Code

- `src/`
  - `config.py` : high-level instructions for scripts (controls the number of epochs plus type of augmentation, base (standard, unity) and active norm)
  - `datasets.py` : script for loading data for denoising and classification tasks
  - `losses.py` : contains loss functions for denoising and classification tasks
  - `normalizations.py` : all augmentations for relaxing the structure of the input
  - `kfold/`
    - `kfold_cnn.py` : trains the CNN classifier over all 50 folds and saves the best model for each split
    - `kfold_combined.py` : trains the transformer-CNN hybrid classifier over all 50 folds and saves the best model for each split
    - `kfold_grad_cam.py` : calculates attribution masks for each instance as aggregates over all 50 models
  - `plotting/`
    - `plot_x_x_masked.ipynb` : plots a random example of input and masked input as seen in the paper
    - `plot_tree_maps.ipynb` : plots structure contributions in terms of TSS (relative square size)
    - `plot_grad_cam.ipynb` : plots a single example of aggregated Guided Grad-CAM over all 50 models as seen in the paper
    - `plot_feature_ranking.ipynb` : box plot for feature ranking via saliency maps as shown in the paper
  - `running_metrics.py` : collects metrics from all 50 folds for each type of data augmentation and saves to a CSV file
  - `running_classifier.py` : runs joint model on single fold
  - `running_cnn.py` : runs CNN on single fold
  - `running_grad_cam.py` : runs guided grad-cam using a single model
  - `running_inputter.py` : runs denoising task for a single fold

Code map for the analysis of multivariate-time series in a data sparse regime

Code

- `src/`
  - `config.py` : high-level instructions for scripts (controls the number of epochs plus type of augmentation, base (standard, unity) and active norm)
  - `datasets.py` : script for loading data for denoising and classification tasks
  - `losses.py` : contains loss functions for denoising and classification tasks
  - `normalizations.py` : all augmentations for relaxing the structure of the input
  - `kfold/`
    - `kfold_cnn.py` : trains the CNN classifier over all 50 folds and saves the best model for each split
    - `kfold_combined.py` : trains the transformer-CNN hybrid classifier over all 50 folds and saves the best model for each split
    - `kfold_grad_cam.py` : calculates attribution masks for each instance as aggregates over all 50 models
  - `plotting/`
    - `plot_x_x_masked.ipynb` : plots a random example of input and masked input as seen in the paper
    - `plot_tree_maps.ipynb` : plots structure contributions in terms of TSS (relative square size)
    - `plot_grad_cam.ipynb` : plots a single example of aggregated Guided Grad-CAM over all 50 models as seen in the paper
    - `plot_feature_ranking.ipynb` : box plot for feature ranking via saliency maps as shown in the paper
  - `running_metrics.py` : collects metrics from all 50 folds for each type of data augmentation and saves to a CSV file
  - `running_classifier.py` : runs joint model on single fold
  - `running_cnn.py` : runs CNN on single fold
  - `running_grad_cam.py` : runs guided grad-cam using a single model
  - `running_inputter.py` : runs denoising task for a single fold

Data format

All instances must be stored in CSV files. If your dataset consists of 1000 examples, then you need to populate a folder (in my case called long) with 1000 CSV files. Each CSV has column names associated with each feature in the multivariate time series, while each row represents a different time (time ordered). The CSV should look like follows:

	94	131	171	193	211	R_VALUE	XR_MAX	target
5	6.948843265893087	23.92038957436665	640.572746249489	804.4976653391947	337.67414279175216	4.719635179978764	6.7971e-07	1
5	6.864979570667067	23.261629814437313	633.1677645227642	799.3059947343635	335.24962964090787	4.703501962562543	7.056669230769231e-07	1
5	7.048615622996006	23.485013125945837	632.7253751156659	799.1419155793244	334.1861423976305	4.687368745146322	7.316238461538463e-07	1
5	6.817406259515682	23.12893834213854	634.1097486817505	799.1015170955628	334.6809585864472	4.7032741441210915	8.077638461538462e-07	1
5	6.72662247746867	22.908430594350097	631.6242477133627	792.701502862173	331.6906935257317	4.7209594662286944	8.866917948717949e-07	1
5	6.666772119475279	22.77720170058708	631.7489945767161	788.460283210315	329.97461265524987	4.712144795280475	1.747964102564102e-06	1
5	6.740538092006397	22.99587515696767	636.2131151166553	787.7879742989974	329.59306028955757	4.7002124780903936	2.701276923076922e-06	1

