Author

Aryan Kukreja

- *Email*: aryansk2@illinois.edu (mailto:aryansk2@illinois.edu)
- *UIN*: 652936393
- *GitHub Repo*: https://github.com/ABusyProgrammer/CS598-DLH (https://github.com/ABusyProgrammer/CS598-DLH)
- *Video Link*: https://mediaspace.illinois.edu/media/t/1_a9jzgzkm (https://mediaspace.illinois.edu/media/t/1_a9jzgzkm)

Introduction

In the rapidly evolving field of healthcare analytics, Electronic Health Records (EHRs) have emerged as a vital resource. These records, which contain a wealth of patient information, hold the potential to significantly improve patient outcomes and optimize the allocation of healthcare resources. However, the inherent complexity of EHRs, characterized by high sparsity and irregular observations, presents a formidable challenge. Traditional time series analysis methods, designed for densely sampled data, often fall short when applied to EHRs. While state-of-the-art methods such as the various categories and implementations of neural network models and attention mechanisms have shown promising results, they come with their own set of limitations. These methods often necessitate substantial computational resources and involve truncating inputs, which can compromise the accuracy of the predictions.

Addressing these challenges, this paper introduces a groundbreaking approach with the Dual Event Time Transformer (DuETT). DuETT represents a significant departure from traditional methods, offering an innovative architecture that attends to both the time and event type dimensions of EHR data. This unique capability allows DuETT to transform sparse time series into a regular sequence, thereby enabling the application of larger and deeper neural networks. This transformation process, which is at the heart of DuETT's innovation, effectively handles the irregularity and sparsity of EHR data. The effectiveness of DuETT is not just theoretical; it has been empirically demonstrated. DuETT outperforms state-of-the-art deep learning models on multiple downstream tasks using the MIMIC-IV and PhysioNet-2012 EHR datasets. By providing a robust and effective representation of EHR data, DuETT makes a significant contribution to the field of healthcare analytics. Its state-of-the-art performance and potential for practical applications in hospitals underscore the importance and relevance of this research in the ongoing efforts to leverage EHRs for improved healthcare outcomes.

Resources

Paper Being Analyzed: DuETT: Dual Event Time Transformer for Electronic Health Records (https://arxiv.org/pdf/2304.13017.pdf)

Original Code From the Author: DuETT GitHub Repository (https://github.com/layer6ai-labs/DuETT/tree/master)

Scope of Reproducibility:

The following are some of the key hypothesis that this paper looks to test:

1. **Capture EHR Structure**: If DuETT attends over both time and event dimensions of EHR data, then it can produce robust representations that capture the structure of EHR data.
2. **Handle Sparsity and Irregularity**: If DuETT transforms sparse and irregularly sampled time series into regular sequences with fixed length, then it can reduce computational complexity and handle the sparsity and irregularity of EHR data.
3. **Improve Model Performance**: If DuETT is applied to multiple downstream tasks using the MIMIC-IV and PhysioNet-2012 EHR datasets, then it can outperform state-of-the-art deep learning models.
4. **Leverage Self-Supervised Learning**: If DuETT utilizes self-supervised prediction tasks for model pre-training, then it can enable the training of larger models with limited labeled data.

These hypotheses form the basis of the paper's investigation into the effectiveness of the DuETT architecture for modeling EHR data. Each hypothesis is designed to test a specific aspect of DuETT's capabilities and its potential advantages over existing methods.

Methodology

The following section has a step-by-step implementation of the given paper. Section headers are named based on the grading **rubric** (https://docs.google.com/document/d/1ftHUFl_eeZNfRYLNI0jh-v8tvkQfvSz-q8jzt2026k8/edit#heading=h.gjdgxs) provided in Piazza to make it easier to follow along.

Environment

Although this notebook is shared via Google Collab, the actual training, testing and validation of the implementation code was performed under an AWS SageMaker instance. This is because the training required a more powerful processor with larger memory, and I was only able to get a more powerful machine via AWS (I work at Amazon, so its free for me).

Specifically, the hardware that worked for me was the `ml.r7i.2xlarge` ML processor provided by Amazon SageMaker.

Required Packages

These are the packages sourced from the DuETT project's requirements.txt (https://github.com/layer6ai-labs/DuETT/blob/master/requirements.txt) file. That file's versions are compatible specifically with Python 3.8.5; to ensure that this project is runnable across any

Python version, I've removed the specific versions from the pip installation instructions below; this resulted in a need to refactor portions of the code.

Python 3.10.3 (the latest currently available in AWS Sagemaker) was used for my project.

In [1]:
```
# !pip freeze > reqs.txt
# !cat reqs.txt | xargs -n 1 pip uninstall -y

!pip install torch
!pip install numpy
!pip install pytorch-lightning
!pip install torch
!pip install torchaudio
!pip install torchvision
!pip install x-transformers
!pip install torchtime
!pip install torchmetrics
```

...

Import Installed Dependencies

Once the above packages are installed, we need to import the relevant dependencies. I've collected all imports across the various `.py` files in the DuETT () project into a single code-block for easier reading, and have annotated them as well.

In [58]:
```
# Standard library imports
from multiprocessing import Manager
import argparse

# Third-party library imports for numerical operations
import numpy as np

# Core PyTorch packages for neural network operations and multiprocess
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.multiprocessing
torch.multiprocessing.set_sharing_strategy('file_system')

# PyTorch utility for data loading
from torch.utils.data import DataLoader

# PyTorch Lightning is a high-level wrapper for PyTorch that aids in o
import pytorch_lightning as pl

# Torchmetrics is a PyTorch library for various machine learning metri
import torchmetrics

# X-Transformers is a PyTorch-based library for transformer models
import x_transformers

# Torchtime is a PyTorch-based library for time series analysis. The d
from torchtime.data import PhysioNet2012
```

Data

For this notebook, the dataset was already implemented in the code block from the above section, as:

```
from torchtime.data import PhysioNet2012
```

This will be used in the below data-model.

Physionet2012 Data-Set

For this paper, we rely on the dataset used from the Physionet in Cardiology Challenge (https://physionet.org/content/challenge-2012/1.0.0/) from 2012, also abbreviated `Physionet2012`. The dataset contains data from ~8,000 patients spread across ~12,000 ICU stays due to various cardiac-related root-causes.

Compared to the main MIMIC-IV data-set that has data spanning 2008 to 2019, this dataset is much smaller, and it has been selected by the authors of this paper for a reason: real-life electronic health-record (EHR) applications have far smaller labelled data-sets. Given that the objective of this paper is to leverage self-supervised learning (SSL) to train models using limited-sized data-sets, relying on Physionet2012 as a data-set suits the approach of this paper.

