FFNN Multiclass Classification

September 11, 2022

1 Initialisation

1.1 Imports and General Functions

[2]:
```python
def IsNotebook():
    """Indicate the shell name, whether code is running on a notebook, and if␣
 ↪so whether it's hosted on googlecolab."""
    isnotebook, isgooglecolab, shell = None, None, None
    try:
        shell = get_ipython().__class__.__name__
        if shell == 'ZMQInteractiveShell':
            isnotebook, isgooglecolab = True, False   # Jupyter notebook or␣
 ↪qtconsole
        elif shell == "Shell":
            isnotebook, isgooglecolab = True, True    # Google Colab
        elif shell == 'TerminalInteractiveShell':
            isnotebook, isgooglecolab = False, False  # Terminal running IPython
        else:
            isnotebook, isgooglecolab = False, False  # Other type (?)
    except NameError:
        isnotebook, isgooglecolab = False, False      # Probably standard␣
 ↪Python interpreter
    return shell, isnotebook, isgooglecolab
shell, isnotebook, isgooglecolab = IsNotebook()
if isnotebook and not isgooglecolab: #If we are in a notebook but not on google␣
 ↪colab, let's use all the available screen
    from IPython.display import display, HTML
    display(HTML("<style>.container { width:99% !important; }</style>"))
    if not isgooglecolab:
        try: #Using the jedi completer takes too long to complete words
            %config Completer.use_jedi = False
        except:
            pass
if isgooglecolab: #If we are in a google colab environment, we probably need to␣
 ↪mount our google drive
    try:
        from google.colab import drive
```

```
        drive.mount('/content/drive')
    except Exception as e:
        print(e)
```

<IPython.core.display.HTML object>

[36]:
```
#######################
### General Imports ###
import os #Making sure we're using all CPU cores for faster calculations
IsWindows = os.name == 'nt'
os.environ["OMP_NUM_THREADS"] = str(os.cpu_count())
os.environ["OPENBLAS_NUM_THREADS"] = str(os.cpu_count())
os.environ["MKL_NUM_THREADS"] = str(os.cpu_count())
os.environ["VECLIB_MAXIMUM_THREADS"] = str(os.cpu_count())
os.environ["NUMEXPR_NUM_THREADS"] = str(os.cpu_count())

import sys #Printing version for posterity
print("Python version:", sys.version)

try: #Allows saving and loading of variables
    import pickle5 as pickle
except:
    import pickle
try: #Printing version for posterity
    print("Pickle version:", pickle.__version__)
except:
    print("Pickle version:", pickle.format_version)

import dill as dill #Allows even deeper saving (associated classes, etc., as
 ↪well)
print("Dill version:", dill.__version__)

import warnings #Ability to create custom warnings, like warnings.
 ↪warn("deprecated", DeprecationWarning)
import itertools #Needed for Confusion Matrix

if IsWindows:
    import winsound #Uses the computer's speakers to alert you (e.g. when
 ↪training is done)
from tqdm import tqdm #Iterations can show a progress bar (like in Training)
from collections import Counter #Allows for frequency counting similar with R's
 ↪"table"
from collections import OrderedDict
#######################


#####################
```

```python
### Date and Time ###
import time #Gets the current time
from pytz import timezone #Allows for timezones to be set. #pytz.all_timezones
from datetime import datetime #Allows for Datetime objects like current Datetime. #datetime.fromisoformat('2021-05-24')
#There's also: np.datetime64('2021-08-01')
#####################


###################
### Mathematics ###
import numpy as np #Working with numeric arrays
print("Numpy version:", np.__version__)
###################


#######################################
### Statistics and Machine Learning ###
#Utility
from sklearn.preprocessing import OrdinalEncoder, StandardScaler, MinMaxScaler #Various ways of scaling the data
from sklearn.model_selection import train_test_split

#Metrics
from sklearn.metrics import f1_score, precision_score, recall_score, RocCurveDisplay, PrecisionRecallDisplay
from sklearn.metrics import roc_auc_score
from sklearn.metrics import confusion_matrix
#######################################


##################
### Dataframes ###
import pandas as pd
##################


#############
### Plots ###
import matplotlib.pyplot as plt #Allows use of Pyplot plots

import seaborn as sns #Allows use of Seaborn plots
sns.set() #Sets default plot theme

from matplotlib.ticker import AutoMinorLocator, MultipleLocator
#############
```

```python
#########################
### Images / Pictures ###
from PIL import Image
#########################


######################
### String or Text ###
import json #Can encode or decode JSON string objects
import string #Provides functions for strings
######################


###################################
### Files, Directories, Folders ###
from pathlib import Path
###################################


################################
### Neural Network Libraries ###
#General
import torch
import torch.nn as nn
# from torchsummary import summary
if isgooglecolab:
    !pip install torchinfo
from torchinfo import summary

#Data
from torch.utils.data import Dataset, TensorDataset

#Images
from torchvision import datasets, transforms, models

#Info and configuration
print()
print(f"PyTorch v{torch.__version__}")
IS_GPU_AVAILABLE = torch.cuda.is_available()
print(f"CUDA device available: {IS_GPU_AVAILABLE}")
if (torch.cuda.is_available()):
    print(f"{torch.cuda.device_count()} devices available")
    for n in range(torch.cuda.device_count()):
        print("\t" + torch.cuda.get_device_name(n))
    print(f"cuda: {torch.cuda.current_device()}") #It can give you information like the GPU is not supported
```

```python
print("Num threads set to:", os.cpu_count())
torch.set_num_threads(os.cpu_count())
################################


########################
### Useful functions ###
if "OrigFigSize" not in locals() and "OrigFigSize" not in globals(): #Just in␣
↪case Initialisation is re-run after any of these have chaned
    OrigFigSize   = plt.rcParams["figure.figsize"]

NonNans = lambda List: List[np.logical_not(np.isnan(List))]
LastNonNan = lambda List: NonNans(List)[-1] if np.sum(np.isnan(List)) <␣
↪len(List) else np.array([]) if type(List) == np.ndarray else []

def ZeroANumber(Number, MaxLength, ForceMaxLength = False):
    """Take a single Number and prepend '0's to it until it meets MaxLength, or␣
↪if ForceMaxLength then also clip digits from the end until it meets␣
↪MaxLength."""
    res = str(Number).zfill(MaxLength)
    if ForceMaxLength: res = res[:MaxLength]
    return res

def SpaceAString(CurString, MaxLength, SpaceTheFront = True, ForceMaxLength =␣
↪False, ForceRemoveFromFront = False):
    """Prepend/Append (SpaceTheFront) spaces to CurString until it meets␣
↪ForceMaxLength or if ForceMaxLength also Clip characters from the beginning/
↪end (ForceRemoveFromFront) until it meets ForceMaxLength."""
    CurLen = len(CurString)
    Result = CurString

    if CurLen < MaxLength:
        if SpaceTheFront:
            Result = (" " * (MaxLength-CurLen)) + CurString
        else:
            Result = CurString + (" " * (MaxLength-CurLen))
    elif CurLen > MaxLength and ForceMaxLength:
        if ForceRemoveFromFront:
            Result = CurString[(CurLen - MaxLength):]
        else:
            Result = CurString[:-(CurLen - MaxLength)]
    return Result

def WriteText(TextParsableVar, FullFilePath):
    """Take a string (or string-parsable variable) and save it as text file on␣
↪the directory and with a name indicated by FullFilePath."""
```

```
    try:
        DirName = Path(FullFilePath).parent.absolute()
        os.makedirs(DirName, exist_ok = True)

        FileOptions = open(FullFilePath, "w")
        FileOptions.writelines(
            f"{TextParsableVar}"
        )
    except Exception as e:
        print(f"Exception:\n{e}")
    finally:
        try:
            FileOptions.close()
        except Exception:
            pass
SaveText = lambda TextParsableVar, FullFilePath: WriteText(TextParsableVar,␣
 ↪FullFilePath) #Alias for WriteText to be the same as Save/Load Variable

def ReadText(FullFilePath):
    """Read the string content of a text file given by FullFilePath and return␣
 ↪it as a string."""
    with open(FullFilePath, "r+", encoding = "utf8") as io:
        TextString = io.read()
    return TextString
LoadText = lambda FullFilePath: ReadText(FullFilePath) #Alias for ReadText to␣
 ↪be the same as Save/Load Variable

def SaveVariable(Variable, FileName):
    """Create the directory path for and pickle Variable under FileName."""
    DirName = Path(FileName).parent.absolute()
    os.makedirs(DirName, exist_ok = True)
    with open(FileName, 'wb') as io:
        pickle.dump(Variable, io)

def SaveVariableDill(Variable, FileName):
    """Create the directory path for and deep-save Variable under FileName␣
 ↪using dill."""
    DirName = Path(FileName).parent.absolute()
    os.makedirs(DirName, exist_ok = True)
    with open(FileName, 'wb') as io:
        dill.dump(Variable, io)

def LoadVariable(FileName):
    """Un-pickle a binary file saved under FileName and return it as a variable.
 ↪"""
    with open(FileName, "rb") as io:
        Res = pickle.load(io)
```

```
    return Res

def LoadVariableDill(FileName):
    """Read the content of a binary file saved under FileName and return it as␣
 ↪a variable."""
    with open(FileName, 'rb') as io:
        Res = dill.load(io)
    return Res

def RemLastLine(s):
    """Remove the last line in the string s."""
    return s[:s.rfind('\n')]
########################

device = "cuda" if torch.cuda.is_available() else "cpu"
# device = "cpu" #To FORCE CPU
print("device=", device)
```

```
Python version: 3.8.13 (default, Mar 28 2022, 06:59:08) [MSC v.1916 64 bit
(AMD64)]
Pickle version: 4.0
Dill version: 0.3.4
Numpy version: 1.21.5

PyTorch v1.11.0
CUDA device available: True
1 devices available
        NVIDIA GeForce RTX 2080 SUPER
cuda: 0
Num threads set to: 48
device= cuda
```

1.2 Architecture

[189]:
```
#FeedForward Neural Network
class Net(nn.Module):
    def __init__(self, K_Length, num_units, activation, dropout, usebias):
        super(Net, self).__init__()
        self.K_Length = K_Length
        self.num_units = num_units
        self.activation = activation
        self.dropout = dropout
        self.usebias = usebias

        self.layers = nn.ModuleList([
            nn.Linear(in_features = self.num_units[0], out_features = self.
 ↪num_units[1], bias = self.usebias[0]),
```

```
            nn.Dropout(p = self.dropout[0], inplace = False),
            self.GetActivationLayer(0),
            nn.Linear(in_features = self.num_units[1], out_features = self.
↪num_units[2], bias = self.usebias[1]),
            nn.Dropout(p = self.dropout[1], inplace = False),
            self.GetActivationLayer(1),
            nn.Linear(in_features = self.num_units[2], out_features = self.
↪num_units[3], bias = self.usebias[2]),
            nn.Dropout(p = self.dropout[2], inplace = False),
            self.GetActivationLayer(2),
            nn.Linear(in_features = self.num_units[3], out_features = self.
↪K_Length, bias = self.usebias[3])
        ])

    def forward(self, x):
        output = x.view(x.size(0), -1)
        output = self.layers[0](output)
        output = self.layers[1](output)
        output = self.layers[2](output)
        output = self.layers[3](output)
        output = self.layers[4](output)
        output = self.layers[5](output)
        output = self.layers[6](output)
        output = self.layers[7](output)
        output = self.layers[8](output)
        output = self.layers[9](output)
        return output

    def GetActivationLayer(self, layer):
        Result = None
        if (self.activation[layer] == "relu"): #Not differentiable at 0.␣
↪Doesn't need Greedy layer-wise pretraining (Hinton) because it doesn't␣
↪suffer from vanishing gradient
            Result = nn.LeakyReLU(ReluAlpha) if ReluAlpha != 0 else nn.ReLU()␣
↪#alpha: Controls the angle of the negative slope
        elif (self.activation[layer] == "relu6"):
            Result = nn.ReLU6()
        elif (self.activation[layer] == "elu"): #Like ReLu but allows values to␣
↪be negative, so they can be centred around 0, also potential vanishing␣
↪gradient on the left side but doesn't matter
            Result = nn.ELU(alpha = EluAlpha) #alpha: Slope on the left side
        elif (self.activation[layer] == "tanh"): #Suffers from Vanishing␣
↪Gradient
            Result = nn.Tanh()
        elif (self.activation[layer] == "sigmoid"): #Suffers from Vanishing␣
↪Gradient
```

```
            Result = nn.Sigmoid() #Result isn't centred around 0. Maximum␣
↪derivative: 0.25
        return Result
print("Done")
```

Done

1.3 Dataset Functions

[6]:
```
def train_valid_test_split(X_Data, train_size, valid_size, Y_Data = None,␣
↪random_state = None, shuffle = True, stratify = None):
    """Split the dataset, optionally in a stratified manner, into a Train,␣
↪Validation and Test set"""

    if (type(train_size) == int and sum([train_size, valid_size]) >=␣
↪len(X_Data)) or (type(train_size) != int and sum([train_size, valid_size])␣
↪>= 1):
        raise ValueError(f"The train_size [{train_size}] + the valid_size␣
↪[{valid_size}] should sum up to less than 100% so that there's some␣
↪percentage left for the test set")

    TrainIdx, ValidTestIdx = train_test_split(np.arange(len(X_Data)),␣
↪train_size = train_size, shuffle = shuffle, stratify = stratify,␣
↪random_state = random_state)
    TrainX     = X_Data[TrainIdx]
    ValidTestX = X_Data[ValidTestIdx]
    if Y_Data is not None:
        TrainY     = Y_Data[TrainIdx]
        ValidTestY = Y_Data[ValidTestIdx]

    if type(train_size) != int: #For the 2nd split we need the validation␣
↪percent relative to the Valid/Test portion of the dataset alone
        test_size  = 1 - train_size - valid_size #Actual test size
        valid_size = 1 - (test_size / (valid_size + test_size)) #Relative (to␣
↪ValidTest) valid size
        test_size  = 1 - valid_size #Relative (to ValidTest) test size

    if Y_Data is not None:
        ValidX, TestX, ValidY, TestY = train_test_split(ValidTestX, ValidTestY,␣
↪train_size = valid_size, shuffle = shuffle, stratify =␣
↪stratify[ValidTestIdx] if stratify is not None else None, random_state =␣
↪random_state)
        return TrainX, ValidX, TestX, TrainY, ValidY, TestY
    else:
        ValidX, TestX = train_test_split(ValidTestX, train_size = valid_size,␣
↪shuffle = shuffle, stratify = stratify[ValidTestIdx] if stratify is not None␣
↪else None, random_state = random_state)
```

```
        return TrainX, ValidX, TestX
```

