Application of Machine Learning in Gravitational Wave Data Analysis: Transformers, Deep Belief Networks, and Graph Neural Networks

Shufan Dong[1]

[1]email: shufandong6011@gmail.com

Abstract

The detection and analysis of gravitational waves (GW) have become a pivotal aspect of modern astrophysics. Applying the use of machine learning (ML) techniques, particularly advanced neural network models, has significantly enhanced the capacity to detect and interpret GW signals containing vast amounts of noisy data. This paper explores the application of various ML models, including Transformer models, Deep Belief Networks (DBNs), and Graph Neural Networks (GNNs), to analyze GW data. The processes of data segmentation, augmentation, preparation, model training, and model evaluation are presented, demonstrating the efficacy of these models in identifying and classifying GW signals.

Contents

1 Introduction

Gravitational waves (GWs), ripples in spacetime caused by strong astrophysical events, were first directly detected by the LIGO and Virgo collaborations in 2015. These detections brought importance to advanced data analysis techniques due to the weak nature of GW signals and the presence of significant background noise. Various ML applications have offered robust tools for signal detection, noise reduction, and data interpretation. The preprocessing of GW data is also critical to the certainty of smooth ML applications, and more details about it can be found in [35]. Besides a brief introduction to GW data preparation that's discussed and expounded more closely in [36], this paper focuses on the utilization of Transformer, DBN, and GNN models for GW data analysis, demonstrating their distinct advantages and methodologies.

2 Data Preparation

2.1 Importing Libraries

The analysis begins with importing essential libraries for data processing, visualization, and ML model implementation.

```python
import numpy as np
import pandas as pd
import requests, os
import matplotlib.pyplot as plt
from scipy.signal import butter, filtfilt
from sklearn.preprocessing import StandardScaler
import tensorflow as tf
import torch
from torch.utils.data import Dataset, DataLoader
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
! pip install torch_geometric
from torch_geometric.nn import GCNConv, global_mean_pool
from torch_geometric.data import Data, DataLoader

import warnings
warnings.filterwarnings('ignore')
# Set tf logging level to suppress warnings and info messages
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
# This ensures that the logging level is set before any tf code runs
tf.get_logger().setLevel('ERROR')
```

Figure 1: Visualization of all the libraries imported.

Each library serves a specific purpose:

- NumPy and Pandas for number and data manipulation.
- Requests and OS for web requests and system operations.
- Matplotlib for data visualization.
- SciPy for signal processing.
- Scikit-Learn for data preprocessing and splitting.
- TensorFlow and PyTorch for building and training ML models.
- Torch Geometric for implementing GNNs.
- Warnings to suppress unnecessary warnings for smooth execution of the codes.

2.2 Data Segmentation and Labeling

Segmentation and augmentation are crucial for managing and enhancing the dataset. The continuous time-series GW data is segmented into smaller chunks, and data augmentation techniques are applied to increase the dataset size, thereby improving model performance. Since the data preparation and re-sampling steps are different for all 3 ML models, the shapes of the data after augmentation do vary.

```
Segments shape: (2047, 8192)
Labels shape: (2047,)
```

Figure 2: The shape of the segments and labels before augmentation.

2.3 Data Preparation

2.3.1 Transformer

Data preparation for the Transformer model involves creating a custom, plain dataset class used to convert the data into PyTorch tensors and using PyTorch's DataLoader for batching, shuffling, and splitting.

```
class GWDataset(Dataset):
    def __init__(self, segments, labels):
        self.segments = segments
        self.labels = labels

    def __len__(self):
        return len(self.segments)

    def __getitem__(self, idx):
        segment = torch.tensor(self.segments[idx], dtype=torch.float32)
        label = torch.tensor(self.labels[idx], dtype=torch.long)
        return segment, label

# Create dataset and dataloader
dataset = GWDataset(segments_aug, labels_aug)
train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
train_dataset, test_dataset = torch.utils.data.random_split(dataset, [train_size, test_size])

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

Figure 3: For batching and shuffling purposes, the data is split into training (80%) and testing (20%) datasets using Python functions and PyTorch.

2.3.2 DBN

For the DBN model, the data is split into training and testing sets using Scikit-Learn's train_test_split function, and it's then converted to PyTorch tensors.

