DL4H Team 1 Final Report

Ted Hsu, Myles Iribarne, Daniel Xu

May 7, 2024

Contents

1 Basic Info

1.1 Team 1

- Ted Hsu (thhsu4@illinois.edu)
- Myles Iribarne (mylesai2@illinois.edu)
- Daniel Xu (dhxu2@illinois.edu)

1.2 Paper Reproduced

Our paper is *Transfer learning for ECG classification* by Weimann and Conrad [1]. The original paper's code is available on Github.

1.3 Project Video/Github Links

A short 4 minute video about our project can be found here.

Our Github repo with all of our project code can be found here.

2 Abstract

This project replicates and extends a study on Atrial Fibrillation (AF) classification using ECG recordings [1]. The original study transferred pre-training learnings from distinct tasks in the *Icentia11k* dataset to enhance performance of the fine-tuned model trained on the *PhysioNet/CinC Challenge 2017* dataset to detect AF. Our results successfully replicated the original study's findings using beat classification for the pre-training step, notably demonstrating tha effects that the quantity of pre-training data has on final model performance. Pre-training on just 1% of the icentia11k dataset increased the F1 score of the fine-tuned model by 0.063. The remaining 99% of data increased the F1 score by a further 0.008.

We also explored an extension to the original method by integrating spectrogram feature preprocessing and a 2-D ResNet architecture, a shift from the original's 1-D convolutional approach. Our findings revealed that the 1-D model's pre-training advantages did extend to the 2-D model. The performance of the spectrogram-based models was equivalent to the 1-D model, and showed similar increase in performance when pre-training was applied.

3 Introduction

3.1 Background of the Problem

- **What type of problem**:

  The problem is to classify Atrial Fibrillation (AF) on electrocardiogram (ECG) recordings.

- **What is the importance/meaning of solving the problem**:
  - A solution to the problem is a tool that will assist physicians in analyzing large amounts of patient ECG data in an automated and time efficient manner.
  - Early detection of AF events may lead to better patient outcomes.

- **What is the difficulty of the problem**:

  - Devices for recording patient ECG data are able to output a *huge* amount of raw data. This is challenging and expensive to annotate for effective Deep Learning training.
  - Large class imbalance due to cardiovascular events of interests being rare.
  - Low ECG signal quality due to sampling frequency, single ECG lead probe.

- **The state of the art methods**:
  - Transfer learning using 1-D residual networks [2]
  - Representation learning using encoder-decoder architectures
    * Stacked Denoising AEs [3]
    * Seq2Seq model [4]

3.2 Paper Explanation

- **What did the paper propose**:
  - Use Transfer learning to build better ECG classifiers.
  - Pre-train 1-D CNNs on the largest publicly available ECG dataset (*Icentia11k*) on several pre-training tasks:
    * Beat Classification
    * Rhythm Classification
    * Heart Rate Classification
    * Future Prediction
  - Fine-tune the pre-trained 1-D CNNs on a *different* task and a *different* dataset (*PhysioNet/CinC Challenge 2017*): classify AF events.

- **What is/are the innovations of the method**:
  - Demonstration of successful large-scale pre-training of 1-D CNNs on the largest publicly available ECG dataset to date.
  - Demonstration of contrastive pre-training (unsupervised representation learning) improving 1-D CNN performance on target task.
  - Novel usage of heart rate classification task for pre-training. Note that in this task, the labels can be automatically generated without manual intervention.

- **How well the proposed method work (in its own metrics)**:
  - The paper provides AF classifier performance comparison among five different pre-training tasks configurations (Random initialization, Beat classification, Rhythm classification, Heart Rate classification, and Future Prediction).

    Macro F1 score of the AF classifier on the PhysioNet 2017 test set is the performance metric.

  - The average macro F1 score of random initialization pre-training task is 0.731 over 10 trials. Average macro F1 scores reported by all proposed four pre-training tasks configurations range from 0.758 to 0.779 over 10 trials.

- **What is the contribution to the research regime (referring the Background above, how important the paper is to the problem)**:
  - Pre-training the 1-D CNN model improves the performance on the target task (i.e. AF classification), effectively reducing the number of labeled data required to achieve the

same performance as 1-D CNNs that are not pre-trained.
  - Unsupervised pre-training (i.e. future prediction) on ECG data is a viable method for improving the performance on the target task and will become more relevant, since labeling ECG data is expensive.

4 Scope of Reproducibility

4.1 Hypothesis 1

Pre-training 1-D CNN models with an extremely large dataset of relatively inexpensively labeled data can improve performance of classification based on a smaller set of labeled data with a different classification objective (i.e. AF).

4.2 Hypothesis 2

The paper does not explore how significant the effects of the pre-training data size are on the final results. We expect the size of the pre-training dataset affects the performance of the target task (i.e. AF classification).

4.3 Verification

We will verify the hypotheses by attempting to reproduce results for a specific model and the following hyperparameter combination with 1%, 10%, 20%, and 100% of the pre-training data used in the paper:

- Model: 1-D ResNet-18v2
- Pre-training Objective: Beat Classification
- Frame Size: 4096 samples
- Sample Rate: 250 Hz
- Fine-tuning objective: Atrial Fibrillation

Note that the data is at patient level, so the percentages will be applied to the total number of patients.

The results will be compared with the performance of a randomly initialized ResNet-18v2.

4.4 Implementation

The paper authors have provided their code online on Github. In our repo, we have forked their code and made small adjustments for convenience in our reproduction work. The code cells below will import modules from the paper authors' code. In this way, we are able to focus primarily on executing the experiments and analyzing the outcomes.

5 Methodology

5.1 Environment

First, we assume that this notebook is run in **Google Colab** with **Python 3.10**. It is highly recommended to have Colab Pro and select **V100** GPU or better. All results were generated using

a V100 GPU with 16GB RAM, except for the ablation results. The ablation results were generated using L4 GPU with 22.5GB RAM.

Below we prepare the environment with which the code in this notebook can run.

First, we clone our project repo which is a fork of the original repo. Our repo also contains our modifications and our own original code.

[1]:
```
REPO = "/tmp/repo"
```

[2]:
```
%%capture
!git clone https://github.com/myles-i/DLH_TransferLearning.git {REPO}
%cd {REPO}
```

Second, we install all of the dependencies specified in `requirements.txt`. This link lists the dependencies used.

[3]:
```
%%time
%%capture
!pip install -r requirements.txt
```

```
CPU times: user 91 ms, sys: 28.4 ms, total: 119 ms
Wall time: 16.2 s
```

Next, we prepare the directories for holding the data files we will use in later code cells of this notebook.

[4]:
```
JOB_DIR = "/tmp/jobs"
DATA_DIR = "/tmp/data"
DEMO_DATA_DIR = DATA_DIR + "/final_demo"
```

[5]:
```
%%capture
# Prepare local directories
!mdkir -p {JOB_DIR}
!mkdir -p {DATA_DIR}
!mkdir -p {DEMO_DATA_DIR}
```

5.2 Data

5.2.1 Pre-training Dataset

The training data is the "Icentia11k Single Lead Continuous Raw Electrocardiogram Dataset," which is freely available online [5][6][7].

Data Download Instructions Source of the data:

- The data comes in two formats, and links to each format are provided below:
  - Raw
  - Compressed

Download steps:

- We utilize the *compressed* data files rather than the raw files.
- The process is demonstrated in this notebook which we wrote and used to download the data. It uses the `libtorrent` library to download the compressed data.
- The compressed data files are saved to an appropriately named sub-directory within the shared Google Drive directory stored in the `DATA_DIR` variable.

Description Statistics:

- 11,000 patients.
- Using 4096 samples per patient, this is 4.5 million samples
- Each patient has up to two weeks of ECG recordings with 250 Hz sampling rate.
- Each ECG recording is accompanied with beat and rhythm labels marked by the ECG signal collection device and specialists, respectively.
- Both beat and rhythm labels are assigned to positions in the signal at irregular intervals.
- The original paper uses 95% of the patients for pre-training and the remaining 5% for validation.

Pre-processing Code and Command First we look at patient#0's data as an example. The data processing code is from the paper authors' original code.

[6]:
```
# First download Patient#0 ECG signal and labels
%%time
%%capture
# Patient#0 ECG signal
!gdown 1hzq0iSICypZTIrJcyPfAR5mPrKS1Vev_ \
  -O {DEMO_DATA_DIR + '/00000_batched.npy'}
# Patient#0 label
!gdown 1vjBhJ_0w-bTn7jhnRPS3Bf3CsLpi7RuN \
  -O {DEMO_DATA_DIR + '/00000_batched_lbls.npz'}
```

```
CPU times: user 86.8 ms, sys: 17 ms, total: 104 ms
Wall time: 16.4 s
```

[7]:
```
from transplant.datasets import icentia11k

(signal, labels) = icentia11k.load_patient_data(
    DEMO_DATA_DIR, 0, include_labels=True, unzipped=True
)
```

Each patient's data is loaded as a tuple of ECG signal and labels.

[8]:
```
print(f"ECG signal is 2D numpy array with shape {signal.shape}.")
print(
    f"With 250 Hz sampling rate, the length of ECG signal is:"
    f" {signal.shape[0]*signal.shape[1]/250/60:.2f} minutes."
)
```

```
ECG signal is 2D numpy array with shape (50, 1048577).
With 250 Hz sampling rate, the length of ECG signal is: 3495.26 minutes.
```

Each patient is labeled with beat type and rythm type:

[9]:
```
print(f"{labels.keys()}")
```

```
dict_keys(['btype', 'rtype'])
```

The beat label in the Icentia11k dataset has 5 different values, as shown in the below code cell. The beat classification task is to classify beat type given a segment or a frame of an ECG signal. The paper experimented with frame sizes ranging from 2 to 60 seconds.

[10]:
```
from pretraining import datasets

print(f"Beat labels: {datasets.icentia11k.ds_beat_names}")
```

```
Beat labels: {0: 'undefined', 1: 'normal', 2: 'pac', 3: 'aberrated', 4: 'pvc'}
```

The dict key `btype`'s value is a list of 50 elements, each element is a lst and corresponds to a row of ECG signal array mentioned above. The list's first element is the column number of the ECG signal array, and the second element is the associated label at the ECG signal.

[11]:
```
print(
    f"Patient #0's first beat label is:"
    f" {icentia11k.ds_beat_names[labels['btype'][0][1][0]]},"
)
print(
    f"at the first row and {labels['btype'][0][0][0]}th column"
    f" of the ECG signal array"
)
```

```
Patient #0's first beat label is: normal,
at the first row and 28th column of the ECG signal array
```

For beat classification pre-training, a frame of continuous ECG signal is extracted from a patient. The beat type associated with the frame is determined by the most occurrence of pac/aberrated/pvc. If none of the three types occur in the frame, the beat type is the most common between normal/undefined. The following is a snippet of the original code to extract an ECG signal frame and an associated label from a patient's data.

[12]:
```
import numpy as np


frame_size = 4096
num_segments, segment_size = signal.shape
patient_beat_labels = labels["btype"]

# original code: np.random.randint(num_segments)
segment_index = 17
# original code: np.random.randint(segment_size - frame_size)
frame_start = 45678
frame_end = frame_start + frame_size
```

```
x = signal[segment_index, frame_start:frame_end]
x = np.expand_dims(x, axis=1)  # add channel dimension

# calculate the count of each beat type in the frame and determine the final
# label
beat_ends, beat_labels = patient_beat_labels[segment_index]
_, frame_beat_labels = icentia11k.get_complete_beats(
    beat_ends, beat_labels, frame_start, frame_end
)
y = icentia11k.get_beat_label(frame_beat_labels)

print("Data sample from patient #0:")
print(
    f"ECG signal segment={segment_index}, frame_start={frame_start},"
    f" frame_end={frame_end}"
)
print(f"Number of beat labels in the frame: {len(frame_beat_labels)}")
print(f"- Undefined: {np.sum(frame_beat_labels == 0)}")
print(f"- Normal: {np.sum(frame_beat_labels == 1)}")
print(f"Final beat label of the data sample: {icentia11k.ds_beat_names[y]}")
```

```
Data sample from patient #0:
ECG signal segment=17, frame_start=45678, frame_end=49774
Number of beat labels in the frame: 26
- Undefined: 6
- Normal: 20
Final beat label of the data sample: normal
```

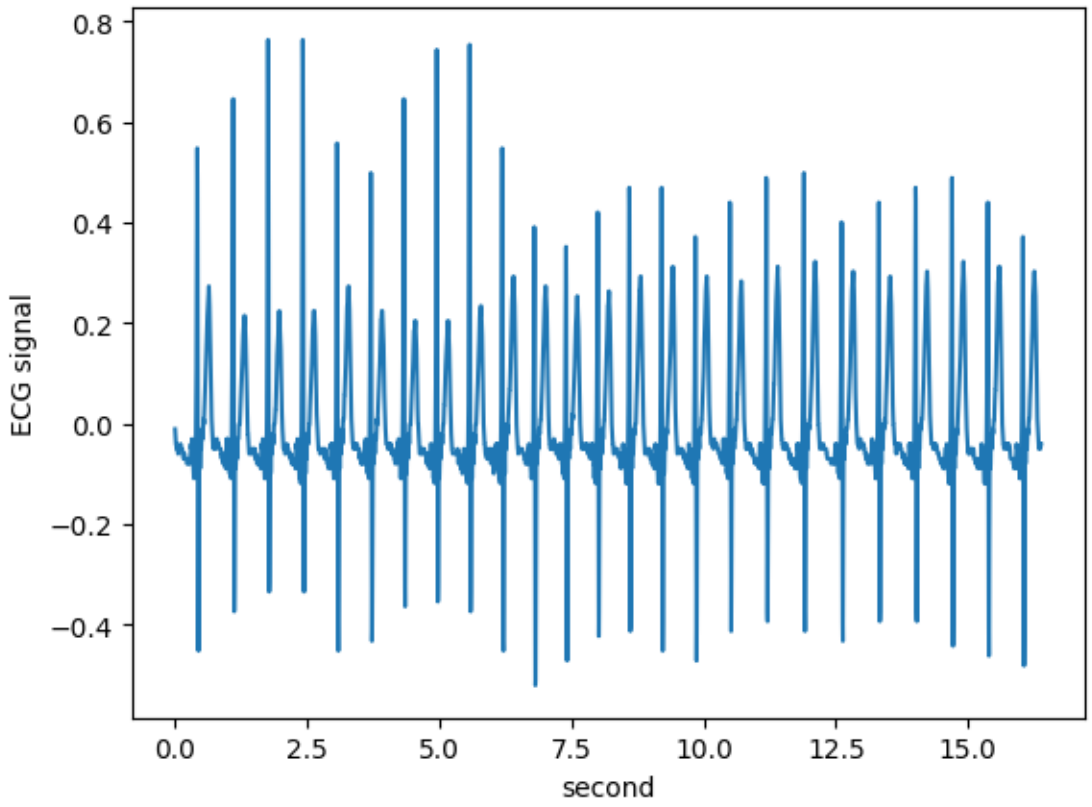
Exploration and visualizations The original code includes a `beat_generator` as data generator for beat classification task. Below is example code to generate a training data sample from Patient#0 with an ECG signal frame size of 4096:

[13]:
```
from pretraining import datasets

gen = datasets.beat_generator(
    DEMO_DATA_DIR,
    patient_ids=[0],  # Patient#0
    frame_size=4096,  # a frame with 4096 continuous ECG signal
    normalize=True,
    unzipped=True,
    samples_per_patient=1,
)
data = next(gen)
```

Each data sample is a tuple of ECG signal and beat label. With a frame size of 4096 and 250 Hz sampling rate, the length of this sample is around 16 seconds.