[40]:
```
def Scale(x_data, scaler_mean, scaler_sd, verbose = True):
    """Scale a Torch Tensor or Numpy Array to have zero mean and unit variance.
↪"""
    if isinstance(x_data, torch.Tensor):
        if (isinstance(scaler_mean, np.number) or isinstance(scaler_sd, np.
↪number)) and x_data.shape[1] != 1:
            if verbose:
                print("Info: Scaler is a scalar but X's observations are not.␣
↪Safely ignore this if you intended to normalise with scalar parameters.")
            return ((x_data - scaler_mean) / scaler_sd).float()
        else:
            return ((x_data - torch.from_numpy(scaler_mean)) / torch.
↪from_numpy(scaler_sd)).float()
    elif isinstance(x_data, np.ndarray):
        if verbose and (isinstance(scaler_mean, np.number) or␣
↪isinstance(scaler_sd, np.number)) and x_data.shape[1] != 1:
            print("Info: Scaler is a scalar but X's observations are not.␣
↪Safely ignore this if you intended to normalise with scalar parameters.")
        return ((x_data - scaler_mean) / scaler_sd).astype(np.float32)
    else:
        raise Exception("Cannot scale the variable because it is neither a␣
↪Torch Tensor nor a Numpy Array")
        return None

def UnScale(x_data, scaler_mean, scaler_sd, verbose = True):
    """Inverse the scaling of a Torch Tensor or Numpy Array that currently have␣
↪zero mean and unit variance."""
    if isinstance(x_data, torch.Tensor):
        if (isinstance(scaler_mean, np.number) or isinstance(scaler_sd, np.
↪number)) and x_data.shape[1] != 1:
            if verbose:
                print("Info: Scaler is a scalar but X's observations are not.␣
↪Safely ignore this if you intended to normalise with scalar parameters.")
            return ((x_data * scaler_sd) + scaler_mean).float()
        else:
            return ((x_data * torch.from_numpy(scaler_sd)) + torch.
↪from_numpy(scaler_mean)).float()
    elif isinstance(x_data, np.ndarray):
        if verbose and (isinstance(scaler_mean, np.number) or␣
↪isinstance(scaler_sd, np.number)) and x_data.shape[1] != 1:
            print("Info: Scaler is a scalar but X's observations are not.␣
↪Safely ignore this if you intended to normalise with scalar parameters.")
        return ((x_data * scaler_sd) + scaler_mean).astype(np.float32)
    else:
```

```
        raise Exception("Cannot unscale the variable because it is neither a␣
↪Torch Tensor nor a Numpy Array")
        return None
```

1.4 Optimisation Functions

[7]:
```
def ClassAccMulti(Targets, Preds, K):
    """Calculate the Class-Wise accuracy for a multi-class task"""
    return(np.mean([(Targets == k) == (Preds == k) for k in range(K)]))
```

[8]:
```
def AccCalculation(Y_Hat, Targets):
    """Calculate the Accuracy given the Actual values and Predictions for␣
↪Binary and Multiclass Classification."""
    if isinstance(Targets, torch.Tensor):
        Targets = Targets.cpu().numpy()

    if isinstance(Y_Hat, torch.Tensor):
        Y_Hat = Y_Hat.cpu().numpy()

    return np.mean(Y_Hat == Targets)
```

[9]:
```
def AUCCalculation(Targets, Y_Prob, Y_Hat, Verbose = True):
    """Calculate the Area Under the Receiver Operating Characteristic Curve␣
↪given the Actual values and Predictions for Binary and Multiclass␣
↪Classification using sklearn's roc_auc_score()."""
    if isinstance(Targets, torch.Tensor):
        Targets = Targets.cpu().numpy()

    if isinstance(Y_Prob, torch.Tensor):
        Y_Prob = Y_Prob.cpu().numpy()

    if isinstance(Y_Hat, torch.Tensor):
        Y_Hat = Y_Hat.cpu().numpy()

    try:
        CurMetric2 = roc_auc_score(Targets, Y_Prob, multi_class = "ovr",␣
↪average = 'weighted') #Calculating Weighted AUC #Cares for performance both␣
↪in Positives and Negatives (but may not fare well with heavy class imbalance)
    except Exception as exc:
        CurMetric2 = np.nan
        if Verbose:
            warnings.warn(f"\nAn error occurred in AUC calculation (probably␣
↪because of missing classes in the random batch of data?).\nThe error reads:␣
↪{exc}")
            print("AUC Warning. set(Targets):", list(set(Targets.reshape(-1))),␣
↪"set(Outputs): ", list(set(Y_Hat.reshape(-1))))
```

```
    return CurMetric2
```

```
[10]: def F1ScoreCalculation(Targets, Y_Hat):
          """Calculate the F1 score given the Actual values and Predictions for␣
      ↪Binary and Multiclass Classification using sklearn's f1_score()."""
          if isinstance(Targets, torch.Tensor):
              Targets = Targets.cpu().numpy()

          if isinstance(Y_Hat, torch.Tensor):
              Y_Hat = Y_Hat.cpu().numpy()

          try:
              CurMetric3 = f1_score(Targets, Y_Hat, average = 'weighted')␣
      ↪#Calculating Weighted F1 #Cares about balance between Precision and Recall␣
      ↪(Sensitivity)
          except Exception as exc:
              CurMetric3 = np.nan
              warnings.warn(f"\nAn error occurred in F1 score calculation (probably␣
      ↪because of missing classes in the random batch of data?).\nThe error reads:␣
      ↪{exc}")

          return CurMetric3
```

```
[11]: def PrintIterationMetrics(it, epochs, t0, train_loss, test_loss, first_metric,␣
      ↪first_metric_Name, second_metric, second_metric_Name, third_metric,␣
      ↪third_metric_Name, MaxTrainLossLen, MaxTestLossLen, MaxMetric1Len,␣
      ↪MaxMetric2Len, MaxMetric3Len):
          """Print information about the Current Epoch, Train/Test losses as well as␣
      ↪metrics and duration, and return the Max length of each metric viewed as a␣
      ↪string in order to keep a consistent text alignment amongst consecutive␣
      ↪epochs."""
          dt = datetime.now() - t0

          strTrainLoss = f"{train_loss:.4f}"
          strTestLoss = f"{test_loss:.4f}"
          strMetric1 = f'{first_metric:.3f}'
          strMetric2 = f'{second_metric:.3f}'
          strMetric3 = f'{third_metric:.3f}'
          if it == 0:
              MaxTrainLossLen = len(strTrainLoss)
              MaxTestLossLen = len(strTestLoss)
              MaxMetric1Len = len(strMetric1)
              MaxMetric2Len = len(strMetric2)
              MaxMetric3Len = len(strMetric3)
```

```
    print(f'Epoch {ZeroANumber(it+1, len(str(epochs)))}/{epochs}, Train Loss: {SpaceAString(strTrainLoss, MaxTrainLossLen)}, Test Loss: {SpaceAString(strTestLoss, MaxTestLossLen)} | {first_metric_Name}: {SpaceAString(strMetric1, MaxMetric1Len)}, {second_metric_Name}: {SpaceAString(strMetric2, MaxMetric1Len)}, {third_metric_Name}: {SpaceAString(strMetric3, MaxMetric1Len)}, Duration: {dt}')
    return MaxTrainLossLen, MaxTestLossLen, MaxMetric1Len, MaxMetric2Len, MaxMetric3Len
```

[12]:

```
def UpdateMetricsAndSaveModel(model, train_loss, test_loss, train_best_loss, test_best_loss, CurMetric1, Metric1, CurMetric2, Metric2, CurMetric3, Metric3):
    """if current model outperform's best model, save current model's state and update best performance metrics to reflect this model's."""
    if (test_loss < test_best_loss): #Saving the model if it outperforms previous iteration's model
        test_best_loss = test_loss
        train_best_loss = train_loss
        torch.save(model.state_dict(), f"model_dict.pt") #Saving Model's Dictionary

        if np.isfinite(CurMetric1) and CurMetric1 >= Metric1:
            Metric1 = CurMetric1
            Metric2 = CurMetric2
            Metric3 = CurMetric3
            torch.save(model.state_dict(), f"acc_model_dict.pt") #Saving Model's Dictionary
    return train_best_loss, test_best_loss, Metric1, Metric2, Metric3
```

[13]:

```
def PrintFinishingInformation(start_time, JustCalculateElapsedTime = False):
    """Calculate and print (and return) the elapsed time over all the training epochs."""
    elapsed_time = time.time() - start_time
    if not JustCalculateElapsedTime:
        FinishedOn = datetime.now(timezone('Europe/Athens')).strftime("%a, %Y-%m-%d %H:%M %Z %z")
        print("Done (" + FinishedOn + ") Elapsed time: " + str(round(elapsed_time, 1)) + " seconds")

    return elapsed_time
```

[14]:

```
def TrainModel(model, optimiser, criterion, X_Train, Y_Train):
    """Train by calculating the gradients and taking one step."""
    model.train() #Putting model in training mode so that things like dropout() are activated again
```

```
    optimiser.zero_grad() #Initialisation of the gradient of
    outputs = model(X_Train) #Getting the prediction using the forward␣
↪direction of the Neural Net

    loss = criterion(outputs, Y_Train) #Calculating the loss according to the␣
↪loss function
    loss.backward() #Calculating the Gradient Δ  of the loss function with␣
↪respect to the parameters

    optimiser.step() #Calculates and updates the parameters   using gradient␣
↪descent, as   =   -  *Δ

    return optimiser, outputs, loss
```

[15]:
```
def EvaluateModelFromPreds(criterion, Y_Prob, Targets, Verbose):
    """Use the forward direction of the model following with a␣
↪sigmoid+threshold or softmax+argmax for binary or multiclass classification␣
↪respectively, and calculate and return the predictions and evaluation␣
↪metrics."""
    with torch.no_grad(): #Making sure that we don't update the gradient␣
↪outside the training part
        loss_scalar = criterion(Y_Prob, Targets).item() #Calculating the loss␣
↪according to the loss function

        Y_Prob = nn.Softmax(dim = 1)(Y_Prob) #dim: every slice along dim will␣
↪sum to 1
        _, Y_Hat = torch.max(Y_Prob, 1) #Prediction. torch.max returns both max␣
↪(value) and argmax (index)

        CurMetric1, CurMetric2, CurMetric3 = GetCategoricalMetrics(Y_Prob,␣
↪Y_Hat, Targets, Verbose = Verbose)
        return loss_scalar, CurMetric1, CurMetric2, CurMetric3
```

[16]:
```
def EvalForwardPass(model, inputs, criterion = None, Targets = None): #This is␣
↪used at the very end on "Evaluation" Section. Unifies the forward pass, but␣
↪doesn't calculate loss/metrics like EvaluateModel() does as we need greater␣
↪granularity.
    """Use the forward direction of the model, potentially following with a␣
↪sigmoid+threshold or softmax+argmax for binary or multiclass classification␣
↪respectively."""

    if Targets is not None and criterion is None:
        warnings.warn(f"\nTargets are present but loss cannot be calculated␣
↪because criterion is None.")
```

```
    model.eval() #Putting model in evaluation mode so that things like␣
↪dropout() are deactivated
    with torch.no_grad(): #Making sure that we don't update the gradient␣
↪outside the training part
        Y_Prob = model(inputs) #Getting the prediction using the forward␣
↪direction of the Neural Net

        if Targets is not None:
            loss_scalar = criterion(Y_Prob, Targets).item() #Calculating the␣
↪loss according to the loss function

        Y_Prob = nn.Softmax(dim = 1)(Y_Prob) #dim: every slice along dim will␣
↪sum to 1
        _, Y_Hat = torch.max(Y_Prob, 1) #Prediction. torch.max returns both max␣
↪(value) and argmax (index)

    if Targets is not None:
        return Y_Prob, Y_Hat, loss_scalar
    else:
        return Y_Prob, Y_Hat
```

[17]:
```
def FixFormatAndDTypes(device, Inputs, Targets):
    """Ensure that the Inputs and Targets are Torch Tensors and of the correct␣
↪shape and dtype before returning them."""
    if isinstance(Inputs, np.ndarray):
        Inputs = torch.from_numpy(Inputs)
    if isinstance(Targets, np.ndarray):
        Targets = torch.from_numpy(Targets)

    Inputs = Inputs.to(device)
    Targets = Targets.to(device)

    Targets = Targets.long()

    return Inputs, Targets
```

1.5 Evaluation Functions

[18]:
```
def plot_confusion_matrix(cm, classes, normalise = False, title = 'Confusion␣
↪matrix', colourmap = plt.cm.Blues):
    """Plot the Confusion Matrix object returned by sklearn's␣
↪confusion_matrix() and normalise it if normalise==True."""
    plt.grid(False)
    if normalise:
        print('Confusion matrix')
        print(cm)
```

```
        cm = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]
        plt.imshow(cm, interpolation = 'nearest', cmap = colourmap)
        plt.clim(0.0, 1.0)
    else:
        plt.imshow(cm, interpolation = 'nearest', cmap = colourmap)
    plt.title(title)
    with warnings.catch_warnings():
        warnings.simplefilter("ignore")
        plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation = 45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalise else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment = "center",
                 color = "white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.show()

def PlotHistory(Train_History, Test_History = None, Key = "Loss", figsize =␣
 ↪(12, 8), MajorLineStyle = "--", MinorLineStyle = ":", MajorLines = 10,␣
 ↪MinorInbetweenLinesEvery = 4, test_alpha = 1.0):
    """Plot a juxtaposition of a Train and Test metric (parametrised by Key),␣
 ↪usually 'Loss'"""
    fig, ax = plt.subplots(figsize = figsize)
    plt.plot(Train_History, label = f"Train {Key}")
    if (Test_History is not None):
        plt.plot(Test_History, label = f"Test {Key}", alpha = test_alpha)
    xfrom, xto = ax.get_xlim()
    yfrom, yto = ax.get_ylim()
    ax.xaxis.set_major_locator(MultipleLocator(int(np.ceil((xto-xfrom)/
 ↪MajorLines))))
    ax.yaxis.set_major_locator(MultipleLocator((yto-yfrom)/MajorLines))
    ax.xaxis.set_minor_locator(AutoMinorLocator(MinorInbetweenLinesEvery))
    ax.yaxis.set_minor_locator(AutoMinorLocator(MinorInbetweenLinesEvery))
    ax.grid(which = 'major', color='#FFFFFF', linestyle = MajorLineStyle)
    ax.grid(which = 'minor', color='#CCCCCC', linestyle = MinorLineStyle)
    plt.legend()
    plt.show()
    return None
```

```
def PlotAllMetrics(Titles, TrainMetrics, TestMetrics = None, figsize = [19,␣
↪13], test_alpha = 1.0):
    fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize = figsize)
    ax1.set_title(Titles[0])
    ax1.plot(TrainMetrics[0], label = f"Train")
    if TestMetrics is not None:
        ax1.plot(TestMetrics[0], label = f"Test", alpha = test_alpha)
    ax1.legend()
    ax2.set_title(Titles[1])
    ax2.plot(TrainMetrics[1], label = f"Train")
    if TestMetrics is not None:
        ax2.plot(TestMetrics[1], label = f"Test", alpha = test_alpha)
    ax2.legend()
    ax3.set_title(Titles[2])
    ax3.plot(TrainMetrics[2], label = f"Train")
    if TestMetrics is not None:
        ax3.plot(TestMetrics[2], label = f"Test", alpha = test_alpha)
    ax3.legend()
    ax4.set_title(Titles[3])
    ax4.plot(TrainMetrics[3], label = f"Train")
    if TestMetrics is not None:
        ax4.plot(TestMetrics[3], label = f"Test", alpha = test_alpha)
    ax4.legend()
    plt.show()
    return None
```