```
# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(segments, labels, test_size=0.2, random_state=42)

X_train_aug, y_train_aug = augment_data(X_train, y_train)

# Pytorch tensors
X_train_aug = torch.tensor(X_train_aug, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_train_aug = torch.tensor(y_train_aug, dtype=torch.float32).view(-1, 1)
y_test = torch.tensor(y_test, dtype=torch.float32).view(-1, 1)
```

Figure 4: The data here is split into training (80%) and testing (20%) datasets with simply the Scikit-Learn's train_test_split function.

2.3.3 GNN

Graph-structured data is created for the GNN model, which captures complex relationships and structures in the GW data.

```
# Graph data
def create_graph_data(gw_signals, labels):
    graphs = []
    for signal, label in zip(gw_signals, labels):
        node_features = torch.tensor(signal, dtype=torch.float).unsqueeze(1)
        edge_index = torch.tensor([[i, i+1] for i in range(len(signal)-1)], dtype=torch.long).t().contiguous()
        y = torch.tensor([label], dtype=torch.long)
        graph = Data(x=node_features, edge_index=edge_index, y=y)
        graphs.append(graph)
    return graphs

graph_data = create_graph_data(segments_aug, labels_aug)

# Dataloader
data_loader = DataLoader(graph_data, batch_size=256, shuffle=True)
```

Figure 5: For its spatial capturing capabilities, GNN requires graphical data input, and PyTorch's DataLoader is utilized for batching and shuffling

Loop Over Signals and Labels

- The `zip(gw_signals, labels)` function pairs each signal with its corresponding label.
- `torch.tensor(signal, dtype=torch.float)`: converts the signal into a PyTorch tensor.

- `.unsqueeze(1)`: adds an extra dimension to the tensor.
- `[[i, i+1] for i in range(len(signal)-1)]`: creates pairs of consecutive indices (i, i+1), representing the edges between consecutive nodes in the graph.
- `torch.tensor(..., dtype=torch.long)`: converts index pairs into a PyTorch tensor.
- `.t()`: transposes the tensor.
- `.contiguous()`: ensures that the tensor's memory layout is compatible for efficient processing.
- `torch.tensor([label], dtype=torch.long)`: converts the label into a PyTorch tensor.
- `Data(x=node_features, edge_index=edge_index, y=y)`: creates a graph data object using the Data class from PyTorch Geometric.

The function at the end returns the list of graph data objects.

3 Model Training and Evaluation

3.1 Transformer

A Transformer model is defined and trained for time-series data classification, utilizing its ability to capture long-range dependencies in the data.

```
# Set hyperparams
input_dim = 1 # time-series data
model_dim = 128
num_heads = 8
num_layers = 4
lr = 1e-4
batch_size = 256
dropout_rate = 0.2
output_dim = 2 # binary classificiation of event presence
num_epochs = 5
```

Figure 6: All the hyperparameters needed to train the Transformer model.

```python
class TransformerModel(nn.Module):
    def __init__(self, input_dim, model_dim, num_heads, num_layers, dropout_rate, output_dim):
        super(TransformerModel, self).__init__()
        self.embedding = nn.Linear(input_dim, model_dim)
        self.positional_encoding = nn.Parameter(torch.zeros(1, 8192, model_dim))
        encoder_layers = nn.TransformerEncoderLayer(model_dim, num_heads, dim_feedforward=2048, dropout=dropout_rate)
        self.transformer_encoder = nn.TransformerEncoder(encoder_layers, num_layers)
        self.fc_out = nn.Linear(model_dim, output_dim)