The following are the entries of each record:

1. **General Descriptors**: Collected at the time of admission, these include data such as *Age*, *Gender*, and *Weight*; all of which tie back to identifying the patient who was admitted into the ICU. A total of 6 such fields exist.
2. **Time-Series Data**. Recorded in the first 48 hours since admission, these are a set of 42 descriptors. Of these 42, at least 1 of them must be defined per record along with a `timestamp` defining exactly when the observation was made. For the same patient's same visit to the ICU, we can have multiple readings of a given time-series descriptor across different timestamps.
3. *__Outcome-Related Descriptors__: These are descriptors that record the outcome of the patient's visit to the ICU, after their stay is complete. It includes 6 variables, such as *length of stay*, *survival*, and *record ID*.

Data Model

The first step is to define the model of the data that will be used throughout the training process. these classes serve as wrappers over the core `Physionet2012` class that is imported to retrieve all the data that will be used for testing.

In [59]:
```
class PhysioNetDataset(torch.utils.data.Dataset):
    """A PyTorch Dataset for the PhysioNet 2012 data."""

    def __init__(self, split_name, n_timesteps=32, use_temp_cache=Fals
        """Initialize the dataset with given parameters."""
        self.split_name = split_name
        self.n_timesteps = n_timesteps
        self.temp_cache = Manager().dict() if use_temp_cache else None

    def setup(self):
        """Prepare the data for the dataset."""
        # Load the data
        tt_data = PhysioNet2012(self.split_name, train_prop=0.7, val_p

        # Split the data into features and labels
        self.X, self.y = tt_data.X, tt_data.y

        # Calculate the statistics of the features
        self.means, self.stds, self.maxes, self.mins = [], [], [], []
        for i in range(self.X.shape[2]):
            vals = self.X[:, :, i].flatten()
            vals = vals[~torch.isnan(vals)]
            self.means.append(vals.mean())
            self.stds.append(vals.std())
            self.maxes.append(vals.max())
            self.mins.append(vals.min())

    def __len__(self):
        """Return the total number of samples in the dataset."""
        return self.X.shape[0]

    def __getitem__(self, i):
        """Get a sample from the dataset."""
        # If the sample is in the cache, return it
        if self.temp_cache is not None and i in self.temp_cache:
            return self.temp_cache[i]

        # Prepare the input features
        ins = self.X[i, ~torch.isnan(self.X[i, :, 0]), :]
        time = ins[:, 0] / 60 / 24
        x_static = torch.zeros(self.d_static_num())
        x_ts = torch.zeros((self.n_timesteps, self.d_time_series_num()

        # Process the time-series data
        for i_t, t in enumerate(time):
            bin = self.n_timesteps - 1 if t == time[-1] else int(t / t
            for i_ts in range(1, 37):
                x_i = ins[i_t, i_ts]
                if not torch.isnan(x_i).item():
                    x_ts[bin, i_ts - 1] = (x_i - self.means[i_ts]) / (
                    x_ts[bin, i_ts - 1 + self.d_time_series_num()] +=

        # Process the static data
        bin_ends = torch.arange(1, self.n_timesteps + 1) / self.n_time
        for i_tab in range(37, 45):
```

```
            x_i = ins[0, i_tab]
            x_i = (x_i - self.means[i_tab]) / (self.stds[i_tab] + 1e-7
            x_static[i_tab - 37] = x_i.nan_to_num(0.)

        # Prepare the final input and output data
        x = (x_ts, x_static, bin_ends)
        y = self.y[i, 0]

        # Cache the data if needed
        if self.temp_cache is not None:
            self.temp_cache[i] = (x, y)

        return x, y

    def d_static_num(self):
        """Return the total dimension of numeric static features."""
        return 8

    def d_time_series_num(self):
        """Return the total dimension of numeric time-series features.
        return 36

    def d_target(self):
        """Return the dimension of the target variable."""
        return 1

    def pos_frac(self):
        """Return the fraction of positive samples in the dataset."""
        return self.y.mean().numpy()
```

Collation Function

This is just a helper method that is used to zip data into sequences. It is used in the Dataloader that is eventually used for training.

In [60]:
```
# Function to collate data into sequences
def collate_into_seqs(batch):
    xs, ys = zip(*batch)
    return zip(*xs), ys
```

Data Module

This is the actual data module that builds atop the dataset class. In addition to the `PhysionetDataset` class, it adds dats setup functionality for the training process, as well as functions to log training progress.

In [61]:
```
# PyTorch Lightning DataModule for PhysioNet data
class PhysioNetDataModule(pl.LightningDataModule):
    def __init__(self, use_temp_cache=False, batch_size=8, num_workers
        """Initialize the data module with given parameters."""
        self.use_temp_cache = use_temp_cache
        self.batch_size = batch_size
        self.num_workers = num_workers
        self.prefetch_factor = prefetch_factor

        # Create datasets for training, validation, and testing
        self.ds_train = PhysioNetDataset('train', use_temp_cache=use_t
        self.ds_val = PhysioNetDataset('val', use_temp_cache=use_temp_
        self.ds_test = PhysioNetDataset('test', use_temp_cache=use_tem

        self.prepare_data_per_node = False
        self.allow_zero_length_dataloader_with_multiple_devices: bool

        # Arguments for the data loader
        self.dl_args = {
            'batch_size': self.batch_size,
            'prefetch_factor': self.prefetch_factor,
            'collate_fn': collate_into_seqs,
            'num_workers': num_workers
        }

    def setup(self, stage=None):
        """Prepare the data for the given stage."""
        if stage is None:
            # If no stage is specified, setup data for all stages
            self.ds_train.setup()
            self.ds_val.setup()
            self.ds_test.setup()
        elif stage == 'fit':
            # If the stage is 'fit', setup data for training and valid
            self.ds_train.setup()
            self.ds_val.setup()
        elif stage == 'validate':
            # If the stage is 'validate', setup data for validation
            self.ds_val.setup()
        elif stage == 'test':
            # If the stage is 'test', setup data for testing
            self.ds_test.setup()

    def prepare_data(self):
        """Prepare the data. This method is intentionally left empty."
        pass

    def _log_hyperparams(self):
        """Log hyperparameters. This method is intentionally left empt
        pass

    def train_dataloader(self):
        """Return a data loader for the training data."""
        return DataLoader(self.ds_train, shuffle=True, **self.dl_args)
```

```
    def val_dataloader(self):
        """Return a data loader for the validation data."""
        return DataLoader(self.ds_val, **self.dl_args)

    def test_dataloader(self):
        """Return a data loader for the testing data."""
        return DataLoader(self.ds_test, **self.dl_args)

    def d_static_num(self):
        """Return the total dimension of numeric static features."""
        return self.ds_train.d_static_num()

    def d_time_series_num(self):
        """Return the total dimension of numeric time-series features.
        return self.ds_train.d_time_series_num()

    def d_target(self):
        """Return the dimension of the target variable."""
        return self.ds_train.d_target()

    def pos_frac(self):
        """Return the fraction of positive samples in the dataset."""
        return self.ds_train.pos_frac()
```

Model

With the necessary packages installed and set up, the implementation of this project can begin. The below sections define the structure of the model, including the multi-layer perceptron (MLP) which will be used as an **Embedding Layer** throughout the model.