[14]:
```
from matplotlib import pyplot as plt
import numpy as np

print(
    f"Beat label of the data sample:"
    f" {datasets.icentia11k.ds_beat_names[data[1]]}"
)
_ = plt.plot(np.arange(len(data[0])) / 250, data[0])
_ = plt.xlabel("second")
_ = plt.ylabel("ECG signal")
```

```
Beat label of the data sample: normal
```



The `beat_generator` is used to create a `tensorflow.Dataset` object. The following code creates a `Dataset` that contains 2048 training data samples from Patient#0, each training data sample with frame size 2048.

Dataset generator code:

[15]:
```
dataset = datasets.beat_dataset(
    db_dir=DEMO_DATA_DIR,
    patient_ids=[0],
```

```
    frame_size=2048,
    unzipped=True,
    samples_per_patient=2048,
)
dataset.element_spec
```

```
WARNING:tensorflow:From /tmp/repo/pretraining/datasets.py:33: calling
DatasetV2.from_generator (from tensorflow.python.data.ops.dataset_ops) with
output_types is deprecated and will be removed in a future version.
Instructions for updating:
Use output_signature instead
WARNING:tensorflow:From /tmp/repo/pretraining/datasets.py:33: calling
DatasetV2.from_generator (from tensorflow.python.data.ops.dataset_ops) with
output_shapes is deprecated and will be removed in a future version.
Instructions for updating:
Use output_signature instead
```

[15]:
```
(TensorSpec(shape=(2048, 1), dtype=tf.float32, name=None),
 TensorSpec(shape=(), dtype=tf.int32, name=None))
```

5.2.2 Fine-tuning Dataset

The fine-tuning dataset is the "AF Classification from a Short Single Lead ECG Recording: The PhysioNet/Computing in Cardiology Challenge 2017" and freely available online for download [7][8].

Data Download Instructions Source of the data:

- Raw.

Download instructions:

- Simply do a direct download of the data from the PhysioNet website.
- The `train2017.zip` file needs to be saved somewhere for subsequent pre-processing.

Description Statistics:

- 8528 short ECG recordings.
- Each ECG recording duration is 9 to 60 seconds with 300 Hz sampling rate.
- Each ECG recording is labeled with one of the following classes: AF, Normal, Other or Noise (too noisy to classify).

Pre-processing Code and Command A brief summary of the steps in pre-processing of the fine-tuning dataset:

- Resampling to 250 Hz to match the sample rate of the pre-training dataset.
- Padding records to 65 seconds.
- Standardizing the data using mean and standard deviation computed on the entire dataset.

The `get_challenge17_data` function (source) in the authors' code both extracts and pre-processes the *PhysioNet 2017* dataset. We split the data extraction and pre-processing steps and saved the

extracted data. This lets us have the flexibility of pre-processing without data extraction in every run. The below cell demonstrates an example of data pre-processing.

[16]:
```
%%time
%%capture
# Downloads the raw PhysioNet record and label pickles, the output of
# extraction.
!gdown 1lI7lTZltRj_zQtYM1UcidZnSAQilPJSg -O {DEMO_DATA_DIR} --folder
```

```
CPU times: user 72.7 ms, sys: 27.6 ms, total: 100 ms
Wall time: 8.74 s
```

Note that downloading the extracted data took less than 10 seconds. The data extraction took 45 minutes in total, thus splitting the original `get_challenge17_data` function made our fine-tuning data preparation much more efficient.

[17]:
```
import functools

import numpy as np

from finetuning import datasets
from transplant.datasets import physionet


def extract_challenge17_data(db_dir, verbose=False):
    # This is the author's extraction code
    records, labels = physionet.read_challenge17_data(db_dir, verbose=verbose)
    return records, labels


def process_extracted_challenge17_data(
    records, labels, fs=None, pad=None, normalize=False, verbose=False
):
    """
    This is our code that performs pre-processing only.

    Args:
        records: See extract_challenge17_data.
        labels: See extract_challenge17_data.
        fs (int): Sampling rate.
        pad (int): Length that each record should be padded to (or truncated)
            pad / fs will give the approximate length in seconds.
        normalize (bool): Whether to standardize the records using mean and s.d.
            computed over the entire dataset. Note that the mean and s.d. have
            already been provided by the authors.
    """
    if normalize:
        normalize = functools.partial(
```

```
            physionet.normalize_challenge17, inplace=True
        )
    # This is the author's pre-processing code
    data_set = datasets._prepare_data(
        records,
        labels,
        normalize_fn=normalize,
        fs=fs,
        pad=pad,
        verbose=verbose,
    )
    return data_set
```

[18]:
```
%%time
from transplant.utils import load_pkl

records = load_pkl(f"{DEMO_DATA_DIR}/records.pkl")["data"]
labels = load_pkl(f"{DEMO_DATA_DIR}/labels.pkl")["data"]
physionet_data = process_extracted_challenge17_data(
    records,
    labels,
    fs=250,
    pad=16384,
    normalize=True,
    verbose=True,
)
```

```
Resampling records: 100%|      | 8528/8528 [00:03<00:00, 2187.59it/s]

CPU times: user 7.18 s, sys: 1.29 s, total: 8.47 s
Wall time: 8.43 s
```

Exploration and Visualization Please note that the exploration is done on the pre-processed data.

Labels are in the form one-hot encoded arrays. The labels `A`, `N`, `O` and `~` represent AF, Normal, Other and Noise, respectively.

The distribution of label values is shown below. Note that less than 10% (738 / 8528) of the data is labeled AF.

[19]:
```
print(f"Shape of the fine-tuning data labels: {labels.shape}")
labels.value_counts()
```

```
Shape of the fine-tuning data labels: (8528, 4)
```

[19]:
```
A  N  O  ~
0  1  0  0    5050
   0  1  0    2456
```

```
1  0  0  0     738
0  0  0  1     284
Name: count, dtype: int64
```

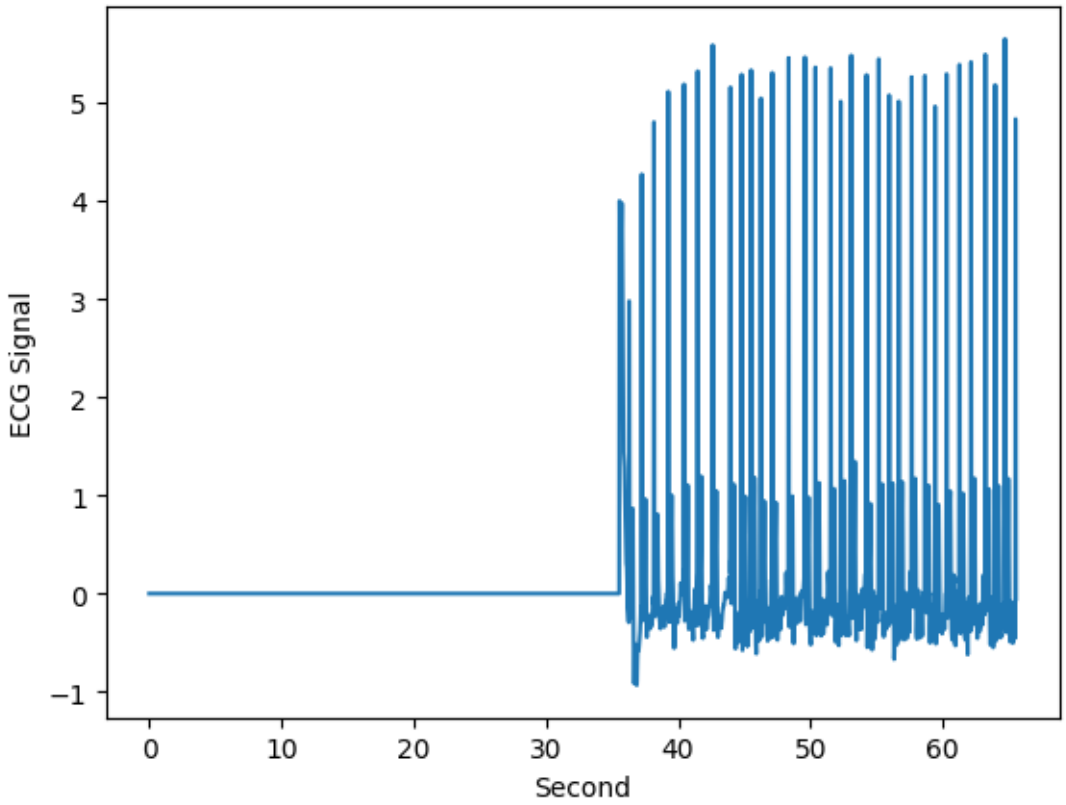
The `_prepare_data()` function packs ECG signals (resampled and padded) and labels into a dictionary ready to be consumed by `train_test_split` in `sklearn`. To save data pre-processing time in the fine-tuning procedure, we split the pre-processed data (i.e. the `physionet_data` variable) into train and test sets and saved them in pickle form.

[20]:
```
print("PhysioNet data summary:")
print(f'Number of samples: {len(physionet_data["x"])}')
print(f"labels: {physionet_data['classes']}")
print(f'Dimensions of a sample: {physionet_data["x"][0].shape}')
```

```
PhysioNet data summary:
Number of samples: 8528
labels: ['A' 'N' 'O' '~']
Dimensions of a sample: (16384, 1)
```

[21]:
```
print(f'Record id: {physionet_data["record_ids"][3]}')
print(f'One-hot encoded label: {physionet_data["y"][3]}')
_ = plt.plot(
    np.arange(physionet_data["x"][3].shape[0]) / 250, physionet_data["x"][3]
)
_ = plt.xlabel("Second")
_ = plt.ylabel("ECG Signal")
```

```
Record id: A00004
One-hot encoded label: [1 0 0 0]
```



5.3 Model

In this project, the CNN model of choice is ResNet-18v2. This was used in the original paper [1].

Link to the original paper's model definition. This implementation code is also in our repo here.

The ResNet architecture is based on the original design by He et al. [9]

We show how to use the paper authors' code to create a 1-D ResNet-18v2 for beat classification in pre-training and AF classification in fine-tuning.

- Model architecture
  - 18 layers
  - Input layer consists of convolution layer with 64 filters, kernel size=3 and stride=2. The output of the convolution layer passes through batch norm, ReLu and max-pooling layers sequentially.
  - Output layer is a classifier consisting of a densely-connected layer followed by softmax function.
  - The middle 16 layers consist of 8 residual blocks. A residual block consists of the following two components and outputs the sum of the two components' outputs.
    1. Two convolution layers, each followed by batch norm and ReLu.
    2. A shortcut that passes the input through a convolution layer followed by batch norm.

    - Configurations of the residual blocks
        * 1st and 2nd: 64 filters, kernel size=7, strides=2 and 1, respectively
        * 3rd and 4th: 128 filters, kernel size=5, strides=2 and 1, respectively
        * 5th and 6th: 256 filters, kernel size=5, strides=2 and 1, respectively
        * 7th and 8th: 512 filters, kernel size=3, strides=2 and 1, respectively
    - Detail of implementation code here.
- Pre-training objectives
    - Loss function: Sparse Categorical Cross Entropy
        * `from_logits=True`
        * All other parameters are set to their default values.
    - Optimizer: Adam
        * `learning_rate`: 0.001 (default)
        * `beta_1`: 0.9
        * `beta_2`: 0.98
        * All other parameters are set to their default values.
    - Metric: Sparse Categorical Accuracy
- Fine-tuning objectives
    - Loss function: Categorical Cross Entropy
        * All parameters are set to their default values.
    - Optimizer: Adam
        * `learning_rate`: 0.001 (default)
        * All other parameters are set to their default values.
    - Metrics:
        * Accuracy
        * Macro F1. See our `my_f1()` function below.

We also include Google Drive links to pre-trained model weights for reader's inspection:

- Pre-trained model weights with patients 0-2047 of the *Icentia11k* dataset (~20%) are here.
- Pre-trained model weights we trained with all patients of the *Icentia11k* dataset (100%) are here.

5.3.1 Pre-training Model Initialization and Summary

Below we construct the pre-training ResNet-18v2 and show its structure.

[22]:
```
import tensorflow as tf

from transplant.modules.resnet1d import ResNet
from transplant.datasets import icentia11k
from transplant.modules.utils import build_input_tensor_from_shape


resnet = ResNet(
    num_outputs=None,
    blocks=(2, 2, 2, 2),
    kernel_size=(7, 5, 5, 3),
    include_top=False,
)
```

```
feature_extractor = tf.keras.Sequential(
    [resnet, tf.keras.layers.GlobalAveragePooling1D()]
)

num_classes = len(icentia11k.ds_beat_names)  # 5

model = tf.keras.Sequential(
    [feature_extractor, tf.keras.layers.Dense(num_classes)]
)

model.compile(
    optimizer=tf.keras.optimizers.Adam(beta_1=0.9, beta_2=0.98, epsilon=1e-9),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=[tf.keras.metrics.SparseCategoricalAccuracy(name="acc")],
)

# Initialize the weights of the model
train_data = dataset.batch(32)
input_shape, _ = tf.compat.v1.data.get_output_shapes(train_data)
input_dtype, _ = tf.compat.v1.data.get_output_types(train_data)
inputs = build_input_tensor_from_shape(
    input_shape, dtype=input_dtype, ignore_batch_dim=True
)
model(inputs)
model.summary(expand_nested=True)
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 sequential (Sequential)     (None, 512)               4492480
|---------------------------------------------------------------|
| res_net (ResNet)           (None, 64, 512)           4492480  |
||-------------------------------------------------------------||
|| conv1d (Conv1D)           multiple                  448      ||
||                                                               ||
|| batch_normalization (Bat  multiple                  256      ||
|| chNormalization)                                              ||
||                                                               ||
|| re_lu (ReLU)              multiple                  0        ||
||                                                               ||
|| max_pooling1d (MaxPoolin  multiple                  0        ||
|| g1D)                                                          ||
||                                                               ||
|| residual_block (Residual  multiple                  57856    ||
|| Block)                                                        ||
```

```
||                                                                  ||
|| residual_block_1 (Residu  multiple                 57856   ||
|| alBlock)                                                        ||
||                                                                  ||
|| residual_block_2 (Residu  multiple                 132608  ||
|| alBlock)                                                        ||
||                                                                  ||
|| residual_block_3 (Residu  multiple                 164864  ||
|| alBlock)                                                        ||
||                                                                  ||
|| residual_block_4 (Residu  multiple                 527360  ||
|| alBlock)                                                        ||
||                                                                  ||
|| residual_block_5 (Residu  multiple                 657408  ||
|| alBlock)                                                        ||
||                                                                  ||
|| residual_block_6 (Residu  multiple                 1316864 ||
|| alBlock)                                                        ||
||                                                                  ||
|| residual_block_7 (Residu  multiple                 1576960 ||
|| alBlock)                                                        ||
|-----------------------------------------------------------------|
| global_average_pooling1d   (None, 512)             0        |
| (GlobalAveragePooling1D)                                        |
-------------------------------------------------------------------

 dense (Dense)               (None, 5)                 2565

=================================================================
Total params: 4495045 (17.15 MB)
Trainable params: 4485445 (17.11 MB)
Non-trainable params: 9600 (37.50 KB)
_________________________________________________________________
```

5.3.2 Fine-tuning Model Initialization and Summary

The fine-tuning model in the paper is the pre-training model with its output layer replaced with a fully connected layer that matches the classes of the *PhysioNet 2017* dataset and has randomly initialized weights. Below is a demonstration to construct fine-tuning CNN model. Note the model's number of parameters is identical to that of the pre-training CNN, and the output layer has four outputs, instead of five.