    def forward(self, x):
        x = self.embedding(x.unsqueeze(-1)) + self.positional_encoding[:, :x.size(1), :]
        x = self.transformer_encoder(x)
        x = x.mean(dim=1)  # Global avg pooling
        x = self.fc_out(x)
        return x
```

Figure 7: Defining the Transformer model.

The class inherits from the base class, `nn.Module`, for all NN modules in PyTorch.

`__init__()` function:

- `nn.Linear(input_dim, model_dim)` is an embedding layer that linearly projects the input from `input_dim` to `model_dim`.
- `nn.Parameter(torch.zeros(1, 8192, model_dim))` creates a positional encoding tensor with shape $(1, 8192, model\_dim)$. This encodes positional information to help the model understand the order of input.
- `nn.TransformerEncoderLayer` defines a transformer encoder layer with:
  - `model_dim`: the dimension of the model.
  - `num_heads`: the number of attention heads.
  - `dim_feedforward=2048`: the dimension of the feedforward network.
  - `dropout=dropout_rate`: the dropout rate.
- `nn.TransformerEncoder` stacks the encoder layers to form the complete transformer encoder.
- `nn.Linear(model_dim, output_dim)` linearly projects the output from `model_dim` to `output_dim`.

`forward()` function:

- `x.unsqueeze(-1)` adds an extra dimension to `x`, making its shape compatible for the embedding layer.
- `self.embedding(x.unsqueeze(-1))` applies the linear transformation to the input.
- `+ self.positional_encoding[:, :x.size(1), :]` adds the positional encoding to the embedded input.

- `self.transformer_encoder(x)` processes the input through the transformer encoder stack.
- `x.mean(dim=1)` performs global average pooling across the sequence dimension, resulting in a tensor of shape (batch_size, model_dim).
- `self.fc_out(x)` linearly transforms the pooled tensor to the desired output dimension.
- The final output tensor is then returned.

```python
def train_and_evaluate(model, train_loader, test_loader, criterion, optimizer, num_epochs):
    train_losses = []
    test_accuracies = []

    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        for segments_aug, labels_aug in train_loader:
            optimizer.zero_grad()
            outputs = model(segments_aug)
            loss = criterion(outputs, labels_aug)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        train_loss = running_loss / len(train_loader)
        train_losses.append(train_loss)
        print(f'Epoch {epoch+1}/{num_epochs}, Loss: {train_loss}')

        model.eval()
        correct = 0
        total = 0
        with torch.no_grad():
            for segments_aug, labels_aug in test_loader:
                outputs = model(segments_aug)
                _, predicted = torch.max(outputs.data, 1)
                total += labels_aug.size(0)
                correct += (predicted == labels_aug).sum().item()

        test_accuracy = correct / total
        test_accuracies.append(test_accuracy)

    return train_losses, test_accuracies
```

Figure 8: Defining the function for training and evaluating the Transformer model.

`train_and_evaluate()` function:

- Epoch Loop: iterates over the epochs.
  - `model.train()`: sets the model to training mode.
  - `running_loss` is initialized to 0.0 to accumulate the training loss over all batches in the epoch.
  - Batch Loop: iterates over all batches in the `train_loader`.
    - `optimizer.zero_grad()`: clears the gradients of all optimized parameters.
    - `outputs = model(segments_aug)`: computes the model outputs for the input batch.
    - `loss = criterion(outputs, labels_aug)`: calculates the loss between the predicted outputs and the true labels.
    - `loss.backward()`: computes the gradient of the loss.
    - `optimizer.step()`: updates the model parameters using the computed gradients.
    - `running_loss += loss.item()`: adds the batch loss to the running total loss for the epoch.
  - `train_loss = running_loss / len(train_loader)`: calculates the average training loss for the epoch.
  - `train_losses.append(train_loss)`: appends the average training loss to `train_losses`.
  - `model.eval()`: sets the model to evaluation mode.
  - `with torch.no_grad()`: disables gradient computation, which reduces memory usage and speeds up computations.
  - Batch Loop: iterates over all batches in the `test_loader`.
    - `outputs = model(segments_aug)`: computes the model outputs for the input batch.
    - `_, predicted = torch.max(outputs.data, 1)`: finds the one with the highest predicted score for each sample.
    - `total += labels_aug.size(0)` and `correct += (predicted == labels_aug).sum().item()`: updates the total number of samples and the number of correct predictions.
- The function returns two lists: `train_losses`, containing the average training loss for each epoch, and `test_accuracies`, containing the test accuracy for each epoch.