Note that the left-most column indicates the example index, while the right-most column is the target

Dataset

```python
class MVTSDataset(Dataset):
    """Dynamically computes missingness (noise) mask for each sample"""

    def __init__(self, indicies, norm_type='unity', mean_mask_length=3, masking_ratio=0.15):
        """
        args:
            indicies: list of indicies of samples to include in dataset
            norm_type: 'unity' or 'standard'
            mean_mask_length: mean length of noise mask
            masking_ratio: ratio of values to mask
        Returns:
            x: (batch, seq_length, feat_dim)
            mask: (batch, seq_length, feat_dim) boolean array: 0s mask and predict, 1s: unaffected input
            label: (batch, 1) 1 or 0
        """
```

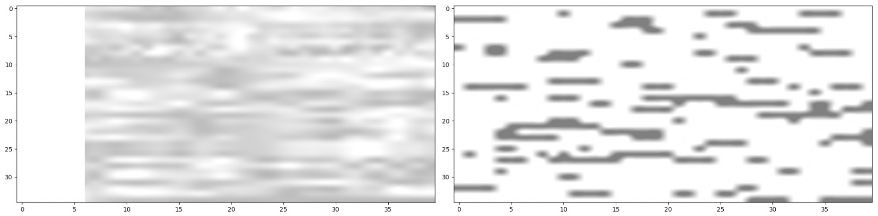
Dataloader

```python
# Create loader
val_dataloader = DataLoader(MVTSDataset(val_indices, norm_type='unity', mean_mask_length=3,
                                        masking_ratio=0.15), batch_size=len(val_indices), shuffle=False, drop_last=False)
```

```python
x, mask, y = next(iter(val_dataloader))

print(f"class: {y[0].item()}")
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 5))  # 1 row, 2 columns
ax1.imshow(x[0].T, aspect='auto', cmap='gray', interpolation='spline16', vmin=0, vmax=np.nanmax(x), alpha=0.5)
ax2.imshow(mask[0].T, aspect='auto', cmap='gray', interpolation='spline16', vmin=0, vmax=np.nanmax(mask), alpha=0.5)
plt.tight_layout()
plt.show()
```

✓ 0.5s Python

```
class: 0
```



Code

- `src/`
  - `config.py` : high-level instructions for scripts (controls the number of epochs plus type of augmentation, base (standard, unity) and active norm)
  - `datasets.py` : script for loading data for denoising and classification tasks
  - `losses.py` : contains loss functions for denoising and classification tasks
  - `normalizations.py` : all augmentations for relaxing the structure of the input
  - `kfold/`
    - `kfold_cnn.py` : trains the CNN classifier over all 50 folds and saves the best model for each split
    - `kfold_combined.py` : trains the transformer-CNN hybrid classifier over all 50 folds and saves the best model for each split
    - `kfold_grad_cam.py` : calculates attribution masks for each instance as aggregates over all 50 models
  - `plotting/`
    - `plot_x_x_masked.ipynb` : plots a random example of input and masked input as seen in the paper
    - `plot_tree_maps.ipynb` : plots structure contributions in terms of TSS (relative square size)
    - `plot_grad_cam.ipynb` : plots a single example of aggregated Guided Grad-CAM over all 50 models as seen in the paper
    - `plot_feature_ranking.ipynb` : box plot for feature ranking via saliency maps as shown in the paper
  - `running_metrics.py` : collects metrics from all 50 folds for each type of data augmentation and saves to a CSV file
  - `running_classifier.py` : runs joint model on single fold
  - `running_cnn.py` : runs CNN on single fold
  - `running_grad_cam.py` : runs guided grad-cam using a single model
  - `running_inputter.py` : runs denoising task for a single fold

