CS 598 Final Project Draft Team 118

Srikur Kanuparthy srikurk2@illinois.edu
Kush Gupta kushg2@illinois.edu
Justin Quall jquall2@illinois.edu

GitHub Repo: https://github.com/justinqquall/CS_598_FINAL_PROJECT

Project Video: https://mediaspace.illinois.edu/media/t/1_tdinfc0b

Citation to Original Paper: Alex, Labach., Aslesha, Pokhrel., Xiaolei, Huang., S., Zuberi., Seung, Eun, Yi., Maksims, Volkovs., Tomi, Poutanen., Rahul, G., Krishnan. (2023). DuETT: Dual Event Time Transformer for Electronic Health Records. arXiv.org, abs/2304.13017 doi: 10.48550/arXiv.2304.13017

Introduction

DuETT: Dual Event Time Transformer for Electronic Health Records

The original paper presents a transformer-based deep learning architecture called DuETT (Dual Event Time Transformer for Electronic Health Records) which considers both time series and event type data. This is optimized for working with tabular data sets marked by sparsity and by irregular distribution over time, notably electronic health record (EHR) data.

The main contribution is the novel architecture itself, which outperforms transformers considering time series and event modalities independently, as well as state-of-the-art deep learning frameworks like XGBoost, mTAND, and Raindrop. The models were all compared in a number of analyses on MIMIC-IV and PhysioNet-2012 datasets. Furthermore, other contributions are made to enable the DuETT architecture.

These include designing an input representation which factors in event frequency and missingness, early fusion of static variables, and aggregates observations to allow deeper model structures to be used (without dramatically increasing computational needs).

Additionally, the authors developed a new self-supervised training method which executes masking of measured event values and missingness for both time and event modalities.

Scope of Reproducibility:

Where feasible, we intend to reproduce the DuETT model and hypothesize seeing similar results to the authors original work. The code is available in a decently well-documented public repository, so the biggest challenge will be fairly heavy computation demands. Even with their thoughtful pre-processing techniques, the authors noted needed 2 days of running an Nvidia A6000 GPU for some DuETT pre-training and fine-turning steps.

Due to that computational feasibility and data wrangling limitations, we do not expect to be able to run DuETT on both datasets (MIMIC-IV and PhysioNet-2012) within the scope of the project's timelines. Hence we will limit the scope of reproducibility to the January 20, 2012 version of PhysioNet-2012 (1.0.0).

HYPOTHESIS 1: We expect to be able to closely reproduce the core results on the PhysioNet-2012 dataset. We will compare our results to those shown in Table 1 for the PhysioNet-2012 dataset, keeping the same key measures, ROC-AUC and PR-AUC.

HYPOTHESIS 2: Furthermore, we plan to test a data ablation piece where 10-20% of the data is omitted for each patient sample. This form of ablation may make sense here because of the paper's use of novel self-supervised learning to augment model training in the context of limited labeled data. This may be relevant to real-world healthcare practice, where EMRs may only cover the scope of one health system (whilst high-acuity patients receive care at multiple health systems on different EMRs/instances).

Methodology

First, we clone the GitHub repository for the paper and install its dependent libraries. Note that for the dependencies to work properly, we need to be using either Python version 3.8 or 3.9. To successfully run the original model. Overall, the key software steps for overall replication of the core results were:

1. Cloning the full public Github repository
2. Setting up a virtual runtime environment running Python 3.9 (not specified below but possible using pyenv or venv)
3. Installing the specific dependencies down to the version level as specified in the requirements.txt file

[1]
```python
# Check that python version is either 3.8 or 3.9
import sys
if sys.version_info < (3, 8):
    sys.exit('Python 3.8 or later is required to run this program.')
if sys.version_info >= (3, 10):
    sys.exit('Python 3.9 or earlier is required to run this program.')
```

```
!git clone https://github.com/layer6ai-labs/DuETT.git
%cd DuETT/
!pip install -r requirements.txt
```

[3] Python

```
Cloning into 'DuETT'...
remote: Enumerating objects: 13, done.
remote: Counting objects: 100% (13/13), done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 13 (delta 3), reused 11 (delta 3), pack-reused 0
Receiving objects: 100% (13/13), 1.54 MiB | 12.85 MiB/s, done.
Resolving deltas: 100% (3/3), done.
/Users/srikur/Desktop/mcs/598-dlh/DuETT
Requirement already satisfied: numpy==1.21.5 in /Users/srikur/miniconda3/envs/cs598env/lib/
Requirement already satisfied: pytorch-lightning==1.6.1 in /Users/srikur/miniconda3/envs/cs
Requirement already satisfied: torch==1.13.1 in /Users/srikur/miniconda3/envs/cs598env/lib/
Requirement already satisfied: torchaudio==0.13.1 in /Users/srikur/miniconda3/envs/cs598env
Requirement already satisfied: torchvision==0.14.1 in /Users/srikur/miniconda3/envs/cs598er
Requirement already satisfied: x-transformers==1.5.3 in /Users/srikur/miniconda3/envs/cs598
Requirement already satisfied: torchtime==0.5.1 in /Users/srikur/miniconda3/envs/cs598env/l
Requirement already satisfied: torchmetrics==0.8.0 in /Users/srikur/miniconda3/envs/cs598er
Requirement already satisfied: tqdm>=4.41.0 in /Users/srikur/miniconda3/envs/cs598env/lib/p
Requirement already satisfied: PyYAML>=5.4 in /Users/srikur/miniconda3/envs/cs598env/lib/py
Requirement already satisfied: fsspec!=2021.06.0,>=2021.05.0 in /Users/srikur/miniconda3/er
Requirement already satisfied: tensorboard>=2.2.0 in /Users/srikur/miniconda3/envs/cs598env
Requirement already satisfied: pyDeprecate<0.4.0,>=0.3.1 in /Users/srikur/miniconda3/envs/c
Requirement already satisfied: packaging>=17.0 in /Users/srikur/miniconda3/envs/cs598env/li
Requirement already satisfied: typing-extensions>=4.0.0 in /Users/srikur/miniconda3/envs/cs
Requirement already satisfied: requests in /Users/srikur/miniconda3/envs/cs598env/lib/pythc
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in /Users/srikur/miniconda3/envs/cs598
...
Requirement already satisfied: patsy>=0.5.4 in /Users/srikur/miniconda3/envs/cs598env/lib/p
Requirement already satisfied: MarkupSafe>=2.1.1 in /Users/srikur/miniconda3/envs/cs598env/
Requirement already satisfied: zipp>=0.5 in /Users/srikur/miniconda3/envs/cs598env/lib/pyth
DEPRECATION: pytorch-lightning 1.6.1 has a non-standard dependency specifier torch>=1.8.*.
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

Data

The data is from the PhysioNet2012 challenge. The data consist of records from 12,000 ICU stays by patients, all adults who were admitted for various reasons. The dataset is publicly available, and the torchtime Python package makes it easy to load and use the data. 70 percent of the data is used for training, and the remaining 30 percent are used as test and validation sets. Most of the data importing and processing occurs in our repository's "physionet.py" file (as originally published).