Model Structures

There is no pre-trained model; the model is defined from scratch (down to the multi-layer perceptron) within this code. At a high-level, the following is the model definition:

1. **Special Embeddings**: This is an embedding layer for special timesteps, such as masked, static, [CLS], etc. It's defined as `nn.Embedding(8, d_embedding)`.
2. **Embedding Layers**: These are the embedding layers for each time series. They are defined using a list comprehension to create a `nn.ModuleList` of `simple_mlp` layers.
3. **Observation Embedding**: This is an embedding layer for observations, defined as `nn.Embedding(16, 1)`.
4. **Event Transformers**: These are transformer layers specifically for events. They are defined using a list comprehension to create a `nn.ModuleList` of `x_transformers.Encoder` layers.
5. **Full Event Embedding**: This is an embedding layer for the full event, defined as `nn.Embedding(d_time_series_num + 1, et_dim)`.
6. **Time Transformers**: These are transformer layers specifically for time. They are defined using a list comprehension to create a `nn.ModuleList` of `x_transformers.Encoder` layers.
7. **Full Time Embedding**: This is an embedding layer for the full time, defined as

`self.cve(batch_norm=True, d_embedding=tt_dim)`.

8. **Full Representation Embedding**: This is an embedding layer for the full representation, defined as `nn.Embedding(tt_dim, 1)`.
9. **Head for Prediction**: This is the final prediction layer, defined as a `simple_mlp` layer.
10. **Pretraining Value Projection**: This is a projection layer for pretraining values, defined as a `simple_mlp` layer.
11. **Pretraining Presence Projection**: This is a projection layer for pretraining presence, defined as a `simple_mlp` layer if `self.pretrain_presence` is True.
12. **Event Prediction Projection**: This is a projection layer for event prediction, defined as a `simple_mlp` layer if `self.predict_events` is True.

Resources

1. Paper Being Analyzed: DuETT: Dual Event Time Transformer for Electronic Health Records (https://arxiv.org/pdf/2304.13017.pdf)
2. Original Code From the Author: DuETT GitHub Repository (https://github.com/layer6ai-labs/DuETT/tree/master)

Defining the Multi-Layer Perceptron

This is where we define the mulit-layer perceptron that is used as the embedding layers of the final model that will be tested throughout this code. The last dimension of the MLP tensor has batch-normalization applied to it.

In [62]:
```
class BatchNormLastDim(nn.Module):
    """A PyTorch Module for applying Batch Normalization to the last d

    def __init__(self, d, **kwargs):
        """Initialize the module with given parameters."""
        super().__init__()
        # Create a 1D BatchNorm layer with 'd' features
        self.batch_norm = nn.BatchNorm1d(d, **kwargs)

    def forward(self, x):
        """Apply the BatchNorm layer to the input tensor."""
        match x.ndim:
            case 2:
                # If the input is a 2D tensor, apply BatchNorm directl
                return self.batch_norm(x)
            case 3:
                # If the input is a 3D tensor, transpose the last two
                return self.batch_norm(x.transpose(1, 2)).transpose(1,
            case _:
                # If the input is not a 2D or 3D tensor, raise an erro
                raise NotImplementedError("BatchNormLastDim not implem
```

In [63]:
```
def simple_mlp(d_in, d_out, n_hidden, d_hidden, final_activation=False
               hidden_batch_norm=False, dropout=0., activation=nn.ReLU
    """A simple Multi-Layer Perceptron (MLP) implementation in PyTorch

    # Initialize the list of layers
    layers = []

    # If there are no hidden layers, create a single linear layer
    if n_hidden == 0:
        if input_batch_norm:
            layers.append(BatchNormLastDim(d_in))
        layers.append(nn.Linear(d_in, d_out))
    else:
        # If there are hidden layers, create them with optional batch
        if input_batch_norm:
            layers.append(BatchNormLastDim(d_in))
        layers.extend([nn.Linear(d_in, d_hidden), activation(), nn.Dro

        for _ in range(n_hidden - 1):
            if hidden_batch_norm:
                layers.append(BatchNormLastDim(d_hidden))
            layers.extend([nn.Linear(d_hidden, d_hidden), activation()

        if hidden_batch_norm:
            layers.append(BatchNormLastDim(d_hidden))
        layers.append(nn.Linear(d_hidden, d_out))

    # If final activation is required, add it to the layers
    if final_activation:
        layers.append(activation())

    # Return the MLP as a sequential model
    return nn.Sequential(*layers)
```

Defining the Model

This is where we define the actual model itself. The components defined above (the `PhysionetDataModule`, `simple_mlp`, etc...) will be applied. This is the main model class that defines the key training functions/methods such as `forward()`. It is a PyTorch Lightning module designed for training and evaluating a machine learning model for time-series data analysis that is being conducted here to eveluate the DuETT approach.