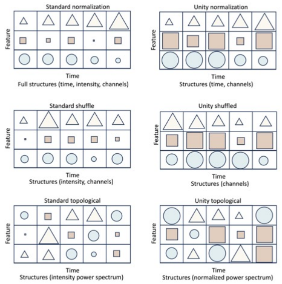
BASE_NORM

This is handled at the level of the dataset an can be either: “standard” or “unity”
Standard applies a robust standard scalar from sklearn while unity integrates out the intensity

ACTIVE_NORM

This is handled dynamically during training loops and acts on top, i.e., in addition to the base norm

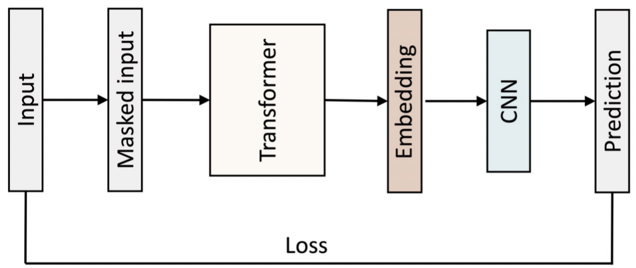
```python
#? Function for first order topological shuffle (shuffle across time acess for each feature)
def shuffle_tensor_along_time(tensor):
    # Runs dynamicaly on during traning loop on batches. Input shape (batch_size, time_steps, d_features)
    batch_size, time_steps, d_features = tensor.size()
    indices = torch.stack([torch.randperm(time_steps) for _ in range(batch_size * d_features)]).view(batch_size, d_features, time_steps)
    shuffled_tensor = tensor.permute(0, 2, 1).gather(2, indices).permute(0, 2, 1)
    return shuffled_tensor

#? Function for second order topological shuffle (shuffle across time and feature access)
def topological_shuffle(tensor):
    # Runs dynamicaly on during traning loop on batches. Input shape (batch_size, time_steps, d_features)
    batch_size, time_steps, d_features = tensor.size()
    shuffled_tensor = tensor.clone()
    for i in range(batch_size):
        indices = torch.randperm(time_steps * d_features)
        shuffled_tensor[i] = tensor[i].view(-1)[indices].view(time_steps, d_features)
    return shuffled_tensor

#? Integrates out the intensity by normalizing each feature from a single mvts by its maximum value
def unity_based_normalization(data):
    # Applied implicitly in the dataloader but can be dynamicly run on single instances if batch size is 1: input shape (time_steps, d_features)
    max_vals = np.nanmax(data, axis=1)
    min_vals = np.nanmin(data, axis=1)
    ranges = max_vals - min_vals
    eps = np.finfo(data.dtype).eps
    ranges[ranges < eps] = eps
    data = (data - min_vals[:, np.newaxis]) / ranges[:, np.newaxis]
    data = data + np.nanmax(data)
    data *= (1 / np.nanmax(data, axis=1)[:, np.newaxis])
    return data

#? Identity normalization
def identity_normalization(tensor):
    return tensor

#? Robust Standardization
# Applied implicitly in the dataloader and cannot be run dynamicaly on single batchs
```



Code

- `src/`
  - `config.py` : high-level instructions for scripts (controls the number of epochs plus type of augmentation, base (standard, unity) and active norm)
  - `datasets.py` : script for loading data for denoising and classification tasks
  - `losses.py` : contains loss functions for denoising and classification tasks
  - `normalizations.py` : all augmentations for relaxing the structure of the input
  - `kfold/`
    - `kfold_cnn.py` : trains the CNN classifier over all 50 folds and saves the best model for each split
    - `kfold_combined.py` : trains the transformer-CNN hybrid classifier over all 50 folds and saves the best model for each split
    - `kfold_grad_cam.py` : calculates attribution masks for each instance as aggregates over all 50 models
  - `plotting/`
    - `plot_x_x_masked.ipynb` : plots a random example of input and masked input as seen in the paper
    - `plot_tree_maps.ipynb` : plots structure contributions in terms of TSS (relative square size)
    - `plot_grad_cam.ipynb` : plots a single example of aggregated Guided Grad-CAM over all 50 models as seen in the paper
    - `plot_feature_ranking.ipynb` : box plot for feature ranking via saliency maps as shown in the paper
  - `running_metrics.py` : collects metrics from all 50 folds for each type of data augmentation and saves to a CSV file
  - `running_classifier.py` : runs joint model on single fold
  - `running_cnn.py` : runs CNN on single fold
  - `running_grad_cam.py` : runs guided grad-cam using a single model
  - `running_inputter.py` : runs denoising task for a single fold

