Praful_Patil_HW4A

March 3, 2024

###**Setting up the environment**

[2]:
```
from pathlib import Path
if 'google.colab' in str(get_ipython()):
    from google.colab import drive
    drive.mount("/content/drive")
    !pip install datasets transformers evaluate wandb accelerate -U -qq
    base_folder = Path("/content/drive/MyDrive")
else:
    base_folder = Path("/home/harpreet/Insync/google_drive_shaannoor/data")

from sklearn.model_selection import train_test_split
import evaluate
import torch
from torch.utils.data import Dataset, DataLoader
import ast
import joblib
import torch.nn as nn
from collections import Counter
import numpy as np
from sklearn.preprocessing import MultiLabelBinarizer
!pip install torchmetrics
from torchmetrics import HammingDistance
from torchmetrics.classification import  MultilabelHammingDistance
from  torchmetrics.functional.classification import multilabel_f1_score,␣
↪multilabel_hamming_distance
from torch.nn.utils import clip_grad_value_
import pandas as pd
from functools import partial
from types import SimpleNamespace
```

```
Mounted at /content/drive
                            510.5/510.5
kB 5.8 MB/s eta 0:00:00
                            8.5/8.5 MB
20.1 MB/s eta 0:00:00
                            84.1/84.1 kB
11.5 MB/s eta 0:00:00
```

```
                            2.2/2.2 MB
37.3 MB/s eta 0:00:00
                            280.0/280.0

kB 32.3 MB/s eta 0:00:00
                            116.3/116.3

kB 14.6 MB/s eta 0:00:00
                            134.8/134.8

kB 16.4 MB/s eta 0:00:00
                            195.4/195.4

kB 22.8 MB/s eta 0:00:00
                            258.5/258.5

kB 26.9 MB/s eta 0:00:00
                            62.7/62.7 kB
8.2 MB/s eta 0:00:00
Collecting torchmetrics
  Downloading torchmetrics-1.3.1-py3-none-any.whl (840 kB)
                            840.4/840.4

kB 7.6 MB/s eta 0:00:00
Requirement already satisfied: numpy>1.20.0 in
/usr/local/lib/python3.10/dist-packages (from torchmetrics) (1.25.2)
Requirement already satisfied: packaging>17.1 in /usr/local/lib/python3.10/dist-
packages (from torchmetrics) (23.2)
Requirement already satisfied: torch>=1.10.0 in /usr/local/lib/python3.10/dist-
packages (from torchmetrics) (2.1.0+cu121)
Collecting lightning-utilities>=0.8.0 (from torchmetrics)
  Downloading lightning_utilities-0.10.1-py3-none-any.whl (24 kB)
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-
packages (from lightning-utilities>=0.8.0->torchmetrics) (67.7.2)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.10/dist-packages (from lightning-
utilities>=0.8.0->torchmetrics) (4.10.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-
packages (from torch>=1.10.0->torchmetrics) (3.13.1)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages
(from torch>=1.10.0->torchmetrics) (1.12)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-
packages (from torch>=1.10.0->torchmetrics) (3.2.1)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages
(from torch>=1.10.0->torchmetrics) (3.1.3)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages
(from torch>=1.10.0->torchmetrics) (2023.6.0)
Requirement already satisfied: triton==2.1.0 in /usr/local/lib/python3.10/dist-
packages (from torch>=1.10.0->torchmetrics) (2.1.0)
Requirement already satisfied: MarkupSafe>=2.0 in
```

```
/usr/local/lib/python3.10/dist-packages (from
jinja2->torch>=1.10.0->torchmetrics) (2.1.5)
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.10/dist-
packages (from sympy->torch>=1.10.0->torchmetrics) (1.3.0)
Installing collected packages: lightning-utilities, torchmetrics
Successfully installed lightning-utilities-0.10.1 torchmetrics-1.3.1
```

##**Loading Dataset**

[3]:
```
base_folder = Path('/content/drive/MyDrive/NLP/HW_4')
data_folder = base_folder
custom_functions = base_folder/'custom-functions'
```

[4]:
```
file_name = 'df_multilabel_hw_cleaned.joblib'
file_path = data_folder / file_name

# Load the dataset using joblib
df_multilabel = joblib.load(file_path)
```