```python
# Def model
model = TransformerModel(input_dim, model_dim, num_heads, num_layers, dropout_rate, output_dim)

# Train model
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=lr)
train_losses, test_accuracies = train_and_evaluate(model, train_loader, test_loader, criterion, optimizer, num_epochs)
```

Figure 9: Building and training the Transformer model.

3.2 DBN

A DBN is trained for binary classification, capturing hierarchical representations in the data.

```python
# Def DBN
class DBN(nn.Module):
    def __init__(self):
        super(DBN, self).__init__()
        self.layer1 = nn.Linear(X_train_aug.shape[1], 256)
        self.layer2 = nn.Linear(256, 128)
        self.layer3 = nn.Linear(128, 64)
        self.output = nn.Linear(64, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.sigmoid(self.layer1(x))
        x = self.sigmoid(self.layer2(x))
        x = self.sigmoid(self.layer3(x))
        x = self.sigmoid(self.output(x))
        return x

model = DBN()
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

Figure 10: Defining the DBN model.

The class inherits from the base class, `nn.Module`, for all NN modules in PyTorch.

`__init__()` function:

- `self.layer1` takes the input data and outputs 256 features.
- `self.layer2` takes the 256 features from `layer1` and outputs 128 features.
- `self.layer3` takes the 128 features from `layer2` and outputs 64 features.
- `self.output` takes the 64 features from `layer3` and outputs a single feature for binary classification or regression.
- Sigmoid activation is used.

`forward()` function:

- It defines the forward pass of the network, which is the way input data flows through the network shown in the constructor.
- The final output `x` is returned. It will be in the range of (0, 1), which is fit for binary classification tasks.

```
# Train model
train_losses = []
train_accuracies = []
val_losses = []
val_accuracies = []

epochs = 100
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
    outputs = model(X_train_aug)
    loss = criterion(outputs, y_train_aug)
    loss.backward()
    optimizer.step()

    # Calc training accuracy
    predicted = (outputs >= 0.5).float()
    accuracy = (predicted.eq(y_train_aug).sum() / float(y_train_aug.shape[0])).item()

    train_losses.append(loss.item())
    train_accuracies.append(accuracy)

    # Evalute model
    with torch.no_grad():
        val_outputs = model(X_test)
        val_loss = criterion(val_outputs, y_test)
        val_predicted = (val_outputs >= 0.5).float()
        val_accuracy = (val_predicted.eq(y_test).sum() / float(y_test.shape[0])).item()

        val_losses.append(val_loss.item())
        val_accuracies.append(val_accuracy)

    print(f'Epoch [{epoch+1}/{epochs}]\n  Loss: {loss.item():.4f}, Accuracy: {accuracy:.4f}\n  Val Loss: {val_loss.item():.4f}, Val Accuracy: {val_accuracy:.4f}')
```

Figure 11: Training and evaluating the DBN model.

Epoch Loop: iterates over the epochs.

- `model.train()`: sets the model to training.
- `optimizer.zero_grad()`: clears the gradients of all optimized parameters.

- `outputs = model(X_train_aug)`: processes the input data `X_train_aug` and produces outputs.
- `loss = criterion(outputs, y_train_aug)`: calculates the difference between the outputs and the true labels `y_train_aug`.
- `loss.backward()`: performs backpropagation to compute the gradients of the loss respective to the parameters.
- `optimizer.step()`: updates the parameters using the computed gradients.
- `predicted = (outputs >= 0.5).float()`: converts the outputs to binary predictions with a threshold of 0.5.
- `accuracy = (predicted.eq(y_train_aug).sum() / float(y_train_aug.shape[0])).item()`: compares the predicted labels to the true labels and calculates the accuracy.
- `with torch.no_grad()`: disables gradient computation, which reduces the memory used and speeds up computations.
- `val_outputs = model(X_test)`: processes the test data `X_test`.
- `val_loss = criterion(val_outputs, y_test)`: calculates the difference between the outputs and the true labels `y_test`.
- `val_predicted = (val_outputs >= 0.5).float()`: converts the outputs to binary predictions.
- `val_accuracy = (val_predicted.eq(y_test).sum() / float(y_test.shape[0])).item()`: calculates the accuracy of the predictions on the test data.

