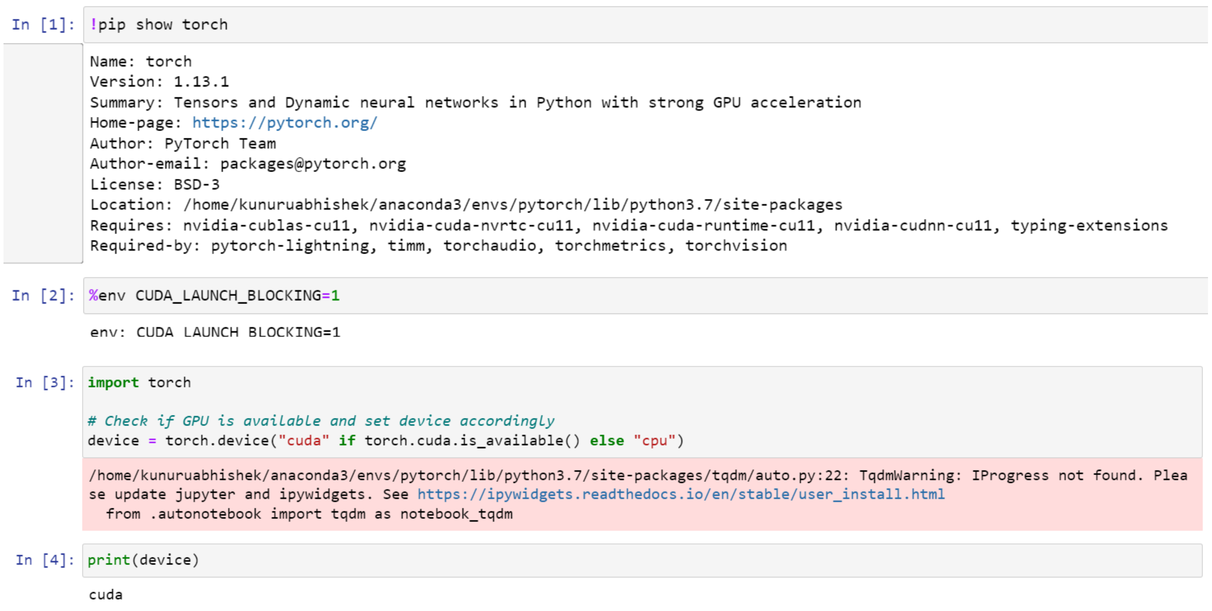
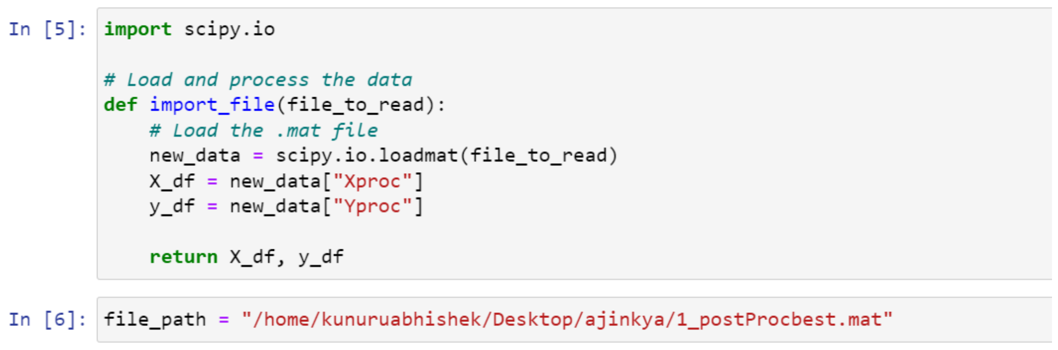


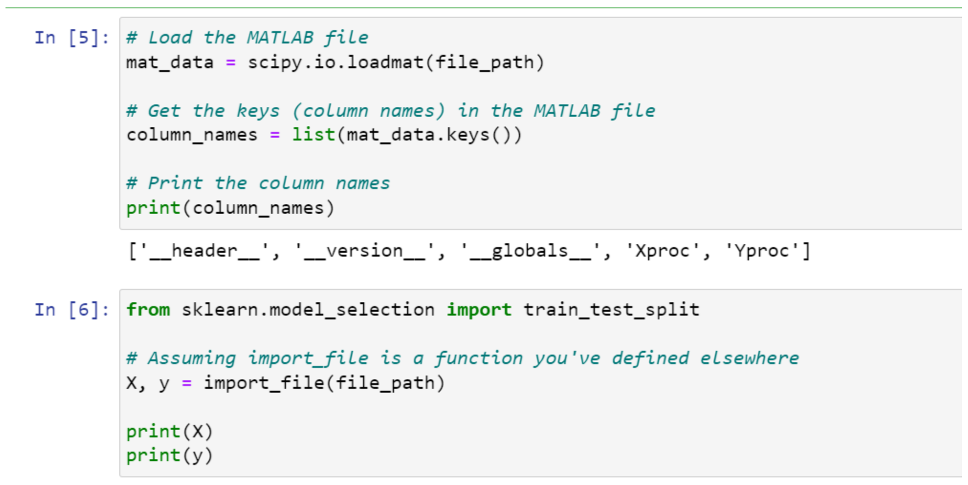
```
In [1]: !pip show torch

Name: torch
Version: 1.13.1
Summary: Tensors and Dynamic neural networks in Python with strong GPU acceleration
Home-page: https://pytorch.org/
Author: PyTorch Team
Author-email: packages@pytorch.org
License: BSD-3
Location: /home/kunuruabhishek/anaconda3/envs/pytorch/lib/python3.7/site-packages
Requires: nvidia-cublas-cu11, nvidia-cuda-nvrtc-cu11, nvidia-cuda-runtime-cu11, nvidia-cudnn-cu11, typing-extensions
Required-by: pytorch-lightning, timm, torchaudio, torchmetrics, torchvision

In [2]: %env CUDA_LAUNCH_BLOCKING=1

env: CUDA_LAUNCH_BLOCKING=1

In [3]: import torch

        # Check if GPU is available and set device accordingly
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

/home/kunuruabhishek/anaconda3/envs/pytorch/lib/python3.7/site-packages/tqdm/auto.py:22: TqdmWarning: IProgress not found. Plea
se update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm

In [4]: print(device)

cuda
```

The code checks if a GPU is available and sets the device accordingly (GPU if available, else CPU).



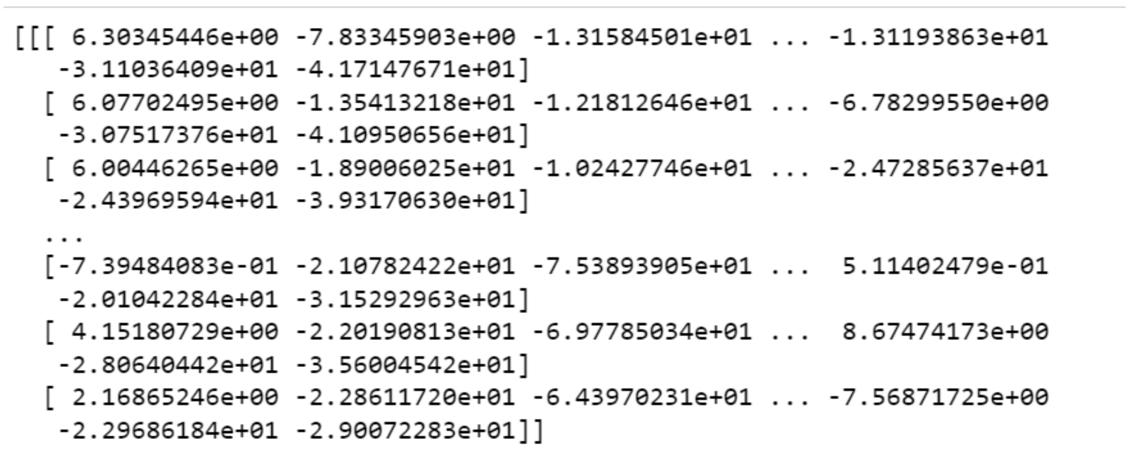
```
In [5]: import scipy.io

        # Load and process the data
        def import_file(file_to_read):
            # Load the .mat file
            new_data = scipy.io.loadmat(file_to_read)
            X_df = new_data["Xproc"]
            y_df = new_data["Yproc"]

            return X_df, y_df

In [6]: file_path = "/home/kunuruabhishek/Desktop/ajinkya/1_postProcbest.mat"
```

It defines a function import_file to load data from a MATLAB file specified by the input path. Retrieves X, y column from it.



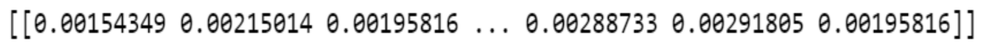
```
In [5]: # Load the MATLAB file
        mat_data = scipy.io.loadmat(file_path)

        # Get the keys (column names) in the MATLAB file
        column_names = list(mat_data.keys())

        # Print the column names
        print(column_names)

        ['__header__', '__version__', '__globals__', 'Xproc', 'Yproc']

In [6]: from sklearn.model_selection import train_test_split

        # Assuming import_file is a function you've defined elsewhere
        X, y = import_file(file_path)

        print(X)
        print(y)
```

Data is loaded using a custom function import_file from the MATLAB file specified by file_path.
It imports the train_test_split function from scikit-learn.
The loaded input data (X) and target data (y) are printed.