[5]:
```
df_multilabel.head(10)
```

[5]:
```
                                        cleaned_text                  Tags  \
0  asp query stre dropdown webpage follow control…          c# asp.net
1  run javascript code server java code want run …     java javascript
2  linq sql throw exception row find change hi li…          c# asp.net
3  run python script php server run nginx web ser…          php python
4  advice write function m try write function res…   javascript jquery
5  jquery auto resize function cause jump browser…   javascript jquery
6  php page redirect operation page php grid subp…      php javascript
7  advice need expert asp.net usercontrol usage n…          c# asp.net
8  revert style apply focus blur user focus text …   javascript jquery
9  hack work look android source develop app app …        java android

  Tag_Number
0     [0, 9]
1     [1, 3]
2     [0, 9]
3     [2, 7]
4     [3, 5]
5     [3, 5]
6     [2, 3]
7     [0, 9]
8     [3, 5]
9     [1, 4]
```

[6]:
```
# Convert the 'Tag_Number' column from string representations of lists to␣
↪actual lists of integers
```

```
df_multilabel['Tag_Number'] = df_multilabel['Tag_Number'].apply(lambda x: ast.
↪literal_eval(x))
```

[7]:
```
# Initialize MultiLabelBinarizer
mlb = MultiLabelBinarizer()

# Fit and transform the 'Tag_Number' column to one-hot encoded format
one_hot_labels = mlb.fit_transform(df_multilabel['Tag_Number'])

# Convert one-hot encoded labels to a DataFrame
one_hot_labels_df = pd.DataFrame(one_hot_labels, columns=mlb.classes_)

# Concatenate the one-hot encoded labels DataFrame with the original DataFrame
df_multilabel = pd.concat([df_multilabel, one_hot_labels_df], axis=1)
```

[8]:
```
df_multilabel.head(10)
```

[8]:
```
                                         cleaned_text                Tags  \
0  asp query stre dropdown webpage follow control…         c# asp.net
1  run javascript code server java code want run …    java javascript
2  linq sql throw exception row find change hi li…         c# asp.net
3  run python script php server run nginx web ser…         php python
4  advice write function m try write function res…  javascript jquery
5  jquery auto resize function cause jump browser…  javascript jquery
6  php page redirect operation page php grid subp…     php javascript
7  advice need expert asp.net usercontrol usage n…         c# asp.net
8  revert style apply focus blur user focus text …  javascript jquery
9  hack work look android source develop app app …       java android

  Tag_Number  0  1  2  3  4  5  6  7  8  9
0     [0, 9]  1  0  0  0  0  0  0  0  0  1
1     [1, 3]  0  1  0  1  0  0  0  0  0  0
2     [0, 9]  1  0  0  0  0  0  0  0  0  1
3     [2, 7]  0  0  1  0  0  0  0  1  0  0
4     [3, 5]  0  0  0  1  0  1  0  0  0  0
5     [3, 5]  0  0  0  1  0  1  0  0  0  0
6     [2, 3]  0  0  1  1  0  0  0  0  0  0
7     [0, 9]  1  0  0  0  0  0  0  0  0  1
8     [3, 5]  0  0  0  1  0  1  0  0  0  0
9     [1, 4]  0  1  0  0  1  0  0  0  0  0
```

##**Splitting the dataset**

[9]:
```
X = df_multilabel['cleaned_text'].values
y = one_hot_labels
```

```
[10]: # Split the data into train, validation, and test sets (60%, 20%, 20%)
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.4,␣
 ↪random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5,␣
 ↪random_state=42)
```

##**CustomDataset class for loading data and labels.**

```
[11]: class CustomDataset(Dataset):
    """
    Custom Dataset class for loading data and labels.
    """
    def __init__(self, X, y):
        self.X = X
        self.y = y

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

    def __getitem__(self, idx):
        texts = self.X[idx]
        labels = self.y[idx]
        sample = (labels, texts)
        return sample
```

```
[12]: train_dataset = CustomDataset(X_train, y_train)
val_dataset = CustomDataset(X_val, y_val)
test_dataset = CustomDataset(X_test, y_test)
```

```
[13]: from collections import Counter
from torchtext.vocab import vocab
!pip install torchinfo
from torchinfo import summary

def get_vocab(dataset, min_freq=1):
    """
    Generate a vocabulary from a dataset.

    Args:
        dataset (list of tuple): List of tuples where each tuple contains a␣
 ↪label and a text.
        min_freq (int): The minimum frequency for a token to be included in the␣
 ↪vocabulary.

    Returns:
        torchtext.vocab.Vocab: Vocabulary object.
    """
```

```
    # Initialize a counter object to hold token frequencies
    counter = Counter()

    # Update the counter with tokens from each text in the dataset
    for (label, text) in dataset:
        counter.update(text.split())

    # Create a vocabulary using the counter object
    # Tokens that appear fewer times than `min_freq` are excluded
    my_vocab = vocab(counter, min_freq=min_freq)

    # Insert a '<unk>' token at index 0 to represent unknown words
    my_vocab.insert_token('<unk>', 0)

    # Set the default index to 0
    # This ensures that any unknown word will be mapped to '<unk>'
    my_vocab.set_default_index(0)

    return my_vocab
```

```
Collecting torchinfo
  Downloading torchinfo-1.8.0-py3-none-any.whl (23 kB)
Installing collected packages: torchinfo
Successfully installed torchinfo-1.8.0
```

