Ahmed Awadallah

Dense Neural Networks

Introduction

The background for this case study is quite brief so it’s hard to write an introduction for it. The vague premise is that we have a dataset with attributes that are linked to either particle A or B and we want to make a model (specifically neural network) that can take these attributes and accurately predict the right particle. One of these particles is a new particle so there is some interest in detecting this particle based on the given attributes.

This dataset contains 7 million entries with a total of 28 features and one binary target.

Methods

The dataset is a single file so it is simply loaded into the program.

N/A values did not exist within the dataset so no imputation was performed.

Inspection of the data reveals that all the features except for the target are numeric..

Numeric variables were scaled as the model used is sensitive to data scale.

The prepared data is now split into training, validation, and test set. The split were respectively 98%, 1% and 1%. These splits were chosen as the dataset is immense in size so relatively small validation/test sets still contain enough data to properly analyze the performance of the model. This allows us to use more of the data for training and hence provide a benefit to performance.

The training and validation set were loaded into a data loader as they will be actively used within the model training process. The test set remained as is since it is only involved with sklearn functionalities.

As an aside, technically I don’t fully utilize the value of a data loader as the data is actually actively stored in RAM instead of being pulled from persistent storage. Still I got familiar with working with them and will keep this critical step in mind for a future study.

Architecture

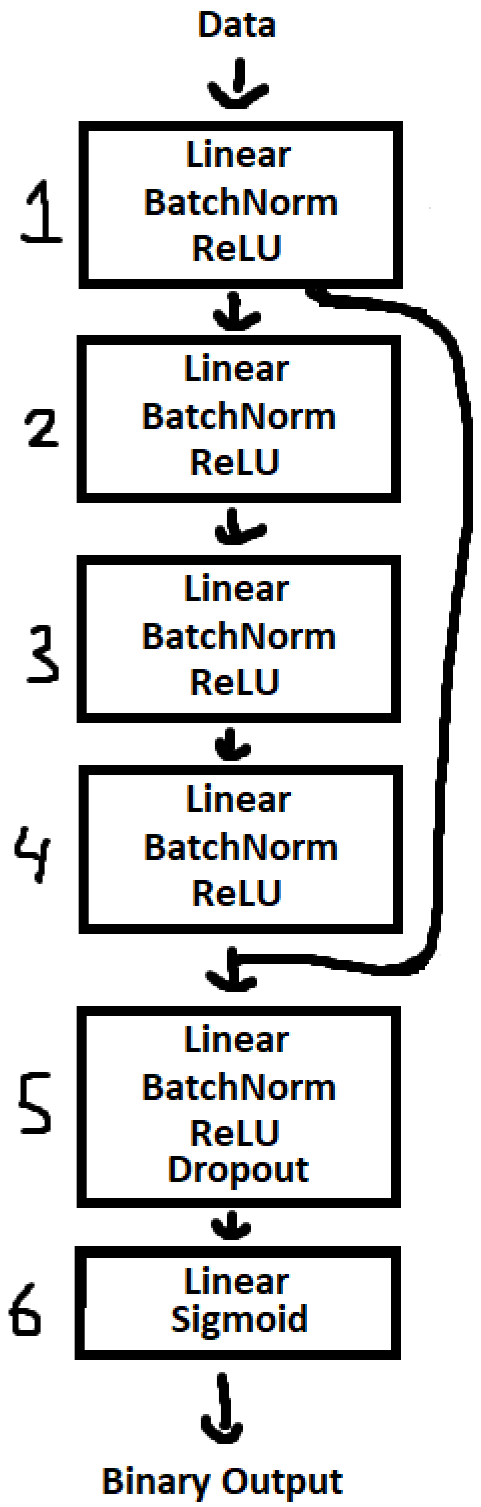


Figure 1: Neural Network Architecture

Model in summary is composed of 6 layers. Overall the model always reduces in size to take advantage of the generalizing effect of reducing the number of weights in a model prior to prediction.

The first four layers follow the same procedure of a dense layer into batch normalization into ReLU activation. The output of each layer reduces the total amount of weights by a factor of 4. For example layer 1 starts with 2048 weights but outputs 512. The dense layer is required for learning in the model. The batch normalization is not required but it helps the model train faster. The ReLU function is not required but an activation function in general is required to introduce nonlinearity into the network. ReLU was chosen as it is generally used as the default choice for hidden layer activations.

The fifth layer acts as a skip connection and regularization layer. The input is the size of the output of layer 4 plus layer 1. Layer 1 has a size of 512 and layer 4 has a size of 8 so the input into layer 5 is 40. A skip connection did not increase the models performance so it was not necessary to keep in the model but was largely kept as a reference guide for the future. The regularization impact of layer 5 is achieved through dropout. Usually the value for dropouts range from 20% to 50% for my model I ended up using 20%. Lastly the layer outputs the same amount of weights as layer 4, 8.

The final layer is the output layer(layer 6), it uses a dense layer with a sigmoid activation. The problem is binary classification, this type of problem is best solved by a final weight count of 1 with sigmoid activation. Technically speaking it could be a softmax with a weight count of 2 but in practice the former is used.

External to the model we used Binary Cross Entropy Loss, Adam optimizer, and Early Stopping.

Binary cross entropy was used as this is a binary problem. The network was set up following this expectation and this loss function follows as a requirement to properly train the model.

Adam optimizer was used as it is a standard optimizer for neural network problems. Default hyper parameters were used for this optimizer as they perform well in general.

Early stopping is accomplished by checking the validation score after a couple training iterations, if the validation score does not drop after N validation score checks the training loop exits and the final model is returned as is. This technically also works as a generalizing tool as it prevents the model from training when the validation score is barely improving. The value chosen for this was 50 iterations of a value larger than the best recorded validation loss. A large number was chosen to squeeze out as much performance as possible before it becomes largely negligible.