[23]:
```python
resnet2 = ResNet(
    num_outputs=None,
    blocks=(2, 2, 2, 2),
    kernel_size=(7, 5, 5, 3),
    include_top=False,
)
```

```python
ft_model = tf.keras.Sequential(
    [resnet2, tf.keras.layers.GlobalAveragePooling1D()]
)

# Initialize the weights of the model
inputs = tf.keras.layers.Input(
    physionet_data["x"].shape[1:], dtype=physionet_data["x"].dtype
)
ft_model(inputs)

# Load pre-training weights, if any
pre_trained_weights = None
if pre_trained_weights:
    ft_model.load_weights(pre_trained_weights)

# Replace output layer
num_classes = len(physionet_data["classes"])  # 4
ft_model.add(
    tf.keras.layers.Dense(num_classes, activation="softmax", name="new_dense")
)

ft_model.compile(
    optimizer=tf.keras.optimizers.Adam(),
    loss=tf.keras.losses.CategoricalCrossentropy(),
    metrics=[tf.keras.metrics.CategoricalAccuracy(name="acc")],
)
ft_model.summary(expand_nested=True)
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 res_net_1 (ResNet)          (None, 512, 512)          4492480
|---------------------------------------------------------------|
| conv1d_1 (Conv1D)          multiple                  448      |
|                                                               |
| batch_normalization_1 (Ba  multiple                  256      |
| tchNormalization)                                             |
|                                                               |
| re_lu_1 (ReLU)             multiple                  0        |
|                                                               |
| max_pooling1d_1 (MaxPooli  multiple                  0        |
| ng1D)                                                         |
|                                                               |
| residual_block_8 (Residua  multiple                  57856    |
| lBlock)                                                       |
|                                                               |
```

```
| residual_block_9 (Residua  multiple                  57856    |
| lBlock)                                                       |
|                                                               |
| residual_block_10 (Residu  multiple                  132608   |
| alBlock)                                                      |
|                                                               |
| residual_block_11 (Residu  multiple                  164864   |
| alBlock)                                                      |
|                                                               |
| residual_block_12 (Residu  multiple                  527360   |
| alBlock)                                                      |
|                                                               |
| residual_block_13 (Residu  multiple                  657408   |
| alBlock)                                                      |
|                                                               |
| residual_block_14 (Residu  multiple                  1316864  |
| alBlock)                                                      |
|                                                               |
| residual_block_15 (Residu  multiple                  1576960  |
| alBlock)                                                      |
-----------------------------------------------------------------

 global_average_pooling1d_1  (None, 512)              0
  (GlobalAveragePooling1D)

 new_dense (Dense)           (None, 4)                2052

=================================================================
Total params: 4494532 (17.15 MB)
Trainable params: 4484932 (17.11 MB)
Non-trainable params: 9600 (37.50 KB)
-----------------------------------------------------------------
```

5.4 Training

5.4.1 Hyperparameters

Pre-training

- Optimizer: Adam
  - `learning_rate`: 0.001 (default)
  - `beta_1`: 0.9
  - `beta_2`: 0.98
  - All other parameters are set to their default values.
- Sample frequency: 250 Hz
- Frame Size: 4096 (about 16 seconds at 250 Hz)
- Batch Size: 512
- `samples_per_patient`: 4096
- Hidden size: Please refer to the "Models" section for details.

Fine-tuning

- Optimizer: Adam
  - `learning_rate`: 0.001 (default)
  - All other parameters are set to their default values.
- Batch size: 128
  - This size was chosen to maximize the GPU RAM usage of the V100 GPU on Google Colab.
- Sample rate: 250 Hz (resampled from 300 Hz)
- Frame size: 16384
  - Approximately 65 seconds
  - Shorter samples are left-padded with zeroes
- Hidden size: Please refer to the "Models" section for details.

5.4.2 Computational Requirements

Pre-training

- Type of Hardware: V100 GPU
  - Batch size of 512 used to fully utilize the 16 GB of RAM on the V100 GPU.
  - However, our training rate has been limited by not by the GPU speed, but the I/O time required to transfer, load, and unzip the data files from google drive distributed file system. For example, training on 32x more data per patient only increased runtime by 2.2x, since most of the time was spent loading the same amount of patient data files.
  - We experimented with using A100 GPUs which has more RAM (i.e. larger batch size) and is generally faster. But we were unable to get significant speed improvements to justify the extra hourly cost of the A100 GPU due to distributed I/O latency associated with Google Drive.
- Average runtime per 1,000 patients: 1 hours, 16 minutes
- Total number of trials: 1
- GPU hours used: 14 hours
  - Frame size 4096
  - 11000 patients x 4096 samples per data = **45 million samples**

Note on epochs: Both the author and our analysis only ever uses a patient's data once during pre-training, thus the 1 epoch. However, the code's definition of an "epoch" during pre-training has a different meaning and refers to the processing one set of data equal to `batch_size x steps_per_epoch`. This can have variable amounts of data or patients.

Also note that during pre-training, we saved 100 checkpoints to be able to study how fine-tuning on different amounts of pre-training affects the final model's performance after fine-tuning.

Fine-tuning

- Type of Hardware: V100 GPU
  - Batch size of 128 used to fully utilize the 16 GB of RAM on the V100 GPU.
  - Unlike in pre-training, we did not experience I/O overhead during training as the processed *PhysioNet 2017* train and test data (pickles) were able to fit into RAM.
- Average runtime for each epoch: Between 21 to 22 seconds.
- Total number of trials: 50

  - 10 trials for each of the five scenarios: Random, 1%, 10%, 20%, and 100%.
- GPU hours used: About 21.2 hours total.
  - Random: 5.3 hours
  - 1%: 3.9 hours
  - 10%: 4.1 hours
  - 20%: 4.3 hours
  - 100%: 3.6 hours
- Number of training epochs: Maximum 200 per trial, but in practice, saw early stopping by 80 epochs.
  - Training for a single trial would typically complete within 30 minutes.

5.4.3 Training Code Snippet Sample

Pre-training For beat classification pre-training, a checkpoint function is created to monitor training loss and save model weights of each epoch. The below cell demonstrates pre-training with a small dataset.

- Note that the model weights of each epoch are saved for fine-tuning model initialization.
- Also note that in `keras` and the way the authors implemented the pre-training step, 1 epoch is defined by batch size and steps-per-epoch (it does not imply training on the entire dataset once).

[24]:
```
FINAL_DEMO_JOB = JOB_DIR + "/final_demo"

checkpoint = tf.keras.callbacks.ModelCheckpoint(
    filepath=str(FINAL_DEMO_JOB + "/epoch_{epoch:02d}" + "/model.weights"),
    monitor="loss",
    save_best_only=False,
    save_weights_only=True,
    mode="auto",
    verbose=1,
)

_ = model.fit(
    train_data,
    steps_per_epoch=64,
    verbose=2,
    epochs=2,
    validation_data=None,
    callbacks=[checkpoint],
)
```

```
Epoch 1/2

Epoch 1: saving model to /tmp/jobs/final_demo/epoch_01/model.weights
64/64 - 17s - loss: 0.3047 - acc: 0.9312 - 17s/epoch - 268ms/step
Epoch 2/2

Epoch 2: saving model to /tmp/jobs/final_demo/epoch_02/model.weights
```

```
64/64 - 2s - loss: 0.0992 - acc: 0.9741 - 2s/epoch - 29ms/step
```

Fine-tuning The procedure to fine-tune the 1-D ResNet-18v2 model is identical to the pre-training procedure, except that a custom checkpoint function is used to calculate macro F1 score on the validation dataset at the end of a training epoch and save the model weights if a better macro F1 score is reported. The following is an example of the `CustomCheckpoint` from the original code, which we do not run in this notebook.

[25]:
```
from transplant.evaluation import f1, CustomCheckpoint

val_data = physionet_data["x"][3]
val_y = physionet_data["y"][3]

checkpoint = CustomCheckpoint(
    filepath=str(FINAL_DEMO_JOB + "/fine-tuning/best_model.weights"),
    data=(val_data, val_y),
    score_fn=f1,
    save_best_only=True,
    verbose=1,
)
```

5.4.4 Training Code - Command Line API

The paper authors provide entrypoint scripts to run the entire pre-training and fine-tuning process with parameters of choice. The following is the high level description of how the scripts work.

Pre-training The source code from the paper authors can be found here.

1. Create train/validate data generator based on patient ID and the number of samples per patient, both specified when calling the entrypoint.
2. A model is generated based on the model architecture and pre-training task specified by the user.
3. Weights of the model are initialized. They can also be loaded from a weights file. For all pre-training in the project, we don't load weights.
4. Checkpoint function is created based on training metric. For pre-training, we use `loss` as training metric.
5. The model fits the train data. At the end of each training epoch, the checkpoint function is called for evaluation and save the model weights.

The paper uses 95% of the patient's ECG data. On average, the paper sampled 4096 ECG frames per patient, which amounts to 42.8 million (11000x0.95x4096) training samples over the course of pre-training. For pre-training with 20% of the data used in paper, we use ECG data from 2048 patients and sample 4096 ECG frames per patient, resulting to roughly 8.4 million (2048x4096) training samples.

The following sample command calls the pre-training entrypoint to run pre-training with 20% of the data used in the paper (Patients 0 through 2047).

- The command is easily adapted to run pre-training with 10% of the data in the paper.

```
!time python -m pretraining.trainer \
--job-dir "jobs/beat_classification_16epochs_to_20percent" \
--task "beat" \
--train {TRAIN_DATASET} \
--arch "resnet18" \
--epochs 16 \
--patient-ids `seq 0 2047 | paste -sd, -` \
--steps-per-epoch 1024 \
--samples-per-patient 4096 \
--batch-size 512 \
--frame-size 4096
```

Explanation of the less obvious parameters:

- `--job-dir`: Output directory, where check points and weights are saved.
- `--task`: Pre-training task, `beat` for Beat classification.
- `--patient-ids`: Patient ID(s) whose ECG data to be used in pre-training.
  - Multiple patients can be specified by passing a space separated sequence of IDs.
- `--frame-size`: Number of ECG samples, with 250 Hz sampling rate, in a ECG frame.

To use all data: number of patients x samples-per-patient = epochs x batch_size x steps-per-epoch.

For demonstration, we run the script in a much smaller scale below:

[26]:
```
PRETRAIN_JOB_DIR = FINAL_DEMO_JOB + "/pretraining"
```

[27]:
```
%%time
!python -m pretraining.trainer \
--job-dir {PRETRAIN_JOB_DIR} \
--task "beat" \
--train {DEMO_DATA_DIR} \
--arch "resnet18" \
--epochs 2 \
--patient-ids 0 \
--steps-per-epoch 8 \
--samples-per-patient 4096 \
--batch-size 256 \
--frame-size 1024 \
--unzipped True \
--seed 2024
```

```
2024-05-07 01:38:13.136488: E
external/local_xla/xla/stream_executor/cuda/cuda_dnn.cc:9261] Unable to register
cuDNN factory: Attempting to register factory for plugin cuDNN when one has
already been registered
2024-05-07 01:38:13.136535: E
external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:607] Unable to register
cuFFT factory: Attempting to register factory for plugin cuFFT when one has
```

```
already been registered
2024-05-07 01:38:13.138311: E
external/local_xla/xla/stream_executor/cuda/cuda_blas.cc:1515] Unable to
register cuBLAS factory: Attempting to register factory for plugin cuBLAS when
one has already been registered
2024-05-07 01:38:14.182325: W
tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not
find TensorRT
Creating working directory in /tmp/jobs/final_demo/pretraining
Setting random state 2024
Building train data generators
# Patient IDs:  1
WARNING:tensorflow:From /tmp/repo/pretraining/datasets.py:33: calling
DatasetV2.from_generator (from tensorflow.python.data.ops.dataset_ops) with
output_types is deprecated and will be removed in a future version.
Instructions for updating:
Use output_signature instead
WARNING:tensorflow:From /tmp/repo/pretraining/datasets.py:33: calling
DatasetV2.from_generator (from tensorflow.python.data.ops.dataset_ops) with
output_shapes is deprecated and will be removed in a future version.
Instructions for updating:
Use output_signature instead
2024-05-07 01:38:15.698961: W
tensorflow/core/common_runtime/gpu/gpu_bfc_allocator.cc:47] Overriding
orig_value setting because the TF_FORCE_GPU_ALLOW_GROWTH environment variable is
set. Original config value was 0.
Building model …
# model parameters: 4,495,045
Epoch 1/2
WARNING: All log messages before absl::InitializeLog() is called are written to
STDERR
I0000 00:00:1715045909.955505    4990 device_compiler.h:186] Compiled cluster
using XLA!  This line is logged at most once for the lifetime of the process.

Epoch 1: saving model to /tmp/jobs/final_demo/pretraining/epoch_01/model.weights
8/8 - 19s - loss: 0.9372 - acc: 0.7720 - 19s/epoch - 2s/step
Epoch 2/2

Epoch 2: saving model to /tmp/jobs/final_demo/pretraining/epoch_02/model.weights
8/8 - 1s - loss: 0.1225 - acc: 0.9712 - 1s/epoch - 151ms/step
Exception ignored in: <function AtomicFunction.__del__ at 0x7f62f4f17880>
Traceback (most recent call last):
  File "/usr/local/lib/python3.10/dist-
packages/tensorflow/python/eager/polymorphic_function/atomic_function.py", line
291, in __del__
TypeError: 'NoneType' object is not subscriptable
CPU times: user 182 ms, sys: 31 ms, total: 213 ms
Wall time: 28.1 s
```

Fine-tuning The source code from the paper authors can be found here.

1. The *PhysioNet 2017* dataset has already been split into train and test datasets (80%-20% split) and are passed to the entrypoint. The validation dataset will be further separated from the train dataset based on user input.
2. A CNN model is generated based on the model architecture specified by the user. Its output layer is fully connected with softmax activation.
3. The weights of the model are initialized. They can also be loaded from a weights file. If using a weights model from pre-training, this first has to be pre-processed using the "pre-training.utils.get_pretrained_weights" function to remove the classification layer, which is replaced during fine-tuning. An example of this process can be see here
4. Checkpoint function is created based on validation metric. For fine-tuning, we use `f1` (macro F1) as the metric.
5. The model fits the train data. At the end of each training epoch, the checkpoint function is called for evaluation and saves the model weights with best macro F1 score on the validation set. Note that the model also is setup to end training early if the validation loss does not decrease after 50 epochs.
6. The model is evaluated on the test set and the predicted probabilities are saved to a csv file.

Ultimately, the two most important outputs of a fine-tuning execution are:

- History file containing validation macro F1 score at the end of each epoch.
- Model predictions on the test set.