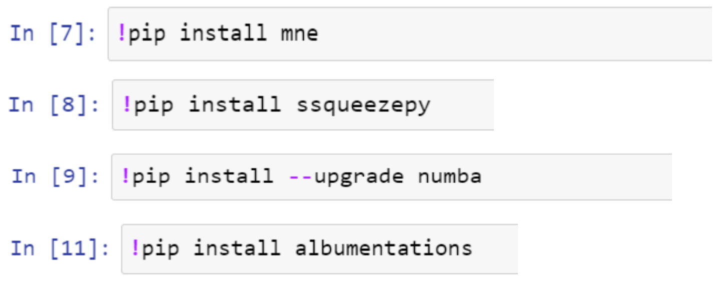
```
[[[ 6.30345446e+00 -7.83345903e+00 -1.31584501e+01 ... -1.31193863e+01
   -3.11036409e+01 -4.17147671e+01]
  [ 6.07702495e+00 -1.35413218e+01 -1.21812646e+01 ... -6.78299550e+00
   -3.07517376e+01 -4.10950656e+01]
  [ 6.00446265e+00 -1.89006025e+01 -1.02427746e+01 ... -2.47285637e+01
   -2.43969594e+01 -3.93170630e+01]
  ...
  [-7.39484083e-01 -2.10782422e+01 -7.53893905e+01 ...  5.11402479e-01
   -2.01042284e+01 -3.15292963e+01]
  [ 4.15180729e+00 -2.20190813e+01 -6.97785034e+01 ...  8.67474173e+00
   -2.80640442e+01 -3.56004542e+01]
  [ 2.16865246e+00 -2.28611720e+01 -6.43970231e+01 ... -7.56871725e+00
   -2.29686184e+01 -2.90072283e+01]]
```

and,



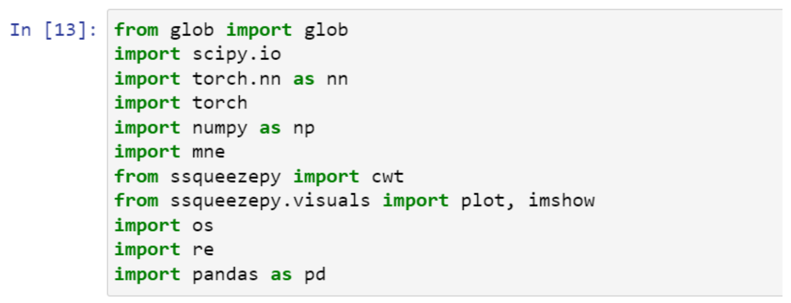
```
[[0.00154349 0.00215014 0.00195816 ... 0.00288733 0.00291805 0.00195816]]
```

These are X and y columns of MATLAB data we used.



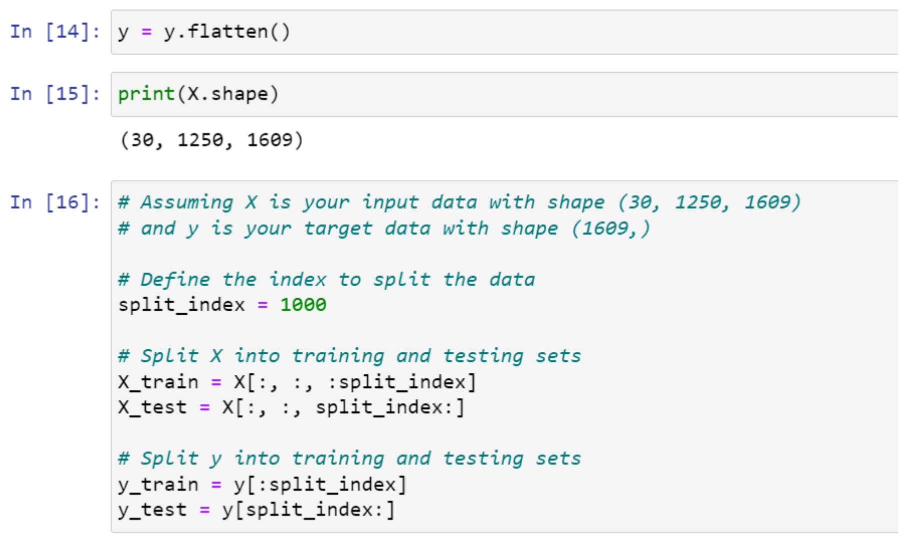
```
In [7]: !pip install mne

In [8]: !pip install ssqueezepy

In [9]: !pip install --upgrade numba

In [11]: !pip install albumentations
```

Numba: Speeds up Python numerical computations using just-in-time compilation.
MNE: Analyses EEG and MEG brain data in Python.
SqueezePy: Implements neural network compression techniques in Python.
Albumnetation: Used for efficient and flexible image augmentation in machine learning tasks to enhance model generalisation and robustness.



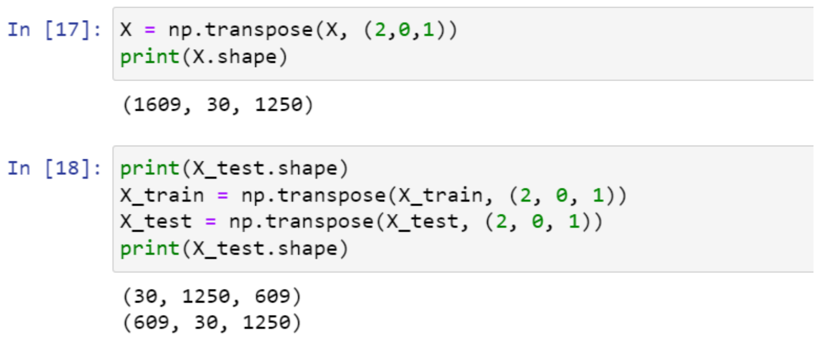
```
In [13]: from glob import glob
         import scipy.io
         import torch.nn as nn
         import torch
         import numpy as np
         import mne
         from ssqueezepy import cwt
         from ssqueezepy.visuals import plot, imshow
         import os
         import re
         import pandas as pd
```

All necessary library imported needed in future.



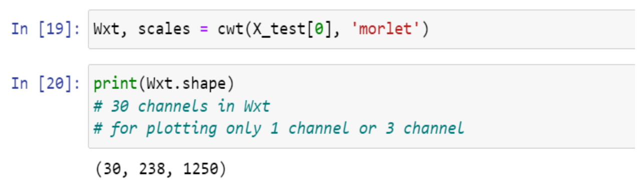
```
In [14]: y = y.flatten()

In [15]: print(X.shape)

         (30, 1250, 1609)

In [16]: # Assuming X is your input data with shape (30, 1250, 1609)
         # and y is your target data with shape (1609,)

         # Define the index to split the data
         split_index = 1000

         # Split X into training and testing sets
         X_train = X[:, :, :split_index]
         X_test = X[:, :, split_index:]

         # Split y into training and testing sets
         y_train = y[:split_index]
         y_test = y[split_index:]
```

Flattens the target data y to convert it into a 1D array.
Dimensions of y : (1x1609) → (1609,)
Shape of X : (30x1250x1609) → 30 channels (locations of brain), 1250 timestamps, 1609 trials.
Using train test split based on trials, 1000 first trials to train and rest 609 are divided between train and test datasets.



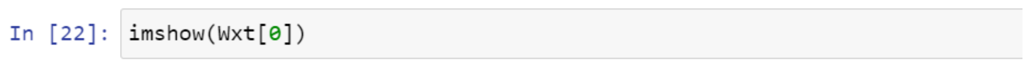
```
In [17]: X = np.transpose(X, (2,0,1))
         print(X.shape)

         (1609, 30, 1250)

In [18]: print(X_test.shape)
         X_train = np.transpose(X_train, (2, 0, 1))
         X_test = np.transpose(X_test, (2, 0, 1))
         print(X_test.shape)

         (30, 1250, 609)
         (609, 30, 1250)
```

Transposes the input data X to reshape it from the original shape (30, 1250, 1609) to (1609, 30, 1250). This reordering of dimensions will be helpful to convert them into images using cwt (where the first dimension represents the number of samples).