Results

Train Loss Plot

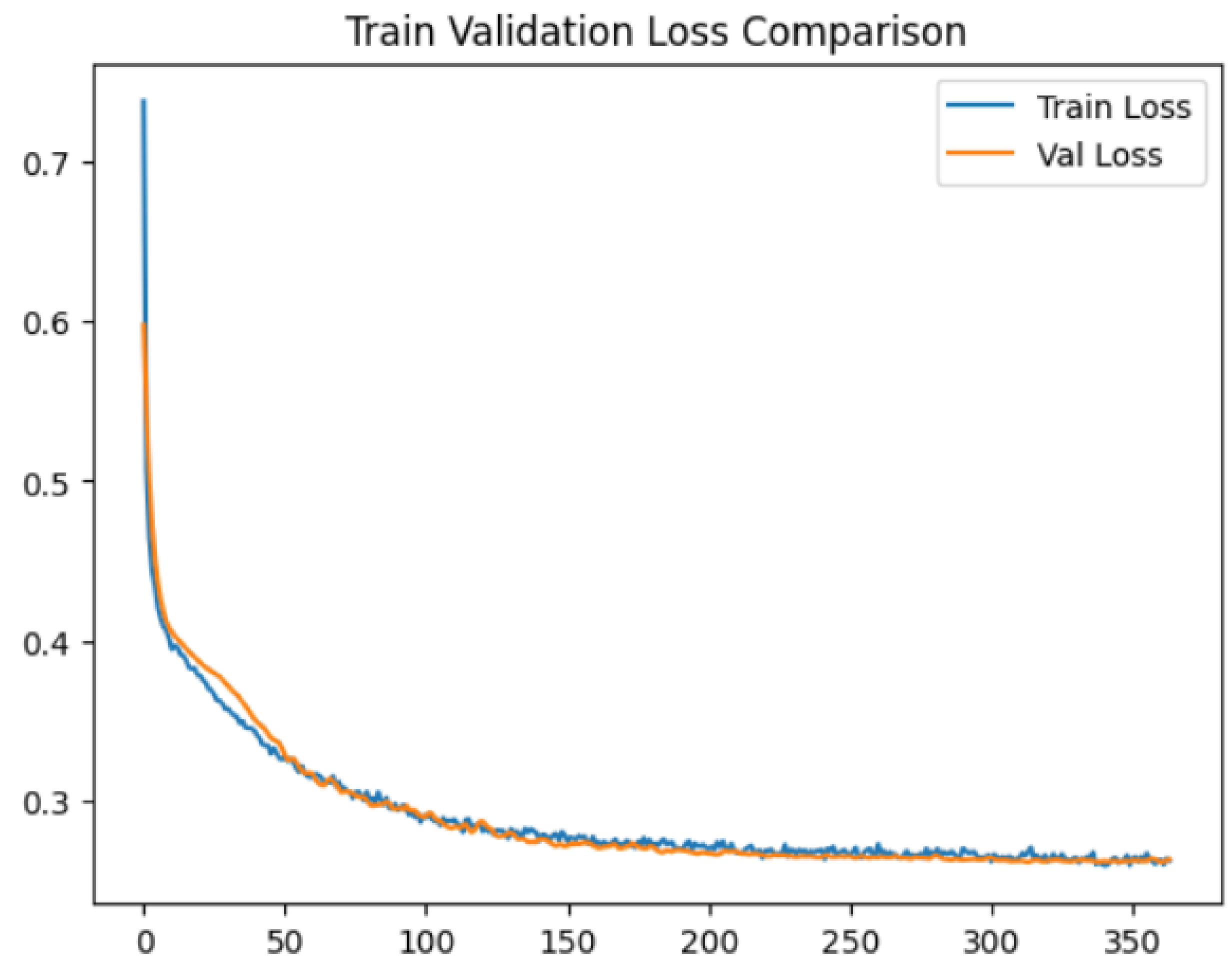


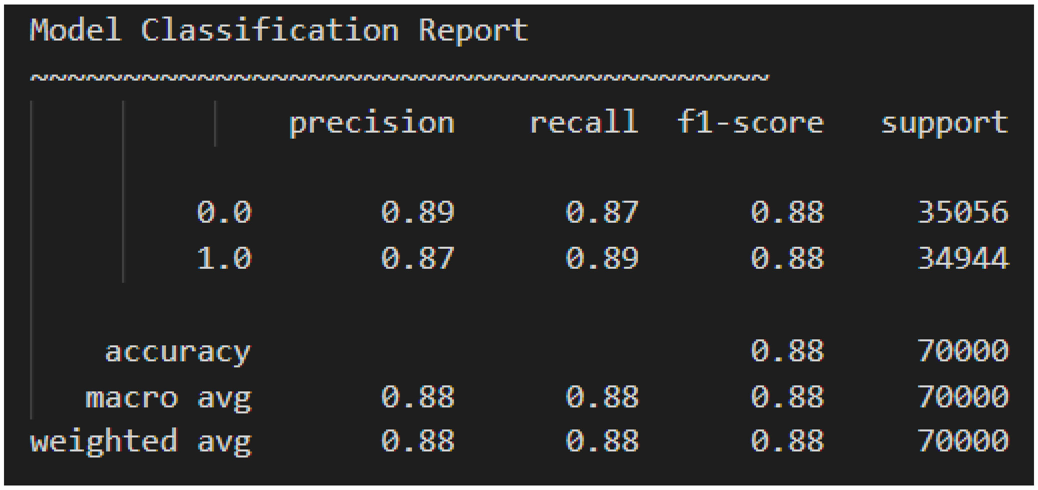
Figure 2:
Train/Validation Loss

Overall this shows the characteristic loss curve expected for a Neural Network. The val and train loss curves are overlapping which has some implications.

First it means that the model does not overfit at all and everything that the model learned is used directly for prediction on unseen data with no loss in performance. This is nice to see as it means our model is regularizing at no loss of performance.

Second the model does not achieve perfect performance which is displayed by the curve not approaching close to 0. This could be a sign that the model is not complex enough, too much regularization was introduced into the model which prevented learning, or the features do not provide enough information to achieve perfect performance. The regularization used in the model is quite small so that discounts the regularization concern. For the not complex enough concern I trained a model with 3x the weights and it did not perform any better so that concern is also covered. This leaves the final case as the most likely, the features present are good at separating the classes but not perfect. If we were to view the clusters in hyperspace the clusters would likely be overlapping.

Classification Report



```
Model Classification Report
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
              precision    recall  f1-score   support

         0.0       0.89      0.87      0.88     35056
         1.0       0.87      0.89      0.88     34944

    accuracy                           0.88     70000
   macro avg       0.88      0.88      0.88     70000
weighted avg       0.88      0.88      0.88     70000
```

Figure 3: Classification Report

Overall the model does well. It’s not a perfect optimizer but it achieves a solid performance across the board without sacrificing one class for the other. Since no preferred class was stated this result is ideal for the neutral prediction preference approach.

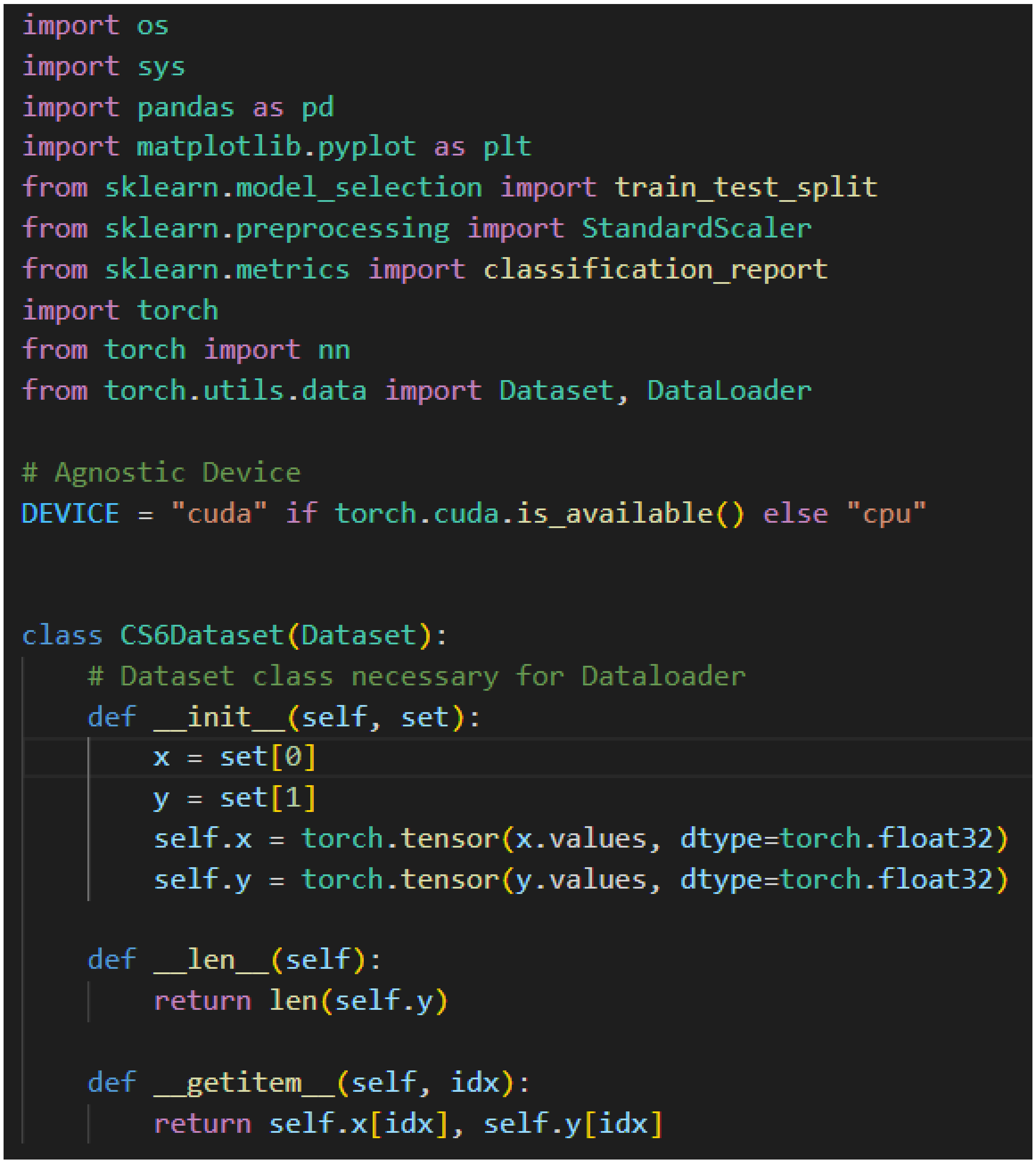
Additionally as further justification for our comparatively small test sets notice how the support for each class is around 35k for both classes. Despite only being 1% of the overall data this is a significant amount of support for both classes and is more than enough for model evaluation.

Conclusion

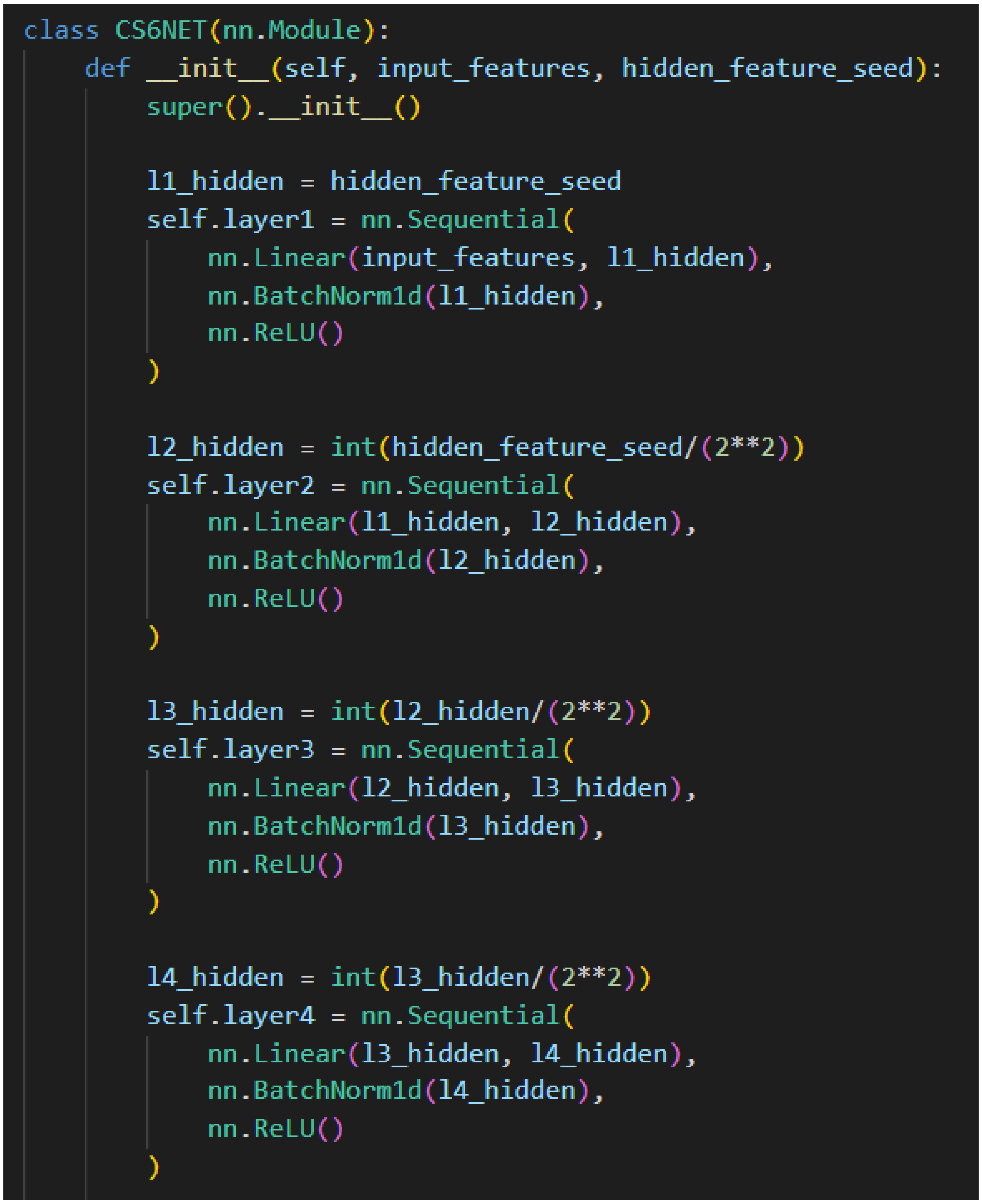
Neural networks have a significant increase in the number of hyperparameters to adjust. From designing the model itself to deciding the exact optimizer to use, the number of possible combinations are seemingly endless. With that said, getting thoroughly familiar with traditional machine learning methods in the past definitely helped as the overall procedure is still the same.