The following sample command calls the fine-tuning entrypoint to run fine-tuning with *random initialization*.

```
python -m finetuning.trainer \
--job-dir {JOB_DIR} \
--train {FINETUNE_TRAIN} \
--test {FINETUNE_TEST} \
--val-size 0.0625 \
--val-metric "f1" \
--arch "resnet18" \
--batch-size 128 \
--epochs 200 \
--seed 2024 \
--verbose
```

Explanation of the less obvious parameters:

- `--job-dir`: Output directory, where checkpoints and weights are saved.
- `--val-size`: This is the percentage of the train set size to set aside for the validation set.
  - Note that the *PhysioNet 2017* data was already split into 80% train, 20% test. The paper uses 5% of the full dataset for validation.
  - Math: $6.25\% * 80\% = 5\%$.
- `--val-metric`: Metric to evaluate the model at the end of each epoch on the validation dataset.
- `--seed`: Random state used to split the train into smaller train and validation.

Below is a small scale demonstration of fine-tuning with random weight initialization. We first download the fine-tuning train and test datasets.

[28]:
```
FINETUNE_TRAIN = DEMO_DATA_DIR + "/physionet_train.pkl"
FINETUNE_TEST = DEMO_DATA_DIR + "/physionet_test.pkl"
```

[29]:
```
%%capture
!gdown 10PVz1nmMaeIgxQ4sqDUeZlzhFVHOth5I -O {FINETUNE_TRAIN}
!gdown 10FKjncGOZD6_BBCBbvpBQbWvfCLj88MC -O {FINETUNE_TEST}
```

[30]:
```
FINETUNE_JOB_DIR = FINAL_DEMO_JOB + "/finetuning_random"
```

Then, we run the fine-tuning entrypoint and run fine-tuning for two epochs.

[31]:
```
%%time
!python -m finetuning.trainer \
--job-dir {FINETUNE_JOB_DIR} \
--train {FINETUNE_TRAIN} \
--test {FINETUNE_TEST} \
--val-size 0.0625 \
--val-metric "f1" \
--arch "resnet18" \
--batch-size 128 \
--epochs 2 \
--seed 2024 \
--verbose
```

```
2024-05-07 01:38:53.286305: E
external/local_xla/xla/stream_executor/cuda/cuda_dnn.cc:9261] Unable to register
cuDNN factory: Attempting to register factory for plugin cuDNN when one has
already been registered
2024-05-07 01:38:53.286352: E
external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:607] Unable to register
cuFFT factory: Attempting to register factory for plugin cuFFT when one has
already been registered
2024-05-07 01:38:53.287814: E
external/local_xla/xla/stream_executor/cuda/cuda_blas.cc:1515] Unable to
register cuBLAS factory: Attempting to register factory for plugin cuBLAS when
one has already been registered
2024-05-07 01:38:54.302713: W
tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not
find TensorRT
Creating working directory in /tmp/jobs/final_demo/finetuning_random
Setting random state 2024
Loading train data from /tmp/data/final_demo/physionet_train.pkl …
Split data into train 93.73% and validation 6.27%
Loading test data from /tmp/data/final_demo/physionet_test.pkl …
Train data shape: (1599, 16384, 1)
2024-05-07 01:39:02.797333: W
tensorflow/core/common_runtime/gpu/gpu_bfc_allocator.cc:47] Overriding
orig_value setting because the TF_FORCE_GPU_ALLOW_GROWTH environment variable is
```

```
set. Original config value was 0.
Building model …
# model parameters: 4,494,532
Epoch 1/2
WARNING: All log messages before absl::InitializeLog() is called are written to
STDERR
I0000 00:00:1715045959.930772    6448 device_compiler.h:186] Compiled cluster
using XLA!  This line is logged at most once for the lifetime of the process.
1/1 [==============================] - 1s 911ms/step

Epoch 00001: f1 improved from -inf to 0.22910, saving model to
/tmp/jobs/final_demo/finetuning_random/best_model.weights
13/13 - 45s - loss: 1.1976 - acc: 0.5641 - val_loss: 2.9232 - val_acc: 0.4860 -
f1: 0.2291 - 45s/epoch - 3s/step
Epoch 2/2
1/1 [==============================] - 0s 120ms/step

Epoch 00002: f1 improved from 0.22910 to 0.26326, saving model to
/tmp/jobs/final_demo/finetuning_random/best_model.weights
13/13 - 11s - loss: 0.9507 - acc: 0.5972 - val_loss: 1.0752 - val_acc: 0.5794 -
f1: 0.2633 - 11s/epoch - 828ms/step
Loading the best weights from file
/tmp/jobs/final_demo/finetuning_random/best_model.weights …
Predicting training data …
13/13 [==============================] - 11s 371ms/step
Predicting validation data …
1/1 [==============================] - 0s 74ms/step
Predicting test data …
54/54 [==============================] - 9s 173ms/step
CPU times: user 551 ms, sys: 89.9 ms, total: 641 ms
Wall time: 1min 32s
```

Observe that after the second epoch completes, the entrypoint then loads the best performing model and uses it to perform inference on the test set.

The following sample command uses the fine-tuning entrypoint to run fine-tuning with pre-trained weights.

```
python -m finetuning.trainer \
--job-dir {JOB_DIR} \
--train {FINETUNE_TRAIN} \
--test {FINETUNE_TEST} \
--weights-file {WEIGHTS_FILE} \
--val-size 0.0625 \
--val-metric "f1" \
--arch "resnet18" \
--batch-size 128 \
--epochs 200 \
--seed 2024 \
```

```
--verbose
```

This is largely the same command as the previous example for fine-tuning a randomly initialized network. But with this addition:

- `--weights-file {WEIGHTS_FILE}`: Path to pre-trained weights or a checkpoint of the model to be used for model initialization.

Below is a small demonstration using weights obtained from pre-training using 20% of the *Icentia11k* dataset. We first download model weights from our experiments.

[32]:
```
%%time
%%capture
!mkdir -p {JOB_DIR + '/finetune_pretrain_20_weights_65sec'}
!gdown 1-AOAqZe9sanj-8MZnoX4TUAYWrgcUTlg \
    -O {JOB_DIR + '/finetune_pretrain_20_weights_65sec'} --folder
```

```
CPU times: user 119 ms, sys: 23 ms, total: 142 ms
Wall time: 22.5 s
```

[33]:
```
WEIGHTS_FILE = (
    JOB_DIR + "/finetune_pretrain_20_weights_65sec/best_model.weights"
)
FINETUNE_JOB_DIR = FINAL_DEMO_JOB + "/finetuning_pretrained"
```

[34]:
```
%%time
!python -m finetuning.trainer \
--job-dir {FINETUNE_JOB_DIR} \
--train {FINETUNE_TRAIN} \
--test {FINETUNE_TEST} \
--weights-file {WEIGHTS_FILE} \
--val-size 0.0625 \
--val-metric "f1" \
--arch "resnet18" \
--batch-size 128 \
--epochs 2 \
--seed 2024 \
--verbose
```

```
2024-05-07 01:40:48.556764: E
external/local_xla/xla/stream_executor/cuda/cuda_dnn.cc:9261] Unable to register
cuDNN factory: Attempting to register factory for plugin cuDNN when one has
already been registered
2024-05-07 01:40:48.556811: E
external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:607] Unable to register
cuFFT factory: Attempting to register factory for plugin cuFFT when one has
already been registered
2024-05-07 01:40:48.558522: E
external/local_xla/xla/stream_executor/cuda/cuda_blas.cc:1515] Unable to
```

```
register cuBLAS factory: Attempting to register factory for plugin cuBLAS when
one has already been registered
2024-05-07 01:40:49.566394: W
tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not
find TensorRT
Creating working directory in /tmp/jobs/final_demo/finetuning_pretrained
Setting random state 2024
Loading train data from /tmp/data/final_demo/physionet_train.pkl …
Split data into train 93.73% and validation 6.27%
Loading test data from /tmp/data/final_demo/physionet_test.pkl …
Train data shape: (1599, 16384, 1)
2024-05-07 01:40:58.120566: W
tensorflow/core/common_runtime/gpu/gpu_bfc_allocator.cc:47] Overriding
orig_value setting because the TF_FORCE_GPU_ALLOW_GROWTH environment variable is
set. Original config value was 0.
Building model …
# model parameters: 4,494,532
Loading weights from file
/tmp/jobs/finetune_pretrain_20_weights_65sec/best_model.weights …
Epoch 1/2
WARNING: All log messages before absl::InitializeLog() is called are written to
STDERR
I0000 00:00:1715046075.660058    8488 device_compiler.h:186] Compiled cluster
using XLA!  This line is logged at most once for the lifetime of the process.
1/1 [==============================] - 1s 929ms/step

Epoch 00001: f1 improved from -inf to 0.73111, saving model to
/tmp/jobs/final_demo/finetuning_pretrained/best_model.weights
13/13 - 46s - loss: 0.4518 - acc: 0.8455 - val_loss: 0.4297 - val_acc: 0.8505 -
f1: 0.7311 - 46s/epoch - 4s/step
Epoch 2/2
1/1 [==============================] - 0s 119ms/step

Epoch 00002: f1 improved from 0.73111 to 0.76197, saving model to
/tmp/jobs/final_demo/finetuning_pretrained/best_model.weights
13/13 - 11s - loss: 0.3807 - acc: 0.8574 - val_loss: 0.3963 - val_acc: 0.8692 -
f1: 0.7620 - 11s/epoch - 835ms/step
Loading the best weights from file
/tmp/jobs/final_demo/finetuning_pretrained/best_model.weights …
Predicting training data …
13/13 [==============================] - 11s 374ms/step
Predicting validation data …
1/1 [==============================] - 0s 74ms/step
Predicting test data …
54/54 [==============================] - 9s 173ms/step
CPU times: user 519 ms, sys: 94 ms, total: 613 ms
Wall time: 1min 33s
```

Observe that after the second epoch completes, the entrypoint then loads the best performing model and uses it to perform inference on the test set.

5.5 Evaluation

5.5.1 Pre-training Evaluation

The original paper doesn't discuss the performance of the 1-D ResNet-18v2 on pre-training task and focuses the analysis on the comparison between the fine-tuned model with random initialized weights and fine-tuned models with pre-trained weights. In addition, our Hypotheses 1 and 2 are about the impact on the fine-tuning task performance. Therefore, we don't evaluate pre-training task performance in this report.

5.5.2 Fine-tuning Evaluation

Metrics Descriptions In fine-tuning, the paper uses macro F1 score for evaluating the model on the *PhysioNet 2017* validation and test sets.

Evaluation Code We define our macro F1 function `my_f1`, shown below and is also on Github here. It is named like that because the paper authors define their F1 function named `f1` and we wish to avoid namespace collision. Additionally, our macro F1 function supports providing per-class F1 scores, which the *original* `f1` function does not support.

See also the Analysis section for how we are using this metric in our results.

[35]:
```
from sklearn.metrics import f1_score
import numpy as np


def my_f1(y_true, y_prob, average="macro"):
    # set average=None to get per-class F1 scores.
    y_pred = y_prob >= np.max(y_prob, axis=1)[:, None]
    return f1_score(y_true, y_pred, average=average)
```

6 Results

As specified, we used the fine-tuning entrypoint script to fine-tune a 1-D ResNet-18v2 model with the following five scenarios relating to the weights used to initialize the model before commencing fine-tuning:

- Random initialization.
- Pre-training weights from training with 1% of the data.
- Pre-training weights from training with 10% of the data.
- Pre-training weights from training with 20% of the data.
- Pre-training weights from training with 100% of the data.

Then we evaluate each model using macro F1 score on the following two datasets:

- Validation set during fine-tuning training.
- Test set.

Finally, we compare the results of all five models.

For each scenario, we ran fine-tuning 10 times with 10 different seeds. We did this in order to obtain average macro F1 scores, just as was done by the paper authors. Recall that the seed controls the split of the input data into a train set and validation set.

The same 10 seeds are used to fine-tune all five scenarios to enable apples-to-apples comparison of the macro F1 scores among the three models. Specifically, we used seeds 10, 20, ..., 100.

To facilitate analysis of the results, we have separately collated the output files from each trial (scenario and seed combination) into two files:

- All history csv file
- All test predictions pickle

More details may be found in this notebook.

In the below cell, we download the collated fine-tuning result files.

[36]:
```
RESULT1D_DIR = "/tmp/results1d"
```

[37]:
```
%%time
%%capture
!mkdir -p {RESULT1D_DIR}
!gdown 1emWqhhDG4fp-Io8dPPa6CiKGS0EgQUVJ -O {RESULT1D_DIR} --folder
```

```
CPU times: user 53.8 ms, sys: 8.17 ms, total: 61.9 ms
Wall time: 10.2 s
```

6.1 Figures and Tables

In this section, we generate the tables and figures to then be referenced in the Analysis section below, as well as in the Discussion. To keep this report short, the analysis code is located in a separate file and imported into this notebook. The code may be inspected on Github here.

Below we import the code as the `analysis1d` module.

[38]:
```
from report import analysis1d
```

6.1.1 Table 1: Epoch Comparison

We create a table of the average and standard deviation of the number of epochs before training is stopped early due to validation loss not decreasing for 50 epochs for each scenario. The `make_epoch_table` code can be found on Github here.

[39]:
```
import pandas as pd

history_all = pd.read_csv(RESULT1D_DIR + "/history_all.csv")
analysis1d.make_epoch_table(history_all)
```

[39]:

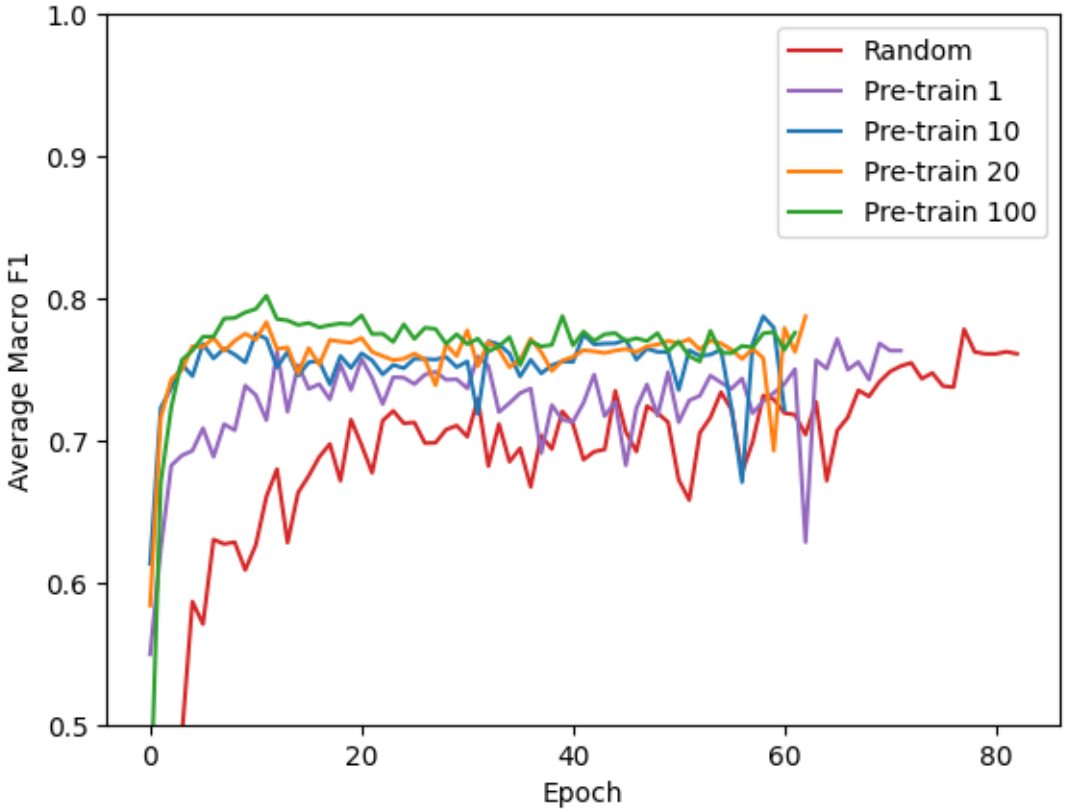
	Scenario	Mean	Std
0	Random	71.0	5.6

```
1    Pre-train 1  62.5  5.2
2   Pre-train 10  56.2  2.1
3   Pre-train 20  56.8  2.6
4  Pre-train 100  57.0  2.1
```

6.1.2 Figure 1: Average Validation Macro F1 Comparison

We plot the average validation macro F1 scores by epoch for each of the five scenarios using `plot_f1_by_epoch()` (source).