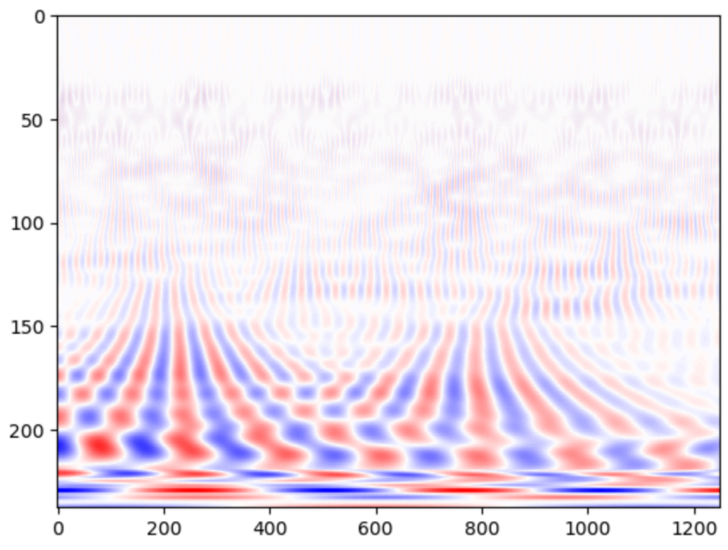


```
In [19]: Wxt, scales = cwt(X_test[0], 'morlet')

In [20]: print(Wxt.shape)
         # 30 channels in Wxt
         # for plotting only 1 channel or 3 channel

         (30, 238, 1250)
```

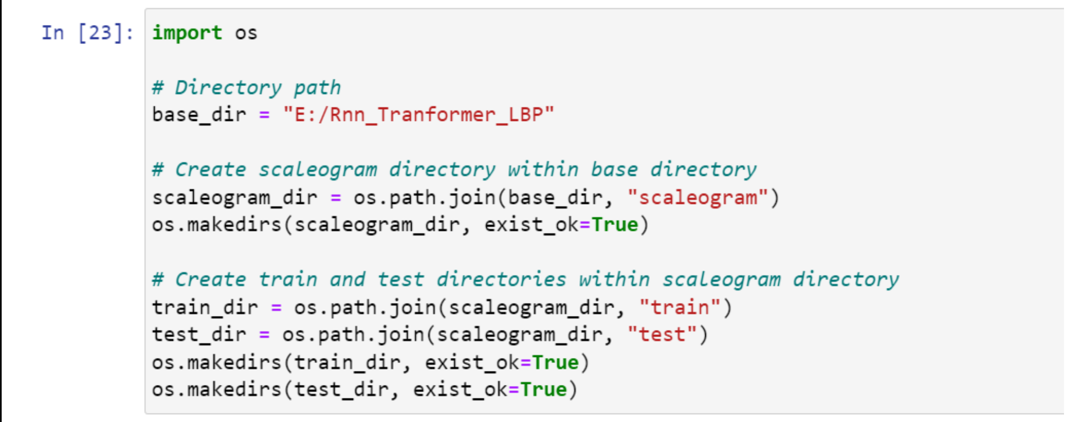
Computes the Continuous Wavelet Transform of the input data X_test[0] using the Morlet wavelet.
The shape typically indicates the number of channels (frequency bands) and the length of the transformed signal.
Wxt[0]) → single channel selected from the transformed data. This would show how the signal varies across different scales for that particular channel.
Wx[:30] → For plotting three channels, compare and visualise the signals across different frequency bands simultaneously for the selected channels.



```
In [22]: imshow(Wxt[0])
```

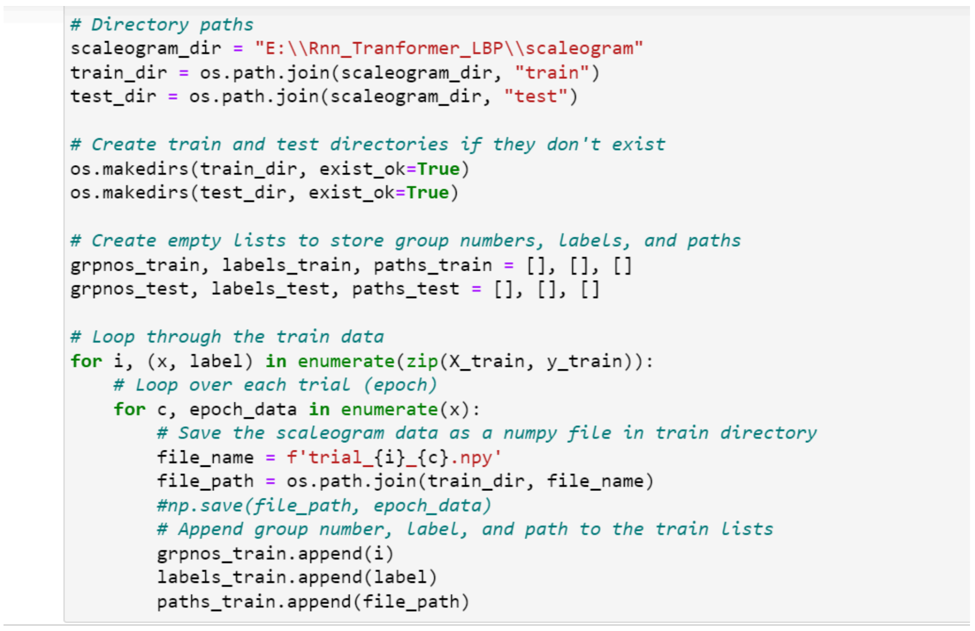


imshow: used to visualise the 2D array as an image where each pixel corresponds to a value in the array. It would provide a spatial representation of the frequency components captured by the wavelet transform.
plot: used to plot 1D images. This would result in a line plot showing how the values of Wxt change with respect to the index. It would provide a temporal or sequential representation of the frequency components captured by the wavelet transform.

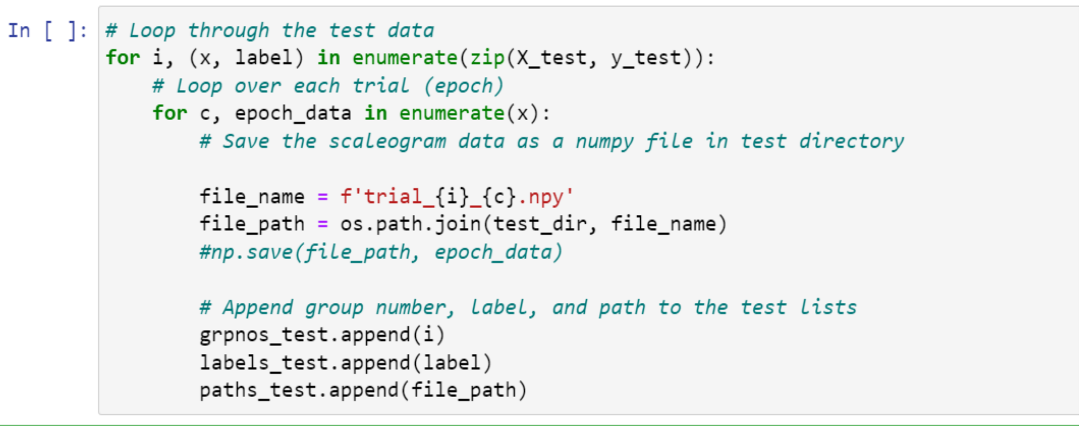


```
In [23]: import os

         # Directory path
         base_dir = "E:/Rnn_Tranformer_LBP"

         # Create scaleogram directory within base directory
         scaleogram_dir = os.path.join(base_dir, "scaleogram")
         os.makedirs(scaleogram_dir, exist_ok=True)

         # Create train and test directories within scaleogram directory
         train_dir = os.path.join(scaleogram_dir, "train")
         test_dir = os.path.join(scaleogram_dir, "test")
         os.makedirs(train_dir, exist_ok=True)
         os.makedirs(test_dir, exist_ok=True)
```

os module, which provides functions for interacting with the operating system.
Created a new directory named scaleogram within the base directory using os.path.join(base_dir, "scaleogram"). The os.makedirs() function is used to create the directory. The exist_ok=True argument ensures that the function does not raise an error if the directory already exists.
Created two subdirectories named train and test within the scaleogram directory. These directories. These directories could be intended for organising data files for training and testing models, respectively.