In [64]:
```
class Model(pl.LightningModule):
    """A PyTorch Lightning Module for a transformer-based model."""

    def __init__(self, d_static_num, d_time_series_num, d_target, lr=3
                 scalenorm=True, n_hidden_mlp_embedding=1, d_hidden_ml
                 d_feedforward=512, max_len=48, n_transformer_head=2,
                 d_hidden_tab_encoder=128, n_hidden_tab_encoder=1, nor
                 fusion_method='masked_embed', n_hidden_head=1, d_hidd
                 aug_mask=0., pretrain=True, pretrain_masked_steps=1,
                 pretrain_d_hidden=64, pretrain_dropout=0.5, pretrain_
                 pretrain_presence=True, pretrain_presence_weight=0.2,
                 transformer_dropout=0., pos_frac=None, freeze_encoder
                 save_representation=None, masked_transform_timesteps=
        """Initialize the model with given parameters."""
        super().__init__()

        # Set up hyperparameters
        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.validation_step_outputs = []

        # Register buffers for multi-GPU training
        self.register_buffer("MASKED_EMBEDDING_KEY", torch.tensor(0))
        self.register_buffer("REPRESENTATION_EMBEDDING_KEY", torch.ten

        # Set up special embeddings for any special timesteps, e.g., m
        self.special_embeddings = nn.Embedding(8, d_embedding)

        # Set up embedding layers
        self.embedding_layers = nn.ModuleList([
            simple_mlp(2, d_embedding, n_hidden_mlp_embedding, d_hidde
            for _ in range(d_time_series_num)])

        # Set up observation embedding
        self.n_obs_embedding = nn.Embedding(16, 1)
```

```
        # Set up feedforward dimension if not provided
        if d_feedforward is None:
            d_feedforward = d_embedding * 4

        # Set up event transformer dimensions
        et_dim = d_embedding * (masked_transform_timesteps + 1)
        tt_dim = d_embedding * (d_time_series_num + 1)

        # Set up event transformers
        self.event_transformers = nn.ModuleList([x_transformers.Encode
                heads=n_transformer_head, pre_norm=norm_first, use_sca
                attn_dim_head=d_embedding // n_transformer_head, ff_gl
                ff_mult=d_feedforward / et_dim, attn_dropout=transform
                ff_dropout=transformer_dropout) for _ in range(n_duett

        # Set up full event embedding
        self.full_event_embedding = nn.Embedding(d_time_series_num + 1

        # Set up time transformers
        self.time_transformers = nn.ModuleList([x_transformers.Encoder
                heads=n_transformer_head, pre_norm=norm_first, use_sca
                attn_dim_head=d_embedding // n_transformer_head, ff_gl
                ff_mult=d_feedforward / tt_dim, attn_dropout=transform
                ff_dropout=transformer_dropout) for _ in range(n_duett

        # Set up full time embedding
        self.full_time_embedding =  self.cve(batch_norm=True, d_embedd

        # Set up full representation embedding
        self.full_rep_embedding = nn.Embedding(tt_dim, 1)

        # Set up representation dimension
        d_representation = d_embedding * (d_time_series_num + 1)  # ti

        # Set up head for prediction
        self.head = simple_mlp(d_representation, d_target, n_hidden_he
                               hidden_batch_norm=True, final_activatio

        # Set up pretraining value projection
        self.pretrain_value_proj = simple_mlp(d_representation, d_time
                                              pretrain_n_hidden, pretr

        # Set up pretraining presence projection if needed
        if self.pretrain_presence:
            self.pretrain_presence_proj = simple_mlp(d_representation,
                                                     pretrain_n_hidden

        # Set up event prediction projection if needed
        if self.predict_events:
            self.predict_events_proj = simple_mlp(et_dim, masked_trans
                                                  pretrain_n_hidden, p
            if self.pretrain_presence:
                self.predict_events_presence_proj = simple_mlp(et_dim,
                                                               pretrai

        # Set up tabular encoder
```

```
        self.tab_encoder = simple_mlp(d_static_num, d_embedding, n_hid
                                      d_hidden_tab_encoder, hidden_bat

        # Set up loss functions
        self.pretrain_loss = F.mse_loss
        self.loss_function = F.binary_cross_entropy_with_logits
        self.pretrain_presence_loss = F.binary_cross_entropy_with_logi

        # Set up metrics
        num_classes = None if d_target == 1 else d_target
        task = 'binary' if d_target == 1 else 'multiclass'
        self.train_auroc = torchmetrics.AUROC(num_classes=num_classes,
        self.val_auroc = torchmetrics.AUROC(num_classes=num_classes, t
        self.train_ap = torchmetrics.AveragePrecision(num_classes=num_
        self.val_ap = torchmetrics.AveragePrecision(num_classes=num_cl
        self.test_auroc = torchmetrics.AUROC(num_classes=num_classes,
        self.test_ap = torchmetrics.AveragePrecision(num_classes=num_c

    def set_pos_frac(self, pos_frac):
        """Set the fraction of positive samples in the dataset."""
        if type(pos_frac) == list:
            pos_frac = torch.tensor(pos_frac, device=torch.device('cud
        self.pos_frac = pos_frac
        if pos_frac is not 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):
        """Create a simple MLP with a single hidden layer and optional
        if d_embedding is 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(), Ba
        return nn.Sequential(nn.Linear(1, d_hidden), nn.Tanh(), nn.Lin

    def feats_to_input(self, x, batch_size, limits=None):
        """Prepare the input features for the model."""
        xs_ts, xs_static, times = x
        xs_ts = list(xs_ts)

        # Process each time series in the batch
        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:]
            # Apply augmentation if needed
            if self.training and self.aug_noise > 0 and not self.pretr
                f[:, :n_vars] += self.aug_noise * torch.randn_like(f[:
            f = torch.cat((f, torch.zeros_like(f[:, :1])), dim=1)
            if self.training and self.aug_mask > 0 and not self.pretra
                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 the time series to the same length
        pad_to = np.max(n_timesteps)
        xs_ts = torch.stack([F.pad(t, (0, 0, 0, pad_to - t.shape[0]))
        xs_times = torch.stack([F.pad(t, (0, pad_to - t.shape[0])) for
        xs_static = torch.stack(xs_static).to(self.device)

        # Apply noise augmentation to the static features if needed
        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):
        """Prepare a batch for pretraining."""
        xs_static, xs_ts, xs_times, n_timesteps = self.feats_to_input(
        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.p
            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_tim
                y_events.append(xs_ts[batch_i, :, event_mask_i])
                y_events_mask.append(xs_ts[batch_i, :, event_mask_i +
                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.pretra
            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
            else:
                keep = torch.logical_or(1 - y_ts_masks, keep)
            keep = torch.cat((keep.tile(1, 2), torch.ones((batch_size,
            xs_ts_clipped *= torch.logical_or(keep.unsqueeze(1), xs_ts
        return (xs_static, xs_ts_clipped, xs_times, n_timesteps), y_ts

    def forward(self, x, pretrain=False, representation=False):
        """
        Forward run
        :param x: input to the model
        :return: prediction output (i.e., class probabilities vector)
        """
        # Unpack the input data
        xs_static, xs_feats, xs_times, n_timesteps = x

        # Determine the number of variables in the time series data
        n_vars = xs_feats.shape[2] // 2

        # If event prediction is enabled, create a mask for the events
        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(
            event_mask_inds = torch.cat((event_mask_inds, event_mask_i

        # Convert the number of observations to integers and clip to t
        n_obs_inds = xs_feats[:, :, n_vars:n_vars*2].to(int).clip(0, s

        # Replace the number of observations in the features with thei
        xs_feats[:, :, n_vars:n_vars*2] = self.n_obs_embedding(n_obs_i

        # Prepare the input for the embedding layers
        embedding_layer_input = torch.empty(xs_feats.shape[:-1] + (n_v
        embedding_layer_input[:, :, :, 0] = xs_feats[:, :, :n_vars]
        embedding_layer_input[:, :, :, 1] = xs_feats[:, :, n_vars:n_va

        # Initialize the output tensor for the embeddings
        psi = torch.zeros((xs_feats.shape[0], xs_feats.shape[1]+1, n_v

        # Apply each embedding layer to its corresponding input featur
        for i, el in enumerate(self.embedding_layers):
            psi[:, :-1, i, :] = el(embedding_layer_input[:, :, i, :])

        # Apply the tabular encoder to the static features
        psi[:, :-1, -1, :] = self.tab_encoder(xs_static).unsqueeze(1)

        # Add the special representation embedding to the last time st
        psi[:, -1, :, :] = self.special_embeddings(self.REPRESENTATION

        # Create a mask for the special masked embedding
        mask_inds = torch.cat((xs_feats[:, :, -1] == 1, torch.zeros((x

        # Apply the special masked embedding to the masked indices
        psi[mask_inds, :, :] = self.special_embeddings(self.MASKED_EMB
```

```
        # If event prediction is enabled, apply the special masked emb
        if self.predict_events:
            psi[event_mask_inds, :] = self.special_embeddings(self.MAS