```
'''tt_data = PhysioNet2012(self.split_name, train_prop=0.7, val_prop=0.15, time=False,
# Here, split_name is the name of the split to use. It can be one of 'train', 'val', 't
```

[12] Python

```
'tt_data = PhysioNet2012(self.split_name, train_prop=0.7, val_prop=0.15, time=False, seed=0
```

Model

Here is a citation to the original paper: Alex, Labach., Aslesha, Pokhrel., Xiaolei, Huang., S., Zuberi., Seung, Eun, Yi., Maksims, Volkovs., Tomi, Poutanen., Rahul, G., Krishnan. (2023). DuETT: Dual Event Time Transformer for Electronic Health Records. arXiv.org, abs/2304.13017 doi: 10.48550/arXiv.2304.13017

And the original paper's repo is located at the following Github repository: https://github.com/layer6ai-labs/DuETT

From the paper, the structure of the model is a series of layers followed by classification or self-supervised learning heads. Each layer is made up of two Transformer sublayers. "The first sublayer consists of multi-head attention over events followed by a feed-forward network operating along the event dimension, which can be collectively identified as an event transformer layer; the second sublayer consists of multi-head attention over time bins followed by a feed-forward network operating along the time dimension, the time transformer layer." After those "DuETT" combination layers, the results are fed into self-supervised learning heads for classification.

Here is the model class code in full:

```
'''class Model(pl.LightningModule):
    def __init__(self, d_static_num, d_time_series_num, d_target, lr=3.e-4, weight_deca
        scalenorm=True, n_hidden_mlp_embedding=1, d_hidden_mlp_embedding=64, d_embe
        max_len=48, n_transformer_head=2, n_duett_layers=2, d_hidden_tab_encoder=12
        norm_first=True, fusion_method='masked_embed', n_hidden_head=1, d_hidden_he
        pretrain=True, pretrain_masked_steps=1, pretrain_n_hidden=0, pretrain_d_hid
        pretrain_value=True, pretrain_presence=True, pretrain_presence_weight=0.2,
        transformer_dropout=0., pos_frac=None, freeze_encoder=False, seed=0, save_r
        masked_transform_timesteps=32, **kwargs):
        super().__init__()
        self.lr = lr
        self.weight_decay = weight_decay
        self.d_time_series_num = d_time_series_num
        self.d_target = d_target
        self.d_embedding = d_embedding
        self.max_len = max_len
        self.pretrain = pretrain
        self.pretrain_masked_steps = pretrain_masked_steps
        self.pretrain_dropout = pretrain_dropout
        self.freeze_encoder = freeze_encoder
        self.set_pos_frac(pos_frac)
        self.rng = np.random.default_rng(seed)
        self.aug_noise = aug_noise
        self.aug_mask = aug_mask
        self.fusion_method = fusion_method
        self.pretrain_presence = pretrain_presence
        self.pretrain_presence_weight = pretrain_presence_weight
        self.predict_events = predict_events
        self.masked_transform_timesteps = masked_transform_timesteps
        self.pretrain_value = pretrain_value
        self.save_representation = save_representation
        self.register_buffer("MASKED_EMBEDDING_KEY", torch.tensor(0)) # For multi-gpu t
        self.register_buffer("REPRESENTATION_EMBEDDING_KEY", torch.tensor(1))

        # For any special timesteps, e.g., masked, static, [CLS], etc.
        self.special_embeddings = nn.Embedding(8, d_embedding)
        self.embedding_layers = nn.ModuleList([
            simple_mlp(2, d_embedding, n_hidden_mlp_embedding, d_hidden_mlp_embedding,
            for _ in range(d_time_series_num)])
```

```
        self.n_obs_embedding = nn.Embedding(16, 1)

        if d_feedforward is None:
            d_feedforward = d_embedding * 4

        et_dim = d_embedding*(masked_transform_timesteps+1)
        tt_dim = d_embedding*(d_time_series_num+1)
        self.event_transformers = nn.ModuleList([x_transformers.Encoder(dim=et_dim, dep
                heads=n_transformer_head, pre_norm=norm_first, use_scalenorm=scalenorm,
                attn_dim_head=d_embedding//n_transformer_head, ff_glu=glu,
                ff_mult=d_feedforward/et_dim, attn_dropout=transformer_dropout,
                ff_dropout=transformer_dropout) for _ in range(n_duett_layers)])
        self.full_event_embedding = nn.Embedding(d_time_series_num + 1, et_dim)
        self.time_transformers = nn.ModuleList([x_transformers.Encoder(dim=tt_dim, dept
                heads=n_transformer_head, pre_norm=norm_first, use_scalenorm=scalenorm,
                attn_dim_head=d_embedding//n_transformer_head, ff_glu=glu,
                ff_mult=d_feedforward/tt_dim, attn_dropout=transformer_dropout,
                ff_dropout=transformer_dropout) for _ in range(n_duett_layers)])
        self.full_time_embedding =  self.cve(batch_norm=True, d_embedding=tt_dim)
        self.full_rep_embedding = nn.Embedding(tt_dim, 1)

        d_representation = d_embedding * (d_time_series_num + 1) # time_series + static
        self.head = simple_mlp(d_representation, d_target, n_hidden_head, d_hidden_head
                hidden_batch_norm=True, final_activation=False, activation=nn.ReLU)
        self.pretrain_value_proj = simple_mlp(d_representation, d_time_series_num,
                pretrain_n_hidden, pretrain_d_hidden, hidden_batch_norm=True)
        if self.pretrain_presence:
            self.pretrain_presence_proj = simple_mlp(d_representation, d_time_series_nu
                    pretrain_n_hidden, pretrain_d_hidden, hidden_batch_norm=True)
        if self.predict_events:
            self.predict_events_proj = simple_mlp(et_dim, masked_transform_timesteps,
                    pretrain_n_hidden, pretrain_d_hidden, hidden_batch_norm=True)
            if self.pretrain_presence:
                self.predict_events_presence_proj = simple_mlp(et_dim, masked_transform
                        pretrain_n_hidden, pretrain_d_hidden, hidden_batch_norm=True)

        self.tab_encoder = simple_mlp(d_static_num, d_embedding, n_hidden_tab_encoder,
                d_hidden_tab_encoder, hidden_batch_norm=True)
```

```
        self.pretrain_loss = F.mse_loss
        self.loss_function = F.binary_cross_entropy_with_logits
        self.pretrain_presence_loss = F.binary_cross_entropy_with_logits
        num_classes = None if d_target == 1 else d_target
        self.train_auroc = torchmetrics.AUROC(num_classes=num_classes)
        self.val_auroc = torchmetrics.AUROC(num_classes=num_classes)
        self.train_ap = torchmetrics.AveragePrecision(num_classes=num_classes)
        self.val_ap = torchmetrics.AveragePrecision(num_classes=num_classes)
        self.test_auroc = torchmetrics.AUROC(num_classes=num_classes)
        self.test_ap = torchmetrics.AveragePrecision(num_classes=num_classes)

    def set_pos_frac(self, pos_frac):
        if type(pos_frac) == list:
            pos_frac = torch.tensor(pos_frac, device=torch.device('cuda'))
        self.pos_frac = pos_frac
        if pos_frac != None:
            self.pos_weight = 1 / (2 * pos_frac)
            self.neg_weight = 1 / (2 * (1 - pos_frac))

    def cve(self, d_embedding=None, batch_norm=False):
        if d_embedding == None:
            d_embedding = self.d_embedding
        d_hidden = int(np.sqrt(d_embedding))
        if batch_norm:
            return nn.Sequential(nn.Linear(1, d_hidden), nn.Tanh(), BatchNormLastDim(d_
        return nn.Sequential(nn.Linear(1, d_hidden), nn.Tanh(), nn.Linear(d_hidden, d_e

    def feats_to_input(self, x, batch_size, limits=None):
        xs_ts, xs_static, times = x
        xs_ts = list(xs_ts)

        for i,f in enumerate(xs_ts):
            n_vars = f.shape[1] // 2
            if f.shape[0] > self.max_len:
                f = f[-self.max_len:]
                times[i] = times[i][-self.max_len:]
            # Aug
            if self.training and self.aug_noise > 0 and not self.pretrain:
                f[:,:n_vars] += self.aug_noise * torch.randn_like(f[:,:n_vars]) * f[:,n
            f = torch.cat((f, torch.zeros_like(f[:,:1])), dim=1)
            if self.training and self.aug_mask > 0 and not self.pretrain:
                mask = torch.rand(f.shape[0]) < self.aug_mask
                f[mask,:] = 0.
                f[mask,-1] = 1.
```

