MACHINE LEARNING - HOMEWORK 2
ADVANCED NEURAL NETWORK MODELS FOR TIME SERIES CLASSIFICATION

Cătălin-Alexandru Rîpanu 341C3
Automatic Control and Computers
National University of Science and Technology POLITEHNICA Bucharest
`catalin.ripanu@stud.acs.upb.ro`
Laboratory Assistant: Mihai Trascău

27 May 2024

ABSTRACT

This project aims to explore the implementation of widely used neural network models for time series classification, specifically focusing on **Multi-Layer Perceptrons (MLPs)** and **Convolutional Neural Networks (CNNs)**. These models are renowned for their efficacy in handling complex data patterns and temporal dependencies. The performance evaluation of these models is multifaceted, considering crucial metrics including **Precision**, **Recall**, **Overall Accuracy**, and **F1 Score**. To facilitate implementation and execution, Python library **PyTorch** is leveraged, offering an extensive array of specialized functions tailored for deep learning tasks. This comprehensive approach seeks to provide insights into the behavior and performance of neural network models in the context of time series data, fostering a deeper understanding of their methodologies and practical applications.

1 Data Analysis

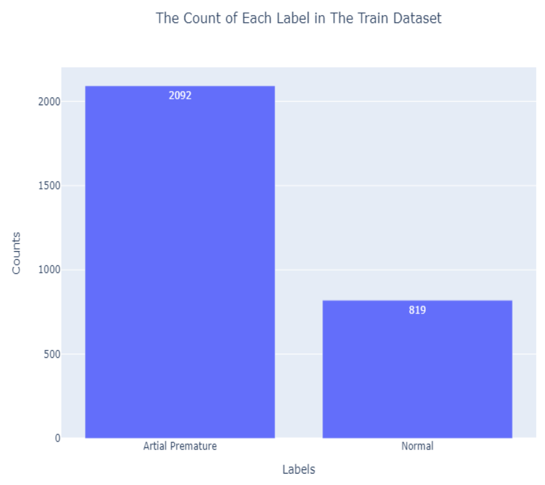
The initial stage comprises thorough data analysis, beginning with an evaluation of class balance within both the training and test datasets from the **MIT-BIH Arrhythmia Database (MIT-BIH)** and the **Physikalisch-Technische Bundesanstalt Diagnostic Database (PTBDB)** for valuable insights into the underlying data structure and helps in identifying any potential biases or anomalies within the datasets. Last but not least, this initial assessment ensures a comprehensive understanding of the distribution of classes, which is crucial for the robustness of subsequent modeling efforts.

For both the MIT-BIH and PTBDB datasets, the mean and standard deviation are plotted to understand the central tendency and dispersion of the data. These plots are essential in highlighting the variability and consistency of the data, which can significantly impact the performance of the neural network models.

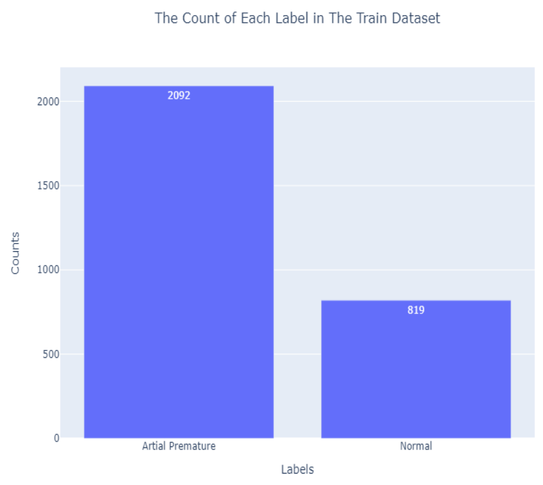
Moreover, examples from both datasets are plotted to visually inspect the time series data. This step involves selecting representative samples from each class and plotting their time series to observe patterns, trends, and any distinctive features. Such visual inspection helps in gaining an intuitive understanding of the data, which can inform the choice of preprocessing techniques and model architectures.

