Mohammad Idrees Bhat

Tech Skills Trainer | AI/ML Consultant

Advanced NLP Techniques

Word embeddings and transformers

AGENDA

1. Introduction to Advanced NLP
2. Word Embeddings
3. Transformers
4. Building the Sentiment Analyzer (OPTIONAL/ADVANCED)

Mohammad Idrees Bhat

Can acts of kindness sometimes be self-serving, even when we don't realize it?

How do we differentiate between genuine kindness and actions that benefit ourselves?

1. Introduction to Advanced NLP

Advanced Natural Language Processing (NLP) refers to the study and application of more complex and powerful techniques in processing human language, going beyond basic tasks like tokenization, part-of-speech tagging, and named entity recognition.

While traditional NLP methods focused on rule-based or statistical models, advanced NLP incorporates deep learning techniques, which allow for better handling of ambiguous, large-scale, and context-dependent language data.

What We Study in Advanced NLP:

1. **Word Embeddings**:

We study methods like **Word2Vec** and **GloVe** to represent words as vectors in continuous space, capturing semantic relationships. These representations are crucial for understanding word meanings in context.

2. **Transformers**:

The introduction of transformer models, with self-attention mechanisms, revolutionized NLP by enabling models to capture long-range dependencies and understand context better than previous models. This is the foundation for many modern NLP models, including **BERT**, **GPT**, and **T5**.

3. **Generative Models & Language Models**:

We dive into **Generative AI** models like GPT and other transformer-based models. These models are designed to generate human-like text, enabling tasks such as text generation, summarization, translation, and creative writing.

4. **Transfer Learning**:

We often use pre-trained models and fine-tune them on specific tasks. This technique has made it easier to apply powerful models to a variety of language tasks with relatively small amounts of labeled data.

