# DL- Developing a Deep Learning Model for NER using LSTM

## AIM

To develop an LSTM-based model for recognizing the named entities in the text.

## THEORY

An LSTM-based model for recognizing named entities is a type of neural network that uses Long Short-Term Memory (LSTM) layers to identify and classify proper names and entities within a text, such as person names, locations, organizations, dates, etc. It is commonly employed in Named Entity Recognition (NER) tasks because LSTMs are effective at capturing sequential dependencies and context within text. Typically, these models process tokenized input data, learn contextual representations, and output labels for each token indicating whether it belongs to a specific entity type. This approach improves the accuracy of extracting meaningful information from unstructured text data.

## DESIGN STEPS

### STEP 1:

Load data, create word/tag mappings, and group sentences.

### STEP 2:

Convert sentences to index sequences, pad to fixed length, and split into training/testing sets.

### STEP 3:

Define dataset and DataLoader for batching.

### STEP 4:

Build a bidirectional LSTM model for sequence tagging.

### STEP 5:

Train the model over multiple epochs, tracking loss.

### STEP 6:

Evaluate model accuracy, plot loss curves, and visualize predictions on a sample.

## PROGRAM

### Name: RESHMA C

### Register Number: 212223040168

```
import pandas as pd
import torch
import torch.nn as nn
import numpy as np
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from torch.nn.utils.rnn import pad_sequence
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)

device=torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device:{device}")
data = pd.read_csv("ner_dataset.csv", encoding="latin1").ffill()
words = list(data["Word"].unique())
tags = list(data["Tag"].unique())
if "ENDPAD" not in words:
    words.append("ENDPAD")
```

