CS598 DL4H Project Draft Spring 2024

Yan Han, Xiaojing Xu, and Jake Xu
{yanhan4, xx33, cxu54} @illinois.edu
Group ID: 39
Paper ID: 46

1. Introduction

The challenge of applying Transformer models [1] to Electronic Health Records (EHR) lies in their complex, multidimensional structure, which sometimes causes these sophisticated models to underperform compared to simpler methods. This gap highlights a missed opportunity to leverage Transformers' advantages, such as efficient transfer learning and scalability, within the realm of EHR analysis.

Addressing this issue, the paper titled "SANSformers: Self-Supervised Forecasting in Electronic Health Records with Attention-Free Models" [2] introduces the SANSformer model. This innovative, attention-free sequential model is specifically designed to navigate the unique complexities of EHR data, with a focus on enhancing predictions for future healthcare utilization. This enhancement is crucial for effectively managing healthcare resources, especially for diverse patient subgroups with distinct health trajectories, including those with rare diseases. The paper also employs a self-supervised pre-training strategy known as Generative Summary Pretraining (GSP), aimed at predicting future health trends from past records.

Our project aims to replicate the findings of this seminal paper, especially its performance on predicting mortality rate on MIMIC data. By doing so, we seek to validate the potential of attention-free models and GSP techniques in improving healthcare resource prediction and management.

2. Scope of reproducibility

The paper established three tasks to evaluate model performance: 1) The number of physical visits to healthcare centers. 2) The counts of physical visits due to six specific disease categories. 3) predicting the probability of inpatient mortality. The first two tasks are both performed on the Pummel dataset, which is a confidential dataset. Therefore, we focus on reproducing the third task, which is performed on MIMIC-IV. We

will train a SANSformer model on patient history, excluding the two most recent visits from each patient. Thus effectively predict the probability of mortality following the subsequent two visits. The tal_expire_flag feature from the MIMIC admissions table will be used as the label for mortality. After training, we will compare its AUC with that of several baseline models including logistic regression, RETAIN, BEHRT, BRLTM and SARD.

3. Methodology

3.1 Data

3.1.1 Data descriptions

In this research we will apply the MIMIC-IV v2.2 dataset[3], drawn from the Beth Israel Deaconess Medical Center (BIDMC), encapsulates the Electronic Health Records (EHRs) of around 250,000 patients. It's a comprehensive collection that incorporates various aspects of patient information, including vitals, doctor's notes, diagnosis, procedures, medication codes, discharge summaries, and additional details from both standard hospital and ICU stays.

We will narrow our focus to hospital admissions and utilize data from patients, admissions, diagnoses_icd, procedures_icd, drgcodes, and services tables. To organize visit information, we will use the admission identifier (hadm_id) and group all visits under each patient's unique identifier (subject_id). The final dataset should include information on 256,878 patients, with an average of 2.04 visits per patient. Some descriptive statistics is listed below:

TABLE I: **Basic statistics of the two EMR datasets.**

	Pummel	**MIMIC-IV**
No. of unique patients	1,050,512	256,878
No. of visits	60,896,305	523,740
Avg. no. of visits per patient	57.94	2.04
Max no. of visits	1,443	238
Avg no. of codes per visit	8.46	18.84
Max no. of codes per visit	164	110
Token Vocabulary Size	5237	5019

3.1.2 Implementation code

We create the main dataframe by grouping all icd codes associated with each patient per admission from the original .csv files.

For example, preprocess.py groups all diagnosis code from diagnoses_icd.csv.gz and generates a .feather file.

Then, by merging all these grouped .feather files, we created the final patient dataset grouped_patient_rem. 80% of this dataset is used for training (35058 entries) and validation and 20% is used for testing (8765 entries).