        # Create the full time embeddings
        time_embeddings = self.full_time_embedding(xs_times.unsqueeze(
        time_embeddings = torch.cat((time_embeddings,
            self.full_rep_embedding.weight.T.unsqueeze(0).expand(xs_fe
            dim=1)

        # Apply each transformer layer to the embeddings
        for layer_i, (event_transformer, time_transformer) in enumerat
            et_out_shape = (psi.shape[0], psi.shape[2], psi.shape[1],
            embeddings = psi.transpose(1, 2).flatten(2) + self.full_ev
            event_outs = event_transformer(embeddings).view(et_out_sha
            tt_out_shape = event_outs.shape
            embeddings = event_outs.flatten(2) + time_embeddings
            psi = time_transformer(embeddings).view(tt_out_shape)

        # Flatten the last two dimensions of the transformed embedding
        transformed = psi.flatten(2)

        # Determine the method for fusing the embeddings
        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), val
                z_ts = []
                for i in range(transformed.shape[0]):
                    z_ts.append(F.pad(transformed[i, masked_ind[i], :]
                z_ts = torch.stack(z_ts)  # batch size x pretrain_mask
            else:
                masked_ind = xs_feats[:, :, -1:]
                z_ts = []
                for i in range(transformed.shape[0]):
                    z_ts.append(transformed[i, torch.nonzero(masked_in
                z_ts = torch.cat(z_ts, dim=0).squeeze()
        elif self.fusion_method == 'averaging':
            z_ts = torch.mean(transformed[:, :-1, :], dim=1)

        # Set the final embeddings
        z = z_ts

        # If only the representation is needed, return it
        if representation:
            return z

        # If pretraining is enabled, prepare the pretraining outputs
        if pretrain:
            rep_token_head = torch.tile(transformed[:, 0, :].unsqueeze
            y_hat_presence = self.pretrain_presence_proj(z).squeeze()
            y_hat_value = self.pretrain_value_proj(z).squeeze(1) if se
            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(
                z_events = torch.stack(z_events)
                y_hat_events = self.predict_events_proj(z_events).sque
                y_hat_events_presence = self.predict_events_presence_p
            return y_hat_value, y_hat_presence, y_hat_events, y_hat_ev

        # If pretraining is not enabled, apply the head to the embeddi
        out = self.head(z).squeeze(1)

        # If the representation is to be saved, return it along with t
        if self.save_representation:
            return out, z
        else:
            return out

    def configure_optimizers(self):
        """Configure the optimizer for the model."""
        optimizers = [torch.optim.AdamW([p for l in self.modules() for
                                         lr=self.lr, weight_decay=self.
        return optimizers

    def training_step(self, batch, batch_idx):
        """Perform a training step."""
        # Unpack the batch
        x, y = batch
        y = torch.tensor(y, dtype=torch.float64, device=self.device)
        batch_size = y.shape[0]

        # If pretraining is enabled, prepare the pretraining outputs
        if self.pretrain:
            x_pretrain, y, mask, y_events, y_events_mask = self.pretra
            y_hat_value, y_hat_presence, y_hat_events, y_hat_events_pr

            # Calculate the pretraining loss
            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] *
                    loss /= self.pretrain_masked_steps
                else:
                    loss = self.pretrain_loss(y_hat_value * mask, y *
            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
                    presence_loss /= self.pretrain_masked_steps
                else:
                    presence_loss = self.pretrain_presence_loss(y_hat_
                loss += presence_loss
            if self.predict_events:
                if self.pretrain_value:
                    loss += self.pretrain_loss(y_hat_events * y_events
```

```
                if self.pretrain_presence:
                    loss += self.pretrain_presence_loss(y_hat_events_p
            else:
                # If pretraining is not enabled, calculate the loss normal
                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_
                    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))

            # Log the training loss
            self.log('train_loss', loss, sync_dist=True)
            return loss

        def validation_step(self, batch, batch_idx):
            """Perform a validation step."""
            # Unpack the batch
            x, y = batch
            y = torch.tensor(y, dtype=torch.float64, device=self.device)
            batch_size = y.shape[0]

            # If pretraining is enabled, prepare the pretraining outputs
            if self.pretrain:
                x_pretrain, y, mask, y_events, y_events_mask = self.pretra
                y_hat_value, y_hat_presence, y_hat_events, y_hat_events_pr

                # Calculate the pretraining loss
                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] *
                        loss /= self.pretrain_masked_steps
                    else:
                        loss = self.pretrain_loss(y_hat_value * mask, y *
                    self.log('val_next_loss', loss, on_epoch=True, sync_di
                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
                        presence_loss /= self.pretrain_masked_steps
                    else:
                        presence_loss = self.pretrain_presence_loss(y_hat_
                    self.log('val_presence_loss', presence_loss, on_epoch=
                    loss += presence_loss
                if self.predict_events:
                    event_loss = self.pretrain_loss(y_hat_events * y_event
                    self.log('val_event_loss', event_loss, on_epoch=True,
                    loss += event_loss
                self.validation_step_outputs.append(loss)
            else:
                # If pretraining is not enabled, calculate the loss normal
```

```
            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_
                loss = self.loss_function(y_hat, y, weight)
            else:
                loss = self.loss_function(y_hat, y)
            self.validation_step_outputs.append(loss)
            self.val_auroc.update(y_hat, y.to(int).to(self.device))
            self.val_ap.update(y_hat, y.to(int).to(self.device))

        # Log the validation metrics
        if not self.pretrain:
            self.log('val_ap', self.val_ap, on_epoch=True, sync_dist=T
            self.log('val_auroc', self.val_auroc, on_epoch=True, sync_
        self.log('val_loss', loss, on_epoch=True, sync_dist=True, prog

    # This method is called at the end of each training epoch
    def on_train_epoch_end(self):
        # If not in pretraining mode, log the training metrics
        if not self.pretrain:
            self.log('train_auroc', self.train_auroc, sync_dist=True,
            self.log('train_ap', self.train_ap, sync_dist=True, rank_z

    # This method is called at the end of each validation epoch
    def on_validation_epoch_end(self):
        # If not in pretraining mode, print the validation metrics and
        if not self.pretrain:
            print("val_auroc", self.val_auroc.compute(), "val_ap", sel
        self.validation_step_outputs.clear()

    # This method is called for each test step
    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 save_representation is True, save the representations
        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
                else:
                    np.savetxt(f, np.concatenate([z.cpu(), y.cpu()], a
        else:
            y_hat = self.forward(self.feats_to_input(x, batch_size))

        # If pos_frac is not None, calculate the loss with weights
        if self.pos_frac is not None:
            weight = torch.where(y > 0, self.pos_weight, self.neg_weig
            loss = self.loss_function(y_hat, y, weight)
        else:
            loss = self.loss_function(y_hat, y)
```

```
        # Log the test metrics
        self.log('test_loss', loss, on_epoch=True, sync_dist=True, ran
        self.test_auroc.update(y_hat, y.to(int).to(self.device))
        self.log('test_auroc', self.test_auroc, on_epoch=True, sync_di
        self.test_ap.update(y_hat, y.to(int).to(self.device))
        self.log('test_ap', self.test_ap, on_epoch=True, sync_dist=Tru

        return loss, self.test_auroc, self.test_ap

    # This method is called when a checkpoint is loaded
    def on_load_checkpoint(self, checkpoint):
        print('Loading from checkpoint')
        state_dict = checkpoint["state_dict"]
        model_state_dict = self.state_dict()
        is_changed = False

        # Update the state_dict with missing keys from the model_state
        for k in model_state_dict:
            if k not in state_dict:
                state_dict[k] = model_state_dict[k]
                is_changed = True

        # Check for mismatched shapes in the state_dict and model_stat
        for k in state_dict:
            if k in model_state_dict:
                if k.startswith('head') and state_dict[k].shape != mod
                    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 the state_dict was changed, remove the optimizer states f
        if is_changed:
            checkpoint.pop("optimizer_states", None)

        # If freeze_encoder is True, freeze the model parameters
        if self.freeze_encoder:
            self.freeze()

    # This method is used to freeze the model parameters
    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)
```