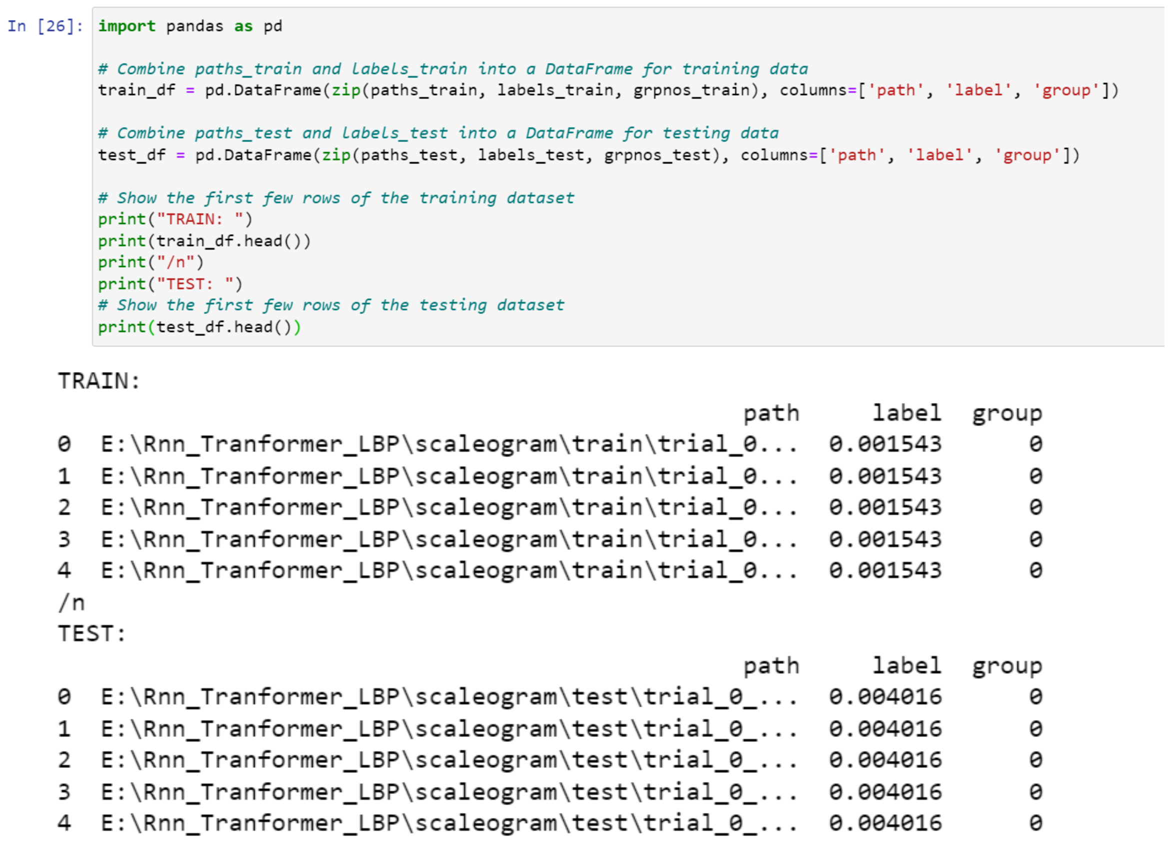


```
# Directory paths
scaleogram_dir = "E:\\Rnn_Tranformer_LBP\\scaleogram"
train_dir = os.path.join(scaleogram_dir, "train")
test_dir = os.path.join(scaleogram_dir, "test")

# Create train and test directories if they don't exist
os.makedirs(train_dir, exist_ok=True)
os.makedirs(test_dir, exist_ok=True)

# Create empty lists to store group numbers, labels, and paths
grpnos_train, labels_train, paths_train = [], [], []
grpnos_test, labels_test, paths_test = [], [], []

# Loop through the train data
for i, (x, label) in enumerate(zip(X_train, y_train)):
    # Loop over each trial (epoch)
    for c, epoch_data in enumerate(x):
        # Save the scaleogram data as a numpy file in train directory
        file_name = f'trial_{i}_{c}.npy'
        file_path = os.path.join(train_dir, file_name)
        #np.save(file_path, epoch_data)
        # Append group number, label, and path to the train lists
        grpnos_train.append(i)
        labels_train.append(label)
        paths_train.append(file_path)
```



```
In [ ]: # Loop through the test data
        for i, (x, label) in enumerate(zip(X_test, y_test)):
            # Loop over each trial (epoch)
            for c, epoch_data in enumerate(x):
                # Save the scaleogram data as a numpy file in test directory

                file_name = f'trial_{i}_{c}.npy'
                file_path = os.path.join(test_dir, file_name)
                #np.save(file_path, epoch_data)

                # Append group number, label, and path to the test lists
                grpnos_test.append(i)
                labels_test.append(label)
                paths_test.append(file_path)
```

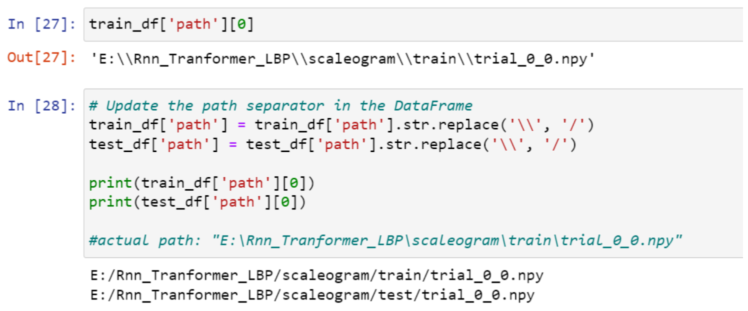
Loops through training data, saving each trial's scaleogram data as a numpy file in the training directory.
Recorded group numbers, labels (output in y), and file paths for each trial in separate lists for further processing.



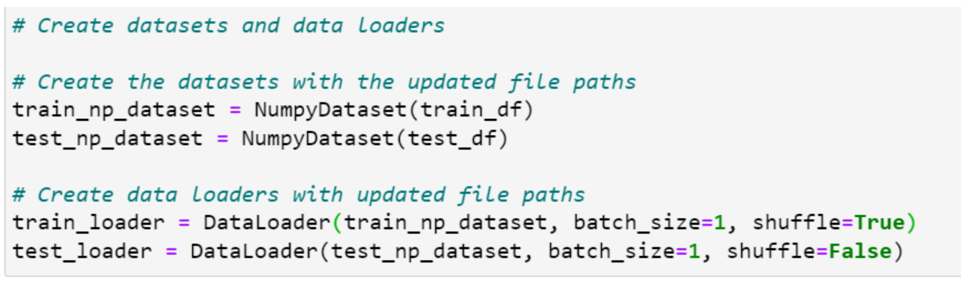
```
In [26]: import pandas as pd

         # Combine paths_train and labels_train into a DataFrame for training data
         train_df = pd.DataFrame(zip(paths_train, labels_train, grpnos_train), columns=['path', 'label', 'group'])

         # Combine paths_test and labels_test into a DataFrame for testing data
         test_df = pd.DataFrame(zip(paths_test, labels_test, grpnos_test), columns=['path', 'label', 'group'])

         # Show the first few rows of the training dataset
         print("TRAIN: ")
         print(train_df.head())
         print("/n")
         print("TEST: ")
         # Show the first few rows of the testing dataset
         print(test_df.head())
```

```
TRAIN:
                                                path     label  group
0  E:\Rnn_Tranformer_LBP\scaleogram\train\trial_0...  0.001543      0
1  E:\Rnn_Tranformer_LBP\scaleogram\train\trial_0...  0.001543      0
2  E:\Rnn_Tranformer_LBP\scaleogram\train\trial_0...  0.001543      0
3  E:\Rnn_Tranformer_LBP\scaleogram\train\trial_0...  0.001543      0
4  E:\Rnn_Tranformer_LBP\scaleogram\train\trial_0...  0.001543      0
/n
TEST:
                                                path     label  group
0  E:\Rnn_Tranformer_LBP\scaleogram\test\trial_0_...  0.004016      0
1  E:\Rnn_Tranformer_LBP\scaleogram\test\trial_0_...  0.004016      0
2  E:\Rnn_Tranformer_LBP\scaleogram\test\trial_0_...  0.004016      0
3  E:\Rnn_Tranformer_LBP\scaleogram\test\trial_0_...  0.004016      0
4  E:\Rnn_Tranformer_LBP\scaleogram\test\trial_0_...  0.004016      0
```

This code creates two Pandas DataFrames, train_df and test_df, to organise the file paths, labels, and group numbers for training and testing data, respectively. Each DataFrame has three columns: 'path' (file path), 'label' (target label), and 'group' (group number).

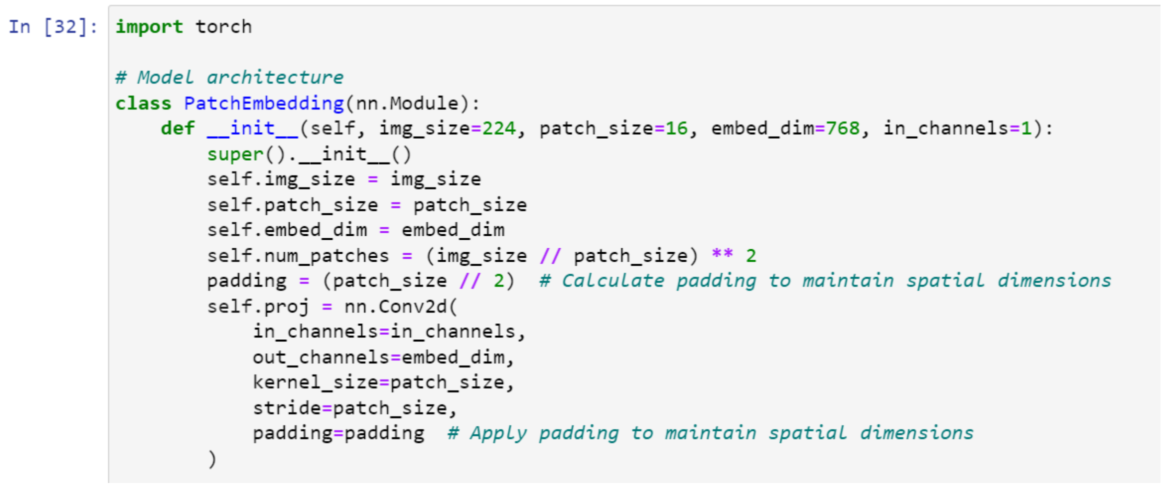


```
In [27]: train_df['path'][0]

Out[27]: 'E:\\Rnn_Tranformer_LBP\\scaleogram\\train\\trial_0_0.npy'

In [28]: # Update the path separator in the DataFrame
         train_df['path'] = train_df['path'].str.replace('\\', '/')
         test_df['path'] = test_df['path'].str.replace('\\', '/')

         print(train_df['path'][0])
         print(test_df['path'][0])

         #actual path: "E:\Rnn_Tranformer_LBP\scaleogram\train\trial_0_0.npy"

         E:/Rnn_Tranformer_LBP/scaleogram/train/trial_0_0.npy
         E:/Rnn_Tranformer_LBP/scaleogram/test/trial_0_0.npy
```



```
# Create datasets and data loaders

# Create the datasets with the updated file paths
train_np_dataset = NumpyDataset(train_df)
test_np_dataset = NumpyDataset(test_df)

# Create data loaders with updated file paths
train_loader = DataLoader(train_np_dataset, batch_size=1, shuffle=True)
test_loader = DataLoader(test_np_dataset, batch_size=1, shuffle=False)
```