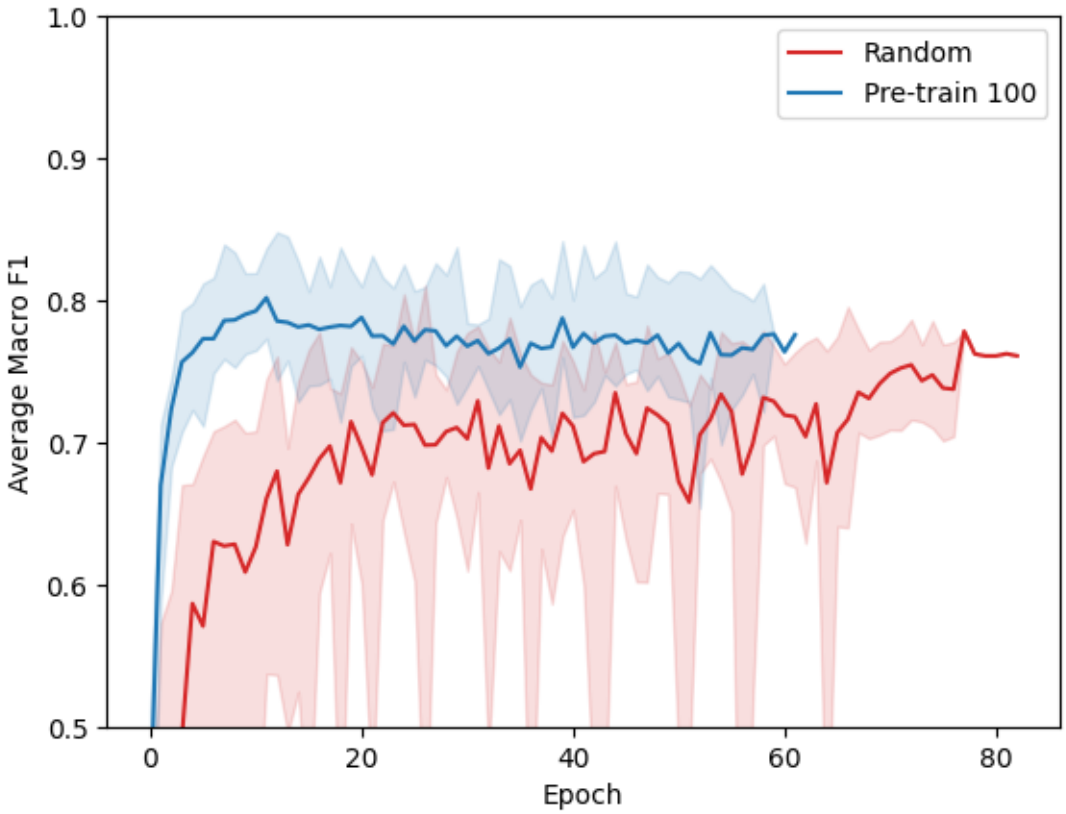
[40]:
```
analysis1d.plot_f1_by_epoch(history_all)
```



6.1.3 Figure 2: Average Validation Macro F1 Between Random and Pre-trained Scenarios

Similar to Figure 1, but with only random and pre-trained 100% scenarios and we also include shaded regions for each line. The shaded region represents the area between the maximum and minimum macro F1 score for that epoch among the 10 trials for a scenario. The `plot_f1_by_epoch_with_range` function is used to generate Figure 2 (source).

[41]:
```
analysis1d.plot_f1_by_epoch_with_range(history_all)
```



6.1.4 Table 2: Average Test Macro F1 Comparison

Recall that the fine-tuning entrypoint performs inference on the test set using the model with the highest validation macro F1 score.

We use `make_f1_table()` (source) to generate a table that is similar in format to Table 1 of the paper.

For each scenario, we report the average macro F1 score (and the standard deviation) on the test set. We also report the average F1 score for each class in the *PhysioNet 2017* dataset:

- Normal (F1n)
- AF (F1a)
- Other (F1o)
- Noisy (F1p)

[42]:
```
from transplant.utils import load_pkl

predictions_all = load_pkl(RESULT1D_DIR + "/predictions_all.pkl")
analysis1d.make_f1_table(predictions_all)
```

[42]:

	Type	F1	F1n	F1a	F1o	F1p
0	Random	.702 (± .030)	.881 (± .011)	.647 (± .048)	.686 (± .030)	.595 (± .063)
1	Pre-train 1	.754 (± .011)	.899 (± .010)	.728 (± .034)	.729 (± .013)	.661 (± .024)
2	Pre-train 10	.765 (± .012)	.904 (± .005)	.745 (± .017)	.735 (± .013)	.675 (± .030)
3	Pre-train 20	.765 (± .014)	.901 (± .004)	.760 (± .026)	.741 (± .012)	.658 (± .032)
4	Pre-train 100	.773 (± .012)	.904 (± .007)	.751 (± .038)	.750 (± .014)	.688 (± .026)

6.2 Analysis

6.2.1 Epoch Analysis

Table 1 shows the average and standard deviation of the number of epochs taken by the fine-tuning process to converge by means of early stopping. Recall that the average statistic is taken over 10 trials for each scenario.

Table 1 shows that: - The model pre-trained with 1% of the data took about 10% fewer epochs to reach convergence during fine-tuning compared to the randomly initialized model. - The models pre-trained with 10%/20%/100% of the data took about 20% fewer epochs to reach convergence during fine-tuning compared to the randomly initialized model. - The models pre-trained with 10%/20%/100% of the data had about the same average epochs to convergence, with the 10% model being marginally faster than the 20% and 100% model. - Since all four pre-trained models converge faster, they all triggered the early stopping sooner than did the randomly initialized model.

Comparing with Paper Results There is not a table in the paper that is directly comparable to Table 1. However, we can perform an indirect comparison by looking at Figure 3(a) of the paper (see the "Appendix" for convenience). Specifically, from Figure 3(a) we see that pre-training takes between 15% to 20% fewer epochs to converge compared to random initialization. This is quite similar to the 10~20% reduction in epoch count for all four pre-trained models in Table 1.

Comparing with Hypothesis We compare our result to **Hypothesis 2**. Note that Table 1 does not address Hypothesis 2 directly as the result does not concern performance on the target task. However, if we consider epoch count to be a substitute, we might expect the model pre-trained with more data to take fewer epochs to fine-tune compared to the model pre-trained with less data.

Table 1 does not support this modified hypothesis. It shows that 10%/20%/100% pre-trained models are very similar in fine-tuning epoch count, and in fact the 10% model needs slightly fewer epochs than the 20% and 100% models.

One explanation for this behavior may be that once the model is pre-trained with a "sufficient" amount of the data, any additional data in pre-training will not reduce the number of epochs to converge in fine-tuning. Since the pre-training and fine-tuning task are similar but not identical,

pre-training with too much data may cause the model to overfit the pre-training task and marginally increase the number of epochs to converge in fine-tuning task.

6.2.2 Validation Macro F1 Analysis

Figure 1 compares the average macro F1 scores of each scenario on the validation set per epoch. Inspecting Figure 1 shows that pre-trained models:

- achieve a high validation macro F1 within just several epochs, whereas the randomly initialized model takes longer to converge and reaches a lower validation macro F1 plateau.
- consistently show better validation performance than the randomly initialized model over the course of fine-tuning.
- produce more stable macro F1 scores as more data is used for pre-training.

Figure 2 is similar to Figure 1 but it only shows two lines corresponding to the random and pre-training on 100% data models. Figure 2 also shows shaded regions for each line that are demarcated by the maximum and minimum macro F1 score per epoch among the 10 trials for each scenario. Thus, inspecting Figure 2 will yield similar observations as mentioned above for Figure 1.

An additional observation we may make about Figure 2 is that it is clear from the shading that the higher stability of the pre-trained model is due to less variance in the macro F1 score over the 10 trials.

Comparing with Paper Results We compare both Figures 1 and 2 to Figure 3(a) in the paper (see the "Appendix" for convenience).

Figure 1 The main difference is that Figure 1 has five lines, one for each scenario whereas Figure 3(a) has just two, as the paper only considers two scenarios: random initialization and pre-trained (on all data).

In terms of the shape of the lines, we see that the lines for all five scenarios in Figure 1 have a very similar shape to the lines in Figure 3(a) in the paper. We also see a similar separation between the 'plateaus' of the pre-trained models and the randomly initialized model in Figure 1. Also the plateaus in Figure 1 appear to hover around similar macro F1 scores as in Figure 3(a).

The main discrepancy between Figure 1 and Figure 3(a) is the number of epochs. In Figure 1, all scenarios completed fine-tuning within about 80 epochs. However in Figure 3(a), that figure is around 140 epochs.

The reason for the discrepancy lies in the metric used for early stopping of fine-tuning. Recall that the fine-tuning process is set up to terminate early if the metric does not 'improve' for 50 epochs. In our work, we used the code as provided by the authors, which uses the validation loss as the metric. However, the paper authors use training accuracy as the metric in their paper.

In a separate trial experiment (not included in this report), we changed the metric to training accuracy and saw fine-tuning take more than 100 epochs to converge, which explained the cause for the discrepancy in epoch count between the two figures. However we did not re-run our experiments to use the correct metric for the following reasons:

- The increase in epochs directly correlates with training time (from 30 minutes to 60 minutes) and compute costs.

- We were constrained on time and compute resources, having spent much of both on running the full set of experiments using the validation loss metric.
- Even with the 'wrong' metric, the fine-tuning results in Figure 1 and Table 2 are very similar to Figure 3(a) and Table 1 in the paper, respectively.

These reasons suggest that the choice of metric used for early stopping does not appear to make a huge difference in terms of the performance of the fine-tuned model.

Figure 2 Much of what has been said for Figure 1 also applies to Figure 2 when comparing Figure 2 to Figure 3(a) in the paper. The difference is that Figure 2 is our attempt to exactly reproduce Figure 3(a). So we may directly compare the two figures.

The two figures are very similar to each other in that they both show:

- a visible gap between the lines of the pre-trained versus random model.
- pre-trained model fine-tuning converges significantly faster compared to the random model.

The differences between the two figures could be attributed to different initial conditions including:

- seed used to split the fine-tuning dataset into train and validation for a trial, for each scenario.
- seed used to split the raw fine-tuning dataset into train and test datasets

Comparing with Hypothesis We compare Figure 1 to both **Hypotheses 1 and 2**, and Figure 2 to **Hypothesis 1**.

Hypothesis 1: Both Figures 1 and 2 appear to support Hypothesis 1 in that there is a clear increase in the performance of pre-trained models compared to a model that is not pre-trained on the validation dataset.

Hypothesis 2: Figure 1 does not appear to strongly support Hypothesis 2 for the validation dataset. We see that the blue (10%) and orange (20%) lines are very similar to each other; the orange line is not a clear winner compared to the blue line. And even for the green line (100%), while it has higher performance before 40 epochs have passed, its performance becomes very similar to that of the 10% and 20% models, even if it is more stable.

This may suggest that increasing the amount of data used for pre-training yields diminishing returns.

6.2.3 Test F1 Analysis

Note that Table 2 should be compared to the None and Beat Classification, Frame 4096 rows of Table 1 in the paper (see "Appendix"), as that is the scope of our replication work.

Table 2 shows that all pre-trained models outperform random initialized model in both:

- Average macro F1 score (F1 column).
- Average per-class F1 score, all classes (`F1n`, `F1a`, `F1o`, and `F1p` columns).

We observe that pre-training on 1% of the data yields large increases in average F1 scores compared to that of the random model.

We also see that both 10% and 20% scenarios had the same average macro F1 scores. There was no consistent winner at the average per-class F1 scores. However, the 100% scenario had the best average F1 scores across the board.

Comparing with Paper Results Our results in Table 2 are consistent with those in Table 1 of the paper.

- The 100% pre-trained model shows 10% higher average macro F1 score than that of random initialization.
- This is higher than the 6.57% figure reported in the paper.
  - This increase is due to the lower test performance of our random model compared to the paper's. The variation may be due to the model sensitivity to the initial conditions such as the splitting of the train and test datasets.

Comparing with Hypothesis We compare the results to **Hypotheses 1 and 2**.

Hypothesis 1 Table 2 indeed validates Hypothesis 1 in that pre-training on any amount of the pre-training dataset does improve the performance of the fine-tuned model on the target dataset, when compared to a not pre-trained model.

Hypothesis 2 Table 2 validates Hypothesis 2 in that pre-training on larger subsets of the dataset generally produces better test performance, compared to pre-training on a smaller subset.

- The 100% model had the global best test performance compared to any of the other pre-trained models.
- Both 10% and 20% models outperformed the 1% model.
- However, the 20% model did not outperform the 10% model. The two models were the most similar to each other among the pre-trained models in test performance.

7 Ablation Study - Experiment Beyond Paper's Results

Given the extent of work for the ablation study, we dedicate a separate top-level section to it, instead of locating it within the "Results" section.

7.1 Background

The original paper is entirely based on 1-D CNNs and the raw ECG signal. To extend the paper's results, we aim to pre-process the raw signals using Fourier transforms to represent the data as a spectrogram – a frequency versus time representation of ECG signals.

Using this representation of the input, we will train a 2-D CNN model (i.e. 2-D ResNet-18v2) and compare the performance of pre-trained and randomly initialized models. Additionally, we will compare the 2-D model performance to the 1-D models originally used by the authors.

This extension is motivated by a study on ECG Arrhythmia classification that demonstrates the effectiveness of CNNs trained on spectrograms [10]. By converting ECG data to spectrogram features and then using spectrograms to pre-train a 2-D ResNet, we intend to illustrate the adaptability of the transfer learning framework in the original paper across diverse model architectures.