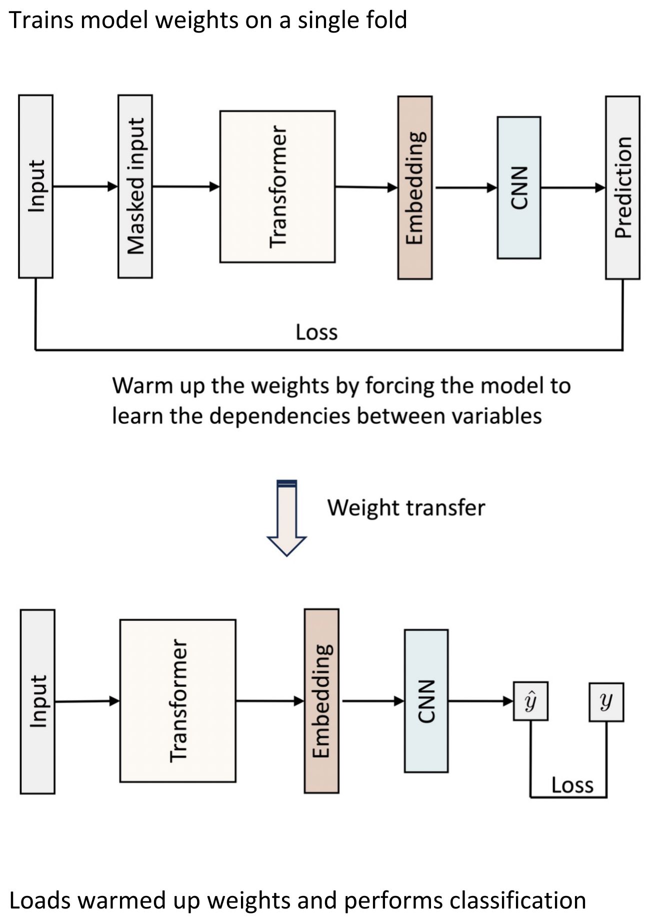
Trains model weights on a single fold



Warm up the weights by forcing the model to learn the dependencies between variables