The code creates datasets (train_np_dataset and test_np_dataset) using the NumpyDataset class initialised with updated file paths. It then constructs data loaders (train_loader and test_loader) with specified batch sizes for efficient data processing during training and testing phases, facilitating seamless integration of numpy data files into PyTorch workflows.



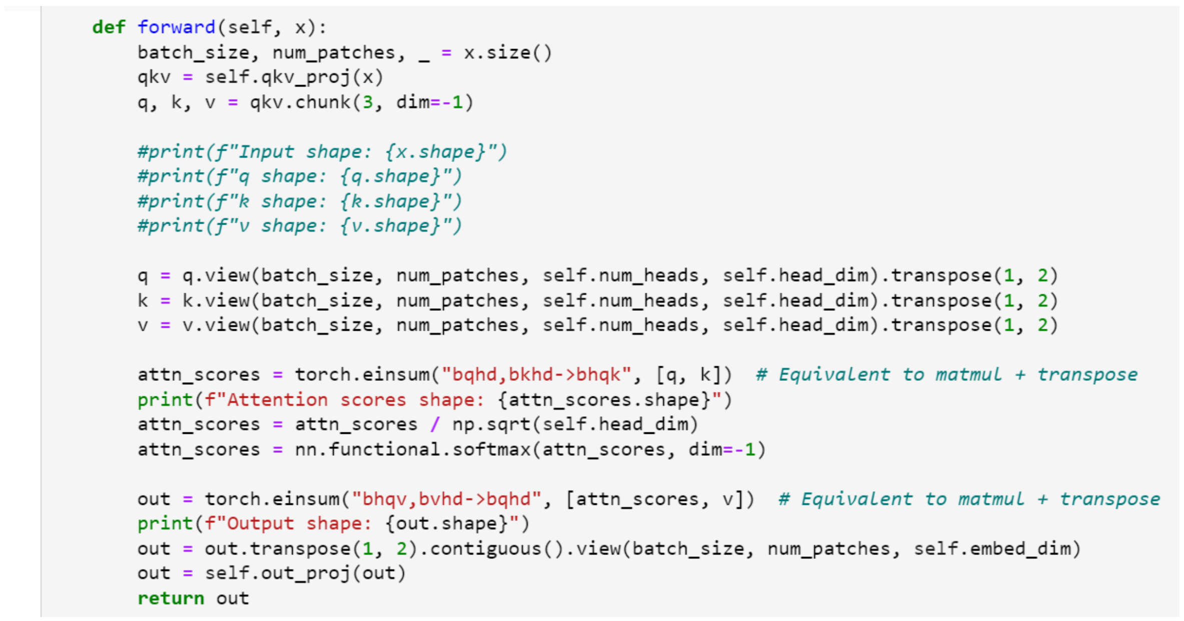
```
In [32]: import torch

         # Model architecture
         class PatchEmbedding(nn.Module):
             def __init__(self, img_size=224, patch_size=16, embed_dim=768, in_channels=1):
                 super().__init__()
                 self.img_size = img_size
                 self.patch_size = patch_size
                 self.embed_dim = embed_dim
                 self.num_patches = (img_size // patch_size) ** 2
                 padding = (patch_size // 2)  # Calculate padding to maintain spatial dimensions
                 self.proj = nn.Conv2d(
                     in_channels=in_channels,
                     out_channels=embed_dim,
                     kernel_size=patch_size,
                     stride=patch_size,
                     padding=padding  # Apply padding to maintain spatial dimensions
                 )
```

```python
    def forward(self, x):
        # If the input tensor is 2D (height, width), add a channel dimension
        if x.ndim == 2:
            x = x.unsqueeze(0).unsqueeze(0)
        # If the input tensor is 3D (channels, height, width), add a batch dimension
        elif x.ndim == 3:
            x = x.unsqueeze(0)

        # Reshape the input tensor to have 4 dimensions (batch_size, channels, height, width)
        x = x.float()  # Convert the input tensor to float
        x = self.proj(x)  # (batch_size, embed_dim, num_patches, 1, 1)
        x = x.flatten(2)  # (batch_size, embed_dim, num_patches)
        x = x.transpose(1, 2)  # (batch_size, num_patches, embed_dim)
        return x
```

This code defines a **PatchEmbedding** module for converting input image data into a sequence of patches.

- **Initialization**: The __init__ method initialises the module with parameters such as image size (img_size), patch size (patch_size), embedding dimension (embed_dim), and input channels (in_channels). It calculates the number of patches based on the image and patch sizes.
- **Convolutional Projection**: The module uses a 2D convolutional layer (nn.Conv2d) to project the input image into a sequence of patches. The convolutional layer has parameters such as input channels, output channels (embedding dimension), kernel size (patch size), stride (patch size), and padding (to maintain spatial dimensions).
- **Forward** Method: The forward method processes the input tensor (x) through the PatchEmbedding module. It first ensures that the input tensor has the correct number of dimensions (4 dimensions for batch processing). Then, it applies the convolutional projection (self.proj) to the input tensor to obtain a sequence of patch embeddings. Finally, it reshapes and transposes the output tensor to match the expected shape (batch_size, num_patches, embed_dim) for further processing.

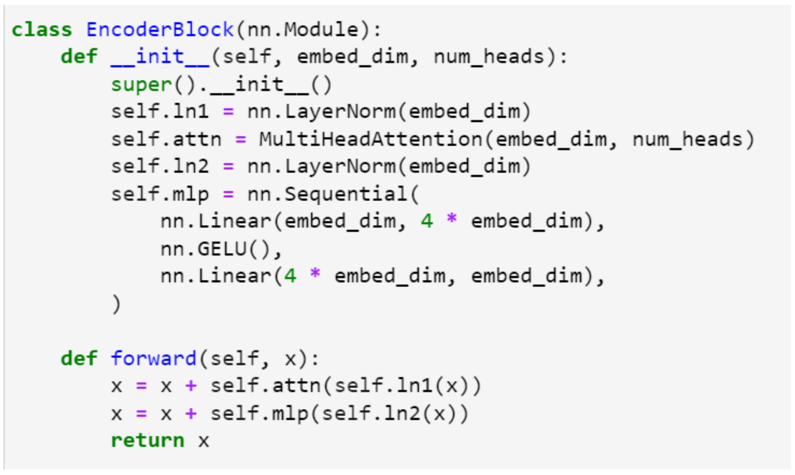
```python
class MultiHeadAttention(nn.Module):
    def __init__(self, embed_dim, num_heads):
        super().__init__()
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads

        self.qkv_proj = nn.Linear(embed_dim, 3 * embed_dim)
        self.out_proj = nn.Linear(embed_dim, embed_dim)
```



```python
    def forward(self, x):
        batch_size, num_patches, _ = x.size()
        qkv = self.qkv_proj(x)
        q, k, v = qkv.chunk(3, dim=-1)

        #print(f"Input shape: {x.shape}")
        #print(f"q shape: {q.shape}")
        #print(f"k shape: {k.shape}")
        #print(f"v shape: {v.shape}")

        q = q.view(batch_size, num_patches, self.num_heads, self.head_dim).transpose(1, 2)
        k = k.view(batch_size, num_patches, self.num_heads, self.head_dim).transpose(1, 2)
        v = v.view(batch_size, num_patches, self.num_heads, self.head_dim).transpose(1, 2)

        attn_scores = torch.einsum("bqhd,bkhd->bhqk", [q, k])  # Equivalent to matmul + transpose
        print(f"Attention scores shape: {attn_scores.shape}")
        attn_scores = attn_scores / np.sqrt(self.head_dim)
        attn_scores = nn.functional.softmax(attn_scores, dim=-1)

        out = torch.einsum("bhqv,bvhd->bqhd", [attn_scores, v])  # Equivalent to matmul + transpose
        print(f"Output shape: {out.shape}")
        out = out.transpose(1, 2).contiguous().view(batch_size, num_patches, self.embed_dim)
        out = self.out_proj(out)
        return out
```