There are 25 features including gender, race, past procedures, drugs prescribed, etc. and one target: patient mortality.

```
grouped_patient_rem2_df = pd.read_feather(full_data_path + '/grouped_patient_rem2.feather')

mimic_train_rem2_df = pd.read_feather(full_data_path + '/mimic_train_rem2.feather')

mimic_test_rem2_df = pd.read_feather(full_data_path + '/mimic_test_rem2.feather')
```

The training set has 35058 data points and the test set has 8765 data points. Each has 25 features and 1 target.

```
print("Training set shape: ", mimic_train_rem2_df.shape)
print("Test set shape: ", mimic_test_rem2_df.shape)
```

```
Training set shape:  (35058, 26)
Test set shape:  (8765, 26)
```

```
[ ] mimic_train_rem2_df.describe()
```

	subject_id	seq_length	encoded_gender	encoded_ethnicity	encoded_language
count	3.505800e+04	35058.000000	35058.000000	35058.000000	35058.000000
mean	1.500361e+07	3.920218	0.469194	12.350990	0.895060
std	2.879765e+06	5.800473	0.499057	6.536986	0.306481
min	1.000003e+07	1.000000	0.000000	0.000000	0.000000
25%	1.250449e+07	1.000000	0.000000	9.000000	1.000000
50%	1.501819e+07	2.000000	0.000000	9.000000	1.000000
75%	1.748134e+07	4.000000	1.000000	15.000000	1.000000
max	1.999978e+07	236.000000	1.000000	32.000000	1.000000

```
[ ] mimic_test_rem2_df.describe()
```

	subject_id	seq_length	encoded_gender	encoded_ethnicity	encoded_language
count	8.765000e+03	8765.000000	8765.000000	8765.000000	8765.000000
mean	1.499825e+07	3.825214	0.469139	12.293098	0.893326
std	2.884189e+06	5.361426	0.499075	6.438045	0.308716
min	1.000140e+07	1.000000	0.000000	0.000000	0.000000
25%	1.253293e+07	1.000000	0.000000	9.000000	1.000000
50%	1.494788e+07	2.000000	0.000000	9.000000	1.000000
75%	1.749759e+07	4.000000	1.000000	15.000000	1.000000
max	1.999877e+07	93.000000	1.000000	32.000000	1.000000

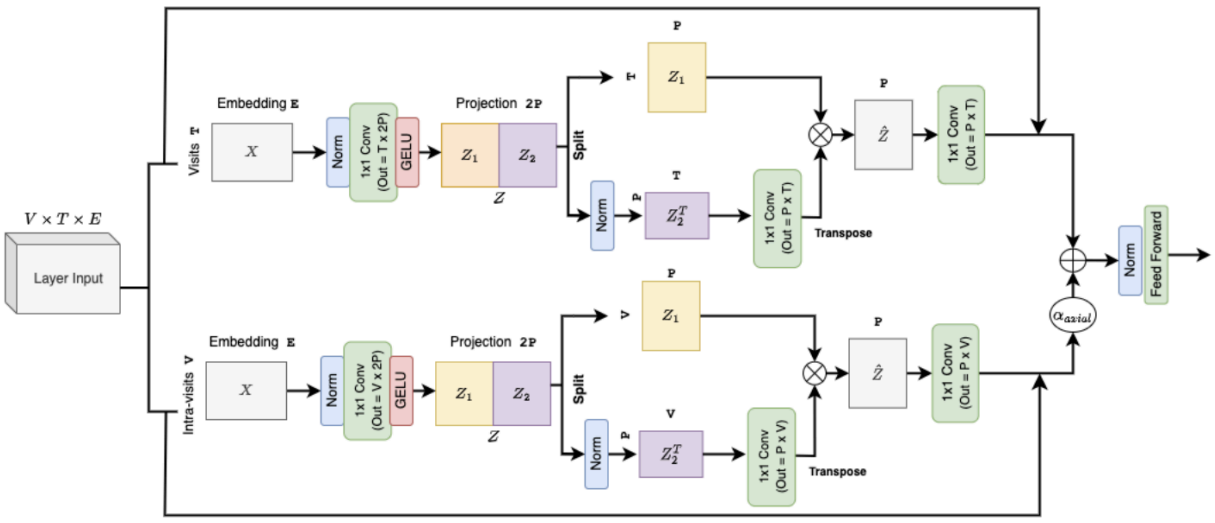
Due to the missing code chunk for creating vectorizer, we implemented our own vectorizer generation code and the result might be different from the original paper. We

already reached out to the author for the code, and in case that we didn't get a prompt response, we'll refine our vectorizer creation throughout this project, before our final submission of it.