[19]:

```
def GetCategoricalMetrics(Y_Prob, Y_Hat, Targets, Verbose = True):
    """Calculate Categorical variable metrics (Accuracy, Area Under the Curve,␣
↪F1 score) given the class Probability vector (binary) / matrix (multiclass),␣
↪the class index (0 to K-1), and the Actual values."""
    test_Acc = AccCalculation(Y_Hat, Targets)
    test_AUC = AUCCalculation(Targets, Y_Prob, Y_Hat, Verbose = Verbose)
    test_F1 = F1ScoreCalculation(Targets, Y_Hat)

    return test_Acc, test_AUC, test_F1
```

[20]:

```
def PlotCategoricalMetrics(Y_Hat, Targets, ClassNames, normalise, figsize =␣
↪None):
    """Plots the Confusion matrix given the class index (0 to K-1), and the␣
↪Actual values."""
    PrevFigSize = plt.rcParams['figure.figsize']
    plt.rcParams['figure.figsize'] = figsize if figsize is not None else␣
↪PrevFigSize

    cm = confusion_matrix(Targets, Y_Hat)
    plot_confusion_matrix(cm, ClassNames, normalise = normalise)
```

```
    plt.rcParams['figure.figsize'] = PrevFigSize
```

[21]:
```
def EvalPredict(model, device, test_loader_or_X_Test):
    """Use EvalForwardPass() to use the forward direction of the model and␣
↪return the Y_probability, Y_Hat, and respective Y given X."""
    Preds_prob = []
    Preds = []
    Targets = []

    for inputs, targets in tqdm(test_loader_or_X_Test, total =␣
↪len(test_loader_or_X_Test), leave = False):
        inputs, targets = FixFormatAndDTypes(device, inputs, targets)
        outputs_prob, outputs = EvalForwardPass(model, inputs, criterion =␣
↪None, Targets = None)

        Preds_prob.append(outputs_prob)
        Preds.append(outputs)
        Targets.append(targets)

    Preds_prob = torch.cat(Preds_prob).cpu().numpy()
    Preds = torch.cat(Preds).cpu().numpy()
    Targets = torch.cat(Targets).cpu().numpy()

    del inputs
    del outputs_prob
    del outputs
    del targets
    if IS_GPU_AVAILABLE:
        torch.cuda.empty_cache()

    return Preds_prob, Preds, Targets
```

1.6 Gradient Descent Functions

[22]:
```
def batch_gd(model, device, criterion, optimiser, scheduler, train_loader,␣
↪test_loader, epochs, PrintInfoEverynEpochs, train_best_loss, test_best_loss,␣
↪BestMetric1, BestMetric2, BestMetric3, Verbose = True):
    """Use the Train, Evaluation, Metrics calculation and printing functions to␣
↪train a model over certain epochs taking steps in every batch and keeping␣
↪track of the metrics on each epoch as well as the overall best metrics"""
    MaxTrainLossLen, MaxTestLossLen, MaxMetric1Len, MaxMetric2Len,␣
↪MaxMetric3Len = None, None, None, None, None #For output text formatting

    start_time = time.time() #To calculate the duration of the whole learning␣
↪procedure
    model.to(device) #If there is a GPU, let's ensure model is sent to the GPU ␣
↪
```

```
    #Initialising the Metrics
    train_losses, test_losses, train_metric1s, train_metric2s, train_metric3s,␣
 ↪test_metric1s, test_metric2s, test_metric3s = np.repeat(np.nan, epochs), np.
 ↪repeat(np.nan, epochs), np.repeat(np.nan, epochs), np.repeat(np.nan,␣
 ↪epochs), np.repeat(np.nan, epochs), np.repeat(np.nan, epochs), np.repeat(np.
 ↪nan, epochs), np.repeat(np.nan, epochs)

    for it in range(epochs):
        t0 = datetime.now() #To calculate the duration of the current epoch
        ProbsTrain = []
        TargetsTrain = []
        Probs = []
        Targets = []

        #== Training ==#
        for inputs, targets in train_loader:
#          for inputs, targets in tqdm(train_loader, total = len(train_loader),␣
 ↪leave = False):
            inputs, targets = FixFormatAndDTypes(device, inputs, targets)␣
 ↪#Making sure we have Tensors of the correct Format and Data Type
            optimiser, outputs, loss = TrainModel(model, optimiser, criterion,␣
 ↪inputs, targets) #Training the model on Train set
            #This loss includes dropout() and stuff as it was not done under␣
 ↪model.eval()
            ProbsTrain.append(outputs.cpu())
            TargetsTrain.append(targets.cpu())
            del inputs, targets, outputs

        ProbsTrain   = torch.cat(ProbsTrain)
        TargetsTrain = torch.cat(TargetsTrain)

        train_loss, CurTrainMetric1, CurTrainMetric2, CurTrainMetric3 =␣
 ↪EvaluateModelFromPreds(criterion, ProbsTrain, TargetsTrain, Verbose =␣
 ↪Verbose) #Evaluating the model on Train set

        #== Evaluation ==#
        for inputs, targets in test_loader:
#          for inputs, targets in tqdm(test_loader, total = len(test_loader),␣
 ↪leave = False):
            inputs, targets = FixFormatAndDTypes(device, inputs, targets)␣
 ↪#Making sure we have Tensors of the correct Format and Data Type
            Y_Prob, _ = EvalForwardPass(model, inputs)
            Probs.append(Y_Prob.cpu())
            Targets.append(targets.cpu())
```

```
            del inputs, targets, Y_Prob#, Y_Hat

        Probs   = torch.cat(Probs)
        Targets = torch.cat(Targets)

        test_loss, CurMetric1, CurMetric2, CurMetric3 =␣
↪EvaluateModelFromPreds(criterion, Probs, Targets, Verbose = Verbose)␣
↪#Evaluating the model on Evaluation set

        if np.any(np.logical_or(torch.isinf(Probs).cpu().numpy(), torch.
↪isnan(Probs).cpu().numpy())):
            print(f"!Predictions contain infinities ({np.mean(np.logical_or(np.
↪isinf(Probs), np.isnan(Probs))) * 100:.2f}%); Stopping!")
            break

        if np.logical_or(np.isinf(test_loss), np.isnan(test_loss)):
            print("!Loss is Infinite; Stopping!")
            break

        if scheduler is not None:
            if list(scheduler.keys())[0].lower() == "Plateau".lower():
                scheduler[list(scheduler.keys())[0]].step(test_loss)
            elif list(scheduler.keys())[0].lower() == "StepLR".lower():
                scheduler[list(scheduler.keys())[0]].step()

        #Saving the metrics
        train_losses[it], train_metric1s[it], train_metric2s[it],␣
↪train_metric3s[it], test_losses[it], test_metric1s[it], test_metric2s[it],␣
↪test_metric3s[it] = train_loss, CurTrainMetric1, CurTrainMetric2,␣
↪CurTrainMetric3, test_loss, CurMetric1, CurMetric2, CurMetric3

        if (it + 1) % PrintInfoEverynEpochs == 0 or it == 0 or it == epochs - 1:
            MaxTrainLossLen, MaxTestLossLen, MaxMetric1Len, MaxMetric2Len,␣
↪MaxMetric3Len = PrintIterationMetrics( #Prints Iteration Metrics
                it, epochs, t0, train_loss, test_loss,
                CurMetric1, "Acc",
                CurMetric2, "AUC",
                CurMetric3, "F1" ,
                MaxTrainLossLen, MaxTestLossLen,
                MaxMetric1Len, MaxMetric2Len, MaxMetric3Len
            )

        train_best_loss, test_best_loss, BestMetric1, BestMetric2, BestMetric3␣
↪= UpdateMetricsAndSaveModel(model, train_loss, test_loss, train_best_loss,␣
↪test_best_loss, CurMetric1, BestMetric1, CurMetric2, BestMetric2,␣
↪CurMetric3, BestMetric3) #Updating Metrics and Saving the model if it␣
↪outperforms previous iteration's model
```

```
    elapsed_time = PrintFinishingInformation(start_time,␣
↪JustCalculateElapsedTime = False) #Prints finishing information
    return train_losses, test_losses, train_best_loss, test_best_loss,␣
↪train_metric1s, train_metric2s, train_metric3s, test_metric1s,␣
↪test_metric2s, test_metric3s, CurMetric1, CurMetric2, CurMetric3,␣
↪BestMetric1, BestMetric2, BestMetric3, elapsed_time
```

2 Data

[23]:
```
#Configuring the basic structure of our current directory
path_root = f"{os.getcwd()}"
path_data = f"{Path(path_root).absolute()}/Data"
path_models = f"{path_root}/Models"
print(path_root)
print(path_data)
print(path_models)
```

```
D:\GiannisM\Downloads\Exercises\Fiver\100. frotribe FFNN Multiclass
Classification
D:\GiannisM\Downloads\Exercises\Fiver\100. frotribe FFNN Multiclass
Classification/Data
D:\GiannisM\Downloads\Exercises\Fiver\100. frotribe FFNN Multiclass
Classification/Models
```

[56]:
```
############################
## Data Hyperparameters ####
Seed            = 42
batch_size      = 256
TrainPerc       = 0.8
ValidPerc       = 0.1
TestPerc        = 1 - TrainPerc - ValidPerc
CustomNAString = None
#############################

########################
### Reading the Data ###
Classes = ['drinking water', 'smoking', 'standing up', 'sit down', 'mopping the␣
↪floor', 'sweeping the floor', 'walking', 'unknown']
XY_DF = pd.read_csv(f"{path_data}/11-16 (1).csv", header = None)
display(XY_DF)
########################

####################
### Handling NAs ###
NBeforeCustomNADrop = None
DroppedCustomNARows = None
```

```python
NBeforeNADrop = len(XY_DF)
XY_DF = XY_DF.dropna()
DroppedNARows = NBeforeNADrop - len(XY_DF)
if DroppedNARows > 0:
    print(f"Dropped NA rows count: {DroppedNARows} (out of {NBeforeNADrop})")

if CustomNAString is not None:
    NBeforeCustomNADrop = len(XY_DF)
    XY_DF = XY_DF.replace(CustomNAString, np.nan, regex = False).dropna()
    DroppedCustomNARows = NBeforeCustomNADrop - len(XY_DF)
    if DroppedCustomNARows > 0:
        print(f"Dropped custom NA rows count: {DroppedCustomNARows} (out of {NBeforeCustomNADrop})", )
if DroppedNARows > 0 or (DroppedCustomNARows is not None and DroppedCustomNARows > 0):
    print()
####################

######################################
### Creating Train/Valid/Test sets ###
X_Data = XY_DF.iloc[:, 1:].values.astype(np.float32)
Y_Data = XY_DF.iloc[:, 0 ].values.astype(int).squeeze() - 1

Labels_Data  = np.array([Classes[y] for y in Y_Data])

#==Stratified Split
TrainIndx, ValidIndx, TestIndx = train_valid_test_split(np.arange(X_Data.shape[0]), train_size = TrainPerc, valid_size = ValidPerc, Y_Data = None, random_state = Seed, shuffle = True, stratify = Y_Data)
X_Train      =      X_Data[TrainIndx]
Y_Train      =      Y_Data[TrainIndx]
Labels_Train = Labels_Data[TrainIndx]
X_Valid      =      X_Data[ValidIndx]
Y_Valid      =      Y_Data[ValidIndx]
Labels_Valid = Labels_Data[ValidIndx]
X_Test       =      X_Data[TestIndx ]
Y_Test       =      Y_Data[TestIndx ]
Labels_Test  = Labels_Data[TestIndx ]
######################################

########################
### Scaling the Data ###
# scaler = LoadVariable(f"{SaveFolder}/scaler") #After loading a model with a different scaler we need to re-run this using the newly loaded scaler.
# print("\n\n\n!!!!!!!!!!!!\nDEBUGGING:\nScaling with SaveFolder scaler!!!\n!!!!!!!!!!!!\n\n\n")
```

