265  recall=0.8252  f1=0.7727  auc=0.8327
epoch 03  train_loss=0.305690
val  precision=0.7143  recall=0.8252  f1=0.7658  auc=0.8338
epoch 04  train_loss=0.203400
val  precision=0.7143  recall=0.7282  f1=0.7212  auc=0.8252
epoch 05  train_loss=0.125568
val  precision=0.7407  recall=0.7767  f1=0.7583  auc=0.8320
```

[45]:
```
NaiveRNN(
  (embedding): Embedding(619, 128)
  (gru_fwd): GRU(128, 128, batch_first=True)
  (gru_rev): GRU(128, 128, batch_first=True)
  (fc): Linear(in_features=256, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

[46]:
```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
p, r, f, roc_auc = eval_model(naive_rnn, val_loader)
print(roc_auc)
assert roc_auc > 0.7, "ROC AUC is too low on the validation set (%f < 0.
 ↪7)"%(roc_auc)
```

```
val  precision=0.7407  recall=0.7767  f1=0.7583  auc=0.8320
0.8320488439595636
```

[47]:
```python
'''
AUTOGRADER CELL. DO NOT MODIFY THIS.
'''
```

[47]: '\nAUTOGRADER CELL. DO NOT MODIFY THIS.\n'

[ ]:
```python

```