3.2 Model

3.2.1 Model descriptions

The SANSformer model, adapted for EHR data, features positional encodings, axial attention, and point-wise feedforward layers, reducing computational complexity. Skip-connections and layer normalization address deep network challenges. Specifically for EHR, axial mixing is modified for time and visit contexts, improving relevance by summarizing visits before mixing. A weighted average replaces simple tensor addition, tuned for specific prediction tasks, enhancing effectiveness and interpretability. This adaptation makes SANSformer suitable for handling the multidimensional nature of EHR data.



The SANSformer model, tailored for Electronic Health Record (EHR) data, employs axial decomposition to create matrices for temporal mixing (visits) and token mixing (intra-visit). It uses "mixers," similar to 1x1 convolutions, enabling efficient cross-token interactions by capturing second-order interactions, a contrast to the third-order interactions of self-attention. The processing involves Gaussian Error Linear Units (GELU) for non-linear transformations, followed by Spatial Gating Units (SGU) for cross-time mixing. Specifically, the SGU applies the sequence:

$$Z = \mathrm{GELU}(XU)$$
$$\widehat{Z} = \mathrm{SGU}(Z)$$
$$Y = \widehat{Z}V$$

, where U and V are weight matrices. Causal masking ensures no future time-step data leakage, optimizing the model for predictive integrity in EHR applications.