[14]:
```
vocab = get_vocab(train_dataset, min_freq=2)
```

##Collate_fn for Data Loaders

[15]:
```
# Creating a function that will be used to get the indices of words from vocab
def tokenizer(x, vocab):
    """Converts text to a list of indices using a vocabulary dictionary"""
    return [vocab[token] for token in x.split()]
```

[16]:
```
def collate_batch(batch, my_vocab):
    """
    Collates a batch of samples into tensors of labels, texts, and offsets.

    Parameters:
        batch (list): A list of tuples, each containing a label and a text.

    Returns:
        tuple: A tuple containing three tensors:
               - Labels tensor
               - Concatenated texts tensor
               - Offsets tensor indicating the start positions of each text in␣
↪the concatenated tensor
```

```python
    """
    # Unpack the batch into separate lists for labels and texts
    labels, texts = zip(*batch)

    # Convert the list of labels into a tensor of dtype int32
    labels = torch.tensor(labels, dtype=torch.long)

    # Convert the list of texts into a list of lists; each inner list contains␣
↪the vocabulary indices for a text
    list_of_list_of_indices = [tokenizer(text, my_vocab) for text in texts]

    # Concatenate all text indices into a single tensor
    indices = torch.cat([torch.tensor(i, dtype=torch.int64) for i in␣
↪list_of_list_of_indices])

    # Compute the offsets for each text in the concatenated tensor
    offsets = [0] + [len(i) for i in list_of_list_of_indices]
    offsets = torch.tensor(offsets[:-1]).cumsum(dim=0)

    return (indices, offsets), labels
```

[17]:
```python
batch_size = 128
collate_partial = partial(collate_batch, my_vocab = vocab)
check_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True,
                                           collate_fn=collate_partial,
                                           )
```

[18]:
```python
torch.manual_seed(22)
for (indices, offset), label in check_loader:
    print(indices, offset, label)
    break
```

```
tensor([   96,   128,  3259,  …,    79,  4981, 25336]) tensor([    0,    49,
654,   705,   727,   749,   924,  1057,  1086,  1182,
         1248,  1280,  1374,  1450,  1510,  1591,  1690,  1717,  1725,  1785,
         1847,  1875,  1918,  2131,  2170,  2198,  2239,  2261,  2558,  2588,
         2616,  2643,  2697,  2780,  2842,  3002,  3267,  3307,  3493,  3539,
         3551,  3580,  3622,  4205,  4217,  4296,  4345,  4361,  4462,  4587,
         4656,  4708,  4732,  4825,  4879,  4932,  5184,  5265,  5312,  5336,
         5382,  5491,  5539,  5562,  5591,  5775,  5830,  6083,  6192,  6378,
         6418,  6481,  6501,  6560,  6628,  6691,  6728,  7006,  7053,  7101,
         7139,  7211,  7238,  7381,  7435,  7467,  7500,  7554,  7793,  7819,
         7862,  7945,  8007,  8028,  8070,  8082,  8135,  8192,  8254,  8289,
         8316,  8508,  8554,  8626,  8677,  8724,  8750,  9122,  9152,  9215,
         9263,  9375,  9405,  9504,  9557,  9645,  9706,  9773,  9805, 10245,
```

```
        10351, 10378, 10423, 10450, 10458, 10669, 10783, 10837]) tensor([[0, 0,
0,  …, 0, 0, 0],
        [0, 1, 0,  …, 0, 0, 0],
        [0, 0, 1,  …, 0, 0, 0],
        …,
        [0, 0, 0,  …, 0, 0, 1],
        [0, 0, 0,  …, 0, 0, 0],
        [1, 0, 0,  …, 0, 0, 1]])
```

```
<ipython-input-16-1bbe20fd9005>:18: UserWarning: Creating a tensor from a list
of numpy.ndarrays is extremely slow. Please consider converting the list to a
single numpy.ndarray with numpy.array() before converting to a tensor.
(Triggered internally at ../torch/csrc/utils/tensor_new.cpp:261.)
  labels = torch.tensor(labels, dtype=torch.long)
```

