Advanced Data Mining: Homework set 3

2024/2025

Problem 1 (4 Points)

In this assignment, you will build and evaluate LSTM and GRU models for a one-step-ahead forecasting task on a financial time series. Use Python with TensorFlow (Keras) or PyTorch to implement the models. The tasks are:

A. Data Preparation: Load the weather time series dataset. For instance the Jena climate dataset, you can obtain it by running `curl https://www.bgc-jena.mpg.de/wetter/mpi_saale_2021b.zip -o mpi_saale_2021b.zip`. This is likely a multivariate time series (e.g., temperature, pressure, humidity, etc.). For the forecasting task, choose one target variable to predict (e.g., daily temperature). Define the forecasting horizon (e.g., predict the next 7 days of the target given the past 30 days of data). Split the data into training/validation/test segments by date.

B. Implement LSTM Model: Using your chosen framework, define an LSTM-based neural network for forecasting the next time step. (Tip: In Keras you can use tf.keras.layers.LSTM, and in PyTorch use torch.nn.LSTM within a custom nn.Module). Include necessary layers (e.g., one or two LSTM layers followed by a dense output layer). Choose a loss function such as Mean Squared Error and an optimizer (e.g., Adam). Train the LSTM model on the training set. Monitor the training and validation loss over epochs to ensure the model is learning (you may use early stopping to prevent overfitting).

C. Implement GRU Model: Define a similar neural network using Gated Recurrent Unit layers (tf.keras.layers.GRU or torch.nn.GRU). Ensure it has a comparable number of parameters to the LSTM model for a fair comparison (adjust the number of units if needed). Train the GRU model on the same training data, using the same training procedure and epochs as the LSTM.

D. Evaluation: For both models, forecast the next time step for each sequence in the test set. Compute error metrics on the test data, including Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE).

E. Hyperparameter Experiments: Perform at least one hyperparameter variation for each model. For example, train a second LSTM with a different number of units or an extra LSTM layer, and note the impact on performance. Do the same for GRU (or vary learning rate, batch size, etc.). This will illustrate sensitivity to hyperparameters.

F. Visualization: Plot the model predictions vs. actual values on the test set for a visual comparison. Create a line plot for a subset of the test period (e.g., last 30 days) showing true vs predicted values for each model. This helps interpret where the model is performing well or failing (e.g., does it capture trends or miss sudden changes?).

Baseline Benchmark: Compute a simple baseline forecast (e.g., "naive" forecast where the prediction is just the last observed value, or a basic linear regression on the last few lags). Compare the deep models' performance to this baseline to quantify the value they add.

Interpretability Discussion (1 Bonus Point): Although LSTMs and GRUs are black-box models, examine their internal behavior if possible. For instance, you can plot the values of the LSTM forget gate over time for a test sequence to see how it modulates information, or analyze which input timesteps had the most influence on the prediction (e.g., by partial input perturbation). Discuss any insight gained about how the model is making decisions.

Problem 2 (A-E for 4 Points, F for 3 Bonus Points)

Encoder-Decoder vs. Decoder-Only: The original Transformer (Vaswani et al. 2017) uses an encoder-decoder structure. However, for forecasting, sometimes a decoder-only (auto-regressive) transformer is used. Briefly discuss the difference and why a decoder-only Transformer can be sufficient for time series forecasting.

Research Extension (1 Bonus Point): Identify one transformer-based model tailored for time series forecasting (for example, the Informer or Temporal Fusion Transformer). Provide a short summary of how it improves on the

basic Transformer for time series (e.g., Informer's sparse attention mechanism for long series, or TFT's attention for interpretable multivariate forecasting)

A. Data Preparation: Load the weather time series dataset (see previous exercise). This is likely a multivariate time series (e.g., temperature, pressure, humidity, etc.).

B. Transformer Model Implementation: Construct a transformer-based model for time series forecasting. You have flexibility in design: One approach is to use an encoder-decoder structure: the encoder processes the past 30 days of multivariate data, and the decoder generates the next 7 days of the target (with appropriate masking so that at training time the decoder can attend to previous target values). Alternatively, implement a decoder-only transformer that takes the 30-day history as context and autoregressively predicts each step up to 7 days (feeding back predictions, using masked self-attention to prevent seeing future data). Use library components where possible (e.g., torch.nn.Transformer or Keras MultiHeadAttention layers) rather than coding attention from scratch. Include positional encoding in your model. For instance, in PyTorch you can use nn.Transformer which includes embedding + positional encoding, or manually add sinusoidal positional features to inputs.

C. Training: Train the Transformer model on the training set. Because transformers have many parameters, use techniques to aid learning: e.g., a relatively large batch size (if data allows), learning rate scheduling, and possibly more epochs. Monitor validation loss for hyperparameter tuning. Training might be slower than RNNs, so be mindful of training time.

D. Evaluation – Forecast Accuracy: Evaluate both models (Transformer and LSTM) on the test set by producing 7-day forecasts for each window and comparing to actual values. Use metrics suitable for multi-step forecasts, e.g., Mean Absolute Error and RMSE for each horizon step and/or overall (you can compute an average MAE over all forecasted points, and also check MAE at day +1, +3, +7 separately to see if error grows). If the dataset is multivariate and you used additional features, make sure to also feed the necessary exogenous features to the models at prediction time (if required by your model design).