```
            xs_ts[i] = f
        n_timesteps = [len(ts) for ts in times]

        pad_to = np.max(n_timesteps)
        xs_ts = torch.stack([F.pad(t, (0, 0, 0, pad_to-t.shape[0])) for t in xs_ts]).to
        xs_times = torch.stack([F.pad(t, (0, pad_to-t.shape[0])) for t in times]).to(se
        xs_static = torch.stack(xs_static).to(self.device)

        if self.training and self.aug_noise > 0 and not self.pretrain:
            xs_static += self.aug_noise * torch.randn_like(xs_static)

        return xs_static, xs_ts, xs_times, n_timesteps

    def pretrain_prep_batch(self, x, batch_size):
        xs_static, xs_ts, xs_times, n_timesteps = self.feats_to_input(x, batch_size)
        n_steps = xs_ts.shape[1]
        n_vars = (xs_ts.shape[2] - 1) // 2
        y_ts = []
        y_ts_n_obs = []
        y_events = []
        y_events_mask = []
        xs_ts_clipped = xs_ts.clone()
        for batch_i, n in enumerate(n_timesteps):
            if n < 2:
                mask_i = n
            elif self.pretrain_masked_steps > 1:
                if self.pretrain_masked_steps > n:
                    mask_i = np.arange(n)
                else:
                    mask_i = self.rng.choice(np.arange(n), size=self.pretrain_masked_st
            else:
                mask_i = self.rng.choice(np.arange(0, n))
            y_ts.append(xs_ts[batch_i,mask_i,:n_vars])
            y_ts_n_obs.append(xs_ts[batch_i,mask_i,n_vars:2*n_vars])

            xs_ts_clipped[batch_i, mask_i, :] = 0.
            xs_ts_clipped[batch_i,mask_i,-1] = 1.

            if self.predict_events:
                event_mask_i = self.rng.choice(np.arange(0, self.d_time_series_num))
                y_events.append(xs_ts[batch_i, :, event_mask_i])
                y_events_mask.append(xs_ts[batch_i, :, event_mask_i + n_vars].clip(0,1)
                xs_ts_clipped[batch_i, :, event_mask_i] = 0
                xs_ts_clipped[batch_i, :, event_mask_i + n_vars] = -1
```

```
        y_ts = torch.stack(y_ts)
        y_ts_n_obs = torch.stack(y_ts_n_obs)
        y_ts_masks = y_ts_n_obs.clip(0,1)
        if len(y_events) > 0:
            y_events = torch.stack(y_events)
            y_events_mask = torch.stack(y_events_mask)
        if self.pretrain_dropout > 0:
            keep = self.rng.random((batch_size, n_vars)) > self.pretrain_dropout
            keep = torch.tensor(keep, device=xs_ts.device)
            # Only drop out values that are unmasked in y
            if y_ts_masks.ndim > 2:
                keep = torch.logical_or(1 - y_ts_masks.sum(dim=1).clip(0,1), keep)
            else:
                keep = torch.logical_or(1 - y_ts_masks, keep)
            keep = torch.cat((keep.tile(1,2), torch.ones((batch_size, 1), device=keep.d
            xs_ts_clipped *= torch.logical_or(keep.unsqueeze(1), xs_ts_clipped == -1)
        return (xs_static, xs_ts_clipped, xs_times, n_timesteps), y_ts, y_ts_masks, y_e

    def forward(self, x, pretrain=False, representation=False):
        """
        Forward run
        :param x: input to the model
        :return: prediction output (i.e., class probabilities vector)
        """
        xs_static, xs_feats, xs_times, n_timesteps = x
        n_vars = xs_feats.shape[2] // 2
        if self.predict_events:
            event_mask_inds = xs_feats[:,:,n_vars:n_vars*2] == -1
            event_mask_inds = torch.cat((event_mask_inds, torch.zeros(xs_feats.shape[:2
            event_mask_inds = torch.cat((event_mask_inds, event_mask_inds[:,:1,:]), dim
        n_obs_inds = xs_feats[:,:,n_vars:n_vars*2].to(int).clip(0, self.n_obs_embedding
        xs_feats[:,:,n_vars:n_vars*2] = self.n_obs_embedding(n_obs_inds).squeeze(-1)

        embedding_layer_input = torch.empty(xs_feats.shape[:-1] + (n_vars, 2), dtype=xs
        embedding_layer_input[:,:,:,0] = xs_feats[:,:,:n_vars]
        embedding_layer_input[:,:,:,1] = xs_feats[:,:,n_vars:n_vars*2]
        # dims: batch, time step, var, embedding
        psi = torch.zeros((xs_feats.shape[0], xs_feats.shape[1]+1, n_vars+1, self.d_emb
        for i, el in enumerate(self.embedding_layers):
            psi[:,:-1,i,:] = el(embedding_layer_input[:,:,i,:])
        psi[:,:-1,-1,:] = self.tab_encoder(xs_static).unsqueeze(1)
        psi[:,-1,:,:] = self.special_embeddings(self.REPRESENTATION_EMBEDDING_KEY.to(se
        mask_inds = torch.cat((xs_feats[:,:,-1] == 1, torch.zeros((xs_feats.shape[0], 1
        psi[mask_inds, :, :] = self.special_embeddings(self.MASKED_EMBEDDING_KEY.to(sel
        if self.predict_events:
```

```
            psi[event_mask_inds, :] = self.special_embeddings[                    _KEY

        # batch, time step, full embedding
        time_embeddings = self.full_time_embedding(xs_times.unsqueeze(2))
        time_embeddings = torch.cat((time_embeddings,
            self.full_rep_embedding.weight.T.unsqueeze(0).expand(xs_feats.shape[0],-1,-
            dim=1)
        for layer_i, (event_transformer, time_transformer) in enumerate(zip(self.event_
            et_out_shape = (psi.shape[0], psi.shape[2], psi.shape[1], psi.shape[3])
            embeddings = psi.transpose(1,2).flatten(2) + self.full_event_embedding.weig
            event_outs = event_transformer(embeddings).view(et_out_shape).transpose(1,2
            tt_out_shape = event_outs.shape
            embeddings = event_outs.flatten(2) + time_embeddings
            psi = time_transformer(embeddings).view(tt_out_shape)
        transformed = psi.flatten(2)

        if self.fusion_method == 'rep_token':
            z_ts = transformed[:,-1,:]
        elif self.fusion_method == 'masked_embed':
            if self.pretrain_masked_steps > 1:
                masked_ind = F.pad(xs_feats[:,:,-1] > 0, (0,1), value=False)
                z_ts = []
                for i in range(transformed.shape[0]):
                    z_ts.append(F.pad(transformed[i, masked_ind[i],:], (0,0,0,self.pret
                z_ts = torch.stack(z_ts) # batch size x pretrain_masked_steps x d_embed
            else:
                masked_ind = xs_feats[:,:,-1:]
                z_ts = []
                for i in range(transformed.shape[0]):
                    z_ts.append(transformed[i, torch.nonzero(masked_ind[i].squeeze()==1
                z_ts = torch.cat(z_ts, dim=0).squeeze()
        elif self.fusion_method == 'averaging':
            z_ts = torch.mean(transformed[:,:-1,:], dim=1)

        z = z_ts
        if representation:
            return z

        if pretrain:
            rep_token_head = torch.tile(transformed[:,0,:].unsqueeze(1), (1, self.maske
            y_hat_presence = self.pretrain_presence_proj(z).squeeze() if self.pretrain_
            y_hat_value = self.pretrain_value_proj(z).squeeze(1) if self.pretrain_value
            z_events = []
            y_hat_events, y_hat_events_presence = None, None
```