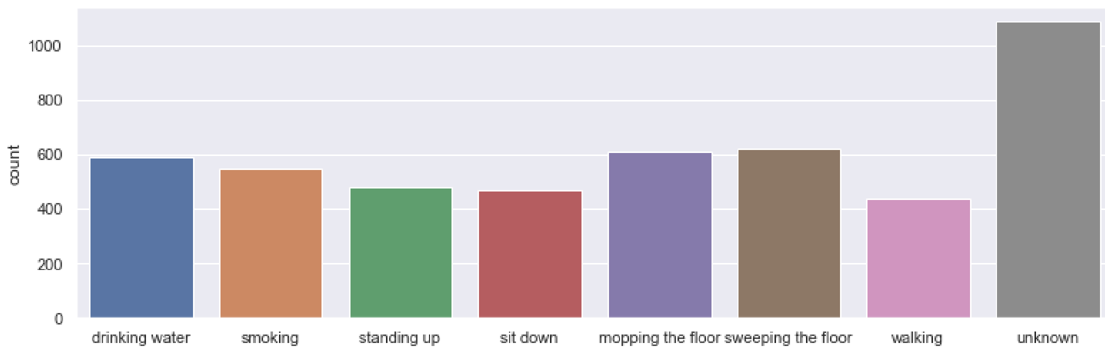
```python
# if os.path.exists(f"{path_models}/scaler"):
#     print("!!\n!! Using saved scaler.\n!!\n")
#     scaler = LoadVariable(f"{path_models}/scaler")
# else:
scaler = StandardScaler(with_mean = True, with_std = True).fit(X_Train)
SaveVariable(scaler, f"{path_models}/scaler")

scaler_mean = scaler.mean_
scaler_sd = scaler.scale_
scaler_mean_sd = (scaler_mean, scaler_sd)

#Numpy takes care of the broadcasting automatically
X_Train      = Scale(X_Train, *scaler_mean_sd)
X_Valid      = Scale(X_Valid, *scaler_mean_sd)
X_Test       = Scale(X_Test , *scaler_mean_sd)
########################

# ###################################
# ### Creating Dataset/Dataloader ###
Dataset_Train = TensorDataset(torch.from_numpy(X_Train), torch.
 ↪from_numpy(Y_Train))
Dataset_Valid = TensorDataset(torch.from_numpy(X_Valid), torch.
 ↪from_numpy(Y_Valid))
Dataset_Test  = TensorDataset(torch.from_numpy(X_Test ), torch.
 ↪from_numpy(Y_Test ))

Loader_Train = torch.utils.data.DataLoader(
    dataset = Dataset_Train,
    batch_size = batch_size,
    shuffle = True,
    pin_memory = True

)
Loader_Valid = torch.utils.data.DataLoader(
    dataset = Dataset_Valid,
    batch_size = batch_size,
    shuffle = False,
    pin_memory = True
)
Loader_Test = torch.utils.data.DataLoader(
    dataset = Dataset_Test,
    batch_size = batch_size,
    shuffle = False,
    pin_memory = True
)
# ###################################
```

```
################################
### Extracting Information ###
tmpx, tmpy = next(iter(Loader_Valid))
K_Length, O_Length, N, D_Length, H1, W1 = len(set(Y_Train.squeeze().tolist())), 1, len(Y_Train), tmpx.shape[1], 1, 1
print(f"X_Data.shape : {(len(X_Data ), *tmpx.shape[1:])} min: {X_Data.min():.2f} max: {X_Data.max():.2f} Y_Data.shape : {Y_Data.shape }")
print(f"X_Train.shape: {(len(X_Train), *tmpx.shape[1:])} min: {X_Train.min():.2f} max: {X_Train.max():.2f} Y_Train.shape: {Y_Train.shape}")
print(f"X_Valid.shape: {(len(X_Valid), *tmpx.shape[1:])} min: {X_Valid.min():.2f} max: {X_Valid.max():.2f} Y_Valid.shape: {Y_Valid.shape}")
print(f"X_Test.shape : {(len(X_Test ), *tmpx.shape[1:])} min: {X_Test.min():.2f} max: {X_Test.max():.2f} Y_Test.shape : {Y_Test.shape }")
print(f"K_Length: {K_Length}")
print(f"N: {N} H1: {H1} W1: {W1} D_Length: {D_Length}")

plt.rcParams['figure.figsize'] = [13, 4]
print(f"\nClasses:")
sns.countplot(x = [Classes[int(y)] for y in sorted(Y_Data.squeeze())])
plt.show()

CountData = sorted(Counter(Y_Data.squeeze()).items())
FreqKeys = [kv[0] for kv in CountData]
FreqData = np.array([kv[1] for kv in CountData]) / len(Y_Data) * 100
for i in range(len(FreqData)):
    print(f"{FreqKeys[i]}: {SpaceAString(f'{FreqData[i]:.2f}', MaxLength = 5)}% [{SpaceAString(f'{CountData[i][1]}', MaxLength = 5)}]")

print(f"\nClasses [TRAIN]:")
sns.countplot(x = [Classes[int(y)] for y in sorted(Y_Train.squeeze())])
plt.show()

print(f"\nClasses [Valid]:")
sns.countplot(x = [Classes[int(y)] for y in sorted(Y_Valid.squeeze())])
plt.show()

print(f"\nClasses [Test ]:")
sns.countplot(x = [Classes[int(y)] for y in sorted(Y_Test.squeeze() )])
plt.show()

CountTrain = sorted(Counter(Y_Train.squeeze()).items())
FreqTrain  = np.array([kv[1] for kv in CountTrain]) / len(Y_Train) * 100
CountValid = sorted(Counter(Y_Valid.squeeze()).items())
FreqValid  = np.array([kv[1] for kv in CountValid]) / len(Y_Valid) * 100
CountTest  = sorted(Counter(Y_Test.squeeze()).items())
FreqTest   = np.array([kv[1] for kv in CountTest ]) / len(Y_Test ) * 100
```

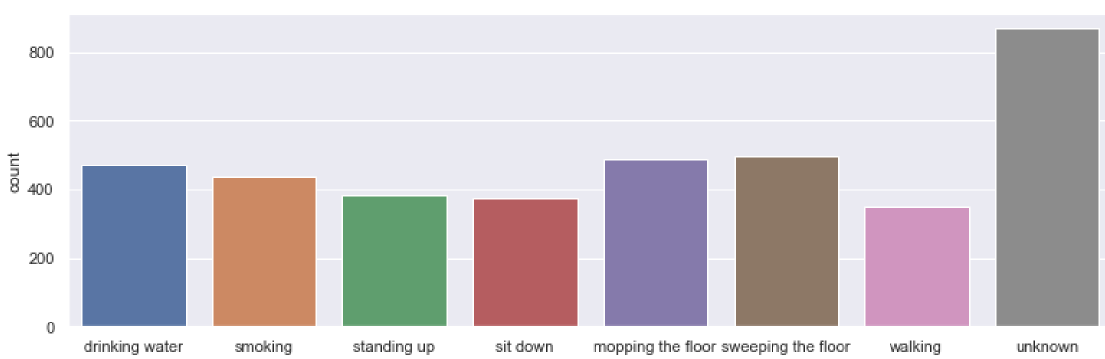
```python
for i in range(len(FreqKeys)):
    print(f"{FreqKeys[i]}: Train {SpaceAString(f'{FreqTrain[i]:.2f}', MaxLength = 5)}% [{SpaceAString(f'{CountTrain[i][1]}', MaxLength = 5)}], Valid {SpaceAString(f'{FreqValid[i]:.2f}', MaxLength = 5)}% [{SpaceAString(f'{CountValid[i][1]}', MaxLength = 5)}], Test {SpaceAString(f'{FreqTest[i]:.2f}', MaxLength = 5)}% [{SpaceAString(f'{CountTest[i][1]}', MaxLength = 5)}]")
###############################

print("\nDone")
```

```
      0         1         2         3         4         5         6    \
0        1  0.147911  0.133120  0.025052 -0.070056 -0.060896 -0.121567
1        1  0.096198 -0.066323 -0.180289 -0.175182 -0.108132 -0.080090
2        1 -0.500452 -0.502092 -0.475572 -0.425861 -0.389736 -0.402447
3        1 -0.307718 -0.320017 -0.344007 -0.309607 -0.289414 -0.333693
4        1 -0.103104 -0.148786 -0.189899 -0.093382  0.025787  0.158881
…     …       …         …         …         …         …         …
4830     8 -0.064396 -0.178618 -0.246944 -0.049874  0.179200  0.079402
4831     8  0.117115  0.049382  0.134941  0.065425  0.011479  0.212872
4832     8  0.213987  0.145185  0.117739  0.140440  0.290152  0.308540
4833     8  0.088052  0.105103 -0.000214  0.090305  0.115749  0.151577
4834     8  0.115733  0.043828  0.224749  0.098849  0.132058  0.092996

          7         8         9     …  1191  1192  1193  1194  1195  1196  \
0     -0.098642 -0.011251 -0.005818  …   0.0   0.0   0.0   0.0   0.0   0.0
1      0.250950 -0.125172  0.287903  …   0.0   0.0   0.0   0.0   0.0   0.0
2     -0.495722 -0.568901 -0.576617  …   0.0   0.0   0.0   0.0   0.0   0.0
3     -0.414461 -0.556066 -0.711506  …   0.0   0.0   0.0   0.0   0.0   0.0
4      0.220521  0.142476  0.063623  …   0.0   0.0   0.0   0.0   0.0   0.0
…          …         …         …  …   …    …    …    …    …
4830   0.045606  0.273751  0.224829  …   0.0   0.0   0.0   0.0   0.0   0.0
4831   0.394126  0.320238  0.258048  …   0.0   0.0   0.0   0.0   0.0   0.0
4832   0.079455  0.088747  0.010922  …   0.0   0.0   0.0   0.0   0.0   0.0
4833   0.101945  0.152081 -0.005337  …   0.0   0.0   0.0   0.0   0.0   0.0
4834   0.096625  0.142984  0.051117  …   0.0   0.0   0.0   0.0   0.0   0.0

      1197  1198  1199  1200
0      0.0   0.0   0.0   0.0
1      0.0   0.0   0.0   0.0
2      0.0   0.0   0.0   0.0
3      0.0   0.0   0.0   0.0
4      0.0   0.0   0.0   0.0
…     …    …    …    …
4830   0.0   0.0   0.0   0.0
4831   0.0   0.0   0.0   0.0
4832   0.0   0.0   0.0   0.0
```

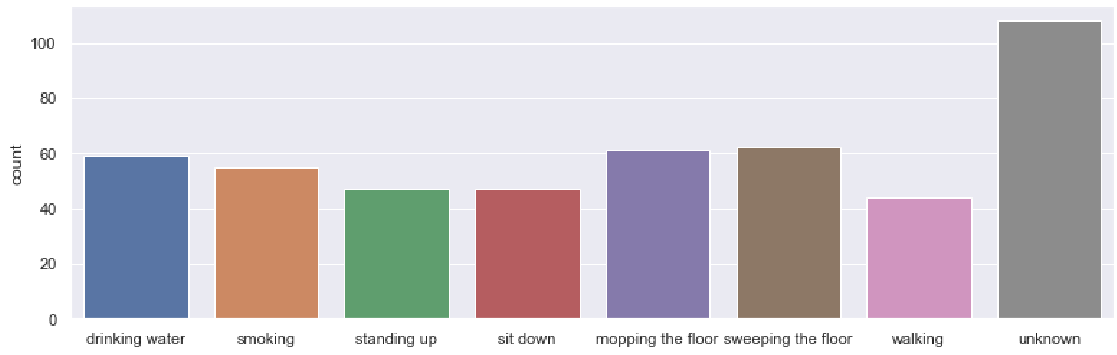
```
4833   0.0   0.0   0.0   0.0
4834   0.0   0.0   0.0   0.0

[4835 rows x 1201 columns]
X_Data.shape : (4835, 1200) min: -40.75 max: 49.11 Y_Data.shape : (4835,)
X_Train.shape: (3868, 1200) min: -48.25 max: 51.08 Y_Train.shape: (3868,)
X_Valid.shape: (483, 1200) min: -54.38 max: 36.09 Y_Valid.shape: (483,)
X_Test.shape : (484, 1200) min: -26.58 max: 37.10 Y_Test.shape : (484,)
K_Length: 8
N: 3868 H1: 1 W1: 1 D_Length: 1200

Classes:
```



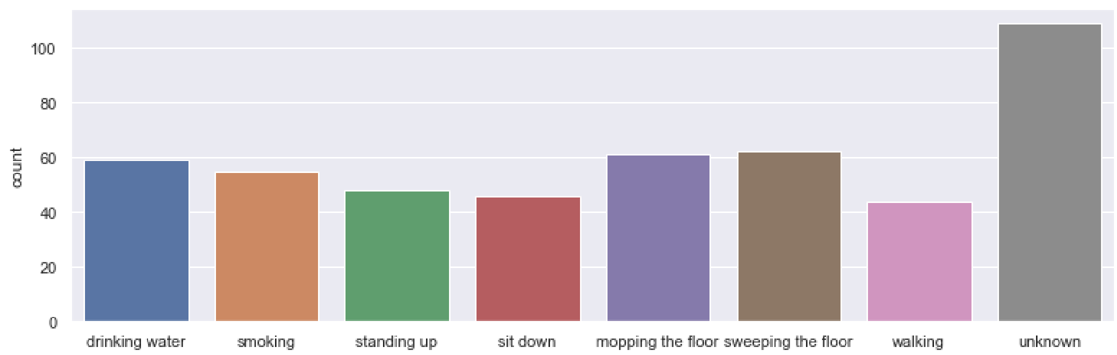

```
0: 12.18% [  589]
1: 11.35% [  549]
2:  9.87% [  477]
3:  9.64% [  466]
4: 12.57% [  608]
5: 12.86% [  622]
6:  9.06% [  438]
7: 22.46% [ 1086]

Classes [TRAIN]:
```



```
Classes [Valid]:
```




```
Classes [Test ]:
```




```
0: Train 12.18% [  471], Valid 12.22% [   59], Test 12.19% [   59]
1: Train 11.35% [  439], Valid 11.39% [   55], Test 11.36% [   55]
2: Train  9.88% [  382], Valid  9.73% [   47], Test  9.92% [   48]
3: Train  9.64% [  373], Valid  9.73% [   47], Test  9.50% [   46]
4: Train 12.56% [  486], Valid 12.63% [   61], Test 12.60% [   61]
5: Train 12.87% [  498], Valid 12.84% [   62], Test 12.81% [   62]
6: Train  9.05% [  350], Valid  9.11% [   44], Test  9.09% [   44]
7: Train 22.47% [  869], Valid 22.36% [  108], Test 22.52% [  109]

Done
```

3 Neural Network

3.1 Hyper Parameters

[155]:
```
conv_input_size = (H1, W1, D_Length)
input_size = np.prod(conv_input_size)
output_size = K_Length
print("conv_input_size: " + str(conv_input_size) + ", input_size: " +␣
↪str(input_size) + ", D_Length: " + str(D_Length) + ", output_size: " +␣
↪str(output_size))
hn1 = D_Length

ReluAlpha = 0
EluAlpha = 0

layer_type = ["conv", "stridedconv", "convpool", "dense", "dense"]
NUM = 2
num_units = [hn1, 32, 64, 128, 256, 512]
num_units = [num_units[0]] + [n_unit * NUM for n_unit in num_units[1:]]
activation = ["relu", "relu", "relu", "relu", "relu"]
dropout = [0.3, 0.3, 0.3, 0.4, 0.4]
usebias = [False, False, False, True, True] + [True]
###
batchnorm_momentum = [0.7, 0.7, 0.7, None, None]
###
conv_filter_size   = 5
conv_mode          = ["same" if l in ["conv", "stridedconv", "convpool"] else␣
↪None for l in layer_type]
conv_stride        = 2
conv_dilation      = 1
###
conv_pool_size     = 2
conv_pool_dilation = 1 #Dilation on Pooling layer
conv_pool_stride   = conv_pool_size #Stride on Pooling: a 2x2 pooling with␣
↪stride 2 will half the size of an image
conv_pool_padding  = 0 #Used when the input size is not an integer multiple of␣
↪the kernel size, so usually just 0.
###
conv_padding, conv_output_size = [2, 2, 2, None, None], [28, 28, 14, 36864,␣
↪*num_units[-2:]] #18432 #36864
print()
print("nPadding:", conv_padding)
print("X's Dims:", conv_output_size)
print("num_units", num_units)
print(f"\nbatch_size: {batch_size}")

l2_lamda = 0.35
```

```
mu = 0.99 #Momentum
```

```
conv_input_size: (48, 48, 1), input_size: 2304, D_Length: 1, output_size: 3

nPadding: [2, 2, 2, None, None]
X's Dims: [28, 28, 14, 50176, 512, 1024]
num_units [1, 64, 128, 256, 512, 1024]

batch_size: 256
```