3.3 GNN

A GNN is trained for classification, using the graph structure of the data to capture complex relationships.

```python
# Def GNN
class GNN(torch.nn.Module):
    def __init__(self, in_channels):
        super(GNN, self).__init__()
        self.conv1 = GCNConv(in_channels=in_channels, out_channels=16)
        self.conv2 = GCNConv(in_channels=16, out_channels=32)
        self.fc = torch.nn.Linear(32, 2)  # Binary classification


    def forward(self, data):
        x, edge_index, batch = data.x, data.edge_index, data.batch
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = self.conv2(x, edge_index)
        x = F.relu(x)
        x = global_mean_pool(x, batch)
        x = self.fc(x)
        return F.log_softmax(x, dim=1)

in_channels = graph_data[0].x.shape[1]
model = GNN(in_channels=in_channels)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
criterion = torch.nn.CrossEntropyLoss()
```

Figure 12: Defining the GNN model.

The class inherits from the base class, `nn.Module`, for all NN modules in PyTorch.

`__init__()` function:

- `self.conv1 = GCNConv(in_channels=in_channels, out_channels=16)`: initializes the first graph convolutional layer with input data and 16 output features.
- `self.conv2 = GCNConv(in_channels=16, out_channels=32)`: initializes the second graph convolutional layer with 16 input features from the first layer and 32 output features.
- `self.fc = torch.nn.Linear(32, 2)`: initializes a fully connected layer that takes 32 input features from the second layer and outputs 2 features used for binary classification.

`forward() function`:

- It defines the forward pass of the network, which is the way input data flows through the network shown in the constructor.

- `x = F.relu(x)`: applies the ReLU activation.
- `x = global_mean_pool(x, batch)`: applies global mean pooling to obtain a graph-level representation.
- `x = self.fc(x)`: applies the fully connected layer to the graph-level representation.
- `return F.log_softmax(x, dim=1)`: applies the log softmax function, converting the raw scores into log-probabilities for classification tasks.

```python
# Training history
train_losses = []
train_accuracies = []

# Training loop
def train():
    model.train()
    epoch_loss = 0
    correct = 0
    for data in data_loader:
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, data.y)
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()
        pred = output.argmax(dim=1)
        correct += (pred == data.y).sum().item()
    train_losses.append(epoch_loss / len(data_loader))
    print(f"Loss: {epoch_loss / len(data_loader)}")
    train_accuracies.append(correct / len(graph_data))
    print(f"Accuracy: {correct / len(graph_data)}")

# Train & evalute GNN
epochs = 10
for epoch in range(epochs):
    print(f"Epoch {epoch+1}/{epochs}")
    train()
```

Figure 13: Training and evaluating the GNN model.

`train()` function:

- `model.train()`: sets the model to training.
- Batch Loop: Iterates over all batches in the `data_loader`.

- `optimizer.zero_grad()`: clears the gradients of all optimized parameters.
- `output = model(data)`: passes the input data to get predictions.
- `loss = criterion(output, data.y)`: calculates the loss between the output and the true labels.
- `loss.backward()`: compute the gradient of the loss respective to the parameters.
- `optimizer.step()`: update the parameters with the computed gradients.
- `.item()` converts the tensor to a number.
- `pred = output.argmax(dim=1)`: obtain the prediction with the index of the highest log probability.

Epoch Loop: applies `train()` function over epochs.

4 Model Visualization

Visualizing the training loss and accuracy over epochs is critical for understanding model performance. The following plots show the performance metrics for the Transformer, DBN, and GNN models.