Code

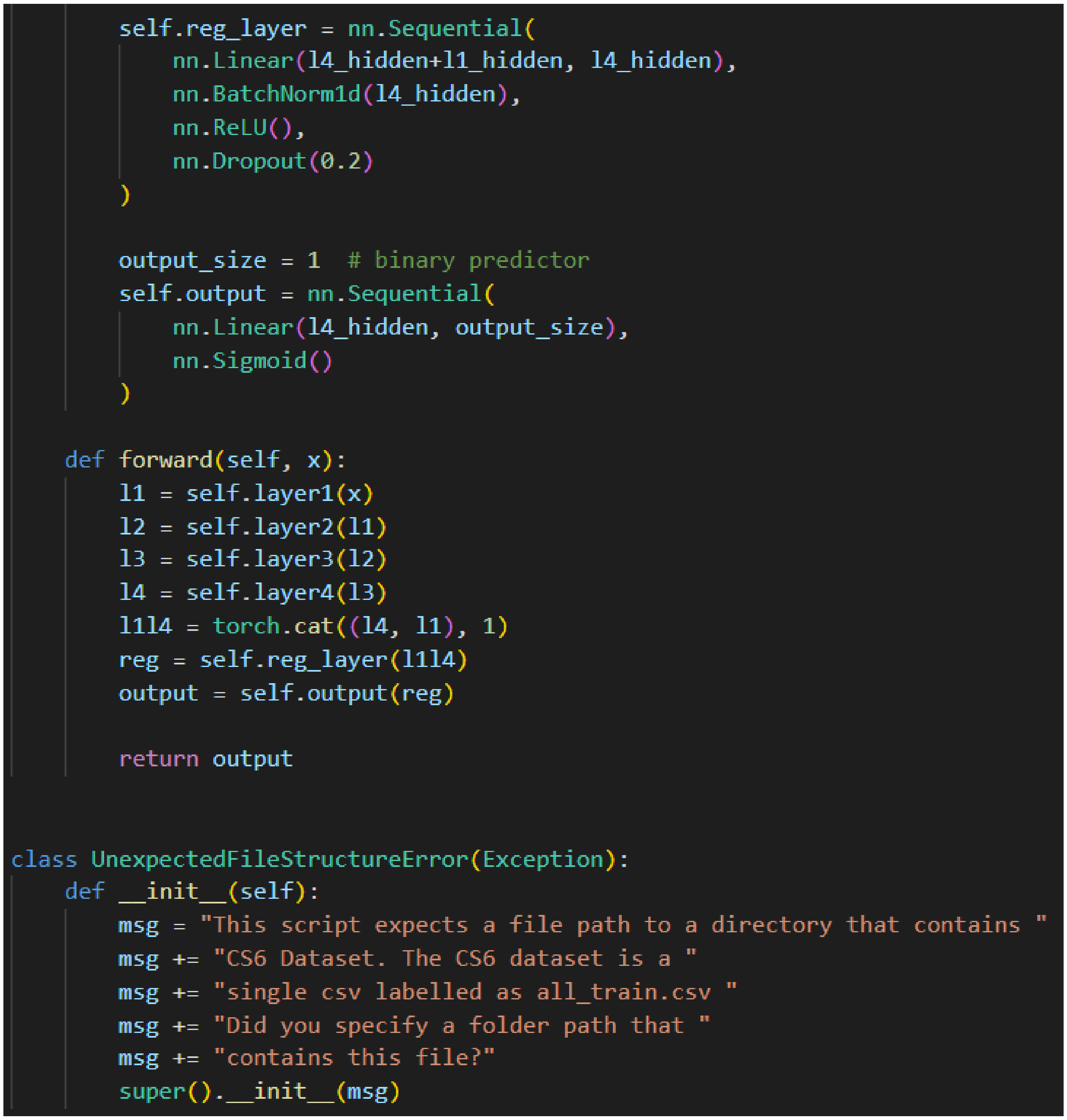
Appendix A: Script



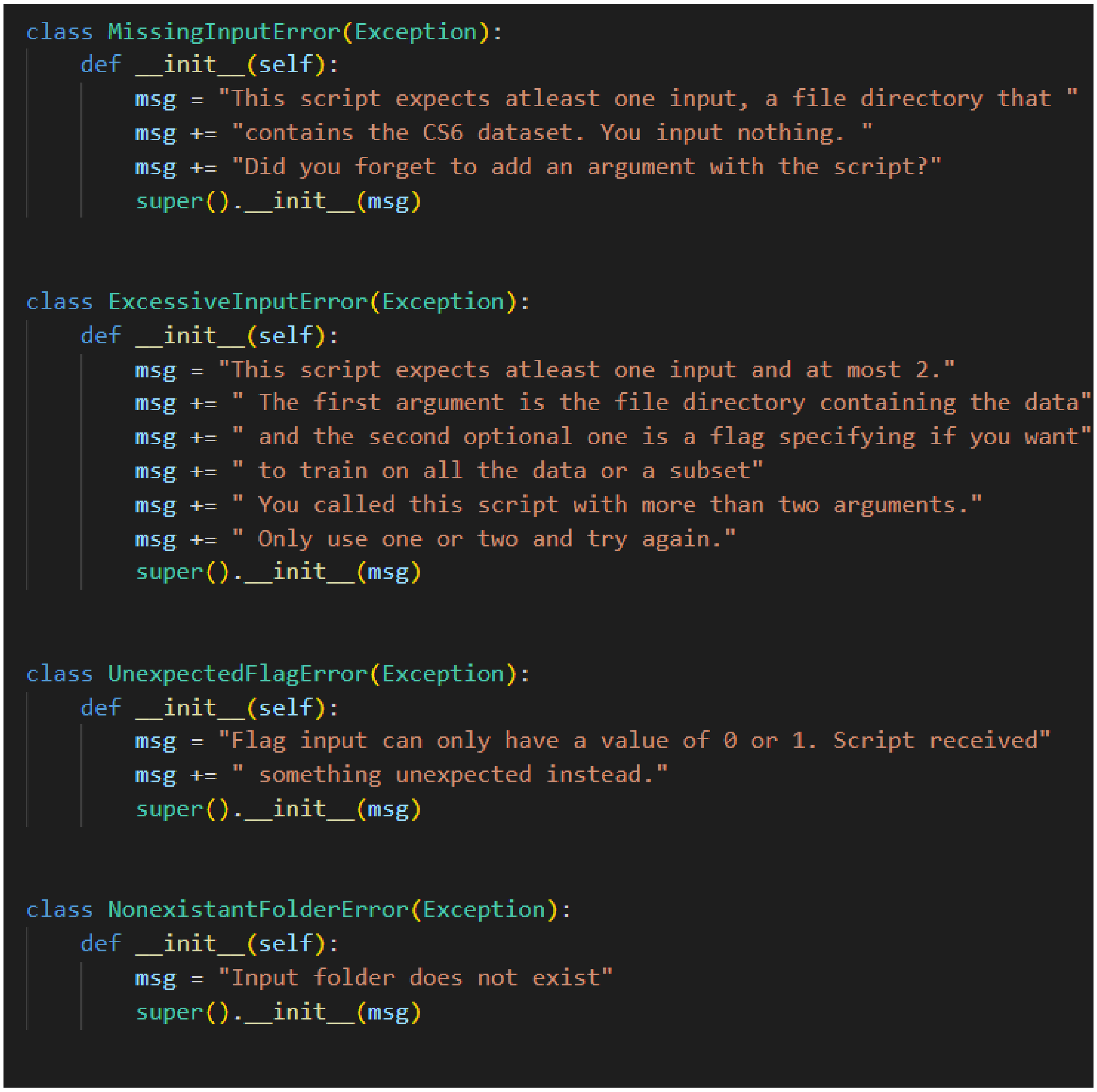
```python
import os
import sys
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report
import torch
from torch import nn
from torch.utils.data import Dataset, DataLoader

# Agnostic Device
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"


class CS6Dataset(Dataset):
    # Dataset class necessary for Dataloader
    def __init__(self, set):
        x = set[0]
        y = set[1]
        self.x = torch.tensor(x.values, dtype=torch.float32)
        self.y = torch.tensor(y.values, dtype=torch.float32)

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

    def __getitem__(self, idx):
        return self.x[idx], self.y[idx]
```



