Lab 12.4 — Text Classification using 1D CNN with Pretrained Word Embeddings

Name: Donikela Laxman
Roll No: 2403A51283 **Course:** Natural Language Processing
Dataset: SMS Spam Collection
Embedding: GloVe 6B 100D

```
"""
STEP 2 — Import Required Libraries

This cell imports libraries required for:

• Data manipulation → numpy, pandas
• Text preprocessing → re, string
• Deep learning → torch
• Evaluation → sklearn
• Vocabulary building → collections
"""

import numpy as np
import pandas as pd
import re
import string
import torch
import torch.nn as nn
import torch.optim as optim

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report, confus

from collections import Counter
from tqdm import tqdm
```

```
"""
STEP 3 — Load and Explore Dataset

Dataset: SMS Spam Collection
Columns:
• label → spam / ham
• text → message content
"""

df = pd.read_csv(
    "/content/SMSSpamCollection (1)",
    sep='\t',
    names=["label", "text"]
```

```
)

print("Dataset Shape:", df.shape)
df.head()
```

```
Dataset Shape: (5572, 2)
```

	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...

Next steps: Generate code with df | New interactive sheet

```
"""
Check class distribution
"""

df['label'].value_counts()
```

label	count
ham	4825
spam	747

dtype: int64

```
"""
STEP 4 — Text Preprocessing

Cleaning operations:
• Lowercasing
• Remove numbers
• Remove punctuation
• Strip spaces
"""

def clean_text(text):

    text = text.lower()
    text = re.sub(r'\d+', '', text)
    text = text.translate(str.maketrans('', '', st
    text = text.strip()
```

```
    return text


df['clean_text'] = df['text'].apply(clean_text)

df.head()
```

	label	text	clean_text
0	ham	Go until jurong point, crazy.. Available only ...	go until jurong point crazy available only in ...
1	ham	Ok lar... Joking wif u oni...	ok lar joking wif u oni
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...	free entry in a wkly comp to win fa cup final...
3	ham	U dun say so early hor... U c already then say...	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...	nah i dont think he goes to usf he lives aroun...

Next steps: Generate code with df | New interactive sheet

```
"""
Tokenize each sentence into words
"""

tokenized_texts = df['clean_text'].apply(lambda x: x.split())

tokenized_texts.head()
```

	clean_text
0	[go, until, jurong, point, crazy, available, o...
1	[ok, lar, joking, wif, u, oni]
2	[free, entry, in, a, wkly, comp, to, win, fa, ...
3	[u, dun, say, so, early, hor, u, c, already, t...
4	[nah, i, dont, think, he, goes, to, usf, he, l...