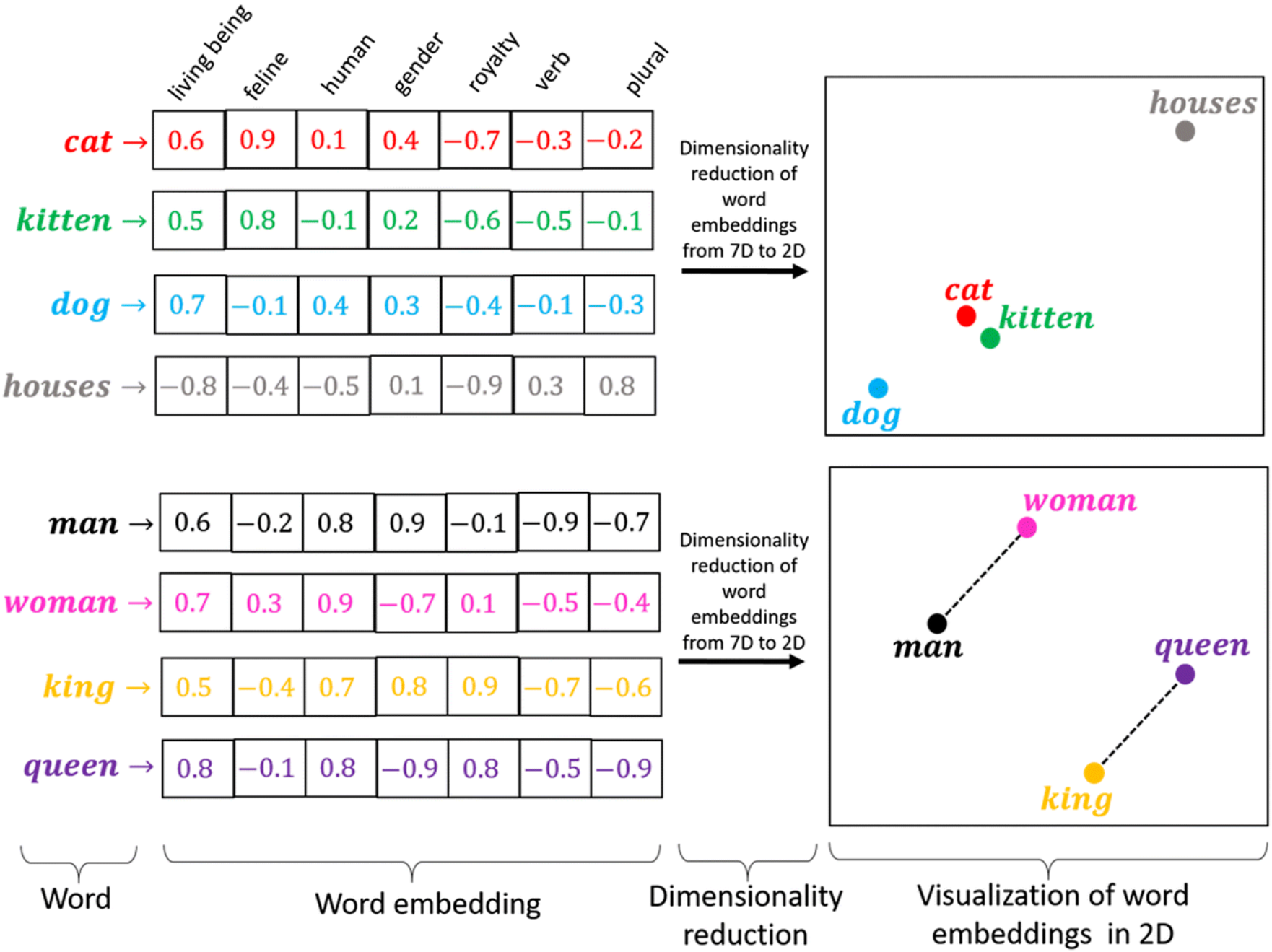
2. Word Embeddings

Learn more from:

- geeks
- ibm
- Medium Article
- Tensorflow

Word embeddings are a way to represent words in numerical form so that computers can understand and process them more efficiently.

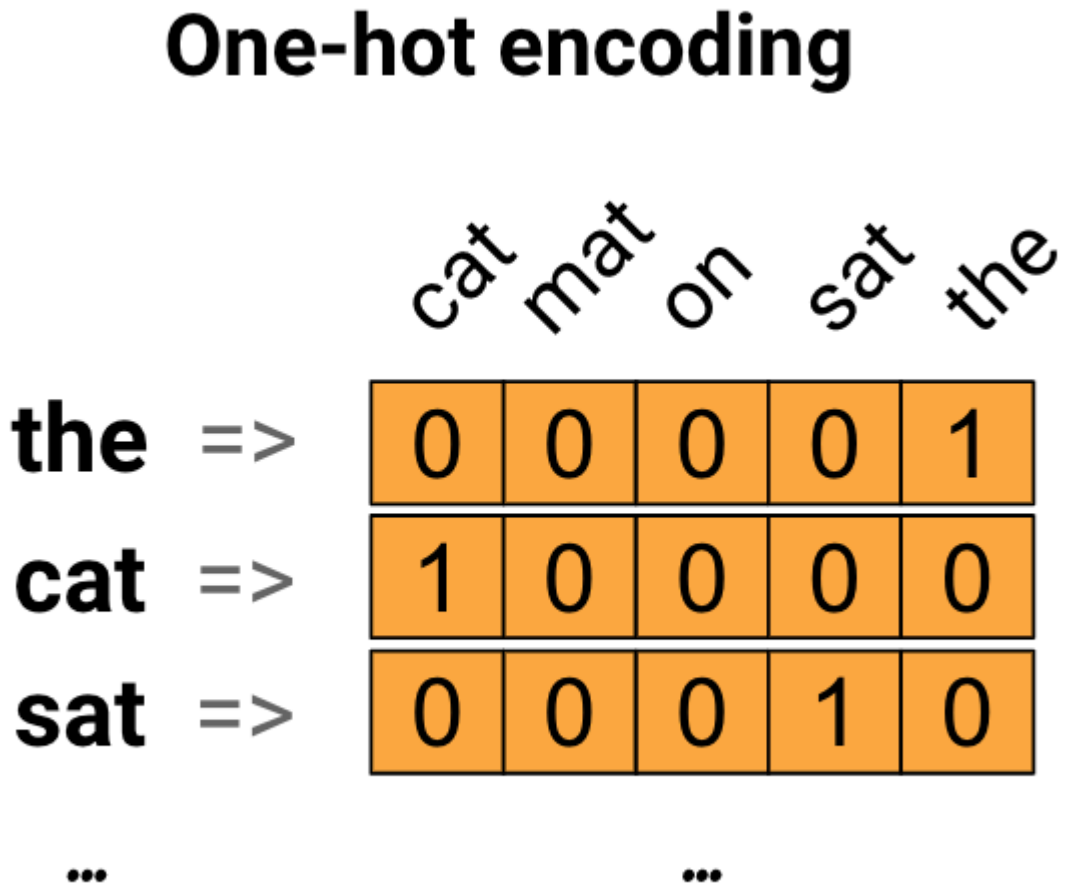
Instead of treating words as individual tokens, word embeddings map each word to a vector (a list of numbers), where the numbers represent the word's meaning in relation to other words.