[19]:

```
# Define the batch size
batch_size = 128
```

[20]:

```
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size,␣
 ↪shuffle=True)
val_loader = DataLoader(dataset=val_dataset, batch_size=batch_size,␣
 ↪shuffle=False)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size,␣
 ↪shuffle=False)
```

##Creating custom model class

[21]:

```
class SimpleMLP(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim1, hidden_dim2,␣
 ↪drop_prob1, drop_prob2, num_outputs):
        super().__init__()

        #␣
 ↪EmbeddingBag_layer->Linear->ReLU->BatchNorm->Dropout->Linear->ReLU->BatchNorm->Dropout->Lin

        # Embedding layer
        self.embedding_bag = nn.EmbeddingBag(vocab_size, embedding_dim)

        # First Linear layer
        self.linear1 = nn.Linear(embedding_dim, hidden_dim1)
        # relu activation function
        self.relu1 = nn.ReLU()
        # Batch normalization for first linear layer
        self.batchnorm1 = nn.BatchNorm1d(num_features=hidden_dim1)
        # Dropout for first linear layer
        self.dropout1 = nn.Dropout(p=drop_prob1)
        # Second Linear layer
        self.linear2 = nn.Linear(hidden_dim1, hidden_dim2)
```

```python
        # relu activation function
        self.relu2 = nn.ReLU()
        # Batch normalization for second linear layer
        self.batchnorm2 = nn.BatchNorm1d(num_features=hidden_dim2)
        # Dropout for second linear layer
        self.dropout2 = nn.Dropout(p=drop_prob2)

        # Final Linear layer
        self.linear3 = nn.Linear(hidden_dim2, num_outputs)

    def forward(self, input_tuple):
        indices, offsets = input_tuple

        # Pass data through the embedding layer
        x = self.embedding_bag(indices, offsets)

        # First linear layer followed by ReLU, BatchNorm, and Dropout
        x = self.linear1(x)
        x = self.relu1(x)
        x = self.dropout1(x)
        x = self.batchnorm1(x)

        # Second linear layer followed by ReLU, BatchNorm, and Dropout
        x = self.linear2(x)
        x = self.relu2(x)
        x = self.dropout2(x)
        x = self.batchnorm2(x)

        # Final linear layer
        x = self.linear3(x)

        return x
```

#Functions to train and evaluate the model

##Step Function

[22]:
```python
def step(inputs, targets, model, device, loss_function=None, optimizer=None,␣
 ↪clip_value=10):
    """
    Performs a forward and backward pass for a given batch of inputs and␣
 ↪targets.

    Parameters:
    - inputs (torch.Tensor): The input data for the model.
    - targets (torch.Tensor): The true labels for the input data.
    - model (torch.nn.Module): The neural network model.
    - device (torch.device): The computing device (CPU or GPU).
```

```
    - loss_function (torch.nn.Module, optional): The loss function to use.
    - optimizer (torch.optim.Optimizer, optional): The optimizer to update␣
 ↪model parameters.

    Returns:
    - loss (float): The computed loss value (only if loss_function is not None).
    - outputs (torch.Tensor): The predictions from the model.
    - correct (int): The number of correctly classified samples in the batch.
    """
    # Move the model and data to the device
    model = model.to(device)
    inputs = tuple(input_tensor.to(device)
                            for input_tensor in inputs)

    targets = targets.to(device)
    targets = targets.float()

    # Step 1: Forward pass to get the model's predictions
    outputs = model(inputs)

    # Step 2a: Compute the loss using the provided loss function
    if loss_function:
        loss = loss_function(outputs, targets)

    # Step 3 and 4: Perform backward pass and update model parameters if an␣
 ↪optimizer is provided
    if optimizer:
        optimizer.zero_grad()
        loss.backward()
        clip_grad_value_(model.parameters(), clip_value=clip_value)
        optimizer.step()

    # Return relevant metrics
    if loss_function:
        return loss, outputs
    else:
        return outputs
```