7.2 Spectrogram Pre-processing

For the spectrogram, the following parameters were chosen:

- Window size: 256 (~1 second)
- Stride: 32 (~0.13 seconds)

- Window type: Hanning -> this is like a pre-defined convolution that is used to smooth the FFTs for each spectrogram slice
- Normalization: Remove mean from each sample
- Scale: Linear (not db)

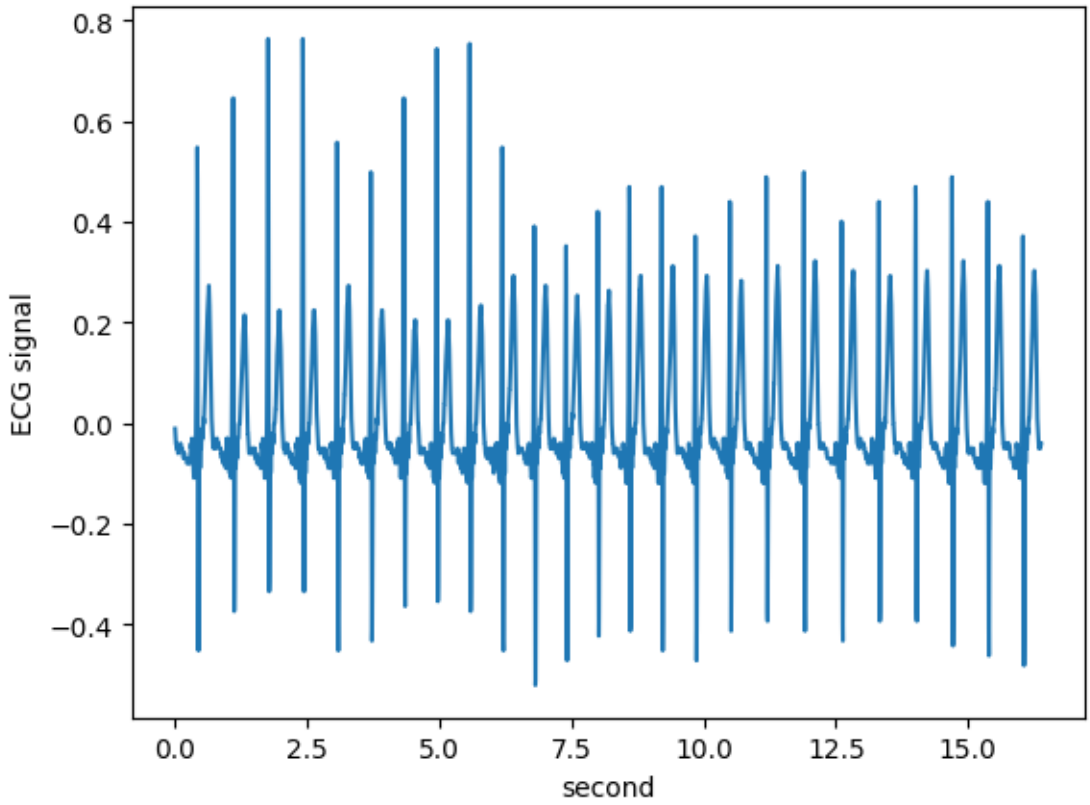
7.2.1 Spectrogram Pre-processing Code

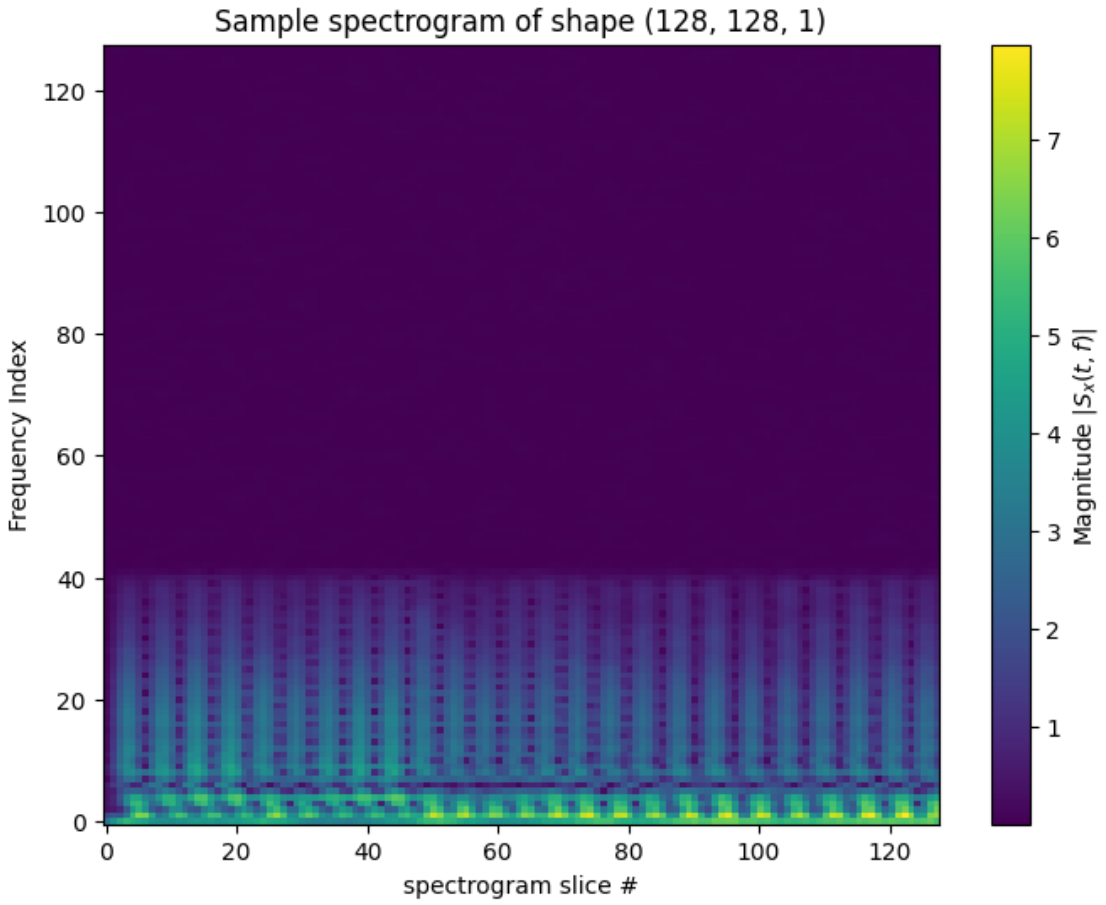
The spectrogram pre-processing code and the data loaders for the pre-training can be found here.

Below, we show the raw ECG signal next to the spectrogram of the same signal using our pre-processing function, and then use the spectrogram pre-processor on the same ECG signal presented in the earlier "Data" section.

[43]:
```
from pretraining import datasets
from transplant.datasets.icentia11k_spectrogram import spectrogram_preprocessor
# use raw data we grabbed earlier

# First let's plot the raw signal for comparison
print(f"Beat label: {datasets.icentia11k.ds_beat_names[data[1]]}")
_ = plt.plot(np.arange(len(data[0])) / 250, data[0])
_ = plt.xlabel("second")
_ = plt.ylabel("ECG signal")

# Now let's calculate and plot the spectrogram of this raw signal
x = spectrogram_preprocessor(
    np.squeeze(data[0]),
    window_size=256,
    stride=32,
    n_freqs=128,
    fs=250.0,
    ref=1,
)
fig, ax = plt.subplots(1, 1, figsize=(8, 6))
im1 = ax.imshow(x, cmap="viridis", aspect="auto")
fig.colorbar(im1, label="Magnitude $|S_x(t, f)|$")
ax.set_title(f"Sample spectrogram of shape {x.shape}")
ax.set_xlabel("spectrogram slice #")
ax.set_ylabel("Frequency Index")
ax.invert_yaxis()
plt.show()
```

```
Beat label: normal
```


0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
ECG signal
0.0
2.5
5.0
7.5
10.0
12.5
15.0
second



7.2.2 Ablation Model: 2-D ResNet-18v2

The model chosen for the ablation study using spectrograms is similar to the original model used, but is a 2-D ResNet-18v2. It is presented here:

Link to base code for the model that we used in this project, with the updated version in our repo here

This ResNet architecture is based on the original design by He et al. [9]

- Model architecture
  - 18 layers
  - Input layer consists of convolution layer with 64 filters, kernel size=7x7 and stride=2. The output of the convolution layer passes through batch norm, ReLu and max-pooling layer sequentially.
  - The middle 16 layers consists of 8 residual blocks. A residual block consists of the following two components and outputs the sum of the two components' outputs.
    1. Two convolution layers, each followed by batch norm and ReLu
    2. A shortcut that passes the input through a convolution layer followed by batch norm.

  - Output layer is a classifier consisting of a densely-connected layer followed by softmax or sigmoid function.
  - Configurations of the residual blocks
    * 1st and 2nd: 64 filters, kernel size=3x3, strides=2 and 1, respectively.
    * 3rd and 4th: 128 filters, kernel size=3x3, strides=2 and 1, respectively.
    * 5th and 6th: 256 filters, kernel size=3x3, strides=2 and 1, respectively.
    * 7th and 8th: 512 filters, kernel size=3x3, strides=2 and 1, respectively.
- Pre-training objectives
  - The same as the original author's 1-D pre-training objectives.
- Fine-tuning objectives
  - The same as the original author's 1-D pre-training objectives (F1 score)

We also include Google Drive links to pre-trained model weights for reader's inspection:

- Pre-trained model weights we trained with patients 0-2047 of the *Icentia11k* dataset (~20%) are here.
- Pre-trained model weights we trained with patients 0-9680 of the *Icentia11k* dataset (~88%) are here.

This model is built and displayed below for reference:

[44]:
```
from transplant.modules.resnet2d import ResNet18_2D
import transplant.datasets.icentia11k as icentia11k

num_classes = len(icentia11k.ds_beat_names)
model = ResNet18_2D(num_classes=num_classes)
model.build(input_shape=(None, 64, 64, 1))
model.compile(
    optimizer=tf.keras.optimizers.Adam(beta_1=0.9, beta_2=0.98, epsilon=1e-9),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics=[tf.keras.metrics.SparseCategoricalAccuracy(name="acc")],
)
model.summary(expand_nested=True)
```

```
WARNING:tensorflow:AutoGraph could not transform <bound method ResnetBlock.call
of <transplant.modules.resnet2d.ResnetBlock object at 0x7b9aa34db2e0>> and will
run it as-is.
Cause: mangled names are not yet supported
To silence this warning, decorate the function with
@tf.autograph.experimental.do_not_convert

WARNING: AutoGraph could not transform <bound method ResnetBlock.call of
<transplant.modules.resnet2d.ResnetBlock object at 0x7b9aa34db2e0>> and will run
it as-is.
Cause: mangled names are not yet supported
To silence this warning, decorate the function with
@tf.autograph.experimental.do_not_convert
Model: "res_net18_2d"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
```

```
=================================================================
 conv2d (Conv2D)             multiple                  3200

 batch_normalization_2 (Bat  multiple                  256
 chNormalization)

 max_pooling2d (MaxPooling2  multiple                  0
 D)

 resnet_block (ResnetBlock)  multiple                  74368
|-----------------------------------------------------------------|
| conv2d_1 (Conv2D)          multiple                  36928     |
|                                                                 |
| batch_normalization_3 (Ba  multiple                  256       |
| tchNormalization)                                               |
|                                                                 |
| conv2d_2 (Conv2D)          multiple                  36928     |
|                                                                 |
| batch_normalization_4 (Ba  multiple                  256       |
| tchNormalization)                                               |
|                                                                 |
| add (Add)                  multiple                  0         |
-------------------------------------------------------------------
 resnet_block_1 (ResnetBloc  multiple                  74368
 k)
|-----------------------------------------------------------------|
| conv2d_3 (Conv2D)          multiple                  36928     |
|                                                                 |
| batch_normalization_5 (Ba  multiple                  256       |
| tchNormalization)                                               |
|                                                                 |
| conv2d_4 (Conv2D)          multiple                  36928     |
|                                                                 |
| batch_normalization_6 (Ba  multiple                  256       |
| tchNormalization)                                               |
|                                                                 |
| add_1 (Add)                multiple                  0         |
-------------------------------------------------------------------
 resnet_block_2 (ResnetBloc  multiple                  231296
 k)
|-----------------------------------------------------------------|
| conv2d_5 (Conv2D)          multiple                  73856     |
|                                                                 |
| batch_normalization_7 (Ba  multiple                  512       |
| tchNormalization)                                               |
|                                                                 |
| conv2d_6 (Conv2D)          multiple                  147584    |
|                                                                 |
```

batch_normalization_8 (Ba tchNormalization)	multiple	512
add_2 (Add)	multiple	0
conv2d_7 (Conv2D)	multiple	8320
batch_normalization_9 (Ba tchNormalization)	multiple	512
resnet_block_3 (ResnetBloc k)	multiple	296192
conv2d_8 (Conv2D)	multiple	147584
batch_normalization_10 (B atchNormalization)	multiple	512
conv2d_9 (Conv2D)	multiple	147584
batch_normalization_11 (B atchNormalization)	multiple	512
add_3 (Add)	multiple	0
resnet_block_4 (ResnetBloc k)	multiple	921344
conv2d_10 (Conv2D)	multiple	295168
batch_normalization_12 (B atchNormalization)	multiple	1024
conv2d_11 (Conv2D)	multiple	590080
batch_normalization_13 (B atchNormalization)	multiple	1024
add_4 (Add)	multiple	0
conv2d_12 (Conv2D)	multiple	33024
batch_normalization_14 (B atchNormalization)	multiple	1024
resnet_block_5 (ResnetBloc k)	multiple	1182208

conv2d_13 (Conv2D)	multiple	590080
batch_normalization_15 (BatchNormalization)	multiple	1024
conv2d_14 (Conv2D)	multiple	590080
batch_normalization_16 (BatchNormalization)	multiple	1024
add_5 (Add)	multiple	0
resnet_block_6 (ResnetBlock)	multiple	3677696
conv2d_15 (Conv2D)	multiple	1180160
batch_normalization_17 (BatchNormalization)	multiple	2048
conv2d_16 (Conv2D)	multiple	2359808
batch_normalization_18 (BatchNormalization)	multiple	2048
add_6 (Add)	multiple	0
conv2d_17 (Conv2D)	multiple	131584
batch_normalization_19 (BatchNormalization)	multiple	2048
resnet_block_7 (ResnetBlock)	multiple	4723712
conv2d_18 (Conv2D)	multiple	2359808
batch_normalization_20 (BatchNormalization)	multiple	2048
conv2d_19 (Conv2D)	multiple	2359808
batch_normalization_21 (BatchNormalization)	multiple	2048
add_7 (Add)	multiple	0
global_average_pooling2d (	multiple	0

```
 GlobalAveragePooling2D)

 flatten (Flatten)           multiple                  0

 dense_1 (Dense)             multiple                  2565

=================================================================
Total params: 11187205 (42.68 MB)
Trainable params: 11177605 (42.64 MB)
Non-trainable params: 9600 (37.50 KB)
_________________________________________________________________
```

7.3 Ablation Training

7.3.1 Hyperparameters

To simplify the training and make comparisons easier, the same hyperparameters for pre-training and fine-tuning were used during the ablation study. These common parameters included the batch size and learning rates, sample rates, and frame size. The only different hyperparameters were those chosen for the pre-processing step. These are already listed in the pre-processing section.

7.3.2 Computational Requirements

The computational requirements of the ablation study were higher for two reasons:

- The addition of the spectrogram pre-processing.
- A significantly larger model (11.2 million versus 4.5 million parameters).

For this reason, the **NVIDIA L4 GPU** with 22.4 GB RAM was chosen for for both pre-training and fine-tuning.

Pre-training Due to the increased computational requirements, the pre-training for the ablation took nearly twice per patient as using the 1-D ResNet-18v2. Training with 88% percent of the patients in the *Icentia11k* dataset took 24 hours, which exceeded the Google Colab Pro process runtime limit and prevented us from using 100% of the data.

- Average runtime per 1,000 patients: 2 hours, 29 minutes
- Total number of trials: 1
- GPU hours used: 24hrs
  - Frame size 4096
  - 9068 patients x 4096 samples per data = **8 million samples** (about 88% of the *Icentia11k* dataset)
- Number of training epochs: 1

Fine-tuning For fine-tuning, all of the data was pre-processed just once to reduce repeated computations during fine-tune training. The notebook to generate this data can be found here.

