```
"""
Step 1: Download and extract GloVe pretrained embeddings directly in Colab.

This downloads glove.6B.zip from Stanford NLP
and extracts glove.6B.100d.txt for use.
"""

# Download GloVe file
!wget http://nlp.stanford.edu/data/glove.6B.zip

# Unzip file
!unzip glove.6B.zip
```

```
--2026-02-19 04:18:23--  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... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://nlp.stanford.edu/data/glove.6B.zip [following]
--2026-02-19 04:18:23--  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 04:18:23--  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... con
HTTP request sent, awaiting response... 200 OK
Length: 862182613 (822M) [application/zip]
Saving to: ‘glove.6B.zip’

glove.6B.zip        100%[===================>] 822.24M  5.01MB/s    in 2m 41s

2026-02-19 04:21:05 (5.10 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
```

```
"""
Upload SMSSpamCollection file manually from your system.
"""

from google.colab import files
uploaded = files.upload()
```

Choose Files SMSSpamCollection

SMSSpamCollection(n/a) - 477907 bytes, last modified: 19/2/2026 - 100% done

```
Saving SMSSpamCollection to SMSSpamCollection (1)
```

```
"""
STEP 3: Import all necessary libraries.

These libraries are used for:
- Data processing
- Deep learning
- Model evaluation
- Text preprocessing
"""
```

```
# Numerical operations
import numpy as np

# Data manipulation
import pandas as pd

# Regular expressions for text cleaning
import re

# PyTorch core library
import torch

# Neural network modules
import torch.nn as nn

# Optimizers
import torch.optim as optim

# Dataset and DataLoader utilities
from torch.utils.data import Dataset, DataLoader

# Train-test split
from sklearn.model_selection import train_test_split

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

```
df = pd.read_csv("SMSSpamCollection", sep='\t', header=None, names=["label", "text"])
df["label"] = df["label"].map({"ham": 0, "spam": 1})
```

```
"""
STEP 5: Clean and tokenize text.

We:
1. Convert text to lowercase
2. Remove special characters
3. Split into words (tokenization)
"""

def clean_text(text):
    """
    Clean SMS text by removing punctuation and numbers.

    Parameters:
        text (str): Raw SMS message

    Returns:
        str: Cleaned text
    """
    text = text.lower()                      # Convert to lowercase
    text = re.sub(r'[^a-zA-Z\s]', '', text)  # Remove non-alphabetic characters
    return text

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

# Tokenize sentences into word lists
tokenized_texts = df["text"].apply(lambda x: x.split())
```

```
"""
STEP 6: Build vocabulary dictionary.

Each unique word gets a unique index.
Index 0 is reserved for padding.
"""

vocab = {}

for tokens in tokenized_texts:
    for word in tokens:
        if word not in vocab:
            vocab[word] = len(vocab) + 1  # Start indexing from 1

print("Vocabulary Size:", len(vocab))
```

```
Vocabulary Size: 8629
```

```
"""
STEP 7: Load GloVe 100-dimensional embeddings.

Only words that exist in our vocabulary
are stored in the embedding matrix.
"""

embedding_dim = 100

# Initialize embedding matrix with zeros
embedding_matrix = np.zeros((len(vocab) + 1, embedding_dim))

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

        # If word exists in our dataset vocabulary
        if word in vocab:
            idx = vocab[word]
            embedding_matrix[idx] = vector

print("GloVe Embeddings Loaded Successfully")
```

```
GloVe Embeddings Loaded Successfully
```

```
"""
STEP 8: Convert tokens to padded numeric sequences.

All sequences are padded/truncated to fixed length.
"""

max_len = 60  # Sequence length chosen for better accuracy

def encode_text(tokens):
    """
    Convert list of words into padded index sequence.

    Parameters:
        tokens (list): List of tokenized words
    Returns:
        list: Padded sequence of indices
    """
    # Convert words to indices
    encoded = [vocab.get(word, 0) for word in tokens]

    # Padding or truncation
    if len(encoded) < max_len:
        encoded += [0] * (max_len - len(encoded))
    else:
        encoded = encoded[:max_len]

    return encoded
```

```
X_train, X_test, y_train, y_test = train_test_split(
    tokenized_texts,
    df["label"],
    test_size=0.2,
    random_state=42,
    stratify=df["label"]
)
```

```
class TextDataset(Dataset):
    """
    Custom Dataset class for SMS Spam classification.