living being
feline
human
gender
royalty
verb
plural
cat → 0.6 0.9 0.1 0.4 −0.7 −0.3 −0.2
kitten → 0.5 0.8 −0.1 0.2 −0.6 −0.5 −0.1
dog → 0.7 −0.1 0.4 0.3 −0.4 −0.1 −0.3
houses → −0.8 −0.4 −0.5 0.1 −0.9 0.3 0.8
Dimensionality reduction of word embeddings from 7D to 2D
houses
cat
kitten
dog
man → 0.6 −0.2 0.8 0.9 −0.1 −0.9 −0.7
woman → 0.7 0.3 0.9 −0.7 0.1 −0.5 −0.4
king → 0.5 −0.4 0.7 0.8 0.9 −0.7 −0.6
queen → 0.8 −0.1 0.8 −0.9 0.8 −0.5 −0.9
Dimensionality reduction of word embeddings from 7D to 2D
woman
man
queen
king
Word
Word embedding
Dimensionality reduction
Visualization of word embeddings in 2D

In simple terms, word embeddings allow *words with similar meanings* to have similar vector representations. This helps the computer understand relationships between words in a much deeper way than just looking at the words themselves.

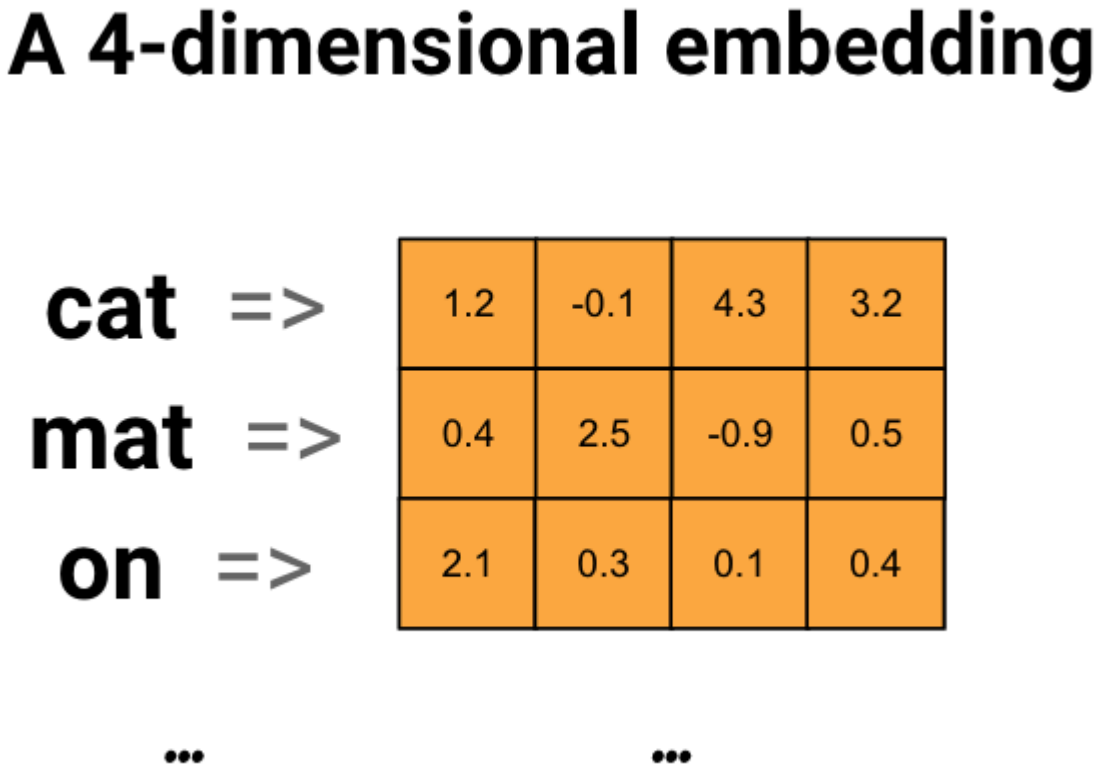
Why Use Word Embeddings?

Traditional methods like one-hot encoding represent each word as a unique vector where only one element is "1" and the rest are "0".



This method doesn’t capture any relationships between words. For example, "cat" and "dog" would be equally distant from each other, even though they are more similar in meaning than "cat" and "car".

With word embeddings, words with similar meanings will be closer in the vector space.



Popular Word Embedding Models

1. **Word2Vec**: Learn more...