##**Train Epoch**

[23]:
```
def train_epoch(train_loader, model, device, loss_function, optimizer,␣
 ↪train_hamming, clip_value=None):
    """
    Trains the model for one epoch using the provided data loader and updates␣
 ↪the model parameters.

    Parameters:
```

```
    - train_loader (torch.utils.data.DataLoader): DataLoader object for the␣
↪training set.
    - model (torch.nn.Module): The neural network model to be trained.
    - device (torch.device): The computing device (CPU or GPU).
    - loss_function (torch.nn.Module): The loss function to use for training.
    - optimizer (torch.optim.Optimizer): The optimizer to update model␣
↪parameters.

    Returns:
    - train_loss (float): Average training loss for the epoch.
    - train_acc (float): Training accuracy for the epoch.
    """
    # Set the model to training mode
    model.train()

    # Initialize variables to track running training loss and correct␣
↪predictions
    running_train_loss = 0.0

    # Iterate over all batches in the training data
    for inputs, targets in train_loader:
        # Perform a forward and backward pass, updating model parameters
        loss, outputs = step(inputs, targets, model, device, loss_function,␣
↪optimizer, clip_value=clip_value)

        # Update running loss and correct predictions counter
        running_train_loss += loss.item()

        with torch.no_grad():
            # Correct prediction using thresholding
            predictions = (outputs >= 0).float()
            # Hamming distance calculation
            train_hamming = train_hamming.to(device)
            train_hamming.update(predictions, targets.to(device))

    # Compute average loss and accuracy for the entire training set
    train_loss = running_train_loss / len(train_loader)

    return train_loss, train_hamming
```

##**Val Epoch**

[24]:
```
def val_epoch(valid_loader, model, device, loss_function, val_hamming):


    # Set the model to evaluation mode
    model.eval()
```

```
    # Initialize variables to track running validation loss and correct␣
↪predictions
    running_val_loss = 0.0

    # Disable gradient computation
    with torch.no_grad():
        # Iterate over all batches in the validation data
        for inputs, targets in valid_loader:
            # Perform a forward pass to get loss and number of correct␣
↪predictions
            loss, outputs= step(inputs, targets, model, device,␣
↪loss_function=loss_function, optimizer=None, clip_value=None)

            # Update running loss and correct predictions counter
            running_val_loss += loss.item()
            y_pred = (outputs >= 0).float()

            # Update Hamming Distance metric
            val_hamming = val_hamming.to(device)
            val_hamming.update(y_pred, targets.to(device))


    # Compute average loss and accuracy for the entire validation set
    val_loss = running_val_loss / len(valid_loader)

    return val_loss, val_hamming
```

###**Train() function**

[25]:
```
def train(train_loader, val_loader, model, optimizer, loss_function, epochs,␣
↪device, train_hamming, val_hamming, clip_value=10):
    """
    Trains and validates the model, and returns history of train and validation␣
↪metrics.

    Parameters:
    - train_loader (torch.utils.data.DataLoader): DataLoader for the training␣
↪set.
    - valid_loader (torch.utils.data.DataLoader): DataLoader for the validation␣
↪set.
    - model (torch.nn.Module): Neural network model to train.
    - optimizer (torch.optim.Optimizer): Optimizer algorithm.
    - loss_function (torch.nn.Module): Loss function to evaluate the model.
    - epochs (int): Number of epochs to train the model.
    - device (torch.device): The computing device (CPU or GPU).
```

```
    Returns:
    - train_loss_history (list): History of training loss for each epoch.
    - train_acc_history (list): History of training accuracy for each epoch.
    - valid_loss_history (list): History of validation loss for each epoch.
    - valid_acc_history (list): History of validation accuracy for each epoch.
    """

    # Initialize lists to store metrics for each epoch
    train_loss_history = []
    val_loss_history = []
    train_hamming_history = []
    val_hamming_history = []
    model = model.to(device)

    # Loop over the number of specified epochs
    for epoch in range(epochs):
        # Train model on training data and capture metrics
        train_loss, train_hamming = train_epoch(
            train_loader, model, device, loss_function, optimizer,␣
↪train_hamming, clip_value=clip_value)

        # Validate model on validation data and capture metrics
        val_loss, val_hamming = val_epoch(
            val_loader, model, device, loss_function, val_hamming)

        epoch_train_hamming, epoch_val_hamming = train_hamming.compute(),␣
↪val_hamming.compute()

        # Store metrics for this epoch
        train_loss_history.append(train_loss)
        train_hamming_history.append(epoch_train_hamming.item())
        val_loss_history.append(val_loss)
        val_hamming_history.append(epoch_val_hamming.item())

        # Output epoch-level summary
        print(f"Epoch {epoch+1}/{epochs}")
        print(f"Train Loss: {train_loss:.4f} | Train hamming:␣
↪{epoch_train_hamming:.4f}")
        print(f"Val Loss: {val_loss:.4f} | Val hamming: {epoch_val_hamming:.
↪4f}")
        print()

        # Reset metric states after each epoch
        train_hamming.reset()
        val_hamming.reset()
```

```
    return train_loss_history, train_hamming_history, val_loss_history,␣
↪val_hamming_history
```