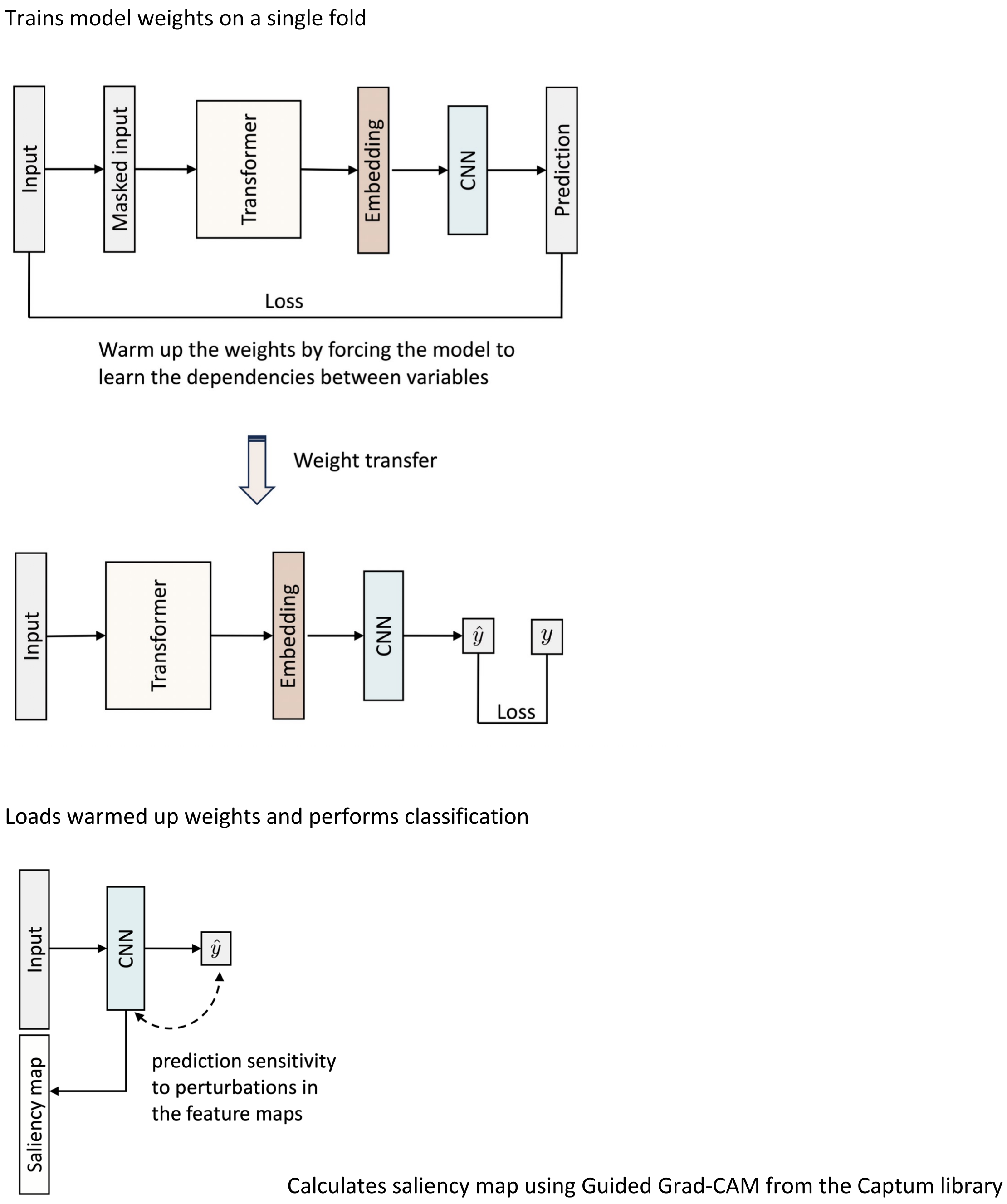
Code

- `src/`
  - `config.py` : high-level instructions for scripts (controls the number of epochs plus type of augmentation, base (standard, unity) and active norm)
  - `datasets.py` : script for loading data for denoising and classification tasks
  - `losses.py` : contains loss functions for denoising and classification tasks
  - `normalizations.py` : all augmentations for relaxing the structure of the input
  - `kfold/`
    - `kfold_cnn.py` : trains the CNN classifier over all 50 folds and saves the best model for each split
    - `kfold_combined.py` : trains the transformer-CNN hybrid classifier over all 50 folds and saves the best model for each split
    - `kfold_grad_cam.py` : calculates attribution masks for each instance as aggregates over all 50 models
  - `plotting/`
    - `plot_x_x_masked.ipynb` : plots a random example of input and masked input as seen in the paper
    - `plot_tree_maps.ipynb` : plots structure contributions in terms of TSS (relative square size)
    - `plot_grad_cam.ipynb` : plots a single example of aggregated Guided Grad-CAM over all 50 models as seen in the paper
    - `plot_feature_ranking.ipynb` : box plot for feature ranking via saliency maps as shown in the paper
  - `running_metrics.py` : collects metrics from all 50 folds for each type of data augmentation and saves to a CSV file
  - `running_classifier.py` : runs joint model on single fold
  - `running_cnn.py` : runs CNN on single fold
  - `running_grad_cam.py` : runs guided grad-cam using a single model
  - `running_inputter.py` : runs denoising task for a single fold