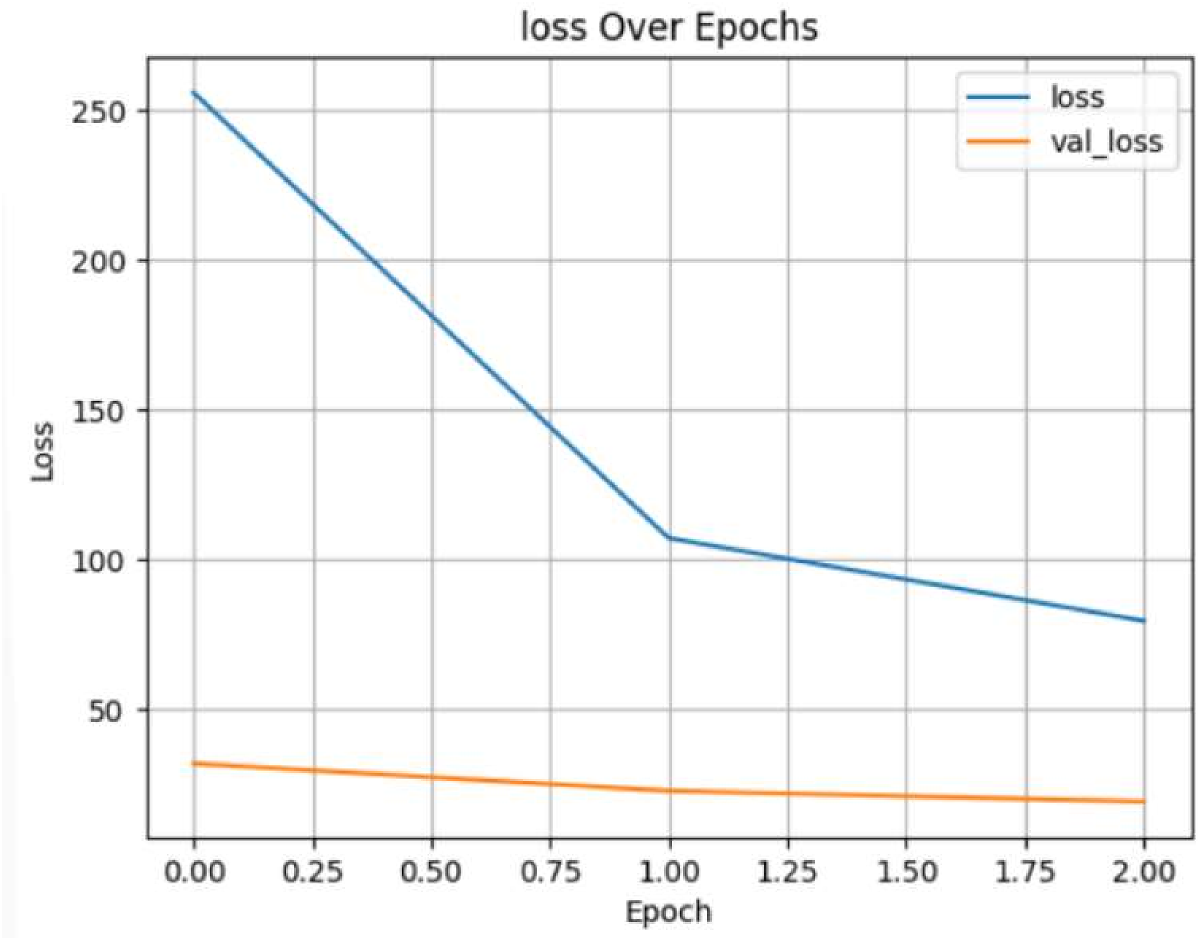
```
word2idx = {w: i + 1 for i, w in enumerate(words)}  # Start indexing from 1
tag2idx = {t: i for i, t in enumerate(tags)}
idx2tag = {i: t for t, i in tag2idx.items()}

print("Unique words in corpus:", data['Word'].nunique())
print("Unique tags in corpus:", data['Tag'].nunique())
print("Unique tags are:", tags)

class SentenceGetter:
    def __init__(self, data):
        self.grouped = data.groupby("Sentence #", group_keys=False).apply(
            lambda s: [(w, t) for w, t in zip(s["Word"], s["Tag"])]
        )
        self.sentences = list(self.grouped)
getter = SentenceGetter(data)
sentences = getter.sentences
print(sentences[35])
X = [[word2idx[w] for w, t in s] for s in sentences]
y = [[tag2idx[t] for w, t in s] for s in sentences]
plt.hist([len(s) for s in sentences], bins=50)
plt.show()
max_len = 50
X_pad = pad_sequence([torch.tensor(seq) for seq in X], batch_first=True, padding
y_pad = pad_sequence([torch.tensor(seq) for seq in y], batch_first=True, padding
X_pad = X_pad[:, :max_len]
y_pad = y_pad[:, :max_len]
print(X_pad[0])
print(y_pad[0])

X_train, X_test, y_train, y_test = train_test_split(X_pad, y_pad, test_size=0.2,
class NERDataset(Dataset):
    def __init__(self, X, y):
        self.X = X
        self.y = y
    def __len__(self):
        return len(self.X)
    def __getitem__(self, idx):
        return {
            "input_ids": self.X[idx],
            "labels": self.y[idx]
        }
train_loader = DataLoader(NERDataset(X_train, y_train), batch_size=32, shuffle=T
test_loader = DataLoader(NERDataset(X_test, y_test), batch_size=32)

class BiLSTMTagger(nn.Module):
    def __init__(self, vocab_size, tagset_size, embedding_dim=50, hidden_dim=10
        super(BiLSTMTagger, self).__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.dropout = nn.Dropout(0.1)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, batch_first=True, bidire
```

```
        self.fc = nn.Linear(hidden_dim * 2, tagset_size)

    def forward(self, x):
        x = self.embedding(x)
        x = self.dropout(x)
        x, _ = self.lstm(x)
        return self.fc(x)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = BiLSTMTagger(len(word2idx) + 1, len(tag2idx)).to(device)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

def train_model(model, train_loader, test_loader, loss_fn, optimizer, epochs=3):
    train_losses, val_losses = [], []
    for epoch in range(epochs):
        model.train()
        total_loss = 0
        for batch in train_loader:
            input_ids = batch["input_ids"].to(device)
            labels = batch["labels"].to(device)
            optimizer.zero_grad()
            outputs = model(input_ids)
            loss = loss_fn(outputs.view(-1, len(tag2idx)), labels.view(-1))
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        train_losses.append(total_loss)

        model.eval()
        val_loss = 0
        with torch.no_grad():
            for batch in test_loader:
                input_ids = batch["input_ids"].to(device)
                labels = batch["labels"].to(device)
                outputs = model(input_ids)
                loss = loss_fn(outputs.view(-1, len(tag2idx)), labels.view(-1))
                val_loss += loss.item()
        val_losses.append(val_loss)
        print(f"Epoch {epoch+1}: Train Loss = {total_loss:.4f}, Val Loss = {val_
    return train_losses, val_losses


def evaluate_model(model, test_loader, X_test, y_test):
    model.eval()
    true_tags, pred_tags = [], []
    with torch.no_grad():
        for batch in test_loader:
            input_ids = batch["input_ids"].to(device)
            labels = batch["labels"].to(device)
            outputs = model(input_ids)
```

```
            preds = torch.argmax(outputs, dim=-1)
            for i in range(len(labels)):
                for j in range(len(labels[i])):
                    if labels[i][j] != tag2idx["O"]:
                        true_tags.append(idx2tag[labels[i][j].item()])
                        pred_tags.append(idx2tag[preds[i][j].item()])

train_losses, val_losses = train_model(model, train_loader, test_loader, loss_fn
evaluate_model(model, test_loader, X_test, y_test)


history_df = pd.DataFrame({"loss": train_losses, "val_loss": val_losses})
history_df.plot(title="loss Over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.grid(True)
plt.show()

i = 125
model.eval()
sample = X_test[i].unsqueeze(0).to(device)
output = model(sample)
preds = torch.argmax(output, dim=-1).squeeze().cpu().numpy()
true = y_test[i].numpy()


print("{:<15} {:<10} {}".format("Word", "True", "Pred"))
print("-" * 40)

for w_id, true_tag, pred_tag in zip(X_test[i], y_test[i], preds):
    if w_id.item() != word2idx["ENDPAD"]:
        word = words[w_id.item() - 1]
        true_label = tags[true_tag.item()]
        pred_label = tags[pred_tag]
        print(f"{word: <15} {true_label: <10} {pred_label}")
```

### OUTPUT

## Loss Vs Epoch Plot




## Sample Text Prediction

```
Word           True      Pred
----------------------------------------



Palestinian    B-gpe     B-gpe
officials      O         O
say            O         O
two            O         O
Palestinians   B-gpe     B-gpe
have           O         O
been           O         O
killed         O         O
in             O         O
an             O         O
accidental     O         O
explosion      O         O
in             O         O
a              O         O
West           B-org     B-org
Bank           I-org     I-org
refugee        O         O
camp           O         O
.              O         O
```

## RESULT

Thus, an LSTM-based model for recognizing the named entities in the text has been developed successfully.