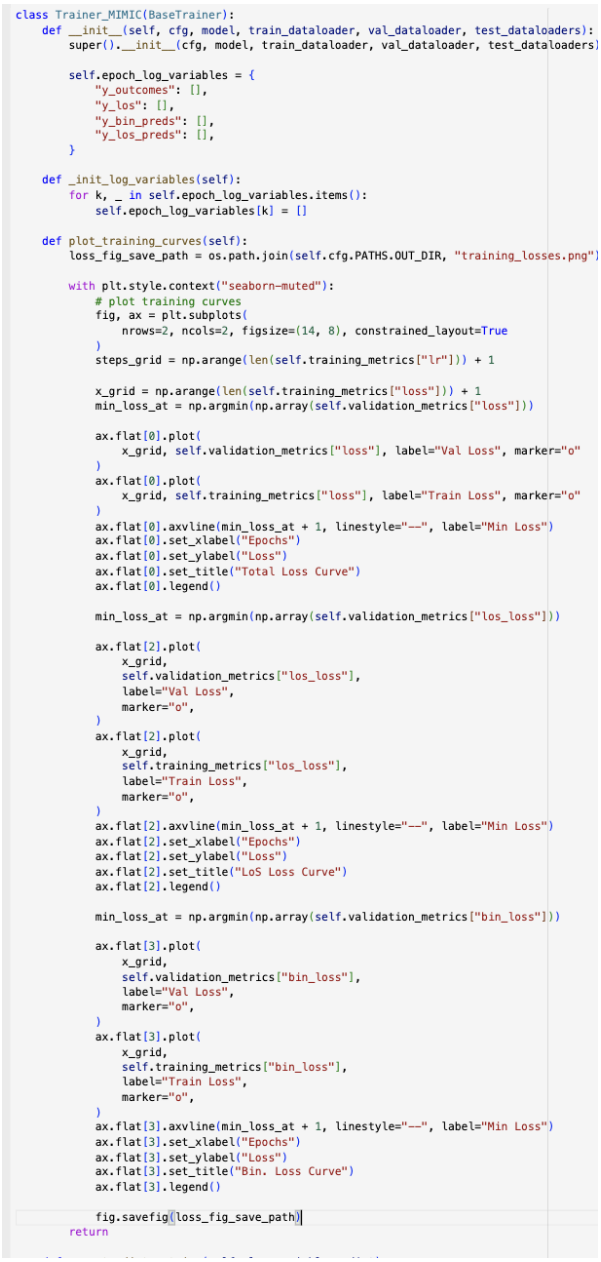
3.2.2 Model Architecture

The trainer_MIMIC model contains a total of 1,953,357 trainable parameters. Indicating its capacity to model complex patterns and relationships in large-scale data effectively. The architecture suggests a deep, multi-layer transformer-based model, using repeated blocks of custom transformers (SANSformer and Visit SANSformer), each with projection, convolution, normalization, and feedforward components to deeply process and transform the input EHR data.

There are embedding layers(pummel_embed.token_embed.weight,ethn_embed.weight, etc.), SANSformer Layers, including projection and convolution layers(proj_ch1,conv_proj,proj_out,etc.), normalization layers(norm1,norm2,norm3,etc.), feedforward layers(ff,etc.).

Also there are activation functions, the specific activation functions used aren't directly listed in the module breakdown, but common choices in similar architectures include ReLU or GELU. After all there are binary output layers(bin_fc, etc.), These layers are the final stages in the network designed for binary classification tasks.

3.2.3 Implementation code



```python
class Trainer_MIMIC(BaseTrainer):
    def __init__(self, cfg, model, train_dataloader, val_dataloader, test_dataloaders):
        super().__init__(cfg, model, train_dataloader, val_dataloader, test_dataloaders)

        self.epoch_log_variables = {
            "y_outcomes": [],
            "y_los": [],
            "y_bin_preds": [],
            "y_los_preds": [],
        }

    def _init_log_variables(self):
        for k, _ in self.epoch_log_variables.items():
            self.epoch_log_variables[k] = []

    def plot_training_curves(self):
        loss_fig_save_path = os.path.join(self.cfg.PATHS.OUT_DIR, "training_losses.png")

        with plt.style.context("seaborn-muted"):
            # plot training curves
            fig, ax = plt.subplots(
                nrows=2, ncols=2, figsize=(14, 8), constrained_layout=True
            )
            steps_grid = np.arange(len(self.training_metrics["lr"])) + 1

            x_grid = np.arange(len(self.training_metrics["loss"])) + 1
            min_loss_at = np.argmin(np.array(self.validation_metrics["loss"]))

            ax.flat[0].plot(
                x_grid, self.validation_metrics["loss"], label="Val Loss", marker="o"
            )
            ax.flat[0].plot(
                x_grid, self.training_metrics["loss"], label="Train Loss", marker="o"
            )
            ax.flat[0].axvline(min_loss_at + 1, linestyle="--", label="Min Loss")
            ax.flat[0].set_xlabel("Epochs")
            ax.flat[0].set_ylabel("Loss")
            ax.flat[0].set_title("Total Loss Curve")
            ax.flat[0].legend()

            min_loss_at = np.argmin(np.array(self.validation_metrics["los_loss"]))

            ax.flat[2].plot(
                x_grid,
                self.validation_metrics["los_loss"],
                label="Val Loss",
                marker="o",
            )
            ax.flat[2].plot(
                x_grid,
                self.training_metrics["los_loss"],
                label="Train Loss",
                marker="o",
            )
            ax.flat[2].axvline(min_loss_at + 1, linestyle="--", label="Min Loss")
            ax.flat[2].set_xlabel("Epochs")
            ax.flat[2].set_ylabel("Loss")
            ax.flat[2].set_title("LoS Loss Curve")
            ax.flat[2].legend()

            min_loss_at = np.argmin(np.array(self.validation_metrics["bin_loss"]))

            ax.flat[3].plot(
                x_grid,
                self.validation_metrics["bin_loss"],
                label="Val Loss",
                marker="o",
            )
            ax.flat[3].plot(
                x_grid,
                self.training_metrics["bin_loss"],
                label="Train Loss",
                marker="o",
            )
            ax.flat[3].axvline(min_loss_at + 1, linestyle="--", label="Min Loss")
            ax.flat[3].set_xlabel("Epochs")
            ax.flat[3].set_ylabel("Loss")
            ax.flat[3].set_title("Bin. Loss Curve")
            ax.flat[3].legend()

            fig.savefig(loss_fig_save_path)
        return
```