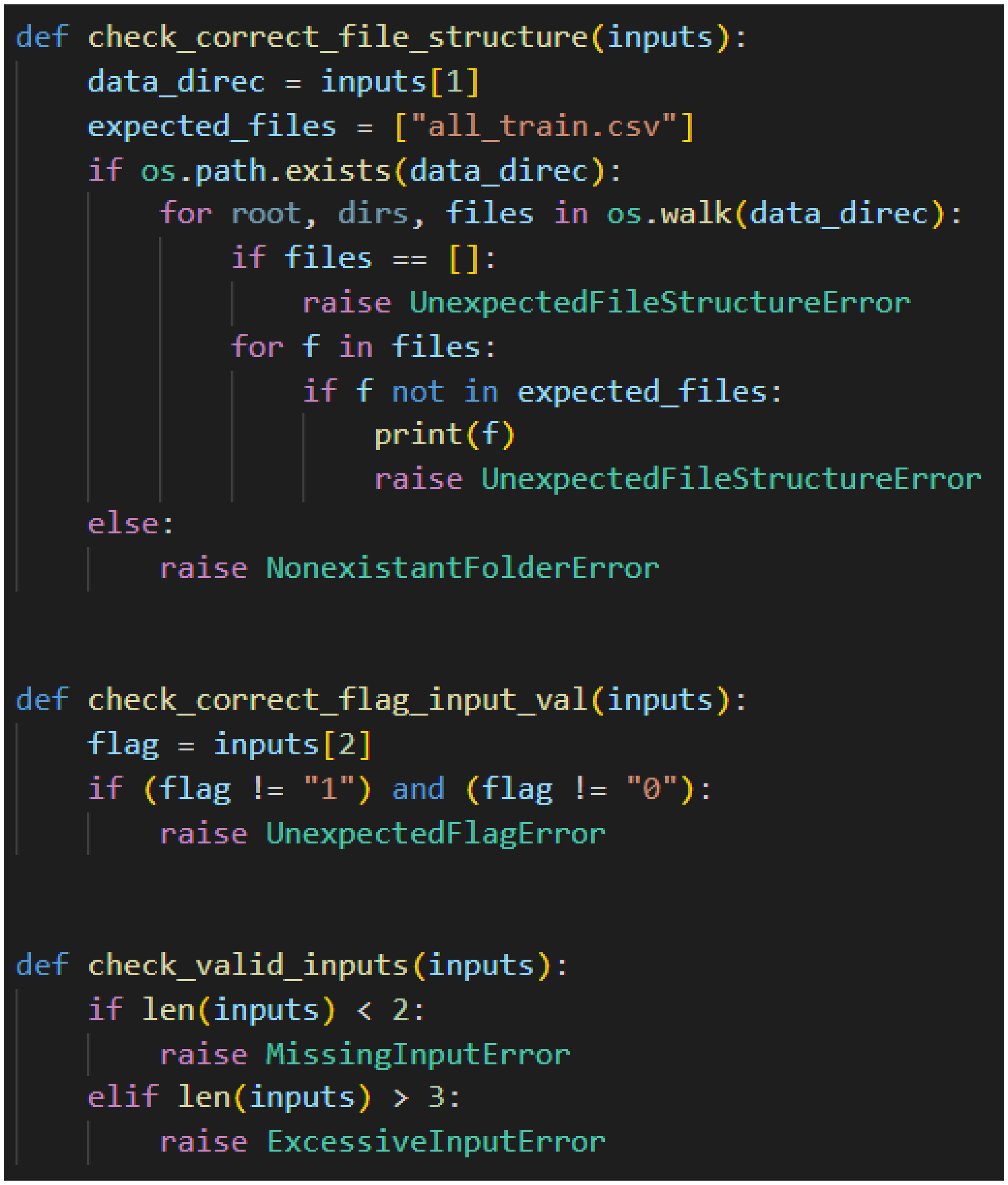
```python
class CS6NET(nn.Module):
    def __init__(self, input_features, hidden_feature_seed):
        super().__init__()

        l1_hidden = hidden_feature_seed
        self.layer1 = nn.Sequential(
            nn.Linear(input_features, l1_hidden),
            nn.BatchNorm1d(l1_hidden),
            nn.ReLU()
        )

        l2_hidden = int(hidden_feature_seed/(2**2))
        self.layer2 = nn.Sequential(
            nn.Linear(l1_hidden, l2_hidden),
            nn.BatchNorm1d(l2_hidden),
            nn.ReLU()
        )

        l3_hidden = int(l2_hidden/(2**2))
        self.layer3 = nn.Sequential(
            nn.Linear(l2_hidden, l3_hidden),
            nn.BatchNorm1d(l3_hidden),
            nn.ReLU()
        )

        l4_hidden = int(l3_hidden/(2**2))
        self.layer4 = nn.Sequential(
            nn.Linear(l3_hidden, l4_hidden),
            nn.BatchNorm1d(l4_hidden),
            nn.ReLU()
        )
```



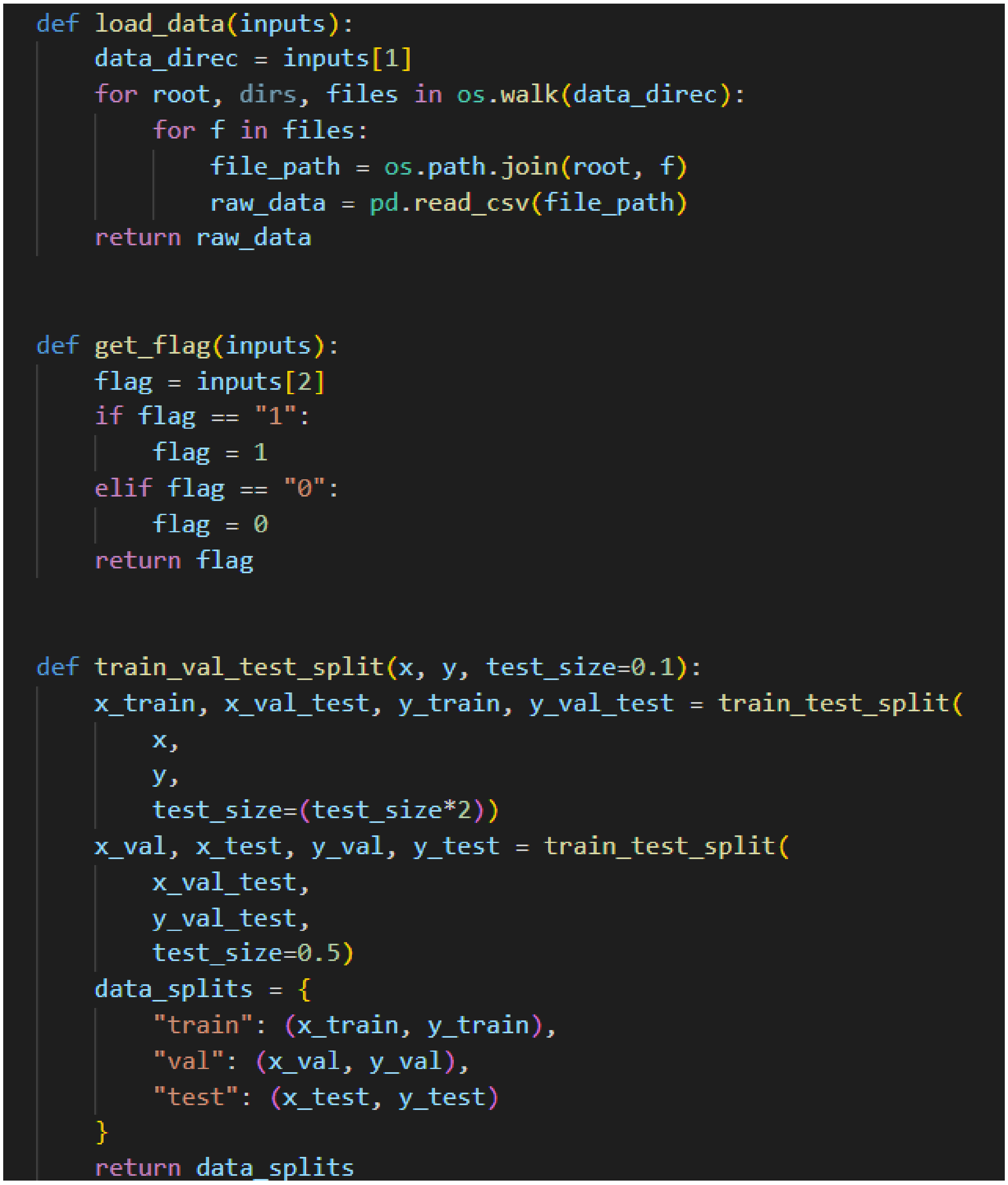
```python
        self.reg_layer = nn.Sequential(
            nn.Linear(l4_hidden+l1_hidden, l4_hidden),
            nn.BatchNorm1d(l4_hidden),
            nn.ReLU(),
            nn.Dropout(0.2)
        )

        output_size = 1  # binary predictor
        self.output = nn.Sequential(
            nn.Linear(l4_hidden, output_size),
            nn.Sigmoid()
        )

    def forward(self, x):
        l1 = self.layer1(x)
        l2 = self.layer2(l1)
        l3 = self.layer3(l2)
        l4 = self.layer4(l3)
        l1l4 = torch.cat((l4, l1), 1)
        reg = self.reg_layer(l1l4)
        output = self.output(reg)

        return output


class UnexpectedFileStructureError(Exception):
    def __init__(self):
        msg = "This script expects a file path to a directory that contains "
        msg += "CS6 Dataset. The CS6 dataset is a "
        msg += "single csv labelled as all_train.csv "
        msg += "Did you specify a folder path that "
        msg += "contains this file?"
        super().__init__(msg)
```