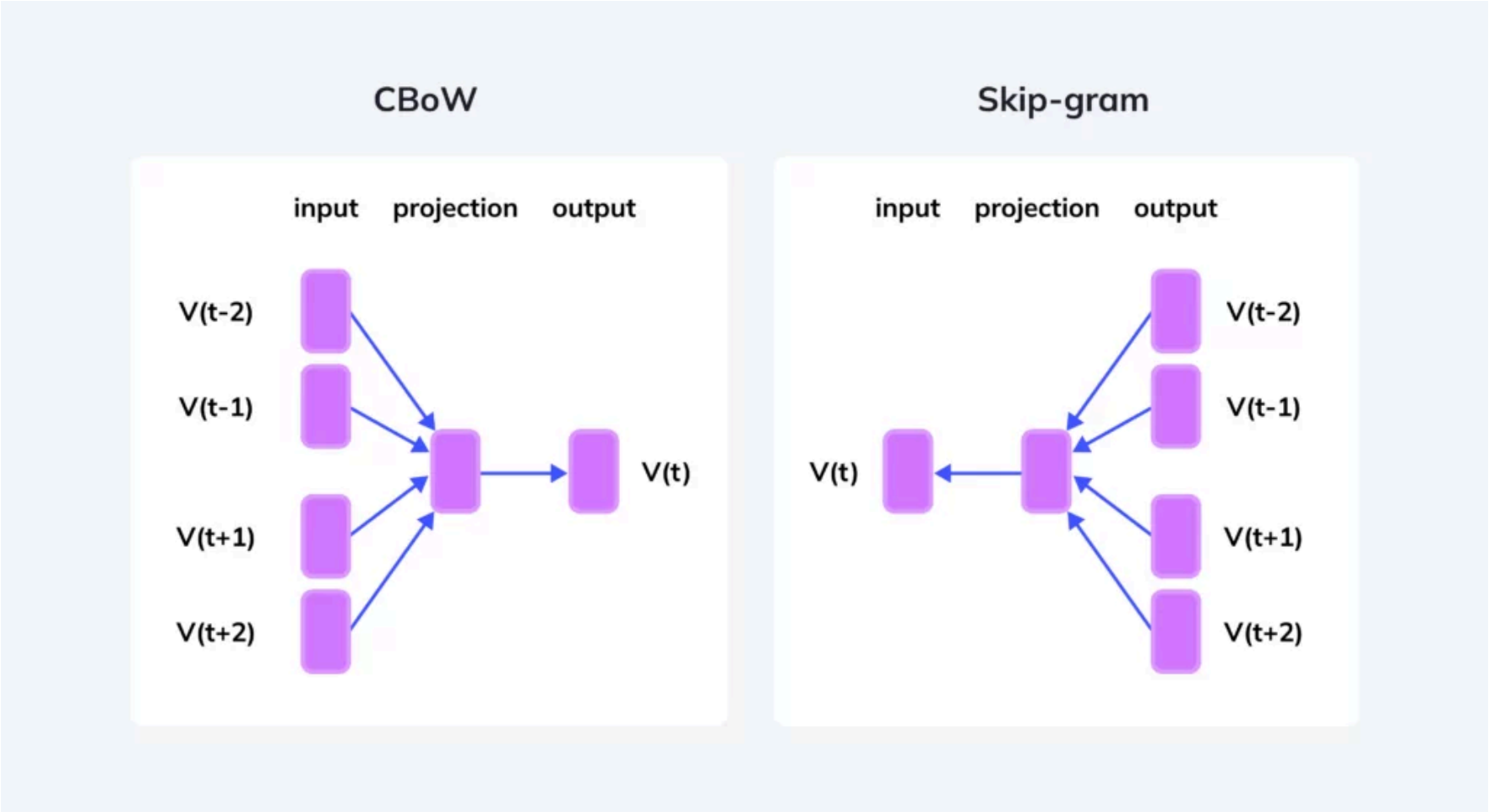
- **Word2Vec** is a model that learns to represent words as vectors in a way that captures word meanings based on their context in large text data.
- It has two main approaches:
- **Continuous Bag of Words (CBOW)**: Predicts a target word from the surrounding context words.
    - **Concept**: The CBOW model predicts a target word (the center word) based on its surrounding context words.
- **Example**:
    - Sentence: The cat sits on the mat.
    - Target word (center): sits
    - Context words: The, cat, on, the, mat

  In CBOW, we use the context words ("The", "cat", "on", "the", "mat") to predict the target word "sits."
- **Skip-Gram**: Predicts surrounding context words from a target word.
    - **Concept**: The Skip-gram model does the opposite of CBOW—it predicts the context words from the target word.
    - **Example**:
    - Sentence: The cat sits on the mat.
    - Target word: sits
    - Context words: The, cat, on, the, mat

  In Skip-gram, we use the target word "sits" to predict the context words ("The", "cat", "on", "the", "mat").