Finally, the significance of each attribute in predicting the target variable, labeled, for example, **"Diagnostic"** in **Patients** dataset, is elucidated through individualized plots. These plots provide a nuanced understanding of attribute relevance, facilitating informed decision-making in subsequent modeling processes. By visualizing the importance of each attribute, we can better understand which features contribute most significantly to the predictive power of the models, thereby optimizing the feature selection process.

This comprehensive data analysis phase ensures that both the MIT-BIH and PTBDB datasets are thoroughly understood and appropriately prepared for the implementation of neural network models for time series classification.

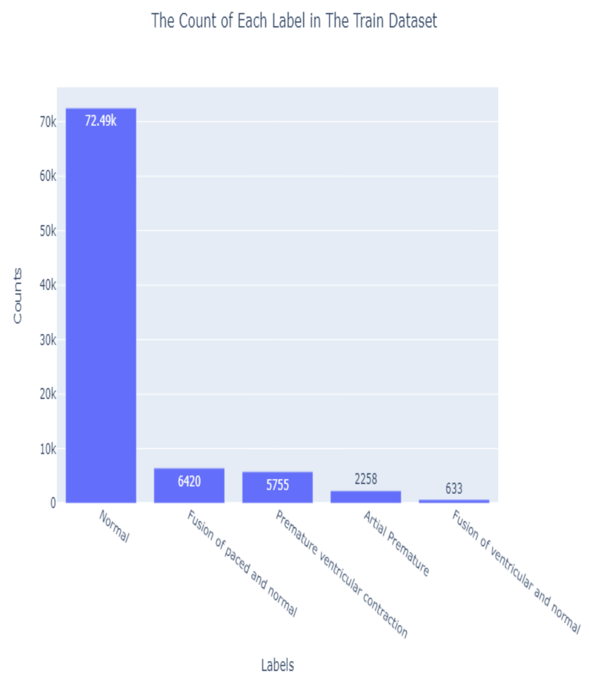


(a) Train Set

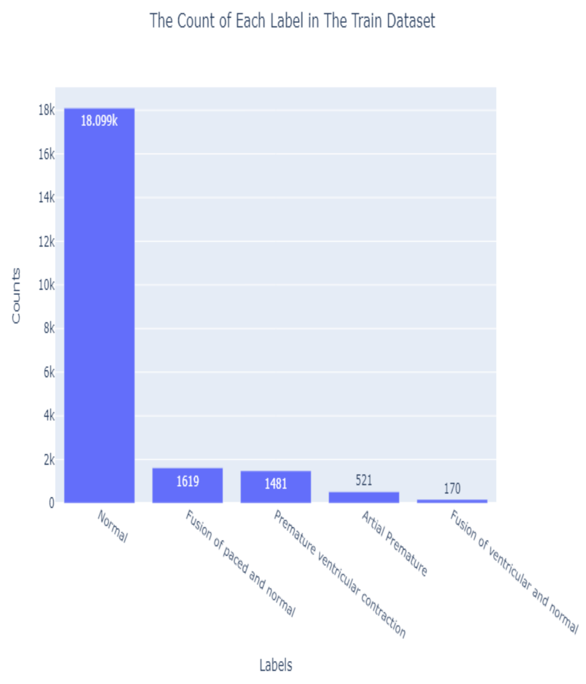


(b) Test Set

Figure 1: Class distribution in both datasets (PTBDB)



(a) Train Set



(b) Test Set

Figure 2: Class distribution in both datasets (MIT_BIH)

Model Types and Configurations

Multi-Layer Perceptron (MLP)

- **From Scratch MLP**: Implemented using numpy and manual backpropagation.
- **PyTorch MLP**: Utilized PyTorch's automatic differentiation and built-in modules for defining MLP architectures.