4.1 Transformer

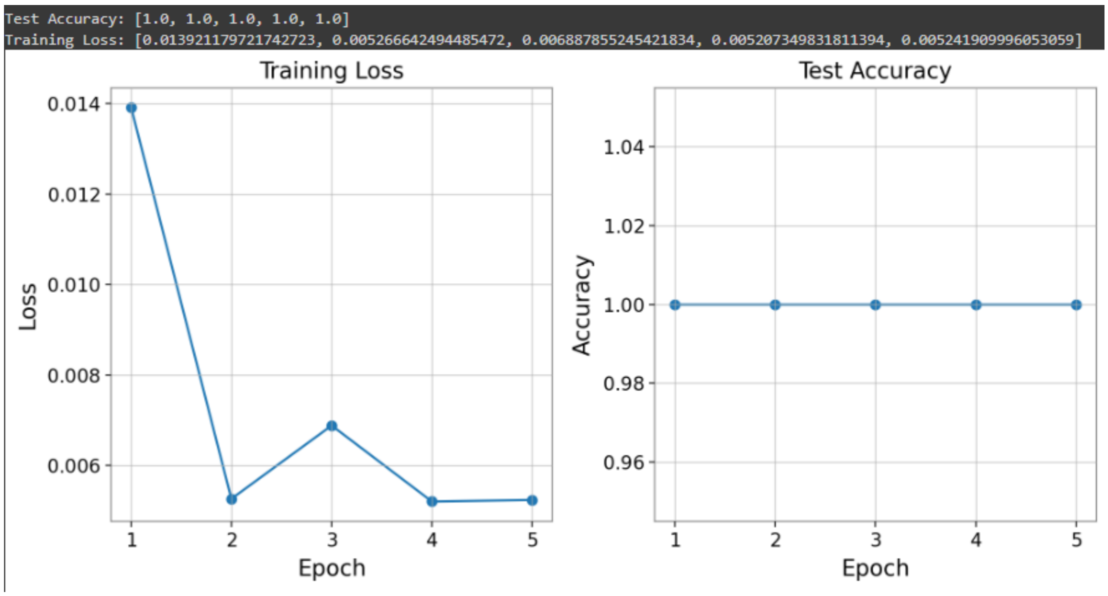

Figure 14: These plots show the training history of the Transformer model, including the loss and accuracy evaluation.

4.2 DBN

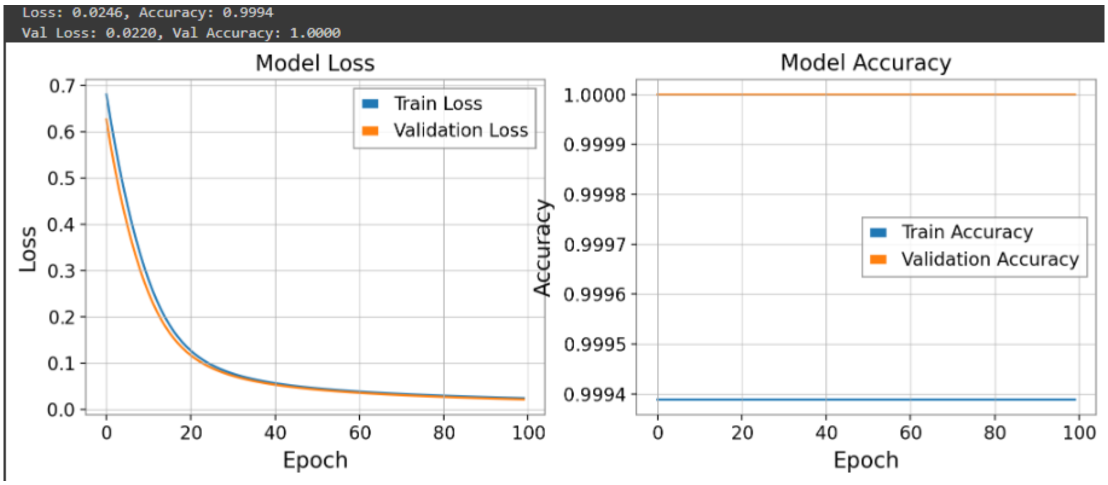

Figure 15: These plots show the training history of the DBN model, including the loss and accuracy evaluation.

