Understanding Word Prediction: A Hands-on Guide to Building RNN-based Models

Minor in AI - IIT ROPAR

27th March, 2025

A Story of Word Prediction

Imagine you're typing a message on WhatsApp or composing an email in Gmail. As you type, these applications seem to "understand" what you are trying to say, offering suggestions for the next word or even completing your sentence. These applications make use of advanced Natural Language Processing (NLP) models that are trained on vast amounts of text data to predict the most likely word or phrase based on the context of the sentence. The models behind this feature are built using techniques like word embeddings and Recurrent Neural Networks (RNNs). In this session, we explored how these techniques work and how we can use them to build models that predict the next word in a sequence given a few words as input. We also examined how relationships between words are learned by these models.

Word Embeddings: Transforming Words into Vectors

- **Definition**: Word embeddings are a technique used to represent words as vectors in a high-dimensional space. These vectors capture the semantic meaning of words, enabling models to understand the context in which words are used. Each word is assigned a vector of fixed length, where the dimension can be adjusted based on the specific requirements of the task at hand.
- **Basic Technique – One-Hot Encoding**: The simplest form of word embedding is *one-hot encoding*, where each word is represented by a vector as long as the vocabulary. For example, for a vocabulary consisting of five words, a word like "cat" could be represented as $[1, 0, 0, 0, 0]$, indicating that "cat" corresponds to the first word in the vocabulary. While easy to understand and implement, one-hot encoding has limitations—it does not capture any relationships between words. For instance, "cat" and "dog" are both animals, but one-hot encoding treats them as entirely separate entities.

Limitations of One-Hot Encoding

- **No Semantic Relationship**: One-hot encoding treats each word as independent, ignoring relationships between similar words. For example, "dog" and "cat" should be close in the vector space since they are both animals, yet one-hot encoding doesn't reflect this similarity.
- **Scalability Issue**: As the vocabulary grows, one-hot encoding becomes increasingly inefficient, with sparse vectors for each word. It also fails to capture context, which is critical in tasks like sentiment analysis, text generation, or next-word prediction.

Advanced Word Embedding Techniques: Word2Vec and Skip-Gram

To overcome the limitations of one-hot encoding, we turned to more sophisticated techniques like **Word2Vec**, which is designed to capture word meanings by learning from their contexts. Word2Vec uses two methods: **Skip-Gram** and **CBOW (Continuous Bag of Words)**.

- **Word2Vec – Skip-Gram Model**: The Skip-Gram model works by taking a target word and predicting the surrounding context words. The idea is that words that appear in similar contexts tend to have similar meanings. For example, for the target word "dog," the model would try to predict context words like "bark," "pet," "animal," etc., which often appear around "dog" in sentences.
- **Window Size in Skip-Gram**: The *window size* parameter defines how many surrounding words will be considered for context. A window size of 2 means the model will look at two words before and two words after the target word.
- **Benefits**: Skip-Gram captures semantic relationships between words, making it much more effective than one-hot encoding. It enables the model to represent words like "dog" and "cat" as similar vectors because they often appear in similar contexts.

Recurrent Neural Networks (RNNs) for Sequence Prediction

Once we understood word embeddings, we moved to **Recurrent Neural Networks (RNNs)**, which are used for processing sequences of data, such as text. In tasks like next-word prediction, RNNs are ideal because they can process input sequences while maintaining an internal state that captures information from previous words in the sequence.

- **RNN Training for Next-Word Prediction**: The RNN model learns to predict the next word in a sequence. For example, if the input is "I am going to," the model will try to predict the next word (e.g., "school" or "the"). The RNN uses its internal state (or memory) to understand the relationship between words in the sequence, enabling it to generate plausible predictions.
- **Data Preparation**: To train the RNN, we first preprocess the text by converting it to lowercase, removing non-textual characters (e.g., punctuation), and splitting it into tokens (words). Then, we create training data by selecting sequences of words (e.g., the first 5 words) and using the 6th word as the target for prediction. This process is repeated across the entire corpus, generating multiple training examples.