Table Description:

1. Architecture Configurations:

- **Mode:** Specifies the optimization algorithm used, either Stochastic Gradient Descent (SGD) or momentum-based optimization.
- **Hidden Units:** Indicates the number of units (neurons) in the hidden layers.
- **Learning Rate (lr):** The step size used in updating the weights during training.
- **Epochs:** The number of times the entire training dataset is passed through the network.
- **Batch Size:** The number of training samples used in one iteration to update the model weights.
- **Layer Structure:** The sequence and type of layers in the MLP, including the input shape, hidden layers, and output layers with their respective activation functions (ReLU).

2. Performance Metrics:

- **Precision:** The ratio of correctly predicted positive observations to the total predicted positives.
- **Recall:** The ratio of correctly predicted positive observations to all observations in the actual class.
- **F1 Score:** The weighted average of Precision and Recall, providing a balance between the two.
- **Accuracy:** The ratio of correctly predicted observations to the total observations.

1.1 From Scratch MLP (Patients Dataset)

Insights:

- Configurations using the momentum-based optimizer generally perform better in terms of precision, recall, F1 score, and accuracy compared to those using SGD.
- Increasing the number of hidden units does not necessarily improve performance, as seen in the comparison between configurations with 128 and 64 hidden units.
- The structure and depth of the network, as well as the choice of optimizer, play crucial roles in determining the overall performance of the model.

Arch. Configs	Precision	Recall	F1	Accuracy
mode: SGD | Hidden_units: 128 | lr: 0.005 | Epoch: 200 | Batch_size: 2000 | Linear(Ex_Shape, 128) ReLU Linear(128, 7)	0.68	0.68	0.66	0.69
mode: momentum | Hidden_units: 128 | lr: 0.005 | Epoch: 200 | Batch_size: 2000 | Linear(Ex_Shape, 128) ReLU Linear(128, 7)	0.73	0.73	0.73	0.74
mode: SGD | Hidden_units: 64 | lr: 0.005 | Epoch: 200 | Batch_size: 2000 | Linear(Ex_Shape, 64) ReLU Linear(64, 32) ReLU Linear(32, 7)	0.68	0.69	0.68	0.70
mode: momentum | Hidden_units: 64 | lr: 0.005 | Epoch: 200 | Batch_size: 2000 | Linear(Ex_Shape, 64) ReLU Linear(64, 32) ReLU Linear(32, 7)	0.74	0.73	0.73	0.74

1.2 From Scratch MLP (PTBDB Dataset)

Insights:

- Configurations using the momentum-based optimizer generally perform better in terms of precision, recall, F1 score, and accuracy compared to those using SGD.

- For both hidden unit configurations (128 and 64), momentum optimization consistently yields higher performance metrics than SGD.

- The structure and depth of the network, as well as the choice of optimizer, significantly impact the overall performance of the model.

Arch. Configs	Precision	Recall	F1	Accuracy
mode: momentum | Hidden_units: 128 | lr: 0.005 | Epoch: 150 | Batch_size: 100 | Linear(Ex_Shape, 128) ReLU Linear(128, 2)	0.93	0.94	0.93	0.95
mode: SGD | Hidden_units: 128 | lr: 0.005 | Epoch: 150 | Batch_size: 100| Linear(Ex_Shape, 128) ReLU Linear(128, 2)	0.82	0.80	0.81	0.85
mode: momentum | Hidden_units: 64 | lr: 0.005 | Epoch: 150 | Batch_size: 100 | Linear(Ex_Shape, 64) ReLU Linear(64, 32) ReLU Linear(10, 2)	0.94	0.96	0.95	0.96
mode: momentum | Hidden_units: 64 | lr: 0.005 | Epoch: 150 | Batch_size: 100 | Linear(Ex_Shape, 64) ReLU Linear(64, 32) ReLU Linear(10, 2)	0.88	0.87	0.88	0.90

1.3 PyTorch MLP (Patients Dataset)

Insights:

- Configurations using the momentum-based optimizer generally perform better in terms of precision, recall, F1 score, and accuracy compared to those using SGD.

- Increasing the number of hidden units does not necessarily improve performance, as seen in the comparison between configurations with 128 and 64 hidden units.