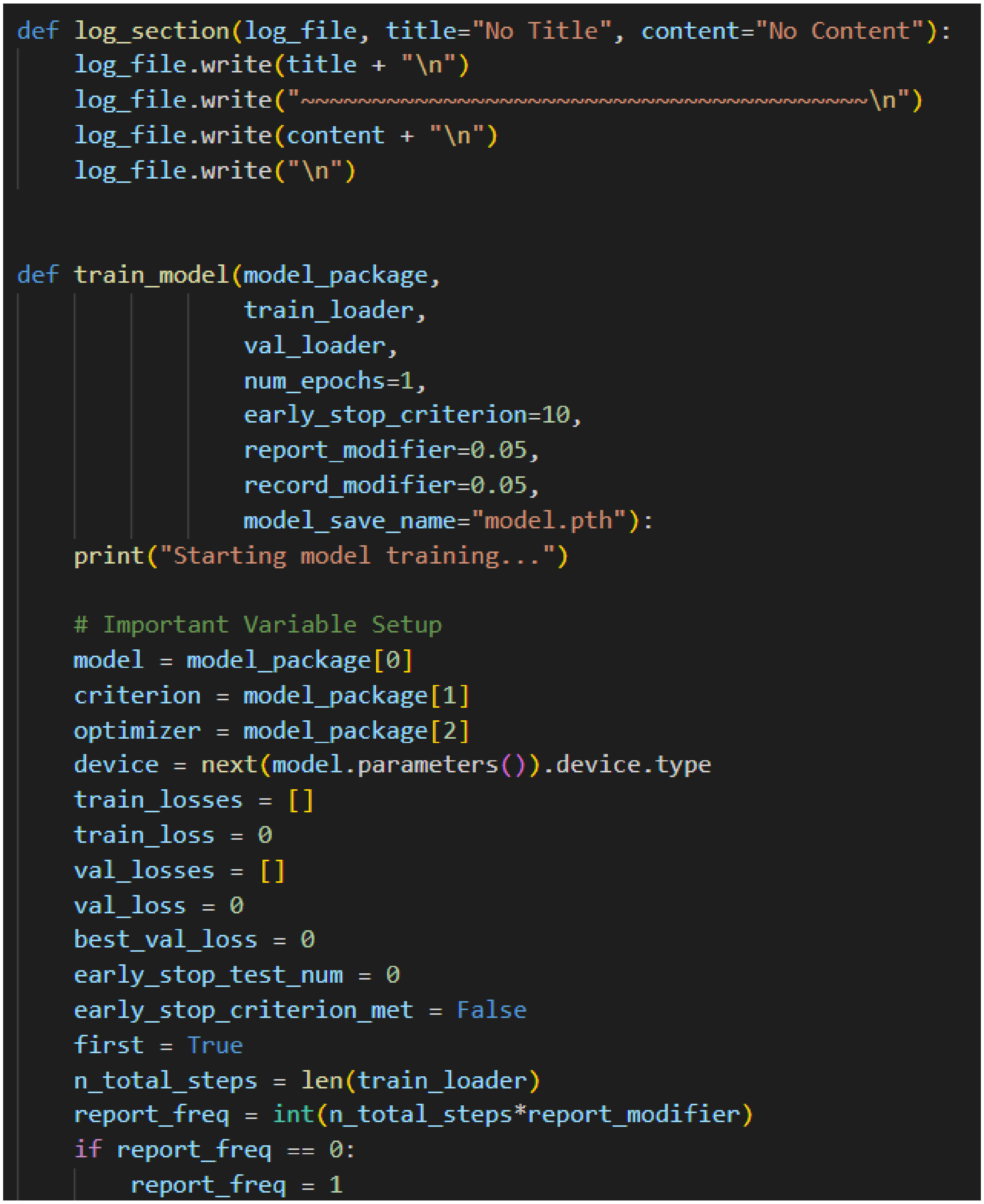


```python
class MissingInputError(Exception):
    def __init__(self):
        msg = "This script expects atleast one input, a file directory that "
        msg += "contains the CS6 dataset. You input nothing. "
        msg += "Did you forget to add an argument with the script?"
        super().__init__(msg)


class ExcessiveInputError(Exception):
    def __init__(self):
        msg = "This script expects atleast one input and at most 2."
        msg += " The first argument is the file directory containing the data"
        msg += " and the second optional one is a flag specifying if you want"
        msg += " to train on all the data or a subset"
        msg += " You called this script with more than two arguments."
        msg += " Only use one or two and try again."
        super().__init__(msg)


class UnexpectedFlagError(Exception):
    def __init__(self):
        msg = "Flag input can only have a value of 0 or 1. Script received"
        msg += " something unexpected instead."
        super().__init__(msg)


class NonexistantFolderError(Exception):
    def __init__(self):
        msg = "Input folder does not exist"
        super().__init__(msg)
```

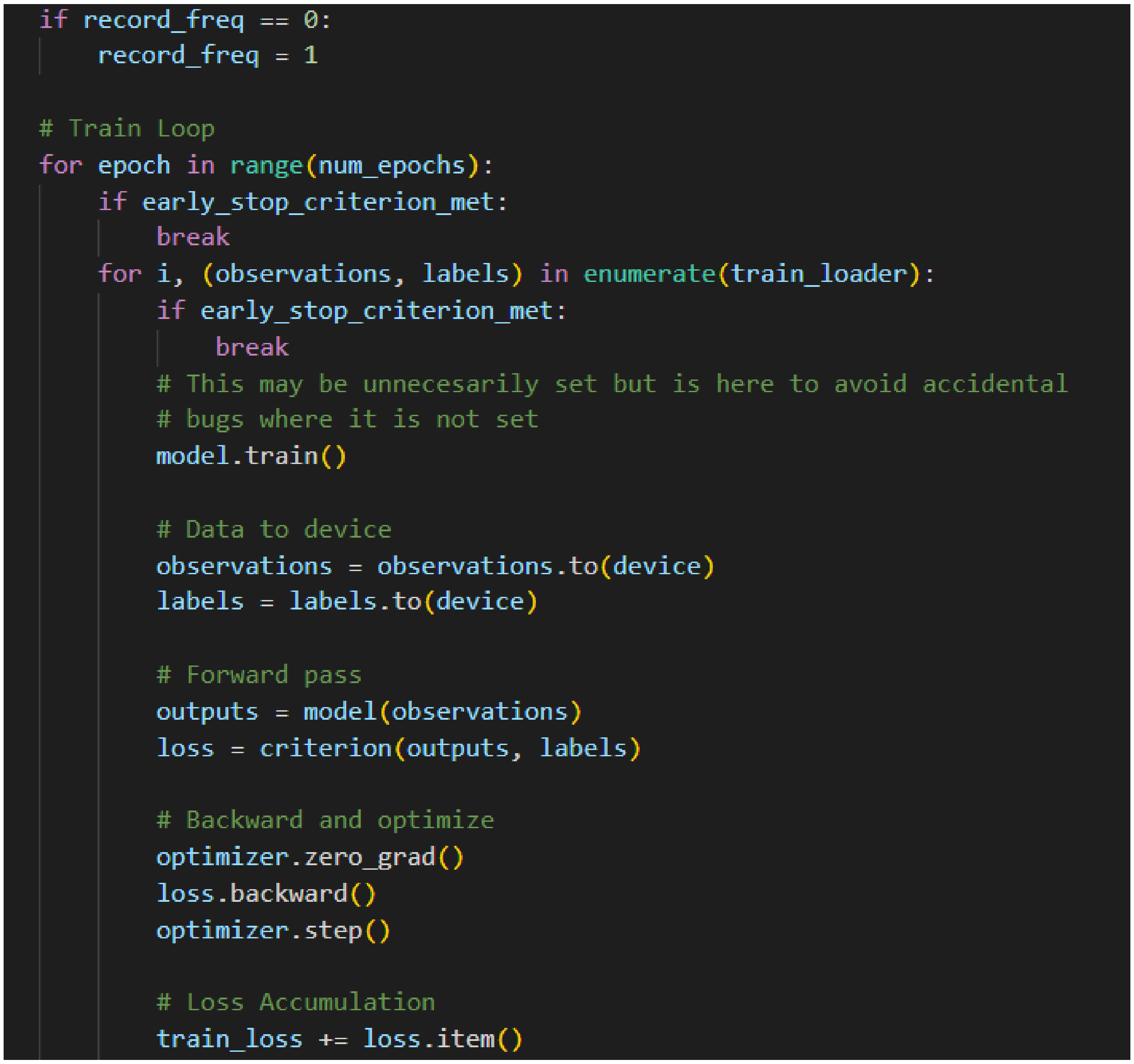


```python
def check_correct_file_structure(inputs):
    data_direc = inputs[1]
    expected_files = ["all_train.csv"]
    if os.path.exists(data_direc):
        for root, dirs, files in os.walk(data_direc):
            if files == []:
                raise UnexpectedFileStructureError
            for f in files:
                if f not in expected_files:
                    print(f)
                    raise UnexpectedFileStructureError
    else:
        raise NonexistantFolderError


def check_correct_flag_input_val(inputs):
    flag = inputs[2]
    if (flag != "1") and (flag != "0"):
        raise UnexpectedFlagError


def check_valid_inputs(inputs):
    if len(inputs) < 2:
        raise MissingInputError
    elif len(inputs) > 3:
        raise ExcessiveInputError
```