This code defines a Multi-Head Attention module, a key component in transformer-based architectures for handling sequence data. Here's a summary:

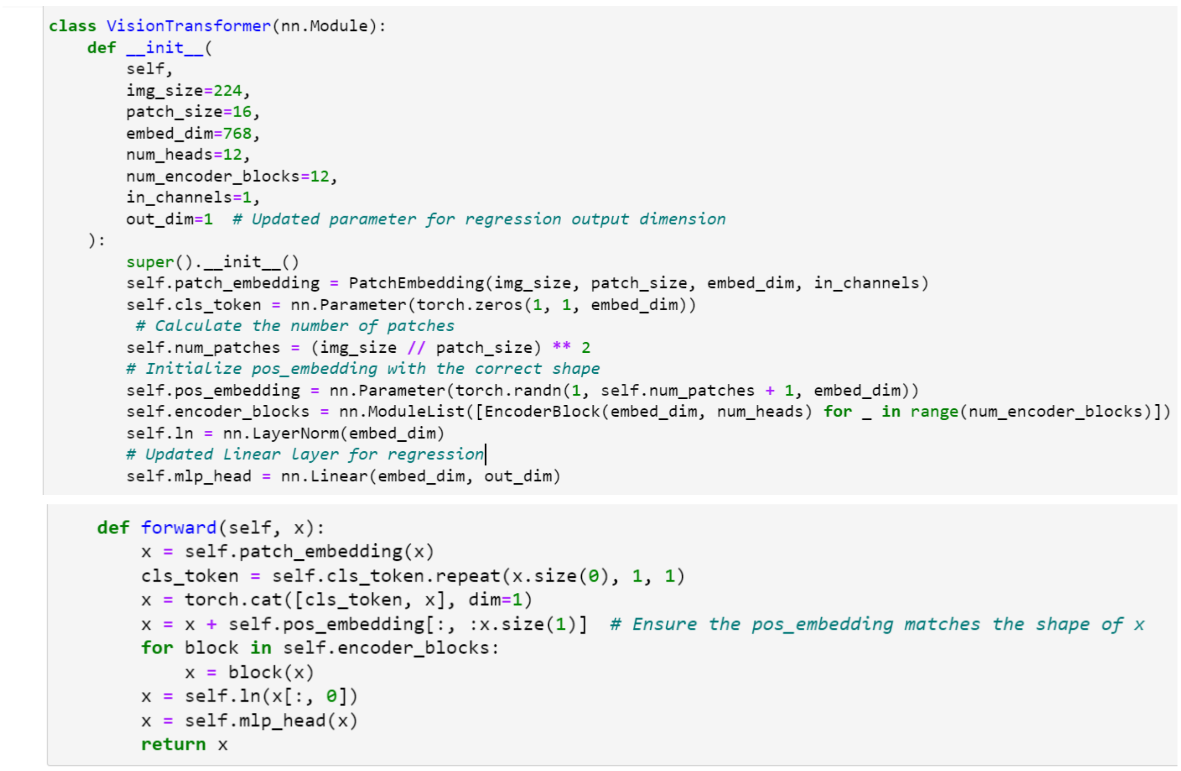
- **Initialization**: Initialises the module with parameters embed_dim (embedding dimension) and num_heads (number of attention heads).
  - Calculates head_dim as the dimensionality of each attention head.
  - Sets up linear projections for queries, keys, and values using nn.Linear.
- **Forward** Method:
  - Computes the forward pass of the attention mechanism.
  - Projects the input tensor x to obtain query (q), key (k), and value (v) tensors using self.qkv_proj.
  - Reshapes and transposes these tensors to facilitate efficient computation of attention scores.
  - Calculates attention scores between queries and keys using the dot product, followed by scaling and softmax normalisation.
  - Computes the weighted sum of values based on attention scores to obtain the output tensor.
  - Transposes and reshapes the output tensor to match the expected shape, and passes it through a linear layer (self.out_proj) to produce the final output of the attention module.
- **Print Statements**:
  - These are added for debugging purposes to print the shapes of intermediate tensors (attn_scores and out) during the forward pass. This helps in understanding the dimensions of tensors at each step of the computation.



```python
class EncoderBlock(nn.Module):
    def __init__(self, embed_dim, num_heads):
        super().__init__()
        self.ln1 = nn.LayerNorm(embed_dim)
        self.attn = MultiHeadAttention(embed_dim, num_heads)
        self.ln2 = nn.LayerNorm(embed_dim)
        self.mlp = nn.Sequential(
            nn.Linear(embed_dim, 4 * embed_dim),
            nn.GELU(),
            nn.Linear(4 * embed_dim, embed_dim),
        )

    def forward(self, x):
        x = x + self.attn(self.ln1(x))
        x = x + self.mlp(self.ln2(x))
        return x
```

This code defines an EncoderBlock module, which is a component of a transformer-based neural network architecture. Here's a summary:

- **Initialization**:
  - Initialises the module with parameters embed_dim (embedding dimension) and num_heads (number of attention heads).
  - Sets up layer normalisation (nn.LayerNorm) for normalisation before and after the multi-head attention mechanism (self.ln1 and self.ln2).
  - Instantiates a multi-head attention module (self.attn) and a multi-layer perceptron (MLP) submodule (self.mlp) for feed-forward processing.
- **Forward** Method:
  - Computes the forward pass of the encoder block.
  - Applies layer normalisation to the input tensor x before passing it through the multi-head attention mechanism (self.attn). The output of the attention mechanism is added to the original input tensor (x) to incorporate the attention information.
  - Applies layer normalisation again to the result and passes it through the MLP submodule (self.mlp). The output of the MLP is added to the result of the previous step, producing the final output tensor of the encoder block.
  - The residual connections (addition of input and output tensors) and layer normalisation ensure stable and effective information flow through the encoder block, facilitating efficient learning of representations from input data.

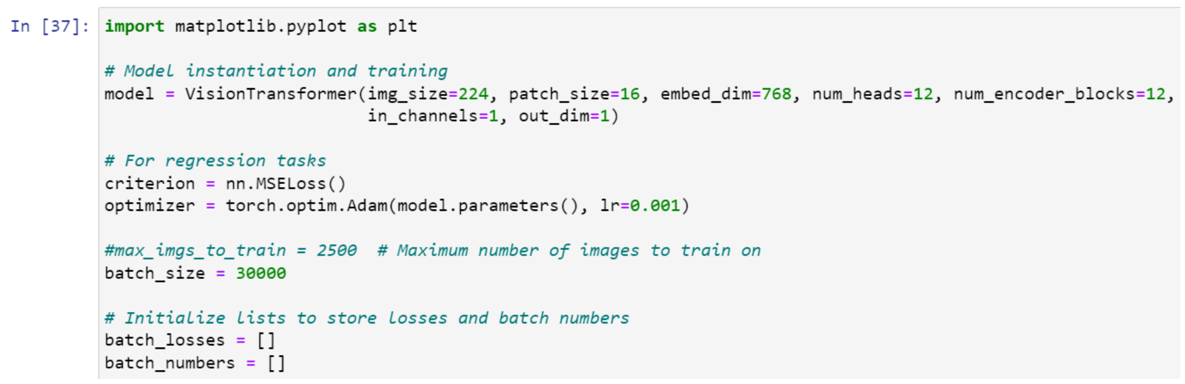


```python
class VisionTransformer(nn.Module):
    def __init__(
        self,
        img_size=224,
        patch_size=16,
        embed_dim=768,
        num_heads=12,
        num_encoder_blocks=12,
        in_channels=1,
        out_dim=1  # Updated parameter for regression output dimension
    ):
        super().__init__()
        self.patch_embedding = PatchEmbedding(img_size, patch_size, embed_dim, in_channels)
        self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))
         # Calculate the number of patches
        self.num_patches = (img_size // patch_size) ** 2
        # Initialize pos_embedding with the correct shape
        self.pos_embedding = nn.Parameter(torch.randn(1, self.num_patches + 1, embed_dim))
        self.encoder_blocks = nn.ModuleList([EncoderBlock(embed_dim, num_heads) for _ in range(num_encoder_blocks)])
        self.ln = nn.LayerNorm(embed_dim)
        # Updated Linear layer for regression
        self.mlp_head = nn.Linear(embed_dim, out_dim)
```