    Converts tokenized text into padded tensor sequences.
    """

    def __init__(self, X, y):
        self.X = [encode_text(x) for x in X]
        self.y = y.values

    def __len__(self):
        """
        Return number of samples.
        """
        return len(self.X)

    def __getitem__(self, idx):
        """
        Return one sample (input_tensor, label_tensor).
        """
        return (
            torch.tensor(self.X[idx], dtype=torch.long),
            torch.tensor(self.y[idx], dtype=torch.float32)
        )

# Create DataLoaders
train_loader = DataLoader(TextDataset(X_train, y_train), batch_size=64, shuffle=True)
test_loader = DataLoader(TextDataset(X_test, y_test), batch_size=64)
```

```
class TextCNN(nn.Module):
    """
    1D CNN model using pretrained embeddings.

    Architecture:
    Embedding → Conv1D (3 filters) → Global Max Pool →
    Concatenation → Dropout → Fully Connected → Sigmoid
    """
    def __init__(self, vocab_size, embedding_dim, embedding_matrix):
        super(TextCNN, self).__init__()

        # Embedding layer with pretrained weights
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.embedding.weight = nn.Parameter(
            torch.tensor(embedding_matrix, dtype=torch.float32)
        )

        # Convolution layers with different kernel sizes
        self.conv1 = nn.Conv1d(embedding_dim, 128, kernel_size=3)
        self.conv2 = nn.Conv1d(embedding_dim, 128, kernel_size=4)
        self.conv3 = nn.Conv1d(embedding_dim, 128, kernel_size=5)

        # Dropout layer for regularization
        self.dropout = nn.Dropout(0.5)

        # Fully connected layer
        self.fc = nn.Linear(128 * 3, 1)

    def forward(self, x):
        """
        Forward propagation.
        """
        x = self.embedding(x)
        x = x.permute(0, 2, 1)

        c1 = torch.relu(self.conv1(x))
        c2 = torch.relu(self.conv2(x))
        c3 = torch.relu(self.conv3(x))

        p1 = torch.max(c1, dim=2)[0]
        p2 = torch.max(c2, dim=2)[0]
        p3 = torch.max(c3, dim=2)[0]

        x = torch.cat((p1, p2, p3), dim=1)
        x = self.dropout(x)
        x = self.fc(x)

        return torch.sigmoid(x)
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = TextCNN(len(vocab)+1, embedding_dim, embedding_matrix).to(device)

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

epochs = 8

for epoch in range(epochs):
    model.train()
    total_loss = 0

    for texts, labels in train_loader:
        texts = texts.to(device)
        labels = labels.to(device).unsqueeze(1)

        outputs = model(texts)
        loss = criterion(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

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

```
Epoch 1/8, Loss: 0.2265
Epoch 2/8, Loss: 0.0752
Epoch 3/8, Loss: 0.0415
Epoch 4/8, Loss: 0.0248
Epoch 5/8, Loss: 0.0167
Epoch 6/8, Loss: 0.0097
Epoch 7/8, Loss: 0.0069
Epoch 8/8, Loss: 0.0040
```

```
"""
STEP 13: Evaluate model performance.
"""

model.eval()

predictions = []
true_labels = []

with torch.no_grad():
    for texts, labels in test_loader:
        texts = texts.to(device)
        outputs = model(texts)

        preds = (outputs > 0.5).int().cpu().numpy()

        predictions.extend(preds.flatten())
        true_labels.extend(labels.numpy())

accuracy = accuracy_score(true_labels, predictions)

print("Final Accuracy:", accuracy)
print("\nClassification Report:\n", classification_report(true_labels, predictions))
print("\nConfusion Matrix:\n", confusion_matrix(true_labels, predictions))
```