[161]:
```
#Regular
conv_input_size = X_Train[0].shape if X_Train is not None else X_Data[0].shape␣
 ↪#Also used in RNNs

input_size = np.prod(conv_input_size)
output_size = np.prod(O_Length)

print("conv_input_size: " + str(conv_input_size) + ", input_size: " +␣
 ↪str(input_size) + ", D_Length: " + str(D_Length) + ", output_size: " +␣
 ↪str(output_size))

hn1 = D_Length

ReluAlpha = 0 #0.01 def leakyRelu
EluAlpha = 0.8

layer_type     = ["dense", "dense", "dense"]
###
NUM = 1
num_units = [hn1, 128, 256, 512]
num_units = num_units if len(num_units) == 1 else [num_units[0]] +␣
 ↪[num_units[LayerIndex+1] if layer_type[LayerIndex] in ["transfenc",␣
 ↪"customtransfenc"] else num_units[LayerIndex+1] * NUM for LayerIndex in␣
 ↪range(len(layer_type))] #MAKING SURE TRANSFORMER INPUT AND OUTPUT num_units␣
 ↪ARE THE SAME AFTER NUM IS APPLIED
###
activation = ["relu"] + ["relu"] * (len(layer_type)-1) #None, "relu6" "relu",␣
 ↪"elu", "softplus", "tanh", "sigmoid"
###
dropout    = [0.3] * 1 + [0.3] * (len(layer_type)-1)
###
usebias    = [True] * len(layer_type) + [True]
###
l2_lamda = 0.25
mu = 0.99 #Momentum
###
print()
```

```
print("num_units", num_units)
print(f"\nbatch_size: {batch_size}")
```

```
conv_input_size: (1200,), input_size: 1200, D_Length: 1200, output_size: 1
doFlatten= False

num_units [1200, 128, 256, 512]

batch_size: 256
```

3.2 Optimisation

3.2.1 Structure

[162]:
```
print(device)
Debug = False

model = Net(K_Length, num_units, activation, dropout, usebias)
# if device != "cpu":
#     model = nn.DataParallel(model)
print(model)

#Initialising the Metrics
train_losses, train_metric1s, train_metric2s, train_metric3s, valid_losses,
 ↪valid_metric1s, valid_metric2s, valid_metric3s = np.array([]), np.array([]),
 ↪np.array([]), np.array([]), np.array([]), np.array([]), np.array([]), np.
 ↪array([])
train_best_loss, valid_best_loss, valid_best_metric1, valid_best_metric2,
 ↪valid_best_metric3 = np.Inf, np.Inf, 0, np.nan, np.nan
```

```
cuda
Net(
  (layers): ModuleList(
    (0): Linear(in_features=1200, out_features=128, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.3, inplace=False)
    (3): Linear(in_features=128, out_features=256, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.3, inplace=False)
    (6): Linear(in_features=256, out_features=512, bias=True)
    (7): ReLU()
    (8): Dropout(p=0.3, inplace=False)
    (9): Linear(in_features=512, out_features=8, bias=True)
  )
)
```

[163]:
```
print("conv_input_size:", conv_input_size, "\n")
```

```
summary(model, input_size = [1, D_Length], device = device, verbose = 1,␣
↪col_names = ["kernel_size", "input_size", "output_size", "num_params",␣
↪"mult_adds"], depth = 3)

if IS_GPU_AVAILABLE:
    torch.cuda.empty_cache()
```

```
conv_input_size: (1200,)

================================================================================
================================================================================
=====
Layer (type:depth-idx)                   Kernel Shape              Input Shape
Output Shape              Param #                   Mult-Adds
================================================================================
================================================================================
=====
Net                                      --                        --
--                        --                        --
 ModuleList: 1-1                        --                        --
--                        --                        --
     Linear: 2-1                        [1200, 128]               [1, 1200]
[1, 128]                  153,728                   153,728
     ReLU: 2-2                          --                        [1, 128]
[1, 128]                  --                        --
     Dropout: 2-3                       --                        [1, 128]
[1, 128]                  --                        --
     Linear: 2-4                        [128, 256]                [1, 128]
[1, 256]                  33,024                    33,024
     ReLU: 2-5                          --                        [1, 256]
[1, 256]                  --                        --
     Dropout: 2-6                       --                        [1, 256]
[1, 256]                  --                        --
     Linear: 2-7                        [256, 512]                [1, 256]
[1, 512]                  131,584                   131,584
     ReLU: 2-8                          --                        [1, 512]
[1, 512]                  --                        --
     Dropout: 2-9                       --                        [1, 512]
[1, 512]                  --                        --
     Linear: 2-10                       [512, 8]                  [1, 512]
[1, 8]                    4,104                     4,104
================================================================================
================================================================================
=====
Total params: 322,440
Trainable params: 322,440
Non-trainable params: 0
Total mult-adds (M): 0.32
```

```
================================================================================
================================================================================
=====
Input size (MB): 0.00
Forward/backward pass size (MB): 0.01
Params size (MB): 1.29
Estimated Total Size (MB): 1.30
================================================================================
================================================================================
=====
```

[164]:
```
#Setting the Loss Function and Optimisation technique
criterion = nn.CrossEntropyLoss() #Using Categorical Cross Entropy loss function
print(criterion.__class__.__name__)
print("Multiclass Classification")

learning_rate = 5e-4
optimiser = torch.optim.AdamW(model.parameters(), lr = learning_rate, betas =␣
 ↪(mu, 0.999), weight_decay = l2_lamda, amsgrad = False)
```

```
CrossEntropyLoss
Multiclass Classification
```

[165]:
```
# for prm_grp in optimiser.param_groups:
#     prm_grp['lr'] = 1e-4
# #     prm_grp['weight_decay'] = 0.1
```

3.2.2 Stochastic Gradient Descent (Dataset)

[167]:
```
Epochs = int(50)
PrintInfoEverynEpochs = 1
scheduler = None
# scheduler = torch.optim.lr_scheduler.StepLR(optimiser, step_size = Epochs //␣
 ↪10 if Epochs > 10 else 3, gamma = 0.8)

new_train_losses, new_valid_losses, train_best_loss, valid_best_loss,␣
 ↪new_train_metric1s, new_train_metric2s, new_train_metric3s,␣
 ↪new_valid_metric1s, new_valid_metric2s, new_valid_metric3s,\
Metric1, Metric2, Metric3, valid_best_metric1, valid_best_metric2,␣
 ↪valid_best_metric3, elapsed_time = \
    batch_gd(model, device, criterion, optimiser, scheduler, Loader_Train,␣
 ↪Loader_Valid, epochs = Epochs, PrintInfoEverynEpochs = PrintInfoEverynEpochs,
             train_best_loss = train_best_loss, test_best_loss =␣
 ↪valid_best_loss, BestMetric1 = valid_best_metric1, BestMetric2 =␣
 ↪valid_best_metric2, BestMetric3 = valid_best_metric3,
             Verbose = False
    )
```

```
train_losses, valid_losses = np.append(train_losses, new_train_losses), np.
 ↪append(valid_losses, new_valid_losses)
train_metric1s, train_metric2s, train_metric3s = np.append(train_metric1s,␣
 ↪new_train_metric1s), np.append(train_metric2s, new_train_metric2s), np.
 ↪append(train_metric3s, new_train_metric3s)
valid_metric1s, valid_metric2s, valid_metric3s = np.append(valid_metric1s,␣
 ↪new_valid_metric1s), np.append(valid_metric2s, new_valid_metric2s), np.
 ↪append(valid_metric3s, new_valid_metric3s)
train_loss, valid_loss, valid_metric1, valid_metric2, valid_metric3 =␣
 ↪LastNonNan(new_train_losses), LastNonNan(new_valid_losses), Metric1,␣
 ↪Metric2, Metric3

print(f'\ntrain_best_loss: {train_best_loss:.5f}, valid_best_loss:␣
 ↪{valid_best_loss:.5f}, Acc: {valid_metric1:.5f}, AUC: {valid_metric2:.5f},␣
 ↪F1: {valid_metric3:.5f}')

if IS_GPU_AVAILABLE:
    torch.cuda.empty_cache()
if IsWindows:
    winsound.PlaySound('SystemExit', winsound.SND_NOSTOP)
```

```
Epoch 01/50, Train Loss: 0.0633, Test Loss: 1.3704 | Acc: 0.907, AUC: 0.987, F1:
0.906, Duration: 0:00:00.156999
Epoch 02/50, Train Loss: 0.0608, Test Loss: 1.3698 | Acc: 0.907, AUC: 0.987, F1:
0.906, Duration: 0:00:00.148500
Epoch 03/50, Train Loss: 0.0604, Test Loss: 1.3692 | Acc: 0.911, AUC: 0.986, F1:
0.910, Duration: 0:00:00.146000
Epoch 04/50, Train Loss: 0.0555, Test Loss: 1.3688 | Acc: 0.911, AUC: 0.986, F1:
0.910, Duration: 0:00:00.144500
Epoch 05/50, Train Loss: 0.0556, Test Loss: 1.3681 | Acc: 0.909, AUC: 0.986, F1:
0.908, Duration: 0:00:00.145000
Epoch 06/50, Train Loss: 0.0553, Test Loss: 1.3660 | Acc: 0.915, AUC: 0.986, F1:
0.915, Duration: 0:00:00.147000
Epoch 07/50, Train Loss: 0.0550, Test Loss: 1.3648 | Acc: 0.915, AUC: 0.986, F1:
0.915, Duration: 0:00:00.158000
Epoch 08/50, Train Loss: 0.0517, Test Loss: 1.3654 | Acc: 0.913, AUC: 0.986, F1:
0.913, Duration: 0:00:00.148000
Epoch 09/50, Train Loss: 0.0543, Test Loss: 1.3657 | Acc: 0.913, AUC: 0.985, F1:
0.913, Duration: 0:00:00.159500
Epoch 10/50, Train Loss: 0.0520, Test Loss: 1.3660 | Acc: 0.913, AUC: 0.985, F1:
0.913, Duration: 0:00:00.154000
Epoch 11/50, Train Loss: 0.0532, Test Loss: 1.3666 | Acc: 0.911, AUC: 0.985, F1:
0.911, Duration: 0:00:00.147999
Epoch 12/50, Train Loss: 0.0471, Test Loss: 1.3669 | Acc: 0.913, AUC: 0.985, F1:
0.913, Duration: 0:00:00.148501
Epoch 13/50, Train Loss: 0.0492, Test Loss: 1.3676 | Acc: 0.911, AUC: 0.985, F1:
0.910, Duration: 0:00:00.148000
```

Epoch 14/50, Train Loss: 0.0486, Test Loss: 1.3693 | Acc: 0.905, AUC: 0.985, F1: 0.904, Duration: 0:00:00.151498
Epoch 15/50, Train Loss: 0.0549, Test Loss: 1.3698 | Acc: 0.907, AUC: 0.985, F1: 0.906, Duration: 0:00:00.151500
Epoch 16/50, Train Loss: 0.0502, Test Loss: 1.3704 | Acc: 0.907, AUC: 0.984, F1: 0.905, Duration: 0:00:00.144498
Epoch 17/50, Train Loss: 0.0449, Test Loss: 1.3707 | Acc: 0.907, AUC: 0.984, F1: 0.906, Duration: 0:00:00.152000
Epoch 18/50, Train Loss: 0.0493, Test Loss: 1.3698 | Acc: 0.905, AUC: 0.984, F1: 0.904, Duration: 0:00:00.150500
Epoch 19/50, Train Loss: 0.0482, Test Loss: 1.3689 | Acc: 0.907, AUC: 0.983, F1: 0.906, Duration: 0:00:00.146001
Epoch 20/50, Train Loss: 0.0491, Test Loss: 1.3692 | Acc: 0.907, AUC: 0.983, F1: 0.906, Duration: 0:00:00.144500
Epoch 21/50, Train Loss: 0.0470, Test Loss: 1.3706 | Acc: 0.905, AUC: 0.983, F1: 0.904, Duration: 0:00:00.146000
Epoch 22/50, Train Loss: 0.0427, Test Loss: 1.3716 | Acc: 0.903, AUC: 0.983, F1: 0.902, Duration: 0:00:00.147500
Epoch 23/50, Train Loss: 0.0487, Test Loss: 1.3719 | Acc: 0.903, AUC: 0.983, F1: 0.902, Duration: 0:00:00.147999
Epoch 24/50, Train Loss: 0.0398, Test Loss: 1.3702 | Acc: 0.907, AUC: 0.983, F1: 0.906, Duration: 0:00:00.145501
Epoch 25/50, Train Loss: 0.0410, Test Loss: 1.3689 | Acc: 0.909, AUC: 0.983, F1: 0.909, Duration: 0:00:00.149502
Epoch 26/50, Train Loss: 0.0420, Test Loss: 1.3679 | Acc: 0.909, AUC: 0.983, F1: 0.909, Duration: 0:00:00.148498
Epoch 27/50, Train Loss: 0.0341, Test Loss: 1.3675 | Acc: 0.909, AUC: 0.984, F1: 0.908, Duration: 0:00:00.151000
Epoch 28/50, Train Loss: 0.0438, Test Loss: 1.3674 | Acc: 0.907, AUC: 0.983, F1: 0.906, Duration: 0:00:00.147498
Epoch 29/50, Train Loss: 0.0446, Test Loss: 1.3674 | Acc: 0.907, AUC: 0.982, F1: 0.906, Duration: 0:00:00.153000
Epoch 30/50, Train Loss: 0.0431, Test Loss: 1.3685 | Acc: 0.907, AUC: 0.982, F1: 0.906, Duration: 0:00:00.144500
Epoch 31/50, Train Loss: 0.0428, Test Loss: 1.3689 | Acc: 0.907, AUC: 0.982, F1: 0.906, Duration: 0:00:00.149500
Epoch 32/50, Train Loss: 0.0413, Test Loss: 1.3716 | Acc: 0.907, AUC: 0.981, F1: 0.906, Duration: 0:00:00.150500
Epoch 33/50, Train Loss: 0.0344, Test Loss: 1.3737 | Acc: 0.907, AUC: 0.981, F1: 0.906, Duration: 0:00:00.151000
Epoch 34/50, Train Loss: 0.0388, Test Loss: 1.3728 | Acc: 0.907, AUC: 0.981, F1: 0.906, Duration: 0:00:00.151000
Epoch 35/50, Train Loss: 0.0310, Test Loss: 1.3710 | Acc: 0.907, AUC: 0.981, F1: 0.906, Duration: 0:00:00.150500
Epoch 36/50, Train Loss: 0.0331, Test Loss: 1.3694 | Acc: 0.905, AUC: 0.981, F1: 0.904, Duration: 0:00:00.146002
Epoch 37/50, Train Loss: 0.0317, Test Loss: 1.3682 | Acc: 0.913, AUC: 0.981, F1: 0.912, Duration: 0:00:00.151000