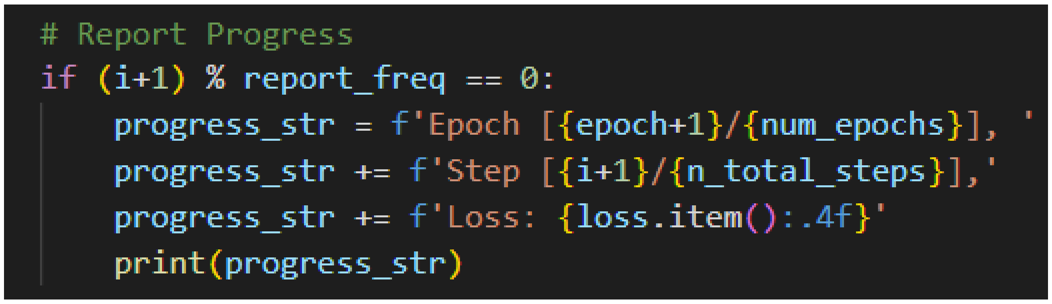

```python
def load_data(inputs):
    data_direc = inputs[1]
    for root, dirs, files in os.walk(data_direc):
        for f in files:
            file_path = os.path.join(root, f)
            raw_data = pd.read_csv(file_path)
    return raw_data


def get_flag(inputs):
    flag = inputs[2]
    if flag == "1":
        flag = 1
    elif flag == "0":
        flag = 0
    return flag


def train_val_test_split(x, y, test_size=0.1):
    x_train, x_val_test, y_train, y_val_test = train_test_split(
        x,
        y,
        test_size=(test_size*2))
    x_val, x_test, y_val, y_test = train_test_split(
        x_val_test,
        y_val_test,
        test_size=0.5)
    data_splits = {
        "train": (x_train, y_train),
        "val": (x_val, y_val),
        "test": (x_test, y_test)
    }
    return data_splits
```



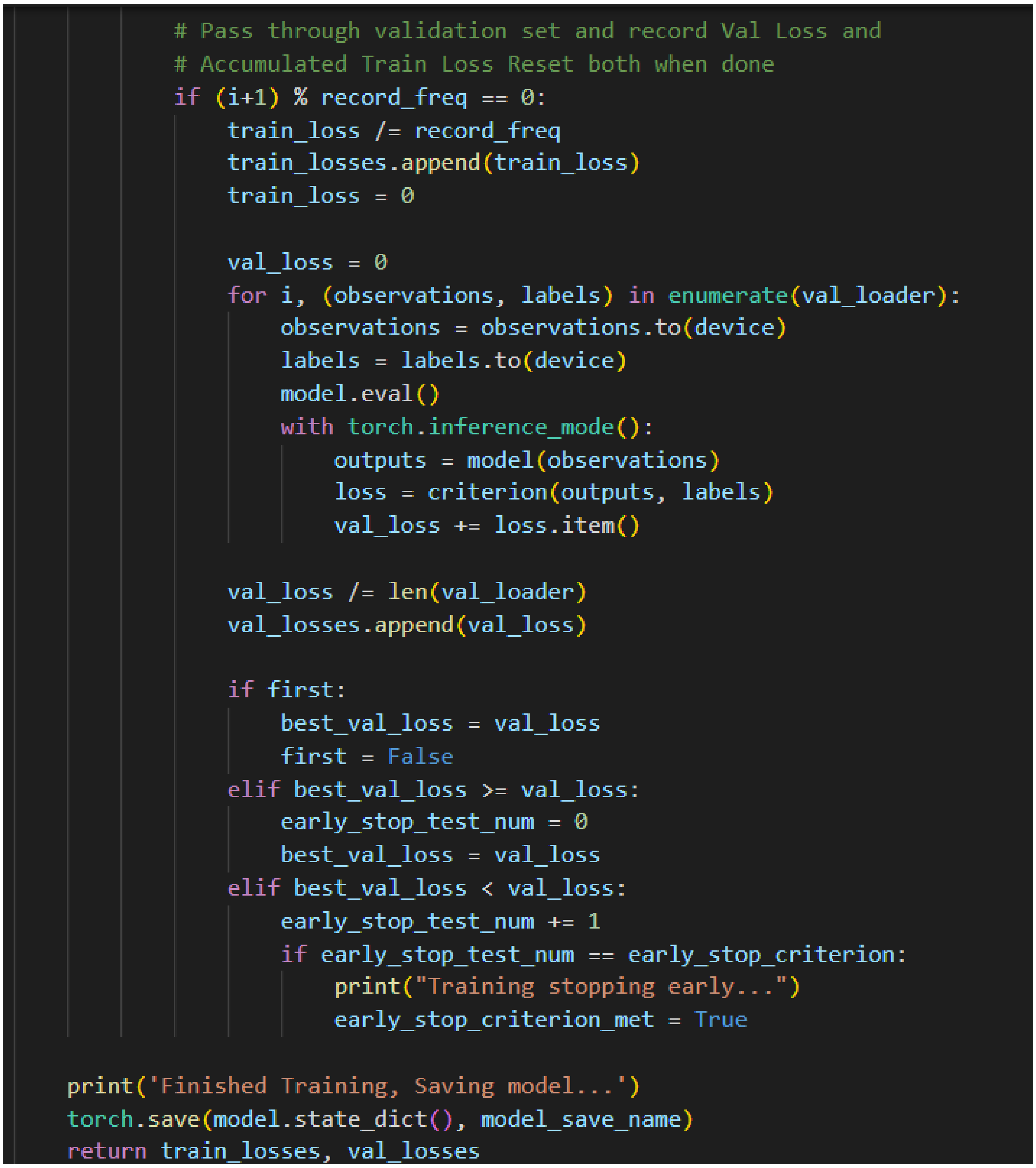
```python
def log_section(log_file, title="No Title", content="No Content"):
    log_file.write(title + "\n")
    log_file.write("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n")
    log_file.write(content + "\n")
    log_file.write("\n")


def train_model(model_package,
                train_loader,
                val_loader,
                num_epochs=1,
                early_stop_criterion=10,
                report_modifier=0.05,
                record_modifier=0.05,
                model_save_name="model.pth"):
    print("Starting model training...")

    # Important Variable Setup
    model = model_package[0]
    criterion = model_package[1]
    optimizer = model_package[2]
    device = next(model.parameters()).device.type
    train_losses = []
    train_loss = 0
    val_losses = []
    val_loss = 0
    best_val_loss = 0
    early_stop_test_num = 0
    early_stop_criterion_met = False
    first = True
    n_total_steps = len(train_loader)
    report_freq = int(n_total_steps*report_modifier)
    if report_freq == 0:
        report_freq = 1
```