4.3 GNN

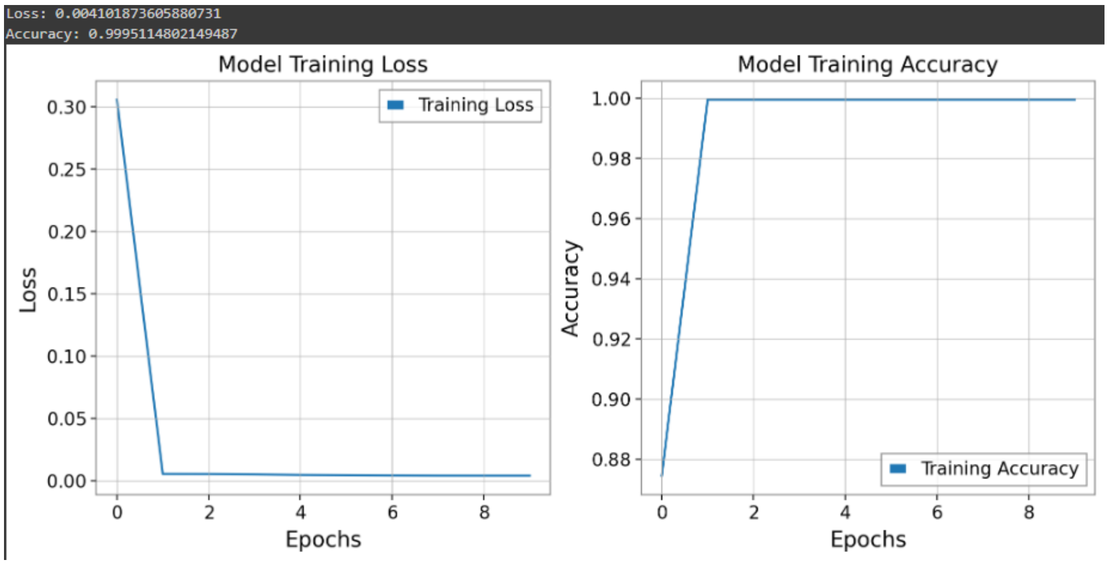

Figure 16: These plots show the training history of the GNN model, including the loss and accuracy evaluation.

5 Conclusion

This comprehensive workflow expounds the necessary steps for preparing data, building, training, evaluating, and visualizing various ML models' capabilities and performance. The ML models (Transformer, DBN, and GNN) bring unique advantages to the field of astrophysical research, enhancing the reliability and applicability of GW data analysis. The application of these advanced ML techniques demonstrates significant potential in improving GW signal detection and classification of the occurrence of merger celestial events, thereby contributing to the broader field of astrophysics.

References

[1] Abbott, B.P., et al. "Population Properties of Compact Objects from the Second LIGO-Virgo Gravitational-Wave Transient Catalog." *Astrophysical Journal Letters*, vol. 913, 2021.

[2] Zheng, Y., et al. "Angular Power Spectrum of Gravitational-Wave Transient Sources as a Probe of the Large-Scale Structure." *Physical Review Letters*, vol. 131, 171403, 2023.

[3] Ghosh, R., et al. "Does the Speed of Gravitational Waves Depend on the Source Velocity?" arXiv preprint arXiv:2304.14820v3 [gr-qc], 2023.

[4] Abbott, R., et al. "Constraints on the Cosmic Expansion History from GWTC-3." *Astrophysical Journal*, vol. 949, no. 11, 2021.

[5] Clavin, W. "LIGO Surpasses the Quantum Limit." *Physical Review X*, 2023.

[6] Reitze, D., et al. "LIGO Congratulates Pulsar Timing Array Teams for New Gravitational Wave Discovery." *LIGO Laboratory News Release*, June 28, 2023.

[7] Ossokine, S., et al. "Multipolar Effective-One-Body Waveforms for Precessing Binary Black Holes: Construction and Validation." *Physical Review D*, vol. 102, 044055, 2020.

[8] Kapadia, S.J., et al. "A Self-Consistent Method to Estimate the Rate of Compact Binary Coalescences with a Poisson Mixture Model." *Classical and Quantum Gravity*, vol. 37, 045007, 2020.