This reduced the extra computational burden of the ablation framework.

- Average runtime for each epoch: 15 seconds
- Total number of trials: 50

  - 10 trials for each of the three scenarios: random, 1%, 10% and 20%, 88%
- GPU hours used: About 14.2 hours total.
  - Random: 2.8 hours
  - 1%: 2.7hrs
  - 10%: 3.1 hours
  - 20%: 2.9 hours
  - 88%: 2.7 hours
- Number of training epochs: Maximum 200 per trial, but in practice, saw early stopping by far fewer epochs
  - Training for a single trial would typically complete within 20 minutes.

7.3.3 Training Code - Command Line Interface

The spectrogram pre-processing and the 2-D ResNet-18v2 are integrated into the original authors' codebase in our fork of the repo. As such, the same commands that were presented earlier can be used, with small modifications.

Pre-training The only modification is the `--arch` (architecture) argument, which should be set to `--arch "resnet18_2d"`.

Fine-tuning For fine-tuning, the same `--arch` modification is needed. Otherwise, the only difference is pointing the trainer to the correct pre-processed spectrogram data files.

As an example, the pre-training command is executed below using the same sample data as was used for the corresponding command in the 1-D ResNet-18v2 section:

[45]:
```
PRETRAIN2D_JOB_DIR = "/tmp/pretrain2d/"
```

[46]:
```
%%time
!python -m pretraining.trainer \
--job-dir {PRETRAIN2D_JOB_DIR} \
--task "beat" \
--train {DEMO_DATA_DIR} \
--arch "resnet18_2d" \
--epochs 2 \
--patient-ids 0 \
--steps-per-epoch 8 \
--samples-per-patient 4096 \
--batch-size 256 \
--frame-size 1024 \
--unzipped True \
--seed 2024
```

```
2024-05-07 01:42:43.094990: E
external/local_xla/xla/stream_executor/cuda/cuda_dnn.cc:9261] Unable to register
cuDNN factory: Attempting to register factory for plugin cuDNN when one has
already been registered
2024-05-07 01:42:43.095045: E
external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:607] Unable to register
```

```
cuFFT factory: Attempting to register factory for plugin cuFFT when one has
already been registered
2024-05-07 01:42:43.096767: E
external/local_xla/xla/stream_executor/cuda/cuda_blas.cc:1515] Unable to
register cuBLAS factory: Attempting to register factory for plugin cuBLAS when
one has already been registered
2024-05-07 01:42:44.158102: W
tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF-TRT Warning: Could not
find TensorRT
Creating working directory in /tmp/pretrain2d
Setting random state 2024
Building train data generators
# Patient IDs:  1
WARNING:tensorflow:From
/tmp/repo/transplant/datasets/icentia11k_spectrogram.py:71: calling
DatasetV2.from_generator (from tensorflow.python.data.ops.dataset_ops) with
output_types is deprecated and will be removed in a future version.
Instructions for updating:
Use output_signature instead
WARNING:tensorflow:From
/tmp/repo/transplant/datasets/icentia11k_spectrogram.py:71: calling
DatasetV2.from_generator (from tensorflow.python.data.ops.dataset_ops) with
output_shapes is deprecated and will be removed in a future version.
Instructions for updating:
Use output_signature instead
2024-05-07 01:42:45.694196: W
tensorflow/core/common_runtime/gpu/gpu_bfc_allocator.cc:47] Overriding
orig_value setting because the TF_FORCE_GPU_ALLOW_GROWTH environment variable is
set. Original config value was 0.
Building model …
WARNING:tensorflow:AutoGraph could not transform <bound method ResnetBlock.call
of <transplant.modules.resnet2d.ResnetBlock object at 0x7cacabed1720>> and will
run it as-is.
Cause: mangled names are not yet supported
To silence this warning, decorate the function with
@tf.autograph.experimental.do_not_convert
# model parameters: 11,187,205
Epoch 1/2
WARNING: All log messages before absl::InitializeLog() is called are written to
STDERR
I0000 00:00:1715046182.263409   10498 device_compiler.h:186] Compiled cluster
using XLA!  This line is logged at most once for the lifetime of the process.

Epoch 1: saving model to /tmp/pretrain2d/epoch_01/model.weights
8/8 - 24s - loss: 0.6896 - acc: 0.7900 - 24s/epoch - 3s/step
Epoch 2/2

Epoch 2: saving model to /tmp/pretrain2d/epoch_02/model.weights
```

```
8/8 - 2s - loss: 0.1245 - acc: 0.9697 - 2s/epoch - 299ms/step
Exception ignored in: <function AtomicFunction.__del__ at 0x7cacd80ff880>
Traceback (most recent call last):
  File "/usr/local/lib/python3.10/dist-
packages/tensorflow/python/eager/polymorphic_function/atomic_function.py", line
291, in __del__
TypeError: 'NoneType' object is not subscriptable
CPU times: user 201 ms, sys: 34.2 ms, total: 235 ms
Wall time: 33.5 s
```

7.4 Ablation Results

The procedure for generating the 2-D results is exactly the same as that for the 1-D results, just only with a different network structure. Thus the code used to generate the 1-D figures and tables above can be reused here.

Below, we download the collated results for subsequent analysis. More details on the collation process can be found in this notebook.

[47]:
```
RESULT2D_DIR = "/tmp/results2d/"
```

[48]:
```
%%time
%%capture
!mkdir -p {RESULT2D_DIR}
!gdown 1GwhDdlvUVmJfQfY3slyfiR-ytcuDRQ8X -O {RESULT2D_DIR} --folder
```

```
CPU times: user 83.6 ms, sys: 21.3 ms, total: 105 ms
Wall time: 11 s
```

Similar to the above "Results" section, we implement the analysis code in a separate file and import the analysis functions into this notebook. The code can be found on Github here.

The code is in the `analysis2d` module, which we import below.

[49]:
```
from report import analysis2d
```

7.4.1 Table 3: Epoch Comparison (2-D)

Same setup as Table 1, but for the 2-D results.

[50]:
```
history_all_2d = pd.read_csv(RESULT2D_DIR + "results2d_all/history_all.csv")
analysis2d.make_epoch_table(history_all_2d)
```

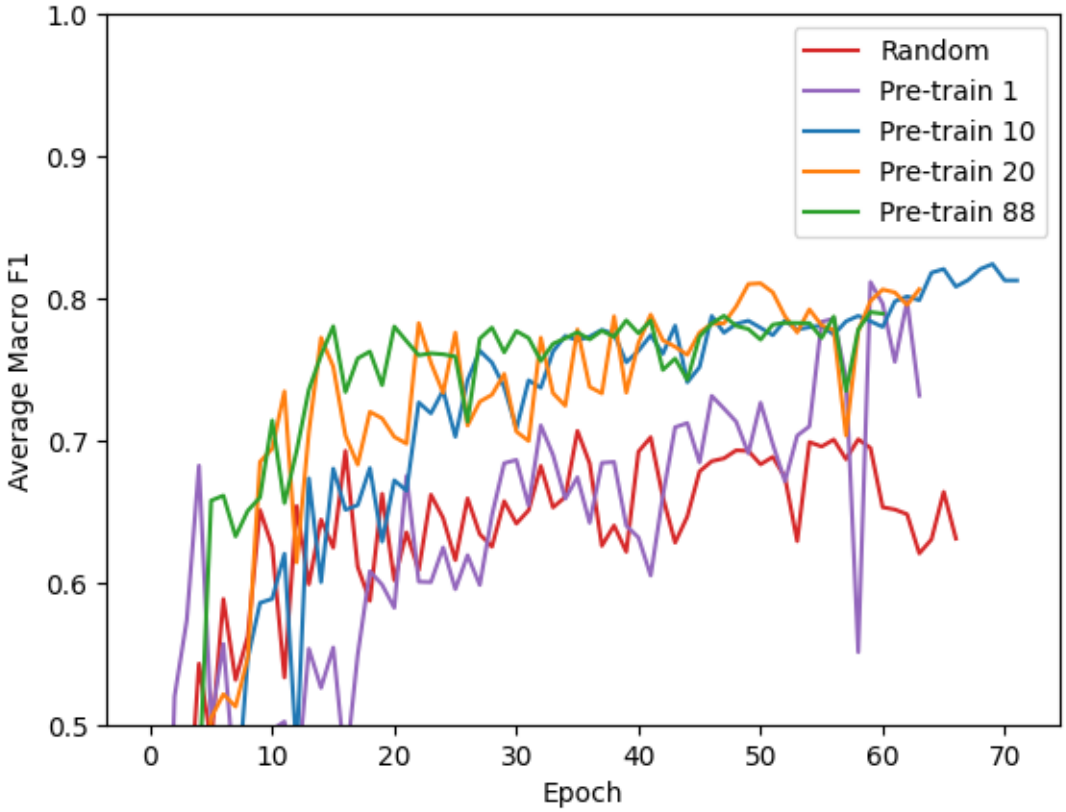
[50]:

	Scenario	Mean	Std
0	Random	60.9	3.2
1	Pre-train 1	54.5	3.2
2	Pre-train 10	60.4	4.4
3	Pre-train 20	58.7	3.1
4	Pre-train 88	56.2	1.7

7.4.2 Figure 3: Average Validation Macro F1 Comparison (2-D)

Similar setup as Figure 1, but for the 2-D results.

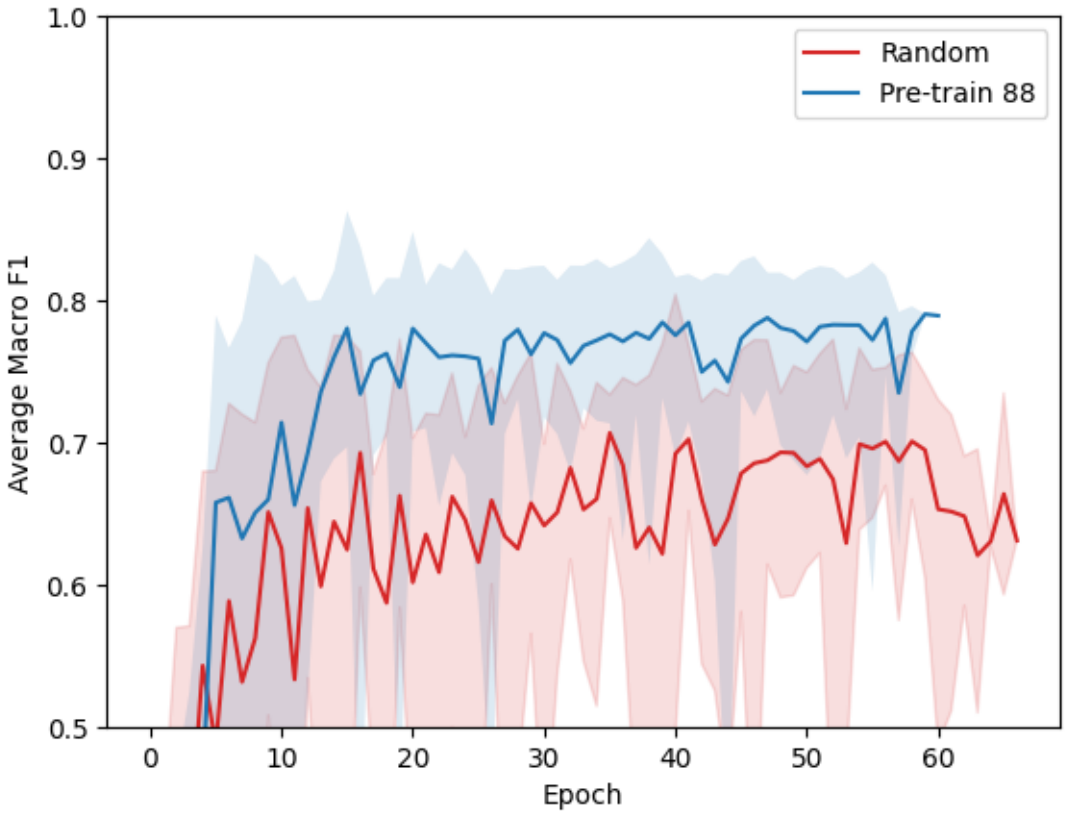
[51]:
```
analysis2d.plot_f1_by_epoch(history_all_2d)
```



7.4.3 Figure 4: Average Validation Macro F1 Between Random and 88% Pre-training (2-D)

Similar set up as Figure 2, but for the 2-D results.

[52]:
```
analysis2d.plot_f1_by_epoch_with_range(history_all_2d)
```



7.4.4 Table 4: Average Test Macro F1 Comparison (2-D)

Same setup as Table 2, but for the 2-D results.

[53]:
```
from transplant.utils import load_pkl

predictions_all_2d = load_pkl(
    RESULT2D_DIR + "/results2d_all/predictions_all.pkl"
)
analysis2d.make_f1_table(predictions_all_2d)
```

[53]:

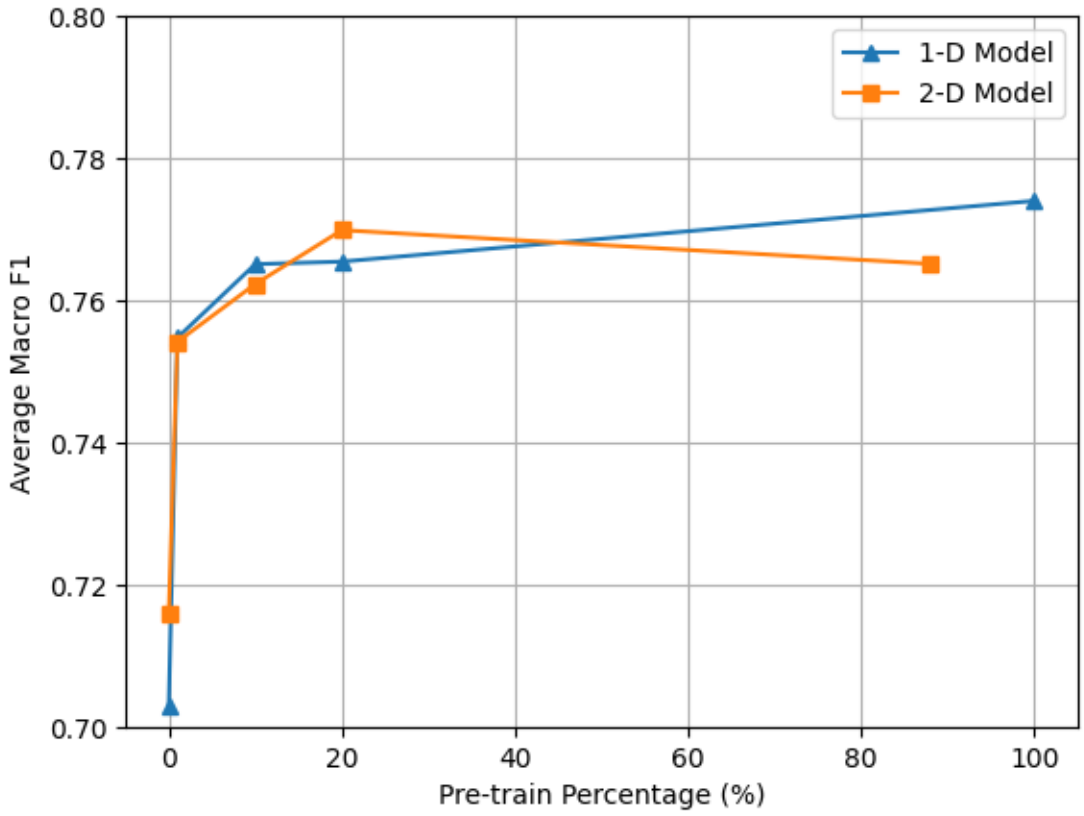
	Type	F1	F1n	F1a	F1o	F1p
0	Random	.715 (± .017)	.887 (± .005)	.681 (± .045)	.694 (± .028)	.600 (± .054)
1	Pre-train 1	.754 (± .013)	.898 (± .007)	.737 (± .022)	.735 (± .016)	
2	Pre-train 10	.762 (± .017)	.898 (± .005)	.751 (± .027)	.736 (± .015)	
3	Pre-train 20	.769 (± .014)	.903 (± .005)	.759 (± .015)	.751 (± .011)	
4	Pre-train 88	.765 (± .012)	.900 (± .005)	.774 (± .029)	.752 (± .010)	.644 (± .044)

```
2  .662 (± .034)
3  .665 (± .041)
4  .632 (± .024)
```

7.4.5 Figure 5: Comparison 1-D Versus 2-D Model Test Performance

We compare 1-D and 2-D model test macro F1 scores as a function of the percentage of data used to pre-train the models.