Conclusion:

- **CBOW**: Predict the center word using the context.
- **Skip-gram**: Predict the context words using the center word.



2. **GloVe (Global Vectors for Word Representation)**:

- **GloVe** is another popular word embedding technique.

It works by analyzing the global co-occurrence of words in a corpus (the whole collection of text) and using this information to create word vectors.

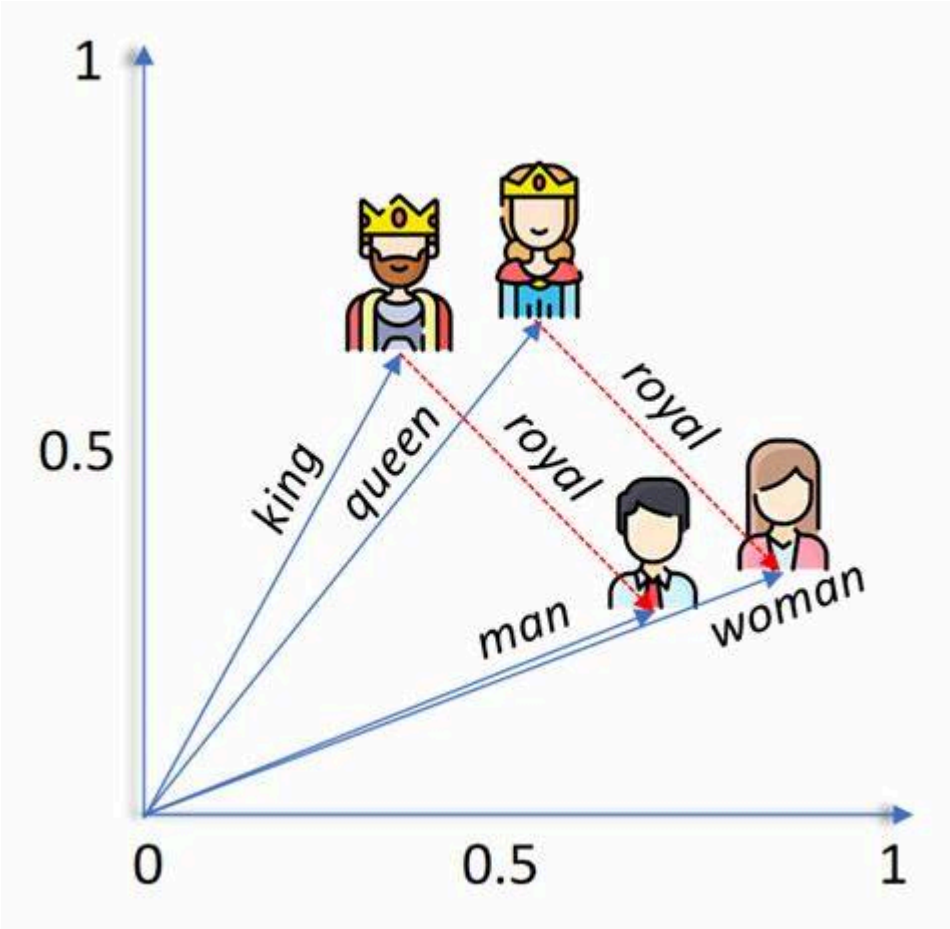
- Unlike Word2Vec, which focuses on the local context (neighboring words), GloVe focuses on the global statistical information across the entire corpus.

- Example: GloVe might place "cat" and "dog" closer to each other in the vector space because they appear in similar contexts (like "pet", "animal").

Example to Understand Word Embeddings:

Imagine you have three words: **"king"**, **"queen"**, and **"man"**.

- Word embeddings help in capturing relationships like:
  - **king** is to **queen** as **man** is to **woman**.



In the vector space, the difference between "king" and "queen" might be similar to the difference between "man" and "woman". This allows for mathematical operations like:

king - man + woman = queen

This shows that embeddings can capture relationships between words and perform operations on them (like analogies) that were not possible with older techniques like one-hot encoding.

Conclusion:

- Word embeddings are a way to convert words into numerical vectors.
- They capture the meaning of words in a way that allows similar words to have similar vectors.
- Popular methods like Word2Vec and GloVe learn these embeddings from text data, allowing for better word relationships, analogies, and understanding.