The Trainer_MIMIC class, an extension of BaseTrainer, initializes with model configurations and data loaders for training, validation, and testing phases. It manages log variables to track outcomes and predictions, which are reset at each epoch's start. The method plot_training_curves creates visual representations of training and validation loss curves, highlighting epochs where minimum losses occur. This plotting function is crucial for monitoring and evaluating model performance over time, allowing for adjustments in training strategies. The visualizations are saved in a specified directory, facilitating easy access and review of the model's training progress and effectiveness.

3.3 Training

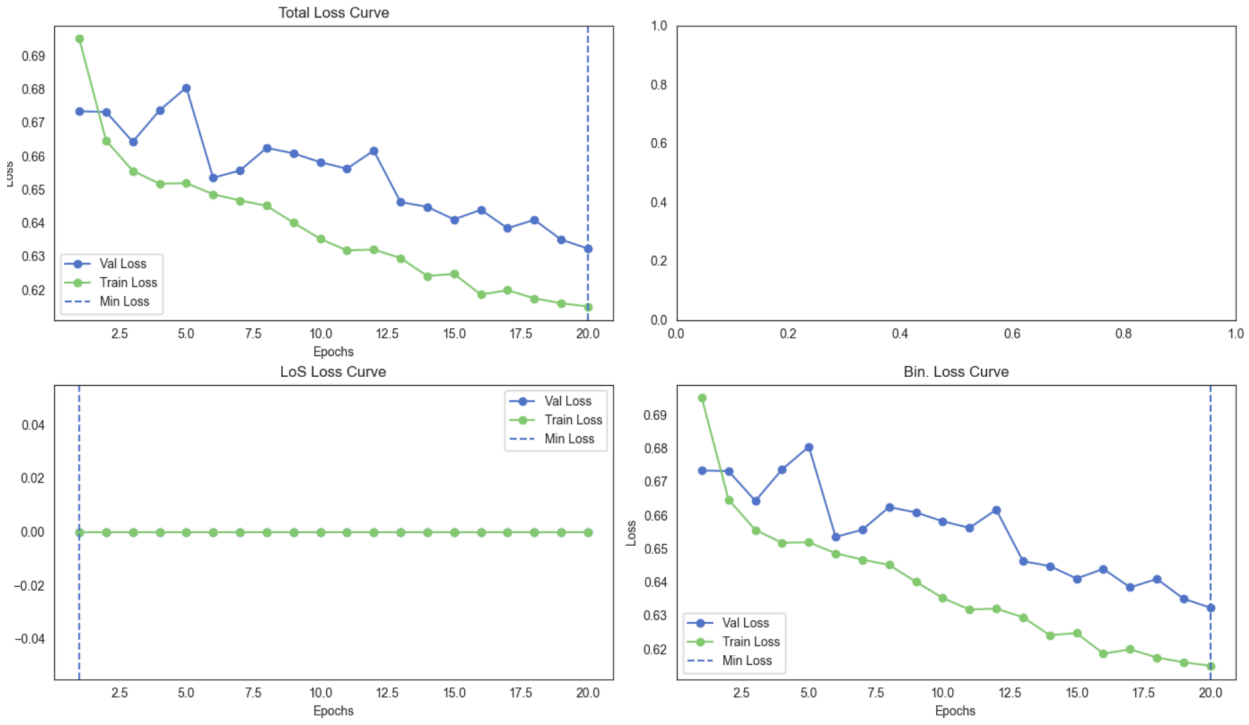
3.3.1 Feasibility of Computation

The original paper performed model training on 256,878 patients using a single NVIDIA Tesla V100 GPU for 20 epochs with a batch size of 32. We should be able to reproduce that on a personal computing system with a 2.3 GHz 8-Core CPU, 32 GB 2400 MHz

DDR4 memory, and a 8GB GDDR6 GPU. We can also utilize Google Colab with its GPU options. In addition, the original paper has a GitHub repository that contains code for data preprocessing, data loading, model training and final prediction. We will run it in our environment and also write code to calculate the ROC score to compare with the baseline.