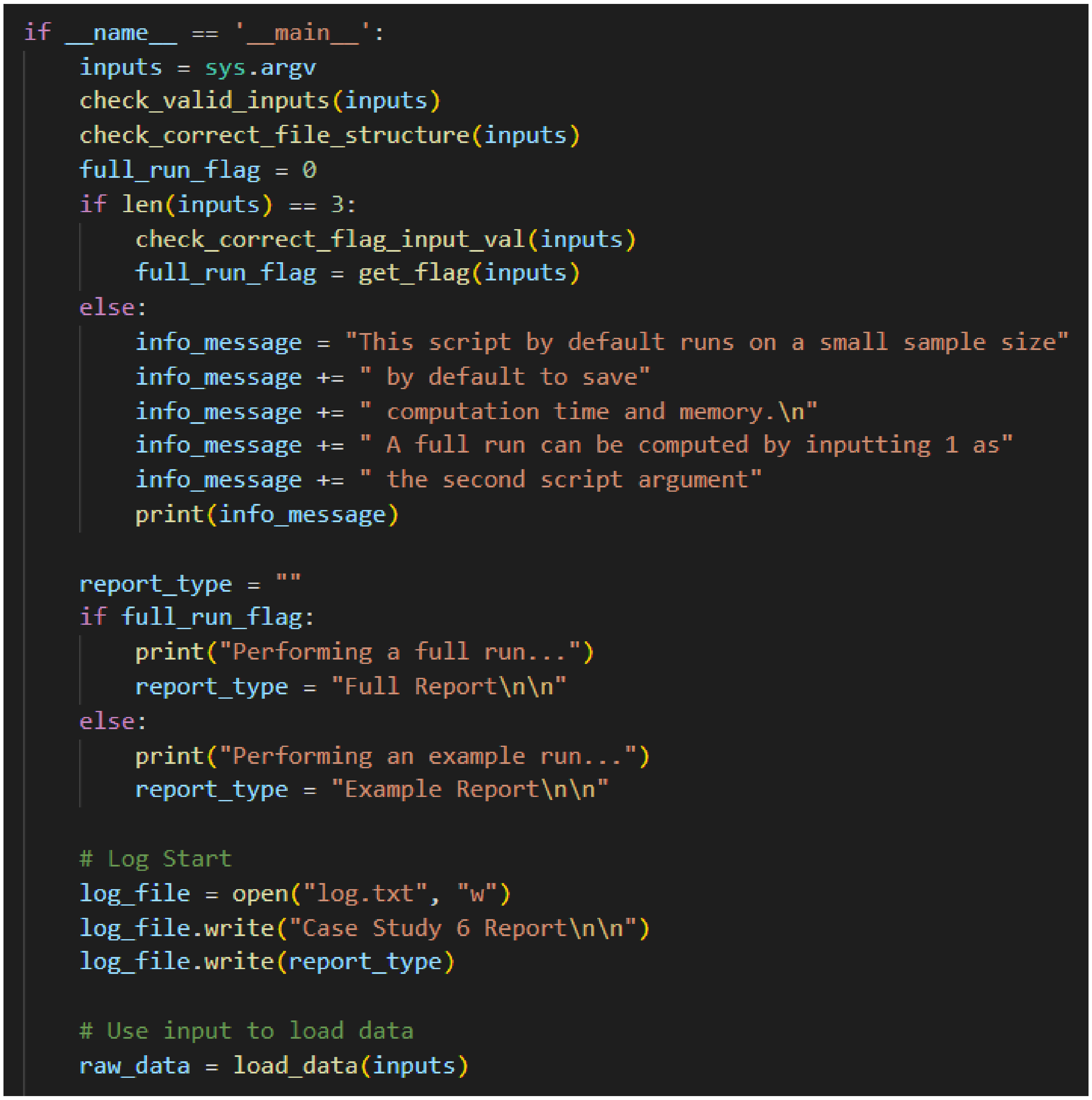
```python
if record_freq == 0:
    record_freq = 1

# Train Loop
for epoch in range(num_epochs):
    if early_stop_criterion_met:
        break
    for i, (observations, labels) in enumerate(train_loader):
        if early_stop_criterion_met:
            break
        # This may be unnecesarily set but is here to avoid accidental
        # bugs where it is not set
        model.train()

        # Data to device
        observations = observations.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(observations)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Loss Accumulation
        train_loss += loss.item()
```

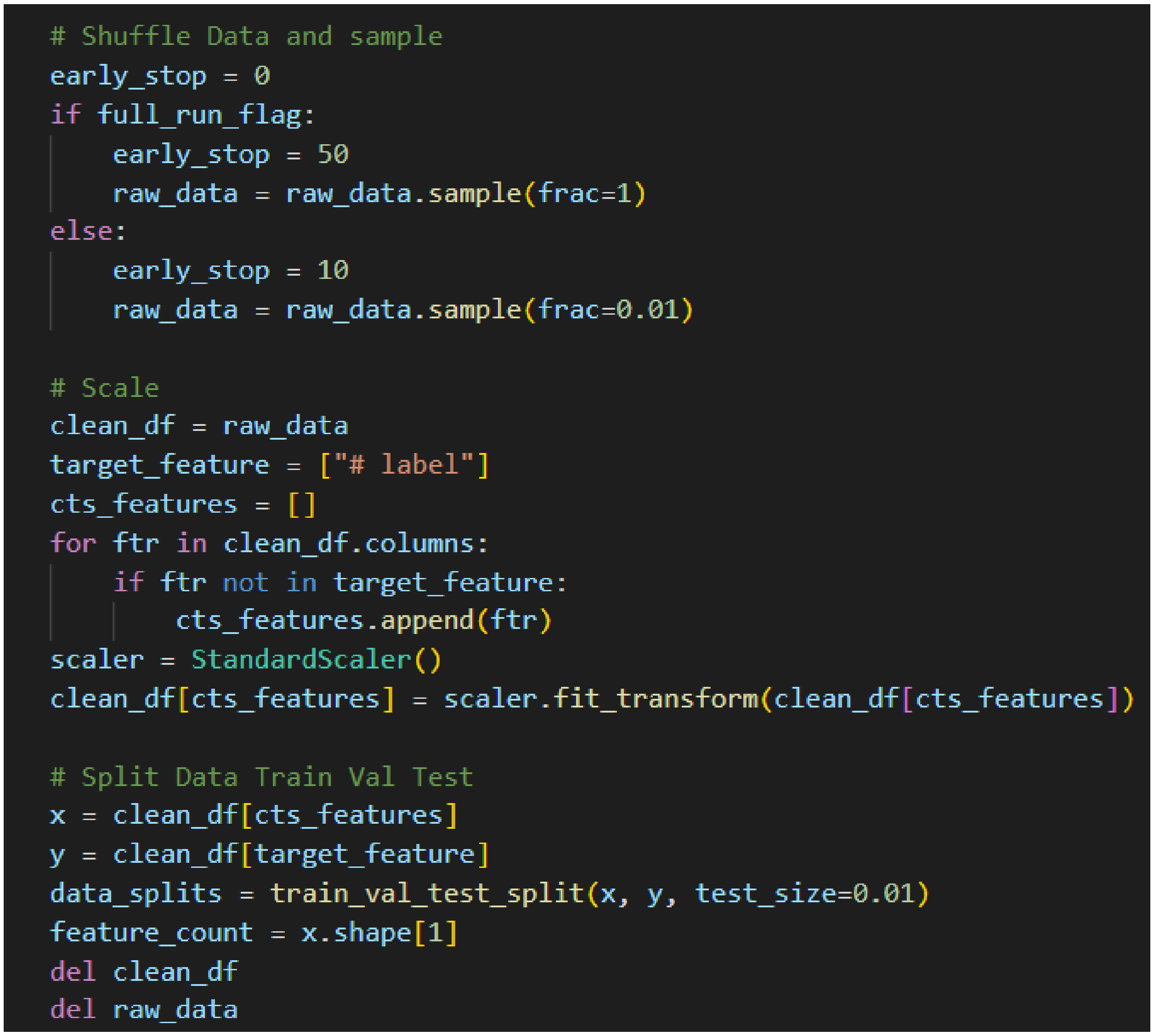


```python
# Report Progress
if (i+1) % report_freq == 0:
    progress_str = f'Epoch [{epoch+1}/{num_epochs}], '
    progress_str += f'Step [{i+1}/{n_total_steps}],'
    progress_str += f'Loss: {loss.item():.4f}'
    print(progress_str)
```

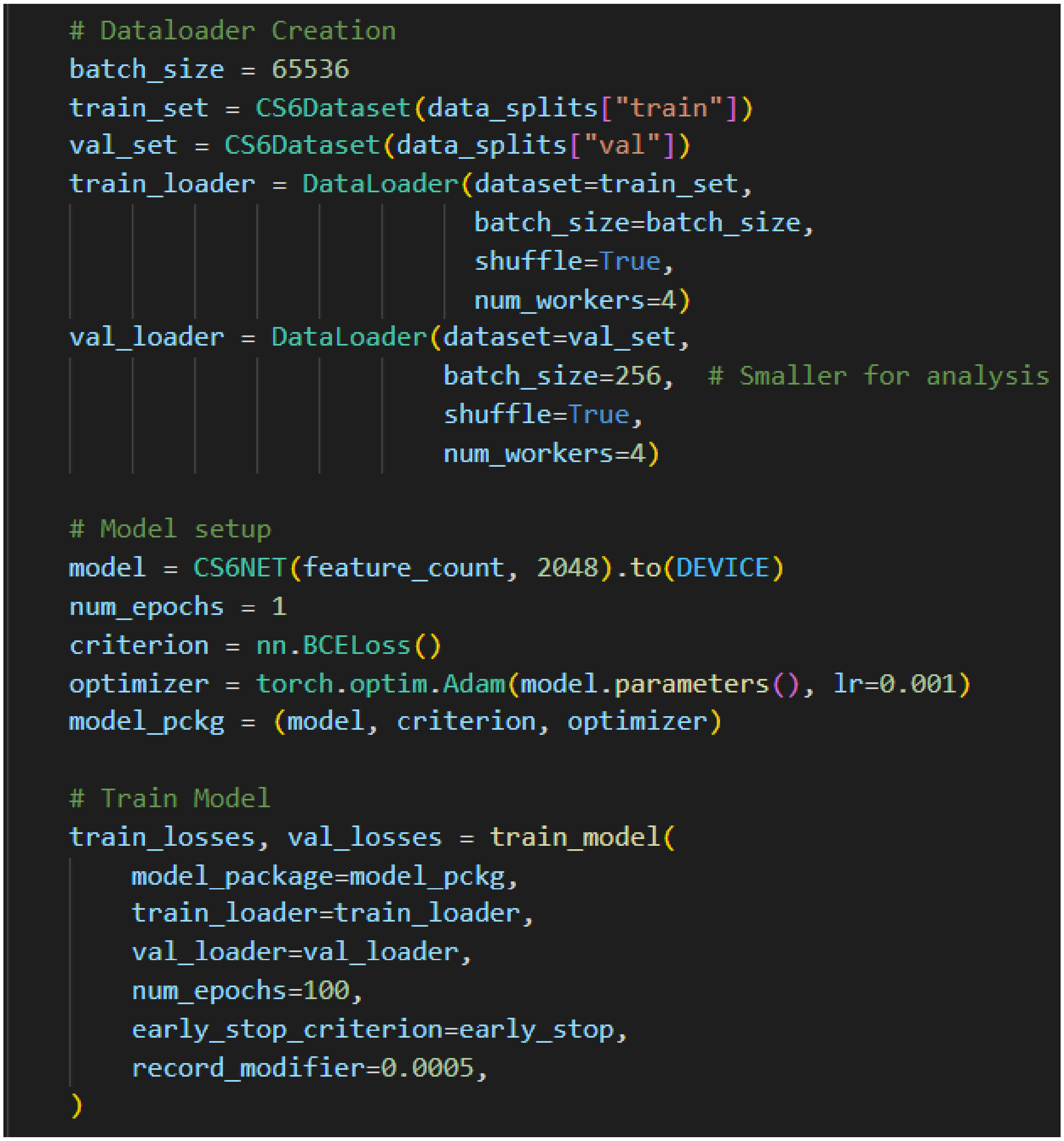


```python
            # Pass through validation set and record Val Loss and
            # Accumulated Train Loss Reset both when done
            if (i+1) % record_freq == 0:
                train_loss /= record_freq
                train_losses.append(train_loss)
                train_loss = 0

                val_loss = 0
                for i, (observations, labels) in enumerate(val_loader):
                    observations = observations.to(device)
                    labels = labels.to(device)
                    model.eval()
                    with torch.inference_mode():
                        outputs = model(observations)
                        loss = criterion(outputs, labels)
                        val_loss += loss.item()

                val_loss /= len(val_loader)
                val_losses.append(val_loss)

                if first:
                    best_val_loss = val_loss
                    first = False
                elif best_val_loss >= val_loss:
                    early_stop_test_num = 0
                    best_val_loss = val_loss
                elif best_val_loss < val_loss:
                    early_stop_test_num += 1
                    if early_stop_test_num == early_stop_criterion:
                        print("Training stopping early...")
                        early_stop_criterion_met = True

    print('Finished Training, Saving model...')
    torch.save(model.state_dict(), model_save_name)
    return train_losses, val_losses
```