Training

Now that the data-object is defined, as well as the model itself, the next step is to setup the

logic for training the model as per the approach taken by the original authors in the project; this is done by implementing the DuETT. The below sections highlight the **hyper-parameters** used in the training of the model as well as the **computational parameters** defined.

Hyper-Parameters

1. **d_static_num**: This is the dimensionality of the static input features. It helps the model understand the size of the static input data.
2. **d_time_series_num**: This is the dimensionality of the time series input features. It helps the model understand the size of the time series input data.
3. **d_target**: This is the dimensionality of the target output. It helps the model understand the size of the output data.
4. **lr (learning rate)**: This is the step size at each iteration while moving toward a minimum of a loss function. It determines how fast or slow the model will learn.
5. **weight_decay**: This is a regularization term that discourages large weights in the model to prevent overfitting.
6. **transformer_dropout**: This is the dropout rate for the transformer layers. It can help prevent overfitting by randomly setting a fraction of inputs to zero during training.
7. **max_epochs**: This is the maximum number of passes over the entire dataset. It determines how long the model will be trained. This was tweaked during the training to ensure that we are able to run the code on the limited hardware we have.
8. **gradient_clip_val**: This is the maximum allowed value for the gradients. It prevents the gradients from becoming too large and causing numerical instability.

Computational Requirements

1. **batch_size**: This is the number of samples that will be propagated through the network at once. It affects the speed and memory usage of model training.
2. **num_workers**: This is the number of subprocesses to use for data loading. More workers can increase the speed of data loading.
3. **seed**: This is the random seed for reproducibility. It ensures that the model's results are consistent across different runs.

Learning-Rate Adjustment

This is an adjustment that is applied in the initial phase of the training (also known as the warm-up training phase). The task here is to gradually increase the learning rate over a provided number of steps. This is to prevent massive weight updates from the get-go of the training that can lead to divergence or poor convergence.

In [65]:
```
class WarmUpCallback(pl.callbacks.Callback):
    """
    This class is a PyTorch Lightning callback that implements a linea
    It increases the learning rate linearly for a certain number of st
    according to the inverse square root schedule.
    """

    def __init__(self, steps=1000, base_lr=None, invsqrt=True, decay=N
        """
        Initializes the callback.

        Args:
            steps (int): Number of steps for the warmup phase.
            base_lr (float, optional): Initial learning rate. If None,
            invsqrt (bool): If True, decrease the learning rate using
            decay (int, optional): Decay rate for the inverse square r
        """
        print(f'warmup_steps {steps}, base_lr {base_lr}, invsqrt {invs
        self.warmup_steps = steps
        self.decay = steps if decay is None else decay
        self.state = {'steps': 0, 'base_lr': float(base_lr) if base_lr
        self.invsqrt = invsqrt

    def set_lr(self, optimizer, lr):
        """
        Sets the learning rate for all parameter groups in the optimiz

        Args:
            optimizer (Optimizer): The optimizer.
            lr (float): The learning rate.
        """
        for param_group in optimizer.param_groups:
            param_group['lr'] = lr

    def on_train_batch_start(self, trainer, model, batch, batch_idx):
        """
        This method is called at the beginning of each training batch.
        It adjusts the learning rate according to the schedule.

        Args:
            trainer (Trainer): The PyTorch Lightning trainer.
            model (LightningModule): The model that is being trained.
            batch: The current batch data.
            batch_idx (int): The index of the current batch.
        """
        optimizers = model.optimizers()

        if self.state['steps'] < self.warmup_steps:
            # During the warmup phase, increase the learning rate line
            if isinstance(optimizers, list):
                if self.state['base_lr'] is None:
                    self.state['base_lr'] = [o.param_groups[0]['lr'] f
                for opt, base in zip(optimizers, self.state['base_lr']
                    self.set_lr(opt, self.state['steps'] / self.warmup
            else:
```

```
                if self.state['base_lr'] is None:
                    self.state['base_lr'] = optimizers.param_groups[0]
                self.set_lr(optimizers, self.state['steps'] / self.war
            self.state['steps'] += 1
        elif self.invsqrt:
            # After the warmup phase, decrease the learning rate using
            if isinstance(optimizers, list):
                if self.state['base_lr'] is None:
                    self.state['base_lr'] = [o.param_groups[0]['lr'] f
                for opt, base in zip(optimizers, self.state['base_lr']
                    self.set_lr(opt, base * (self.decay / (self.state[
            else:
                if self.state['base_lr'] is None:
                    self.state['base_lr'] = optimizers.param_groups[0]
                self.set_lr(optimizers, self.state['base_lr'] * (self.
            self.state['steps'] += 1

    def load_state_dict(self, state_dict):
        """
        Loads the state of the callback from a dictionary.

        Args:
            state_dict (dict): A dictionary containing the state of th
        """
        self.state.update(state_dict)

    def state_dict(self):
        """
        Returns a dictionary containing the state of the callback.

        Returns:
            dict: A dictionary containing the state of the callback.
        """
        return self.state.copy()
```

Methods to Assist With Training Process

The following methods are called throughout the training process; the key methods are: 1. **Pre-Training**: This is where the model is initialized for pre-training 2. **Fine-Tuning**: This method is invoked after the initial round of trainings as part of the DuETT task that this project is experimenting with. 3. **Averaging-Models**: Once we have different versions of the fine-tuned model, we average them to get the *best* model (according to this paper).