3.3.2 Training Objective

The model aims to predict mortality rate and uses spearman loss for training. If the prediction is highly correlated with the ground truth, the model will have a small loss. It also uses AUC as another metric for model evaluation. The figure below shows the training loss curve for the first 20 epochs.



3.3.3 Implementation code

```python
def run_epoch(self, split, dataloader, epoch_count=0):
    assert split.lower() in ("train", "validation", "test")

    self._init_log_variables()

    is_train = True if split.lower() == "train" else False
    self.model.train(is_train)

    lr = 0.0
    losses = []
    bin_losses = []
    los_losses = []
    pbar = (
        tqdm(enumerate(dataloader), total=len(dataloader))
        if is_train
        else enumerate(dataloader)
    )
    self.model.zero_grad()
    acc_steps = self.cfg.MODEL.ACCU_GRAD_STEPS
    for it, batch in pbar:
        # place data on the correct device
        batch_dict = self._prepare_batch(batch)
        # forward the model
        with torch.set_grad_enabled(is_train):
            with autocast(enabled=self.cfg.USE_AMP):
                (
                    loss,
                    bin_loss,
                    los_loss,
                    y_pred_bin,
                    y_los_pred,
                    y_los_true,
                    patient_vec,
                ) = self.model(**batch_dict)
        report_loss = loss
        if is_train:
            loss = loss / acc_steps
            # backprop and update the parameters
            self.grad_scaler.scale(loss).backward()
            if (it + 1) % self.cfg.MODEL.ACCU_GRAD_STEPS == 0:
                self.grad_scaler.unscale_(self.optimizer)
                torch.nn.utils.clip_grad_value_(
                    self.model.parameters(), self.cfg.OPTIM.GRAD_CLIP_T
                )
                self.grad_scaler.step(self.optimizer)
                self.grad_scaler.update()
                self.model.zero_grad()

                lr = 0.0
                for param_group in self.optimizer.param_groups:
                    lr = param_group["lr"]

                if self.cfg.OPTIM.LR_POLICY in ["1cycle"]:
                    self.scheduler.step()
                self.training_metrics["lr"].append(lr)

            # report progress
            pbar.set_description(
                f"epoch {epoch_count + 1} iter {it}: train loss {loss.item() * acc_steps:.5f} lr {lr:e}"
            )

        # store metrics for logging
        losses.append(report_loss.item())
        bin_losses.append(bin_loss.item())
        los_losses.append(los_loss.item())

        self.epoch_log_variables["y_outcomes"].extend(
            batch_dict["y_outcome"].cpu().detach().numpy().tolist()
        )
        self.epoch_log_variables["y_los"].extend(
            y_los_true.cpu().detach().numpy().tolist()
        )
        self.epoch_log_variables["y_bin_preds"].extend(
            y_pred_bin.cpu().detach().numpy().tolist()
        )
        self.epoch_log_variables["y_los_preds"].extend(
            y_los_pred.cpu().detach().numpy().tolist()
        )

    metrics_d = {
        "loss": float(np.mean(losses)),
        "bin_loss": float(np.mean(bin_losses)),
        "los_loss": float(np.mean(los_losses)),
    }

    if not is_train:
        pass
        metrics_d.update(self.compute_dist_metrics(self.epoch_log_variables))

    # compute metrics
    print_str = f"{split} epoch: {epoch_count+1} "
    for k, v in metrics_d.items():
        print_key = str(k)
        print_str += f" | {print_key}: {v:.5f}"

    print(print_str)
    print("=" * 100)
    return metrics_d
```