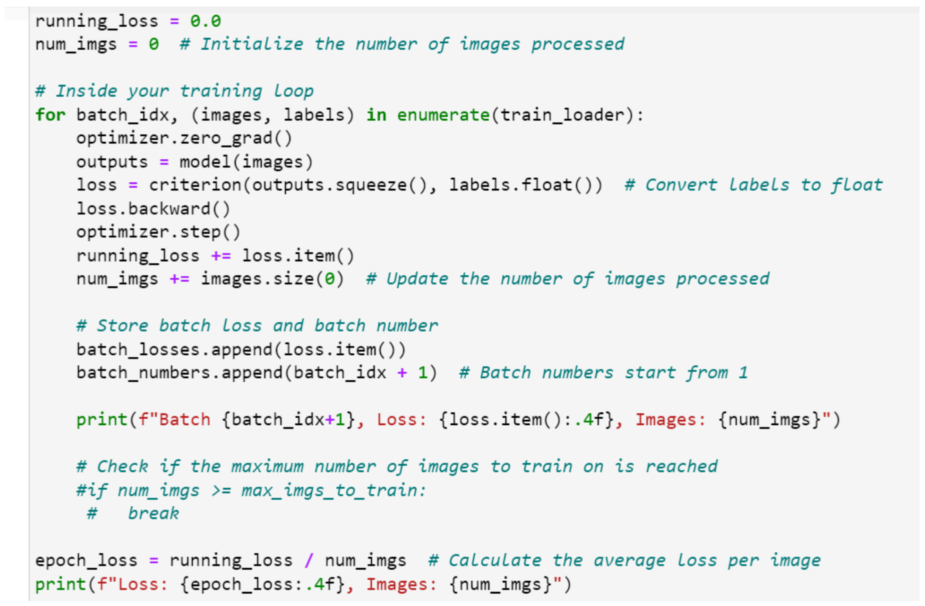
```python
    def forward(self, x):
        x = self.patch_embedding(x)
        cls_token = self.cls_token.repeat(x.size(0), 1, 1)
        x = torch.cat([cls_token, x], dim=1)
        x = x + self.pos_embedding[:, :x.size(1)]  # Ensure the pos_embedding matches the shape of x
        for block in self.encoder_blocks:
            x = block(x)
        x = self.ln(x[:, 0])
        x = self.mlp_head(x)
        return x
```

This VisionTransformer class implements a transformer-based model tailored for regression tasks on image data. Here's a succinct breakdown:

- **Patch Embedding**: Breaks down input images into patches, embedding them into a lower-dimensional space (embed_dim). This enables efficient processing of image data by the subsequent transformer layers.
- **Class Token & Position Embedding**: Introduces a learnable class token and positional embeddings. The class token serves as a global representation of the entire image, while positional embeddings preserve spatial information crucial for regression tasks.
- **Multi-layer Transformer Encoder**: Employs multiple transformer encoder blocks to iteratively refine features extracted from the input patches. Each encoder block enhances the model's ability to capture hierarchical patterns and long-range dependencies in the image data.
- **Linear Regression Head**: Concludes the model with a linear layer tailored for regression output. This layer adapts the high-dimensional features learned by the transformer encoder into a single prediction corresponding to the regression task.

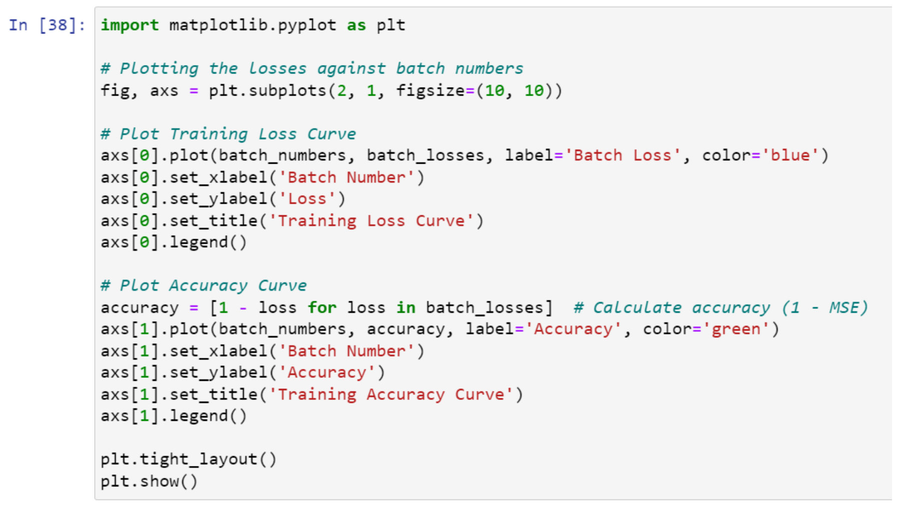


```
In [37]: import matplotlib.pyplot as plt

         # Model instantiation and training
         model = VisionTransformer(img_size=224, patch_size=16, embed_dim=768, num_heads=12, num_encoder_blocks=12,
                                   in_channels=1, out_dim=1)

         # For regression tasks
         criterion = nn.MSELoss()
         optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

         #max_imgs_to_train = 2500  # Maximum number of images to train on
         batch_size = 30000

         # Initialize lists to store losses and batch numbers
         batch_losses = []
         batch_numbers = []
```

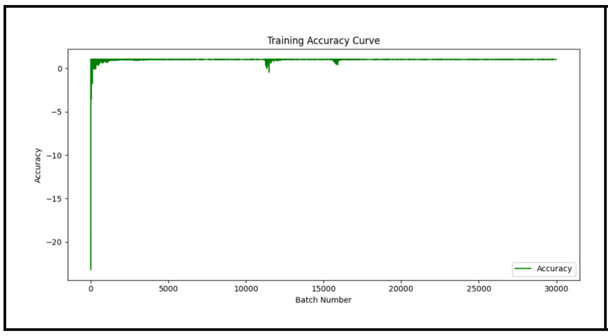


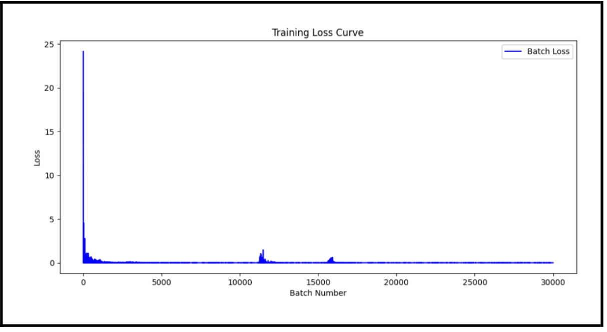
```
running_loss = 0.0
num_imgs = 0  # Initialize the number of images processed

# Inside your training loop
for batch_idx, (images, labels) in enumerate(train_loader):
    optimizer.zero_grad()
    outputs = model(images)
    loss = criterion(outputs.squeeze(), labels.float())  # Convert labels to float
    loss.backward()
    optimizer.step()
    running_loss += loss.item()
    num_imgs += images.size(0)  # Update the number of images processed

    # Store batch loss and batch number
    batch_losses.append(loss.item())
    batch_numbers.append(batch_idx + 1)  # Batch numbers start from 1

    print(f"Batch {batch_idx+1}, Loss: {loss.item():.4f}, Images: {num_imgs}")

    # Check if the maximum number of images to train on is reached
    #if num_imgs >= max_imgs_to_train:
     #   break

epoch_loss = running_loss / num_imgs  # Calculate the average loss per image
print(f"Loss: {epoch_loss:.4f}, Images: {num_imgs}")
```

- **Model Instantiation**:
  - Initialises a VisionTransformer model with specified parameters for image size, patch size, embedding dimension, number of heads, number of encoder blocks, input channels, and output dimension (for regression tasks).
- **Loss Function and Optimizer**:
  - Sets up a cross-entropy loss criterion for calculating the loss between model predictions and target labels.
  - Utilises the Adam optimizer to update model parameters during training, with a learning rate of 0.001.
- **Training Loop**:
  - Iterates over each epoch in the specified number of epochs (num_epochs).
  - Resets the running loss to zero at the beginning of each epoch.
  - Within each epoch, iterates over batches of data from the train_loader.
  - Performs forward pass through the model to obtain predictions (outputs), computes the loss between predictions and ground truth labels, and updates model parameters using backpropagation and optimizer steps.
  - Accumulates the loss for each batch to compute the average loss per epoch.