```
            if self.predict_events:
                for i in range(event_mask_inds.shape[0]):
                    z_events.append(psi[i][event_mask_inds[i].nonzero(as_tuple=True)].f
                z_events = torch.stack(z_events)
                y_hat_events = self.predict_events_proj(z_events).squeeze()
                y_hat_events_presence = self.predict_events_presence_proj(z_events).squ
            return y_hat_value, y_hat_presence, y_hat_events, y_hat_events_presence

        out = self.head(z).squeeze(1)

        if self.save_representation:
            return out, z
        else:
            return out

    def configure_optimizers(self):
        optimizers = [torch.optim.AdamW([p for l in self.modules() for p in l.parameter
                lr=self.lr, weight_decay=self.weight_decay)]
        return optimizers

    def training_step(self, batch, batch_idx):
        x, y = batch
        y = torch.tensor(y, dtype=torch.float64, device=self.device)
        batch_size = y.shape[0]
        if self.pretrain:
            x_pretrain, y, mask, y_events, y_events_mask = self.pretrain_prep_batch(x,
            y_hat_value, y_hat_presence, y_hat_events, y_hat_events_presence = self.for

            loss = 0
            if self.pretrain_value:
                if self.pretrain_masked_steps > 1:
                    for i in range(self.pretrain_masked_steps):
                        loss += self.pretrain_loss(y_hat_value[:,i]*mask[:,i], y[:,i]*m
                    loss /= self.pretrain_masked_steps
                else:
                    loss = self.pretrain_loss(y_hat_value*mask, y*mask)
            if self.pretrain_presence:
                if self.pretrain_masked_steps > 1:
                    presence_loss = 0
                    for i in range(self.pretrain_masked_steps):
                        presence_loss += self.pretrain_presence_loss(y_hat_presence[:,i
                    presence_loss /= self.pretrain_masked_steps
                else:
                    presence_loss = self.pretrain_presence_loss(y_hat_presence, mask) *
```

```
                loss += presence_loss
            if self.predict_events:
                if self.pretrain_value:
                    loss += self.pretrain_loss(y_hat_events*y_events_mask, y_events*y_e
                if self.pretrain_presence:
                    loss += self.pretrain_presence_loss(y_hat_events_presence, y_events
        else:
            y_hat = self.forward(self.feats_to_input(x, batch_size))
            if self.pos_frac is not None:
                weight = torch.where(y > 0, self.pos_weight, self.neg_weight)
                loss = self.loss_function(y_hat, y, weight)
            else:
                loss = self.loss_function(y_hat, y)
            self.train_auroc.update(y_hat, y.to(int))
            self.train_ap.update(y_hat, y.to(int))

        # Workaround to fix the loss=nan issue on the train progress bar
        # self.trainer.train_loop.running_loss.append(loss)
        self.log('train_loss', loss, sync_dist=True)
        return loss

    def validation_step(self, batch, batch_idx):
        x, y = batch
        y = torch.tensor(y, dtype=torch.float64, device=self.device)
        batch_size = y.shape[0]
        if self.pretrain:
            x_pretrain, y, mask, y_events, y_events_mask = self.pretrain_prep_batch(x,
            y_hat_value, y_hat_presence, y_hat_events, y_hat_events_presence = self.for

            loss = 0
            if self.pretrain_value:
                if self.pretrain_masked_steps > 1:
                    for i in range(self.pretrain_masked_steps):
                        loss += self.pretrain_loss(y_hat_value[:,i]*mask[:,i], y[:,i]*m
                    loss /= self.pretrain_masked_steps
                else:
                    loss = self.pretrain_loss(y_hat_value*mask, y*mask)
                self.log('val_next_loss', loss, on_epoch=True, sync_dist=True, rank_zer
            if self.pretrain_presence:
                if self.pretrain_masked_steps > 1:
                    presence_loss = 0
                    for i in range(self.pretrain_masked_steps):
```

```
                    presence_loss += self.pretrain_presen                    :,i
                presence_loss /= self.pretrain_masked_steps
            else:
                presence_loss = self.pretrain_presence_loss(y_hat_presence, mask) *
            self.log('val_presence_loss', presence_loss, on_epoch=True, sync_dist=T
            loss += presence_loss
        if self.predict_events:
            event_loss = self.pretrain_loss(y_hat_events*y_events_mask, y_events*y_
            self.log('val_event_loss', event_loss, on_epoch=True, sync_dist=True, r
            loss += event_loss
    else:
        y_hat = self.forward(self.feats_to_input(x, batch_size))
        if self.pos_frac is not None:
            weight = torch.where(y > 0, self.pos_weight, self.neg_weight)
            loss = self.loss_function(y_hat, y, weight)
        else:
            loss = self.loss_function(y_hat, y)
        self.val_auroc.update(y_hat, y.to(int).to(self.device))
        self.val_ap.update(y_hat, y.to(int).to(self.device))

    if not self.pretrain:
        self.log('val_ap', self.val_ap, on_epoch=True, sync_dist=True, rank_zero_on
        self.log('val_auroc', self.val_auroc, on_epoch=True, sync_dist=True, rank_z
    self.log('val_loss', loss, on_epoch=True, sync_dist=True, prog_bar=True, rank_z

def training_epoch_end(self, training_step_outputs):
    if not self.pretrain:
        self.log('train_auroc', self.train_auroc, sync_dist=True, rank_zero_only=Tr
        self.log('train_ap', self.train_ap, sync_dist=True, rank_zero_only=True)

def validation_epoch_end(self, validation_step_outputs):
    if not self.pretrain:
        print("val_auroc", self.val_auroc.compute(), "val_ap", self.val_ap.compute(

def test_step(self, batch, batch_idx):
    x, y = batch
    y = torch.tensor(y, dtype=torch.float64, device=self.device)
    batch_size = y.shape[0]
    if self.save_representation:
        y_hat, z = self.forward(self.feats_to_input(x, batch_size))

        print("saving representations...")
        with open(self.save_representation, 'ab') as f:
            if y.ndim == 1:
```

```
                    np.savetxt(f,np.concatenate([z.cpu(), y.unsqueeze(1).cpu()], axis=1
                else:
                    np.savetxt(f,np.concatenate([z.cpu(), y.cpu()], axis=1))
        else:
            y_hat = self.forward(self.feats_to_input(x, batch_size))
        if self.pos_frac is not None:
            weight = torch.where(y > 0, self.pos_weight, self.neg_weight)
            loss = self.loss_function(y_hat, y, weight)
        else:
            loss = self.loss_function(y_hat, y)
        self.log('test_loss', loss, on_epoch=True, sync_dist=True, rank_zero_only=True)
        self.test_auroc.update(y_hat, y.to(int).to(self.device))
        self.log('test_auroc', self.test_auroc, on_epoch=True, sync_dist=True, rank_zer
        self.test_ap.update(y_hat, y.to(int).to(self.device))
        self.log('test_ap', self.test_ap, on_epoch=True, sync_dist=True, rank_zero_only

        return loss, self.test_auroc, self.test_ap

    def on_load_checkpoint(self, checkpoint):
        # Ignore errors from size mismatches in head, since those might change between
        # and supervised training
        # Adapted from https://github.com/PyTorchLightning/pytorch-lightning/issues/469
        print('Loading from checkpoint')
        state_dict = checkpoint["state_dict"]
        model_state_dict = self.state_dict()
        is_changed = False
        for k in model_state_dict:
            if k not in state_dict:
                state_dict[k] = model_state_dict[k]
                is_changed = True
        for k in state_dict:
            if k in model_state_dict:
                if k.startswith('head') and state_dict[k].shape != model_state_dict[k].
                    print(f"Skip loading parameter: {k}, "
                                f"required shape: {model_state_dict[k].shape}, "
                                f"loaded shape: {state_dict[k].shape}")
                    state_dict[k] = model_state_dict[k]
                    is_changed = True
            else:
                print(f"Dropping parameter {k}")
                is_changed = True
```

```
            if is_changed:
                checkpoint.pop("optimizer_states", None)

            if self.freeze_encoder:
                self.freeze()

        def freeze(self):
            print('Freezing')
            for n, w in self.named_parameters():
                if "head" not in n:
                    w.requires_grad = False
                else:
                    print("Skip freezing:", n)
    '''
```

[11] Python

Training

Explanation of selected hyperparameters used in model training:

1. Maximum Epochs (max_epochs): This parameter is set for the trainer and defines the maximum number of epochs to run the training process. It is set to 300 in the pretraining phase and 50 in the fine-tuning phase.
2. Gradient Clipping Value (gradient_clip_val): Set in the trainer to a value of 1.0. Gradient clipping is a technique used to prevent exploding gradients in neural networks by capping them to a maximum value during backpropagation.
3. Learning Rate Decay (decay): This is used in the WarmUpCallback to adjust the rate of learning rate decay after the warm-up period. The default is set to the same as steps if not specified.

The **WarmUpCallback** class is designed to manage learning rate adjustments during model training. It starts with a warm-up period where the learning rate increases linearly from zero to a predefined or automatically detected base learning rate over a specified number of steps. Once the warm-up phase is completed, the callback can optionally apply an inverse square root decay to the learning rate for additional specified steps. This approach helps in stabilizing the model's training early on by gradually increasing the learning rate and then adjusting it to prevent overshooting as training progresses. The callback also includes functionalities to save and load its state, facilitating training resumption or state transfer between sessions.

The code below loads a batch of the physionet data and pretrains the model using the duett model. This pretrained model is then Checkpointed and saved. The pl Trainer then fits the pretrained model on the physionet data that we pulled. Since the training takes a while, this step is commented out and we can use the checkpointed model to run the next steps.