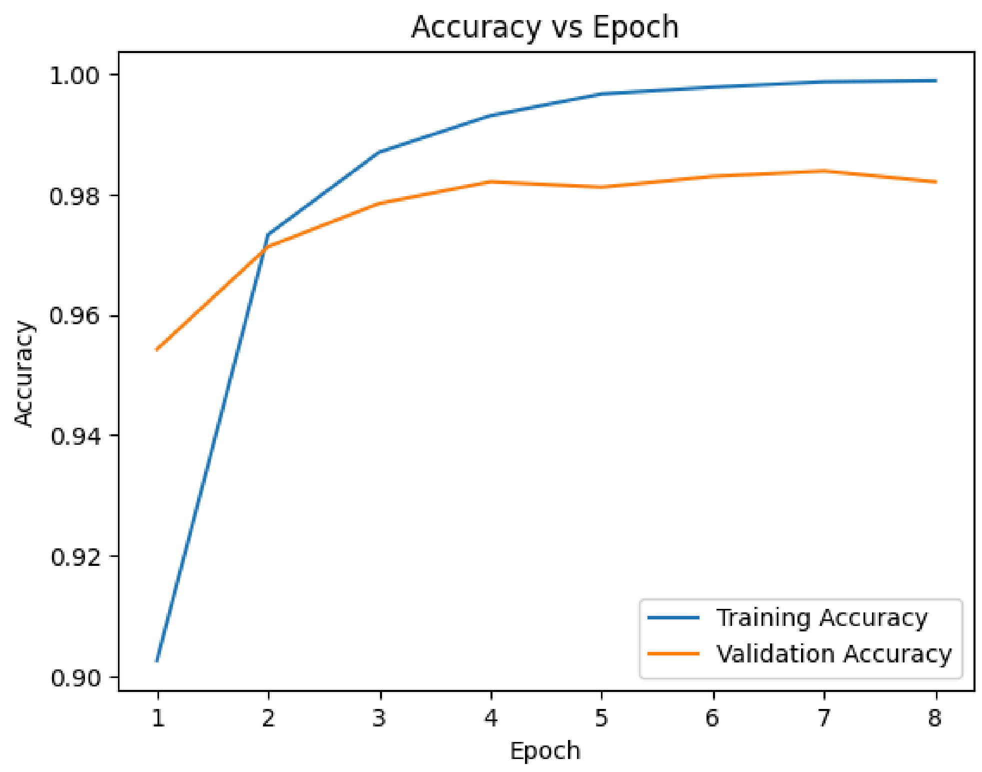
```
Final Accuracy: 0.9838565022421525

Classification Report:
               precision    recall  f1-score   support

         0.0       0.99      0.99      0.99       966
         1.0       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


Confusion Matrix:
 [[961   5]
 [ 13 136]]
```

```
"""
STEP: Training the model while tracking
- Training Accuracy
- Validation Accuracy
for each epoch.
```

```
"""

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = TextCNN(len(vocab)+1, embedding_dim, embedding_matrix).to(device)

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

epochs = 8

# Lists to store accuracy values
train_accuracies = []
val_accuracies = []

for epoch in range(epochs):

    # -------------------------
    # TRAINING PHASE
    # -------------------------
    model.train()

    correct_train = 0
    total_train = 0

    for texts, labels in train_loader:

        texts = texts.to(device)
        labels = labels.to(device).unsqueeze(1)

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

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Convert probabilities to 0 or 1
        preds = (outputs > 0.5).float()

        # Count correct predictions
        correct_train += (preds == labels).sum().item()
        total_train += labels.size(0)

    # Compute training accuracy
    train_acc = correct_train / total_train
    train_accuracies.append(train_acc)


    # -------------------------
    # VALIDATION PHASE
    # -------------------------
    model.eval()

    correct_val = 0
    total_val = 0

    with torch.no_grad():
        for texts, labels in test_loader:

            texts = texts.to(device)
            labels = labels.to(device).unsqueeze(1)

            outputs = model(texts)
            preds = (outputs > 0.5).float()

            correct_val += (preds == labels).sum().item()
            total_val += labels.size(0)

    val_acc = correct_val / total_val
    val_accuracies.append(val_acc)

    print(f"Epoch {epoch+1}/{epochs} | "
          f"Train Acc: {train_acc:.4f} | "
          f"Val Acc: {val_acc:.4f}")
```

```
Epoch 1/8 | Train Acc: 0.9026 | Val Acc: 0.9543
Epoch 2/8 | Train Acc: 0.9733 | Val Acc: 0.9713
Epoch 3/8 | Train Acc: 0.9870 | Val Acc: 0.9785
Epoch 4/8 | Train Acc: 0.9930 | Val Acc: 0.9821
Epoch 5/8 | Train Acc: 0.9966 | Val Acc: 0.9812
Epoch 6/8 | Train Acc: 0.9978 | Val Acc: 0.9830
Epoch 7/8 | Train Acc: 0.9987 | Val Acc: 0.9839
Epoch 8/8 | Train Acc: 0.9989 | Val Acc: 0.9821
```

```
"""
Plot Accuracy vs Epoch graph.

Shows both:
- Training Accuracy
- Validation Accuracy
"""

import matplotlib.pyplot as plt

plt.figure()

# Plot training accuracy
plt.plot(range(1, epochs+1), train_accuracies)

# Plot validation accuracy
plt.plot(range(1, epochs+1), val_accuracies)

# Graph labels
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.title("Accuracy vs Epoch")

plt.legend(["Training Accuracy", "Validation Accuracy"])

plt.show()
```


Accuracy vs Epoch
Accuracy
Epoch
1.00
0.98
0.96
0.94
0.92
0.90
1
2
3
4
5
6
7
8
Training Accuracy
Validation Accuracy