Training the RNN Model

- **Model Architecture**: The RNN is built using **PyTorch**, a deep learning library. We define the architecture by specifying the **vocabulary size** (the total number of unique words), the **embedding dimension** (the size of the vector used to represent each word), and the **hidden layer dimensions** (the number of units in the RNN's hidden layers).
- **Loss Function**: The model uses **cross-entropy loss** to measure the difference between the predicted word and the actual target word. The optimizer (*Adam optimizer*) is then used to update the weights of the model to minimize the loss, making the model's predictions more accurate over time.

Enhancements: Stop Words and Advanced Embedding

To improve the model's performance, we made several enhancements:

- **Stop Word Removal**: Words like "the," "is," and "and" (known as stop words) appear very frequently in text but don't carry much semantic meaning. By removing stop words during preprocessing, we reduce the size of the vocabulary and allow the model to focus on more meaningful words, improving the model's efficiency and accuracy.
- **Word2Vec for Better Embeddings**: Instead of using basic embeddings, we switched to **Word2Vec**, which uses the Skip-Gram model to learn better word representations. This allows the model to capture word context and semantic meaning more effectively than simple embeddings.

Skip-Gram Model in Word2Vec

- **Skip-Gram in Action**: In the **Skip-Gram** model, given a target word (e.g., ”dog”), the model tries to predict the surrounding context words. If the target word is ”dog,” the model might predict words like ”pet,” ”animal,” ”bark,” depending on the context in which ”dog” appears.
- **Building Word Pairs**: For each target word, the model forms pairs with its surrounding words. For instance, with the word ”dog” as the target and a window size of 2, the context words might be ”pet,” ”animal,” ”bark,” ”canine,” and so on.

Training Process and GPU Utilization

Training the RNN involves feeding the sequences of words into the model in **batches**. For efficiency, the model is trained using **batch processing** (e.g., processing 32 words at a time), and the weights of the model are updated after processing each batch. The training continues for several epochs, and after each epoch, the **loss** is calculated and minimized using backpropagation.

- **Using GPU**: The training of neural networks can be very resource-intensive, especially with large datasets. To speed up the process, we used **Google Colab**, which provides access to a GPU. This significantly reduced the time taken for training.

Model Prediction and Real-World Application

Once the model is trained, we use it to predict the next word in a given sequence. The model can take any sequence of words as input and, based on its training, predict the most likely next word. This ability to predict the next word is similar to features in WhatsApp and Gmail, where the model suggests completions or auto-fills your sentences.

Improvements for Better Performance

- **Using Word2Vec**: Replacing simple word embeddings with **Word2Vec** improved the model’s ability to capture semantic relationships between words. This change helped the model predict words more accurately by learning the context in which words appear.
- **Stop Word Removal**: Removing common but meaningless words (stop words) improved the efficiency of the model, ensuring it focused on words that actually contributed to understanding the context.

Coding Part

Step 1: Setting Up the Environment and Importing Libraries

We start by importing the required libraries. These libraries include PyTorch for creating the neural network, NumPy for numerical operations, and several others for text processing.

```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import random
import re
from collections import Counter
import matplotlib.pyplot as plt
from nltk.corpus import stopwords
from torch.utils.data import DataLoader, TensorDataset
import nltk
```

- `torch`: The core library for building and training deep learning models in Python.

- `torch.nn`: Contains the neural network components such as layers and loss functions.
- `torch.optim`: Provides optimization algorithms such as Adam for training the model.
- `re`: A regular expression library for cleaning text.
- `Counter`: Used to count occurrences of words in the text.
- `matplotlib.pyplot`: For plotting graphs to visualize the results.
- `nltk.corpus.stopwords`: For handling stopwords, which are common words that don't add much meaning.
- `torch.utils.data`: For handling datasets and batching.

Step 2: Setting Random Seeds for Reproducibility

To ensure that our results are consistent each time we run the code, we set random seeds. This ensures that the model's initialization and training remain the same across different runs.

```
torch.manual_seed(182)
random.seed(231)
np.random.seed(248)
```

- **torch.manual_seed()**: Ensures the randomness in PyTorch is the same each time. - **random.seed()**: Ensures Python's built-in random functions are consistent. - **np.random.seed()**: Ensures NumPy's random number generation is fixed.