```
    '''
    pl.seed_everything(seed)
    dm = physionet.PhysioNetDataModule(batch_size=512, num_workers=16, use_temp_cache=True)
    dm.setup()
    pretrain_model = duett.pretrain_model(d_static_num=dm.d_static_num(),
            d_time_series_num=dm.d_time_series_num(), d_target=dm.d_target(), pos_frac=dm.p
            seed=seed)
    checkpoint = pl.callbacks.ModelCheckpoint(save_last=True, monitor='val_loss', mode='min
    warmup = WarmUpCallback(steps=2000)
    trainer = pl.Trainer(gpus=1, logger=False, num_sanity_val_steps=2, max_epochs=300,
            gradient_clip_val=1.0, callbacks=[warmup, checkpoint])
    trainer.fit(pretrain_model, dm)
    '''
```

[13] Python

··· "\npl.seed_everything(seed)\ndm = physionet.PhysioNetDataModule(batch_size=512, num_workers

The computational requirements were fairly vast. To get the model trained in less than three hours, we needed to use a virtual machine with 32 GB of GPU memory, basically fully dedicated to the task. Compared to local hardware acceleration on an upgraded commercial Macbook, this reduce runtime per epoch from around 5 minutes per epoch to around 30 seconds per epoch (10x faster). With the standard training completed over 300 epochs, this all added up pretty quickly and turned out to be a very computationally intensive model to train.

Results

*** Note: due to computational intensity, the model was trained and fine-tuned on a virtual machine with strong hardware acceleration (via T4 GPU). ***

The commented out code blocks below will load in the dataset and run through different key steps of setting up the model parameters and training functions. Please uncomment these out if you'd like to go step-by-step, otherwise at the end of this section the full results from running on a GPU-accelerated virtual machine are shown (this is what was done to reproduce the paper's metrics as closely as possible).

```
# import torch
# from DuETT import duett
# from DuETT import physionet

# # Load the PhysioNet2012 Data
# dm = physionet.PhysioNetDataModule(batch_size=512, num_workers=16)
# dm.setup()
```

[ ] Python

```
Running cells with 'Python 3.12.3' requires the ipykernel package.
Run the following command to install 'ipykernel' into the Python environment.
Command: '/opt/homebrew/bin/python3 -m pip install ipykernel -U --user --force-reinstall'
```

The following class and average_models function are copied directly from train.py in the repository, since the file contains top level code that we do not wish to run.

```
# import pytorch_lightning as pl
# from pathlib import Path

# class WarmUpCallback(pl.callbacks.Callback):
#     """Linear warmup over warmup_steps batches, tries to auto-detect the base lr"""
#     def __init__(self, steps=1000, base_lr=None, invsqrt=True, decay=None):
#         print('warmup_steps {}, base_lr {}, invsqrt {}, decay {}'.format(steps, base_
#         self.warmup_steps = steps
#         if decay is None:
#             self.decay = steps
#         else:
#             self.decay = decay

#         if base_lr is None:
#             self.state = {'steps': 0, 'base_lr': base_lr}
#         else:
#             self.state = {'steps': 0, 'base_lr': float(base_lr)}

#         self.invsqrt = invsqrt

#     def set_lr(self, optimizer, lr):
#         for param_group in optimizer.param_groups:
#             param_group['lr'] = lr

#     def on_train_batch_start(self, trainer, model, batch, batch_idx):
#         optimizers = model.optimizers()

#         if self.state['steps'] < self.warmup_steps:
#             if type(optimizers) == 'list':
#                 if self.state['base_lr'] is None:
#                     self.state['base_lr'] = [o.param_groups[0]['lr'] for o in optimiz
#                 for opt,base in zip(optimizers, self.state['base_lr']):
#                     self.set_lr(opt, self.state['steps']/self.warmup_steps * base)
#             else:
#                 if self.state['base_lr'] is None:
#                     self.state['base_lr'] = optimizers.param_groups[0]['lr']
#                 self.set_lr(optimizers, self.state['steps']/self.warmup_steps * self.
#             self.state['steps'] += 1
```

```
#         elif self.invsqrt:
#             if type(optimizers) == 'list':
#                 if self.state['base_lr'] is None:
#                     self.state['base_lr'] = [o.param_groups[0]['lr'] for o in optimiz
#                 for opt,base in zip(optimizers, self.state['base_lr']):
#                     self.set_lr(opt,base * (self.decay / (self.state['steps'] - self.
#             else:
#                 if self.state['base_lr'] is None:
#                     self.state['base_lr'] = optimizers.param_groups[0]['lr']
#                 self.set_lr(optimizers, self.state['base_lr'] * (
#                             self.decay / (self.state['steps'] - self.warmup_steps + s
#             self.state['steps'] += 1

#     def load_state_dict(self, state_dict):
#         self.state.update(state_dict)

#     def state_dict(self):
#         return self.state.copy()

# def average_models(models):
#     """Averages model weights and loads the resulting weights into the first model, r
#     models = list(models)
#     n = len(models)
#     sds = [m.state_dict() for m in models]
#     averaged = {}
#     for k in sds[0]:
#         averaged[k] = sum(sd[k] for sd in sds) / n
#     models[0].load_state_dict(averaged)
#     return models[0]
```

[ ] Python

```
Running cells with 'Python 3.12.3' requires the ipykernel package.
Run the following command to install 'ipykernel' into the Python environment.
Command: '/opt/homebrew/bin/python3 -m pip install ipykernel -U --user --force-reinstall'
```

Next, we can load the pre-trained model from checkpoint (also included in this repository) and fine-tune the model and test it using the validation set:

```
# path = "https://github.com/srikur/cs_598_final_project/raw/main/model.ckpt"
# for seed in range(2020, 2023):
#     pl.seed_everything(seed)

#     # performs fine-tuning and then averages the models. final_model is the averaged
#     fine_tune_model = duett.fine_tune_model(path, d_static_num=dm.d_static_num(),
#             d_time_series_num=dm.d_time_series_num(), d_target=dm.d_target(), pos_fra
#     checkpoint = pl.callbacks.ModelCheckpoint(save_top_k=5, save_last=False, mode='ma
#     warmup = WarmUpCallback(steps=1000)
#     trainer = pl.Trainer(gpus=0, logger=False, max_epochs=50, gradient_clip_val=1.0,
#             callbacks=[warmup, checkpoint])
#     trainer.fit(fine_tune_model, dm)
#     final_model = average_models([duett.fine_tune_model(path, d_static_num=dm.d_stati
#             d_time_series_num=dm.d_time_series_num(), d_target=dm.d_target(), pos_fra
#             for path in checkpoint.best_k_models.keys()])
```

[ ] ⊗ Python

```
Running cells with 'Python 3.12.3' requires the ipykernel package.
Run the following command to install 'ipykernel' into the Python environment.
Command: '/opt/homebrew/bin/python3 -m pip install ipykernel -U --user --force-reinstall'
```

+ Code + Markdown

```
# # Make predictions on the validation set
# trainer.test(final_model, dataloaders=dm)
```

[ ] ⊗ Python