dtype: object

```
"""
STEP 5 — Create Vocabulary

We assign index to each unique word.
Special tokens:
• <PAD> → 0
• <UNK> → 1
"""

all_words = [word for tokens in tokenized_texts for word in tokens]

word_counts = Counter(all_words)

vocab = {
    word: idx + 2
    for idx, (word, _) in enumerate(word_counts.items())
}

vocab["<PAD>"] = 0
vocab["<UNK>"] = 1

vocab_size = len(vocab)

print("Vocabulary Size:", vocab_size)
```

```
Vocabulary Size: 8709
```

```
"""
Convert tokens → index sequences
"""

def encode(tokens, vocab):
    return [
        vocab.get(word, vocab["<UNK>"])
        for word in tokens
    ]

encoded_texts = tokenized_texts.apply(
    lambda x: encode(x, vocab)
)
```

```
"""
Pad sequences to fixed length
"""

MAX_LEN = 50

def pad_sequence(seq, max_len):

    if len(seq) < max_len:
        seq += [0] * (max_len - len(seq))
    else:
```

```
            seq = seq[:max_len]

        return seq


    padded_texts = np.array([
        pad_sequence(seq, MAX_LEN)
        for seq in encoded_texts
    ])

    padded_texts.shape
```

```
(5572, 50)
```

```
"""
Convert labels → numeric

Ham  → 0
Spam → 1
"""

labels = df['label'].map({
    "ham": 0,
    "spam": 1
}).values
```

```
"""
STEP 6 — Split dataset
"""

X_train, X_test, y_train, y_test = train_test_split(
    padded_texts,
    labels,
    test_size=0.2,
    random_state=42
)

print("Train Size:", len(X_train))
print("Test Size:", len(X_test))
```

```
Train Size: 4457
Test Size: 1115
```

```
"""
Download pretrained GloVe embeddings
"""

!wget http://nlp.stanford.edu/data/glove.6B.zip
!unzip glove.6B.zip
```

```
--2026-02-19 04:26:23--  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... con
HTTP request sent, awaiting response... 302 Found
Location: https://nlp.stanford.edu/data/glove.6B.zip [following]
--2026-02-19 04:26:23--  https://nlp.stanford.edu/data/glove.6B.zip
Connecting to nlp.stanford.edu (nlp.stanford.edu)|171.64.67.140|:443... co
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://downloads.cs.stanford.edu/nlp/data/glove.6B.zip [followi
--2026-02-19 04:26:24--  https://downloads.cs.stanford.edu/nlp/data/glove.
Resolving downloads.cs.stanford.edu (downloads.cs.stanford.edu)... 171.64.
Connecting to downloads.cs.stanford.edu (downloads.cs.stanford.edu)|171.64
HTTP request sent, awaiting response... 200 OK
Length: 862182613 (822M) [application/zip]
Saving to: ‘glove.6B.zip’

glove.6B.zip        100%[===================>] 822.24M   661KB/s    in 25m

2026-02-19 04:52:07 (546 KB/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
```

```
"""
Load GloVe word vectors (100D)
"""

embedding_dim = 100
glove_path = "glove.6B.100d.txt"

embeddings_index = {}

with open(glove_path, encoding="utf8") as f:

    for line in tqdm(f):

        values = line.split()
        word = values[0]
        vector = np.asarray(values[1:], dtype='float32')

        embeddings_index[word] = vector

print("Total vectors loaded:", len(embeddings_index))
```

```
400000it [00:13, 29406.58it/s]Total vectors loaded: 400000
```

```
"""
Map vocabulary → GloVe vectors

If word not found → zero vector
```

```
"""

embedding_matrix = np.zeros(
    (vocab_size, embedding_dim)
)

for word, idx in vocab.items():

    vector = embeddings_index.get(word)

    if vector is not None:
        embedding_matrix[idx] = vector
```

```
"""
STEP 7 — 1D CNN Model
Using pretrained embeddings
"""

class TextCNN(nn.Module):

    def __init__(
        self,
        vocab_size,
        embedding_dim,
        embedding_matrix
    ):
        super(TextCNN, self).__init__()

        self.embedding = nn.Embedding(
            vocab_size,
            embedding_dim
        )

        self.embedding.weight.data.copy_(
            torch.from_numpy(embedding_matrix)
        )

        self.embedding.weight.requires_grad = False

        self.conv = nn.Conv1d(
            in_channels=embedding_dim,
            out_channels=128,
            kernel_size=3
        )

        self.relu = nn.ReLU()
        self.pool = nn.MaxPool1d(2)

        self.fc = nn.Linear(128 * 24, 2)


    def forward(self, x):
```

```
        x = self.embedding(x)
        x = x.permute(0, 2, 1)

        x = self.conv(x)
        x = self.relu(x)
        x = self.pool(x)

        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return x
```

```
model = TextCNN(
    vocab_size,
    embedding_dim,
    embedding_matrix
)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(
    model.parameters(),
    lr=0.001
)
```

```
"""
Convert numpy arrays → PyTorch tensors
"""

X_train_t = torch.LongTensor(X_train)
X_test_t  = torch.LongTensor(X_test)

y_train_t = torch.LongTensor(y_train)
y_test_t  = torch.LongTensor(y_test)
```

```
"""
STEP 8 — Train Model
"""

epochs = 5
batch_size = 64

for epoch in range(epochs):

    permutation = torch.randperm(
        X_train_t.size(0)
    )

    total_loss = 0

    for i in range(
```

```
            0,
            X_train_t.size(0),
            batch_size
        ):

            indices = permutation[i:i+batch_size]

            batch_x = X_train_t[indices]
            batch_y = y_train_t[indices]

            optimizer.zero_grad()

            outputs = model(batch_x)
            loss = criterion(outputs, batch_y)

            loss.backward()
            optimizer.step()

            total_loss += loss.item()

        print(
            f"Epoch {epoch+1}, Loss: {total_loss:.4f}"
        )
```

```
Epoch 1, Loss: 14.9339
Epoch 2, Loss: 5.6697
Epoch 3, Loss: 3.7304
Epoch 4, Loss: 2.5621
Epoch 5, Loss: 1.7596
```

```
"""
STEP 9 — Evaluate Model
"""

model.eval()

with torch.no_grad():

    outputs = model(X_test_t)

    predictions = torch.argmax(
        outputs,
        dim=1
    )

accuracy = accuracy_score(
    y_test,
    predictions
)

print("Accuracy:", accuracy)
```

```
Accuracy: 0.9730941704035875
```

```
"""
Classification metrics
"""

print(
    classification_report(
        y_test,
        predictions
    )
)
```

```
              precision    recall  f1-score   support

           0       0.98      0.99      0.98       966
           1       0.93      0.86      0.90       149

    accuracy                           0.97      1115
   macro avg       0.96      0.92      0.94      1115
weighted avg       0.97      0.97      0.97      1115
```

```
"""
Confusion Matrix
"""

confusion_matrix(
    y_test,
    predictions
)
```

```
array([[957,   9],
       [ 21, 128]])
```

ANALYSIS

Pretrained embeddings improved semantic understanding of words by representing them in dense vector space. The CNN model converged faster due to prior linguistic knowledge from GloVe. Accuracy and F1-score improved compared to random embeddings. The model effectively captured contextual patterns in spam messages. However, embedding loading increased computation time. Unknown words were mapped to zero vectors, slightly affecting performance. Overall, pretrained embeddings enhanced classification efficiency and generalization.