[9] Buikema, A., et al. "Sensitivity and Performance of the Advanced LIGO Detectors in the Third Observing Run." *Physical Review D*, vol. 102, 062003, 2020.

[10] Bertacca, D., et al. "Projection Effects on the Observed Angular Spectrum of the Astrophysical Stochastic Gravitational Wave Background." *Physical Review D*, vol. 101, 103513, 2020.

[11] Nitz, A.H., et al. "2-OGC: Open Gravitational-Wave Catalog of Binary Mergers from Analysis of Public Advanced LIGO and Virgo Data." *Astrophysical Journal*, vol. 891, 123, 2019.

[12] Abbott, B.P., et al. "Prospects for Observing and Localizing Gravitational-Wave Transients with Advanced LIGO, Advanced Virgo, and KAGRA." *Living Reviews in Relativity*, vol. 21, 3, 2018.

[13] Talbot, C., et al. "Measuring the Binary Black Hole Mass Spectrum with an Astrophysically Motivated Parameterization." *Astrophysical Journal*, vol. 856, 173, 2018.

[14] Thrane, E., et al. "Determining the Population Properties of Spinning Black Holes." *Physical Review D*, vol. 96, 023012, 2017.

[15] Vaswani, A., et al. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 5998-6008.

[16] Devlin, J., et al. (2018). BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.

[17] Radford, A., et al. (2019). Language models are unsupervised multitask learners. OpenAI Blog, 1(8), 9.

[18] Brown, T. B., et al. (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33, 1877-1901.

[19] Liu, Y., et al. (2019). RoBERTa: A robustly optimized BERT pretraining approach. arXiv preprint arXiv:1907.11692.

[20] Lan, Z., et al. ALBERT: A lite BERT for self-supervised learning of language representations. arXiv preprint arXiv:1909.11942.

[21] Hinton, G. E., et al. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7), 1527-1554.

[22] Bengio, Y. (2009). Learning deep architectures for AI. *Foundations and Trends® in Machine Learning*, 2(1), 1-127.

[23] Lee, H., Grosse, et al. (2009). Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. *Proceedings of the 26th Annual International Conference on Machine Learning*, 609-616.

[24] Ranzato, M. A., et al. (2010). Factored 3-way restricted Boltzmann machines for modeling natural images. *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*, 621-628.

[25] Salakhutdinov, R., & Hinton, G. (2009). Deep Boltzmann machines. *Artificial Intelligence and Statistics*, 448-455.

[26] Srivastava, N., & Salakhutdinov, R. (2012). Multimodal learning with deep Boltzmann machines. *Advances in Neural Information Processing Systems*, 25.

[27] Larochelle, H., & Bengio, Y. (2008). Classification using discriminative restricted Boltzmann machines. *Proceedings of the 25th International Conference on Machine Learning*, 536-543.

[28] Scarselli, F., et al. (2009). The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1), 61-80.

[29] Kipf, T. N., & Welling, M. (2016). Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907.

[30] Velickovic, P., et al. (2017). Graph attention networks. arXiv preprint arXiv:1710.10903.

[31] Hamilton, W. L., et al. (2017). Inductive representation learning on large graphs. *Advances in Neural Information Processing Systems*, 30.

[32] Xu, K., et al. (2018). How powerful are graph neural networks? arXiv preprint arXiv:1810.00826.

[33] Gilmer, J., et al. (2017). Neural message passing for quantum chemistry. *Proceedings of the 34th International Conference on Machine Learning*, 1263-1272.

[34] Defferrard, M., et al. (2016). Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in Neural Information Processing Systems*, 29, 3844-3852.

[35] Dong, S. (2024). Astrophysical Insights Through Gravitational Wave Data Analysis: Data Preprocessing. viXra preprint viXra:2407.0026.

[36] Dong, S. (2024). Gravitational Wave Event Detection: Developing Convolutional Neural Networks and Recurrent Neural Networks for Gravitational Wave Data Analysis. viXra preprint viXra:2407.0029.