In [66]:
```
def pretrain_model(d_static_num, d_time_series_num, d_target, **kwargs
    """
    This function initializes a model for pretraining.

    Args:
        d_static_num (int): The dimensionality of the static input fea
        d_time_series_num (int): The dimensionality of the time-series
        d_target (int): The dimensionality of the target output.
        **kwargs: Additional keyword arguments for the Model construct

    Returns:
        Model: The initialized model.
    """
    return Model(d_static_num, d_time_series_num, d_target, **kwargs)


def fine_tune_model(ckpt_path, **kwargs):
    """
    This function loads a pretrained model from a checkpoint and prepa

    Args:
        ckpt_path (str): The path to the checkpoint file.
        **kwargs: Additional keyword arguments for the Model construct

    Returns:
        Model: The model loaded from the checkpoint.
    """
    return Model.load_from_checkpoint(
        ckpt_path,
        pretrain=False,
        aug_noise=0.,
        aug_mask=0.5,
        transformer_dropout=0.5,
        lr=1.e-4,
        weight_decay=1.e-5,
        fusion_method='rep_token',
        **kwargs
    )


def average_models(models):
    """
    This function averages the weights of a list of models and loads t

    Args:
        models (list): A list of models whose weights are to be averag

    Returns:
        Model: The first model in the list, but with the weights repla
    """
    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]
```

Pre-Train the Model

As part of the pre-training, we:

1. Initialize the data-module .
2. Define the model pre-training parameter setup.
3. Define the checkpoint-tracking so that the model with the best results is recorded during pre-training.
4. Define the learning rate for the process.

The code setup in the above code cells is referrred to here.

In [67]:
```
# Set the seed for reproducibility
seed = 2020
pl.seed_everything(seed)

# Initialize the data module
dm = PhysioNetDataModule(batch_size=64, num_workers=2, use_temp_cache=
dm.setup()

# Initialize the pretraining model
pretrain_model = 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.pos_frac(),
    seed=seed
)

# Initialize the checkpoint callback for saving the best model during
checkpoint = pl.callbacks.ModelCheckpoint(
    save_last=True,
    monitor='val_loss',
    mode='min',
    save_top_k=1,
    dirpath='checkpoints'
)

# Initialize the warmup callback for learning rate scheduling
warmup = WarmUpCallback(steps=2000)

# Initialize the trainer and start pretraining
trainer = pl.Trainer(
    logger=False,
    num_sanity_val_steps=2,
    max_epochs=50,
    gradient_clip_val=1.0,
    callbacks=[warmup, checkpoint],
    accelerator='cpu'
)
trainer.fit(pretrain_model, dm)

# Get the path of the pretrained model
pretrained_path = checkpoint.best_model_path

trainer.test(final_model, dataloaders=dm)
```

```
Seed set to 2020

Validating cache...
Validating cache...
Validating cache...

GPU available: False, used: False
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
```

```
warmup_steps 2000, base_lr None, invsqrt True, decay None
Validating cache...
Validating cache...

/home/ec2-user/anaconda3/envs/pytorch_p310/lib/python3.10/site-packag
es/pytorch_lightning/callbacks/model_checkpoint.py:653: Checkpoint di
rectory /home/ec2-user/SageMaker/checkpoints exists and is not empty.
/home/ec2-user/anaconda3/envs/pytorch_p310/lib/python3.10/site-packag
es/torch/optim/adamw.py:50: UserWarning: optimizer contains a paramet
er group with duplicate parameters; in future, this will cause an err
or; see github.com/pytorch/pytorch/issues/40967 for more information
  super().__init__(params, defaults)

   | Name                          | Type                    | Params
-------------------------------------------------------------------------
----
0  | special_embeddings            | Embedding               | 192
1  | embedding_layers              | ModuleList              | 67.7 K
2  | n_obs_embedding               | Embedding               | 16
3  | event_transformers            | ModuleList              | 1.8 M
4  | full_event_embedding          | Embedding               | 29.3 K
5  | time_transformers             | ModuleList              | 2.0 M
6  | full_time_embedding           | Sequential              | 26.8 K
7  | full_rep_embedding            | Embedding               | 888
8  | head                          | Sequential              | 57.1 K
9  | pretrain_value_proj           | Sequential              | 32.0 K
10 | pretrain_presence_proj        | Sequential              | 32.0 K
11 | predict_events_proj           | Sequential              | 25.4 K
12 | predict_events_presence_proj | Sequential              | 25.4 K
13 | tab_encoder                   | Sequential              | 4.5 K
14 | train_auroc                   | BinaryAUROC             | 0
15 | val_auroc                     | BinaryAUROC             | 0
16 | train_ap                      | BinaryAveragePrecision | 0
17 | val_ap                        | BinaryAveragePrecision | 0
18 | test_auroc                    | BinaryAUROC             | 0
19 | test_ap                       | BinaryAveragePrecision | 0
-------------------------------------------------------------------------
----
4.1 M     Trainable params
0         Non-trainable params
4.1 M     Total params
16.279    Total estimated model params size (MB)

Epoch 49: 100%                          132/132 [01:01<00:00,  2.15it/s, val_loss=0.380]
```

A Jupyter widget could not be displayed because the widget state could not be found. This could happen if the kernel storing the widget is no longer available, or if the widget state was not saved in the notebook. You may be able to create the widget by running the appropriate cells.

A Jupyter widget could not be displayed because the widget state could not be found. This could happen if the kernel storing the widget is no longer available, or if the widget state was not saved in the notebook. You may be able to create the widget by running the appropriate cells.

A Jupyter widget could not be displayed because the widget state could not be found. This could happen if the kernel storing the widget is no longer available, or if the widget state was not saved in the notebook. You may be able to create the widget by running the appropriate cells.

A Jupyter widget could not be displayed because the widget state could not be found. This could happen if the kernel storing the widget is no longer available, or if the widget state was not saved in the notebook. You may be able to create the widget by running the appropriate cells.

A Jupyter widget could not be displayed because the widget state could not be found. This could happen if the kernel storing the widget is no longer available, or if the widget state was not saved in the notebook. You may be able to create the widget by running the appropriate cells.

A Jupyter widget could not be displayed because the widget state could not be found. This could happen if the kernel storing the widget is no longer available, or if the widget state was not saved in the notebook. You may be able to create the widget by running the appropriate cells.

A Jupyter widget could not be displayed because the widget state could not be found. This could happen if the kernel storing the widget is no longer available, or if the widget state was not saved in the notebook. You may be able to create the widget by running the appropriate cells.

```
`Trainer.fit` stopped: `max_epochs=50` reached.
```

```
Validating cache...
```

Testing DataLoader 0: 100% 29/29 [00:15<00:00, 1.91it/s]

Test metric	DataLoader 0
test_ap	0.3977024257183075
test_auroc	0.7883566617965698
test_loss	0.6280285530276216

Out[67]:
```
[{'test_loss': 0.6280285530276216,
  'test_auroc': 0.7883566617965698,
  'test_ap': 0.3977024257183075}]
```

Fine-Tune Model

Once the pre-training is complete, we fine-tune the model across different seeds and record the best one.