- The structure and depth of the network, as well as the choice of optimizer, play crucial roles in determining the overall performance of the model.

Arch. Configs	Precision	Recall	F1	Accuracy
mode: Adam() | Hidden_units: 128 | lr: 0.005 | Epoch: 200 | Batch_size: 2000 | Linear(Ex_Shape, 128) ReLU Linear(128, 7) ReLU	0.78	0.77	0.76	0.77
mode: Adam() | Hidden_units: 128 | lr: 0.005 | Epoch: 200 | Batch_size: 2000 | Linear(Ex_Shape, 128) ReLU Linear(128, 7) Sigmoid	0.78	0.78	0.77	0.78
mode: SGD() | Hidden_units: 128 | lr: 0.005 | Epoch: 200 | Batch_size: 2000 | Linear(Ex_Shape, 128) ReLU Linear(128, 7) TanH	0.69	0.67	0.65	0.68
mode: Adam() | Hidden_units: 64 | lr: 0.005 | Epoch: 200 | Batch_size: 2000 | Linear(Ex_Shape, 64) ReLU Linear(64, 32) ReLU Linear(10, 7) ReLU	0.72	0.72	0.71	0.73
mode: Adam() | Hidden_units: 64 | lr: 0.005 | Epoch: 200 | Batch_size: 2000 | Linear(Ex_Shape, 64) ReLU Linear(64, 32) ReLU Linear(10, 7) Sigmoid	0.77	0.76	0.75	0.77
mode: SGD() | Hidden_units: 64 | lr: 0.005 | Epoch: 200 | Batch_size: 2000 | Linear(Ex_Shape, 64) ReLU Linear(64, 32) ReLU Linear(10, 7) TanH	0.48	0.48	0.38	0.5

1.4 PyTorch MLP (PTBDB Dataset)

Arch. Configs	Precision	Recall	F1	Accuracy
mode: Adam() | Hidden_units: 128 | lr: 0.005 | Epoch: 200 | Batch_size: 2000 | Linear(Ex_Shape, 128) ReLU Linear(128, 2) ReLU	0.97	0.96	0.96	0.97
mode: SGD() | Hidden_units: 64 | lr: 0.005 | Epoch: 200 | Batch_size: 2000 | Linear(Ex_Shape, 64) ReLU Linear(64, 32) ReLU Linear(10, 2) ReLU6	0.14	0.50	0.22	0.28
mode: Adam() | Hidden_units: 64 | lr: 0.005 | Epoch: 200 | Batch_size: 2000 | Linear(Ex_Shape, 64) ReLU Linear(64, 32) ReLU Linear(10, 2) Sigmoid	0.95	0.94	0.95	0.96

1.5 PyTorch MLP (MIT-BIH Dataset)

Arch. Configs	Precision	Recall	F1	Accuracy
mode: Adam() | Hidden_units: 128 | lr: 0.005 | Epoch: 200 | Batch_size: 2000 | Linear(Ex_Shape, 128) ReLU Linear(128, 5) TanH	0.97	0.96	0.96	0.97

Convolutional Neural Networks (CNNs)

- **ResNet CNN**: A residual network designed for handling time series data with varying depths and configurations.

- **Inception CNN**: A network that employs multiple kernel sizes to capture diverse features in time series data.

1.6 PyTorch CNN - InceptionModel (PTBDB Dataset)

Arch. Configs	Precision	Recall	F1	Accuracy
mode: Adam() | Num_blocks = 1 | Kernel_size = 41	0.78	0.74	0.76	0.82
mode: RAdam() | Num_blocks = 1 | Kernel_size = 41	0.77	0.74	0.75	0.81
mode: Adam() | Num_blocks = 1 | Kernel_size = 90	0.81	0.77	0.78	0.84
mode: Adam() | Num_blocks = 1 | Kernel_size = 5	0.67	0.59	0.59	0.74
mode: Adam() | Num_blocks = 3 | Kernel_size = 20	0.67	0.59	0.59	0.74
mode: Adam() | Num_blocks = 10 | Kernel_size = 20	0.99	0.98	0.98	0.99