Step 3: Loading and Preprocessing the Data

Next, we load the text data and preprocess it. The goal is to clean the text by converting it to lowercase, removing unwanted characters, and splitting it into individual words (tokens).

```
file_path = "War_and_Peace.txt"  # Replace with your text file
with open(file_path, "r", encoding="utf-8") as f:
    lines = f.read()

def pre_process_data(text):
    text = text.lower()  # Convert to lowercase
    text = re.sub(r"[^a-z\s]", "", text)  # Remove non-alphabetic characters
    words = text.split()  # Split text into words (tokens)
    return words

tokens = pre_process_data(lines)  # Apply preprocessing
print("Sample␣tokens␣->", tokens[:10])  # Show first 10 tokens (words)
```

- **Lowercasing**: Converts all text to lowercase so that "The" and "the" are treated the same. - **Regex cleaning**: Removes punctuation and numbers using a regular expression. - **Tokenization**: Splits the text into words for further processing.

Step 4: Building Vocabulary and Word Indices

We now build the vocabulary, which consists of all unique words in the text. We also create two dictionaries: one mapping words to indices and another mapping indices to words.

```
word_count = Counter(tokens)

vocab = sorted(word_count.keys())  # Sort the vocabulary alphabetically

word_to_index = {word: i for i, word in enumerate(vocab)}
index_to_word = {i: word for i, word in enumerate(vocab)}

print("Sample␣word␣to␣index␣->", list(word_to_index.items())[:10])
```

- **word_count**: This counts the frequency of each word in the text. - **vocab**: This is the list of unique words, sorted alphabetically. - **word_to_index**: A dictionary mapping each word to a unique index. - **index_to_word**: A reverse dictionary that maps each index back to a word.

Step 5: Creating Sequences for Training

In order to train our model to predict the next word, we need to prepare sequences of words. For each sequence, the model will try to predict the word that comes after it.

```python
seq_length = 5  # Number of words in each sequence

def create_sequence(tokens, seq_length):
    inputs = []
    targets = []

    for i in range(len(tokens) - seq_length):
        seq_in = tokens[i:i + seq_length]  # Input sequence
        seq_out = tokens[i + seq_length]   # Target word (the word after the input
            sequence)

        inputs.append([word_to_index[word] for word in seq_in])  # Convert words to
            indices
        targets.append(word_to_index[seq_out])  # Convert target to index

    return inputs, targets

inputs, targets = create_sequence(tokens, seq_length)  # Create sequences
print(inputs[0])  # Show first input sequence
print(targets[0])  # Show corresponding target word
```

- **Input Sequence**: We take `seq_length` words from the text as input and use the next word as the target. - **Conversion to Indices**: Each word is converted to its corresponding index using the `word_to_index` dictionary.

Step 6: Defining the Deep RNN Model

Now we define the model, which is a Recurrent Neural Network (RNN). The model consists of an embedding layer, an RNN layer, and a fully connected layer that outputs the probabilities of the next word.

```python
class DeepRnn(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_layers):
        super(DeepRnn, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embed_dim)  # Word embeddings
        self.rnn = nn.RNN(embed_dim, hidden_dim, num_layers, batch_first=True)  # RNN
            layer
        self.fc = nn.Linear(hidden_dim, vocab_size)  # Fully connected layer for
            predictions

    def forward(self, x):
        embeddings = self.embeddings(x)  # Convert word indices to embeddings
        output, hidden = self.rnn(embeddings)  # Process with RNN
        output = self.fc(output[:, -1, :])  # Get output from the last RNN time step
        return output
```

- **Embedding Layer**: Converts each word into a fixed-size vector. - **RNN Layer**: Processes sequences of word embeddings and learns relationships over time. - **Fully Connected Layer**: Outputs predictions for the next word.

Step 7: Training the Model

Now, we define the training process. This includes specifying the loss function, optimizer, and data loader. The model is trained by minimizing the cross-entropy loss using the Adam optimizer.

```python
# Hyperparameters
EMBED_DIM = 64
HIDDEN_DIM = 128
NUM_LAYERS = 3
VOCAB_SIZE = len(vocab)
BATCH_SIZE = 32
EPOCHS = 50
LR = 0.001

# Create model and move to the appropriate device (GPU/CPU)
```

```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = DeepRnn(VOCAB_SIZE, EMBED_DIM, HIDDEN_DIM, NUM_LAYERS).to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()  # Cross-entropy loss for classification
optimizer = optim.Adam(model.parameters(), lr=LR)  # Adam optimizer

# Create data loader
train_dataset = TensorDataset(torch.tensor(inputs), torch.tensor(targets))
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
```

- **Cross-Entropy Loss**: Used to measure the difference between the predicted and actual word indices. - **Adam Optimizer**: Used to optimize the model's weights during training. - **DataLoader**: Handles batching of data during training.