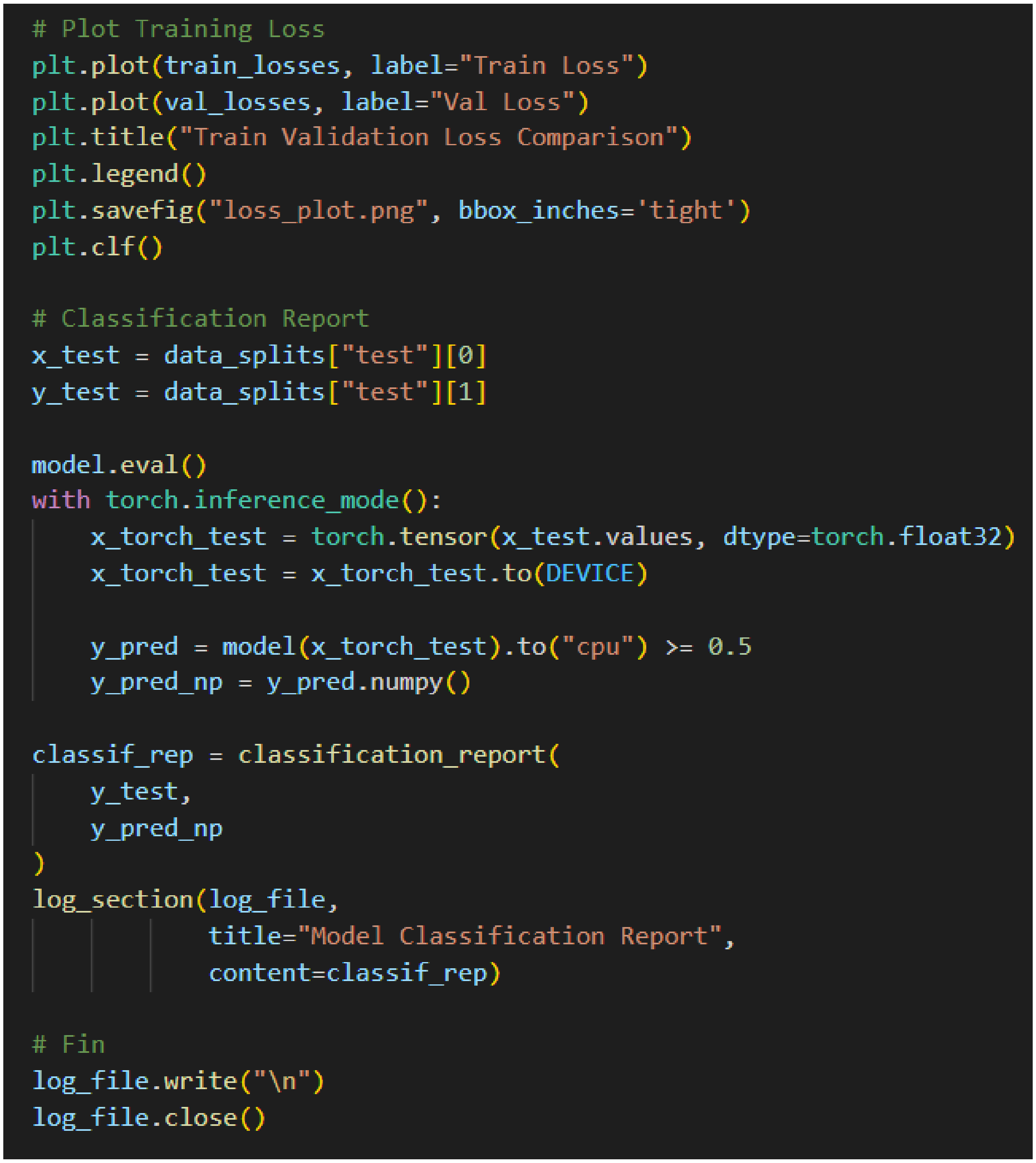

```python
if __name__ == '__main__':
    inputs = sys.argv
    check_valid_inputs(inputs)
    check_correct_file_structure(inputs)
    full_run_flag = 0
    if len(inputs) == 3:
        check_correct_flag_input_val(inputs)
        full_run_flag = get_flag(inputs)
    else:
        info_message = "This script by default runs on a small sample size"
        info_message += " by default to save"
        info_message += " computation time and memory.\n"
        info_message += " A full run can be computed by inputting 1 as"
        info_message += " the second script argument"
        print(info_message)

    report_type = ""
    if full_run_flag:
        print("Performing a full run...")
        report_type = "Full Report\n\n"
    else:
        print("Performing an example run...")
        report_type = "Example Report\n\n"

    # Log Start
    log_file = open("log.txt", "w")
    log_file.write("Case Study 6 Report\n\n")
    log_file.write(report_type)

    # Use input to load data
    raw_data = load_data(inputs)
```



```python
# Shuffle Data and sample
early_stop = 0
if full_run_flag:
    early_stop = 50
    raw_data = raw_data.sample(frac=1)
else:
    early_stop = 10
    raw_data = raw_data.sample(frac=0.01)

# Scale
clean_df = raw_data
target_feature = ["# label"]
cts_features = []
for ftr in clean_df.columns:
    if ftr not in target_feature:
        cts_features.append(ftr)
scaler = StandardScaler()
clean_df[cts_features] = scaler.fit_transform(clean_df[cts_features])

# Split Data Train Val Test
x = clean_df[cts_features]
y = clean_df[target_feature]
data_splits = train_val_test_split(x, y, test_size=0.01)
feature_count = x.shape[1]
del clean_df
del raw_data
```



```python
# Dataloader Creation
batch_size = 65536
train_set = CS6Dataset(data_splits["train"])
val_set = CS6Dataset(data_splits["val"])
train_loader = DataLoader(dataset=train_set,
                          batch_size=batch_size,
                          shuffle=True,
                          num_workers=4)
val_loader = DataLoader(dataset=val_set,
                        batch_size=256,  # Smaller for analysis
                        shuffle=True,
                        num_workers=4)

# Model setup
model = CS6NET(feature_count, 2048).to(DEVICE)
num_epochs = 1
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
model_pckg = (model, criterion, optimizer)

# Train Model
train_losses, val_losses = train_model(
    model_package=model_pckg,
    train_loader=train_loader,
    val_loader=val_loader,
    num_epochs=100,
    early_stop_criterion=early_stop,
    record_modifier=0.0005,
)
```



```python
# Plot Training Loss
plt.plot(train_losses, label="Train Loss")
plt.plot(val_losses, label="Val Loss")
plt.title("Train Validation Loss Comparison")
plt.legend()
plt.savefig("loss_plot.png", bbox_inches='tight')
plt.clf()

# Classification Report
x_test = data_splits["test"][0]
y_test = data_splits["test"][1]

model.eval()
with torch.inference_mode():
    x_torch_test = torch.tensor(x_test.values, dtype=torch.float32)
    x_torch_test = x_torch_test.to(DEVICE)

    y_pred = model(x_torch_test).to("cpu") >= 0.5
    y_pred_np = y_pred.numpy()

classif_rep = classification_report(
    y_test,
    y_pred_np
)
log_section(log_file,
            title="Model Classification Report",
            content=classif_rep)

# Fin
log_file.write("\n")
log_file.close()
```