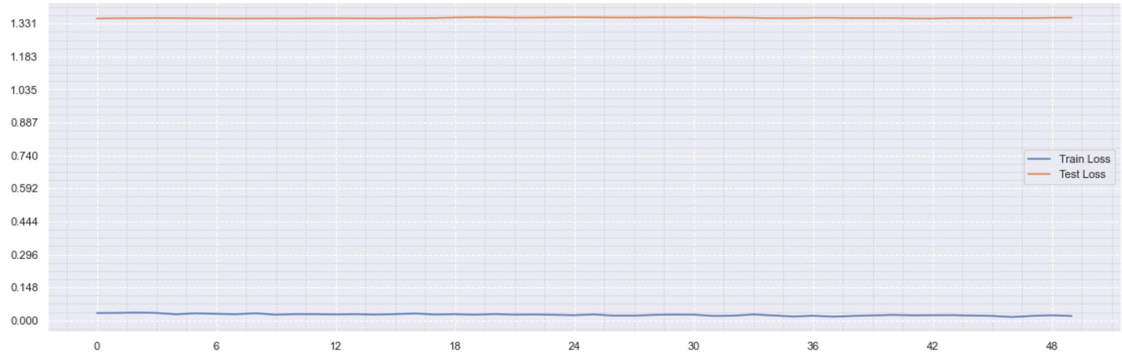
```
Epoch 38/50, Train Loss: 0.0417, Test Loss: 1.3676 | Acc: 0.911, AUC: 0.980, F1:
0.910, Duration: 0:00:00.148000
Epoch 39/50, Train Loss: 0.0368, Test Loss: 1.3671 | Acc: 0.911, AUC: 0.979, F1:
0.910, Duration: 0:00:00.147999
Epoch 40/50, Train Loss: 0.0324, Test Loss: 1.3665 | Acc: 0.915, AUC: 0.978, F1:
0.914, Duration: 0:00:00.153003
Epoch 41/50, Train Loss: 0.0357, Test Loss: 1.3665 | Acc: 0.915, AUC: 0.978, F1:
0.914, Duration: 0:00:00.151998
Epoch 42/50, Train Loss: 0.0309, Test Loss: 1.3676 | Acc: 0.915, AUC: 0.978, F1:
0.914, Duration: 0:00:00.150000
Epoch 43/50, Train Loss: 0.0287, Test Loss: 1.3697 | Acc: 0.907, AUC: 0.980, F1:
0.905, Duration: 0:00:00.148000
Epoch 44/50, Train Loss: 0.0317, Test Loss: 1.3717 | Acc: 0.905, AUC: 0.980, F1:
0.903, Duration: 0:00:00.149500
Epoch 45/50, Train Loss: 0.0326, Test Loss: 1.3707 | Acc: 0.907, AUC: 0.980, F1:
0.905, Duration: 0:00:00.155002
Epoch 46/50, Train Loss: 0.0275, Test Loss: 1.3670 | Acc: 0.917, AUC: 0.980, F1:
0.916, Duration: 0:00:00.149998
Epoch 47/50, Train Loss: 0.0276, Test Loss: 1.3664 | Acc: 0.915, AUC: 0.981, F1:
0.915, Duration: 0:00:00.148500
Epoch 48/50, Train Loss: 0.0255, Test Loss: 1.3667 | Acc: 0.915, AUC: 0.980, F1:
0.915, Duration: 0:00:00.147502
Epoch 49/50, Train Loss: 0.0329, Test Loss: 1.3671 | Acc: 0.913, AUC: 0.980, F1:
0.912, Duration: 0:00:00.156498
Epoch 50/50, Train Loss: 0.0300, Test Loss: 1.3672 | Acc: 0.913, AUC: 0.980, F1:
0.912, Duration: 0:00:00.151502
Done (Sun, 2022-09-11 19:42 EEST +0300) Elapsed time: 7.5 seconds

train_best_loss: 0.05500, valid_best_loss: 1.36480, Acc: 0.91304, AUC: 0.98049,
F1: 0.91242
```

[150]:
```
#Loading the best trained model (in case the last one was overfitted)
model.load_state_dict(torch.load("model_dict.pt"))
model.eval()
train_loss = train_best_loss
valid_loss = valid_best_loss
valid_metric1 = valid_best_metric1
valid_metric2 = valid_best_metric2
valid_metric3 = valid_best_metric3
```

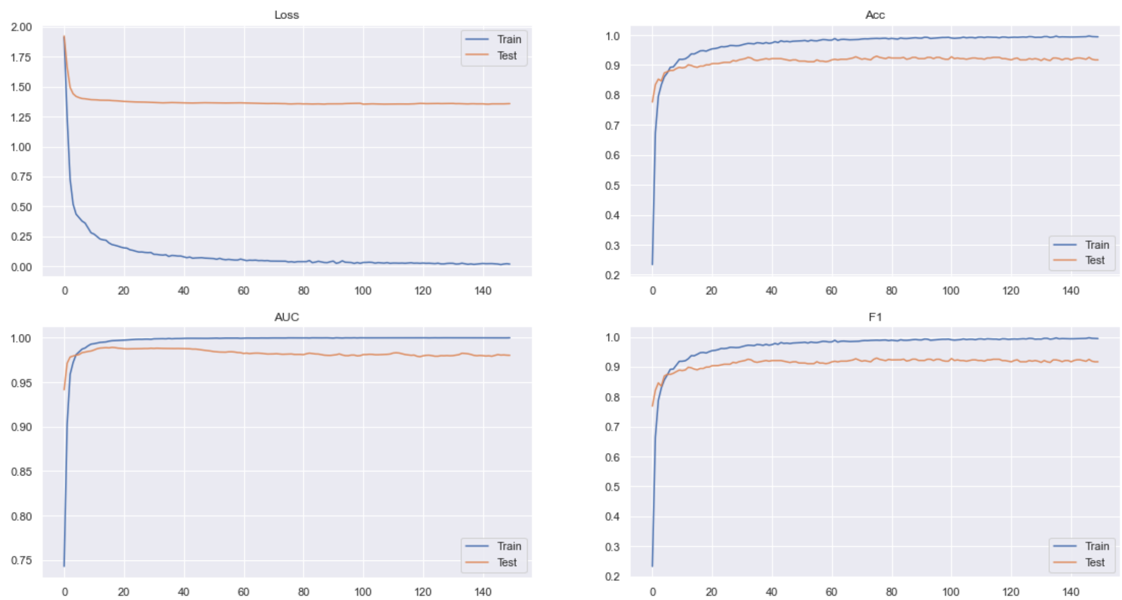
4 Evaluation

4.1 Training Metrics

[184]:
```
PlotHistory(train_losses[-(50):], valid_losses[-(50):], figsize=(19, 6),␣
↪test_alpha = 1)
```



[185]:
```
PlotAllMetrics(["Loss", "Acc", "AUC", "F1"],
               [train_losses, train_metric1s, train_metric2s, train_metric3s],
               TestMetrics = [valid_losses, valid_metric1s, valid_metric2s,␣
 ↪valid_metric3s],
               figsize = [19, 10], test_alpha = 0.90)
```



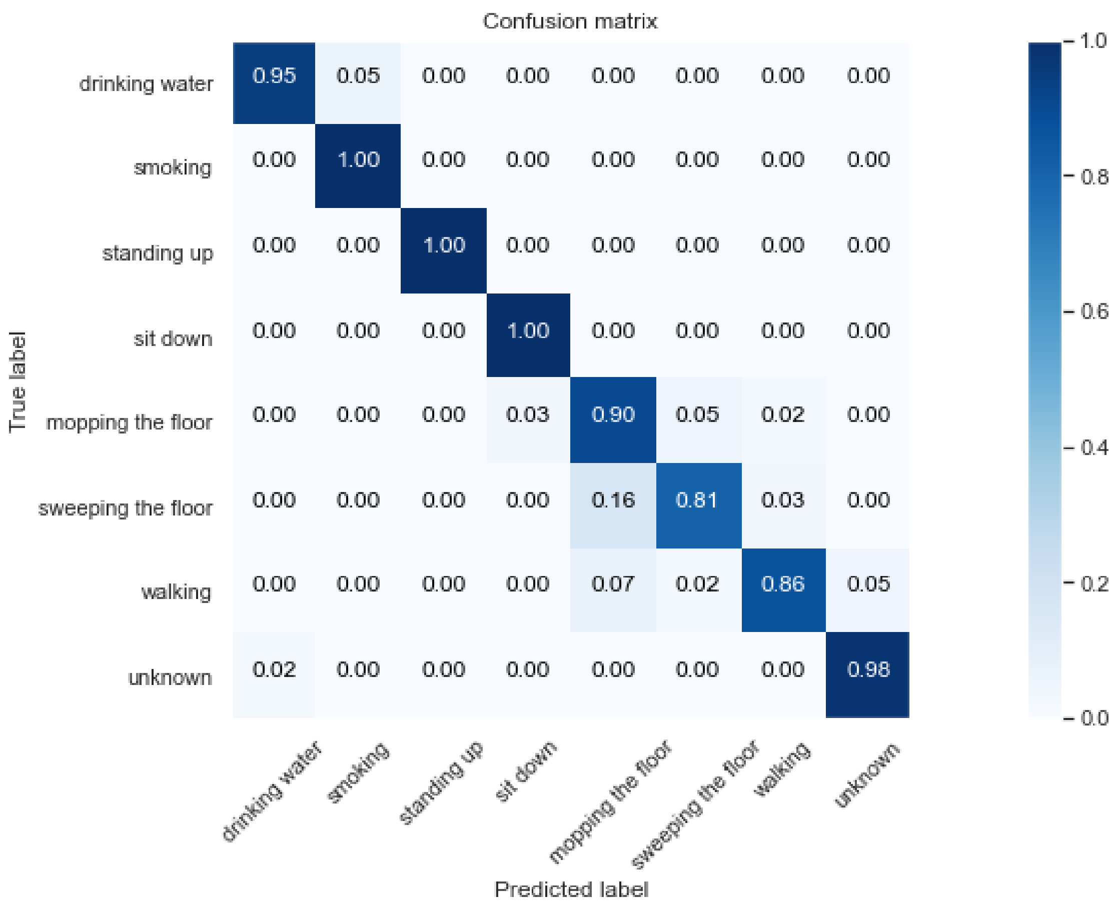
4.2 Multiclass Classification

[195]:
```
Labels = Classes
with torch.no_grad(): #Making sure that we don't update the gradient outside␣
 ↪the training part
        model.eval() #Putting model in evaluation mode so that things like␣
 ↪dropout() are deactivated
```

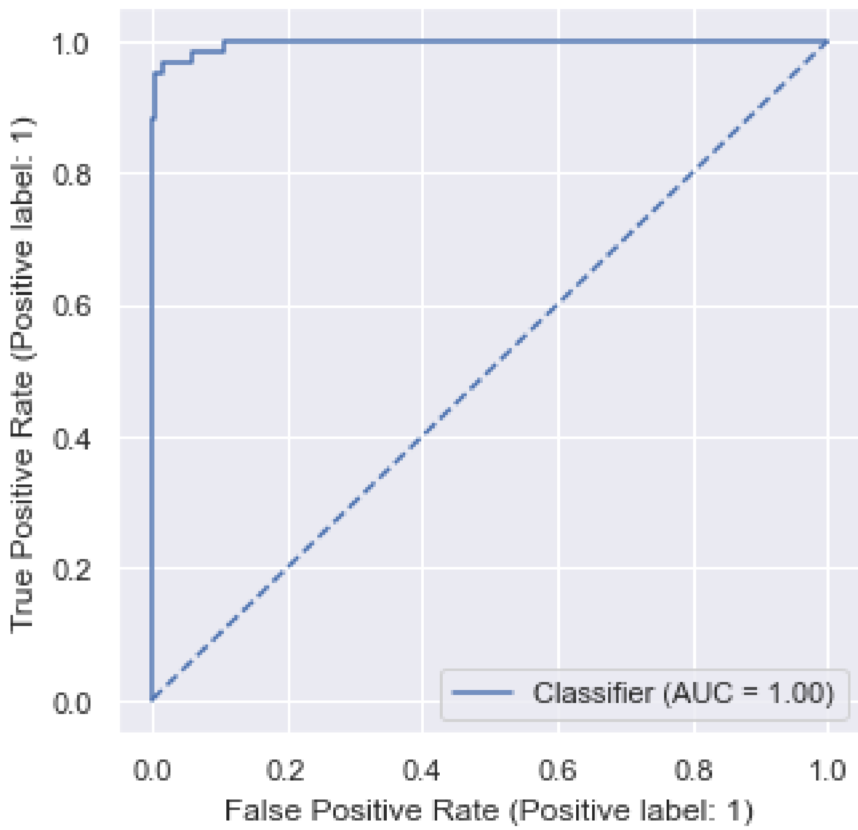
```python
        model.to(device) #Moving the model to the appropriate device (CPU/CUDA/
 ↪etc.)

        #Using the Forward direction of the model to get the Predictions (also␣
 ↪returning the corresponding Targets in case there's suffling and X_Test␣
 ↪isn't indexed the same)
        Preds_prob, Preds, Targets = EvalPredict(model, device, Loader_Test)

test_Acc, test_AUC, test_F1 = GetCategoricalMetrics(Preds_prob, Preds, Targets)
print(f'Sample-wise Acc: {test_Acc * 100:.2f}%, AUC: {test_AUC:.2f}, F1:␣
 ↪{test_F1:.2f}')
print(f"Class-wise  Acc: {ClassAccMulti(Targets, Preds, K_Length) * 100:.2f}%,␣
 ↪Recall: {recall_score(Targets, Preds, average = 'weighted'):.3f}, Precision:␣
 ↪{precision_score(Targets, Preds, average = 'weighted'):.3f}\n") #'micro',␣
 ↪'macro', 'weighted', 'samples'

#Viewing the overall Categorical metrics and Plotting the Confusion Matrix
PlotCategoricalMetrics(Preds, Targets, Labels, normalise = True, figsize = [17,␣
 ↪6.5])
print("")

PrevFigSize = plt.rcParams['figure.figsize']
plt.rcParams['figure.figsize'] = [5, 5]
for k in range(K_Length):
    PredClass = Preds   == k
    TrueClass = Targets == k
    print(f"Class {Classes[k]}. Sample-Wise Acc: {np.mean(TrueClass ==␣
 ↪PredClass):.3f}, Recall: {recall_score(TrueClass, PredClass):.3f}, Precision:
 ↪ {precision_score(TrueClass, PredClass):.3f}, F1: {f1_score(TrueClass,␣
 ↪PredClass):.3f}")
    RocCurveDisplay.from_predictions(TrueClass, Preds_prob[:, k])
    plt.plot(np.linspace(0, 1, num = 20), np.linspace(0, 1, num = 20), 'b--')
    plt.show()
    print()
plt.rcParams['figure.figsize'] = PrevFigSize
#Sample-wise Acc: 94.01%, AUC: 0.99, F1: 0.94
```

```
Sample-wise Acc: 94.01%, AUC: 0.99, F1: 0.94
Class-wise  Acc: 98.50%, Recall: 0.940, Precision: 0.942

Confusion matrix
[[ 56   3   0   0   0   0   0   0]
 [  0  55   0   0   0   0   0   0]
 [  0   0  48   0   0   0   0   0]
 [  0   0   0  46   0   0   0   0]
 [  0   0   0   2  55   3   1   0]
```