Step 8: Training Loop

In this step, we define the training loop. The model is trained for a specified number of epochs. After each epoch, we compute the average loss and update the model's weights.

```python
def train(model, train_loader, criterion, optimizer, epochs):
    model.train()
    loss_values = []

    for epoch in range(epochs):
        total_loss = 0
        for batch_inputs, batch_targets in train_loader:
            batch_inputs, batch_targets = batch_inputs.to(device), batch_targets.to(
                device)

            optimizer.zero_grad()  # Reset gradients
            outputs = model(batch_inputs)  # Forward pass

            loss = criterion(outputs, batch_targets)  # Compute loss
            loss.backward()  # Backpropagation
            optimizer.step()  # Update weights

            total_loss += loss.item()

        avg_loss = total_loss / len(train_loader)
        loss_values.append(avg_loss)
        print(f"Epoch␣[{epoch+1}/{epochs}],␣Loss:␣{avg_loss:.4f}")

    return loss_values

# Train the model
loss_values = train(model, train_loader, criterion, optimizer, EPOCHS)
```

- **Forward Pass**: The model makes predictions based on the current weights. - **Loss Calculation**: We calculate the loss (difference between predicted and actual words). - **Backpropagation**: Gradients are computed, and weights are updated.

Step 9: Predicting the Next Word

Once the model is trained, we can use it to predict the next word in a sequence. This is done by passing a sequence of words through the trained model.

```python
def predict_next_word(model, input_text, seq_length):
    model.eval()

    words = input_text.lower().split()  # Convert input to lowercase and split into
        words
    tokens = [word_to_index[word] if word in word_to_index else 0 for word in words]
    tokens = tokens[-seq_length:]  # Use only the last sequence
    tokens = torch.tensor(tokens).unsqueeze(0).to(device)

    with torch.no_grad():
        output = model(tokens)  # Get the model output
        predicted_index = torch.argmax(output, dim=1).item()  # Get the most probable
            word

    return index_to_word[predicted_index]

# Test with a sample input
sample_input = "deep␣learning␣is␣the␣most␣important␣and␣decisive"
predicted_word = predict_next_word(model, sample_input, seq_length=5)
print(f"Predicted␣next␣word:␣{predicted_word}")
```

- **Prediction**: The input sequence is processed by the model to predict the next word. - **Model Output**: The model outputs a probability distribution, and we select the word with the highest probability.

Key Takeaways

- **Word Prediction Task**: The task is to predict the next word in a sequence given a few input words, which is similar to the word completion or suggestion features in apps like WhatsApp and Gmail.
- **Word Embeddings**: Word embeddings represent words as vectors in a high-dimensional space to capture semantic meanings and context. This helps models understand the relationships between words.
- **Limitations of One-Hot Encoding**: One-hot encoding treats each word as independent and does not capture semantic relationships between words, making it inefficient as the vocabulary grows.
- **Advanced Word Embedding (Word2Vec)**: Word2Vec's Skip-Gram model captures the semantic relationships between words by predicting context words based on a target word, improving upon simple methods like one-hot encoding.
- **Recurrent Neural Networks (RNNs)**: RNNs are ideal for sequence prediction tasks because they process sequences while maintaining an internal state that captures information from previous words.
- **Training the RNN Model**: The model is trained using cross-entropy loss to measure prediction accuracy, and the Adam optimizer is used to update the model's weights and minimize the loss.
- **Enhancements**: Stop word removal reduces the size of the vocabulary and improves model efficiency, while using Word2Vec embeddings enhances the model's ability to capture word meanings and relationships.
- **GPU Utilization**: Using a GPU significantly speeds up the training process, which is especially important when working with large datasets.
- **Model Prediction**: After training, the model predicts the next word in a sequence by processing the input words and outputting the most likely next word based on learned patterns.
- **Practical Use**: The trained model can be used for real-world applications such as text autocompletion and next-word prediction, similar to the predictive text features in modern messaging apps.