Code

- `src/`
  - `config.py` : high-level instructions for scripts (controls the number of epochs plus type of augmentation, base (standard, unity) and active norm)
  - `datasets.py` : script for loading data for denoising and classification tasks
  - `losses.py` : contains loss functions for denoising and classification tasks
  - `normalizations.py` : all augmentations for relaxing the structure of the input
  - `kfold/`
    - `kfold_cnn.py` : trains the CNN classifier over all 50 folds and saves the best model for each split
    - `kfold_combined.py` : trains the transformer-CNN hybrid classifier over all 50 folds and saves the best model for each split
    - `kfold_grad_cam.py` : calculates attribution masks for each instance as aggregates over all 50 models
  - `plotting/`
    - `plot_x_x_masked.ipynb` : plots a random example of input and masked input as seen in the paper
    - `plot_tree_maps.ipynb` : plots structure contributions in terms of TSS (relative square size)
    - `plot_grad_cam.ipynb` : plots a single example of aggregated Guided Grad-CAM over all 50 models as seen in the paper
    - `plot_feature_ranking.ipynb` : box plot for feature ranking via saliency maps as shown in the paper
  - `running_metrics.py` : collects metrics from all 50 folds for each type of data augmentation and saves to a CSV file
  - `running_classifier.py` : runs joint model on single fold
  - `running_cnn.py` : runs CNN on single fold
  - `running_grad_cam.py` : runs guided grad-cam using a single model
  - `running_inputter.py` : runs denoising task for a single fold



Code

Same as single instance running routines but over 50 random folds and for a set base and active augmentation

- `src/`
  - `config.py` : high-level instructions for scripts (controls the number of epochs plus type of augmentation, base (standard, unity) and active norm)
  - `datasets.py` : script for loading data for denoising and classification tasks
  - `losses.py` : contains loss functions for denoising and classification tasks
  - `normalizations.py` : all augmentations for relaxing the structure of the input
  - `kfold/`
    - `kfold_cnn.py` : trains the CNN classifier over all 50 folds and saves the best model for each split
    - `kfold_combined.py` : trains the transformer-CNN hybrid classifier over all 50 folds and saves the best model for each split
    - `kfold_grad_cam.py` : calculates attribution masks for each instance as aggregates over all 50 models
  - `plotting/`
    - `plot_x_x_masked.ipynb` : plots a random example of input and masked input as seen in the paper
    - `plot_tree_maps.ipynb` : plots structure contributions in terms of TSS (relative square size)
    - `plot_grad_cam.ipynb` : plots a single example of aggregated Guided Grad-CAM over all 50 models as seen in the paper
    - `plot_feature_ranking.ipynb` : box plot for feature ranking via saliency maps as shown in the paper
  - `running_metrics.py` : collects metrics from all 50 folds for each type of data augmentation and saves to a CSV file
  - `running_classifier.py` : runs joint model on single fold
  - `running_cnn.py` : runs CNN on single fold
  - `running_grad_cam.py` : runs guided grad-cam using a single model
  - `running_inputter.py` : runs denoising task for a single fold

Code

- `src/`
    - `config.py` : high-level instructions for scripts (controls the number of epochs plus type of augmentation, base (standard, unity) and active norm)
    - `datasets.py` : script for loading data for denoising and classification tasks
    - `losses.py` : contains loss functions for denoising and classification tasks
    - `normalizations.py` : all augmentations for relaxing the structure of the input
    - `kfold/`
        - `kfold_cnn.py` : trains the CNN classifier over all 50 folds and saves the best model for each split
        - `kfold_combined.py` : trains the transformer-CNN hybrid classifier over all 50 folds and saves the best model for each split
        - `kfold_grad_cam.py` : calculates attribution masks for each instance as aggregates over all 50 models
    - `plotting/`
        - `plot_x_x_masked.ipynb` : plots a random example of input and masked input as seen in the paper
        - `plot_tree_maps.ipynb` : plots structure contributions in terms of TSS (relative square size)
        - `plot_grad_cam.ipynb` : plots a single example of aggregated Guided Grad-CAM over all 50 models as seen in the paper
        - `plot_feature_ranking.ipynb` : box plot for feature ranking via saliency maps as shown in the paper
    - `running_metrics.py` : collects metrics from all 50 folds for each type of data augmentation and saves to a CSV file
    - `running_classifier.py` : runs joint model on single fold
    - `running_cnn.py` : runs CNN on single fold
    - `running_grad_cam.py` : runs guided grad-cam using a single model
    - `running_inputter.py` : runs denoising task for a single fold