```
Running cells with 'Python 3.12.3' requires the ipykernel package.
Run the following command to install 'ipykernel' into the Python environment.
Command: '/opt/homebrew/bin/python3 -m pip install ipykernel -U --user --force-reinstall'
```

To train the model and show it's effectiveness on a validation set of data, the following command can be run:

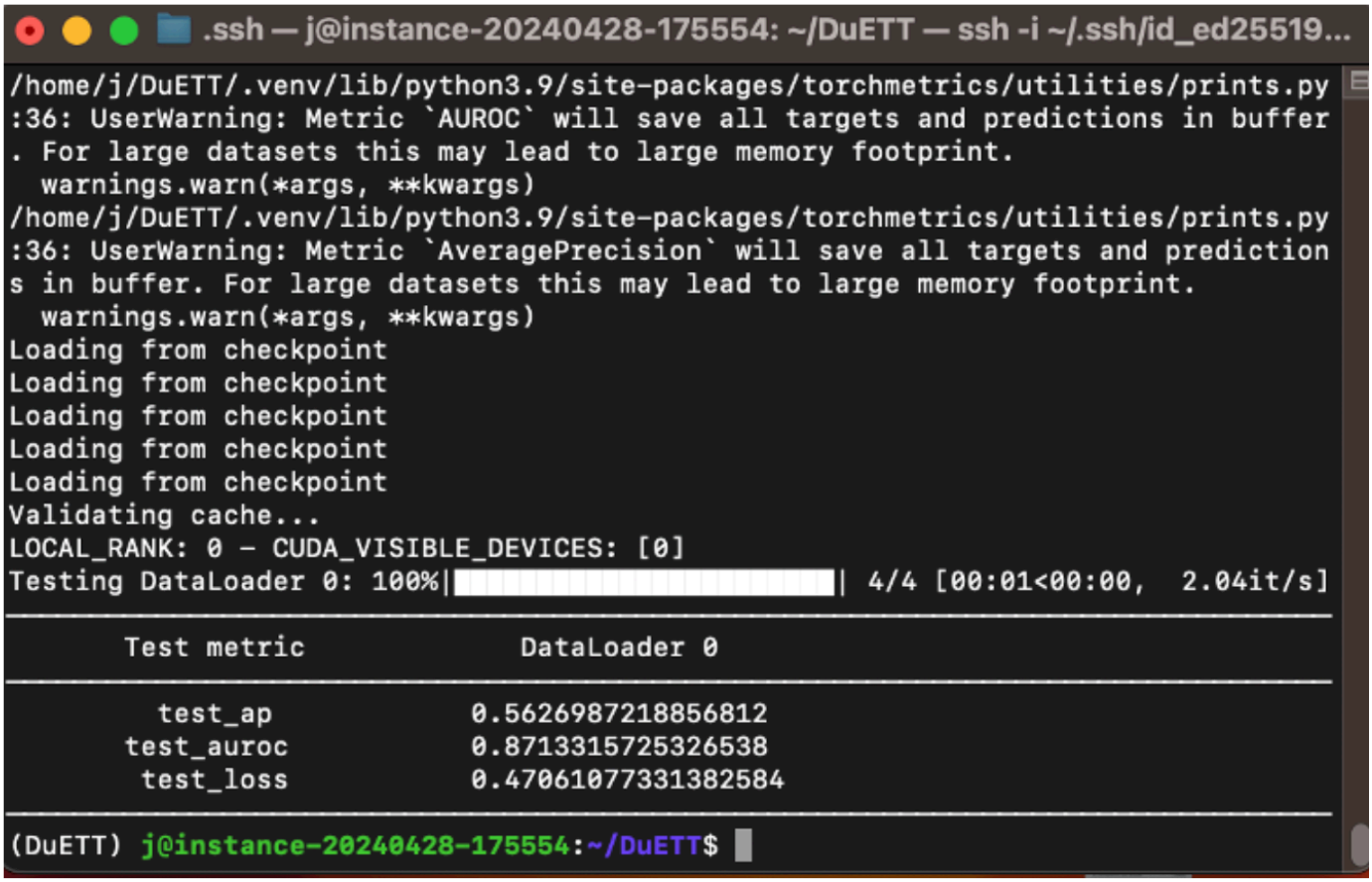
```
# !python3 train.py
```

[ ] ⊗ Python

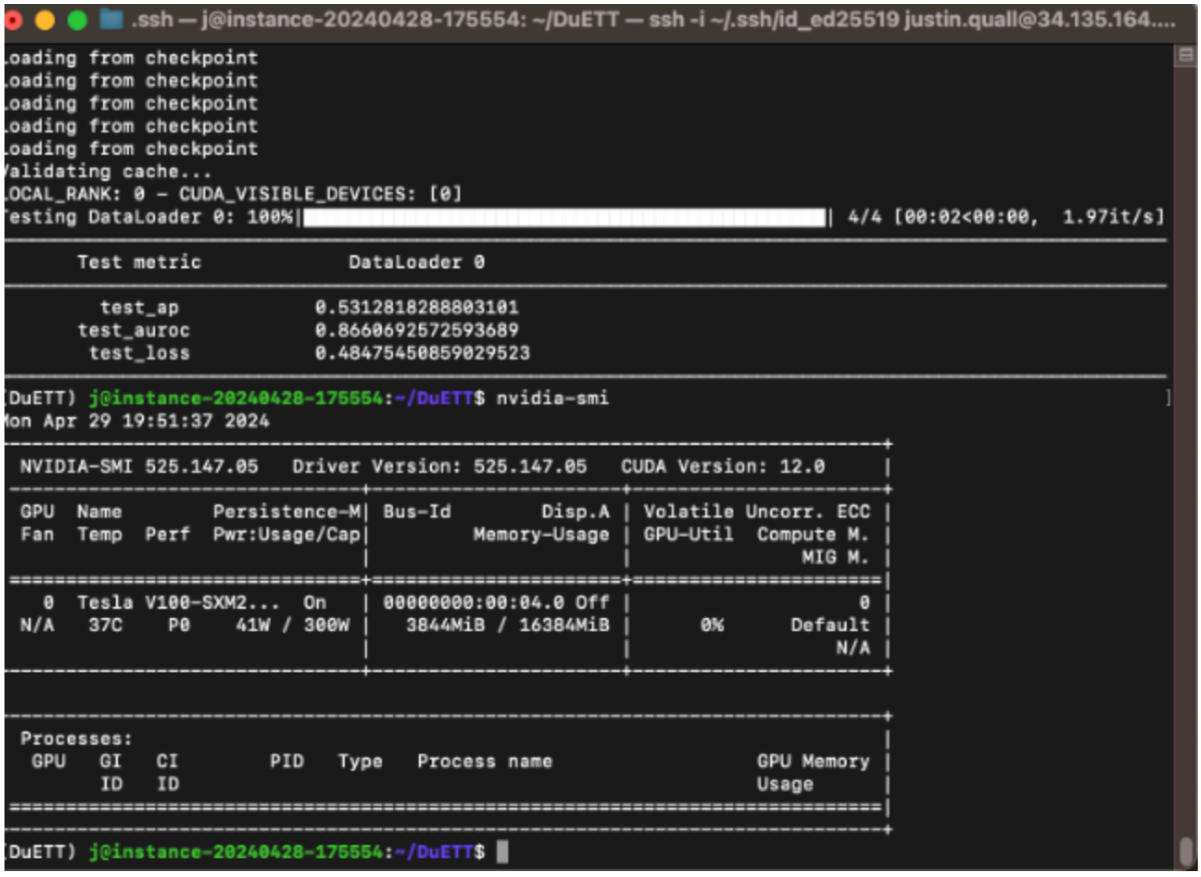
```
Running cells with 'Python 3.12.3' requires the ipykernel package.
Run the following command to install 'ipykernel' into the Python environment.
Command: '/opt/homebrew/bin/python3 -m pip install ipykernel -U --user --force-reinstall'
```

Here are the results from running through the full 300-epoch training and 50-epoch finetuning steps of the original paper, and a discussion of the different hardware configurations used.

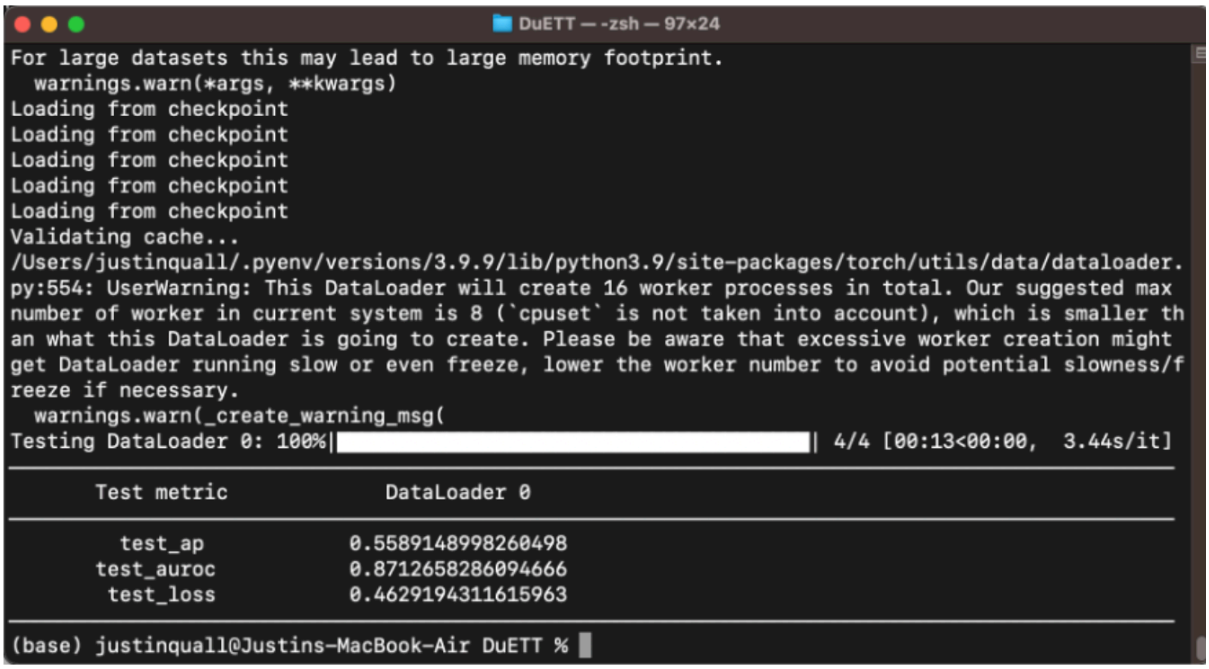
Running the original code on a virtual machine with native GPU acceleration (V100) took approximately 3 hours:



VM specs:



Running the original file on a local machine with MPS hardware acceleration took approximately 24 hours:



Ablation Analysis

We tested a data ablation piece where 20% of the data is omitted for each patient sample. This form of ablation may make sense here because of the paper's use of novel self-supervised learning to augment model training in the context of limited labeled data. This may be relevant to real-world healthcare practice, where EHRs may only cover the scope of one health system (whilst high-acuity patients receive care at multiple health systems on different EHRs/instances). The authors tested various other types of model feature ablations within the original work, for example a single vs. double transformer architecture, self-supervised learning, and input representation.

Here is the code which can added to the original physionet.py to conduct the ablation training:

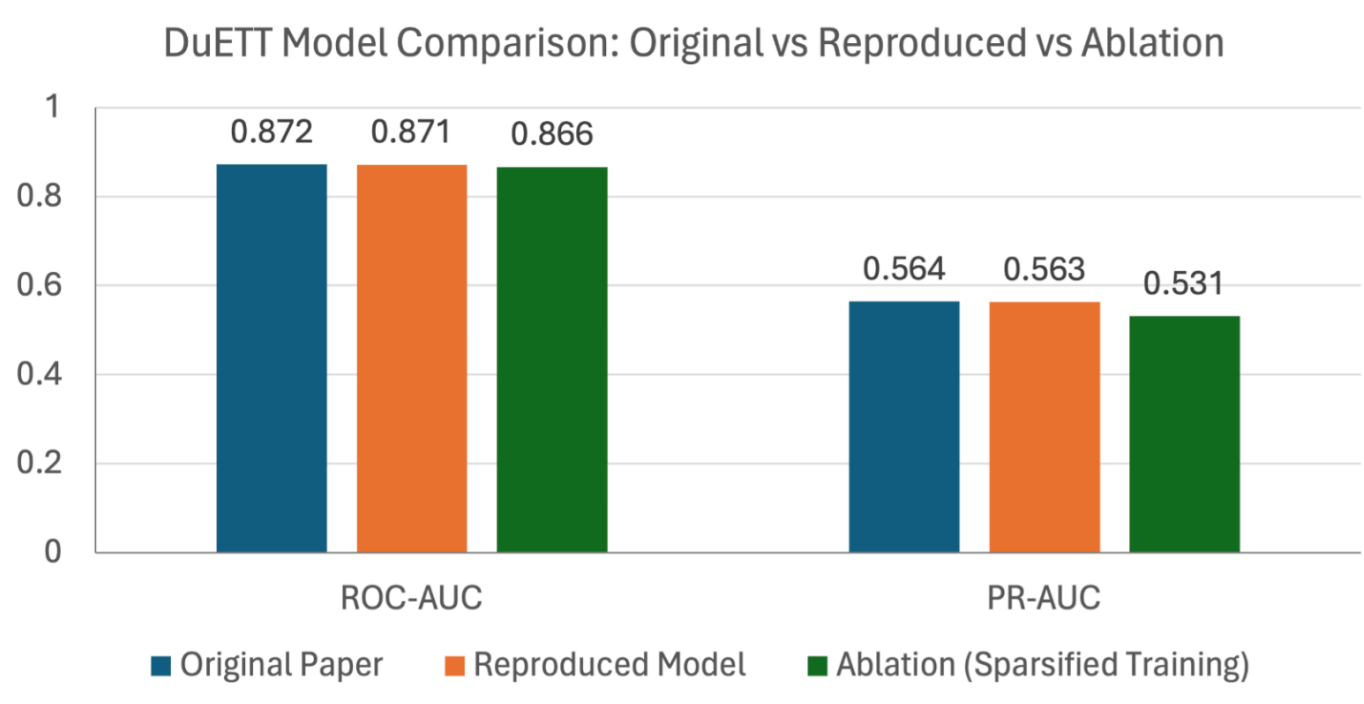
```
#   def setup(self):
#     tt_data = PhysioNet2012(self.split_name, train_prop=0.7, val_prop=0.15, time=False, seed=0)

