```
# Numerical computations
import numpy as np

# Data handling and reading dataset
import pandas as pd

# PyTorch main library
import torch

# Neural network layers
import torch.nn as nn

# Optimizer for training
import torch.optim as optim

# Splitting dataset into train and test
from sklearn.model_selection import train_test_split

# Evaluation metrics
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix

# For creating custom dataset loader
from torch.utils.data import Dataset, DataLoader

# For cleaning text using regular expressions
import re
```

```
# Read the SMS dataset file
# sep="\t" means values are separated by tab space
df = pd.read_csv("SMSSpamCollection", sep="\t", header=None, names=["label","text"])

# Display first 5 rows
print(df.head())

# Show total rows and columns
print(df.shape)

# Count how many spam and ham messages exist
print(df["label"].value_counts())
```

```
  label                                               text
0   ham  Go until jurong point, crazy.. Available only ...
1   ham                      Ok lar... Joking wif u oni...
2  spam  Free entry in 2 a wkly comp to win FA Cup fina...
3   ham  U dun say so early hor... U c already then say...
4   ham  Nah I don't think he goes to usf, he lives aro...
(5572, 2)
label
ham     4825
spam     747
Name: count, dtype: int64
```

```
# Function to clean text data
def clean_text(text):
    text = text.lower()  # convert text to lowercase
    text = re.sub(r"[^a-zA-Z\s]", "", text)  # remove numbers & symbols
    return text

# Apply cleaning function to text column
df["text"] = df["text"].apply(clean_text)

# Convert labels into numbers (ham=0, spam=1)
df["label"] = df["label"].map({"ham":0,"spam":1})
```

```
# Split each sentence into words
tokenized = [t.split() for t in df["text"]]

# Create dictionary with special tokens
vocab = {"<PAD>":0,"<UNK>":1}

# Add every unique word into vocabulary
```

```
for sentence in tokenized:
    for word in sentence:
        if word not in vocab:
            vocab[word] = len(vocab)

# Total number of unique words
vocab_size = len(vocab)
print("Vocabulary Size:", vocab_size)
```

```
Vocabulary Size: 8631
```

```
# Download GloVe embeddings from internet
!wget http://nlp.stanford.edu/data/glove.6B.zip

# Extract the zip file
!unzip glove.6B.zip
```

```
--2026-02-19 04:30:40--  http://nlp.stanford.edu/data/glove.6B.zip
Resolving nlp.stanford.edu (nlp.stanford.edu)... 171.64.67.140
Connecting to nlp.stanford.edu (nlp.stanford.edu)|171.64.67.140|:80... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://nlp.stanford.edu/data/glove.6B.zip [following]
--2026-02-19 04:30:40--  https://nlp.stanford.edu/data/glove.6B.zip
Connecting to nlp.stanford.edu (nlp.stanford.edu)|171.64.67.140|:443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://downloads.cs.stanford.edu/nlp/data/glove.6B.zip [following]
--2026-02-19 04:30:40--  https://downloads.cs.stanford.edu/nlp/data/glove.6B.zip
Resolving downloads.cs.stanford.edu (downloads.cs.stanford.edu)... 171.64.64.22
Connecting to downloads.cs.stanford.edu (downloads.cs.stanford.edu)|171.64.64.22|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 862182613 (822M) [application/zip]
Saving to: ‘glove.6B.zip’

glove.6B.zip        100%[===================>] 822.24M  5.02MB/s    in 2m 42s

2026-02-19 04:33:22 (5.08 MB/s) - ‘glove.6B.zip’ saved [862182613/862182613]

Archive:  glove.6B.zip
  inflating: glove.6B.50d.txt
  inflating: glove.6B.100d.txt
  inflating: glove.6B.200d.txt
  inflating: glove.6B.300d.txt
```

```
# Set embedding size (each word vector length)
embedding_dim = 100

# Create empty matrix for storing embeddings
embedding_matrix = np.zeros((vocab_size, embedding_dim))

# Open GloVe file
with open("glove.6B.100d.txt", encoding="utf8") as f:
    for line in f:
        values = line.split()  # split line into words
        word = values[0]  # first element is word
        vector = np.asarray(values[1:], dtype='float32')  # rest are vector values

        # If word exists in our vocab, store its vector
        if word in vocab:
            embedding_matrix[vocab[word]] = vector
```

```
# Maximum sentence length
max_len = 50

# Function to convert sentence into number sequence
def encode(text):
    tokens = text.split()  # split words
    seq = [vocab.get(w,1) for w in tokens]  # convert words to numbers

    # Add padding if sentence is short
    seq = seq[:max_len] + [0]*(max_len-len(seq))

    return seq

# Convert all sentences
X = np.array([encode(t) for t in df["text"]])
```

```
# Labels
y = df["label"]
```

```
# Split into training and testing data
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.2,random_state=42)
```

```
# Custom dataset class
class TextDataset(Dataset):
    def __init__(self,X,y):
        self.X = torch.LongTensor(X)  # convert to tensor
        self.y = torch.LongTensor(y.values)

    def __len__(self):
        return len(self.X)  # total samples

    def __getitem__(self,i):
        return self.X[i], self.y[i]  # return one sample

# Create batch loaders
train_loader = DataLoader(TextDataset(X_train,y_train),64,True)
test_loader = DataLoader(TextDataset(X_test,y_test),64)
```

```
class TextCNN(nn.Module):
    def __init__(self):
        super().__init__()

        # Load pretrained embeddings
        self.embedding = nn.Embedding.from_pretrained(
            torch.FloatTensor(embedding_matrix),
            freeze=False
        )

        # 1D convolution layer
        self.conv = nn.Conv1d(100,128,5)

        # Max pooling layer
        self.pool = nn.MaxPool1d(2)

        # Fully connected layer
        self.fc = nn.Linear(128*23,2)

    def forward(self,x):
        x = self.embedding(x)  # convert words to vectors
        x = x.permute(0,2,1)  # reshape for CNN
        x = self.pool(torch.relu(self.conv(x)))  # apply CNN
        x = x.view(x.size(0),-1)  # flatten
        return self.fc(x)  # final output

# Create model object
model = TextCNN()
```

```
# Loss function
criterion = nn.CrossEntropyLoss()

# Optimizer
optimizer = optim.Adam(model.parameters(),0.001)

# Training loop
for epoch in range(8):
    for X,y in train_loader:
        optimizer.zero_grad()  # reset gradients
        loss = criterion(model(X),y)  # compute loss
        loss.backward()  # backpropagation
        optimizer.step()  # update weights
```

```
preds=[]
true=[]
```

```
with torch.no_grad():  # no gradient needed
    for X,y in test_loader:
        p = torch.argmax(model(X),1)  # predicted class
        preds.extend(p.numpy())
        true.extend(y.numpy())

# Print accuracy
print("Accuracy:",accuracy_score(true,preds))

# Print classification metrics
print(classification_report(true,preds))

# Print confusion matrix
print(confusion_matrix(true,preds))
```

```
Accuracy: 0.9838565022421525
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       966
           1       0.96      0.91      0.94       149

    accuracy                           0.98      1115
   macro avg       0.98      0.95      0.96      1115
weighted avg       0.98      0.98      0.98      1115

[[961   5]
 [ 13 136]]
```