#**Hyperparameters and training config**

[26]:
```
import torch
from torch import nn
from torch.optim import AdamW
from torchmetrics import HammingDistance

# Set the seed for reproducibility
SEED = 100
torch.manual_seed(SEED)
torch.cuda.manual_seed_all(SEED)

# Define hyperparameters
HIDDEN_DIM1 = 200
HIDDEN_DIM2 = 100
EMBED_DIM = 300
EPOCHS = 5
BATCH_SIZE = 128
LEARNING_RATE = 0.001
WEIGHT_DECAY = 0.000
CLIP_VALUE = 10
PATIENCE = 5
NUM_OUTPUTS = 10
VOCAB_SIZE = len(vocab)
DROP_PROB1 = 0.5
DROP_PROB2 = 0.5

# Determine the computing device
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")

# Initialize the model with the specified hyperparameters
model = SimpleMLP(
    vocab_size=VOCAB_SIZE,
    embedding_dim=EMBED_DIM,
    hidden_dim1=HIDDEN_DIM1,
    hidden_dim2=HIDDEN_DIM2,
    drop_prob1=DROP_PROB1,
    drop_prob2=DROP_PROB2,
    num_outputs=NUM_OUTPUTS
)

# Transfer the model to the device
model.to(device)
```

```
# Configure the optimizer
optimizer = AdamW(model.parameters(), lr=LEARNING_RATE,␣
↪weight_decay=WEIGHT_DECAY)

# Define the loss function
loss_function = nn.BCEWithLogitsLoss()

# Define collate function with a fixed vocabulary using the 'partial' function
collate_fn = partial(collate_batch, my_vocab=vocab)

# Data Loaders for training, validation, and test sets
# These loaders handle batching, shuffling, and data processing using the␣
↪custom collate function
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size =␣
↪BATCH_SIZE, shuffle=True,
                                           collate_fn=collate_fn, num_workers=4)
valid_loader = torch.utils.data.DataLoader(val_dataset, batch_size=BATCH_SIZE,␣
↪shuffle=False,
                                           collate_fn=collate_fn, num_workers=4)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=BATCH_SIZE,␣
↪shuffle=False,
                                          collate_fn=collate_fn, num_workers=4)

train_hamming = HammingDistance(task="multilabel", num_labels=NUM_OUTPUTS)
val_hamming = HammingDistance(task="multilabel", num_labels=NUM_OUTPUTS)
```

```
Using device: cpu
```

```
/usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.py:557:
UserWarning: This DataLoader will create 4 worker processes in total. Our
suggested max number of worker in current system is 2, which is smaller than
what this DataLoader is going to create. Please be aware that excessive worker
creation might get DataLoader running slow or even freeze, lower the worker
number to avoid potential slowness/freeze if necessary.
  warnings.warn(_create_warning_msg(
```

[27]:
```
for input_, targets in train_loader:
    # Move inputs and targets to GPU
    input_ = tuple(input_tensor.to(device) for input_tensor in input_)
    targets = targets.to(device).float()  # Convert targets to float32

    model = model.to(device)
    model.eval()

    # Forward pass
    output = model(input_)
```

```
    loss = loss_function(output, targets)
    print(f'Actual loss: {loss}')
    break
print(f'Expected Theoretical loss: {np.log(2)}')
```

```
Actual loss: 0.6891480684280396
Expected Theoretical loss: 0.6931471805599453
```