In [68]:
```
# Fine-tune the model for different seeds
final_model = None
for seed in range(2020, 2023):
    pl.seed_everything(seed)
    fine_tuned_model = fine_tune_model(
        pretrained_path,
        d_static_num=dm.d_static_num(),
        d_time_series_num=dm.d_time_series_num(),
        d_target=dm.d_target(),
        pos_frac=dm.pos_frac(),
        seed=seed
    )

    # Initialize the checkpoint callback for saving the best models du
    checkpoint = pl.callbacks.ModelCheckpoint(
        save_top_k=5,
        save_last=False,
        mode='max',
        monitor='val_ap',
        dirpath='checkpoints'
    )

    # Initialize the warmup callback for learning rate scheduling
    warmup = WarmUpCallback(steps=1000)

    # Initialize the trainer and start fine-tuning
    trainer = pl.Trainer(
        logger=False,
        max_epochs=20,
        gradient_clip_val=1.0,
        callbacks=[warmup, checkpoint],
        accelerator='cpu'
    )
    trainer.fit(fine_tuned_model, dm)

    # Average the weights of the best models and test the final model
    final_model = average_models([
        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_frac=dm.pos_frac()
        ) for path in checkpoint.best_k_models.keys()
    ])
    trainer.test(final_model, dataloaders=dm)

# Test the final model
trainer.test(final_model, dataloaders=dm)
```

```
Seed set to 2020
GPU available: False, used: False
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
```

```
Loading from checkpoint
warmup_steps 1000, base_lr None, invsqrt True, decay None
Validating cache...
Validating cache...
```

```
/home/ec2-user/anaconda3/envs/pytorch_p310/lib/python3.10/site-package
s/pytorch_lightning/callbacks/model_checkpoint.py:653: Checkpoint dire
ctory /home/ec2-user/SageMaker/checkpoints exists and is not empty.
/home/ec2-user/anaconda3/envs/pytorch_p310/lib/python3.10/site-package
s/torch/optim/adamw.py:50: UserWarning: optimizer contains a parameter
group with duplicate parameters; in future, this will cause an error;
see github.com/pytorch/pytorch/issues/40967 for more information
  super().__init__(params, defaults)
```

Results

Below, we will discuss the results of this experiment. The following are the 3 key metrics discussed here:

1. **Loss Score**: This is the value of the loss function on the test set. The loss function measures the difference between the model's predictions and the actual values. In general, a lower loss value is better as it indicates that the model's predictions are close to the actual values.
2. **AUROC Score**: This is the Area Under the Receiver Operating Characteristic curve (AUROC) on the test set. The AUROC is a measure of how well a model can distinguish between different classes. The value ranges from 0 to 1, where a value of 0.5 indicates a model that is no better than random guessing, and a value of 1 indicates a perfect model.
3. **Average Precision**: This is the Average Precision (AP) on the test set. The AP summarizes the precision-recall curve, which shows the trade-off between precision and recall for different threshold. A higher AP indicates better precision and recall balance. The value ranges from 0 to 1, where a value of 1 indicates a perfect model.

Evaluate Base Model

At this point, we have the base iteration of our model, which will be fine-tuned as per the DuETT implementation. These are the metrics returned.

Loss Score	AUROC Score	Average Precision
0.6280285530276216	0.7883566617965698	0.3977024257183075

The key metrics surfaced here are:

1. **test_loss**: Our model's test loss is `0.6280285530276216` , which is fairly high.
2. **test_auroc**: Our model's test AUROC is `0.7883566617965698` , which is reasonably good.
3. **test_ap**: Our model's test AP is `0.3977024257183075` , which is not very high.

Evaluate Final Model

At this point, we have the final iteration of the model, post fine-tuning. As seen in the code above, the fine-tuning is performed across multiple `seed` values, and it returns the following scores:

Seed	Loss Score	AUROC Score	Average Precision
2020	0.5696057305793529	0.8056058883666992	0.43770408630371094
2021	0.5696057305793529	0.7875723838806152	0.34840890765190125
2022	0.5783784175941128	0.7970470786094666	0.44990038871765137

The key metrics surfaced here are:

1. **Loss Score**: Our model's lowest test loss is `0.5696057305793529` .
2. **AUROC Score**: Our model's test AUROC is `0.8056058883666992` .
3. **Average Precision**: Our model's test AP is `0.44990038871765137` .

Comparisons

If we compare the 3 key metrics:

1. **Loss Score**: We see a significant improvement in the fine-tuned model over the pretrained model. This score also falls closely in line with the scores seen in the original paper from the author (see *Table 2* on page 15 (https://arxiv.org/pdf/2304.13017)), actually being an improvement despite fewer training epochs.
2. **AUROC Score**: We see a significant improvement in the fine-tuned model over the pretrained model. However, due to machine limitations, the score still comes out lower than what was in the paper by ~0.6% (see *Table 1* on page 12 (https://arxiv.org/pdf/2304.13017)). It is interesting to note that the AUROC went up from 0.59 to 0.81 during the training; therefore, it is safe to assume that if we fine-tuned the model for 50 epochs instead of 20, we would have reached much closer to the paper's results.
3. **Average Precision**: This is not a metric studied in the paper; however, we see a significant improvement in the fine-tuned model over the pretrained model.

Comparing to Hypothesis

Comparing these results on `Physionet2012` with the findings on the MIMIC-IV dataset recorded in the paper indicate that we were able to reproduce similar results, and demonstrate the effectiveness of self-supervised learning (SSL) to build a strong model without having to rely on a massive dataset (e.g MIMIC-IV). Desipte the sparsity of the `Physionet2012` dataset (with time-series fields being sporadically defined), the training results match the findings using the regular dataset.

Discussions

Implications

Given that we were largely able to obtain similar results to what was achieved in the paper (improved result scores for some metrics, and slightly diminished results in other cases), this paper has a high reproducibility.

What Was Easy

This paper had many strenghts that made it easy to replicate:

1. The biggest benefit of replicating this author's project was the simplicity of accessing the dataset; the main data-set that all the hypotheses can simply be imported as a Python package and integrated into the code, all of which is already set up.
2. The code is also compact in nature, as well as well-structured (although with minimal to no comments) and organized, which makes it a lot easier to trace and understand as part of replicating it.

Challenges

Despite the benefits, there were still some issues faced when replicating this paper.

1. The main concern was setting up the environment. The Python version used by the authors is not specified; therefore, I had to tweak the dependencies using trial-and-error when running the code for the first time on my local computer. This problem amplified when attempting to upload my code and running it on an online instance, because the default Python version in Google Collab and AWS SageMaker was Python 3.10.3, and changing that was near-impossible. This meant that I:
   A. Had to remove all dependency versions and install the latest ones to break the conflict issues.
   B. Make modifications to the code because some dependencies *f*rom the requirements.txt (https://github.com/layer6ai-labs/DuETT/blob/master/requirements.txt) of the package were severely outdated, and the API specifications had changed.
2. Another problem was the computing requirements. Despite being a smaller dataset, my local computer would fail halfway into the training process due to a lack of memory. Google Collab was not helpful as its processor was even weaker than my local computers. I had to then set up an AWS SageMaker notebook on a powerful `ml.r7i.2xlarge` instance that was able to complete the entire training.

Feedback to Authors

Despite the challenges above, the paper is well-written and implemented, making it easy to reproduce. My only feedback would be directed at using the latest version of dependencies, especially when uploading the code to a public repository for allowing anyone to reproduce it; given that the requirements.txt (https://github.com/layer6ai-labs/DuETT/blob/master/requirements.txt) was only uploaded 9 months ago (from the time of writing this), an effort could have been made to rely on the newer dependencies.