3. Transformers

- Geeks
- Hugging Face

Transformers have fundamentally changed the way machines understand and generate human language, making them the go-to architecture for state-of-the-art NLP applications today.

Transformers are a type of deep learning models that have revolutionized Natural Language Processing (NLP).

They are designed to handle sequential data (like text) but in a much more efficient and powerful way compared to previous models, such as RNNs and LSTMs.

What Makes Transformers Special?

1. **Attention Mechanism**:

- The key innovation behind transformers is the **attention mechanism**. In simple terms, attention **allows the model to "focus"** on different parts of the input sequence when making predictions, rather than just processing the data in order (like RNNs).
- This means that when processing a word in a sentence, the **model can consider the entire sentence** and give more importance to relevant words, regardless of their position. This is especially helpful for tasks like translation or text generation.

Example: In the sentence "The cat sat on the mat," when predicting the word "sat," the model might pay more attention to "cat" than to "the" or "on."

2. **Parallel Processing**:

- Unlike RNNs, which process data sequentially (one word at a time), transformers can process all words in the sentence simultaneously. This leads to **faster training** and better results on long sequences of text.

3. **Positional Encoding**:

- Since transformers process all words at once, they need a way to understand the order of words in the sentence. This is where **positional encoding** comes in. It adds information about the position of each word in the sequence, allowing the model to maintain an understanding of word order.

Transformers in Action:

Transformers have led to the development of powerful models like:

- **BERT (Bidirectional Encoder Representations from Transformers)**:

A transformer-based model trained to understand the context of words in both directions (left-to-right and right-to-left). For understanding the context of a word in relation to its surrounding words, making it powerful for tasks like question answering, sentiment analysis, and language understanding.

- **GPT (Generative Pre-trained Transformer)**:

A model trained to generate text in a conversational style, often used for text completion, summarization, and even creative tasks like writing poetry.

Real-World Example:

Imagine you’re using a machine translation system (like Google Translate).

A transformer model can take the sentence:

and translate it to French as

"I love reading books"

"J'aime lire des livres."

The attention mechanism helps the model focus on key words like "love" and "reading" when making the translation, ensuring the output makes sense.

- **BERT (Bidirectional Encoder Representations from Transformers)**: A model that understands the context of a word by looking at the words before and after it, used for tasks like text classification and question answering.
- **GPT (Generative Pre-trained Transformer)**: A language generation model that predicts the next word in a sequence, excelling at tasks like text generation and conversational agents.
- **T5 (Text-to-Text Transfer Transformer)**: Treats all NLP tasks as text-to-text problems, such as translation, summarization, and question answering.
- **RoBERTa (Robustly Optimized BERT Approach)**: An optimized version of BERT trained with more data and without next-sentence prediction, improving its performance across NLP tasks.
- **XLNet**: A model that combines autoregressive and bidirectional training to capture richer context dependencies, outperforming BERT in many tasks.
- **ALBERT (A Lite BERT)**: A memory-efficient version of BERT with shared parameters across layers, reducing the model size without sacrificing performance.
- **BART (Bidirectional and Auto-Regressive Transformers)**: A hybrid model combining BERT and GPT, used for sequence-to-sequence tasks like text generation and summarization.
- **DeBERTa (Decoding-enhanced BERT with Disentangled Attention)**: An advanced version of BERT with improved attention mechanisms, enhancing performance on text classification and understanding tasks.

How Do Transformers Work?

Transformers are made up of two main components:

1. **Encoder**: The encoder takes in the input sequence (like a sentence) and processes it into a sequence of vectors. Each word in the sentence is converted into a vector using an embedding.
2. **Decoder**: The decoder takes the output from the encoder and generates the final prediction (such as the next word in a sentence or a translation).

The encoder and decoder are made up of multiple layers of attention and other operations that help the model learn better representations of the text.

The Transformer Architecture:

- **Self-Attention**: This allows the model to determine which words in the input sequence are important for predicting the next word.
- **Multi-Head Attention**: Instead of focusing on just one aspect of the sentence, the model uses multiple attention heads to learn different relationships at once.
- **Feedforward Neural Networks**: After attention layers, transformers use simple neural networks to process the data further.
- **Layer Normalization**: This technique helps stabilize and speed up training.

Conclusion

- **Transformers** use an attention mechanism to understand the relationships between words in a sentence.
- They process all words in a sequence at once, making them faster and more powerful than older models like RNNs.
- They are the foundation for modern models like **BERT**, **GPT**, and **T5**, which perform well on a wide range of NLP tasks.