##**Training**

[28]:
```
# Call the training function to start the training process
train_losses, train_acc, valid_losses, valid_acc = train(
    train_loader, valid_loader, model, optimizer, loss_function, EPOCHS, device,
    train_hamming, val_hamming, clip_value= CLIP_VALUE)
```

```
Epoch 1/5
Train Loss: 0.3918 | Train hamming: 0.1699
Val Loss: 0.1730 | Val hamming: 0.0599

Epoch 2/5
Train Loss: 0.1666 | Train hamming: 0.0584
Val Loss: 0.1385 | Val hamming: 0.0499

Epoch 3/5
Train Loss: 0.1352 | Train hamming: 0.0479
Val Loss: 0.1229 | Val hamming: 0.0445

Epoch 4/5
Train Loss: 0.1168 | Train hamming: 0.0416
Val Loss: 0.1185 | Val hamming: 0.0429

Epoch 5/5
Train Loss: 0.1040 | Train hamming: 0.0373
Val Loss: 0.1103 | Val hamming: 0.0390
```

[29]:
```
import torch

def get_acc_pred(data_loader, model, device, threshold=0.5):
    model.to(device)
    model.eval()

    all_predictions = []
    all_targets = []

    with torch.no_grad():
        for inputs_tuple, targets in data_loader:
```

```
            inputs_tuple = tuple(input_tensor.to(device) for input_tensor in␣
↪inputs_tuple)
            targets = targets.to(device)

            outputs = model(inputs_tuple)
            predicted = (torch.sigmoid(outputs) > threshold).float()

            all_predictions.append(predicted.cpu())
            all_targets.append(targets.cpu())

    all_predictions = torch.cat(all_predictions)
    all_targets = torch.cat(all_targets)

    return all_predictions, all_targets
```

[30]:
```
predictions_test, target_test = get_acc_pred(test_loader, model, device)
predictions_train, target_train = get_acc_pred(train_loader, model, device)
predictions_valid, target_valid = get_acc_pred(valid_loader, model, device)
```

[31]:
```
#multi-label classification where labels = 10
num_labels = 10

# Calculate Hamming Distance for the training dataset
train_hd = multilabel_hamming_distance(predictions_train.int(), target_train.
↪int(), num_labels=num_labels)

# Calculate Hamming Distance for the validation dataset
valid_hd = multilabel_hamming_distance(predictions_valid.int(), target_valid.
↪int(), num_labels=num_labels)

# Calculate Hamming Distance for the test dataset
test_hd = multilabel_hamming_distance(predictions_test.int(), target_test.
↪int(), num_labels=num_labels)
```

[32]:
```
print(f"Training Hamming Distance: {train_hd}")
print(f"Validation Hamming Distance: {valid_hd}")
print(f"Test Hamming Distance: {test_hd}")
```

```
Training Hamming Distance: 0.026061290875077248
Validation Hamming Distance: 0.039008963853120804
Test Hamming Distance: 0.0415559783577919
```

##**Understanding model's performance with F1_score** f1_score is called with average='micro' to compute the micro-averaged F1 score, which aggregates the contributions of all classes to compute the average metric.

[37]:

```python
from torchmetrics.functional import f1_score

# F1 Score for the training dataset
train_f1_micro = f1_score(predictions_train, target_train,␣
↪num_labels=num_labels, average='micro',task='multilabel')

# F1 Score for the validation dataset
valid_f1_micro = f1_score(predictions_valid, target_valid,␣
↪num_labels=num_labels, average='micro',task='multilabel')

# F1 Score for the test dataset
test_f1_micro = f1_score(predictions_test, target_test, num_labels=num_labels,␣
↪average='micro',task='multilabel')

print(f"Training F1 Score (Micro): {train_f1_micro}")
print(f"Validation F1 Score (Micro): {valid_f1_micro}")
print(f"Test F1 Score (Micro): {test_f1_micro}")
```

```
Training F1 Score (Micro): 0.9364752769470215
Validation F1 Score (Micro): 0.9043828845024109
Test F1 Score (Micro): 0.8981553316116333
```

#####**As indicated by the low Hamming Distance and high F1 score on the training set. There is a slight performance drop on the validation and test sets, as expected, since these sets contain unseen data. However, the overall performance remains relatively high, suggesting that the model generalizes well to new data.**