The run_epoch method in the Trainer_MIMIC class executes a single epoch for either training, validation, or testing phases, asserting that the split argument is one of these three. It starts by reinitializing logging variables and setting the model to train or evaluation mode based on the phase. During training, it employs a progressive gradient accumulation strategy to update the model parameters efficiently, adjusting the learning rate and clipping gradients to stabilize training. It collects and logs loss metrics and predictions for detailed tracking. For non-training phases, it simply updates metrics without backpropagation. The method concludes by printing and returning the collected metrics, providing a snapshot of model performance for the epoch.

3.4 Evaluation

3.4.1 Hypothesis to be tested

Transformers have shown superior ability to model natural language. However, they often require extensive pretraining on large datasets, making them less applicable to small or specialized datasets. In addition, the consecutive visits in EHR are not as

strongly correlated as consecutive words in a sentence, therefore this study hypothesized that using self-attention to model their time-wise interaction might be an overkill and proposed sequential architecture instead. The SANSformer introduces a novel concept known as attention-free mixers. These mixers are designed to facilitate the interaction between tokens in the input sequence without the need for calculating attention scores. The authors believed this new architecture can usually achieve a satisfactory result for most EHR applications.

3.4.2 Metrics descriptions

The model uses Area Under Curve (AUC) as the evaluation metric for mortality rate. In the original paper, SANSformer showed superior performance over many prevalent model architectures such as RETAIN and BEHRT.

Model	AUC ↑
L_1-reg. Logistic Regression [57]	0.728
RETAIN [11]	0.707 ± 0.007
BEHRT [40]	0.693 ± 0.006
BRLTM [48]	0.695 ± 0.004
SARD [34]	0.742 ± 0.002
Additive SANSformer (ours)	0.759 ± 0.002
Axial SANSformer (ours)	**0.761 ± 0.004**

3.4.3 Implementation code

Test pretrained model

```
[ ]  test_epoch_metrics_l = []
     for i, test_loader in enumerate(test_dataloaders):
         test_epoch_metrics = trainer.run_epoch("test", test_loader)
         test_epoch_metrics_l.append(test_epoch_metrics)
```

```
[ ]  avg_auc = 0
     for epoch in range(len(test_epoch_metrics_l)):
       avg_auc += test_epoch_metrics_l[epoch]['auc_bin']
       print("Epoch %d AUC: %f" % (epoch, test_epoch_metrics_l[epoch]['auc_bin']))
     avg_auc /= len(test_epoch_metrics_l)
     print("Average test AUC:", avg_auc)
```

4. Results

4.1 Results

Our test loss is 0.639 and AUC is 0.694, suggesting a fair distinction between classes but leaving room for improvement.

4.2 Analyses

Our AUC is slightly lower than the original paper (0.761), but since we only trained around 20 epochs, and the training curve shows that the model is still underfitting, we plan to train more epochs and perform hyperparameter tuning to improve performance.

4.3 Plans

4.3.1 Ablations planned

A key parameter introduced in this paper is α_axial, which facilitates the processing of visit and code information separately. We plan to study its impact by adjusting its value to zero, training the new model on the same data using the same process, and comparing its performance with the original model's.

4.3.2 Further Steps

A. Train for more epochs to allow more learning.
B. Experiment with different learning rates, batch sizes, and other relevant hyperparameters to see if the model's performance can be improved.

5. Citations

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Advances in Neural Information Processing Systems, pages 5998–6008, 2017.
[2] Yogesh Kumar , Alexander Ilin , Henri Salo , et al. SANSformers: Self-Supervised Forecasting in Electronic Health Records with Attention-Free Models. TechRxiv. August 14, 2023.
[3] Johnson, A., Bulgarelli, L., Pollard, T., Horng, S., Celi, L. A., & Mark, R. (2023). MIMIC-IV (version 2.2). PhysioNet. https://doi.org/10.13026/6mm1-ek67.