Hugging Face library

The **Hugging Face** library is one of the most popular and powerful tools for working with state-of-the-art models in Natural Language Processing (NLP).

It provides easy access to a wide variety of **pre-trained models** and **datasets**, which can be used for tasks such as text classification, translation, summarization, and question answering.

What is Hugging Face?

Hugging Face is an open-source company that focuses on advancing NLP by providing tools that simplify working with complex models like transformers. The **Transformers** library by Hugging Face allows developers to download pre-trained models, fine-tune them, and use them for specific NLP tasks without needing to train the model from scratch.

Why Use Hugging Face?

1. **Pre-trained Models**: Hugging Face provides a large number of pre-trained models for various NLP tasks. These models have been trained on massive datasets and can be used immediately or fine-tuned for specific tasks.
2. **Ease of Use**: With just a few lines of code, you can load, use, and fine-tune models on your own data. It makes working with advanced NLP models accessible to beginners and experts alike.
3. **Integration with Other Libraries**: Hugging Face integrates seamlessly with libraries like PyTorch and TensorFlow, which allows users to leverage the full power of these frameworks for training and deploying models.
4. **Community and Resources**: Hugging Face has a large community that shares models, datasets, and tutorials. It is a great place to learn and collaborate on NLP projects.

Generative Models and Language Models

Generative Models

Generative models are machine learning models that generate new data based on the patterns they learn from existing data. In NLP, these models create new text or predict possible sequences of text.

Key Concept: A **generative model** learns the underlying distribution of data and generates new content that resembles the data it was trained on. It doesn't just classify or predict based on input; it creates new, realistic outputs.

Examples of Generative Models in NLP:

1. **GPT (Generative Pretrained Transformer)**:
    - GPT is a transformer-based generative model. It can generate coherent and contextually relevant text, answer questions, or write essays, all based on a prompt.
    - Trained on massive text corpora from the internet, GPT models like GPT-3 are capable of creating highly human-like text.
2. **BERT (Bidirectional Encoder Representations from Transformers)**:
    - BERT is a transformer model designed to understand context by looking at both left and right of a word. It's often used in pretraining models to understand the relationship between words in a sentence.
    - Although BERT is primarily used for tasks like classification, it is a key part of the foundation for generative models, especially in language understanding.

Language Models

Language models are a type of generative model, specifically designed to predict the likelihood of a sequence of words or generate coherent sentences.

Key Concept: A **language model** is a model trained to understand and predict the probability of sequences of words in a language. It can be used for tasks like text generation, text completion, and more.

Examples of Language Models:

1. **RNN (Recurrent Neural Networks)**:
    - RNNs are one of the earliest types of models used for sequential data, including text. They are capable of maintaining a memory of previous inputs, making them suitable for language modeling tasks.
2. **LSTM (Long Short-Term Memory)**:
    - LSTM networks are an advanced version of RNNs, specifically designed to address the issue of vanishing gradients and maintain long-term dependencies in sequences, making them better suited for language modeling than standard RNNs.
3. **Transformer Models**:

- Transformers, like GPT and BERT, have revolutionized language modeling by using self-attention mechanisms. They can consider the entire context of a sentence or document in parallel, which greatly improves performance in a variety of NLP tasks.

Transfer Learning

Transfer learning is a technique where a model trained on one task is reused on a new but related task. Instead of training a model from scratch, you take advantage of the knowledge learned from a different task, and adapt it for your own needs.

This concept has been widely used in natural language processing (NLP), especially with large pre-trained language models.

Key Concepts:

1. **Pre-trained Models**:
   - A model is first trained on a large, general corpus of text (such as Wikipedia or a news dataset) to learn the basic structure of language, like grammar, sentence structure, and common word relationships.
   - These models are then fine-tuned for specific NLP tasks, such as sentiment analysis, named entity recognition, or text classification.
2. **Fine-tuning**:
   - Once a model has been pre-trained, it can be adapted to a specific task with additional training on a smaller, task-specific dataset. This is called **fine-tuning**.
   - Fine-tuning adjusts the model's parameters to make it more effective for the new task, often requiring less data and time compared to training a model from scratch.

Examples of Transfer Learning in NLP:

1. **BERT** (Bidirectional Encoder Representations from Transformers):
   - BERT is pre-trained on large corpora like Wikipedia and BookCorpus. Once pre-trained, it can be fine-tuned for tasks like text classification, question answering, and language inference.
   - This allows BERT to leverage its general understanding of language and apply it to specific problems.
2. **GPT** (Generative Pre-trained Transformer):

- Similar to BERT, GPT models are pre-trained on vast amounts of text. They can then be fine-tuned to perform tasks like text generation, summarization, or machine translation.
- For example, GPT can be fine-tuned to generate text based on a given prompt or to answer specific questions.