[54]:
```
analysis2d.plot_f1_by_pretrain_percentage(predictions_all, predictions_all_2d)
```



7.5 Ablation Discussion

Table 4 and Figure 4 both show that pre-training with the 2-D ResNet-18v2 spectrogram-based model shows significant improvement to the fine-tuned model's F1 score relative to the that of the randomly initialized model. This shows that the framework in the original paper is robust to different input and model types.

However, Table 3 shows that pre-training the model did not reduce the number of epochs needed in the fine-tuning step. This differs from the 1-D results in the original paper.

Figure 5 visually shows how the fine-tuned model's macro F1 score changed as we used different

percentages of the *Icentia11k* dataset in the pre-training step. For both the 1-D results and 2-D results, most of the improvement is achieved using only 1% of the *Icentia11k* dataset. For both models, macro F1 scores appear mostly flat after 10% or 20% of the pre-training data, with only minor changes to the F1 scores out to 100/88% of the pre-training data, respectively.

In general, performance is generally comparable for the 1-D and 2-D models, which shows that the pre-training principle is broadly applicable. However, the lack of performance improvement does not justify the increased computational cost and model size of the spectrogram / 2-D ResNet-18v2 approach.

7.6 Ablation Future Work

Due to time limitations, there were many aspects of the spectrogram pre-processing that could be explored further.

- There were many spectrogram hyperparameters that were not adequately explored. It is possible that significantly better performance could be achieved with a higher or lower resolution spectrogram. The hyperparameters not explored were:
  - Stride
  - Window size
  - Window type
- It is noticeable that many higher frequency bins in the pre-training spectrogram contain low energy. This could be consistent across the board and introduce useless or noisy information.
- Downsampling the fine-tuning dataset from 360 Hz to 250 Hz to match the pre-training data may introduce aliasing that might affect the performance of the spectrogram model. These results are not present in the pre-training model and might affect the ability to transfer learned features.

8 Discussion

8.1 Implications of the Experimental Results

In general, we were able to reproduce the results in the original paper. Pre-training the 1-D ResNet-18v2 showed significant improvement in model performance as shown by the average F1 scores in Table 2. The pre-training also increased the speed of fine-tuning by reducing the number of epochs to convergence as shown in Table 1.

We also replicated the increased performance from pre-training in our ablation study which used pre-processing to generate spectrogram inputs to a 2-D ResNet-18v2 model. This shows the extensibility of the approach taken in the original paper. This increased performance can be seen in Figure 3, Figure 4, and Table 4.

We were not able to replicate decreased training time by pre-training the 2-D ResNet-18v2, as evidenced by Table 3. This, however, is a less important result than increased test performance.

Figure 5 compares the performance results of the 1-D ResNet-18v2 (original paper), and 2-D ResNet (our ablation). This shows that both 1-D and 2-D models similarly benefit from pre-training. In both cases, the majority of the benefit from pre-training is realized from using just 1% of the pre-training data, with more modest improvements seen by using more data. However, if the goal is optimal performance, more training data does seem to produce better results.

In general, the comparable results of using the paper authors' 1-D ResNet-18v2 and our ablation study's 2-D ResNet-18v2 with spectrogram pre-processing does not justify the increased computational requirements and model size.

8.2 What Was Easy

8.2.1 Code Quality

We have been very fortunate that not only was the source code publicly available, but also it was well written and organized. This allowed us to more effectively spend our limited time on the core pieces of the code base, namely the pre-training and fine-tuning entrypoints, data pre-processing code, and model definitions.

The command line API was thoughtfully setup, which made the execution of the pre-training and fine-tuning trials relatively easy and gave us capacity to focus our attention to the results and learn more deeply about the deep learning mechanisms instead of getting bogged down in debugging work.

8.2.2 Data Accessibility and Quantization

Both the pre-training and fine-tuning datasets were truly publicly available online. Also, the pre-training data came in quantized form. This played a subtly important role in that it allowed us to easily obtain subsets of the pre-train data despite the fact that the uncompressed version is too large to store with the resources at our disposal.

8.3 What Was Difficult

Much of the time spent on this project may be considered standard work for deep learning projects, but each piece took significant amount of time, and had its own learning curve. In order to be able to reproduce the results and address Hypotheses 1 and 2, we had to work through:

- **Setting up development environment**: Google Colab + Google Drive + `git`.
- **Downloading large datasets**: The large pre-training dataset (> 200 GB) doesn't fit on our local hard drives, so this required interacting with *Academic Torrents* and writing a script to download the data overnight directly to Google Drive.
- **Python dependencies**: Setting up a working environment, eliminating deprecated functions
- **Understanding the codebase**: Despite being relatively well documented, understanding the pieces of a foreign codebase takes time. For example, the pre-training code's definition of an epoch (a set number of samples) is different from the typical definition.
- **Understanding compute requirements**: Figuring out which GPUs make sense from a cost and time perspective. Planning and distributing all the training amongst the group.
- **Discrepancy between paper and code**: It is true that we should not blindly trust that the code is correct ("trust but verify"). But much close work was needed to reconcile what we were seeing in the code versus what the authors were reporting in the paper. Also, due to the lack of explanation for the code discrepancy, we spent much thought on whether the deviations we observed in our experimental results could be attributable to the discrepancies in the code as opposed to some other confounding factors.

As an extension to the original paper's work, our ablation work of adding spectrogram pre-processing and integrating a new model (2-D ResNet-18v2) had its own set of unique challenges:

- **Understanding spectrograms**: The Fourier transform is a fundamental mathematical tool used in a wide range of applications. In order to use these expert features as an input for our model, we had to have a reasonable understanding of the meaning of these outputs and how they might affect the model. For example, only after it was too late did we realize that downsampling from 360 Hz to 250 Hz might introduce aliasing that would be be visible in the spectrogram. Controlling for this might have been beneficial.
- **Limited time to tweak hyperparameters**: Due to the time constraints, many educated guesses at hyperparameters like window size, stride, and normalization techniques were made.
- **Integrating 2-D ResNet-18v2**: This model was not previously supported by the codebase, so integrating it to function within the existing API required carefully separating the convolutional layer and classification layer to match the approach taken by the paper. Significant training time was duplicated by making small mistakes in how to transfer the model weights correctly.
- **Exceeding Google Colab Pro runtime limit**: The increased training time from pre-processing and increased model size meant our training time exceeded the 24 hours runtime limit Google Colab Pro imposes. This meant we were only able to train using 88% of the pre-training data for the ablation study.

8.4 Recommendations For Reproducibility

It would be a very good idea for such paper code to provide a **completely** specified environment configuration. The original `requirements.txt` provided did not specify all packages used, the exact versions used, nor performed complete pinning of dependencies.

For example, the `samplerate` dependency in the original requirements was deprecated when we started the project. We had to remove it and implement the code from scratch.

Second, it would be good for the authors to ensure that their public code contents match what they report in their paper. Or if it is not possible, clearly document the discrepancy for the benefit of reproduction work. Here are some notable examples:

- The paper claims to pre-process the fine-tuning dataset to produce 60 second samples. However we found that the code documentation recommended the user to generate ~65 second samples.
  - This discrepancy did not materially affect our experimental results. However, it should be understood that such a discrepancy may affect reproduction work of another project more severely.
- The paper claims to use training accuracy as the early stopping metric for fine-tuning training. Specifically, terminate fine-tuning early if the training accuracy does not improve over 50 epochs. But an inspection of the fine-tuning entrypoint code revealed that the code was using validation loss as the metric for early stopping.
  - This discrepancy did materially affect our work in that our fine-tuning trials took fewer epochs to converge than reported by the authors.

In sum, it is still a very good thing that the authors released their paper source code to the public because it enables reproduction work to be more efficient and enables independent verification of the correctness of the original work.

9 Public GitHub Repo

The repo with our code is available on Github. Specifically, the reports are located here.

Github full url: https://github.com/myles-i/DLH_TransferLearning/tree/master

9.1 READMEs

A top level README.md for our code and results can be found at the root of the directory here. In addition, we have README.md files in subdirectories including:

- The report folder (our work)
- The pre-training directory (from paper's repo)
- The fine-tuning directory (from paper's repo)

9.2 High-level Project Statistics (For Fun)

- Total pre-training samples used: **4.5 million (2dResNet), 4 million (2dResNet)**
- Total fine-tuning experiments run: **100**
- Longest model training run: **24 hours - cut off by Google Colab**
- Hours of GPU time used to generate our results: **73 hours**
- Estimated hours of GPU time used during development: **220 hours**
- GPUs used: **NVIDIA V100 and NVIDIA L4**
- Number of `git` commits by our team: **240**
- Timezones: **2 (PST and Argentina)**

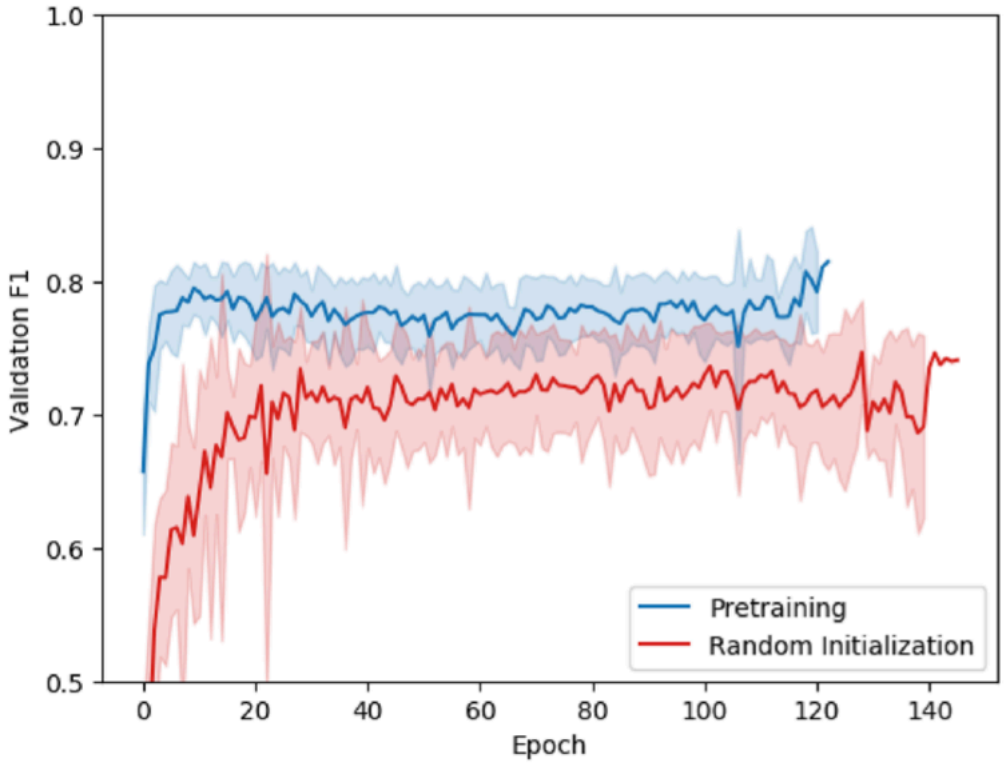
10 Appendix

We include figures and tables from the original paper here for the reader's convenience.

10.1 Table 1

Pretraining method	Frame	F_1	F_{1n}	F_{1a}	F_{1o}	F_{1p}
None (random weight initialization)		.731 (± .019)	.898 (± .005)	.711 (± .027)	.701 (± .017)	.613 (± .062)
Beat classification	512	.769 (± .011)	.911 (± .010)	.760 (± .018)	.758 (± .016)	.647 (± .022)
	2048	**.779 (± .014)**	**.915 (± .007)**	**.777 (± .014)**	**.763 (± .014)**	**.661 (± .040)**
	4096	.768 (± .010)	.908 (± .009)	.764 (± .021)	.754 (± .015)	.646 (± .025)

10.2 Figure 3(a)



(a) Beat Classification

11 References

[1] Kuba Weimann and Tim O. F. Conrad. Transfer learning for ecg classification. *Scientific Reports*, 11(1):5251, 2021.

[2] Mohammad Kachuee, Shayan Fazeli, and Majid Sarrafzadeh. Ecg heartbeat classification: A deep transferable representation. In *2018 IEEE International Conference on Healthcare Informatics (ICHI)*, pages 443–444, 2018.

[3] M.M. Al Rahhal, Yakoub Bazi, Haikel AlHichri, Naif Alajlan, Farid Melgani, and R.R. Yager. Deep learning approach for active classification of electrocardiogram signals. *Information Sciences*, 345:340–354, 2016.

[4] Deepta Rajan, David Beymer, and Girish Narayan. Generalization studies of neural network models for cardiac disease detection using limited channel ecg, 2019.

[5] Shawn Tan, Satya Ortiz-Gagné, Nicolas Beaudoin-Gagnon, Pierre Fecteau, Aaron Courville, Yoshua Bengio, and Joseph Paul Cohen. Icentia11k single lead continuous raw electrocardiogram dataset (version 1.0). https://doi.org/10.13026/kk0v-r952, April 2022.

[6] Shawn Tan, Guillaume Androz, Ahmad Chamseddine, Pierre Fecteau, Aaron Courville, Yoshua Bengio, and Joseph Paul Cohen. Icentia11k: An unsupervised representation learning dataset for arrhythmia subtype discovery, 2019.

[7] A. Goldberger, L. Amaral, L. Glass, J. Hausdorff, P. C. Ivanov, R. Mark, J. E. Mietus, G. B. Moody, C. K. Peng, and H. E. Stanley. Physiobank, physiotoolkit, and physionet: Components of a new research resource for complex physiologic signals. https://physionet.org/, 2000.

[8] Gari D Clifford, Chengyu Liu, Benjamin Moody, Li-Wei H Lehman, Ikaro Silva, Qiao Li, A E Johnson, and Roger G Mark. Af classification from a short single lead ecg recording: the physionet/computing in cardiology challenge 2017. *Comput Cardiol (2010)*, 44, Sep 2017.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks, 2016.

[10] J. Huang, B. Chen, B. Yao, and W. He. Ecg arrhythmia classification using stft-based spectrogram and convolutional neural network. *IEEE Access*, 7:92871–92880, 2019.