#     self.X = tt_data.X
#     self.y = tt_data.y

#     # ...

#     # Application of Sparsify ablation data preparation (next 3 lines of code)

#     for i in range(self.X.shape[0]):
#         if self.split_name == 'train':
#             self.sparsify(self.X[i])

#     # ...

#   # Application of Sparsify data prep

#   def sparsify(self, vals):
#       for idx in range(215):
#           if (idx + 1) % 5 == 0 :
#               vals[idx] = torch.full((45,), float('nan'))
```

[ ]

Python

+ Code + Markdown

Model comparison



The paper's original results on the PhysioNet-2012 dataset included a ROC-AUC of 0.872 and a PR-AUC of 0.564. While DuETT model is super neat and thoughtful, subjectively, these results did not seem to be a meaningful improvement over XGBoost (0.865 and 0.531, respectively). With XGBoost's reputation for being a fast and robust model type for working with tabular data, it seems like DuETT may not be be worth the difference in cost and support at this time.

In any case, we were able to reproduce the paper's results on the PhysioNet-2012 dataset very closely. The reproduced model showed a ROC-AUC of 0.871 (within 0.001 of the paper's 0.872) and a PR-AUC of 0.563 (within 0.001 of the paper's 0.563). These results are aligned with the hypothesis of being able to recreate the original paper's results pretty closely. Please find a chart comparing the model resutls below.

Within the original paper, the authors ran an incredibly broad set of ablations, including ablating event- and time-components of the transformer (running on one type only), an ablation analysis of the self-supervised learning ablation, and an input representation ablation. All of these were run using the MIMIC-IV dataset.

We extended the ablation analysis by also considering the impact of reducing the thoroughness of the training set - specifically, removing 20% of the actual observation time bins (every fifth one) for each patient. We run this analysis on the PhysioNet-2012 dataset and include all patients (testing for "sparsity" within each patient) as compared to the paper's data redution analysis which was on MIMIC-IV and may have excluded entire patient records (unclear).

This "sparsity" ablation is a practically meaningful consideration given the model's goal of being effective on sparse datasets and given real-world healthcare applications, where one hospital's electronic health record will often miss significant data corresponding with patients being treated by other healthcare providers (who utilize other EHR instances).

The ablation study's results were quite interesting. The reduction to PR-AUC and ROC-AU were modest, to 0.531 and 0.866 respectively. Those metrics exceed or are close to the "thorough dataset" results of other top deep learning models such as LSTM, mTAND, Raindrop, etc - and were on par with the industry standard (for tabular machine learning) XGBoost.

Further confirmatory analysis would be needed, but this indicates that DuETT may indeed successfully solve for training and inference on sparse, irregular datasets as are observed in real-world EHR data.

Discussion

As shown in the Results section, the paper was confirmed to be reproducible when focusing on application to the PhysioNet-2012 dataset.

During the reproduction, we found that running the code locally was fairly easy once the proper environment was established (for example running the correct version of dependency packages), however maintaining an aligned environment within Google CoLab took some getting used to.

It was also pretty easy to learn the big picture goals and structure of the DuETT model, thanks to a clear and well-written paper by the original authors.

During the reproduction, the three most difficult things to address were hardware limitations, specific dependencies, and developing a robust understanding of the model's input data presentation.

Even when focusing on the PhysioNet-2012 dataset (a fraction of the size of MIMIC-IV), training the sizeable and complex DuETT model required significant hardware capacity. With local GPU acceleration (via PyTorch MPS) on a M1 Silicon Macbook with upgraded memory, training the model took ~22 hours and it was not feasible to iterate and debug. This was solved by renting a virtual machine with a very powerful GPU (Nvidia Tesla V100), which reduced the training time to ~2.5 hours.

Specific dependencies also led to some initial headaches while setting up a proper runtime environment. Because the paper is fairly recent, this was not expected. But perhaps due to how fast the AI field is evolving, we found specific matching of the requirements.txt dependencies was key (as was downgrading the runtime version of Python to 3.9).

Finally, as with most of the models we ran in DLH, it was important to keep in mind the data pre-processing steps and how the tensors were all set up (key thing here being the 8,400 x 215 x 45 dimensions of patients, potential visit time bins, and attributes; respectively). Incorrect early assumptions on this drove a need for debugging issues such as dimension mix-ups and an off-by-one error.

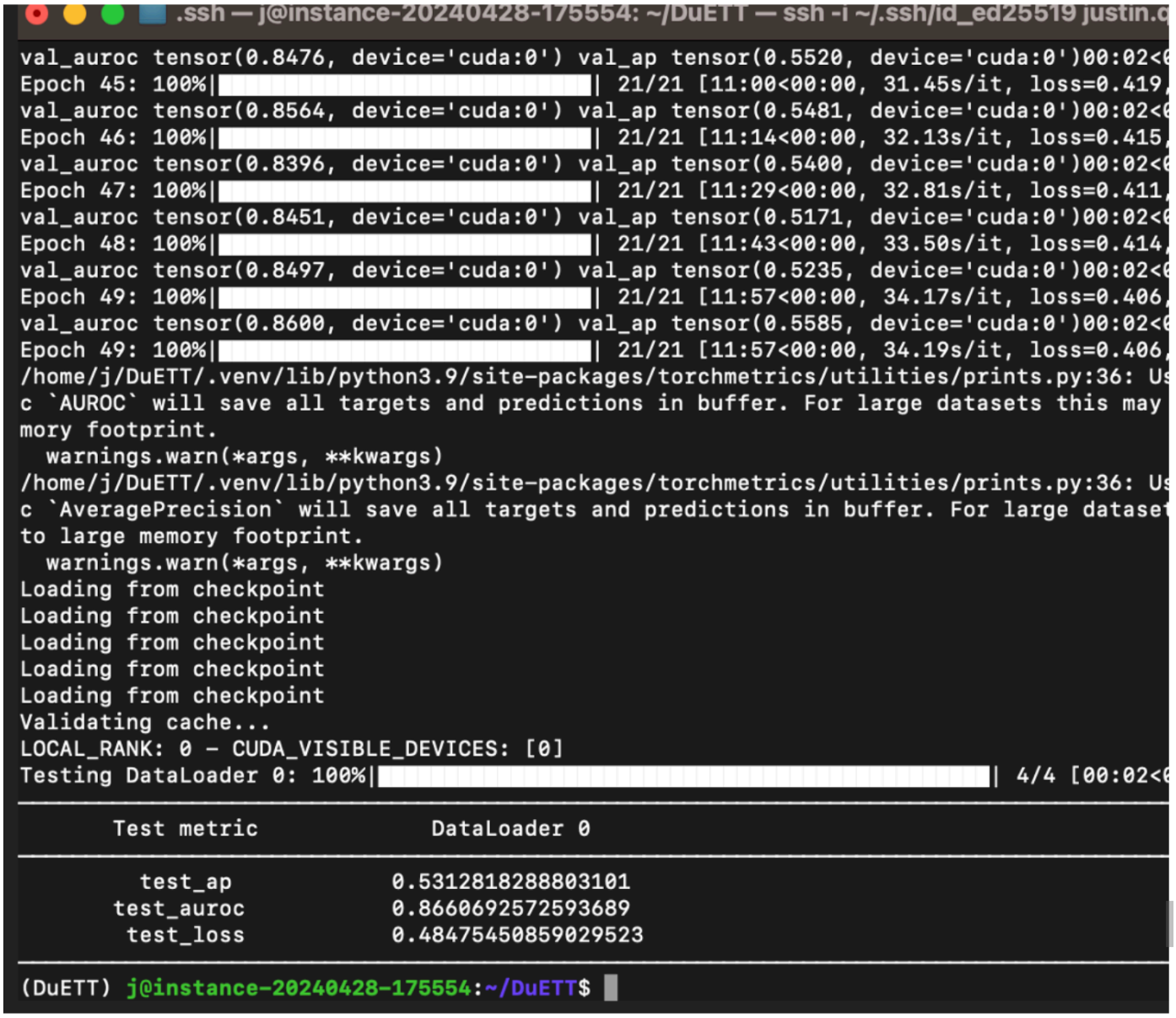
We do not have further suggestions to the authors on how to improve the reproducibility, as they did a great job of setting up a fairly well-documented public repository.

References

1. Alex, Labach., Aslesha, Pokhrel., Xiaolei, Huang., S., Zuberi., Seung, Eun, Yi., Maksims, Volkovs., Tomi, Poutanen., Rahul, G., Krishnan. (2023). DuETT: Dual Event Time Transformer for Electronic Health Records. arXiv.org, abs/2304.13017 doi: 10.48550/arXiv.2304.13017
2. Goldberger, A., Amaral, L., Glass, L., Hausdorff, J., Ivanov, P. C., Mark, R., ... & Stanley, H. E. (2000). PhysioBank, PhysioToolkit, and PhysioNet: Components of a new research resource for complex physiologic signals. Circulation [Online]. 101 (23), pp. e215–e220.
3. Ikaro Silva, George Moody, Roger Mark, and Leo Anthony Celi. Predicting mortality of ICU patients: The PhysioNet/Computing in Cardiology challenge 2012, Jan 2012. URL https://physionet.org/content/challenge-2012/1.0.0/.
4. Johnson, A., Bulgarelli, L., Pollard, T., Horng, S., Celi, L. A., & Mark, R. (2022). MIMIC-IV (version 2.0). PhysioNet. https://doi.org/10.13026/7vcr-e114.

Appendix: Ablation Analysis

Results from running model on "sparsified" training data (removing ~20% of observations [technically time "bins"] for each patient).



Sample illustration of removing ~20% of the data (each patient has 215 potential time bins of 45 observation parameters):