Why Transfer Learning is Important in NLP:

- **Reduced Training Time**: Instead of starting from scratch, you build upon the knowledge gained from pre-training, making the learning process faster.
- **Improved Performance with Less Data**: Fine-tuning a pre-trained model requires fewer data points than training from scratch, especially for tasks that are closely related.
- **Generalization**: Pre-trained models learn patterns from diverse data, allowing them to generalize better to new, unseen tasks.

Conclusion Transfer learning enables us to use models that have learned general language patterns and apply them to specific NLP tasks. By fine-tuning pre-trained models, you can achieve high performance with limited task-specific data and significantly reduce computational costs.

3. Building the Sentiment Analyzer (OPTIONAL/ADVANCED)

Fine-tuning a Pre-trained Sentiment Analysis Model

Now, let's walk through how to fine-tune a pre-trained sentiment analysis model using Hugging Face's **Transformers** library. We'll use a model like **BERT** (Bidirectional Encoder Representations from Transformers) for sentiment analysis, which is pre-trained on a large corpus of text and can be fine-tuned on a smaller dataset for specific tasks.

Steps to Fine-tune a Pre-trained Model:

1. **Install the Required Libraries:**

   If you haven't already, you need to install Hugging Face's `transformers` and `datasets` libraries. You can do this by running:

Might need rust for this: https://rustup.rs/

In [ ]:
```
pip install transformers
```

In [ ]:
```
from transformers import BertTokenizer, BertForSequenceClassification
from datasets import load_dataset

# Load a pre-trained BERT model and tokenizer
model_name = "bert-base-uncased"
model = BertForSequenceClassification.from_pretrained(model_name, num_labels=2)  # 2 labels for binary sentiment (positive, ne
tokenizer = BertTokenizer.from_pretrained(model_name)
```

2. **Load and Preprocess the Dataset:**

For sentiment analysis, we can use datasets such as IMDb or SST-2. Hugging Face's datasets library makes it easy to load these.

In [ ]:
```
# Load the IMDb dataset
dataset = load_dataset("imdb")

# Tokenize the dataset
def preprocess_function(examples):
    return tokenizer(examples['text'], padding=True, truncation=True)

# Apply the tokenizer to the dataset
tokenized_datasets = dataset.map(preprocess_function, batched=True)
```

3. **Set Up Training Arguments:**

Fine-tuning a model requires specifying the training configuration, such as batch size, number of epochs, and learning rate. Hugging Face's Trainer API makes this process much simpler.

In [ ]:
```
from transformers import TrainingArguments, Trainer

# Set up the training arguments
training_args = TrainingArguments(
    output_dir="./results",          # output directory
    num_train_epochs=3,              # number of training epochs
    per_device_train_batch_size=8,   # batch size for training
```

```
    per_device_eval_batch_size=8,    # batch size for evaluation
    warmup_steps=500,                # number of warmup steps for learning rate scheduler
    weight_decay=0.01,               # strength of weight decay
    logging_dir='./logs',            # directory for storing logs
    logging_steps=10,
)
```

4. **Train the Model:**

Once everything is set up, you can start fine-tuning the model using the Trainer API.

In [ ]:
```
# Initialize the Trainer
trainer = Trainer(
    model=model,                         # the model to be trained
    args=training_args,                  # training arguments
    train_dataset=tokenized_datasets['train'],   # training dataset
    eval_dataset=tokenized_datasets['test'],    # evaluation dataset
)

# Start fine-tuning the model
trainer.train()
```

5. **Evaluate the Model:**

After fine-tuning, you can evaluate the model's performance on the test set.

In [ ]:
```
# Evaluate the model
results = trainer.evaluate()
print(results)
```

6. **Make Predictions:**

Once the model is fine-tuned, you can use it to make predictions on new sentences.

In [ ]:
```
# Make a prediction
text = "I love this movie!"
```

```
inputs = tokenizer(text, return_tensors="pt")
outputs = model(**inputs)
predictions = outputs.logits.argmax(dim=-1)
print("Prediction:", "Positive" if predictions == 1 else "Negative")
```

Live Exercise

Now it's your turn!

Task 1: description of task

- instructions

END

THANK YOU!

Live Exercise Solutions

In [ ]:
```
solutions
```

Programming Interveiw Questions

1. topic:
    - question

In [ ]:

Mohammad Idrees Bhat

Tech Skills Trainer | AI/ML Consultant