Derives statistics for all folds and all augmentations and compiles them into a single SCV file

augmentation	tss	auc	hss	bss	accuracy
cnn_unity_topological	0.0	0.48471320346320346	1.0	-0.01353009943156458	0.45901639344262296
cnn_unity_topological	0.0	0.4491228070175438	1.0	-0.005673041956777425	0.4672131147540984
cnn_unity_topological	0.0	0.48185483870967744	1.0	-0.0006382308161057004	0.4918032786885246

Code

- `src/`
  - `config.py` : high-level instructions for scripts (controls the number of epochs plus type of augmentation, base (standard, unity) and active norm)
  - `datasets.py` : script for loading data for denoising and classification tasks
  - `losses.py` : contains loss functions for denoising and classification tasks
  - `normalizations.py` : all augmentations for relaxing the structure of the input
  - `kfold/`
    - `kfold_cnn.py` : trains the CNN classifier over all 50 folds and saves the best model for each split
    - `kfold_combined.py` : trains the transformer-CNN hybrid classifier over all 50 folds and saves the best model for each split
    - `kfold_grad_cam.py` : calculates attribution masks for each instance as aggregates over all 50 models
  - `plotting/`
    - `plot_x_x_masked.ipynb` : plots a random example of input and masked input as seen in the paper
    - `plot_tree_maps.ipynb` : plots structure contributions in terms of TSS (relative square size)
    - `plot_grad_cam.ipynb` : plots a single example of aggregated Guided Grad-CAM over all 50 models as seen in the paper
    - `plot_feature_ranking.ipynb` : box plot for feature ranking via saliency maps as shown in the paper
  - `running_metrics.py` : collects metrics from all 50 folds for each type of data augmentation and saves to a CSV file
  - `running_classifier.py` : runs joint model on single fold
  - `running_cnn.py` : runs CNN on single fold
  - `running_grad_cam.py` : runs guided grad-cam using a single model
  - `running_inputter.py` : runs denoising task for a single fold

```
import torch
import numpy as np
from normalizations import shuffle_tensor_along_time, topological_shuffle, identity_normalization

N_EPOCHS = 50 # Number of epochs to train models for the binary classification task
N_EPOCHS_AR = 200 # Number of epochs to train models for the autoregressive denoising task
BASE_NORM = 'standard' # Normalization type for the binary classification task| Options: 'unity', 'standard'
ACTIVE_NORM = topological_shuffle # Normalization type for the autoregressive denoising task | Options:
shuffle_tensor_along_time, topological_shuffle, identity_normalization
RUN_NAME = 'combined_std_topological' # Name of the run
```

Controles how many epochs for each mode as well as the types of normalizations

Code

- `src/`
  - `config.py` : high-level instructions for scripts (controls the number of epochs plus type of augmentation, base (standard, unity) and active norm)
  - `datasets.py` : script for loading data for denoising and classification tasks
  - `losses.py` : contains loss functions for denoising and classification tasks
  - `normalizations.py` : all augmentations for relaxing the structure of the input
  - `kfold/`
    - `kfold_cnn.py` : trains the CNN classifier over all 50 folds and saves the best model for each split
    - `kfold_combined.py` : trains the transformer-CNN hybrid classifier over all 50 folds and saves the best model for each split
    - `kfold_grad_cam.py` : calculates attribution masks for each instance as aggregates over all 50 models
  - `plotting/`
    - `plot_x_x_masked.ipynb` : plots a random example of input and masked input as seen in the paper
    - `plot_tree_maps.ipynb` : plots structure contributions in terms of TSS (relative square size)
    - `plot_grad_cam.ipynb` : plots a single example of aggregated Guided Grad-CAM over all 50 models as seen in the paper
    - `plot_feature_ranking.ipynb` : box plot for feature ranking via saliency maps as shown in the paper
  - `running_metrics.py` : collects metrics from all 50 folds for each type of data augmentation and saves to a CSV file
  - `running_classifier.py` : runs joint model on single fold
  - `running_cnn.py` : runs CNN on single fold
  - `running_grad_cam.py` : runs guided grad-cam using a single model
  - `running_inputter.py` : runs denoising task for a single fold