E. Hyperparameter Tuning: Experiment with at least one or two hyperparameters on the Transformer. For instance, try changing the number of attention heads (e.g., 4 vs 8 heads) or the depth (number of Transformer layers). Observe the effect on validation performance and training time. You might also try different sequence lengths for input or different forecast horizon lengths to see how the transformer handles them versus the LSTM.

F. Attention Visualization: For one example in the test set, visualize the Transformer's attention weights. For instance, pick a specific forecast date and plot the attention scores of the Transformer decoder for that prediction with respect to the input timeline (and/or previous outputs). This could be done by extracting the attention matrix from the model (in PyTorch's nn.Transformer, you can register hooks or use the attn_output_weights if using MultiheadAttention modules). The goal is to interpret which past time steps the model found relevant for predicting a particular future step. Include a brief analysis of this: do the attention weights align with intuitive patterns (e.g., focusing on daily patterns or recent trends)?

Problem 3 (A-F for 4 Points, G for 3 Bonus Points)

A. Select Dataset: Choose a real-world healthcare time series dataset suitable for a classification or anomaly detection task. Examples: an ECG dataset where each time series is labeled as normal or arrhythmia (e.g., the ECG200 dataset from the UCR archive, which contains 200 heartbeat time series for two classes), or a clinical dataset where sequences of patient vital signs are labeled by outcome. Ensure the data is preprocessed (e.g., each series has the same length or you truncate/zero-pad as needed, and consider normalization).

B. Install and Configure Models: Use official or existing implementations for TS2Vec and T-Rep (see the notebooks presented during the lecture). Make sure you understand the input format each expects (TS2Vec may require creating augmented views; T-Rep's API might handle that internally). If needed, use a smaller subset of data to tune any hyperparameters due to time constraints.

C. Unsupervised Training: Train the TS2Vec model on the training portion of your dataset to learn representations. This involves feeding the raw time series (without labels) into TS2Vec's training routine. Similarly, train T-Rep on the same training data (T-Rep will learn time-step level embeddings and an encoder). Both models will output some form of encoded representation for each time series or each time step. For consistency, decide on the representation you will extract for the downstream task. For example, you might use

instance-level embeddings (one vector per entire time series) by aggregating TS2Vec's timestamp embeddings (e.g., average or using the final timestamp) and using T-Rep's output for the final timestamp (or an average as well).

D. Obtain Representations: After training, use the learned models to encode all training and test samples into representation vectors. For TS2Vec, this might mean passing each time series (or sub-sequence) through the trained network to get a sequence of embeddings, then pooling or selecting the relevant part. For T-Rep, you can use trep.encode(data) as in the provided example to get an array of representations. Ensure the dimension of these representations is noted (e.g., TS2Vec default might be 128 or 256 per timestamp, T-Rep default output_dims=128 unless changed).

E. Downstream Classification Task: Using the labeled data, train a simple classifier on top of the learned representations. For example, train a Logistic Regression or Support Vector Machine (SVM) using the instance-level representation of each time series as features to predict its class. (In the ECG example, each heartbeat time series is represented by a learned vector, and you predict "normal" vs "abnormal"). Do this for both TS2Vec-derived embeddings and T-Rep embeddings separately. Evaluate the classification accuracy (or F1-score, etc., depending on the problem) on the test set for each case. If the dataset has a class imbalance, ensure you use appropriate metrics or techniques.

F. Baseline Comparison: For context, also evaluate a baseline approach. For instance, use raw features or a simpler representation: you could take the raw time series (normalized) and flatten it or use basic statistics (mean, std, etc.) as features for the same classifier. Alternatively, train a fully supervised deep learning model (like a small CNN or RNN) directly on the labeled data for comparison. The goal is to see how the self-supervised representations stack up against a naive representation or a fully supervised approach with the same amount of labeled data.

G. Analysis of Representations: Analyze the quality of the learned representations. Perform the following:

- Visualization: Use a dimensionality reduction technique (e.g., t-SNE or PCA) to project the learned representations of the test set into 2D. Create a scatter plot where each point is a time series in the embedding space, colored by its true label. Do this for TS2Vec and T-Rep embeddings. Discuss any observable clustering – e.g., "We see two distinct clusters for the two classes with T-Rep, indicating good class separation, whereas TS2Vec embeddings overlap more" or vice versa.
- Nearest Neighbors: Pick a few example time series and find their nearest neighbors in the representation space (using Euclidean distance or cosine similarity). Check if those neighbors share the same class or similar characteristics. This can qualitatively show whether the embedding is grouping similar time series together meaningfully.
- Robustness test (1 Bonus Point): If time permits, test robustness to missing data: e.g., remove or mask out a portion of the time series and see how the representations change or if the models can still encode them well (this relates to T-Rep's claim of resilience to missing data.

Dataset Healthcare Time Series (e.g., ECG Classification): ECG200 from UCR Archive – This dataset contains 200 electrocardiogram sequences each of length 96, labeled as either normal or arrhythmia. We will use 100 series for training and 100 for testing (balanced classes). The goal is to classify heartbeats using learned representations.

Alternate options: Another possible dataset is the PhysioNet MIT-BIH Arrhythmia ECG dataset (longer heartbeat signals with multiple classes), or a wearable sensor time series dataset for activity recognition (where each series is an acceleration signal labeled by activity). Ensure whichever dataset you choose, it has a clear train/test split and labels for a supervised evaluation. (The ECG200 is convenient and small; if a larger challenge is desired, choose a more complex dataset but be mindful of training time for representation learning models.)