1.7 PyTorch CNN - InceptionModel (MIT-BIH Dataset)

Arch. Configs	Precision	Recall	F1	Accuracy
mode: Adam() | Num_blocks = 10 | Kernel_size = 20	0.99	0.98	0.98	0.97

1.8 PyTorch CNN - ResNetBaseline (PTBDB Dataset)

Arch. Configs	Precision	Recall	F1	Accuracy
mode: Adam() | in_channels = 1	0.85	0.92	0.87	0.88

Experimental Configurations

Batch Size and Epochs

- Various batch sizes were experimented with, ranging from 100 to 2000 to find the optimal balance between convergence speed and stability.
- The number of epochs varied from 50 to 200, allowing for observation of the models' learning and generalization capabilities over time.

Optimizers

- **Adam Optimizer**: Used for its adaptive learning rate capabilities, which help in faster convergence.
- **SGD with Learning Rate Scheduler**: Standard Stochastic Gradient Descent combined with a learning rate scheduler that reduces the learning rate upon reaching certain milestones to fine-tune the training process.

MLP Architectures

- **Simple MLP**: Fewer hidden layers with a moderate number of neurons, suitable for simpler datasets.
- **Complex MLP**: More hidden layers with a larger number of neurons, aimed at capturing complex patterns in the data.

CNN Architectures for Time Series Data

- **ResNet CNN**:
  - Varying depths and block configurations to handle different scales of data representation.
  - Blocks included neck channels and strides adjusted to balance between depth and computational efficiency.
- **Inception CNN**:
  - Multiple kernel sizes (e.g., 5x5, 20x20, 41x41, 90x90) used in parallel to capture multi-scale features.
  - Configuration adjustments to optimize performance for time series data.

Results and Observations

From Scratch MLP vs. PyTorch MLP

- The PyTorch MLP was significantly easier to implement and modify due to PyTorch's automatic differentiation and extensive library support.
- Performance-wise, the PyTorch MLP showed better convergence and stability, especially with complex architectures.

ResNet CNN vs. Inception CNN

- **ResNet CNN**:
  - Benefitted from residual connections that helped mitigate the vanishing gradient problem.
  - Showed robustness in handling deeper networks and complex time series patterns.
- **Inception CNN**:
  - Effectively captured multi-scale features, making it versatile for various time series data patterns.
  - Computationally intensive due to multiple convolution paths, but provided rich feature representations.

Optimizer Performance

- **Adam**:
  - Provided faster convergence across different model configurations.
  - Effective in scenarios with sparse gradients or noisy data.
- **SGD with Learning Rate Scheduler**:
  - Helped in fine-tuning models by reducing the learning rate at specified intervals, improving final model performance.
  - Required careful tuning of initial learning rates and scheduler milestones.

Conclusion

The experiments showcased the flexibility and power of PyTorch for implementing and tuning deep learning models. The use of advanced architectures like ResNet and Inception, combined with effective optimizers, significantly enhanced the ability to model complex time series data. The manual implementations, while insightful for understanding underlying mechanics, were outperformed by the more sophisticated PyTorch implementations in terms of ease of use, scalability, and performance.

These statistics provide a detailed overview of each model's configuration and performance metrics. By including information on feature scaling, selection methods, and evaluation scores (for **Patients** dataset), stakeholders gain deeper insights into the models' behavior and effectiveness for the given task.

In conclusion, it's evident that the choice of hyperparameters significantly influences the predictive power of all the models analyzed above.

This observation underscores the importance of careful tuning and optimization of hyperparameters to achieve optimal performance in predictive modeling tasks. By selecting appropriate hyperparameters practitioners can enhance the predictive capabilities of machine learning models and improve their effectiveness in real-world applications. Additionally, thorough experimentation and validation processes are essential for identifying the most suitable hyperparameter configurations for specific datasets and tasks, ultimately leading to more accurate and reliable predictions.