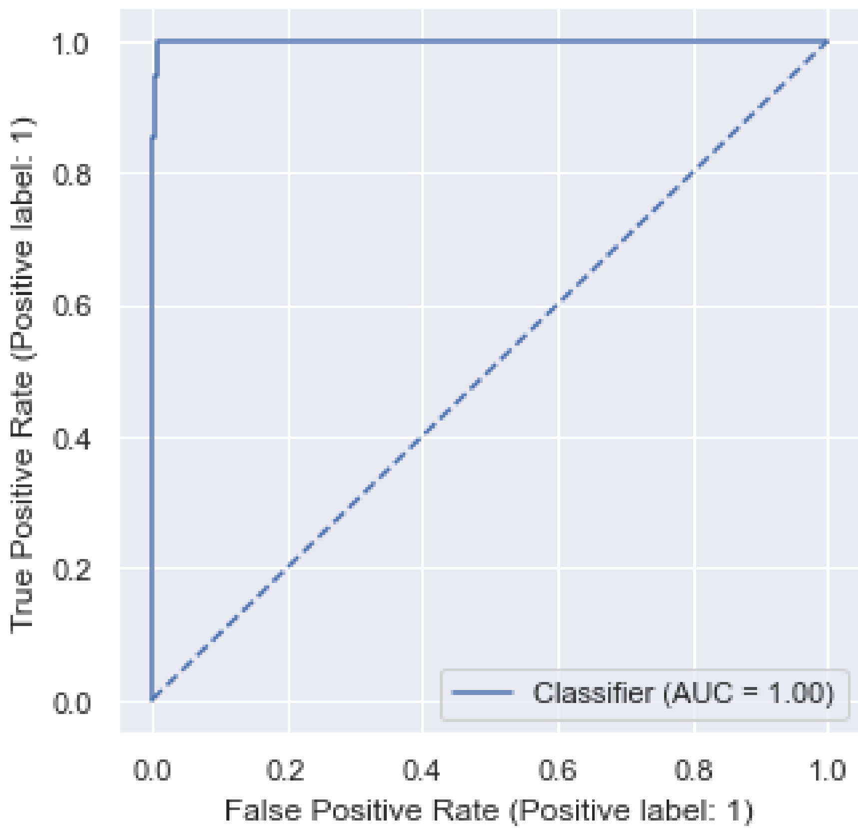
```
[  0   0   0   0  10  50   2   0]
[  0   0   0   0   3   1  38   2]
[  2   0   0   0   0   0   0 107]]
```



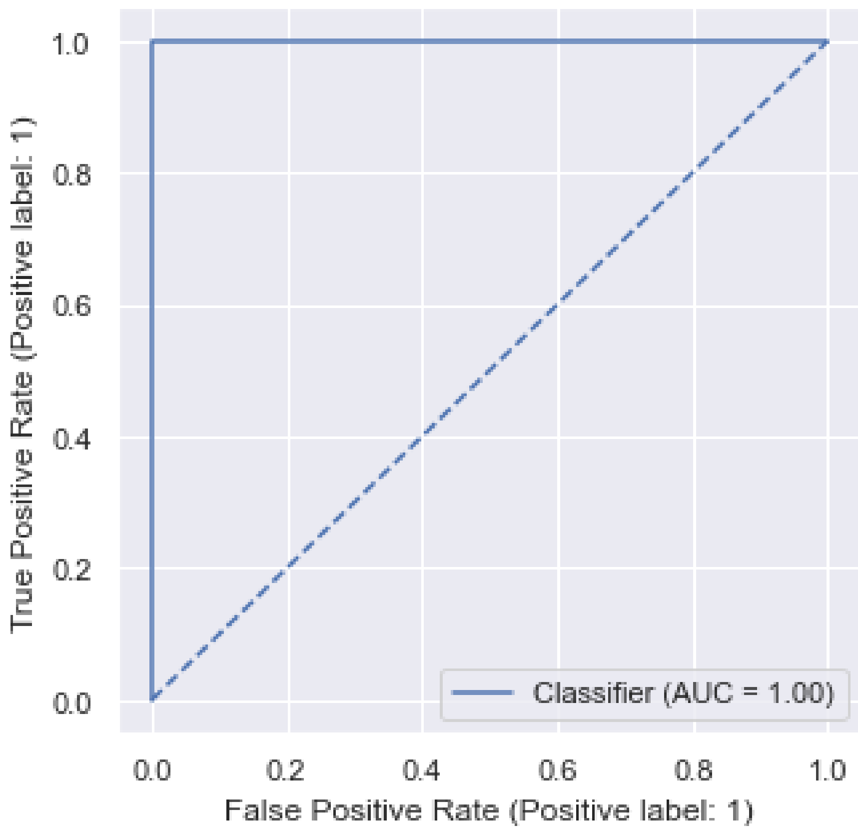

```
Class drinking water. Sample-Wise Acc: 0.990, Recall: 0.949, Precision: 0.966,
F1: 0.957
```



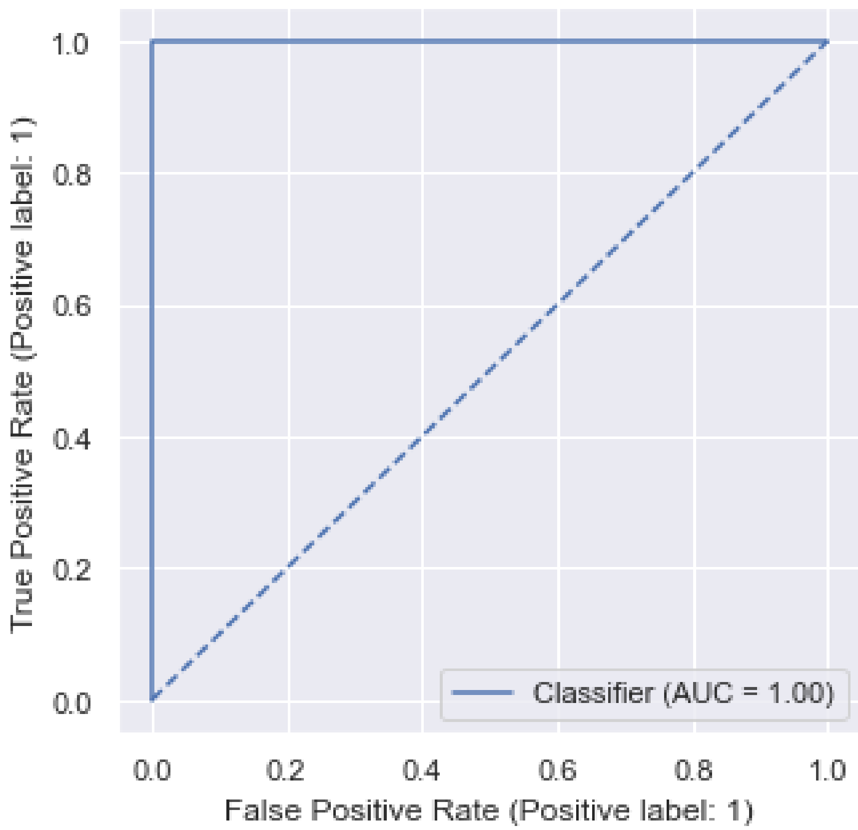

```
Class smoking. Sample-Wise Acc: 0.994, Recall: 1.000, Precision: 0.948, F1:
0.973
```



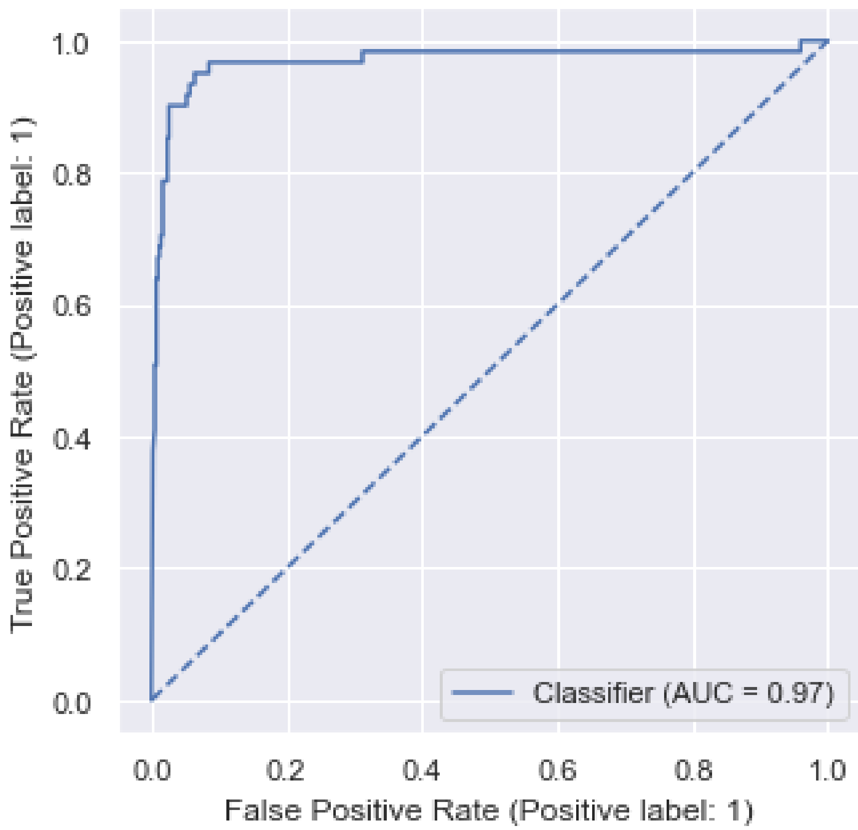
`Class standing up. Sample-Wise Acc: 1.000, Recall: 1.000, Precision: 1.000, F1: 1.000`



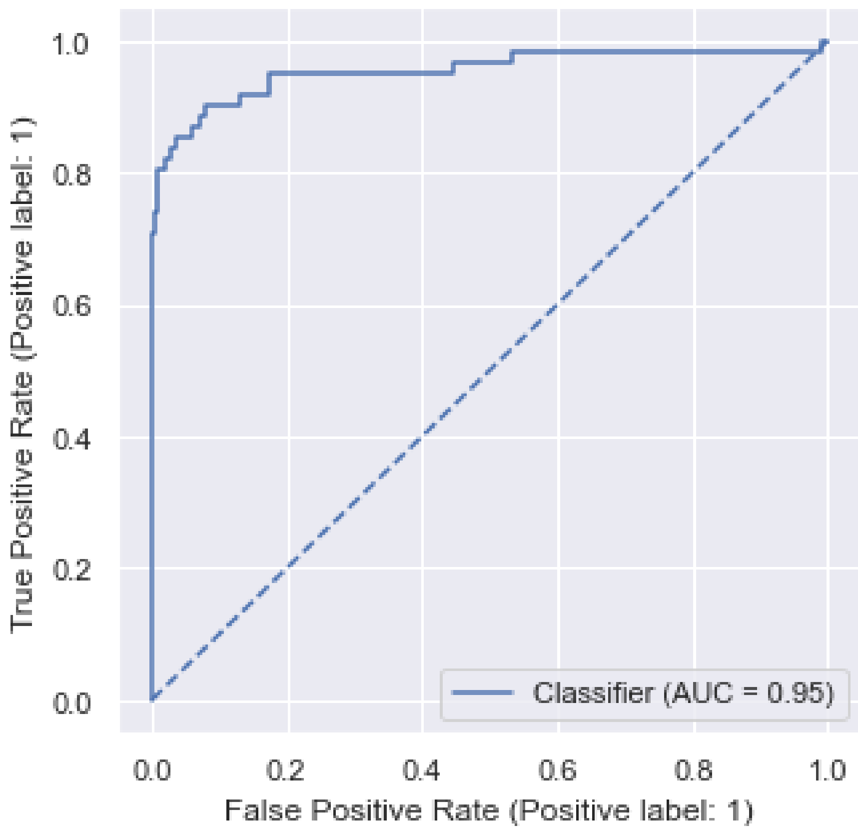

```
Class sit down. Sample-Wise Acc: 0.996, Recall: 1.000, Precision: 0.958, F1:
0.979
```