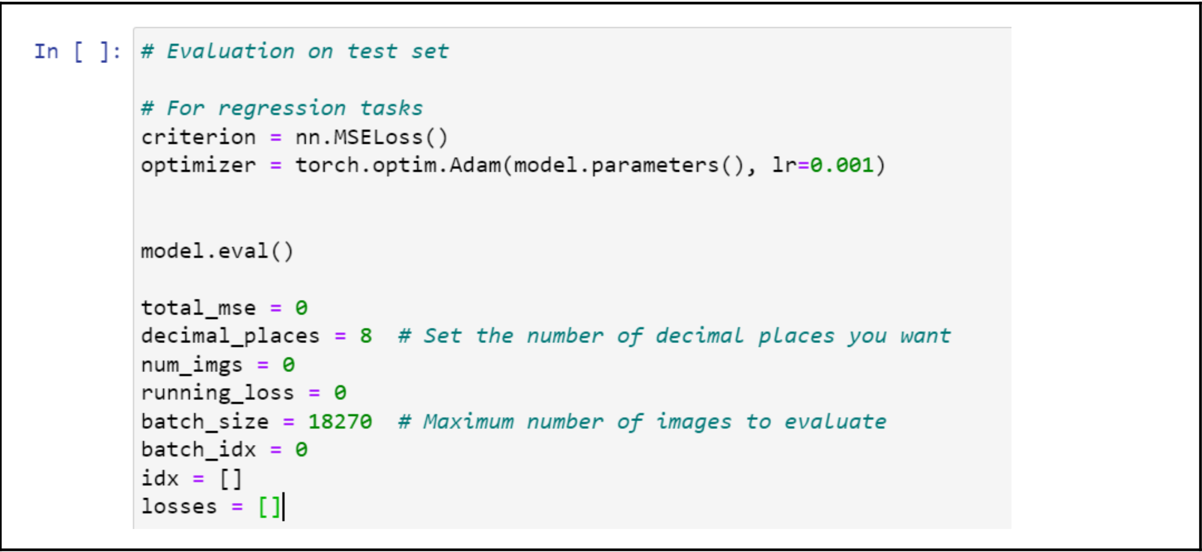


```
In [38]: import matplotlib.pyplot as plt

         # Plotting the losses against batch numbers
         fig, axs = plt.subplots(2, 1, figsize=(10, 10))

         # Plot Training Loss Curve
         axs[0].plot(batch_numbers, batch_losses, label='Batch Loss', color='blue')
         axs[0].set_xlabel('Batch Number')
         axs[0].set_ylabel('Loss')
         axs[0].set_title('Training Loss Curve')
         axs[0].legend()

         # Plot Accuracy Curve
         accuracy = [1 - loss for loss in batch_losses]  # Calculate accuracy (1 - MSE)
         axs[1].plot(batch_numbers, accuracy, label='Accuracy', color='green')
         axs[1].set_xlabel('Batch Number')
         axs[1].set_ylabel('Accuracy')
         axs[1].set_title('Training Accuracy Curve')
         axs[1].legend()

         plt.tight_layout()
         plt.show()
```

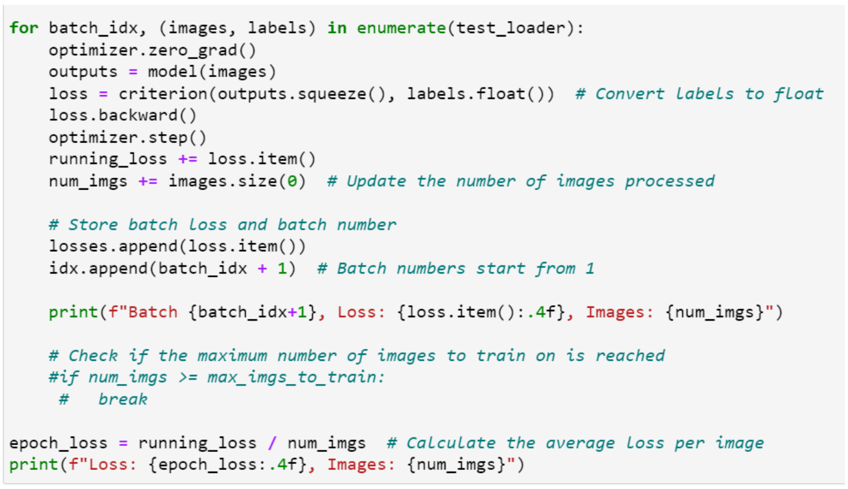




The code snippet above plots the accuracy and loss curve during training of ViT.
Loss = 0.0097 and Accuracy = 1 -Loss = 99.03%

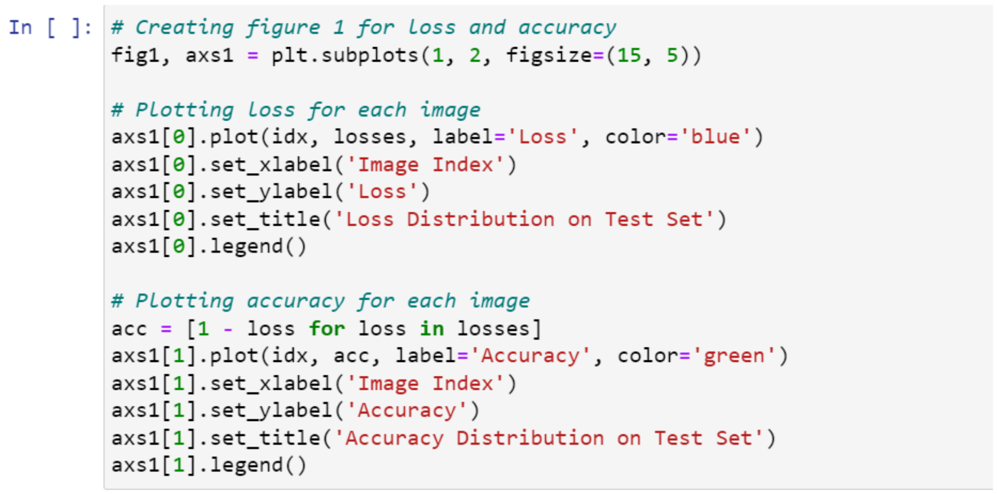


```
In [ ]: # Evaluation on test set

        # For regression tasks
        criterion = nn.MSELoss()
        optimizer = torch.optim.Adam(model.parameters(), lr=0.001)


        model.eval()

        total_mse = 0
        decimal_places = 8  # Set the number of decimal places you want
        num_imgs = 0
        running_loss = 0
        batch_size = 18270  # Maximum number of images to evaluate
        batch_idx = 0
        idx = []
        losses = []
```



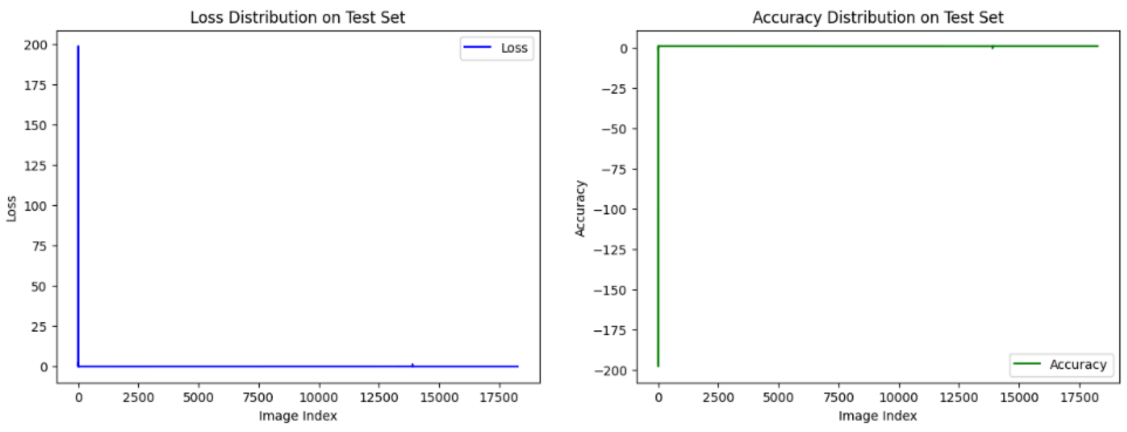
```
for batch_idx, (images, labels) in enumerate(test_loader):
    optimizer.zero_grad()
    outputs = model(images)
    loss = criterion(outputs.squeeze(), labels.float())  # Convert labels to float
    loss.backward()
    optimizer.step()
    running_loss += loss.item()
    num_imgs += images.size(0)  # Update the number of images processed

    # Store batch loss and batch number
    losses.append(loss.item())
    idx.append(batch_idx + 1)  # Batch numbers start from 1

    print(f"Batch {batch_idx+1}, Loss: {loss.item():.4f}, Images: {num_imgs}")

    # Check if the maximum number of images to train on is reached
    #if num_imgs >= max_imgs_to_train:
     #   break

epoch_loss = running_loss / num_imgs  # Calculate the average loss per image
print(f"Loss: {epoch_loss:.4f}, Images: {num_imgs}")
```

This code segment evaluates the trained VisionTransformer model on the test set by computing the Mean Squared Error (MSE) between model predictions and ground truth labels. It accumulates MSE values across batches and calculates the average MSE as a measure of regression performance, providing a concise assessment of the model's accuracy on unseen data.



```
In [ ]: # Creating figure 1 for loss and accuracy
        fig1, axs1 = plt.subplots(1, 2, figsize=(15, 5))

        # Plotting loss for each image
        axs1[0].plot(idx, losses, label='Loss', color='blue')
        axs1[0].set_xlabel('Image Index')
        axs1[0].set_ylabel('Loss')
        axs1[0].set_title('Loss Distribution on Test Set')
        axs1[0].legend()

        # Plotting accuracy for each image
        acc = [1 - loss for loss in losses]
        axs1[1].plot(idx, acc, label='Accuracy', color='green')
        axs1[1].set_xlabel('Image Index')
        axs1[1].set_ylabel('Accuracy')
        axs1[1].set_title('Accuracy Distribution on Test Set')
        axs1[1].legend()
```

This code snippet plots Loss on test dataset over validating and Also plots actual and predicted reaction times.
Loss = 0.017 and Accuracy = 1 -Loss = 98.73%



The plot shows the test dataset loss and accuracy respectively.