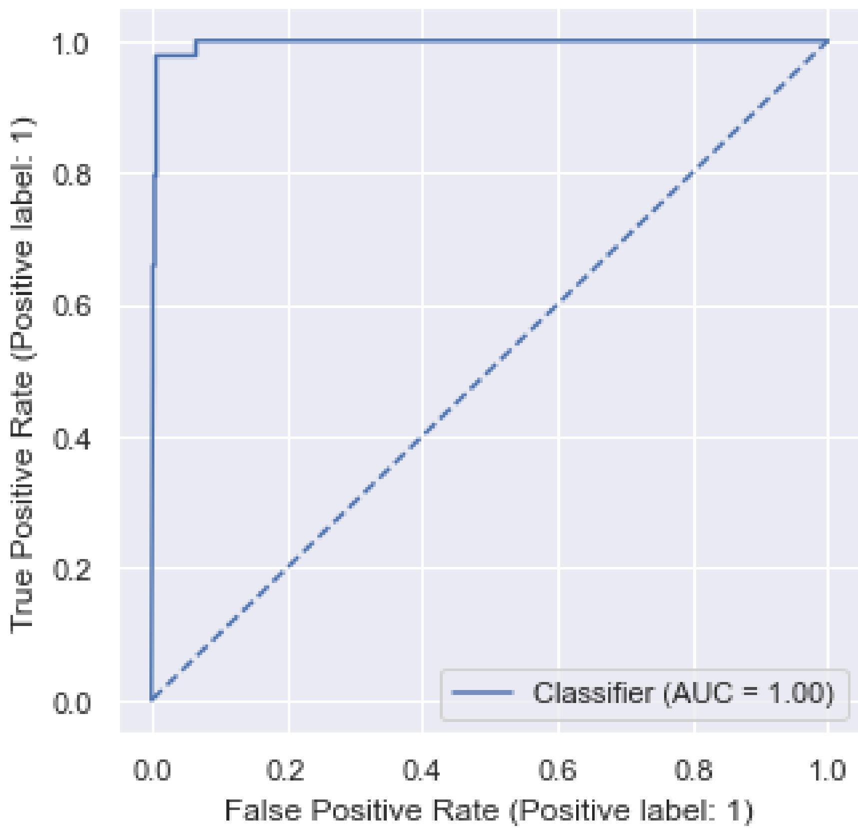
Class mopping the floor. Sample-Wise Acc: 0.961, Recall: 0.902, Precision: 0.809, F1: 0.853



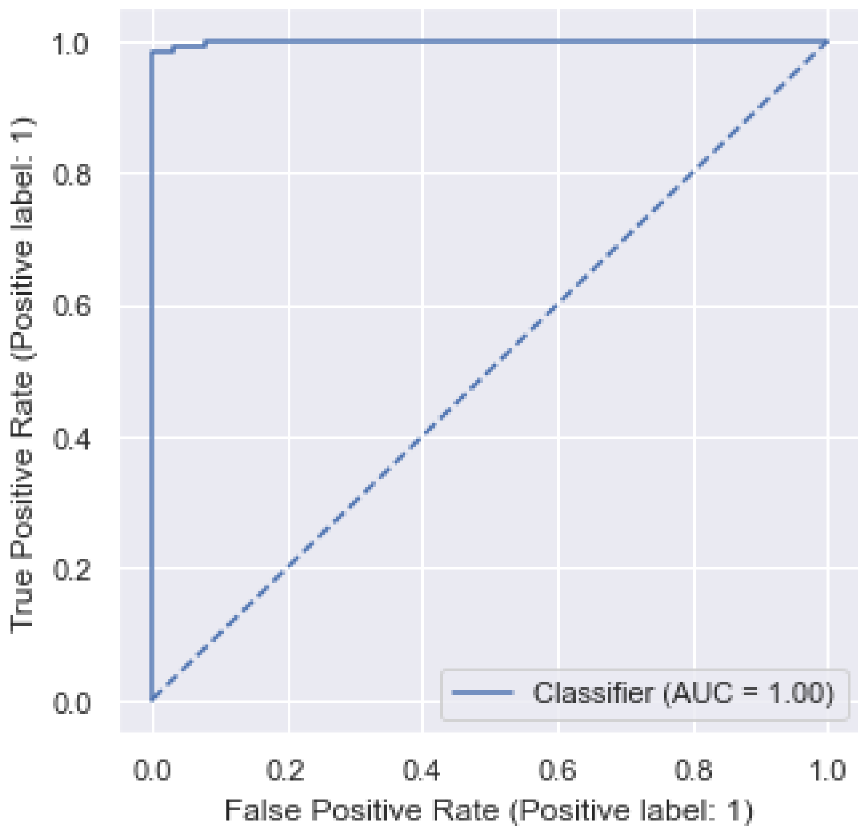
`Class sweeping the floor. Sample-Wise Acc: 0.967, Recall: 0.806, Precision: 0.926, F1: 0.862`




```
Class walking. Sample-Wise Acc: 0.981, Recall: 0.864, Precision: 0.927, F1:
0.894
```




```
Class unknown. Sample-Wise Acc: 0.992, Recall: 0.982, Precision: 0.982, F1:
0.982
```



5 Saving the Model

[ ]:
```
print("path_root:", path_root, "\n")

SaveFolder = (f'{path_root}/Models/' +
              f'{datetime.now(timezone("Europe/Athens")).strftime("%Y-%m-%d␣
 ↪%H-%M")}' +
              f", Valid Loss {valid_loss:.2f}" +
              f' Acc {valid_metric1:.2f}' +
              f', Test Acc {test_Acc:.2f}' +
              f' AUC {test_AUC:.2f}' +
              f' F1 {test_F1:.2f}'
)
print(SaveFolder)
os.makedirs(SaveFolder, exist_ok = True)
```

[98]:
```
if isnotebook:
    i = 2
```

[ ]:
```
if isnotebook:
    i += 1
    print(i)
    display(_ih[i])
```

[102]:
```
if isnotebook:
    PossibleNetClass = _ih[i]
```

[172]:
```
### Saving the Model ###
QuoteIfNone = lambda x: f"'{x}'" if x is not None else "None"

#Saving the Parameters
WriteText(
    f"#PyTorch v{torch.__version__}\n#CUDA device available:␣
↪{IS_GPU_AVAILABLE}\n{f'#{torch.cuda.device_count()} devices available' if␣
↪torch.cuda.is_available() else ''}\n#device = {device}\n" +
    f"#isnotebook = {isnotebook}\n#isgooglecolab = {isgooglecolab}\n#shell =␣
↪{shell}\n\n" +
    f"layer_type = {json.dumps(layer_type)}\nSeed = {Seed}\nnum_units = {json.
↪dumps(num_units)}\nactivation = {json.dumps(activation)}\ndropout = {json.
↪dumps(dropout)}\nusebias = {usebias}\n" +
    f"batch_size = {batch_size}\n" +
    f"K_Length = {K_Length}\nD_Length = {D_Length}\nH1 = {H1}\nW1 =␣
↪{W1}\nconv_input_size = {conv_input_size}\ninput_size =␣
↪{input_size}\noutput_size = {output_size}\nhn1 = {hn1}\n\n" +
    f"l2_lamda = {l2_lamda}\nmu = {mu}\nbatchnorm_momentum =␣
↪{batchnorm_momentum}\nconv_pool_size = {conv_pool_size}\nconv_pool_stride =␣
↪{conv_pool_stride}\n" +
    f"conv_pool_padding = {conv_pool_padding}\nconv_pool_dilation =␣
↪{conv_pool_dilation}\n" +
    f"\nPrintInfoEverynEpochs = {PrintInfoEverynEpochs}\n" +
    f"\ntrain_best_loss = {train_best_loss}\nvalid_best_loss =␣
↪{valid_best_loss}\nReluAlpha = {ReluAlpha}\nEluAlpha = {EluAlpha}\n" +
    f"valid_metric1 = {valid_metric1}\nvalid_metric2 =␣
↪{valid_metric2}\nvalid_metric3 = {valid_metric3}\n"
    f"valid_best_metric1 = {valid_best_metric1}\nvalid_best_metric2 =␣
↪{valid_best_metric2}\nvalid_best_metric3 = {valid_best_metric3}\n",
f"{SaveFolder}/Parameters.py")

#Saving the Losses so we can plot them in the future
WriteText(json.dumps(train_losses.tolist())   , f"{SaveFolder}/Metrics/
↪train_losses.json"  )
WriteText(json.dumps(valid_losses.tolist())   , f"{SaveFolder}/Metrics/
↪valid_losses.json"  )
WriteText(json.dumps(train_metric1s.tolist()), f"{SaveFolder}/Metrics/
↪train_metric1s.json")
```

```
WriteText(json.dumps(train_metric2s.tolist()), f"{SaveFolder}/Metrics/
 ↪train_metric2s.json")
WriteText(json.dumps(train_metric3s.tolist()), f"{SaveFolder}/Metrics/
 ↪train_metric3s.json")
WriteText(json.dumps(valid_metric1s.tolist()), f"{SaveFolder}/Metrics/
 ↪valid_metric1s.json")
WriteText(json.dumps(valid_metric2s.tolist()), f"{SaveFolder}/Metrics/
 ↪valid_metric2s.json")
WriteText(json.dumps(valid_metric3s.tolist()), f"{SaveFolder}/Metrics/
 ↪valid_metric3s.json")

#Saving the Optimiser
WriteText(optimiser, f"{SaveFolder}/Optimiser.txt")
#Saving the optimiser's parameters
torch.save(optimiser.state_dict(), f"{SaveFolder}/optimiser_dict.pt")

#Saving the Criterion
SaveVariable(criterion, f"{SaveFolder}/criterion.pt")
#Saving the Criterion's name (easier to see by looking at the file)
WriteText(criterion, f"{SaveFolder}/criterion.txt")

#Saving the Net() Class
if isnotebook:
    print(PossibleNetClass.partition('\n')[0])
    WriteText(PossibleNetClass, f"{SaveFolder}/Net.py")
else:
    WriteText(f"#Net() not found.\n#Probably not a notebook?\n", f"{SaveFolder}/
 ↪Net.py")

#Saving Model itself
torch.save(model.state_dict(), f"{SaveFolder}/model_dict.pt")
#Saving Model itself
SaveVariableDill(model, f"{SaveFolder}/model.pt")

#Saving the Scheduler
SaveVariableDill(scheduler, f"{SaveFolder}/scheduler.pt")
print("Done!")
```

```
#Dynamic with "layer.append()"
Done!
```

6 Loading the Model

[190]:
```
SaveFolder = f"{path_root}/Models/2022-09-11 19-38, Valid Loss 1.36 Acc 0.92,␣
 ↪Test Acc 0.94 AUC 0.99 F1 0.94"
```

[191]:
```
#Loading the Parameters
exec(ReadText(f"{SaveFolder}/Parameters.py"))

#Loading the Criterion
criterion = LoadVariable(f"{SaveFolder}/criterion.pt")

#Loading Model itself
model = LoadVariableDill(f"{SaveFolder}/model.pt").to(device)
model.eval() #Putting model in evaluation mode so that things like dropout()␣
↪are deactivated

#Loading the Optimiser
optimiser = torch.optim.AdamW(model.parameters(), lr = 0.1, betas = (0.9, 0.9),␣
↪weight_decay = 0, amsgrad = False)
optimiser.load_state_dict(torch.load(f"{SaveFolder}/optimiser_dict.pt"))

#Loading the Scheduler
scheduler = LoadVariableDill(f"{SaveFolder}/scheduler.pt")
print("Done!")
```

```
Done!
```

[192]:
```
train_losses   = np.array(json.loads(ReadText(f"{SaveFolder}/Metrics/
↪train_losses.json")))
valid_losses   = np.array(json.loads(ReadText(f"{SaveFolder}/Metrics/
↪valid_losses.json")))
train_metric1s = np.array(json.loads(ReadText(f"{SaveFolder}/Metrics/
↪train_metric1s.json")))
train_metric2s = np.array(json.loads(ReadText(f"{SaveFolder}/Metrics/
↪train_metric2s.json")))
train_metric3s = np.array(json.loads(ReadText(f"{SaveFolder}/Metrics/
↪train_metric3s.json")))
valid_metric1s = np.array(json.loads(ReadText(f"{SaveFolder}/Metrics/
↪valid_metric1s.json")))
valid_metric2s = np.array(json.loads(ReadText(f"{SaveFolder}/Metrics/
↪valid_metric2s.json")))
valid_metric3s = np.array(json.loads(ReadText(f"{SaveFolder}/Metrics/
↪valid_metric3s.json")))
print("Done!")
```

```
Done!
```

[194]:
```
model = Net(K_Length, num_units, activation, dropout, usebias)

if device != "cpu":
    model.load_state_dict(torch.load(f"{SaveFolder}/model_dict.pt"))
#     model = nn.DataParallel(model)
```

```
else:
    model.load_state_dict(torch.load(f"{SaveFolder}/model_dict.pt",␣
 ↪map_location = torch.device("cpu")))

#model.load_state_dict(OrderedDict((k.replace("module.", ""), v) for k, v in␣
 ↪torch.load(f"{SaveFolder}/model_dict.pt", map_location = "cpu").items()))␣
 ↪#De-DataParallel the model
model.eval()
print("Done!")
```

```
Done!
```

7 Predicting on External Data

[180]:
```
External_DF = pd.read_csv(f"{path_data}/11-16 (1).csv", header = None).iloc[:,␣
 ↪1:]
X_External = External_DF.values.astype(np.float32)
X_External = Scale(X_External, *scaler_mean_sd)

MYDataset = TensorDataset(torch.from_numpy(X_External))
MYDataLoader = torch.utils.data.DataLoader(
    MYDataset,
    batch_size = batch_size,
    pin_memory = True
)

model.to(device)
model.eval()
with torch.no_grad():
    TestPredictions = []
    for inputs in MYDataLoader:
        TestPredictions.append(model(inputs[0].to(device)))
    _, TestPredictions = torch.max(nn.Softmax(dim = 1)(torch.
 ↪cat(TestPredictions)), 1)
    TestPredictions += 1
    TestResults = pd.DataFrame(TestPredictions.cpu().numpy()).reset_index()
    TestResults.columns = ["Id", "Class"]

FileExportPath = f"{path_root}/Exports/Results.csv"
TestResults.to_csv(FileExportPath, sep = ',', header = True, index = False)
print(f"Results saved on: {FileExportPath}")

TestResults
```

```
Results saved on: D:\GiannisM\Downloads\Exercises\Fiver\100. frotribe FFNN
Multiclass Classification/Exports/Results.csv
```

[180]:

```
        Id  Class
0        0      1
1        1      1
2        2      1
3        3      1
4        4      1
…      …    …
4830  4830      8
4831  4831      8
4832  4832      8
4833  4833      8
4834  4834      8

[4835 rows x 2 columns]
```

[ ]:

```
```